Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Vytvoření výukového programu pro výuku programovacího jazyka Prolog

Plzeň 2014 Tomáš Cigler

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 27. června 2014

Tomáš Cigler

Abstract

Creation of learning program for Prolog learning

This bachelor thesis deals with creation of simple learning program for novice programmers or even advanced ones, who want to explore the world of constraint logic programming.

You will find here lots of practical examples, which should guide you to understand different way of programming. After reading this thesis, you should be able to write Prolog programs on your own.

Abstrakt

Vytvoření výukového programu pro výuku programovacího jazyka Prolog

Tato práce je zaměřena na začínající i pokročilé programátory, kteří chtějí získat povědomí o logickém přístupu k programování. Najdeme zde spoustu praktických ukázek, které se jednoduchým způsobem snaží ukázat čtenářům možnosti použití tohoto odlišného přístupu.

Programátor by měl být po přečtení práce schopen sám vytvářet programy v jazyce Prolog a samostatně řešit obvyklé problémy související s logickým programováním.

Obsah

1 Úvod

Práce je cílena na skupinu začínajících i pokročilých programátorů, kteří chtějí získat zkušenosti s přístupem k programování logických úloh. Jedná se o základy logického programování v jazyce Prolog, s rozšířením o nadstandardní možnosti implementace *B-Prolog*.

Prostudováním jednotlivých kapitol a příkladů s nimi úzce spjatých se čtenář postupně dozví, jak s programováním začít, co k tomu vše potřebuje a následuje představení celého procesu programování. Při čtení by čtenář neměl mít pocit informačního přesycení, přesto je zachována informační hodnota a její vzrůstající trend.

První kapitoly jsou určeny úplným začátečníkům a informace jsou podány jednoduše a přirozeně. Metoda výuky umožňuje zvolit si k práci i jiný interpretátor, než byl k psaní a spouštění programů výhradně používán. Velká část práce je realizována za pomoci standardních zápisů Prologu s následným obohacením o další funkce B-Prologu.

Závěrem se čtenář dozví o pozitivech a negativech propojení logického a imperativního programování v jeden funkční celek a možnosti použití ve velkých projektech. Poslední kapitoly kladou vyšší nároky na znalosti klasického programování v kombinaci se znalostmi získanými v předchozích kapitolách této práce.

2 Seznámení s jazykem Prolog

2.1 Co je Prolog?

Programovací jazyk *Prolog* vznikl na začátku 70. let ve Francii. P. Roussel jej začal vyvíjet za účelem umožnění komunikace člověka s počítačem v přirozeném jazyce. Prolog pracuje na bázi predikátové logiky prvního řádu. Jednoduše řečeno, je to odvozování poznatků na základě známých faktů a pravidel neboli vyplývání.

Moderní vysokoúrovňové jazyky (např. Java) jsou imperativní. Programy popisují algoritmus, kterým se řeší specifický problém, a využívá se k tomu většinou procedurální programování a objekty. Prolog je zkratka anglické věty „**Pro**gramming in **log**ic“. Ta značí, že se jedná o logické programování. Definujeme pouze co se má vyhodnotit a neřešíme způsob, čímž se Prolog řadí do deklarativních programovacích jazyků. Vnitřní algoritmy vyhodnocování dotazů (cílů) jsou pro nás skryté. Tím je jazyk přívětivější i pro laiky, nebot' není třeba znát architekturu a princip fungování počítače [8].

Pro snadnou orientaci v kódu je důležité znát základy predikátové logiky. Programátor vytváří posloupnost určitých výroků (predikátů), které dohromady v daném sledu a kontextu vytvářejí databázi pro reprezentaci znalostí. Jazyk je interpretovaný, čili nemáme předem určený postup řešení, ale zadáváme dotazy, které chceme vyhodnotit. K tomu interpretátor využívá rekurzi, unifikaci a backtracking (podrobněji viz kapitola 4).

2.2 Motivace

Už na úvod můžeme namítat, že jazyk Prolog dnes není ve světě rozšířený a bude mít malé využití. Rozhodně jej nevyužijeme pro složité matematické výpočty, vykreslování grafiky a podobné výpočetně náročné akce, kde potřebujeme vytvářet výkonné algoritmy.

Prolog má ale vlastnosti, díky kterým se výborně hodí na specifické úkony. Například již zmiňovaná komunikace se strojem v přirozeném jazyce byla využita v roce 2005 agenturou NASA ve vesmírné stanici. V systémech umělé inteligence našel Prolog využití v nedávné době i při programování superpočítače Watson od IBM. Stejně důležité jsou znalostní báze, „chytré“ vyhledávání apod. [1]

Prolog přináší jiný pohled na programování a přechod z klasického přístupu může být pro některé programátory výzva. V Prologu se dají napsat rychle jednoduché a efektivní aplikace, které jsou kódově transparentní a tedy snadno udržitelné, což je většinou naším cílem. V kombinaci s imperativním jazykem, jako je Java nebo C, má Prolog ještě mnohem větší potenciál. Nejprve si řekneme, jak s programováním v Prologu vlastně začít.

2.3 Vývojové prostředí

U každého vysokoúrovňového programovacího jazyka potřebujeme software, který nám umožní spuštění programu, aniž bychom museli znát nižší programovací jazyky. U kompilovaných jazyků píšeme kód v námi zvoleném textovém editoru a následně použijeme překladač, který přeloží náš program do mezikódu nebo strojového kódu. Jazyk Prolog je ale interpretovaný, proto potřebujeme vhodný interpretátor.

První intepretátor naprogramoval již zmíněný P. Roussel společně s A. Colmerauerem v roce 1972. Postupně se rozšiřovaly implementace a každá v sobě obsahovala trochu jinou sadu zabudovaných predikátů. Kvůli přenositelnosti Prologovských programů bylo nutné zavést pevný standard. Návrh standardu sepsal v roce 1984 R. O'Keefe, přesto ho málokterá verze Prologu dodržovala. V průběhu let se standard upřesňoval a v roce 1993 vyšla další neoficiální sumarizace návrhu. V roce 1995 byl standard oficiálně zaveden, ale až v posledních několika letech je implementacemi více dodržován [6].

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

2.3.1 Přehled interpretátorů

K dispozici je celá řada starších implementací, které se neustále vyvíjejí. Většina z nich je vytvořena na otevřených licencích. Proto není nutné sahat pro zpoplatněné verze, jako je např. SICStus Prolog, který je velice drahý. Na internetových diskuzích a v článcích najdeme uživatelská doporučení a hodnocení různých implementací. Přehledné srovnání sepsal C. Heng na svých

internetových stránkách [5]. Bohužel přehled již není aktuální, proto se podíváme na každou implementaci zvlášť.

Nejzajímavější jsou pro nás interpretátory s možností propojení s jinými jazyky. To znamená, že interpretátor dokáže přeložit a provést kódovou sekvenci těchto jazyků. Ekvivalentně můžeme v procedurálním programu spolupracovat s interpretátorem, který zde může vyřešit dílčí problém efektivněji, než původní programovací jazyk.

- **tuProlog** – intepretátor je vytvořen pod licencí GNU LGPL. Skládá se z minimalistického jádra, které zvládá základní práci s Prologem a lze dále rozšířit dalšími sadami predikátů. Původně byl vytvořen pro webové applety. Výsledné programy kompiluje do .JAR formátu, takže jsou spustitelné napříč platformami s JVM[1]. Dále existuje verze pro Android a .NET. S těmito jazyky je potom kompatibilní.

- **Visual Prolog** – podporuje pouze platformu Windows. Kombinuje logické, klasické i objektové programování. Je kompatibilní s jazyky C a C++ a může používat funkce Win32. Prostředí podporuje i grafické vytváření a úpravy formulářů a oken. Pro soukromé účely je Visual Prolog zdarma, komerčně se může využívat až se zaplacenou licencí.

- **Ciao Prolog** – v základu plně podporuje ISO standard, ale mohou se přidat pro každý projekt rozšiřující moduly. Poslední verze je ze srpna roku 2011 a funguje na všech nejpoužívanějších platformách. Ciao Prolog se může propojit s jazyky C i Java anebo s relačními databázemi. Existují i moduly pro síťovou komunikaci a webové aplikace.

- **C#Prolog** – stále aktualizovaná verze pod licencí GNU GPLv2. Podporuje jednoduché propojení s jazykem C# a tedy i databázemi, se kterými tento jazyk dokáže pracovat. Má podporu pro formát XML a stále se rozšiřující struktury JSON. Funguje na platformě Windows i Linux.

- **GNU Prolog** – kromě standardu ISO obsahuje více než 300 dalších zabudovaných predikátů. Podporuje ladění a propojení s jazykem C. Obsahuje optimalizovaný a rychlý kompilátor, který vytváří jednoduché spouštěcí soubory. Pracuje na všech základních platformách a je vytvořen pod licencí GNU LGPL.

[1] Java Virtual Machine – virtuální prostředí, které umožňuje spustit programy kompilované do binárního mezikódu jazyka Java

- **SWI Prolog** – největší projekt na volné GNU GPL licenci, který se vyvíjí již od roku 1987. Standard ISO Prolog je rozšířen o robustní grafické rozhraní s možností ladění programů a přidání rozšiřujících balíčků. Můžeme jej propojit s jazykem C a C++, databázemi, případně nahrát hotovou základní knihovnu pro rozpoznávání přirozeného jazyka apod. Samozřejmostí je podpora všech nejpoužívanějších platforem.
- **B-Prolog** – implementace od spol. Afany Software, která také splňuje standard ISO Prolog, dokáže obousměrně spolupracovat s jazyky C, C++ a Java a pro osobní, nekomerční nebo akademické účely je zdarma. Podrobněji si ji představíme v kapitole 6.

2.3.2 Příprava prostředí

Když už máme povědomí o tom, co to Prolog vlastně je a k čemu jej můžeme využít, vybereme si vhodnou implementaci interpretátoru. Pro další práci a ukázky budu po celou dobu využívat právě B-Prolog na systému Linux.

Instalace interpretátoru je jednoduchá. Stačí z oficiálních webových stránek `http://www.probp.com/download.html` stáhnout nejnovější verzi pro naši platformu, rozbalit do požadovaného umístění pomocí archivačního programu a spustit `bp.exe` na platformě Windows, eventuálně na Linuxu soubor `bp` v příkazové řádce.

V obou případech před sebou vidíme konzolovou aplikaci, která vypíše na obrazovku (v mém případě):

```
B-Prolog Version 8.1, All rights reserved, (C) Afany Software
    1994-2014.
| ?-
```

Pokud se Vám B-Prolog nezamlouvá, pro základy jazyka není důležité, jakou implementaci si vyberete. Následující postupy by měly ve všech implementacích podporujících standard ISO Prolog fungovat totožně.

3 Začínáme programovat

Jelikož B-Prolog plně podporuje české znaky v sadě `UTF-8`, mohli bychom používat česká slova včetně diakritiky. Z důvodu přenositelnosti by ale bylo nejlepší psát kód v angličtině. Knihy zabývající se výukou jazyka Prolog se vyskytují převážně v anglickém jazyce, proto zde budu pro změnu v ukázkových příkladech používat češtinu, ale bez diakritiky.

Prostředí nás nyní vyzývá k zápisu příkazů (anglicky *prompt*), je to vlastně interaktivní mód interpretátoru. Většina knih, návodů na internetu i odborných článků týkajících se výuky programovacího jazyka začíná stejně – vypsat na standardní výstup (tedy většinou na obrazovku) text „Hello world!“ neboli v překladu „Ahoj, světe!“ Podíváme se, jak se to dělá v Prologu:

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

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

Slovo *yes* na konci znamená, že byl cíl vyhodnocen úspěšně. Pokud by se vyhodnocení nepovedlo, interpretátor vypíše *no*. Ať už interpretátor vyhodnotí predikát úspěšně či neúspěšně, následuje opět interaktivní mód, takže je na obrazovku vypsán prompt a čeká se na vstup od uživatele. Výjimkou je zabudovaný predikát *halt*, který ukončí práci interpretátoru. O dalších zabudovaných predikátech si povíme v kapitole 5.

3.1 Základní syntax a terminologie

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

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

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

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

Atom je jinými slovy určitý bezkontextový „nápis“, neboli objekt. Zapisujeme jej třemi způsoby.

1) Musí začínat malým písmenem a jako další znaky jsou kromě malých a velkých písmen povoleny pouze číslice a podtržítka.

2) Libovolný řetězec uzavřený v jednoduchých uvozovkách, kde můžeme používat i speciální znaky (např. `'Ahoj, svete!'`).

3) Jeden (nebo posloupnost) z povolených speciálních znaků (`+ - * /` apod.). Dále si některé z nich ukážeme.

Číslo může být celé (integer) nebo s plovoucí desetinnou čárkou a není omezené velikostí. Integer můžeme zapisovat v desítkové (`567`), osmičkové (`0o123`), hexadecimální (`0xC2A`) i binární (`0b011`) soustavě. Desetinné číslo zapisujeme pouze v desítkové soustavě a používáme desetinnou tečku. Zápisy mohou vypadat následovně:

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

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

Proměnná se vždy zapisuje s počátečním velkým písmenem nebo podtržítkem a mohou následovat další znaky. Můžeme použít i číslice. Pro jednoduché programy používáme běžně jednopísmenné názvy, u složitějších bychom měli pojmenovávat proměnné dle kontextu, abychom se v programu vyznali.

Pokud je proměnná zapsána pouze podtržítkem, říkáme, že je anonymní. Interpretátor ji přiřadí hodnotu, ale po vyhodnocení ji nezmiňuje, ani s ní nemůžeme dále pracovat. Proměnné se dosazují do struktur, které chceme určit abstraktně. Používají se místo konkrétních objektů.

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

Hodí se převážně pro případy, kdy potřebujeme pro nějaký funktor proměnlivý počet argumentů. Nepotřebujeme předem vědět, kolik jich bude a se-

znam se vždy přizpůsobí. Může obsahovat atomy, struktury nebo další seznamy. Seznamy se mohou vnořovat neomezeně:

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

3.2 Vlastní program a ukázkové příklady

Když už známe základní syntax a stavební prvky Prologu, vytvoříme si vlastní program, kterým definujeme *relační databázi*. K tomu potřebujeme libovolný textový editor. Vytvoříme si prázdný textový soubor, který si vhodně pojmenujeme.

Jako přípona souboru se dle konvencí používá *.pl*. Ze zkušeností uživatelů interpretátoru SWI Prolog se ale v některých případech může přiřazení této koncovky krýt se zdrojovými soubory programů jazyka Perl. V těchto případech tvůrci SWI Prologu v dokumentaci doporučují používat příponu *.pro* [10].

3.2.1 Zápis kódu

Pro ukázku si navrhneme malou databázi domácích mazlíčků do souboru `Mazlicci.pro`:

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

Programový segment 3.1: Mazlíčci

Podívejme se blíže na význam tohoto kódu – definovali jsme si celkem 8 faktů pomocí třech unárních predikátů *pes*, *kocka* a *kralik*. Predikáty nám určují relace mezi termy. Jako je zapsán fakt v Prologu `pes(rony)`, přirozeně můžeme říct „Rony je pes“.

U námi vytvořených programů velice záleží na pořadí predikátů. Prolog při vyhodnocování postupuje v databázi shora dolů. V programu 3.1 rozdíl nepoznáme, ale když začneme definovat složitější pravidla, na problém s pořadím zápisu můžeme narazit.

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

Za každým termem musí být napsána tečka. Ta slouží k odlišení začátku zápisu dalšího termu stejně jako středníky v jazyce Java. Proto můžeme v programech zapsat více termů na jednu řádku. Pokud budeme chtít, tak můžeme na jednu řádku zapsat celý program. U rozsáhlých webových projektů se díky této vlastnosti mohou využít tzv. *minimizéry*, které automaticky odstraní všechna odřádkování a komentáře, takže se výsledný soubor o něco zmenší a rapidně se zhorší čitelnost pro případné plagiátory.

3.2.2 Čistý kód

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

K přehlednosti patří i správné a logické pojmenování predikátů. Programátor si jistě může ušetřit práci tím, že bude definovat predikáty se zkratkovitými názvy. Komentáře také stojí nějaký čas. Podívejme se, jak může vypadat nedbale zapsaná stejná databáze mazlíčků. Představíme si náš myšlenkový pochod:

```
p(alik). k(micka). k(mnauka). p(rony). p(besi). k(andy).
```

Takový zápis se může objevit právě v případech, kdy chceme Prologovský kód spojit s jiným jazykem, protože vnitřní zápisy jsou pro koncového uživatele neviditelné.

Programátor definuje postupně jednotlivá zvířata, jak mu přijdou pod ruku. Řekněme, že *p/1* představuje psa a *k/1* kočku. Žádná jiná zatím nepotřebuje a bude s nimi dále pracovat v Javě. Následně bude chtít do databáze přidávat ještě králíky. Predikát *k/1* ale již zabírají kočky. Použije *kr/1* a přidá další fakta. Takto může vypadat výsledný kód:

```
p(alik). k(micka). k(mnauka). p(rony). p(besi). k(andy).
kr(ferda). kr(matilda).
```

Porovnejme nyní zápis s programem 3.1. Ve výsledku jistě budou oba programy fungovat. Poznáte ale programátorův záměr? Nejspíš ano, ale bude to

trvat déle než u čistě napsaného kódu. Jistě si tvůrce takového programu ušetřil pár minut času, který by mu zabralo formátování, delší názvy nebo komentáře, ale sám bude za měsíc chvíli přemýšlet, co vlastně znamená predikát *kr/1*, jestli to třeba není křeček.

Žádný čas nedbalým přístupem neušetříme. Kdyby měli v budoucnu na našem projektu pracovat i externí programátoři, takový kód je nepřijatelný. V této práci se budu držet obdobného zápisu jako v původní ukázce.

3.2.3 Načtení programu a základní dotazy

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

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

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

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

```
| ?- listing.
kralik(ferda).
kralik(matilda).
pes(alik).
pes(rony).
pes(besi).
kocka(micka).
kocka(mnauka).
kocka(andy).
yes
```

Jak je vidět, interpretátor seřadil fakta tak, aby bylo vyhledávání optimalizované. Pokud jednotlivé termy v souboru `Mazlicci.pro` zpřeházíme,

intepretátor vypíše na obrazovku informaci

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

a obdobně i pro ostatní predikáty. Přesto jsou ve výsledku vnitřně seřazeny stejně, jako u předchozího programu. Přehazují se mezi sebou ale pouze celé bloky spolu souvisejících predikátů. Stejná fakta nebo pravidla mezi sebou kvůli závislosti na pořadí interpretátor prohodit samozřejmě nesmí.

Na takové databázi si vyzkoušíme zadávání cílů:

```
| ?- kralik(ferda).
yes
| ?- pes(rony).
yes
| ?- pes(micka).
no
| ?- krecek(tony).
*** Undefined procedure: krecek/1
```

Jestliže se ptáme, zda Ferda je králík a Rony je pes, intepretátor odpoví kladně, jelikož je to přímo jeden ze známých faktů. Micka je ale kočka, proto odpověď na dotaz, zda je Micka pes, je negativní. Stejně tak by byla negativní, když jako argument jednoho z predikátů použijeme v dotazu atom, který není mezi fakty uveden. Při dotazu na křečka interpretátor vypíše výjimku, protože jsme v programu predikát *krecek/1* nedefinovali.

3.2.4 Proměnné a pravidla v databázi

Na posledním příkladu této části si vysvětlíme zápis proměnných a pravidel. Pro tyto účely pouze rozšíříme naši databázi mazlíčků. Definujeme nový predikát *zvire/1*. Můžeme tvrdit, že všechna jména v naší databázi představují jména zvířat. Abychom nemuseli definovat pro každý atom nový fakt (například že Alík je zvíře, Micka je zvíře atp.), použijeme proměnnou a zapíšeme následující pravidlo:

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

Posloupnost znaků `:-` má význam jako slovo „když“. Středník v tomto zápisu znamená *disjunkci* neboli „nebo“. Čteme „*X* je zvíře, KDYŽ *X* je pes NEBO kočka NEBO králík“. Kromě disjunkce můžeme použít ještě logickou konjunkci „a zároveň“. K tomu slouží operátor ,/2. Definujeme si pro ukázku několik nových pravidel. Přidáme do databáze ještě křečka a řekneme, že psi, kočky a křečci žerou maso a králící i křečci žerou býlí. Tím dokážeme odlišit skupiny masožravců, býložravců a všežravců:

```
% Krecci
krecek(tony).

% Strava
zere_maso(X) :- pes(X); kocka(X); krecek(X).
zere_byli(X) :- kralik(X); krecek(X).

% Potravinove strategie
masozravec(X) :- zere_maso(X), not zere_byli(X).
bylozravec(X) :- zere_byli(X), not zere_maso(X).
vsezravec(X) :- zere_byli(X), zere_maso(X).
```

V pravidlech se objevuje nový predikát *not/1*, který má význam negace predikátu. Pokud tedy interpretátor vyhodnotí u masožravce predikát *zere_byli/1* jako pravdivý, not vrátí opak a tím je nepravdivé celé pravidlo. O vyhodnocování si blíže povíme v následující kapitole.

Znovu načteme změněný program a zkusíme si několika dotazy funkčnost nového řešení:

```
| ?- bylozravec(ferda).
yes
| ?- bylozravec(tony).
no
| ?- masozravec(tony).
no
| ?- vsezravec(tony).
yes
```

Programový segment 3.2: Potravinové strategie

Abychom využili celého potenciálu jazyka Prolog, je třeba pochopit základní principy, jak interpretátor pracuje.

4 Činnost interpretátoru a standardní predikáty

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

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

Tedy přidat křečka do výčtu zvířat. Potom je vše v pořádku. Musíme si ale dávat velký pozor na souvislosti. V úvodu jsme si řekli, že Prolog je založen na vyhodnocování cílů, rekurzi a backtrackingu (tzv. zpětném chodu).

4.1 Vyhodnocování cílů

Vrat'me se k prvnímu příkladu této práce. Abychom vypsali na obrazovku text „Ahoj, světe!“, využili jsme k tomu predikát *write/1*. Ten je navržen tak, aby jeho vyhodnocení vždy uspělo. Vypsání atomu uvnitř predikátu na obrazovku není při vyhodnocení cíle standardní chování, proto můžeme říci, že je to vedlejší efekt.

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

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

Interpretátor vyhodnocuje cíle vždy zleva doprava a prohledává databázi shora dolů. Proto záleží na pořadí. Nejprve se kontroluje hlava predikátu, následně arita a nakonec tělo. Na predikátu *pozdrav* si vyzkoušíme, jak funguje přetěžování. Argumentem určíme, koho v pozdravu oslovíme, pokud argument neuvedeme, oslovení bude obecně „lidi“:

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

Blíže se podíváme na vyhodnocení cíle *vypis_pozdrav/0*. Interpretátor se pokusí vyhodnotit `write('Prvni radka')`. Říkali jsme si, že je to zabudovaný predikát a je navržen tak, že jeho vyhodnocení vždy uspěje. Další predikát v řade je *nl/0*. To je také zabudovaný predikát, který na aktuální výstup vypíše novou řádku a také se ho vždy podaří vyhodnotit. Interpretátor pokračuje voláním predikátu *pozdrav/0*. Ten jsme si sami definovali, takže interpretátor vyhodnotí jeho tělo a zjevně uspěje. Vyhodnocování se dokončí výpisem textu na novou řádku.

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

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

Vyhodnocení zabudovaného predikátu *fail/0* je vždy neúspěšné, tedy uměle určíme, že predikát *pozdrav/0* bude také vždy neúspěšný. V okamžik, kdy některý z posloupnosti predikátů je neúspěšně vyhodnocen, přichází na řadu backtracking.

4.2 Jak funguje backtracking?

Český význam slova *backtracking* je „zpětný chod“. Pokud má interpretátor na výběr více možností vyhodnocení predikátu, vždy vybere první možnost. Pokud se nepodaří vyhodnotit tuto část posloupnosti, zkusí použít jiné řešení, když existuje. Jestliže další řešení neexistuje, vrátí se od tohoto místa o jeden krok zpět a postup opakuje – postupuje zprava doleva.

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

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

Podívejme se na vyhodnocení predikátu *vypis_pozdrav/0*:

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

Až do predikátu *pozdrav/1* (včetně), který se volá z *pozdrav/0*, je vše stejné jako při předchozím vyhodnocení. Změna je v následujícím predikátu *fail/0*, který neuspěje. Interpretátor se vrátí zpět na predikát *write/1*, který se při zpětném chodu nikdy nevyhodnotí kladně a znovu nic nevypíše, stejně jako *nl/0*. Dále interpretátor pokračuje na `pozdrav(lidi)` a zkusí najít jinou možnost řešení nebo jinou definici takového pravidla.

Žádná jiná možnost není, proto vystoupí zpět tam, odkud vyhodnocování začalo, tedy do těla struktury *vypis_pozdrav/0*. Interpretátor zkusí vyhledat další výskyty predikátu *pozdrav/0*. Ten nalezne a při vyhodnocování uspěje s výpisem „Pozdrav se nezdaril“ a celé vyhodnocení je úspěšné, proto je konečná odpověď `yes`.

Abychom mohli pokročit dále, vysvětlíme si nejdůležitější funkci interpretátoru – unifikaci.

4.3 Unifikace

Používá se při vyhodnocování cíle s proměnnými. Ukážeme si to konkrétně na programu `Mazlicci.pro`. Budeme chtít například vidět jména zvířat v databázi. Použijeme následující dotaz:

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

Programový segment 4.1: Použití proměnné

Všimněme si použití proměnné `Zvire`. Ta zastupuje mezi argumenty (nyní je pouze jeden) konkrétní atom. V našem případě je proměnná v dotazu nepřiřazená – nemá žádnou výchozí hodnotu. Zde zafunguje právě proces unifikace. Je to jinými slovy přiřazení do prázdné proměnné tak, aby byly dva termy shodné. Slouží při vyhledávání predikátu v databázi Prologu [2].

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

Následně se vyhodnocuje tělo pravidla zleva doprava. První je uveden predikát `kocka(X)`. Proměnná `X` stále nemá žádnou hodnotu a hledá se predikát *kocka/1* v databázi shora dolů. První je nalezen fakt `kocka(micka)` a aby byl shodný s `kocka(X)`, musí platit `X = micka`. Je to pouze fakt, proto vyhodnocení uspěje a kladná odpověď se vrací zpět do těla predikátu *zvire/1*. Žádné další predikáty již vyhodnocovat nemusíme (ty jsou za středníkem, tedy volitelné), proto i celé pravidlo uspěje.

Nyní již máme přiřazení `Zvire = X = micka`. Žádné další návaznosti u cíle `zvire(Zvire)` nebyly, cíl je tedy vyhodnocen celý. Pokud jsme s výsledkem spokojeni, potvrdíme klávesou enter, interpretátor odpoví `yes` a jsme zpět v interaktivním uživatelském režimu. My se ale s takovou odpovědí nespokojíme a podíváme se na to, jak řídit backtracking.

4.4 Řízení backtrackingu

Vrat'me se zpět k ukázce kódu 4.1. Ze zápisu víme, že predikát *zvire/1* jistě splňuje více mazlíčků v databázi definované programem 3.1. Pro nalezení další možnosti použijeme středník a potvrdíme klávesou enter. Přiřazení proměnné se v průběhu celého procesu změní na `mnauka` a znovu se vypíše. V praxi to vypadá takto:

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

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

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

Proces se vrátí opět až do těla pravidla s hlavou `zvire(X)` s tím, že proměnná X je nyní opět nepřiřazena, a pokouší se najít další výskyt predikátu *kocka/1*. Ten nalezne, uspěje, přiřadí proměnnou `Zvire = X = mnauka` a vypíše.

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

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

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

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

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

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

Abychom viděli, jak interpretátor postupuje, potřebujeme se podívat, jak je definován predikát *bylozravec/1*. Z těla se vyhodnocují ještě predikáty *zere_byli/1* a *zere_maso/1*:

```
bylozravec(X) :- zere_byli(X), not zere_maso(X).
zere_maso(X) :- pes(X); kocka(X); krecek(X).
zere_byli(X) :- kralik(X); krecek(X).
```

Vyhodnocování probíhá totožně s předchozím příkladem. Jelikož jsou pravidla složitější, interpretátor dopředu neví, jestli je `matilda` poslední možnost a ještě čeká na interakci od uživatele. Chceme-li vyvolat další backtracking, cíl se již vyhodnotit nepovede a dostaneme negativní odpověď.

4.4.1 Řízení v pravidlech

Backtracking je pro nás velice užitečný, ale někdy nám dokáže přidělat vrásky. Proto potřebujeme mít možnost jej nějakým způsobem kontrolovat i při definování pravidel. Řekněme, že všechna zvířata mají srst, až na Besi, která je naháč. Definujeme následující pravidla:

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

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

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

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

Pro konkrétní dotazy nyní interpretátor reaguje správně:

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

Zápisu `!, fail` se říká „cut with failure“. Bohužel ani takové ošetření nefunguje ve všech případech správně. Položme obecný dotaz „kdo má srst?“:

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

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

Lepší přístup je takový, jaký jsme si již ukázali u potravinových strategií. Definujeme si nový predikát *nahac/1* a budeme tvrdit, že srst má zvíře, které není naháč. Takto bude vypadat upravená část:

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

Nyní když zkusíme vyhodnotit cíl `ma_srst(X)`, dostaneme správné výsledky. V kódu vidíme nový predikát (operátor) `\+/1`. Ten má stejnou funkci jako `not`, který jsme si již vysvětlovali v příkladu 3.2.

4.5 Návratová hodnota

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

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

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

Při definování predikátu nemůžeme nijak explicitně určit návratovou hodnotu, proto je predikát binární a návratovou hodnotu představuje druhý argument. Nyní program opět načteme a vyzkoušíme funkčnost. Když chceme vědět, jak vypadá Matilda, položíme dotaz tímto způsobem:

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

S takto definovaným predikátem také můžeme zadat požadovanou návratovou hodnotu a ptát se na vyhovující zvířata. Takové chování nám umožňuje backtracking společně s unifikací. Jsou to procesy, které v Prologu fungují automaticky všude a nemusíme je popisovat žádným algoritmem, jak bychom to museli udělat v jazyce Java.

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

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

Tento příklad bude fungovat správně. Jako třetí argument zadáme v dotazu nedefinovanou proměnnou, do které je předáno větší číslo. Všimněme si při této příležitosti použití řezu. Kdybychom jej vynechali, může nastat následující problém [2]:

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

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

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

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

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

Proto nebudeme spoléhat na postupné vyhodnocování, protože mohou nastat případy, kdy se bude chovat program jinak, než si představujeme. Raději budeme přidávat do pravidel kontroly navíc, abychom zajistili robustnost řešení.

5 Standard ISO Prolog

Část standardních zabudovaných predikátů jsme si již představili v předchozích kapitolách, ale stále existuje hodně zabudovaných predikátů, které by se nám v další práci mohly hodit. Kompletní seznam najdeme v příloze A knihy Prolog Programming in Depth od M. Covingtona [3]. Ukážeme si zde na příkladech pro nás nejzajímavější.

5.1 Operátory

Kromě standardních struktur se v Prologu setkáme i s operátory. Ty se používají nejen pro numerické operace, ale můžeme si definovat i vlastní s požadovanou funkcí. Zavedeny jsou proto, abychom například operaci sčítání nemuseli zapisovat ve funktorové notaci, tedy `+(4, 7)`, ale mohli jsme použít přirozenější zápis `4 + 7`.

Existují tři druhy operátorů:
Prefixové – jejich zástupcem je predikát *not/1*. Píše se vždy před term, který představuje jeho argument. Například `not zere_maso(X).`
Infixové – jsou všechny binární operátory. Zapisují se mezi dva termy, které představují jeho argumenty. Je to např. oprátor disjunkce *(,)/2* nebo matematických operací jako sčítání *(+)/2* apod.
Postfixové – píší se za argument, vyskytují se velice zřídka.

Zkusme nyní vyhodnotit cíl `5+3.`:

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

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

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

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

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

Za *is* můžeme psát jakékoliv složité rovnice, bohužel ale nemůžeme používat nedefinované proměnné. Jak aproximovat výsledek složité rovnice s neznámými napsal M. Covington ve svém výzkumu [4]. Nám bude prozatím stačit základní práce s operátory.

5.1.1 Definice vlastních operátorů

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

```
% Muzi
muz(jan). muz(roman). muz(ales).
% Zeny
zena(stela). zena(ema). zena(lucie). zena(petra).
% Vztahy
manzele(jan, stela). manzele(ales, petra).

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

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

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

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

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

Priorita je celé číslo od nuly výše a znamená, v jakém pořadí se bude operátor vyhodnocovat před ostatními. Čím vyšší číslo, tím nižší priorita, tedy

pozdější vyhodnocení. Například násobení má prioritu 400, sčítání a odčítání 500 apod.

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

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

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

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

I v kódu musí být na začátku řádky zapsány znaky `?-`, jimiž začíná výzva v uživatelském režimu, event. znaky `:-`. Tomuto zápisu se říká *direktiva* [2]. Predikát *miluje/2* můžeme používat dále libovolně i ve funktorovém zápisu.

5.2 Problém symetrických relací

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

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

Odpověď na druhý dotaz je negativní, protože jsme definovali tělo pravidla *miluje* pouze jednostranně. Aby pravidlo platilo symetricky, definujeme nový predikát *chot/2* a upravíme pravidlo následujícím způsobem:

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

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

```
| ?- chot(X, Y).
X = jan
Y = stela ?;
X = ales
Y = petra ?;
X = stela
Y = jan ?;
X = petra
Y = ales
yes
```

Tento problém se objevuje u každé symetrické relace. Vhodné řešení může být v různých případech individuální a neexistuje žádné univerzální pravidlo. V našem případě stačí říci, že symetrické řešení chceme hledat pouze tehdy, když neexistuje řešení původní. K tomu použijeme nový zápis – podmínku.

5.2.1 Podmínka

Upravíme predikát *chot* následujícím způsobem:
`chot(X, Y) :- \+ manzele(X, Y) -> manzele(Y, X) ; manzele(X, Y).`

Zápis `predikat1 -> predikat2 ; predikat3` je podmínka „if“ jak ji známe z jiných programovacích jazyků. První se vyhodnotí `predikat1`, pokud uspěje, zavolá se `predikat2`, v opačném případě se zavolá `predikat3` – ten je ale v zápisu nepovinný. Pokud neuspěje `predikat2`, jako další se `predikat3` vyhodnocovat nebude kvůli závislosti implikace `->` [2].

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

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

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

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

5.2.2 Abstrakce

Na základě již existujících a dobře navržených faktů a pravidel lze vždy snadno přidat další funkcionalitu. Při psaní programů za pomoci klasických i deklarativních jazyků bychom měli udržovat vhodnou míru abstrakce. U Prologu abstrakci využijeme pro zjednodušení složitých pravidel. Každý velký problém se skládá z částí, ale záleží na nás, jak drobné části se budou vyskytovat na jednom místě.

Dekompozice by na vysoké úrovni neměla být úplná, ale měla by být rozvržena do více částí. To přispěje čitelnosti kódu a případně následné použitelnosti částí v jiných pravidlech. Budeme se snažit psát kód čistě a jednoduše. Pro ukázku zadáme nová fakta:

```
% Muzi
muz(jan).
muz(roman).
muz(ales).
muz(michal).
muz(josef).

% Zeny
zena(stela).
zena(ema).
zena(lucie).
zena(petra).

% Vztahy
manzele(jan, stela).
manzele(roman, ema).
manzele(ales, petra).

milenci(roman, petra).
milenci(lucie, ales).
```

A definujeme rozšiřující pravidla vztahů za použití doposud získaných znalostí:

```
% Obecna pravidla
chot(X, Y) :- \+ manzele(X, Y) -> manzele(Y, X) ; manzele(X, Y).
v_mileneckem_vztahu(X, Y) :- \+ milenci(X, Y) -> milenci(Y, X);
   milenci(X, Y).

?- op(200, xfx, podvadi_s).
X podvadi_s Y :- chot(X, Z), v_mileneckem_vztahu(X, Y).

podvadejici(X) :- X podvadi_s _.

?- op(200, xfx, podvadi).
X podvadi Chot :- chot(X, Chot), podvadejici(X).

?- op(200, xfx, miluje).
miluje(X, Y) :- milenci(X, Y) ; milenci(Y, X).
miluje(X, Y) :- manzele(X, Y), \+ podvadejici(X).
miluje(X, Y) :- manzele(Y, X), \+ podvadejici(X).
```

Pro program takto malého rozsahu je nyní dekompozice dostatečná. Všimněme si použití negací, podmínek, anonymní proměnné, definic operátorů atp. Je dobré používat korektnější názvy proměnných než jen `X` a `Y`. U binárních predikátů to ale většinou není potřeba, protože je nám z kontextu zápisu jasné, co proměnné znamenají.

5.3 Vstup od uživatele

Jestliže vytváříme program, ve kterém chceme nechat uživatele zadat libovolný vstup, aniž by pro něj byly viditelné vnitřní struktury, použijeme k tomu predikát *read/1*. Jeho argumentem je proměnná, ve které bude následně přiřazen zadaný vstup. Predikát si můžeme vyzkoušet i v interaktivním prostředí interpretátoru:

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

Můžeme napsat pouze term dle konvencí jazyka Prolog (viz 3.1) ukončený tečkou a nakonec potvrdíme klávesou enter. Abychom mohli napsat libovolný řetězec znaků bez znalosti konvencí, definujeme si v kapitole 5.7 predikát, který to bude umožňovat.

5.4 Cyklus

Cykly jako `while` nebo `for`, jak je známe z klasických programovacích jazyků, v Prologu nenajdeme. K vytvoření cyklu využijeme backtracking a zabudovaný predikát *repeat/0*. Ten při přímém vyhodnocení uspěje a při zpětném chodu také. Před tento predikát se backtrackingem již nedostaneme a program se chová, jako kdyby našel další možné řešení. Po použití *repeat* musí existovat možnost, jak cíl splnit, nebo se bude cyklus opakovat donekonečna.

Vytvoříme si pro ukázku jednoduchý kvíz:

```
kviz :- write('Vitejte v pocetnim kvizu!'), nl, otazka.

otazka :- write('Kolik je 5+3?'), nl, repeat, read(X), odpoved(X).
odpoved(8) :- write('Spravna odpoved, dekuji').
odpoved(X) :- write('Odpoved '), write(X),
   write(' je chybna. Zkuste to prosim znovu'), nl, fail.
```

Pro lepší pochopení se podíváme rovnou na funkci tohoto programu:

```
| ?- kviz
Vitejte v pocetnim kvizu!
Kolik je 5+3?
| 6.
Odpoved 6 je chybna. Zkuste to prosim znovu
| 8.
Spravna odpoved, dekuji
yes
```

Startovací bod je predikát *kviz/0*. Vidíme výpis uvítání a volání predikátu *otazka/0*. Ten v těle obsahuje samotnou otázku a predikát *repeat/0*, který představuje cyklus při backtrackingu. Následuje načtení ze standardního vstupu do proměnné `X` a kontrola tohoto vstupu.

Otázka je jasně položena a má právě jednu možnost odpovědi. Proto stačí napevno definovat atom správné odpovědi. V případě, že je odpověď jiná, vypíšeme na obrazovku upozornění a zajistíme nesplnění cíle predikátem

fail. Tím nebude splněn ani cíl `odpoved(X)`, *read* při backtrackingu neuspěje, ale na predikátu *repeat* se backtracking zastaví a pokračujeme opět přímým chodem od zadání hodnoty.

5.5 Rekurze

Kvíz v předchozím příkladu je velice primitivní a zaslouží si rozšíření. Obohatíme jej o cyklus, který je řešen za pomoci jednoduché rekurze, a přidáme dynamičnost:

```
kviz :-
   write('Vitejte v pocetnim kvizu! Ukoncite jej slovem ''konec'''),
   nl, otazka(1, 1).

otazka(C1, C2) :-
   write('Kolik je '), write(C1), write(' + '), write(C2),
   write('?'), nl, repeat, read(X), Vysledek is (C1+C2),
   odpoved(X, Vysledek).

odpoved(konec, _) :- write('Dekuji za spolupraci, nashledanou.').

odpoved(X, Vysledek) :-
   (X == Vysledek) -> write('Spravna odpoved, dekuji'), nl,
   Z is ceiling(X*1.3), otazka(X, Z).

odpoved(X, _) :-
   write('Odpoved '), write(X), write(' je chybna. Zkuste to prosim
   znovu'), nl, fail.
```

Jak kvíz funguje si již určitě dokážeme představit. Novinka je přepočítávání nových hodnot k sečtení. Predikátu *odpoved/2* musíme předávat mimo tipovanou hodnotu i hodnotu správného výsledku, jelikož bude v každém kole jiná. Nové hodnoty vypočteme až po napsání správného výsledku a voláme rekurzivně predikát *otazka/2* s těmito hodnotami.

Kvíz bude aktivní až do napsání klíčového slova „konec“. Při chybné odpovědi se bude vracet backtrackingem až k predikátu *repeat/0*, při správné se bude rekurzivně pokládat nová otázka.

Zajímavějším příkladem pro rekurzi je počítání součtu všech čísel od nuly do daného čísla. Rekurze spočívá v tom, že pro požadovanou sumu do čísla N musíme spočítat součet do čísla $N - 1$ a k tomu číslo N přičíst. Pokud je

číslo $N = 1$, nic nepřičítáme:

```
suma_do(1, Vysledek) :- Vysledek is 1.
suma_do(X, Vysledek) :-
   C1 is X-1, suma_do(C1, C2), Vysledek is C2 + X, !.
```

Pokud bychom v druhém pravidle nepoužili řez, umožnili bychom vyvolat zpětný chod a interpretátor by rekurzivně počítal záporná čísla do nekonečna.

5.6 Seznamy

Velice užitečné jsou v prologu seznamy, neboli listy. V kapitole 3.1 jsme si již nastínili možnosti použití a zápis. Zde se podíváme na problematiku podrobněji včetně ukázkových příkladů. Zápis v hranatých závorkách `[a, b, c]` je zjednodušený zápis funktorové notace `.(a, .(b, .(c, [])))`.

5.6.1 Přidávání prvků

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

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

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

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

Zápis `[5,10,25|10]` je sice dle konvencí správný, ale s takovou strukturou se dále špatně pracuje. Řešením je vytvoření jednoprvkového seznamu:

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

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

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

5.6.2 Zpracovávání prvků

Nejdůležitější část práce s prvky seznamu je jejich zpracování. Záleží, na co seznam využíváme, ale postup je většinou ekvivalentní. Jak již bylo řečeno, můžeme provést dekompozici seznamu na první prvek (hlavu) a zbytek seznamu. Využijeme unifikaci s pravidlem, které jako argument přijímá zápis `[Hlava | Telo]`. Jako nejjednodušší příklad si ukážeme výpis všech prvků seznamu pod sebe:

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

Vyzkoušíme si navržené řešení:

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

Pokud bychom nedefinovali predikát `vypis_prvky_ocislovane([], _)`, vyhodnocení by se provedlo, ale odpověď by byla negativní, jelikož prázný seznam by se s žádným predikátem již neunifikoval. Hlava seznamu je vždy první term neprázdného seznamu a zbytek neboli tělo původního seznamu je vždy seznam bez prvního prvku, může být tedy i prázdný.

Definujeme si s aktuálními znalostmi jednoduchou operaci sčítání číselných prvků seznamu. Vytvoříme pro to prefixový operátor *+/1*, který se napíše před seznam, který chceme sečíst.

Výsledek sčítání potřebujeme předat do proměnné, tak nám unární predikát stačit nebude. U numerické operace se nám naskytne použití operátoru *is/2*, jenže ten takový způsob sčítání nepodporuje. Zabudované predikáty nemůžeme předefinovat, proto si vytvoříme vlastní operátor *je/2*:

```
?-op(800, xfx, je).
?-op(200, fx, '+').

X je +[X] :- !.
X je +[H|T] :- Y je +T, X is Y + H.
```

Opět unifikujeme hlavu a tělo seznamu do zvláštních proměnných. Dokud je v seznamu více než jeden prvek, použijeme rekurzi. Poslední prvek přiřadíme do proměnné a použijeme řez, protože jiné možnosti součtu čísel neexistují. Výsledky rekurzivních volání přičítáme k současné hodnotě. Tímto způsobem se pole sečte zprava doleva:

```
| ?- X je +[-6,15+6,3*3,2**4,sqrt(36)].
X = 46.0
yes
```

Jak je vidět, v jednotlivých prvcích pole se mohou nacházet i složitější matematické výrazy, které podporuje operátor *is/2*. Ekvivalentně můžeme za pomoci nového prdikátu *je* definovat libovolné operace se seznamy nebo s rozšířené matematické operace. Další využití má seznam při zápisu řetězců.

5.7 Řetězce

Libovolné textové řetězce nejsou jenom obyčejné atomy, ale je to speciální posloupnost znaků uvozená dvojitými uvozovkami. Jako v jazyce C se řetězec reprezentuje pomocí pole (seznamu) jednotlivých znaků a každý znak má určený svůj číselný kód v ASCII tabulce:

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

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

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

S jeho pomocí si vyrobíme predikát *cti_radku/1*, který bude číst libovolný text ze vstupu až do odřádkování, bez nutnosti uvozovek nebo ukončení tečkou. Vytvoříme si proto nový soubor `Vstup.pro`:

```
cti_radku(X) :- cti_znak_rekurzivne([], List), name(X, List), !.
cti_znak_rekurzivne(Stary_list, Novy_list) :-
   get0(X), zpracuj_znak(X, Stary_list, Novy_list).

% Odchyceni odradkovani v Linuxu
zpracuj_znak(10, Stary_list, Stary_list).

zpracuj_znak(X, Stary_list, Novy_list) :-
   append(Stary_list, [X], L), cti_znak_rekurzivne(L, Novy_list).
```

Pro správnou funkci jsme využili znalosti rekurze, řezu i práce s listy. Vidíme nový zabudovaný predikát *get0/1*, který čte jeden libovolný znak ze vstupu uživatele. Predikát *get/1* funguje velice podobně, ale využít bychom jej nemohli, jelikož ignoruje všechny „bílé znaky“ neboli anglicky „whitespace“. Mezi ně se řadí merezy, odsazení a hlavně odřádkování, které potřebujeme odchytávat. Takto náš predikát funguje:

```
| ?- cti_radku(X).
Muzeme pouzit libovolne znaky jako treba: '?:_-/\%"' apod.
X = 'Muzeme pouzit libovolne znaky jako treba: \'?:_-/\%"\' apod.'
yes
```

Aby se z libovolného řetězce mohl stát atom, volání *name* ho vloží mezi jednoduché uvozovky. Na všechny výskyty těchto uvozovek uvnitř řetězce automaticky použije tzv. *escape sekvenci*. To je v tomto případě proces přidání zpětného lomítka před každou uvozovku.

5.8 Práce se soubory

V Prologu můžeme místo standardního vstupu a výstupu definovat jednoduše konkrétní soubory. Ty je možné otevřít pro čtení nebo zápis.

5.8.1 Čtení

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

Vytvoříme zkušební soubor `test` s obsahem:

```
Zkusebni soubor
se dvema radky
```

Abychom si vyzkoušeli čtení, přesměrujeme vstup na tento soubor a predikátem *seeing/1* ověříme, jestli opravdu ze souboru interpretátor čte [3]:

```
| ?- see('test').
yes
| ?- seeing('test').
yes
```

Pro jednoduchou ukázku, se souborem vytvořeným v Linuxu, můžeme použít již navržený a vyzkoušený predikát *cti_radku/1*:

```
| ?- cti_radku(X).
X = 'Zkusebni soubor'
yes
```

Čtení vstupu funguje jako fronta. Představme si, že máme otevřený pomyslný „průchod“, kterým přicházejí znaky. Nezáleží na tom, jestli pochází z klávesnice nebo ze souboru. Predikát *get0* postupně odebere první v řadě pokaždé, když je zavolán.

Jestliže je vstup standardní a žádné znaky nejsou ve frontě, čeká se na vstup z klávesnice ze strany uživatele. Vstup ze souboru je ovšem konečný a předem určený, takže musíme konec souboru hlídat.

Jistě víte, že konec řádky v Unixovém systému je určený netisknutelným znakem s označením `LF` neboli anglicky „Line Feed“. V systému Windows se používají dva znaky v pořadí `CR+LF`, kde `CR` znamená v angličtině „Carriage Return“.

Tyto znaky pochází ještě z počátků komunikace. V tehdejších dobách se vypisovalo mechanicky a znak `CR` měl význam návratu tiskové hlavy (volně přeloženo „návrat vozíku“) na začátek řádky. Znak `LF` měl za následek posun papíru o řádek výše.

Jednoduše tak můžeme přidat k původní kontrole konce řádky v Unixu i kontrolu na konec řádky v systému Windows. Další související kontrola je kontrola na konec souboru. Zde při pokusu o přečetení dalšího znaku dostaneme neexistující ASCII kód -1. Takto nyní vypadají všechny možnosti zpracování přečteného znaku:

```
% Odchyceni odradkovani ve Windows
zpracuj_znak(13, Stary_list, Stary_list) :- get0(_).
% Odchyceni odradkovani v Linuxu
zpracuj_znak(10, Stary_list, Stary_list).
% Konec souboru
zpracuj_znak(-1, Stary_list, Stary_list) :- !,fail.

zpracuj_znak(X, Stary_list, Novy_list) :-
   append(Stary_list, [X], L), cti_znak_rekurzivne(L, Novy_list).
```

Pokud bychomo chtěli podporovat ještě navíc platformu Mac OS, využívající pro odřádkování pouze `CR`, již by byly kontroly složitější. Když z kontrolního pravidla pro Windows odstraníme tělo, tedy přečtení dalšího znaku (u Windows následuje LF), bude program v Mac OS fungovat správně na úkor platformy Windows, kde se bude ještě vypisovat právě znak `LF`.

S takto definovanými pravidly můžeme zavolat predikát `cti_radku(X)` opakovaně:

```
| ?- cti_radku(X).
X = 'Zkusebni soubor'
yes
| ?- cti_radku(X).
X = 'se dvema radky'
yes
| ?- cti_radku(X).
no
```

Je vidět, že se voláním predikátu *cti_radku* rekurzivně přečetly všechny znaky až do prvního výskytu odřádkování, které jsme do výpisu nezahrnuli. Díky dvěma kontrolám funguje zápis ekvivalentně i se soubory vytvořenými pod systémem Windows. Opakované volání čte další znaky od posledního přečteného. Na základě těchto znalostí jistě zvládneme vytvořit predikát,

který po řádcích přečte celý soubor. To si ukážeme dále v této kapitole.

Když nyní připíšeme do souboru další řádky a zkusíme znovu přečíst další řádku, nic se už nevypíše. Je to dáno tím, že interpretátor B-Prolog soubor při provedení *see* načte jednorázově na vstup a s tímto vstupem můžeme dále libovolně pracovat.

Pokud chceme definovat opět standardní vstup, nemusíme znát jeho pojmenování, ale můžeme použít predikát *seen/0*.

```
| ?- seen.
yes
| ?- seeing(X).
X = user
yes
```

Za pomoci predikátu *seeing* jsme zjistili, že standardní vstup je pojmenován *user*.

Nyní si vytvoříme nový soubor pro program, který bude definovat jednoduchá pravidla pro práci se soubory. Nazveme si jej `Soubory.pro` a definujeme si predikát pro čtení. Využijeme přitom kód z programu `Vstup.pro`. Předpokládejme, že jsou soubory ve stejné složce, tak abychom část nemuseli kopírovat, stačí použít dobře známý predikát *consult*:

```
?-consult('Vstup.pro').

% opakujeme, dokud se dari vypisovat
cti_soubor(Soubor) :-
   see(Soubor), repeat, \+ vypis_radku, !, seen.
vypis_radku :- cti_radku(X), write(X), nl.
```

V těle predikátu je použit jednoduchý cyklus. Nejprve zvolíme vstupní soubor a použijeme „zarážku“ v podobě predikátu *repeat*. Pokaždé, kdy se povede vypsat řádku, zajístíme negací, aby pravidlo neuspělo. Tím se přečte další řádka. Až když není co dál číst, vyhodnocení je pozitivní a dostaneme se k řezu. Ten zde zajišťuje jednoznačnost výsledku. Následuje přepnutí zpět na standardní vstup.

Zkusme vypsat na obrazovku upravený textový soubor `test`:

```
| ?- cti_soubor(test).
Zkusebni soubor
se dvema radky
PS: jeste jeden radek.
yes
```

5.8.2 Zápis

Při zápisu do souboru postupujeme podobně jako při čtení. Také stačí pouze intepretátoru přesměrovat standardní výstup na náš soubor a můžeme používat predikáty jako při klasickém vypisování na obrazovku.

Jako u čtení jsou pro nás pro zápis důležité tři predikáty *tell/1*, *told/0* a *telling/1* [3]. Jako ekvivalent k *tell/1* je v některých verzích prologu predikát *append/1*, který umožní zápis na konec zvoleného souboru, zatímco *tell/1* aktuální obsah přepíše, nebo vytvoří soubor nový, pokud neexistuje [3]. Ukažme si použití v pravidlech:

```
zapis_radku_do_souboru(Soubor, X) :-
       tell(Soubor), write(X), nl, told.
zapis_vstup_do_souboru(Soubor) :-
       cti_radku(X), zapis_radku_do_souboru(Soubor, X).
```

Predikát *zapis_radku_do_souboru/2* pouze přesměruje výstup na požadovaný soubor, zapíše text předaný proměnnou `X` a přesměruje zpět na standardní výstup. Predikát *zapis_vstup_do_souboru/1* rozšiřuje stávající o možnost zadat text z klávesnice nebo z jiného souboru. Podobným postupem bychom mohli např. upravit obsah vstupního souboru a po řádcích jej opět uložit do jiného souboru.

B-Prolog bohužel nepodporuje predikát *append/1*, který je pro práci se soubory velice užitečný. Naštěstí mimo výše jmenované způsoby existuje ještě přístup, který ocení spíše pokročilejší programátoři, jelikož je podobný standardům v jiných jazycích.

5.8.3 Jiný způsob

Pro pokročilejší práci s proudem dat směřujících z/do souboru, použijeme zabudovaný predikát *open/3* nebo *open/4*. Zápis vypadá následovně:

`open(Soubor, Akce, Proud, Moznosti)`

kde `Soubor` představuje soubor, se kterým chceme pracovat, `Akce` určuje, co chceme se souborem dělat. Máme možnosti `read` = čtení, `write` = zápis nebo `append` = přidání na konec souboru. `Proud` je výstupní proměnná představující pomyslný „komunikační kanál“ se souborem. `Moznosti` je seznam doplňujících nastavení. Tento argument můžeme vynechat.

Mezi nejzajímavější možnosti nastavení patří přepnutí typu čtení. Primárně se pracuje s obsahem souboru jako s textem a čtení či zápis probíhá po

znacích. Můžeme přepnout typ na binární soubor nastavením `type(binary)` a poté jsou operace bitové.

Další možností je pojmenování proudu (tzv. *alias*). K tomu použijeme libovolný atom, pomocí kterého můžeme proud při další práci identifikovat. Pojmenování se určuje predikátem `alias(proud1)`.

Ve standardu jsou definovány ještě možnosti nastavení chování při dosažení konce souboru pomocí predikátu *eof_action/1*. Nejzajímavější nastavení je `eof_action(reset)`, díky kterému se pokusí interpretátor po dosažení konce souboru znovu jej prohledat a zkontrolovat, zda je možné číst ještě dále, v případě že by do souboru někdo zapsal.

Poslední možností je umožnění posouvání se v souboru dopředu nebo dozadu pomocí predikátu `reposition(true)`. Posouvání je ve výchozím nastavení deaktivováno. Řídicí predikáty je možné v seznamu zapsat v libovolném pořadí i počtu.

Abychom mohli pracovat s nově otevřeným proudem, jsou standardem určeny speciální predikáty:

Čtení:
`get_char(Proud, Znak)` – čtení jednoho znaku
`get_code(Proud, Kod)` – čtení numerického kódu znaku
`read_term(Proud, Term, Moznosti)` a `read(Proud, Term)` – čtení jednoho termu

Zápis:
`put_char(Proud, Znak)` – zápis jednoho znaku
`put_code(Proud, Kod)` – zápis numerického kódu znaku
`write_term(Proud, Term, Moznosti)` a `write(Proud, Term)` – zápis jednoho termu

Pokud použijeme predikát *set_output/1* nebo *set_input/1*, kterým nastavíme aktuální proud čtení/zápisu na náš nově otevřený, můžeme používat i predikáty *read/1*, *write/1* a podobné, které nepotřebují jako argument tento proud [3].

Pro ukázku si můžeme zkusit jednoduchý zápis na konec našeho testovacího souboru:

```
| ?- open(test, append, Proud), set_output(Proud), write('ahoj'),
   close(Proud).
Proud = (stream)[10002]
yes
```

Musíme si dávat pozor také na důsledné uzavírání otevřených proudů pomocí predikátu *close/1*. Pokud uzavřeme proud, na který jsme přesměrovali vstup nebo výstup, zruší se i přesměrování a je použit standardní proud. Aktuálně nastavený proud zjistíme voláním predikátu *current_input/1* nebo *current_output/1*.

Výše zmíněné predikáty *tell*, *see* apod. používají vnitřně právě tyto konstrukce, jen jsou zapouzřené do přívětivější podoby, na úkor možnosti jejich úprav.

5.9 Moduly

Pozornost si zaslouží i moduly, přestože interpretátor B-Prolog je v aktuální verzi nepodporuje. Modul je ekvivalent třídy v jazyce Java. Do modulu můžeme zapouzdřit predikáty, které spolu úzce souvisí a zpřístupníme veřejně pouze ty, které chceme. Ukažme si definici takového modulu, bohužel bez možnosti praktického vyzkoušení:

```
?- module(informace).
?- export([vypis_info/0]).
?- begin_module(informace).

vypis_info :- write('Nasleduji informace: '), info(X), write(X), !.
info('Verejna informace').
info('Tajna informace').

?- end_module.
```

Při definici záleží na pořadí predikátů. První musíme definovat nový modul direktivou *module/1* a následuje seznam veřejně přístupných predikátů, který předáme jako argument při volání *export/1*. Mezi direktivami *begin_module/1* a *end_module/0* definujeme libovolný program, kde se musí vyskytovat predikáty vypsané v direktivě *export*.

Pokud chceme nyní v jiném programu volat predikáty definované v určitém modulu, musí být modul v interpretátoru načtený a v požadovaném programu jej importujeme direktivou *import/1*. Direktivy jsou vždy platné pouze v rámci jednoho souboru.

Když budeme chtít zavolat neveřejný predikát z nějakého modulu, je to také možné za pomoci zápisu `nazev_modulu:vnitrni_predikat`. Pomocí zápisu `predikat @ jiny_modul` řekneme interpretátoru, že chceme zavolat

predikát `predikat` z modulu `jiny_modul` i v případě, že bude `predikat` definován v aktuálním programu [3]. Předpokládejme načtení programu, kde je importován modul *informace* definovaný výše:

```
| ?- vypis_info.
Nasleduji informace: Verejna informace
yes
| ?- informace:info(X).
X = 'Verejna informace' ?;
X = 'Tajna informace'
yes
```

Při volání predikátu *info/1* bez definování modulu by interpretátor skončil s výjimkou.

5.10 Úprava nahrané databáze

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

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

Jestliže chceme přidat více predikátů najednou nebo zadávat pravidla, použijeme volání `consult(user)`. To místo ze souboru načítá vstup z klávesnice a můžeme psát stejným způsobem, jako zapisujeme do souboru. V B-Prologu ukončíme zadávání kombinací kláves `ctrl + D`.

V případě, že chceme fakta z databáze odstranit, použijeme predikát *retract/1*. Jako argument zapíšeme predikát a první v databázi, se kterým se tento predikát unifikuje, bude z databáze vymazán. Pro vymazání všech odpovídajících použijeme *retractall/1*. Pro vymazání všech definic konkrétního predikátu použijeme `abolish(Funktor/Arita)` [11].

Pozor na fakta a pravidla nahraná pomocí predikátu *consult*. Na jejich

smazání nemáme dostatečná oprávnění. Vyzkoušíme si budování databáze v praxi:

```
| ?- assert(pred(a)), assert(pred(b)), assert(pred(c)
yes
| ?- asserta(pred(d)).
yes
| ?- assert(pred(1, a)).
yes
| ?- listing.
pred(d).
pred(a).
pred(b).
pred(c).
pred(1, a).
yes
| ?- retract(pred(X)).
X = d ?;
X = a ?
yes
| ?- listing.
pred(b).
pred(c).
pred(1, a).
yes
| ?- abolish(pred/1).
yes
| ?- listing.
pred(1, a).
| ?- abolish.
yes
| ?- listing.
yes
```

Předoposlední volání predikátu *abolish/0* maže všechny námi definovaná fakta.

6 Vývojové prostředí B-Prolog

V závěru se podíváme podrobněji na interpretátor B-Prolog. Hlavním cílem je ukázat možnosti Prologu nejen jako interpretovaného jazyka, ale také jako nástroje pro řešení dílčích problémů ve formě samostatného spustitelného programu nebo jako součást programu napsaného v jiném jazyce.

Jak se prostředí instaluje a spouští jsme si již řekli v kapitole 2.3.2. Víme, že interpretátor volí výchozí složku takovou, ze které byl z příkazové řádky příkazem spuštěn. Při spuštění můžeme používat parametry. Nejzajímavější je parametr `-g "prikazy"`, který vykoná zadané příkazy (cíle) bezprostředně po startu interpretátoru.

Jestliže chceme provést příkaz ještě před startem, napíšeme na konec sledu příkazů predikát `$bp_top_level`, který inicializuje start interpretátoru až po těchto úkonech.

6.1 Změny oproti standardu

B-Prolog z velké části podporuje standard ISO Prolog. Někdy ale nejsou funkce totožné s navrhovanými požadavky standardu. Obsahuje také velké množství přidaných predikátů. Všechny možnosti B-Prologu jsou sepsány v rozsáhlé příručce přikládané vždy k aktuální verzi interpretátoru [11]. Ukážeme si možnosti, které by nám při tvorbě předchozích příkladů usnadnily práci nebo pomohly v jejich rozšíření.

6.1.1 Vlastní programy

Při práci oceníme predikát *[]/1*, který má stejnou funkci jako již dobře známý predikát *consult/1*. Cesta k souboru se pouze zapíše do hranatých závorek stejně jako seznam. Můžeme také vypsat najednou více souborů, které se mají nahrát. Fakta a pravidla definovaná v těchto souborech se nahrají do databáze a vidíme je při použití *listing/0*.

Interpretátor umožňuje kompilaci do bajtkódu podobně jako v jazyce Java. Z našich programů můžeme takto vytvořit samostatně funkční celky. Ke kompilaci slouží predikát *compile/1*, který jako argument přijímá název souboru a na jeho základě vytvoří binární soubor se stejným názvem a příponou `.out`.

Zkompilovaný program načteme buď pomocí *load/1* nebo zadáním názvu jako argumentu při spouštění interpretátoru. Pro rychlou kompilaci a spuštění využijeme predikát *cl/1*, který ale nevytvoří binární soubor. Standardně v samostatných programech definujeme vstupní predikát *main*, který by se měl automaticky po načtení programu vyhodnotit. Při zadání argumentem se po dokončení běhu programu ukončí i interpretátor. Za název programu můžeme vypsat libovolné argumenty, které v programu načteme použitím `get_main_args(Argumenty)` do proměnné `Argumenty` jako seznam.

Spuštění funguje ve starších verzích, bohužel v aktuální verzi 8.1 se *main* nezavolá, ale pouze se spustí interpretátor s nahraným programem.

6.1.2 Komunikace s OS

Interpretátor má v sobě navíc predikáty pro komunikaci s operačním systémem. Kromě predikátu *write*, který umožňuje výpis na obrazovku je zde řada dalších. Libovolný příkaz systému zadáme prostřednictvím predikátu `system(Prikaz, Status)`, kde `Status` je volitelná proměnná, kam se uloží návratová hodnota po vyhodnocení příkazu `Prikaz`.

Pro zjištění aktuální cesty, se kterou interpretátor pracuje, použijeme `getcwd(Cesta)`. Pro změnu cesty spouží predikát *cd/1*, který jako argument přijímá cestu absolutní nebo relativní v našem souborovém systému. Seznam souborů ve složce získáme vyhodnocením `directory_files(Slozka, Seznam)`.

Lze také kopírovat soubory pomocí *copy_file/2*, případně mazat soubory voláním *delete_file/1*. Složky mažeme voláním *delete_directory/1* a vytváříme pomocí *make_directory/1*.

Aktuální datum vypíšeme predikátem *date/1* a aktuální systémový čas voláním `time(Hodiny, Minuty, Sekundy)`.

6.1.3 Volání predikátů

Predikáty voláme ve většině interpretátorů stejným způsobem. Pro vyhodnocení cíle existuje ve standardu ještě predikát `call(Cil)`, který funguje totožně jako klasické zadání cíle do interaktivního rozhraní, a dále predikát `once(Cil)`, který zabraňuje backtrackingu stejně jako volání `call(Cil, !)`.

Užitečné rozšíření představuje predikát *time_out/3*, který se zapisuje ve tvaru `time_out(Cil, Cas, Vysledek)`. Ten jednoduše vyhodnotí `Cil` po-

mocí volání *once*, ale pokud před vyhodnocením uplyne čas nastavený argumentem `Cas`, volání se ukončí a v proměnné `Vysledek` je přiřazen atom `time_out`. V opačném případě se `Vysledek` unifikuje na `success`.

Můžeme se setkat s případem, kdy potřebujeme vyvolat akci až když se přiřadí proměnné hodnota. K tomu využijeme predikát *freeze/2*. První argument je proměnná a druhý je cíl, který se má zavolat. Ten je zavolán pomocí *once* právě tehdy, když je proměnná unifikována s konkrétním termem.

Predikát `forall(Moznosti, Volani)` nalezne postupně všechny možnosti vyhodnocení termu `Moznosti` a pro každou z možností vyvolá cíl `Volani`. Jako příklad je v manuálu uvedeno procházení prvků seznamu. Zajímavé je pro nás i tělo definice tohoto predikátu:

```
forall(Moznosti, Volani) :- \+ (call(Moznosti), \+ call(Volani)).
| ?- forall(member(X,[a,b,c]),write(X)).
abc
yes
```

6.1.4 Práce se seznamy

B-Prolog definuje mimo jiné i základní operace se seznamy, které se ve standardu nevyskytují. V posledním příkladu vidíme použití nového predikátu `member(X, Seznam)`, který kontroluje, zda `X` je prvkem seznamu `Seznam`. Pokud proměnnou `X` nepřiřadíme, bude unifikována s prvním prvkem seznamu. Pomocí backtrackingu můžeme postupně prohledat celý seznam od začátku do konce.

Používaný je také predikát *length/2*, který zjistí délku seznamu uvedeného v prvním argumentu a číslo unifikuje s druhým argumentem.

Pomocí *sort/2* seřadíme seznam v prvním argumentu vzestupně. U predikátu *sort/3* můžeme navíc jako první argument definovat řadící operátor `<`, `>`, `=<` nebo `>=`. Predikát `keysort(S1, S2)` pracuje se seznamem párů a jeho výstupem je seznam seřazený podle prvního prvku z páru:

```
| ?- keysort([(3,a), (2,c), (1,b)], Serazene).
Serazene = [(1,b),(2,c),(3,a)]
Yes
```

Je to obdoba *mapy* v jazyce Java, kde první prvek páru je klíč, druhý prvek je hodnota.

Ze seznamu můžeme konkrétní prvky úplně vymazat, nebo je postupně odebírat následujícím způsobem:

```
| ?- delete([a,b,a,b,c,a,g], a, List).
List = [b,b,c,g]
yes
| ?- select(a, [a,b,a,b,c,a,g], List).
List = [b,a,b,c,a,g] ?;
List = [a,b,b,c,a,g] ?;
List = [a,b,a,b,c,g] ?;
no
```

Prvek ze seznamu na konkrétní pozici vybereme pomocí predikátu *nth/2* (prvky číslované od jedničky) nebo *nth0/2* (prvky jsou číslované od nuly):

```
| ?- nth(2, [1,2,3], X).
X = 2
yes
| ?- nth0(2, [1,2,3], X).
X = 3
yes
```

6.1.5 Cyklus, kolekce a formátování výstupu

Cykus `foreach` se zapisuje se ve tvaru:

```
foreach(X1 in K1, ..., Xn in Kn, Lokalni_promenne, Cil)
```

X_i představuje proměnnou případně term, K_i je kolekce, přes kterou se bude iterovat. Iterací může být libovolné množství a projdou se všechny možnosti stejným postupem, jak funguje backtracking, tedy zleva doprava a s vyvolanou opakovanou změnou možných hodnot.

Proměnná `Lokalni` představuje seznam proměnných, které jsou dočasné pro jednu iteraci a `Cil` je libovolný cíl, který se bude při každé iteraci vyhodnocovat.

Abychom prošli všechny prvky seznamu, můžeme použít cyklus `foreach` následovně:

```
| ?- foreach(N in [1,2,3], (write('Cislo '), writeln(N))).
Cislo 1
Cislo 2
Cislo 3
yes
```

Podívejme se na složitější zápis včetně použití kolekce čísel a formátování výstupu:

```
| ?- foreach(I in 1..3,J in 2..-1..1,format("{~d, ~d}.\n",[I,J])).
{1, 2}.
{1, 1}.
{2, 2}.
{2, 1}.
{3, 2}.
{3, 1}.
yes
```

Kolekce čísel ve tvaru `Od..Do` nebo `Od..Krok..Do` se používá pouze v cyklu. `Od` je počáteční číslo, `Do` je konečné číslo a `Krok` se k původnímu číslu v každé iteraci přičítá. Výchozí hodnota kroku je 1.

Z jazyka C známe funkci `printf`, které zadáme prvním argumentem řetězec se speciálními formátovacími značkami a druhý argument je výčet prvků, které se do řetězce za formátovací značky dosadí. Ekvivalentně funguje predikát *format/2*. Značky v řetězci začínají znakem '~'. V našem příkladu `~d` značí číslo. Můžeme také zapsat `~Nd`, kde `N` je číslo, které určuje, jak velké místo má být pro číslo vyhrazeno. Pokud se nevyužije celé místo, číslo se doplní zleva mezerami.

Další možností je `~a` pro výpis atomu bez uvozovek, `~Nc` pro výpis `N` stejných znaků (pokud `N` neuvedeme, vypíše se jeden). Pro vypsání znaku '~' použijeme zápis `~~`. Druhý argument bude obsahovat jeden nebo seznam všech prvků použitých ve formátovacím řetězci. Formátovací značky jsou vypsány v návodu [11] na straně 45.

6.1.6 Mód ladění

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

Mód v interpretátoru spustíme predikátem *trace/0*. V tomto módu vidíme vyhodnocení jednotlivých cílů od začátku do konce, včetně veškeré unifikace. Ladicí mód opustíme voláním *notrace/0*:

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

Vyhodnocování se hned zastaví a pomocí klávesy enter se můžeme posouvat po krocích, které interpretátor činí. Písmeno '`s`' přeskakuje aktuálně volaný predikát a zastaví se až po jeho úspěšném či neúspěšném vyhodnocení. Písmenem '`r`' necháme ladící mód vypsat všechny kroky až do konce vyhodnocení. V zastaveném ladění se dá vyvolat nápověda pomocí písmene '`h`' nebo znakem '`?`'.

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

6.1.7 Globální proměnné

V programech můžeme definovat proměnné, které jsou poté použitelné napříč celým během programu. Globální proměnnou definujeme predikátem `global_set(Nazev, Hodnota)`. Hodnotu globální proměnné získáme voláním `global_get(Nazev, Hodnota)`.

Název musí být atom, hodnota je libovolný term. Zda je zadaný atom globální proměnná zjistíme pomocí *is_global/1*:

```
| ?- global_set(promenna, 3).
yes
| ?- is_global(promenna).
yes
| ?- global_get(promenna, Hodnota).
Hodnota = 3
yes
```

6.2 Propojení s imperativními jazyky

Klíčovou vlastností dnešních interpretátorů je možnost oboustranné spolupráce Prologovského programu s klasickým přístupem k programování. Nejčastěji se setkáme s propojením s jazyky Java, C, C++ nebo C#. To se odvíjí od toho, v jakém jazyce byl interpretátor napsán.

Abychom mohli realizovat např. propojení s jazykem C, potřebujeme speciální knihovny nebo zdrojové kódy interpretátoru. V některých implementacích jsou volně dostupné, ale B-Prolog se bohužel řadí mezi ty, které mají tyto funkce draze zaplacené. Podíváme se pouze teoreticky na propojení s jazykem C. Podpora jazyka Java není v aktuální verzi interpretátoru příliš rozšířena.

V návodu je podrobně popsáno, jakým způsobem můžeme vyvolávat programové segmenty jazyka C v Prologu a obráceně, včetně praktických příkladů. Ukážeme si stručně, jak s propojením začít.

Nejprve je třeba přidat do systému konstantu `BPDIR`, která bude odkazovat na instalaci B-Prologu. Zdrojové kódy bychom měli po zakoupení licence najít ve složce `Emulator`.

6.3 Propojení s jazykem C

6.3.1 Volání C z Prologu

Pro vytvoření funkcí v jazyce C takových, abychom je mohli volat z interpretátoru jako predikáty, je potřeba mít k dispozici veškeré zdrojové kódy interpretátoru. Vytváření takového predikátu probíhá ve třech krocích:

1) Vytvoříme soubor, kam definujeme funkci, která bude provádět požadovanou akci. Funkce musí být bez argumentů a musí vracet celočíselnou hodnotu. Ke komunikaci s interpretátorem využíváme sadu předpřipravených funkcí. Ty jsou deklarované v hlavičkovém souboru `bprolog.h`, který musíme zahrnout do našeho programu pomocí `#include "bprolog.h"`. Použití těchto funkcí si přiblížíme dále.

2) Nově vytvořený soubor s funkcí musíme zahrnout do kódu interpretátoru, nejčastěji přímo do `cpreds.c`, kde jsou definovány zabudované predikáty zapsané v jazyce C. Řekněme, že naše funkce je deklarována jako `int moje_funkce()` a nový predikát určíme jako *volaniC/2*.

Do těla funkce `Cboot()` přidáme následující řádky:

```
extern int moje_funkce();
insert_cpred("volaniC", 2, moje_funkce);
```

3) Překompilujeme celý interpretátor a po spuštění můžeme volat predikát *volaniC/2* stejně, jako ostatní zabudované predikáty.

Vraťme se k vytváření samotné funkce. Každý term je popsán strukturou `TERM`, se kterou pracují předdefinované funkce. Strukturu získáme jako návratovou hodnotu funkce nebo ji předáme argumentem. V aktuální verzi intepretátoru začínají názvy všech funkcí předponou „bp_“.

I přesto, že nový predikát bude binární, funkci deklarujeme bez argumentů. Ty následně získáme voláním funkce:

```
TERM bp_get_call_arg(int i, int arita)
```

kde `i` je *i-tý* argument (počítáno od 1) a `arita` je arita predikátu.

Existuje sada funkcí, které kontrolují typ termu. Návratové hodnoty jsou `BP_TRUE` při úspěchu nebo `BP_FALSE` při neúspěchu. Následuje výčet několika funkcí a podmínky jejich úspěšného vyhodnocení:
`int bp_is_integer(TERM t)` – `t` je celé číslo
`int bp_is_atom(TERM t)` – `t` je atom
`int bp_is_list(TERM t)` – `t` je seznam
a další.

Kontroly potřebujeme pro konverzi struktury `TERM` na datový typ, který můžeme snadno zpracovat pomocí klasických postupů v C. Pokud známe typ termu, můžeme používat odpovídající funkce:
`int bp_get_integer(TERM t)` – vrátí celočíselnou hodnotu termu `t`
`double bp_get_float(TERM t)` – vrátí reálnou hodnotu termu `t`
`(char *) bp_get_name(TERM t)` – vrátí ukazatel na řetězec, který odpovídá názvu atomu nebo struktury `t`
`int bp_get_arity(TERM t)` – vrátí aritu atomu nebo struktury `t`

Pokud se konverze nepovede, jsou vráceny výchozí (nulové) hodnoty a globalní proměnná `exception` je nastavena na chybovou hodnotu. Jazyk C nepodporuje vyvolávání a odchytávání výjimek, proměnnou proto musíme kontrolovat sami.

Dva termy můžeme mezi sebou unifikovat pomocí funkce:

```
int bp_unify(TERM t1, TERM t2)
```

a jako návratovou hodnotu dostaneme `BP_TRUE` nebo `BP_FALSE`. Argument na *i-té* pozici získáme voláním funkce:

```
TERM bp_get_arg(int i, TERM t)
```

jejíž návratovou hodnotou je term na požadované pozici, pokud existuje. V opačném případě je oznámena chyba.

Práce se seznamy je realizována funkcemi:

```
TERM bp_get_car(TERM t) a TERM bp_get_cdr(TERM t)
```

kde `car` je první prvek seznamu a `cdr` je zbytek, tedy opět seznam. Výpis termu zajistí funkce `void bp_write(TERM t)`.

Poslední možností je vytváření vlastních termů v kódu jazyka C:
`TERM bp_build_var()` – vytvoří prázdnou proměnnou
`TERM bp_build_atom(char *nazev)` – vytvoří atom
`TERM bp_build_integer(int cislo)` – vytvoří celočíselný term
`TERM bp_build_nil()` – vytvoří prázdný seznam
`TERM bp_build_list()` – vytvoří prázdný seznam, který můžeme libovolně definovat
`TERM bp_build_structure(char *nazev, int arita)` – vytvoří strukturu *nazev/arita*, jejíž argumenty můžeme libovolně definovat

Jednoduchý příklad k této problematice nalezneme v manuálu [11] na straně 99.

6.3.2 Volání Prologu z C

Pokud chceme z programu napsaného v jazyce C zavolat sekvenci jazyka Prolog, změníme zdrojový kód spouštěcího souboru `main.c` za náš vlastní. Pred voláním predikátů musíme použít funkci:

```
initialize_bprolog(int argc, char *argv[])
```

Ta alokuje všechnu potřebnou paměť a nahraje sadu zabudovaných predikátů. Pokud se nepodaří interpretátor spustit, funkce vrátí hodnotu `BP_ERROR`.

Jednotlivé cíle můžeme volat pomocí `int bp_call_string(char *cil)` zadáním termu ve formě řetězce nebo pomocí struktury `TERM` voláním funkce `int bp_call_term(TERM cil)`. Ta navíc podporuje unifikaci proměnných.

Další možností je použití funkce *bp_mount_query_string* přijímající řetězec nebo *bp_mount_query_term* přijímající strukturu `TERM`. Tyto funkce „naplánují“ další cíl k vyhodnocení. Poté můžeme vícekrát za sebou použít funkci `int bp_next_solution()`, která nalezne další řešení. Pokud není žádný cíl naplánován, její návratová hodnota bude `BP_ERROR`, pokud již další cíl nelze vyhodnotit, vrátí `BP_FALSE`.

7 Závěr

Již od začátku práce je brán zřetel na čtenáře, kteří s Prologem tepreve začínají. Přestože pracuji výhradně s prostředím B-Prolog, čtenář má šanci naučit se základy jazyka Prolog i za použití jiného interpretátoru. Může si vybrat na základě stručného shrnutí aktuálních implementací, případně jakýkoliv jiný interpretátor podporující standard ISO Prolog. Záleží na individuálních požadavcích na následné využití jazyka.

Praktické příklady se prolínají s teorií. Samozřejmostí je pozvolné stupňování náročnosti prezentovaných příkladů. Ty jsou nejdříve napsány jednoduchým způsobem a dále rozšiřovány. Cílem je vytvářet programy dostatečně dekomponované s následnou možností snadného rozšíření funkcionality.

Na ukázkách poukazuji na zápisy, kterým je dobré se vyvarovat a navrhuji, jak dále a lépe při tvorbě programů postupovat. Čtenář by měl být následně schopen psát programy čistě, a měl by mít dostatečné znalosti pro samostatné řešení problémů spojených s návrhem složitějších programů v Prologu.

Pro účely výuky předmětu KIV/UZI byla, na základě podkladů od vedoucího práce, vytvořena přehledná prezentace shrnující důležité faktory základní práce s Prologem. Ta bude následně umístěna na webové stránky předmětu společně s programy, které jsou k náhledu i v textu práce. Mimo to bude k dispozici i množina dalších klasických Prologovských problémů pro prezentaci získaných znalostí a funkčnosti interpretátoru samotného. Shodný obsah, včetně návodu k obsluze, bude k nalezení na přiloženém CD.

B-Prolog skýtá mnohá vylepšení oproti standardu. Největší výhodou je možnost propojení s jazykem C. Nevýhodou je, že některé zabudované predikáty nefungují dle příručky, ale to mohou mít za následek časté aktualizace interpretátoru, který se neustále vyvíjí. Přestože na oficiálních webových stránkách projektu a v licencích je uvedeno, že B-Prolog je pro osobní a akademické účely zdarma, zdrojové kódy potřebné k propojení s jazykem C k dispozici ke stažení nejsou. Prostřednictvím elektronické komunikace s p. Zhou mi bylo sděleno, že zdrojové kódy je možné zpřístupnit za poplatek $2,980, který vytvořil velkou překážku při realizaci kombinovaných programů. Proto je tato problematika probrána pouze stručně a teoreticky.

Spojení s jazykem Java je bohužel v aktuální verzi podporováno minimálně, a to pouze na platformě Windows 32-bit. Rozšíření se údajně plánuje v následujících verzích B-Prologu. Výrobci se nyní více soustředí na nový projekt – *Picat*, který je nepřímým následovníkem B-Prologu.

Literatura

[1] Blackburn, P.; Bos, J. and Striegnitz, K.: Learn Prolog Now!. *Texts in Computing*, 2006, vol. 7. ISBN: 1-904987-17-6, s. 6-15.

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

[3] Covington, M. A.; Nute, D.; Vellino, A.: *Prolog Programming in Depth*. The University of Georgia, Athens, September 1993. Appendix A – ISO Prolog: A Summary of the Draft Proposed Standard, 1993. ISBN: 0-673-18659-8, 27 s.

[4] Covington, M. A.: Research Report AI-1989-02. *A Numerical Equation Solver in Prolog*. The University of Georgia, Athens, March 1989, 14 s.

[5] Heng, Ch.: *Free Prolog Compilers and Interpreters* [online]. Last updated March 24, 2014 ©1999-2014 [cit. 26. prosince 2013].
Dostupné z: `http://www.thefreecountry.com/compilers/prolog.shtml`

[6] Kantrowitz. M.: *Prolog Resource Guide* [online]. Version: 1.36, Last updated Feb 20, 1997. ©1992-1994. [cit. 13. dubna 2014].
Dostupné z: `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/html/faqs/lang/prolog/prg/top.html`

[7] Kryl, R.: *Neprocedurální programování: Úvod do programovacího jazyka PROLOG* [online]. Verze 3.03. KSVI MFF UK, Praha. ©Rudolf Kryl, 2013 [cit. 13. dubna 2014].
Dostupné z: `http://ksvi.mff.cuni.cz/~kryl/prolog.pdf`

[8] Skoupil, D.: *Úvod do paradigmat programování*. Katedra matematické informatiky, Univerzita Palackého v Olomouci, 1997, s. 9-23

[9] Sterling, L.; Shapiro, E.: *The Art of Prolog: advanced programming techniques*. Massachusetts Institute of Technology, 1994. Appendix A – Operators. ISBN: 0-262-19338-8, s. 479-481

[10] Wielemaker, J. aj.: *SWI Prolog Reference Manual*. Herstellung und Verlag: Books on Demand GmbH, Norderstedt, 2012. ISBN 978-3-84-822617-7, s. 59.

[11] Zhou, Neng-Fa: *B-Prolog User's Manual* [online]. Version 7.8. ©Afany Software, 1994-2012. Last updated Jan 24, 2013 [cit. 26. prosince 2013]. Dostupné z: `http://www.probp.com/download/manual.pdf`