UČEBNÍ TEXTY OSTRAVSKÉ UNIVERZITY

Přírodovědecká fakulta

PROLOG

Hashim Habiballa

Ostravská Univerzita 2003

Prolog
KIP/PROLO

distanční studijní opora

PaedDr. Mgr. Hashim Habiballa, PhD.

Prolog

Hashim Habiballa

Cíl předmětu

Seznámit studenty s principy logického programování pomocí jazyka Prolog. Ukázat způsoby tvorby logických programů a inferenční mechanismus. Probrat základní témata jazyka Prolog – historie, syntax, základní techniky, datové struktury atd.

Obsah:

Úvod pro práci s textem pro distanční studium.

Průvodce studiem:

Účel textu

Tento text má sloužit pro potřeby výuky logického programování pomocí jazyka Prolog. Má ukázat základní problémy tvorby logických programů. Jde o prakticky orientovaný kurz, který má spočívat především v programování s pomocí některé s implementací jazyka Prolog – k dispozici je stará (nicméně velmi dobře zpracovaná) implementace Turbo Prolog 2.0 od firmy Borland.
Předpokládá se, že máte znalosti na úrovni kurzu Logické základy umělé inteligence II. – tedy dobře ovládáte výrokovou a predikátovou logiku, axiomatické systémy a principy formální dedukce a především jste dobře obeznámeni s klauzulární logikou, která Vám bude velmi připomínat práci s Prologem.

Struktura

V textu jsou dodržena následující pravidla:

- je specifikován cíl lekce (tedy co by měl student po jejím absolvování umět, znát, pochopit)
- výklad učiva
- řešené příklady a úlohy k zamyšlení
- důležité pojmy – otázky
- korespondenční úkoly (mohou být sdruženy po více lekcích)

Zpracujte prosím všechny korespondenční úkoly a zašlete je včas.

Zároveň Vás chci laskavě poprosit, abyste jakékoliv věcné i formální chyby v textu zdokumentovali a kontaktovali mne. Budu Vám za tuto pomoc vděčen a usnadní to učení i Vašim kolegům.

1. Úvod do logického programování a jazyka Prolog

Cíl:

Získáte základní přehled o těchto otázkách:
- na jakých principech je jazyk Prolog založen
- jaká je základní struktura logických programů

Logické programování

Na rozdíl od běžných programovacích jazyků jako je C++ nebo Pascal pracují logické programovací jazyky na velmi vysoké úrovni. Tyto jazyky vznikaly v době rozvoje počítačů a tak se dostatečně vyvinuly na efektivní nástroje pro odvozování logických důsledků.
Prolog je jazyk pro programování symbolických výpočtů. Jeho název je odvozený ze slov **Pro**gramming in **Log**ic a vychází z principů matematické logiky. Jeho úspěch byl podnětem pro vznik nové disciplíny matematické informatiky – **logického programování**, což je perspektivní styl programování na vyšší abstraktní úrovni. Prolog je také strojovým jazykem nejmodernějších počítačů. Má doposud specifické oblasti použití - umělou inteligenci, znalostní inženýrství, atp.

1.1. Logika a Prolog

Jak již bylo napsáno v předchozím odstavci je Prolog založen především na matematické logice. Jak již víte z kurzů vztahujících se k tomuto oboru teoretické informatiky, je způsob formulace znalostí pomocí logiky stavěn na symbolické reprezentaci pomocí formulí. Setkali jste se s logikou výrokovou a predikátovou. První z nich je poměrně jednoduchá a umožňuje formulovat platné věty pomocí výroků (tvrzení o modelované realitě, která mohou pravdivá nebo nepravdivá) a logických spojek (umožňují spojovat tvrzení do složitějších formulí např. ve tvaru implikace – podmínky). Právě tyto podmínky – implikace – jsou základním stavebním kamenem programů v Prologu. Pomocí nich můžeme formulovat bázi znalostí. V Prologu je ovšem nutné respektovat jistá omezení, aby bylo možné programy efektivně vyhodnocovat.
Jistě si pamatujete na převody do normálních forem. Jejich existence umožňovala jednodušší dokazování splnitelnosti a platnosti formulí např. pomocí rezolučního pravidla. Vzniklé klauzule byly sice méně přehledné než původní zápis v obecné výrokové logice, avšak práce s nimi byla při určitých operacích jednodušší a efektivnější. Právě těchto principů bylo využito při konstrukci mechanismů Prologu. Omezení je však pro vyšší efektivitu ještě větší. V Prologu je nutné znalosti formulovat pomocí tzv. Hornových klauzulí.

Hornovy klauzule

Hornova klauzule je konečná množina literálů spojených spojkou disjunkce, kde nejvýše jeden literál je pozitivní (není negovaný):

$$\neg \mathbf{p}_1 \vee \neg \mathbf{p}_2 \vee ... \vee \neg \mathbf{p}_n \vee \mathbf{q}.$$

Tato klauzule má v implikativním zápise tvar:

$$(\mathbf{p}_1 \,\&\, \mathbf{p}_2 \,\&...\&\, \mathbf{p}_n) \rightarrow \mathbf{q}$$

Intuitivní význam takového zápisu je jednoduchý. Jde o platnost výroku **q** podmíněnou současnou platností výroků $\mathbf{p}_1$, $\mathbf{p}_2$, ..., $\mathbf{p}_n$.
Jinak řečeno logické programování spočívá ve formulaci množiny podmínek typu: Pokud platí $\mathbf{p}_1$, $\mathbf{p}_2$, ..., $\mathbf{p}_n$, pak platí také **q.** Pomocí těchto podmínek lze formulovat chování celé modelované báze znalostí. V případě, že máme formuli, která neobsahuje žádné $\mathbf{p}_1$, $\mathbf{p}_2$, ..., $\mathbf{p}_n$, jde o nepodmíněný fakt.

Řešený příklad 1:

Vezměme v úvahu následující zápis tvrzení v přirozeném jazyce.

Student složil úspěšně přijímací zkoušku, pokud udělal test z matematiky a zároveň i z informatiky.

V jazyce výrokové logiky bychom tuto bázi znalostí zapsali například následovně.

Výroky:

z – student složil úspěšně přijímací zkoušku,
m – student udělal zkoušku z matematiky,
i – student udělal zkoušku z informatiky,

$(m \,\&\, i) \rightarrow z$

V Prologu se podobná formule dá vyjádřit poněkud jiným zápisem.

z :- m, i.

Vidíte, že způsob zápisu je „opačný“. Tedy nejprve vyjádříte konsekvent (tedy co platí) a teprve za symbolem :- podmínky, které musí být splněny.

Pokud by Prolog měl pouze tyto možnosti, byl by sice schopen řešit velmi malou skupinu problémů, ale jeho pravé využití mu dává teprve obohacení o prvky predikátové logiky. Jistě si vzpomínáte, že predikátová logika je

už mnohem složitější než výroková. Umožňuje používat také predikáty, funktory a proměnné (zjednodušeně by se dalo říct, že umožňuje formulovat vztahy a vlastnosti objektů pomocí relací). Díky této vlastnosti, už nejsme odkázáni na prosté výroky, ale můžeme dosazovat proměnné a jejich objekty. Další důležitou charakteristikou predikátové logiky jsou kvantifikátory. Ty je při logickém programování bohužel nutno obětovat opět kvůli efektivitě programování v Prologu. Všechny proměnné jsou zde chápány jako univerzální.
Podívejme se nyní na podobný problém jako v příkladu 1, avšak nyní již mnohem bohatěji popsaný pomocí predikátové logiky.

Řešený příklad 2:

Vezměme v úvahu následující zápis tvrzení v přirozeném jazyce.

Student složil úspěšně přijímací zkoušku, pokud udělal test z matematiky nejméně na 200 bodů a zároveň i z informatiky nejméně na 100 bodů.

V jazyce predikátové logiky bychom tuto bázi znalostí zapsali například následovně.

Predikáty:

zk(<student>) – student složil úspěšně přijímací zkoušku,
mat(<student>,<počet bodů>) – student složil zkoušku z matematiky na počet bodů,
inf(<student>,<počet bodů>) – student složil zkoušku z informatiky na počet bodů

$\forall X ((mat(X, M) \& inf(X, I) \& M \geq 200 \& I \geq 100) \rightarrow zk(X))$

V Prologu by se tato formule dala zapsat následovně.

```
zk(X) :- mat(X, M), inf(X, I), M >= 200, I >= 100.
```

Proměnné jsou od funktorů a predikátových symbolů rozlišeny pomocí velkého písmena na začátku. Vidíte, že v Prologu se rovněž mohou používat relační operátory. Prolog obsahuje ještě mnoho dalších pomocných operátorů a predikátů, se kterými se seznámíte v tomto kurzu. Přesto je způsob zápisu, který jsme prezentovali na příkladech, klíčový a věřím, že pokud jste jej pochopili, nebude pro Vás Prolog nijak složitý k naučení.

1.2. Dedukce, formální odvozování a klauzulární logika

Umíme-li pomocí Prologu sestavit složitý systém podmínek a faktů, pak ještě stále chybí něco, co ho činí použitelným pro praxi. Samotná báze znalostí by nám jako taková příliš neposloužila, pokud bychom z ní nemohli nějakým způsobem dedukovat nové neznámé znalosti. Opět si vzpomeňte na výuku logiky. Máme-li slovní úlohu na dedukci, potřebujeme kromě zápisu předpokladů (tedy báze znalostí), také nějaký závěr, který chceme ověřit. Ten se v Prologu nazývá dotaz (u některých implementací bývá uvozen řetězcem "?-". A tou nejdůležitější věcí, kterou potřebujeme je mechanismus (metoda), kterou dotaz můžeme potvrdit. Znáte jistě celou řadu metod z výrokové i predikátové logiky. Např. tabulkovou metodu, sémantické tablo nebo rezoluci. Jak jste již poznali, rezoluce je zvlášť účinnou metodou, pokud se omezíme pouze na klauzule. Pokud se navíc omezujeme na Hornovy klauzule, je dokazování ještě efektivnější. Provádí se tak formální odvozování s pomocí pravidla, které je obdobou pravidla řezu v klauzulární logice. Jelikož však je vždy v klauzuli nejvíce jeden literál bez negace má pravidlo jednodušší tvar a máme možnost jednoduše nahradit jeho výskyt všemi podmínkami pro pravidlo se stejným literálem jako má tento negovaný literál. Odvozování je prováděno nepřímou metodou – tedy dotaz se považuje za negovaný a jádro Prologu se snaží dospět k prázdné klauzuli (množině literálů). Podívejme se na příklad.

Řešený příklad 3:

Mějme pravidlo z příkladu 2, které nám říká, za jakých podmínek složí student přijímací zkoušku.

zk(X) :- mat(X, M), inf(X, I), M >= 200, I >= 100.

Přidejme k nim následující fakta, která o studentovi Jiřím říkají, že složil zkoušku matematiky na 203 bodů a z informatiky na 152 bodů.

mat(jiří, 203). inf(jiří, 152).

Položme nyní v Prologu dotaz (zkusme prověřit), zda Jiří složil zkoušku.

Dotaz: zk(jiří).

Prolog, pak provede díky svému vnitřnímu mechanismu následující nepřímou důkazovou sekvenci.

Krok 1: Dotaz se považuje za negovaný a tedy je možné provést s pravidlem řez, pokud provedeme substituci jiří za X.

mat(jiří, M), inf(jiří, I), M >= 200, I >= 100.

Krok 2: Na výslednou formuli je možné opět aplikovat pravidlo řezu s faktem mat(jiří, 203), pokud provedeme substituci 203 za M.

inf(jiří, I), 203 >= 200, I >= 100.

Krok 3: Výsledek je možné dále zpracovat s faktem inf(jiří, 152), pokud substituujeme 152 za I.

203 >= 200, 152 >= 100.

Krok 4: Interně se zpracuje výsledek porovnání 203 >= 200 s výsledkem true.

152 >= 100.

Krok 5: Zpracuje se výsledek 152 >= 100 s výsledkem true a dostáváme prázdnou klauzuli.

Vidíte, že i inferenční jádro Prologu pracuje na podobných principech, které již znáte z teoretické výuky. Prolog tedy pro Vás doufám bude praktickou ukázkou toho, co logika dokáže a kde ji můžete využít. Věřím, že se tím pro Vás logika stane přitažlivější a že pochopíte, že i teoretické otázky informatiky mají své uplatnění v praxi.

1.3. Vztah k procedurálním jazykům

Výhody a nevýhody

Jak jste viděli na jednoduchých ukázkách prologovských klauzulí jde o velmi odlišný způsob programování, než na jaké jste zvyklí u klasických procedurálních jazyků jako je např. Pascal nebo C++. U procedurálních programovacích jazyků totiž programujete celý postup řešení a datové struktury. Při programování logickém máte k dispozici inferenční mechanismus, který „řeší“ problém podle zadání jeho objektů a vazeb mezi nimi a nemusíte se tedy starat o to, jak se bude řídit výpočet. Programujete tedy nikoliv „postup“, ale jen „logiku problému“. Má to samozřejmě své výhody i nevýhody. Výhoda spočívá v tom, že zapisujete problém ve velice jednoduchém jazyce (používá se v něm oproti procedurálním jazyků velice málo syntaktických struktur). Další výhoda je přehlednost takového programu, ve kterém skutečně vidíte, jak je problém definován. Nevýhody unifikovaného jádra řešení může být v jeho malé efektivnosti v některých případech. Je logické, že když si sami programujete postup, můžete navrhnout to nejrychlejší řešení pro daný specifický případ za cenu větší pracnosti pro vás jako programátory. Stejně je nevýhodné používat logiku s situacích, které příliš mnoho „logiky“ nevyžadují. Příkladem může být např. grafické rozhraní a obecně vstupy a výstupy programu. Jde většinou o posloupnost volání funkcí a v tomto ohledu je pak Prologovská syntaxe spíše uměle a násilně použita, aby bylo

možno tyto ryze procedurální problémy řešit i v Prologu. Proto je nutné Prolog chápat jako prostředek pro přehlednou manipulaci s daty a jejich logickými vazbami a hledání nových explicitně nedeklarovaných znalostí, nikoliv jako univerzálně lepší prostředek programování.
Stejně jako jsou například čistě objektové jazyky (Smalltalk, Beta) velice ilustrativní při výuce objektově orientovaného paradigmatu, učí správným návyků a lze jimi řešit specifické problémy systematicky a přehledně, v praxi jsou tyto jazyky využívány jen zřídka stejně jako Prolog a logické programování. Jde o to, že v praxi jsou využívány spíše nečisté objektové jazyky jako C++, které sice nejsou z hlediska objektově orientovaného paradigmatu ideální, ale díky své flexibilitě a jistým objektově nečistým prvkům umožňují programovat rychleji a efektivněji za cenu menší ztráty systematičnosti. Proto i v Prologu je snaha zavádět jisté „nečisté“ prvky do logického programování, aby se tím zvýšila jeho použitelnost. Pomocí těchto prvků pak částečně můžete ovlivňovat i samotný výpočet – inferenční proces. Prolog prostě není univerzálním prostředkem, který by vytlačil procedurální jazyky. V některých ohledech je lepším, jak už jsme dozvěděli, ale původní euforie jeho tvůrců a jejich pokračovatelů z 80. let, byla postupně ochlazena. Snahy vytvářet například na strojové úrovni logicky programovatelné (na základě logického programování) procesory se nedostala na příliš viditelné místo. Přesto se díky Prologu můžete naučit používat logiku v praxi při programování a vidět zajímavé úlohy, které řeší.

Nejdůležitější probrané pojmy:

- logické programování
- klauzule
- dedukce a dotazy

Úkoly a otázky k textu:

1. Formulujte libovolný problém v přirozeném jazyce pomocí jazyka Hornových klauzulí. Formulujte dotaz a snažte se ho dokázat nepřímo pomocí rezoluce (můžete si zvolit problém, který jste již řešili v předchozím studiu logiky).

2. Pokuste se vyjádřit způsob výpočtu funkce faktoriál procedurálně a pomocí jazyka logiky. Vyjádřete hlavní odlišnosti.

2. Turbo Prolog

Cíl:

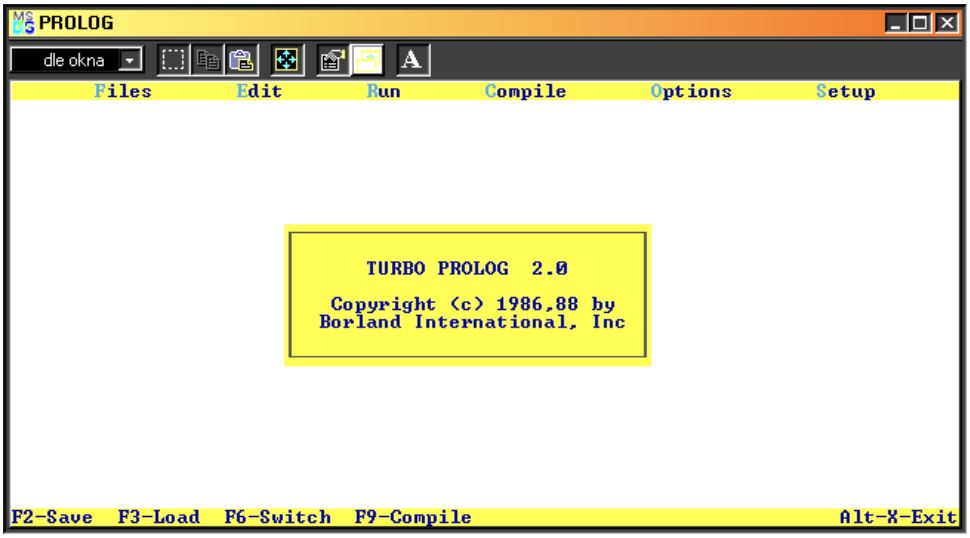
Seznámíte se:
- s konkrétní implementací Prologu, s kterou budete pracovat v rámci praktického programování – Turbo Prologem
- se základní strukturou logických programů v Turbo Prologu

Na úvod této kapitoly bych rád podotknul, že jsem velmi váhal zda ji nezařadit až jako třetí. Vám kteří máte raději hlubší teoretický úvod by to možná připadalo jako logičtější volba. Jelikož by tento kurz měl být praktický a každý informatik si asi rád co nejdříve zasedne ke klávesnici a začne programovat (doufám, že nejde jen o zbožné přání, ale stále ještě o realitu), chci vám co nejdříve ukázat konkrétní implementaci Prologu.
Možná Vám má volba této implementace nebude připadat jako nejlepší, ale věřte, že jsem zkoušel mnoho nových nástrojů založených na Prologu a stále se mi nejlépe pracuje s dnes již historickým Turbo Prologem 2.0 od firmy Borland. Jde o produkt pracující pod operačním systémem MS-DOS s minimálním nárokem na počítač PC-XT! Byl vyvinut již na konci 80. let 20. století, ale přesto si myslím, že jde o velmi dobrý výukový prostředek, který vám dá vše potřebné. V minulé kapitole jsme mluvili o tom, že používání Prologu spočívá především v práci s objekty a jejich vazbami. Právě proto je naprosto lhostejné, zda používaná implementace poskytuje elegantní grafické rozhraní či obrovské balíky speciálních funkcí. Jediné, co budeme požadovat je prostředí s editorem logických programů, inferenční jádro, ladící nástroje a především dobré příklady. A právě příkladů nabízí Turbo Prolog 2.0 obrovské množství.
V žádném případě vás ale nenutím k používání tohoto konkrétního produktu. Jak jsem již psal, je lhostejné, zda máte k dispozici vyšperkované grafické rozhraní a proto si můžete zvolit i novou implementaci Prologu dle Vaší volby. Existuje jich řada i freewarově licencovaných, stačí tedy, když si najdete informace o nich na Internetu (viz literatura).

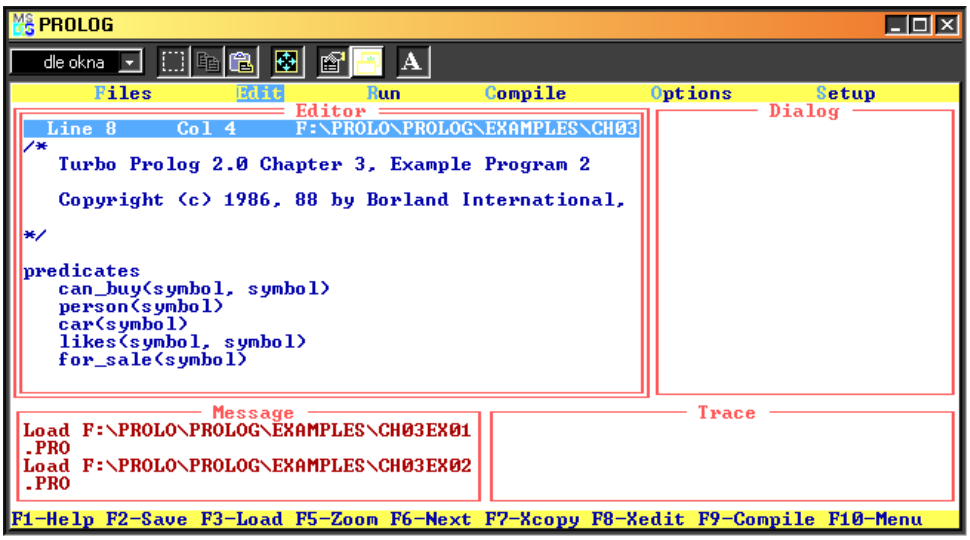
2.1. Turbo Prolog a jeho prostředí

Turbo Prolog 2.0 je dostupný pro studenty OU s elektronickou verzí tohoto textu buď ve dvou adresářích (instalačních disketách) nebo jako samorozbalovací archiv s již hotovou instalací. Jelikož jde o opravdu jednoduchý a nenáročný program, měla by Vám tato instalace po rozbalení na jakékoliv místo bez problému fungovat. Velikost celého zabaleného Turbo Prologu včetně příkladů je jen zhruba 720 kB. Tato distribuce má integrované vývojové prostředí (IDE) vzhledem i ovládáním velmi

podobné Turbo Pascalu ve verzi 5.5 a nižší (na obrázku je v inverzních barvách).



IDE tedy nedosahuje kvalit srovnatelných s Turbo Pascalem verze 6.0 a vyšší, kde například při editaci souborů bylo možné využívat standardních klávesových zkratek, ale vzhledem k jednoduchosti jazyka Prolog by mělo být možné si velmi rychle zvyknout na tyto poněkud starší způsoby práce s bloky textu.
Hlavní nabídka obsahuje standardní dialogy pro založení nového souboru, otevření existujícího apod. Po otevření existujícího souboru lze editovat soubor po výběru nabídky Edit z hlavního menu. Návrat zpět do hlavní nabídky je možný s pomocí klávesy Esc. Zdrojový kód si můžete nejlépe vyzkoušet na připravených příkladech.



Turbo Prolog obsahuje především tyto adresáře s příklady:

- Answers: obsahuje řešení kontrolních úkolů z knihy Turbo Prolog: User's guide.
- Examples: příklady po kapitolách věnované jednotlivým tématu z výše zmíněné knihy podle obtížnosti (doporučuji si je postupně procházet během výuky).
- Pie: komplexní příklad v Prologu (malý expertní systém s inferenčním mechanismem).
- Programs: soubor složitějších příkladů (např. Hanojská věž v Prologu).
- Refexams: příklady z referenčního manuálu, které jsou věnovány zejména speciálním funkcím Turbo Prologu (např. grafické prvky).

Hlavní menu Turbo Prologu

Kromě Editoru obsahuje prostředí ještě okna Message (chybové hlášky kompilace apod.), Trace (výpisy při trasování programu) a zejména Dialog. Okno Dialog umožňuje klást dotazy na bázi znalostí a sledovat odpovědi na ně.
Další nabídkou hlavního menu je Run. Ten spouští Prologovský program resp. jeho interpret. Po úspěšném spuštění programu se automaticky nastaví jako aktivní okno Dialog a můžete klást svému programu dotazy. Nabídka Compile umožňuje vytvořit z programu spustitelný soubor nezávislý na prostředí Turbo Prologu – tzn. standardní exe soubor MS-DOSu, jak je znáte z procedurálních jazyků. Nabídka Options umožňuje měnit chování interpretu – nastavovat trasování programu, velikost paměti zásobníku apod. Poslední nabídkou je Setup, který mění vzhled vývojového prostředí.
Prolog umožňuje i rozdělení programu do více souborů, jak to znáte z jazyků jako Pascal nebo C. V tom případě však dbejte na správné nastavení aktuálního adresáře nebo volby Directories v menu Setup. Týká se to především složitějších příkladů z adresáře Programs.

2.2. Zdrojové kódy

I když ještě neznáte přesnou syntaxi jazyka Prolog a jeho implementace, pokusíme se nyní podívat na jednoduché ukázky zdrojových kódů a jejich struktury.

Řešený příklad 4: **(Examples\Ch03Ex02.pro)**

```
predicates
   can_buy(symbol, symbol)
   person(symbol)
   car(symbol)
   likes(symbol, symbol)
   for_sale(symbol)

clauses
   can_buy(X, Y) :-
     person(X),
```

```
    car(Y),
    likes(X, Y),
    for_sale(Y).

person(kelly).
person(judy).

car(lemon).
car(hot_rod).

likes(kelly, hot_rod).
likes(judy, pizza).

for_sale(pizza).
for_sale(lemon).
for_sale(hot_rod).
```

Vidíme, že tento program v Prologu je rozdělen do dvou sekcí. Sekce `predicates` obsahuje definice používaných predikátů stejně jako to znáte i z predikátové logiky, kdy jsme popisovali strukturu predikátů. Zde máme pět predikátů a všechny mají jako argumenty typ symbol. Uvidíte, že existuje ještě mnohem více typů – zde je poměrně jednoduchý typ symbol – tedy identifikátor s malým písmenem na začátku. Příklad obsahuje tyto predikáty.

`can_buy` – osoba může koupit předmět
`person(symbol)` – identifikátor označuje osobu
`car(symbol)` – identifikátor označuje automobil
`likes(symbol, symbol)` – osoba má ráda předmět
`for_sale(symbol)` – předmět je na prodej

Další sekce v programu je `clauses`, která obsahuje samotné znalosti ve formě klauzulí. Vidíte podmínku, která definuje kritéria, aby předmět mohl být zakoupen osobou. Dále se definuje, kdo je osoba, co automobil, kdo má co rád a nakonec co je na prodej. Na tuto bázi znalostí se nyní můžeme dotazovat. Po spuštění můžeme v okně Dialog zadat například dotaz, zda si Judy může koupit pizzu.

Dotaz: `can_buy(judy, pizza)`

Výsledek: `No`

Dostali jsme odpověď, že Judy si nemůže koupit pizzu. Můžeme ale klást i složitější dotazy, ve kterých se vyskytují proměnné. Interpret se pak bude snažit najít všechny možnosti, které lze dosadit za proměnné.

Dotaz: `can_buy(Person, Thing)`

Výsledek:

```
Person=kelly, Thing=hot_rod
1 Solution
```

V tomto případě jsme dostali jednu možnou odpověď a to, že Kelly si může koupit automobil Hot_rod.

Nejdůležitější probrané pojmy:

- prostředí Turbo Prologu
- struktura zdrojových kódů

Úkoly a otázky k textu:

1. Seznamte se s prostředím Turbo Prologu. Otevřete si příklad – Hanojské věže v Turbo Prologu (Hanoi.pro) z adresáře Programs a projděte si jejich zdrojové kódy.

3. Syntaktické struktury jazyka Prolog

Cíl:

Naučíte se používat:

- základní syntaktické struktury jazyka Prolog
- speciální struktury Turbo Prologu

V minulé kapitole jsme si ukázali implementaci Prologu – Turbo Prolog a příklady práce s ním. Nyní si zavedeme definice syntaktických struktur. V druhé části se pak vrátíme k speciálním strukturám Turbo Prologu, se kterým budeme pracovat. Řešené příklady, které jsou převzaté z distribuce Turbo Prologu budou vždy označeny svým umístěním v adresářové struktuře.

V současnosti existuje několik dosti odlišných implementací Prologu, které se od sebe liší nejen syntaxí, ale především standardními predikáty a programovým prostředím pro editování, ladění a uchovávání programů, avšak zde budou zmíněny prostředky, které jsou pro většinu implementací společné.

V této kapitole podáme poněkud formálnější přehled syntaxe jazyka Prolog prostřednictvím jednoho typu gramatiky. Nejprve popíšeme prostředky pro popis pravidel skladby jazyka (tj. definování gramatických pravidel).

Termy a predikáty

Při určitém zjednodušení můžeme říci, že v každé úloze jde o *objekty a relace* (vazby) mezi nimi. Symbolická logika používá k označení objektů výrazy, kterým říkáme **termy**. Jménům relací říkáme **predikáty**. Termy jsou obdobou aritmetických výrazů v programovacích jazycích, kde označují hodnotu, která se má vypočítat. Predikáty jsou obdobou jmen procedur definující vztah mezi hodnotami vstupních parametrů a předávanými výstupními hodnotami.

3.1. Základní stavební prvky

Fakta, pravidla a příkazy

Program v Prologu je tvořen množinou **klauzulí**. Každá klauzule je buď **fakt, pravidlo nebo příkaz**. Prolog poskytuje uniformní datovou strukturu, zvanou **term**, pro všechna data. Term je buď proměnná nebo konstanta nebo struktura. Kromě toho se v programu může objevit **komentář** uzavřený do komentářových závorek tvořených obvykle dvojznakem /* komentář */.

Nejjednoduššími stavebními kameny výrazů v prologu jsou znaky. Ty se dělí na abecední, číslicové a speciální znaky.

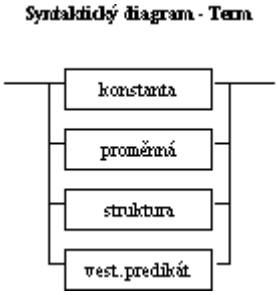
a) abecední znaky

Dnes již existují implementace Prologu, které umožňují používat znaky s diakritickými znaménky (př. Prolog 80), což se velmi nedoporučuje.

b) číslicové znaky

c) speciální znaky
Tyto znaky mohou buď samy být jménem atomu, nebo se používají při vytváření struktur. Zvláštní postavení v Prologu má symbol "_" (podtržítko).
př.: ! ’ " ( ) [ ] | , ; _

3.2. Datové typy



Z uvedených znaků se vytvářejí veškeré další syntaktické elementy jazyka Prolog. Jedná se o **konstanty**, **proměnné** a **struktury**. Souhrnně je označujeme jako **termy**.
Každá implementace jazyka Prolog obsahuje řadu tzv. **vestavěných predikátů** (procedury nabízené tou kterou implementací jazyka Prolog), které jsou ze syntaktického hlediska také považovány za termy.

Termy

Konstanta nebo proměnná se nazývá **atomický term**. Atomickými termy jsou tedy jak jednoduchá jména objektů, tak i čísla a proměnné. Z atomických termů můžeme vytvářet i složitější výrazy, které se nazývají struktury.
Konstanty jsou termy reprezentující objekty nebo relace. Rozlišujeme číselné konstanty, atomy (někdy označované nepřesně jako znakové konstanty), řetězce a speciální konstanty. Čísla a znakové konstanty se označují pojmem atomické konstanty.

a) číselné konstanty
Tyto konstanty reprezentují přirozená čísla, tj. výrazy, s nimiž lze provádět aritmetické operace. Systémy implementující programovací jazyk Prolog se dost podstatně navzájem liší druhy použitelných čísel a rozsahem jejich hodnot; všechny však pracují alespoň s celými kladnými čísly z intervalu 0 –32767.
př.: 324, 99, 1024

b) atomy
Znakové konstanty – identifikátory – začínají vždy malým písmenem a jsou tvořeny libovolnou posloupností písmen, číslic a některých speciálních znaků. Protože atom nemůže obsahovat mezeru, slouží k nahrazení mezery speciální symbol "_"

(podtržítko).
př.:petr, model321, karel_IV, první_krabice

c) řetězce
Libovolná posloupnost znaků uzavřená do apostrofů. Řetězcové konstanty mohou obsahovat libovolné znaky přípustné v Prologu (mohou začínat velkým písmenem, skládat se ze speciálních znaků nebo abecedně-číslicových znaků).
př.: 'krátký řetězec', 'Karel IV'

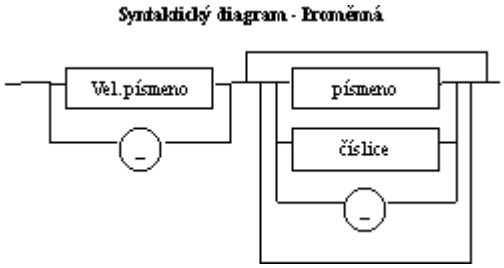
d) speciální konstanty
Jsou sestaveny ze speciálních znaků.
př.: ?-

Programy v Prologu **pracují s atomy** pojmenovanými *slovy začínajícími malými písmeny* (např. hugo, emil, manz), **aniž by „rozuměly", co pro nás tyto objekty představují**. Z tvaru programu se jen pozná, že atomy hugo a emil označují objekty (nulární predikáty) a atom manz označuje binární predikát.

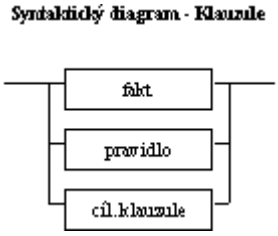
Proměnné

Proměnné jsou termy, které mohou zastupovat libovolné objekty, jež nemůžeme nebo nechceme pojmenovat přímo. Začínají velkým písmenem nebo speciálním symbolem "_" (podtržítko) a jsou tvořeny libovolnou posloupností písmen, číslic a znaku "_".
Proměnnou je také samotný znak "_". Takovou proměnnou nazýváme **anonymní proměnná**, která je obdobou zájmen cokoli nebo kdokoli. Její zvláštností je to, že každý její výskyt představuje jinou proměnnou, tedy výraz **p(_,_)** je totéž co **p(X,Y)**, nikoli **p(X,X)**. Hodnoty anonymních proměnných se v dialogu uživateli nevypisují.

V programovacím jazyce Prolog se proměnná nepoužívá stejným způsobem jako v běžných imperativních jazycích typu Pascal či Basic, zde je proměnná lokální v rámci jedné klauzule. U logické proměnné se nehovoří o hodnotě proměnné, ale o **vazbě proměnné na hodnotu**, o vázané proměnné či o instanci (instanciované) proměnné nebo instalované proměnné. Během programu může dojít ke zrušení vazby proměnné na hodnotu (u tzv. navracení) a taková proměnná (tj. „bez hodnoty") je označována jako **volná proměnná** nebo neinstanciovaná proměnná či neinstalovaná proměnná.
př.: X, _proměnná, Proměnná, POM, X_



3.3. Klauzule

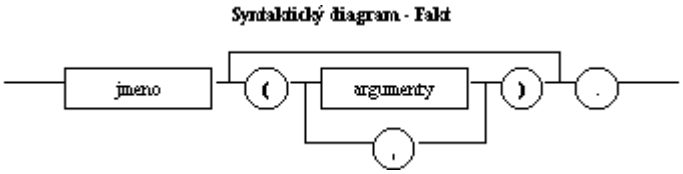


Programování v Prologu spočívá v deklarování faktů o objektech a relacích mezi objekty, v definování pravidel o objektech a relacích platících mezi nimi a v zodpovídání dotazů.
Klauzule jsou výrazy jazyka, pomocí nichž se zapisují nepodmíněné příkazy – **fakta**, podmíněné příkazy, které vyjadřují **pravidla** a výrazy, které mají tvar dotazu – **cílové klauzule**.

Základní jednotkou prologovského programu je tzv. **klauzule**, která *vždy končí tečkou*.

Nepodmíněné příkazy

Nepodmíněné příkazy vyjadřují fakta o relaci. Fakta jsou vztahy, které mají jméno, za nímž následuje výčet objektů, kterých se vztah týká. Objekty jsou odděleny čárkami a jejich výčet je uzavřen do kulatých závorek. Každý fakt je (klauzule a tedy je) ukončen tečkou.
Pokud fakt neobsahuje proměnné, hovoříme o tzv. **základním faktu.**



Řešený příklad 5:

Popsat vztahy mezi objekty můžeme pomocí čtyř predikátů:

predikát:	**význam:**
muz(X)	%X je muž
zena(X)	%X je žena
manz(X,Y)	%X a Y jsou manželé, (X manžel, Y manželka)
rodic(X,Y)	%X je rodičem Y

První dva **predikáty** mají jeden argument (jsou unární) a reprezentují tedy **vlastnosti objektů**, druhé dva mají dva argumenty (jsou binární) a vystihují tedy **vztah dvou objektů**.
Podíváme-li se na levý sloupec (program v Prologu) vidíme, že **je vlastně výčtem tvrzení** (formulovaných pomocí těchto predikátů). Ze syntaktických důvodů musí jména objektů v prologovském programu

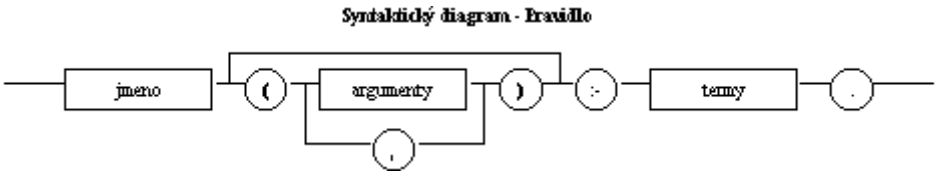
začínat malým písmenem a nesmí se v něm používat specifické znaky české abecedy, jako jsou č , ž a ě (záleží na implementaci viz. výše).
Náš program je velmi jednoduchý a obsahuje jen **nejjednodušší typ klauzulí tzv. fakta**. Fakt v prologovském programu je konstatováním, že pro nějaké objekty platí nějaké vlastnosti.

Poznámka:
Bude-li program obsahovat dvojici faktů: rodic(a,b), rodic(b,a), každý poznáme, že to není pikanterie, ale nesmysl. Jak to ale naučit Prolog? Odpověd je snadná: učit ho to vůbec nebudeme.
Rezignujeme zcela na význam predikátů, se kterými pracujeme, a budeme s nimi pracovat jen jako se symboly, které mají svoji strukturu, ne však význam, který by mohl program využít. Všechny vlastnosti, které o predikátech a objektech předpokládáme, musíme v prologovském programu specifikovat. Logika se zabývá takovými souvislostmi mezi jednotlivými výroky, které vyplývají jen z jejich struktury, ne z jejich významu. To jí umožňuje odvozovat "obecně platné" závěry.

Vyjadřování poznatků pomocí faktů, tj. nepodmíněných výrazů, nemusí být vždy efektivní. Kdybychom chtěli pomocí nepodmíněných výrazů vyjádřit rozmanitost některých poznatků, museli bychom do databáze uložit více informací. Toto lze vyřešit pomocí **podmíněných výrazů** – **pravidel**.
Zkoumáme-li fakta, můžeme na základě logických či věcných souvislostí odvodit jiné skutečnosti z nich vyplývající. Toto nám umožní odpovědět i na otázky o faktech, která nejsou explicitně obsažená v databázi.

Podmíněný příkaz

Podmíněný příkaz je oddělen znaménkem implikace "**:-**" na dvě části, ukončený tečkou. Část vlevo od dvojznaku se nazývá **hlava pravidla**, napravo od něho je **tělo pravidla**.



Řešený příklad 6:

Program, se kterým jsme dosud pracovali, obsahoval pouze **fakta**. V prologovských programech však mohou být i klauzule složitější, **tzv. pravidla**. Pravidlo vlastně definuje platnost jednoho predikátu pro nějaké objekty na základě platnosti (jiných) predikátů pro (jiné) objekty.

manzdite(On, Ona, Ono):-manz(On, Ona), rodic(On, Ono), muz(On), rodic(Ona, Ono), zena(Ona)

„Argumenty“ predikátů v předchozí klauzuli jsou proměnné, nikoli jména objektů. Je to proto, že klauzule nevyjadřuje žádné tvrzení o jednotlivých objektech, ale je vlastně definicí predikátu manželské dítě.

Poznámka:
Operátor čárka má v Prologu význam **konjunkce** - logické spojky & (a) , která vyjadřuje současnou platnost výroků, které spojuje. Prolog se snaží této současné platnosti dosáhnout postupným splňováním jednotlivých cílů odleva (s případným návratem při neúspěchu).
V Prologovských programech lze používat i operátor středník „;“, který má význam **disjunkce**.

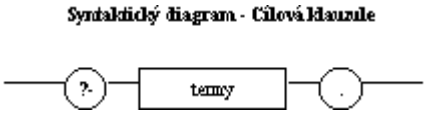
Program může obsahovat dotazy, kterým říkáme **cílové klauzule**. Cílové klauzule v textu programu nesmějí obvykle obsahovat proměnné, na něž není vázaná hodnota.

Dotaz-cílová klauzule

Otázka (cílová klauzule) začíná dvojicí znaků "?-" a končí tečkou. Na cílovou klauzuli položenou v průběhu dialogu uživatele se systémem nejsou kladena žádná omezení týkající se proměnných v ní obsažených. Dotaz, který neobsahuje proměnné se nazývá **základní otázka**.

Odpověď (řešení), kterou systém vrací, je buď „**yes**“, „**no**“, nebo **výpis hodnot proměnných** obsažených v otázce (kromě hodnot anonymních proměnných -> ty se v dialogu uživateli nevypisují). Jelikož databáze v Prologu nemůže obsahovat negativní data, chápeme odpověď „no“ dvěma způsoby, a to: záporná odpověď nebo neexistující odpověď.

V souvislosti s terminologií používající pojem cíl pak hovoříme o tom, že implementace jazyka Prolog splňuje cíl, nebo o tom, že je cíl znovu splňován. Znovusplňování je součástí procesu navracení**.**



Řešený příklad 7:

?-*manz(jan,lucie).*	% Jsou Jan a Lucie manžely?
yes.	% Ano, jsou.
?-*manz(lucie,jan).*	% Jsou Lucie a Jan manžely?
no.	% Nejsou.

V prvním případě byla odpověď ano, protože v našem programu je uložena jen informace manz(jan,lucie). O tom, že by platilo manz(lucie, jan), v něm však nic není. Jak již bylo zmíněno, Prolog nic neví o tom, že my v některých případech vztahy manz(X,Y) a manz(Y,X) považujeme za ekvivalentní.

3.4. Predikáty

To co jsme dříve nazývali procedurou můžeme přesněji popsat jako všechny klauzule začínající **stejným funktorem** a mající **shodný počet argumentů**. Mohou mít stejné jméno struktury a různý počet argumentů.

Proto se často uvádí jméno procedury spolu s počtem jejích argumentů (tj. aritou. Procedury (jinými slovy predikáty) existují **standardní** (označované jako **vestavěné predikáty**). To jsou procedury, které jsou implementovány přímo v systému. Dále jsou procedury, které si uživatel definuje sám. Program je pak tvořen z jednotlivých procedur, které se skládají z **příkazů**. Příkazy jsou **nepodmíněné (tj. fakta)** a **podmíněné (tj. pravidla)**. Jedná se jen o jiné chápání – výklad – posloupností klauzulí. Není syntaktickou chybou, pokud se v jednom programu vyskytují dva stejně pojmenované funktory s různými aritami. Tedy např. klauzule

```
ppp(X,Y,Z):- ppp(X,Y) , ppp(Y,Z).
```

je naprosto přípustná a atom ppp v ní označuje dva různé funktory (jeden binární a druhý ternární), které oba mají jméno ppp.

3.5. Speciální struktury v Turbo Prologu

Již v minulé kapitole jsme si ukázali, že program v Prologu má i speciální části. Jde především o následující sekce.

Sekce Turbo prologu

1. Domains – umožňuje definovat vlastní identifikátory pro typy používané v predikátech (lze používat obvyklé typy – integer, real, symbol, string nebo strukturované typy tvořené pomocí funktorů)
2. Predicates – definuje jména používaných funktorů a počet a typ jejich argumentů
3. Clauses – vlastní klauzule (fakta i pravidla)
4. Goals – externí dotazy (cíle); tyto cíle se nevypisují do Dialogu přímo jako dotazy interní, proto je třeba použít predikáty pro výstup

Řešený příklad 8: **(upravený Examples\Ch05Ex01.pro)**

```
domains
   title, author = symbol
   pages         = integer

predicates
   book(title, pages)
   written_by(author, title)
   long_novel(title)

clauses
   written_by(fleming, "DR NO").
   written_by(melville, "MOBY DICK").
   book("MOBY DICK", 250).
```

```
   book("DR NO", 310).
   long_novel(Title) :-
      written_by(_, Title),
      book(Title, Length),
      Length > 300.

goal
  long_novel(X),
  write(X).
```

V příkladu jsou definovány domény title, author (jméno a autor knihy) typu identifikátor. Dále pages (počet stran) typu integer (celé číslo). Predikáty definují jméno a počet stran knihy, autora a dále zda je kniha dlouhá novela. To se určí počtem stran. V sekci goal se pak pokusíme zjistit, která kniha je dlouhá novela. Odpověď je, že to je pouze kniha „DR NO“.

Další příklad ukazuje využití strukturovaných domén.

Řešený příklad 9: **(Examples\Ch06Ex01.pro)**

```
domains
   person                  = person(name,address)
   name                    = name(first,last)
   address                 = addr(street,city,state)
   street              = street(number,street_name)
   city,state,street_name = string
   first,last              = string
   number                  = integer

goal
   P1   =   person(name(jim,mos),addr(street(5,"1st
st"),igo,"CA")),
   P1 = person(name(_,mos),Address),
   P2 = person(name(jane,mos),Address),
   write(P2).
```

Prolog umožňuje vytvářet strukturované domény pomocí funktorů. Zde například existuje doména adresa, která obsahuje tři argumenty – ulici, město, stát. Ulice je dále strukturovaná.

Řešený příklad 10: **(Examples\Ch06Ex04.pro)**

```
domains
   articles            = book(title, author) ;
                         horse(name) ; boat ;
                         bankbook(balance)
```

```
   title, author, name = symbol
   balance             = real

predicates
   owns(name,articles)

clauses
   owns(john, book("A friend of the family",
"Irwin Shaw")).
   owns(john, horse(blacky)).
   owns(john, boat).
   owns(john, bankbook(1000)).
```

Domény lze deklarovat rovněž jako alternativní, jak to lze vidět u domény articles.

Nejdůležitější probrané pojmy:

- syntaktické struktury: znaky, konstanty, termy, klauzule
- hlavní sekce Turbo Prologu

Úkoly a otázky k textu:

1. Uspořádejte syntaktické struktury podle jejich hierarchie (tzn. od nejjednodušších prvků k nesložitějším). K této hierarchie vytvořte vhodné příklady.
2. Vytvořte vlastní jednoduchý program v Turbo Prologu, které vyjadřuje fakta o 6 osobách z Vašeho okolí a přidejte informace o jejich vzájemných přátelských a partnerských vazbách. Vytvořte také podmínku, která vyjádří, že partner je přítelem. Formulujte dotaz, který zobrazí všechny přátelské vztahy.

4. Srovnávání a navracení

Cíl:

Po prostudování této kapitoly si uvědomíte:
- jakým způsobem lze v Prologu provádět hledání všech řešení problému

Naučíte se:
- vytvářet jednoduché programy
- prakticky si vyzkoušíte klást dotazy na tyto programy
- používat negaci v logických programech

Aby byl Prolog schopen odpovídat na Vaše dotazy, musí mít schopnost jistým způsobem řídit svůj proces odvozování nových formulí při dedukci. Jinak řečeno musí být schopen postupně hledat všechny možnosti v logickém programu, jak váš zadaný dotaz prověřit. Těchto možností může být v jednom kroku odvození velmi mnoho a větvení těchto variant připomíná šachovou partii, která je hráčem promýšlena na několik tahů dopředu. Je jasné, že tak vzniká jakýsi stavový prostor řešení, který je exponenciálně velký. Každá možnost s sebou na další úrovni přináší mnoho nových k prověření a tento strom je nutné nějakým způsobem prohledat. Jistě znáte princip backtrackingu, kdy se po prověření jedné takové větve vracíme u úroveň výše a zkoušíme další neprověřené možnosti. Přesně takto pracuje i Prolog. Má to však ještě složitější základ, který spočívá v používání proměnných. Na každé úrovni se do proměnné dosadí dočasně hodnota, kterou zkoumáme i v celém podstromu řešení dále. Tato dočasná vazba pomine až v případě, že se backtrackingem vrátíme na úroveň, kde byla hodnota přiřazena a přiřadíme ji další možnou hodnotu.

4.1. Princip srovnávání a negace

Srovnávání a negace

Prolog se při hledání řešení pokouší srovnávat hledaný cíl s nějakou informací, která je obsažena v logickém programu. Samozřejmě, že ale existují i situace, kdy chceme formulovat negativní výskyt. To znamená, že chceme říct, že klauzule platí pouze pokud se některá informace v programu nevyskytuje. Jinak řečeno hledáme **negaci** určitého faktu, neboť se pohybujeme pouze ve dvouhodnotové logice. Cokoliv tedy v programu není řečeno, považuje se za neplatné. Tento postup znáte již z logiky jako princip uzavřeného světa. To lze v Prologu implementovat pomocí vestavěného predikátu **not**.

Podívejme se na následující příklad:

Řešený příklad 11: (Examples\Ch05Ex11.pro)

```
predicates
   likes_shopping(symbol)
   has_credit_card(symbol, symbol)
   bottomed_out(symbol, symbol)

clauses
   likes_shopping(Who) :-
      has_credit_card(Who, Card),
      not (bottomed_out(Who, Card) ) ,
      write(Who, " can shop with the ",Card, "
credit card.\n").

   has_credit_card(chris, visa).
   has_credit_card(chris, diners).
   has_credit_card(joe, shell).
   has_credit_card(sam, mastercard).
   has_credit_card(sam, citibank).

   bottomed_out(chris, diners).
   bottomed_out(sam, mastercard).
   bottomed_out(chris, visa).
```

V tomto příkladu se pokoušíme definovat, kdo má rád nakupování a s jakou kartou může nakupovat. Takový člověk jednak musí vlastnit kreditní kartu a navíc tato karta nesmí být prázdná. Pokud položíme Prologu dotaz začne vlastní srovnávání.

Dotaz: `likes_shopping(sam).`

Krok 1: Prolog se pokouší v programu najít srovnání s predikátem `likes_shopping` a nachází ho u podmínky. Aby však mohl úspěšně srovnat, musí provést ještě substituci sam za proměnnou Who. Potom lze odvodit novou sekvenci:

```
has_credit_card(sam, Card),
      not (bottomed_out(sam, Card) ) ,
      write(sam, " can shop with the ",Card, "
credit card.\n").
```

Krok 2.1: Nyní se interpret pokouší hledat srovnání s predikátem `has_credit_card`. Nachází první možnost a to je srovnání při substituci mastercard za Card. V tom případě lze opět provést odvození na:

```
      not (bottomed_out(sam, mastercard) ) ,
```

```
      write(sam, " can shop with the ",mastercard,
" credit card.\n").
```

Krok 2.1.1: Nyní se snažíme zjistit, zda karta není prázdná. Pomocí predikátu not srovnáváme predikát `bottomed_out`. Zjišťujeme, že karta prázdná je a tím pádem je srovnání negované neúspěšné a musíme se tedy vrátit o úroveň výše bez možnosti odvození.

Krok 2.2: Druhou možností je dosazení citibank za Card a odvození:

```
      not (bottomed_out(sam, citibank) ) ,
      write(sam, " can shop with the ",citibank, "
credit card.\n").
```

Krok 2.2.1: Nyní opět zkoušíme srovnání s predikátem not na prázdnou kartu. V tomto případě se nepodaří srovnat požadovanou informaci a tím pádem je výsledek s not úspěšný a můžeme pokračovat na další úroveň, kde už pouze predikát Turbo Prologu write provede výpis do dialogového okna.

4.2. Vazby proměnné

V minulém příkladě jsme se dotazovali na konkrétní konstantu – sam. Ale co stane, pokud bychom v dotazu použili proměnnou a chtěli tak získat informace o všech osobách, které mají rády nakupování? Pak Prolog musí hledat způsob, jak postupně za proměnnou dosazovat všechny termy, které se při srovnávání dají dosadit.
Mluvíme pak o vazbě proměnné na hodnotu, která je dočasná pouze po dobu, kdy se snažíme dospět k jednomu popření při nepřímém důkazu. V momentě, kdy se vrátíme až na úroveň, která vazbu způsobila, provedeme vazbu na další hodnotu. To děláme až nelze další hodnotu dosadit, aby přitom došlo k novému srovnání.
Podívejme se opět na příklad, ve kterém využijeme již deklarovaný program.

Řešený příklad 12: (Examples\Ch05Ex11.pro)

Dotaz: `likes_shopping(X).`

Krok 1: Prolog se pokouší v programu najít srovnání s predikátem `likes_shopping` a nachází ho u podmínky. Aby však mohl úspěšně srovnat, musí provést ještě vazbu X s proměnnou Who. Potom lze odvodit novou sekvenci:

```
has_credit_card(X, Card),
      not (bottomed_out(X, Card) ) ,
```

```
      write(X, " can shop with the ",Card, "
credit card.\n").
```

Krok 2.1: Nyní se interpret pokouší hledat srovnání s predikátem `has_credit_card`. Nachází první možnost a to je srovnání při substituci chris za X a visa za Card. V tom případě lze opět provést odvození na:

```
      not (bottomed_out(chris, visa) ) ,
      write(chris, " can shop with the ",visa, "
credit card.\n").
```

Krok 2.1.1: Nyní se snažíme zjistit, zda karta není prázdná. Pomocí predikátu not srovnáváme predikát `bottomed_out`. Zjišťujeme, že karta prázdná je a tím pádem je srovnání negované neúspěšné a musíme se tedy vrátit o úroveň výše bez možnosti odvození.

Krok 2.2: Druhou možností je dosazení opět chris za X a diners za Card a odvození:

```
      not (bottomed_out(chris, diners) ) ,
      write(chris, " can shop with the ",diners, "
credit card.\n").
```

Krok 2.2.1: Nyní opět zkoušíme srovnání s predikátem not na prázdnou kartu. Zjišťujeme, že karta prázdná je a tím pádem je srovnání negované neúspěšné a musíme se tedy vrátit o úroveň výše bez možnosti odvození.

Krok 2.3+: Třetí možnost je kombinace joe za X a shell za Card, která nás již postupně dovede až k výpisu…

Krok 2.4+: Po návratu zkoušíme další možnost sam za X a mastercard za Card (viz výše).

Krok 2.5+: Poslední možností je již také prozkoumaná úspěšná možnost sam za X a citibank za Card.

Vidíte, že mechanismus srovnávání a navracení je vcelku přirozený – jde o hledání všech možných kombinací srovnáváním predikátů a vázáním proměnných.

4.3. Další praktické úlohy

Pomocí vazeb proměnných lze realizovat další úlohy, který by Vám v procedurálním programování připadaly jako triviální. Podívejme se na následující příklad, kde vytváříme predikáty pro sčítání a násobení.

Řešený příklad 13: **(Examples\Ch04Ex01.pro)**

```
domains
   product, sum = integer

predicates
   add_em_up(sum, sum, sum)
   multiply_em(product, product, product)

clauses
   add_em_up(X, Y, Sum) :- Sum = X + Y.
   multiply_em(X, Y, Product) :- Product = X * Y.
```

Dotaz: `multiply_em(11,7,Result)`

```
Result=77
1 Solution
```

V následujícím příkladě testujeme, jaká jídla má rád Bill na základě toho, zda dobře chutná nebo ne.

Řešený příklad 14: **(Examples\Ch05Ex02.pro)**

```
predicates
   likes(symbol,symbol)
   tastes(symbol,symbol)
   food(symbol)

clauses
   likes(bill, X) :-
      food(X), tastes(X, good).

   tastes(pizza, good).
   tastes(brussels_sprouts, bad).

   food(brussels_sprouts).
   food(pizza).
```

Dotaz: `likes(bill,X)`

```
X=pizza
1 Solution
```

Nejdůležitější probrané pojmy:
- princip srovnávání a negace
- vazby proměnných

Kontrolní otázka:

Jak spolu souvisí negace v Prologu a princip uzavřeného světa, který znáte z logiky?

Řešení:

Princip uzavřeného světa umožňuje v Prologu deklarovat negované klauzule. To, co nelze dokázat, se považuje za nesplněné.

Úkol k textu:

Napište program v Prolog, který obsahuje predikát pro výpočet druhé mocniny.

5. Rekurze a řízení výpočtu

Cíl:

Po prostudování této kapitoly si uvědomíte:

- význam prvků Prologu pro řízení výpočtu

Naučíte se:
- používat rekurzi v logických programech
- používat řídící predikáty – řez, fail a další

5.1. Rekurze

O rekurzi hovoříme tehdy, když se v těle některé z klauzulí, tvořící definici jistého predikátu, vyskytuje atomická formule téhož predikátu. Jinými slovy, „predikát se ve své definici odvolává na sebe sama". Rekurze je v Prologu jednou z nejpoužívanějších programátorských technik. Její předností je jednak (zpravidla) dobrá deklarativní sémantika, jednak relativně rychlé zpracování odvozovacím mechanismem. Zde si ukážeme její použití v rámci databáze příbuzenských vztahů.

Naším cílem bude přidat do rodokmenové databáze definici predikátu předek, který bude vyjadřovat, že jedna osoba je předkem druhé osoby. První pokus o tuto definici může vypadat např. takto:

```
predek(X.Y)   :- rodic(X.Y).
predek(X.Y)   :- predek(X,Z),rodic(Z,Y).
```

Definice vyjadřuje, že X je předkem Y pokud je jeho rodičem, a dále tehdy, jestliže je *předkem* nějakého Z, které je teprve rodičem dotyčného Y. Je zjevné, že tato definice skutečně pokrývá všechny možnosti, jak být něčím předkem, protože můžeme nejprve libovolněkrát aplikovat druhou klauzuli, a teprve nakonec klauzuli první, která celý řetěz rodičovských vztahů uzavře. Lze říci, že definice libovolných rekurzivních predikátů se skládají z jedné (nebo více) klauzulí rekurzivních, a z jedné (nebo více) klauzulí nerekurzivních - „uzavíracích".

Zkusme ověřit fungování definice v Prologu. Přidejte si do logického programu vlastní informace o rodičích (např. ve vaší vlastní rodině). Pro základní dotazy, např.

```
predek(karel, vaclav).
predek(jan, vaclav).
```

poskytne spolehlivě správnou odpověď. Horší situace ovšem nastane, když použijeme dotaz s proměnnou, a naformulujeme ho takto:

?- predek(X,vaclav).

Vcelku přirozený požadavek na nalezení všech předků Václava povede sice k jejich vypsání:
X = …, X=…,…

Prolog s námi ovšem na čas přestane komunikovat, a po delší či menší době (v závislosti na implementaci a výkonu počítače) vypíše chybové hlášení - o přetečení zásobníku (angl. „stack overflow").

Nekonečný cyklus

Zkušení programátoři již tuší, že věc má co dočinění s *nekonečným cyklem* - druhá, rekurzivní klauzule již neměla možnost „zavolat" klauzuli první, nerekurzivní, a tak volala „donekonečna" sebe sama. Příčinou bylo nevhodné *uspořádání cílů v* těle rekurzivní klauzule, v důsledku kterého se cíl predek(X,Z) volal s oběma proměnnými volnými. Když cíle přehodíme:

predek(X,Y) :- rodic(X.Y).
predek(X.Y) :- rodic(Z.Y),predek(X,Z).

nekonečný cyklus již nemůže nastat, protože k volání druhého, rekurzivního cíle dojde vždy jen poté, co se první cíl rodic(Z,vaclav) unifikuje s některou základní klauzulí z databáze; přitom se proměnná Z konkretizuje hodnotou, která se předá do dalšího kroku rekurze. To platí i v každém následujícím kroku, tak dlouho, až nastane situace, že se první cíl (s ohledem na konečný počet klauzulí predikátu rodič) nepodaří splnit.

Uvědomme si ovšem závažnou skutečnost, že z hlediska predikátové logiky jsou obě definice predikátu předek ekvivalentní, přesto jedna z nich vede na nekorektní výpočet a druhá nikoliv. Při psaní programů tedy musíme chtě nechtě brát ohled i na procedurální aspekt práce Prologu.

5.2. Nepřímá rekurze

Ve výše uvedeném příkladě byla rekurze na první pohled patrná - v těle klauzule se vyskytoval tentýž predikátový symbol, jako v hlavě. O rekurzi ovšem můžeme hovořit i v případě, kdy je *autoreference* (odkazování na sebe sama) predikátu zprostředkována jiným predikátem. Představme si kupříkladu, že budeme chtít do relace strýc zahrnout nejen pokrevní, ale i „přiženěné" strýce. Intuitivní Prologovská definice by pak mohla vypadat třeba takto:

stryc(X.Y) :- bratr(X,Z), rodic(Z,Y).
stryc(X,Y) :- teta(Z,Y), manzelka(Z,X).

Pokud bychom ovšem obdobným způsobem definovali relaci teta

teta(X.Y) :- sestra(X,Z), rodic(Z,Y).
teta(X.Y) :- stryc(Z,Y), manželka(X,Z).

měli bychom tu opět rekurzi, byť nepřímou, a opět by hrozil nekonečný cyklus - tentokrát by se v něm střídavě navzájem volaly druhá klauzule obou predikátů. Nepřímá rekurze je o to problematičtější, že nemusí být v programu na první pohled patrná.

Další příklady ilustrují správná a nesprávná použití rekurze.

Řešený příklad 15: (Examples\Ch07Ex05.pro)

Následující predikáty badcount mají realizovat jednoduché počítadlo. Jejich deklarace však má svá úskalí, které zde rozebereme. Jde především o to, že skončí s přetečením zásobníku po několika stovkách operací.

```
predicates
   badcount1(real)
   badcount2(real)
   badcount3(real)
   check(real)

clauses
/* badcount1:
   Problém spočívá v tom, že badcount1 není
posledním cílem v podmínce. To způsobí přetečení
zásobníku, který se bude plnit informacemi pro
navracení za jeho volání*/

   badcount1(X) :-
      write(X), nl,
      NewX = X+1,
      badcount1(NewX),
      nl.

/* badcount2:
   Problém této verze je v tom, že za první
klauzulí existuje ještě další, která nebyla
vyzkoušena. To opět způsobuje nutnost zachovat
návratové informace. */

   badcount2(X) :-
      write(X), nl,
      NewX = X+1,
      badcount2(NewX).
```

```
   badcount2(X) :-
      X < 0,
      write("X is negative.").

/* badcount3:
   Před voláním rekurzivní procedury existuje
predikát, který musí být splněn. To opět vyžaduje
návratovou informaci*/

   badcount3(X) :-
      write(X), nl,
      NewX = X+1,
      check(NewX),
      badcount3(NewX).

   check(Z) :- Z >= 0.
   check(Z) :- Z < 0.
```

Řešený příklad 16: (Examples\Ch07Ex04.pro)

V následujícím programu je již rekurze implementována správně a nedochází k zahlcení zásobníku.

```
predicates
   count(real)
clauses
   count(N) :-
      write(N), nl,
      NewN = N+1,
      count(NewN).

goal
   count(1).
```

5.3. Rekurzivní cyklus s aritmetickým výpočtem

Podstatou rekurzivního cyklu s numerickým výpočtem je „trik", kdy je nově zavedená proměnná (zde W) konkretizována hodnotou proměnné, která je výstupním argumentem rekurzivního cíle , změněnou o určitou hodnotu (např. „krok cyklu" - zde 1). Obsah nové proměnné je pak vrácen opět jako výstupní argument pro vyšší úroveň rekurze.

Jednodušším příkladem rekurzivního cyklu s výpočtem je program pro výpis zadaného počtu hvězdiček:

hvězd(0).
hvezd(N) :- write("*"), NI is N-l, hvezd(Nl).
Oblíbenou ukázkou rekurzivního cyklu je také definice funkce faktoriál, kterou lze zapsat takto:

faktorial(l, l).
faktorial(N, F) :- NI is N-l, faktoriál(NI,Fl), F is F1*N.

Definice má deklarativní sémantiku: faktoriálem l je l, a faktoriálem libovolného většího čísla je součin tohoto čísla a faktoriálu čísla o l menšího. V jednom kroku rekurze zde máme dokonce dva numerické výpočty - zmenšení daného čísla o l, a vynásobení získaného faktoriálu tímto číslem.
Definice má ovšem dva nedostatky. Prvním je, že se Prolog při případné žádosti o alternativní řešení zacyklí - aplikuje druhou klauzuli i na N=l, a posléze na nulu a na stále menší záporná čísla, přičemž už neexistuje nerekurzivní klauzule, která by výpočet „uzavřela". Tento problém lze vyřešit s pomocí *řezu*.

5.4. Řez

Řez je jedním z prostředků jak ovlivňovat výpočet při zpracování dotazu. Umožňuje „odříznout" varianty při navracení, které již nechceme procházet. Než řez definujeme přesně, podívejme se na následující příklad, jak lze v Turbo Prologu realizovat funkci faktoriál bez nekonečného cyklu pomocí řezu.

Řešený příklad 17:

```
predicates
   factorial(integer, real)

clauses
   factorial(1, 1) :- !.

   factorial(X, FactX) :-
      Y = X-1,
      factorial(Y, FactY),
      FactX = X*FactY.
```

Řezem je v tomto případě první klauzule, která obsahuje místo podmínek symbol „!", který je právě aplikací řezu. Pokud by na tomto místě řez nebyl, pak by výpočet faktoriálu pokračoval po zjištění hodnoty pro 1 dále a neustále by snižoval argument do záporných hodnot. Řez zajistí, že po vyčíslení hodnoty pro faktoriál 1, se již dále nesrovnává predikát factorial a dojde k postupnému návratu z rekurze.

Řez

Nechť **R** je logický program, ve kterém je použit řez.
Zásobník se vytváří podobně jako u čistého Prologu s **doplněním právě pro řez**:
Zatímco klasické dotazy jsou v závorce { } doplněny o číselný parametr, který odpovídá číslu další klauzule programu **R**, pro řez je doplněn údaj **a** označující číslo dotazu v zásobníku, kam je nutné se vrátit při zpětném chodu přes řez (t.j. dotazy s čísly vyššími než **a** jsou pak ze zásobníku odebrány = vymazány).

Nechť vrchní prvek zásobníku (jedná se o právě řešený dotaz) je označen číslem **d** a má tvar **?- Q1{k1},Q2{k2}, ... , Qm{km}.**.

Pokud Q1 není řez, zjisti, zda existuje v programu **R** nejaká klauzule s číslem vyšším než **k1**, kterou lze použít pro řešení cíle **Q1**:
Nechť taková klauzule existuje, má číslo **j** a výsledkem jejího použití na Q1 je R1,**!**,...,Rp. V takovém případě proveď následující 2 kroky:
uprav odkaz v právě zpracovávaném dotazu takto
?- Q1**{ j },**Q2{k2}, ... , Qm{km}.
a do zásobníku přidej nový dotaz
?- R1{0},!**{ d-1 },**...,Rp{0}, Q2{k2},...,Qm{km}.

Nechť Q1 je řez, t.j. jedná se o dotaz ?- **! {k1}**,Q2{k2}, ... , Qm{km}.
Pokud **k1** je číslo **i**, pak proveď následující 2 kroky:
uprav právě zpracovávaný dotaz takto
?- !**{jump(i) },**Q2{k2}, ... , Qm{km}.
a do zásobníku přidej nový dotaz
?- Q2{k2},...,Qm{km}.

Má-li **k1** tvar **jump(i)**, vymaž v zásobníku všechny dotazy s číslem vyšším než je **i** a pokračuj v řešení s takto upraveným zásobníkem.

Řez - označovaný **! - je cíl, který vždy uspěje při přímém chodu**. Ovšem při požadavku na jeho alternativní splnění (při zpětném chodu) NIKDY NEUSPĚJE řez sám ani cíl v hlavě toho pravidla, díky kterému se řez stal součástí právě řešeného dotazu: **řez fixuje zvolenou větev**!

Názorné vysvětlení funkce řezu

Mějme program **P** obsahující následující 3 klauzule
b(X,Y):- e(X,Y), f(Y), !, g, h.
b(X,Y):- k(X,Y).
p(X):- a(X), b(X,Y), c(Y), d.

Řez fixuje přijaté „částečné řešení“ – omezuje splnění podcílů vlevo od řezu (e & f) na jedinou možnost.

Překročení řezu zamezí využití ostatních pravidel. Uvažme např., že platí (e & f). V takovém případě se při dotazu
?- b.
nikdy nedostaneme ke zjišťování, zda platí k.

Řez neovlivňuje zpětný chod vpravo do něj, t.j. mezi cíli g a h.

Pokud cíl, ve kterém se vyskytuje ! neuspěje, zpětný chod se vrací k jeho předchůdci:
?- p(X).
?- a(X), b(X,Y), c(Y), d. % nechť platí a(o1), kde o1 je konstanta
?- b(o1,Y), c(Y), d.
?- e(o1,Y), f(Y) , !, g, h, c(Y), d. % nechť platí e(o1,v1), f(v1) a neplatí g.
% V tomto okamžiku se již nehledá nová možnost splnit e(o1,Y) ani b(o1,Y)
% Zpětný chod začíná od a(X) – hledá se další vhodná substituce za X (jiná
% než konstanta o1

Příklady a záludnosti použití řezu

databáze údajů o studentech

domaci (‘Jan Antoš’). **….**

zahranicni(‘Sumi Akimoto’).
zahranicni(‘Hans Cozreck’).
zahranicni(‘Kim Nowak’). **….**

jazykovy_test (‘Sumi Akimoto’, 60).
jazykovy_test (‘Hans Cozreck’, 65).
jazykovy_test (‘Kim Nowak’, 75). **….**

test_odborny (‘Jan Antoš’, 85). **….**

prijmout(X):- zahranicni(X), jazykovy_test(X, ZnamkaX), ZnamkaX >= 50.
Výčet všech kandidátů na přijetí

?- prijmout(X).

X = ‘Sumi Akimoto’; ‘Hans Cozreck’; ‘Kim Nowak’; …

jazykovy_test
prohledává se zbytečně, ačkoliv pro každého studenta je zaznamenána jediná známka

prijmout1(X):- zahranicni(X), jazykovy_test(X, ZnamkaX),!, ZnamkaX >= 50.

?- prijmout1(X).
X = ‘Sumi Akimoto’; **no**

jazykovy_test1(‘Sumi Akimoto’, 60) :- !.
jazykovy_test1(‘Hans Cozreck’, 65) :- !.
jazykovy_test1(‘Kim Nowak’, 75) :- !.

prijmout2(X):- zahranicni(X), jazykovy_test1(X, ZnamkaX), ZnamkaX >= 50.

?- prijmout2(X).
X = ‘Sumi Akimoto’; ‘Hans Cozreck’; ‘Kim Nowak’; …

5.5. Další řídící predikáty

Predikát fail

Dalším důležitým řídícím predikátem je **fail**. Z jeho názvu vyplývá, že má simulovat selhání při splňování cíle. Tato konstrukce může být účinná především při používání vestavěných predikátů, kdy chceme, aby byly hledány další alternativy. Podívejme se na příklad.

Řešený příklad 18: (Examples\Ch05Ex06.pro)

```
domains
   name = symbol

predicates
   father(name, name)
   everybody

clauses
   father(leonard, katherine).
   father(carl, jason).
   father(carl, marilyn).
   everybody :-
      father(X, Y),
```

```
        write(X, " is ", Y, "'s father\n"),
        fail.
```

Pokud by program neobsahoval predikát fail na konci podmínky, pak by se již nehledaly další možnosti a výpis by obsahoval pouze informaci, že Leonard je otec Katherine.

Podobnou funkci má také predikát true, který naopak vede vždy ke splnění.

Nejdůležitější probrané pojmy:

- rekurze, nepřímá rekurze
- rekurzivní cyklus
- řez, predikát fail, true

Korespondenční úkol:

Vytvořte program a predikát pro výpočet Fibonnaciho čísel.

6. Seznamy a stromy

Cíl:

Po prostudování této kapitoly pochopíte:
- jak lze v Prologu implementovat seznamy a stromy

Naučíte se:
- implementovat základní algoritmy na seznamech a stromech

6.1. Seznam

Seznam je základním datovým prvkem jazyka PROLOG. Je typickou rekurzivně definovanou datovou strukturou. Patří do skupiny vestavěných predikátů a operátorů. Je to uspořádaná konečná množina prvků, pomocí kterých se dají vyjádřit stromy, grafy a pole. Seznam může být prvkem jiného seznamu (tzn. že se vnoří do jiného seznamu) a tak umožňuje reprezentovat libovolné symbolické struktury. Vybírání nebo přidávání prvků do seznamu definuje operace se seznamy.

Seznam

Zápis seznamu

Zápis je tvořen výčtem všech jeho prvků, které jsou od sebe odděleny čárkou a jsou uzavřeny v hranatých závorkách **[,]**. Nejjednodušší seznam je prázdný seznam a označujeme ho **[]**, složitější seznamy vznikají přidáváním dalších prvků do seznamu. Seznam může obsahovat konstanty **[1,2,3]**, další seznamy **[1, [2,3],4]** nebo proměnné **[A,B,C]**.

Zřetězení seznamu

Pro tento způsob zápisu je nutné znát délku seznamu (tzn. počet jeho prvků).
V tomto případě bude seznam výsledkem operace „*zřetězení*“:
a) prvku – tento prvek bude v novém seznamu na prvním místě a nazveme ho **hlava seznamu**
b) a dalšího seznamu – nazveme ho **tělo seznamu**.Takovýto seznam bude mít zápis **[Hlava|Tělo]**, kde “|“ je oddělovač. Je možné zapsat zřetězený seznam i následujícím způsobem: vlevo od symbolu “|“ může být zapsáno více prvků oddělených čárkou, ale vpravo od symbolu “|“ musí být zapsán pouze jediný prvek (tělo seznamu).

Příklad: Uvedeme 3 odlišné zápisy, které jsou v logickém slova smyslu naprosto totožné.
[pavla,katka,monika,katka], **[pavla|[katka,monika,katka]]**,

[pavla,katka|[monika,katka]].
Tento výraz **[pavla,katka|monika],** není seznamem, protože vpravo od operátoru "|" musí být seznam (tzn. výraz uzavřený v závorkách **[,]**).

Vybírání prvku ze seznamu

K vybírání určitého prvku ze seznamu můžeme použít **unifikaci**.
Příklad: Vybereme 3. prvek ze seznamu **[pavla,katka,monika,katka]**. Výsledný seznam bude vypadat takto: **[pavla,katka,monika|[katka]]**. 3.prvek seznamu, získáme jako výsledek substituce za proměnnou **C** při jeho unifikaci s výrazem **[A,B,C|D]**.

Program "Seznam"

Budeme rozpoznávat nebo vytvářet seznamy složené pouze z číslic 1,2 a 3.

Cislo(1). Cislo(2). Cislo(3).
C_seznam([]).
C_seznam([A|B]) :- cislo(A) , C_seznam(B)
a) zda má seznam požadovaný tvar, např. odpověď na dotaz **?- C_seznam([3,2,3])** bude kladná (tzn. yes).

b) umožňuje generovat seznamy předem stanovené délky (např. 3) dotazem **?- C_seznam([X,Y,Z])**. Prolog vygeneruje všech 27 řešení v pořadí [1,1,1], [1,1,2], [1,1,3], [1,2,2], [1,2,3],…, [3,3,3]. Žádáme-li další řešení po poslední odpovědi [3,3,3], dostaneme odpověď zápornou (tzn. no), protože v prostoru všech rezolvent nezbyla žádná neprozkoumaná cesta.

c) umožňuje generovat seznamy s pevně specifikovaným prvkem na vybraném místě zleva, např. seznamy s číslicí 3 na druhém místě s dotazem **?- C_seznam([X,3|B])**. Prolog najde tato řešení:
X=1, B=[]
X=1, B=[1]
X=1, B=[1,1]
X=1, B=[1,1,1] …
V tomto případě se hledání alternativ zastaví až po vyčerpání paměti (nova řešení jsou vlastně stále delší seznamy číslic 1, k ostatním seznamům se nikdy nedostaneme. To je důsledkem použité strategie prohledávání. Se stejným problémem se můžeme setkat i při zkoumání dotazu **?- C_seznam(M)**, kdy Prolog generuje tato řešení M=[1], M=[1,1], M=[1,1,1] …

Následující program realizuje výpis prvků seznamu čísel na obrazovku.

Řešený příklad 19: **(Examples\Ch08Ex01.pro)**

```
domains
   list = integer*

predicates
   write_a_list(list)

clauses
   write_a_list([]).
   write_a_list([H|T]) :-
      write(H), nl,
      write_a_list(T).

goal
   write_a_list([1, 2, 3]).
```

V příkladu se výpis seznamu čísel realizuje rekurzivní klauzulí, která vždy vypisuje hlavu seznamu a pak se rekurzivně zavolá na zbytek seznamu (tělo). Bylo by velmi jednoduché změnou domény vytvořit takový seznam a jeho výpis pro libovolný jiný typ Prologu.

Řešený příklad 20: **(Examples\Ch08Ex02.pro)**

```
domains
   list = integer*
predicates
   length_of(list, integer)

clauses
   length_of([], 0).
   length_of([_|T], L):- length_of(T, TailLength),
                          L = TailLength + 1.
```

Příklad umožňuje spočítat počet prvků seznamu opět pomocí rekurze, kdy za každý nalezený prvek inkrementujeme hodnotu proměnné (pomocí vazby).

Řešený příklad 21: **(Examples\Ch08Ex05.pro)**

```
domains
   list = integer*

predicates
   discard_negatives(list, list)

clauses
   discard_negatives([], []).
```

```
discard_negatives([H|T], ProcessedTail) :-
   H < 0,
   !,
   discard_negatives(T, ProcessedTail).

discard_negatives([H|T], [H|ProcessedTail]) :-
   discard_negatives(T, ProcessedTail).
```

Tento program provádí úpravu seznamu, který je prvním argumentem predikátu a výsledek je ukládán do druhého argumentu. Program odstraní ze seznamu celých čísel záporné prvky. Pokud je číslo, které je aktuálně hlavou zbytku seznamu, pak ho ignoruje a do výsledku uloží jen tělo zbytku.

Řešený příklad 22: (Examples\Ch08Ex06.pro)

```
domains
   namelist = name*
   name = symbol

predicates
   is_a_member_of(name, namelist)

clauses
   is_a_member_of(Name, [Name|_]).
   is_a_member_of(Name,[_|Tail]):-
is_a_member_of(Name, Tail).
```

V příkladu deklarujeme predikát, který pomocí rekurze zjistí, zda se v seznamu vyskytuje prvek, který je prvním argumentem.

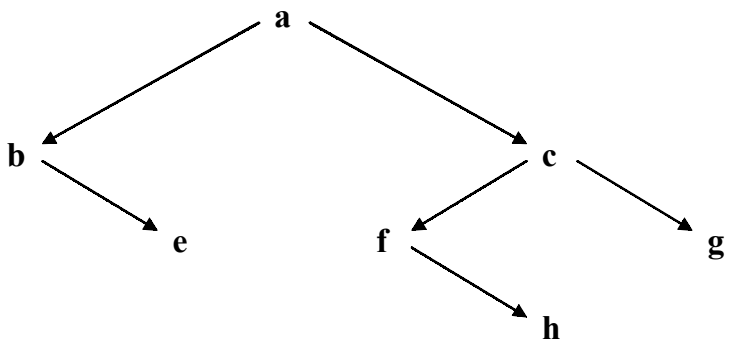
6.2. Stromy

Stromy

Rekurzivní charakter jazyka a to, že "počítá přímo s termy", dělá z Prologu velmi vhodný nástroj pro práci s datovými strukturami, které jsou definovány rekurzivně. Nejdůležitějším případem takových struktur jsou **binární stromy**. Můžeme je reprezentovat například takto:

Binárním stromem rozumíme buď atom **nil** reprezentující prázdný strom nebo strukturu **t(L,V,R)**, kde **V** je vrchol stromu, **L** je levý podstrom a **R** je pravý podstrom , reprezentující binární strom

Následujícímu stromu



odpovídá prologovský term:

```
t(t(nil, b, t(nil, e, nil)),
% levý podstrom
a,
% vrchol
t(t(nil, f, t(nil, h, nil)), c, t(nil, g, nil)) )
% pravý podstrom
```

Abychom ho nemuseli opakovaně zadávat z klávesnice, můžeme si ho "uložit do programu" klauzulí.

```
nas_strom(t(t(nil, b, t(nil, e, nil)),
          a,
          t(t(nil, f, t(nil, h, nil)), c, t(nil, g,
      nil))))
```

Velmi často potřebujeme "projít" stromem, abychom buď provedli pro každý prvek obsažený ve stromě nějakou akci nebo abychom převedli prvky stromu v nějakém specifickém pořadí do seznamu či na výstup.

Nejprve sestrojíme jednu proceduru, která prochází strom do hloubky.

```
%
============================================================
% preorder(+Strom,-Seznam) průchod stromem do hloubky
%
============================================================

preorder(t(L, V, R), [V|S]):-
preorder(L, SL),
preorder(R, RL),
conc(SL, RL, S).
preorder(nil, []).
```

Na dotaz

?- nas_strom(T), preorder(T, L).

pak dostaneme odpověď

T = ...
L = [a,b,e,c,f,h,g] ,

tedy skutečně průchod stromu do hloubky.

Na první pohled se zdá, že naprogramování procházení stromem do šířky musí být v Prologu podstatně obtížnější než průchod do hloubky, který máme v Prologu díky "vestavěnému" backtrackingu "zdarma". To je sice pravda, ale zvolíme-li vhodný úhel pohledu, **liší se algoritmy průchodu do hloubky a do šířky jen použitou pomocnou datovou strukturou**.

Pro průchod budeme používat pomocný seznam, ve kterém si budeme pamatovat "rozpracované podstromy".

- Začneme tím, že do tohoto seznamu dáme celý vstupní strom.
- Jeden krok algoritmu pak spočívá v tom, že z pomocného seznamu vyzvedneme první prvek - to je nějaký strom t(L,V,R) - a pošleme V na výstup a podstromy L a R přidáme do pomocného seznamu.
- Algoritmus končí vyčerpáním pomocného seznamu.

```
%
============================================================
% projdi(+Strom,-Vystup)
%     projde stromem Strom a vydá jeho prvky do seznamu
Vystup, pořadí prvků v
%     seznamu výstup závisí na použité proceduře pridej
%
============================================================
projdi(nil,[]).                          % projít prázdný strom
projdi(Strom,Vysl):-                     % projít neprázdný strom
zpr([Strom],Vysl).                       % inicializace pomocného seznamu
zpr([],[]).                              % ukončující klauzule
zpr([t(L,V,R)|Z],[V|Kon]):-              % vrchol V do výstupu
pridej(L,R,Z,Z1),zpr(Z1,Kon).            % podstromy L a R do pomocného
seznamu
```

Přidáváme-li podstromy na začátek pomocného seznamu, chová se tento jako zásobník, a dostáváme opět procházení do hloubky.

```
%
============================================================
% pridej(+Levy,+Pravy,+Kam,-Vysl)
%     přidá podstromy Levy a Pravy na začátek seznamu Kam,
vznikne seznam Vysl
%
============================================================
pridej(L,R,Z,Z1):-pridej_na_zac(L,R,Z,Z1).
pridej_na_zac(nil,nil,Z,Z).   % prázdné stromy do seznamu nedávám
pridej_na_zac(nil,R,Z,[R|Z]).
pridej_na_zac(L,nil,Z,[L|Z]).
pridej_na_zac(L,R,Z,[L,R|Z]).
```

Na dotaz

?- nas_strom(T),projdi(T,L).

pak dostaneme odpověď

T = ... L = [a,b,e,c,f,h,g]

tedy průchod do hloubky.

Přidáváme-li podstromy na konec pomocného seznamu, chová se tento jako fronta, a dostáváme procházení stromu do šířky.

```
%
============================================================
% pridej(+Levy,+Pravy,+Kam,-Vysl)
%     přidá podstromy Levy a Pravy na konec seznamu Kam,
vznikne seznam Vysl
%
============================================================
pridej(L,R,Z,Z1):-pridej_na_kon(L,R,Z,Z1).
pridej_na_kon(nil,nil,Z,Z).
pridej_na_kon(nil,R,Z,Z1):- conc(Z,[R],Z1).
pridej_na_kon(L,nil,Z,Z1):- conc(Z,[L],Z1).
pridej_na_kon(L,R,Z,Z1):- conc(Z,[L,R],Z1). .
```

Na dotaz

?- nas_strom(T),projdi(T,L).

pak dostaneme odpověď

T = ... L = [a,b,c,e,f,g,h]

tedy průchod do šířky .

Binární vyhledávací stromy

Binární vyhledávací stromy

Nejčastěji se setkáváme se speciální případem binárních stromu, s tzv. **binárními vyhledávacími stromy**. Binární strom nazveme vyhledávacím, pokud pro každý uzel U tohoto stromu platí, že hodnoty v jeho levém podstromě jsou menší než U a hodnoty v jeho pravém podstromě jsou větší než U.
Sestavme několik procedur pro práci s binárními vyhledávacími stromy:

```
%
============================================================
% prvek(?X,+Strom) test, zda X je obsaženo ve stromě Strom
%
============================================================
```

```
prvek(X,t(_,X,_)).              % vrchol stromu je jeho prvkem
prvek(X,t(L,V,R)):-
            X<V,                % X je menší než vrchol V,
            prvek(X,L).         % může tedy být jen v levém podstromě
prvek(X,t(L,V,R)):-
            X>V,                % X je větší než vrchol V,
            prvek(X,R).         % může tedy být jen v pravém podstromě

%
============================================================
% vloz(+X,+Strom,-Strom1)
%     Strom1 vznikne vložením prvku X do stromu Strom (pokud
tam již byl, bude
%     Strom1 stejný jako Strom)
%
============================================================
vloz(X,nil,t(nil,X,nil)).       % Vložení prvku X do prázdného stromu
vloz(X,t(L,X,R),t(L,X,R)).      % X je již vrcholem vstupujícího stromu
vloz(X,t(L,V,R),t(L1,V,R)):-
            X<V,                % X je menší než vrchol stromu
            vloz(X,L,L1).       % L1 vznikne vložením prvku X do levého
                                    podstromu L
vloz(X,t(L,V,R),t(L,V,R1)):-
            X>V,                % X je větší než vrchol stromu
            vloz(X,R,R1).       % R1 vznikne vložením prvku X do pravého
                                    podstromu R
```

Algoritmus vypuštění prvku z binárního vyhledávacího stromu je poněkud obtížnější.

- Je-li vypouštěný prvek listem stromu (tj. nemá-li v něm ani jednoho syna), je jeho vypuštění snadné - nikdo nemá co dědit.
- Má-li vypouštěný prvek nejvýše jednoho syna, pak tento syn zdědí postavení otce.
- Má-li vypouštěný prvek oba syny, je situace komplikovanější. Na místo vypouštěného prvku může přijít buď nejmenší prvek z pravého podstromu nebo největší z levého. V našem programu jsme si vybrali druhou možnost. Nejpravější prvek nemůže mít pravého syna - jde ho tedy vypustit jednoduše.

Myslím, že mi dáte za pravdu, že následující program v Prologu je srozumitelnějším (a pochopitelně i přesnějším) popisem tohoto algoritmu.

```
%
============================================================
% vypust(+Prvek,+Vstrom,-Zbytek)
%     Zbytek je strom, který vznikne ze vstupního stromu
Vstrom vypuštěním
%     prvku Prvek
%
============================================================
vypust(X,nil,nil).                  % vypuštění z prázdného stromu
vypust(X,t(nil,X,R),R).             % vypuštění prvku, který nemá
levého syna
 vypust(X,t(L,X,nil),L).            % vypuštění prvku, který nemá
pravého syna
```

```
vypust(X,t(L,X,R),t(L1,Y,R)):-          % vypouštěný prvek X,
                    který má oba syny
       nejprav(L,L1,Y).                 % bude nahrazen prvkem
                                        Y, který je nejpravějším uzlem v levém
                                        podstromu L
vypust(X,t(L,V,R),t(L1,V,R):-
       X<V,                             % prvek X menší než je kořen stromu V
       vypust(X,L,L1).                  % se vypouští z levého podstromu
vypust(X,t(L,V,R),t(L,V,R1):-
       X>V,                             % prvek X větší než je kořen stromu V
       vypust(X,R,R1).                  % se vypouští z pravého podstromu R

%
============================================================
% nejprav(+Vstrom,-Zbytek,-Nejpr)
%     pomocná procedura pro vypust:Zbytek je strom, který
vznikne ze vstupního %       stromu Vstrom vypuštěním jeho
nejpravějšího prvku Nejpr
%
============================================================
nejprav(t(L,X,nil),L,X).            % je-li pravý podstrom prázdný, je
                                    nejpravějším prvkem kořen stromu
nejprav(t(L,X,R),t(L,X,R1),Y):-     % jinak je nejpravějším prvkem
            nejprav(R,R1,Y).        % nejpravější prvek pravého podstromu
```

Binární vyhledávací stromy jsou výhodnou datovou strukturou pro vyhledávání, pokud jsou jejich podstromy na všech úrovních, co nejvyváženější. V nejhorším případě však jde jen o složitěji realizovaný seznam (je-li u každého uzlu jeden podstrom prázdný).

Dokonale vybalancované binární vyhledávací stromy jsou takové binární vyhledávací stromy, ve kterých se v každém uzlu počet prvků v jeho pravém a levém podstromě liší nejvýše o jeden.
Následující procedura **staví ze začátku (dané délky) rostoucí posloupnosti prvků dokonale vyvážený strom**:

```
%
============================================================
% postav(+N,+RostSez,-VybStrom,-Zbytek)
%     Předpokládá, že RostSez je rostoucí seznam. Je-li
délky alespoň N,
%       postaví použití procedury z jeho prvních N prvků
dokonale vybalancovaný
%     strom. Je-li seznam kratší, postaví se strom ze všech
prvků seznamu.
%     Zbytek je nespotřebovaný zbytek seznamu
%
============================================================
postav(_,[],nil,[]).                % vstupní seznam byl vyčerpán
postav(0,T,nil,T).      % do stromu již není třeba žádný prvek přidávat
postav(N,In,t(L,V,R),Z):-
            N1 is N-1,          % v obou podstromech bude N1 prvků
```

```
        NR is N1 div 2,          % vpravo případně o 1 méně
        NL is N1-NR,             % než vlevo
        postav(NL,In,L,[V|In1]), % do levého přijde NL prvků
        postav(NR,In1,R,Z).      % do pravého přijde NR prvků
```

Řešený příklad 23: **(Examples\Ch07Ex09.pro)**

```
domains
  treetype = tree(string, treetype, treetype) ;
empty()

predicates
   print_all_elements(treetype)

clauses
   print_all_elements(empty).

   print_all_elements(tree(X, Y, Z)) :-
      write(X), nl,
      print_all_elements(Y),
      print_all_elements(Z).

goal
   print_all_elements(tree("Cathy",
                           tree("Michael",
                           tree("Charles",       empty,
empty),
                      tree("Hazel", empty, empty)),
                           tree("Melody",
                           tree("Jim", empty, empty),
                           tree("Eleanor",       empty,
empty)))).
```

V příkladu vytváříme strom rodičů a potomků, který následně vypisujeme pomocí rekurzivní procedury.

Řešený příklad 24: **(Examples\Ch07Ex10.pro)**

```
domains
   treetype = tree(string, treetype, treetype) ;
empty()

predicates
   create_tree(string, treetype)
   insert_left(treetype, treetype, treetype)
   insert_right(treetype, treetype, treetype)

clauses
   create_tree(A, tree(A, empty, empty)).
```

```
    insert_left(X, tree(A, _, B), tree(A, X, B)).
    insert_right(X, tree(A, B, _), tree(A, B, X)).

goal


    create_tree("Charles", Ch),
    create_tree("Hazel", H),
    create_tree("Michael", Mi),
    create_tree("Jim", J),
    create_tree("Eleanor", E),
    create_tree("Melody", Me),
    create_tree("Cathy", Ca),


    insert_left(Ch, Mi, Mi2),
    insert_right(H, Mi2, Mi3),
    insert_left(J, Me, Me2),
    insert_right(E, Me2, Me3),
    insert_left(Mi3, Ca, Ca2),
    insert_right(Me3, Ca2, Ca3),

    write(Ca3), nl.
```

Na rozdíl od předchozího příkladu, zde strom vytváříme postupně, nikoliv jako jeden term.

Nejdůležitější probrané pojmy:

- seznam
- strom

Úkol k textu:

Vytvořte stromový zápis 2 libovolných aritmetických výrazů se sčítáním, násobením a celými čísly (alespoň se čtyřmi operandy) a dále predikát, který vypíše v programu výraz normálním infixním zápisem.

Korespondenční úkol:

Vyberte si jeden z následujících úkolů a naprogramujte jej v Prologu. Podrobnosti zadání specifikujte s tutorem.

1. Nalezení nejkratší cesty mezi 2 městy.
2. Symbolická derivace polynomu.
3. Obarvení mapy.
4. Vytvoření seznamu četnosti výskytu symbolů v textu.
5. Řešení úlohy typu Zebra (např. známe neúplnou informaci o domech, jejich majitelích, zvířatech a chceme zjistit kompletní situaci).

Další úlohy podobné složitosti dle Vaší volby lze konzultovat s tutorem.

7. Práce s databází a ladění

Cíl:

Po prostudování této kapitoly pochopíte:
- způsob práce s interní databází Prologu
- možnosti ladění

Naučíte se:

- vytvářet jednoduché programy s databází

Prolog umožňuje pracovat kromě znalostní báze tvořené samotným logickým programem také s interní databází. Zdálo by se, že taková možnost je zbytečná, když celý logický program je vlastně „databáze“ faktů a pravidel. Tato interní databáze však poskytuje mnohem více. Představte si, že chcete v Prologu vytvořit program, který by se mohl učit novým faktům a pravidlům. To však nemůžete jednoduše realizovat jen s pomocí logického programu. Jistě mohli byste si vytvořit vlastní datové struktury pro ukládání a manipulaci s těmito informacemi, ale to byste dělali zbytečnou práci, neboť ten nejúčinnější nástroj máte přece již k dispozici – je jím Prolog samotný. Mnohem efektivnější je tedy mít nástroj, který Vám i za běhu programu umožní do Prologovského programu přidávat informace. Tímto nástrojem právě tato databáze je.

7.1. Predikáty pro práci s databází

Predikáty assert a retract

Pro práci s databází slouží především tyto predikáty:

- assert: vkládá novou klauzuli do databáze pro daný predikát
- asserta: vkládá novou klauzuli na začátek databáze pro daný predikát
- assertz: vkládá novou klauzuli na konec databáze pro daný predikát
- retract: vyhledá v databázi klauzuli pro daný predikát a odstraní ji
- retracta: vyhledává od začátku v databázi klauzuli pro daný predikát a odstraní ji
- retractz: vyhledává od konce v databázi klauzuli pro daný predikát a odstraní ji
- retractall: vyhledává v databázi klauzule pro daný predikát a odstraní je všechny z databáze

Prodívejme se na příklady práce s těmito predikáty.

Řešený příklad 25: (Examples\Ch10Ex03.pro)

```
domains
   list = symbol*

database
   owns(symbol,symbol)

predicates
   assert_facts
   gather_goods(symbol)
   write_list(list)

clauses
   assert_facts :-
         assertz( owns(micki, dog) ) ,
         assertz( owns(micki, car) ) ,
         assertz( owns(micki, dress) ) ,
         assertz( owns(eloise, car) ) ,
         assertz( owns(claudio, tennis_racket) ) ,
         assertz( owns(claudio, sneakers) ) ,
         assertz( owns(kenny, boots) ).

   gather_goods(Who) :-
         findall(Goods,           owns(Who,Goods),
Goods_list) ,
         write(Who, " owns:\n") ,
         write_list(Goods_list).

   write_list([]):- !.
   write_list([H|T]):-write(H,"\n"),
write_list(T).

goal
   assert_facts ,
   gather_goods(micki) ,
   write("That's all!\n").
```

V programu se nejprve pomocí predikátu assertz přidávají od konce fakta o tom, kdo co vlastní (predikát owns). Poté se pomocí predikátu findall naleznou všechny vložené informace a uloží do seznamu. Pomocí predikátu write_list pak můžete vypsat obsah tohoto seznamu. Vidíte, že každý predikát v interní databázi musí deklarován ve speciální sekci database.

7.2. Ladění

Trace

O možnostech ladění v Prologu se zmíníme jen velmi stručně. Jelikož mechanismus tohoto jazyka je velmi odlišný od procedurálních jazyků, jsou i způsoby ladění v Prologu jiné. Základním mechanismem je trasování logického odvozování nových klauzulí resp. srovnávání a navracení. Pro tento účel slouží především direktiva **trace**.
Tento prvek jazyka způsobí, že nedojde k vyhodnocení dotazu přímo najednou, ale máte možnost po krocích sledovat srovnávání a navracení v logickém programu (v Turbo Prologu 2.0 pomocí klávesy F10).V okně Edit se pak nastavuje kurzor podle tohoto, který cíl se zrovna splňuje a v okně Trace můžete sledovat postupně srovnávání a navracení. Trasování lze aktivovat i z nabídky Options.

Velice ilustrativní je následující příklad.

Řešený příklad 26: (Examples\Ch10Ex03.pro)

```
trace
domains
   name, sex, occupation, object, vice, substance
= symbol
   age=integer

predicates
   person(name, age, sex, occupation)
   had_affair(name, name)
   killed_with(name, object)
   killed(name)
   killer(name)
   motive(vice)
   smeared_in(name, substance)
   owns(name, object)
   operates_identically(object, object)
   owns_probably(name, object)
   suspect(name)

clauses
   person(bert, 55, m, carpenter).
   person(allan, 25, m, football_player).
   person(allan, 25, m, butcher).
   person(john, 25, m, pickpocket).

   had_affair(barbara, john).
   had_affair(barbara, bert).
   had_affair(susan, john).

   killed_with(susan, club).
```

```
killed(susan).

motive(money).
motive(jealousy).
motive(righteousness).

smeared_in(bert, blood).
smeared_in(susan, blood).
smeared_in(allan, mud).
smeared_in(john, chocolate).
smeared_in(barbara, chocolate).

owns(bert, wooden_leg).
owns(john, pistol).


operates_identically(wooden_leg, club).
operates_identically(bar, club).
operates_identically(pair_of_scissors, knife).
operates_identically(football_boot, club).

owns_probably(X, football_boot) :-
   person(X, _, _, football_player).
owns_probably(X, pair_of_scissors) :-
   person(X, _, _, hairdresser).
owns_probably(X, Object) :-
   owns(X, Object).


suspect(X) :-
   killed_with(susan, Weapon) ,
   operates_identically(Object, Weapon) ,
   owns_probably(X, Object).


suspect(X) :-
   motive(jealousy) ,
   person(X, _, m, _) ,
   had_affair(susan, X).


suspect(X) :-
   motive(jealousy) ,
   person(X, _, f, _) ,
   had_affair(X, Man) ,
   had_affair(susan, Man).

suspect(X) :-
```

```
      motive(money) , person(X, _, _, pickpocket).

killer(Killer) :-
      person(Killer, _, _, _) ,
      killed(Killed) ,
      Killed <> Killer , /* It is not a suicide */
      suspect(Killer) ,
      smeared_in(Killer, Goo) ,
      smeared_in(Killed, Goo).
```

V tomto příkladu se pokoušíme na základě informací o osobách najít mezi nimi vraha - dotaz na killer(X). Pomocí trasování můžete velmi dobře pochopit, jak daný logický program funguje a odhalovat případné chyby.

Nejdůležitější probrané pojmy:

- interní databáze
- trasování a ladění

Úkol k textu:

Vyzkoušejte ladit libovolný program v Prologu (nejlépe Váš vlastní).

Korespondenční úkol:

Vytvořte program, který umožní zadávat za běhu programu informace o zvolených objektech (např. automobil, kniha apod.).

8. Principy logického programování

Cíl:

Po prostudování této kapitoly pochopíte:

- hlouběji principy vyhodnocení dotazu

Program v Prologu tvoří vždy konečná uspořádaná množina faktů a pravidel zapsaných za sebou tak, že všechna fakta týkající se nějakého predikátu a pravidla s týmž predikátem v hlavě jsou sdruženy vedle sebe. Množině všech takových klauzulí programu se někdy říká **definice příslušného predikátu.** Budeme předpokládat, že klauzule definující predikát jsou očíslovány vždy od čísla 1 pořadím, ve kterém jsou uvedeny.

Jak Prolog postupuje při hledání odpovědi na předložený dotaz? Vytváří postupně posloupnost dotazů s cílem získat **dotaz prázdný**, tj. dotaz, ve kterém se nevyskytují žádné podcíle, neboť má tvar
?-.
Pro postupnou redukci dotazu slouží klauzule programu, které se chápou jako návod ke „zjednodušení“ uvažované posloupnosti cílů. Podaří-li se dospět k prázdnému dotazu, Prolog odpovídá na výchozí dotaz *kladně*, tj.
pokud výchozí dotaz neobsahoval žádné proměnné, ohlásí splnění dotazu odpovědí yes,
jinak podá zprávu i o substitucích za všechny proměnné výchozího dotazu, které bylo nutné cestou k prázdné klauzuli provést.

K ostatním způsobům zakončení práce programu v Prologu se vrátíme na konci tohoto odstavce.

8.1. Procedurální interpretace Prologu

Povšimněme si podrobněji, postupu generování dotazů, tj. podejme **procedurální interpretaci programu** v Prologu, která přesně popisuje postup hledání řešení předloženého výchozího dotazu
?-G_1,…G_n. (V)
Uvedli jsme již, že Prolog vytváří **posloupnost dotazů**. Každý dotaz je v ní navíc doplněn o informace vztahující se k
dosud použitým postupům řešení jeho podcílů (viz údaj *Pořadí* v následujícím popisu),
aktuální substituci za proměnné v dotazu

Tato posloupnost je realizována jako zásobník, který je položením výchozího dotazu „vytvořen“. V okamžiku svého zrodu má zásobník

jediný prvek, kterým je právě výchozí dotaz - je to jediný prvek zásobníku, pro který není definován předchůdce. Abychom mohli o všech prvcích zásobníku hovořit jednotně, uvedeme i pro výchozí dotaz informace odkazující na použité klauzule programu, tedy každému podcíli G_1, … G_n je přiřazeno číslo 0, neboť dosud pro ně nebylo použito žádné klauzule z programu. Substituce je na tomto místě prázdná. Dále se pracuje se zásobníkem dotazů takto:

1) Je-li zásobník prázdný, pak zpracování dotazu končí neúspěchem a Prolog odpovídá na dotaz (V) *negativně* zprávou no, jinak se nejvrchnější prvek zásobníku stává aktuálně řešeným dotazem:

Je-li aktuálním dotazem prázdný dotaz, pak prohledávání skončilo úspěchem a výchozí dotaz je řešen pozitivně, Prolog poskytne informace o tom, jaké substituce za proměnné v dotazu (V) bylo nutné na cestě za prázdným dotazem udělat a odpovídá yes.

jinak má aktuální dotaz tvar

$?-G_1,\ldots G_n.$

(A)

Prolog dotaz ruší tak, že se snaží nalézt řešení pro jeho první podcíl zleva, tj. pro G_1, přitom bere v úvahu údaj *Pořadí 1* spojený s tímto cílem v aktuálním dotazu - viz bod 2.

2) Při **řešení zvoleného cíle** G_1 s údajem *Pořadí 1* hledá Prolog v programu první takový fakt či pravidlo, jehož hlavu lze s tímto cílem ztotožnit (fakt zde chápeme jako pravidlo s prázdným tělem) a jehož pořadové číslo v definici predikátu cíle G_1 je vyšší než *Pořadí 1*. Při tom plně respektuje uspořádání klauzulí programu:

Nalezne-li Prolog vhodnou klauzuli (K) uvažovaného programu

$D:-E_1,\ldots E_n.$

(K)

jehož hlavu *D* lze ztotožnit s G_1 , pak vytvoří nový dotaz tak, že

nahradí zvolený cíl podle doporučení pravidla, tj. provede rezoluci pro rodičovský pár (A), (K). Postupuje tak, že nalezne nejobecnější unifikaci δ atomů *D* a C_1, kterou použije pro vytvoření varianty δ pravidla (K) i dotazu (A). Rezolventa vznikne tak, že 1. cíl v uvažované variantě dotazu je nahrazen tělem varianty pravidla (K). Výsledkem je dotaz tvaru

$?–(B_1,\ldots B_n,C_2,\ldots C_n)$

(N)

Označme *Pořadí 2* pořadové číslo použitého pravidla (K). Aktuální dotaz (A) v zásobníku upravíme tak, že aktualizujeme informaci o poslední klauzuli použité k řešení cíle C_1 – údaj *Pořadí 1* je zaměněn údajem *Pořadí 2*. (Tato změna zajišťuje to, že při zpětném chodu bude hledáno řešení na jiné větvi stavového prostoru.)

Výsledný dotaz (N) je pak uložen navrch zásobníků – všechny predikáty (B_1, … B_k) δ převzaté z těla pravidla (K) získávají index 0, ostatním zůstává stejné pořadí, jaké měly v dotazu (A). Aktuální substituce dotazu (N) vznikne složením substitutce příslušející dotazu (A) s použitou substitucí δ. Řešení dále pokračuje od kroku 1.

jinak (tj. pokud se nepodaří aktuálně řešený podcíl unifikovat s hlavou žádného vhodného pravidla v programu) systém použije **zpětný chod** a hledá alternativní cestu k řešení tak, že ze zásobníku odejme poslední prvek a pokračuje od kroku1.
Povšimněme si, že odejmutím posledního prvku ze zásobníku se ruší substituce realizovaná v posledním kroku k dotazu (A) a odpovídající proměnná při té příležitosti „ztratí“ získanou konkrétní hodnotu!

Nenaučený způsob vede k nalezení prvního řešení zvolenou strategií. Máme-li zájem hledat i **ostatní možná řešení**, můžeme pokračovat ve vyšetřování vygenerovaného zásobníku i po nalezení prvního prázdného dotazu. Stačí využít zpětný chod, tj. odstranit ze zásobníku poslední prázdný dotaz a věnovat se zbylému obsahu tak, jak popisuje bod 1. Tento postup lze opakovat vždy po vygenerování prázdného dotazu. Pokyn ke zpětnému chodu vydává uživatel žádostí o nalezení alternativního řešení.
Zřejmě lze rozlišit tyto **způsoby ukončení práce programu**:
Pozitivní řešení: V okamžiku, kdy se podaří vyřešit všechny podcíle výchozího dotazu, systém předá uživateli odpověď složenou z postupně vygenerovaných hodnot pro všechny proměnné vyskytující se v původním dotazu (pokud původní dotaz žádné proměnné neobsahoval, pak systém potvrdí vyřešení dotazu hlášením `yes`).
Negativní řešení: Pokud se ani po využití všech možností zpětného chodu nepodaří vyřešit všechny podcíle výchozího dotazu a systém se zastavil, neboť má prázdný zásobník dotazů, pak zní odpověď `no`.
Nekonečná smyčka: Pokud systém upadne do smyčky – například vygenerované dotazy se periodicky opakují, nebo je vytvářený stavový prostor příliš rozsáhlý, pak výpočet vytrvale pokračuje až do chvíle, kdy končí chybovým hlášením, které upozorňuje na vyčerpání paměti.

Nejdůležitější probrané pojmy:

- postup při vyhodnocení dotazu
- hledání všech řešení

9. Historie logického programování a jazyka Prolog

Cíl:

Získáte základní přehled o těchto otázkách:

- na jakých principech je jazyk Prolog založen
- jak tento jazyk vznikal a jaké jsou jeho implementace a alternativní prostředky založené na logickém programování

9.1. Historie Prologu

Jazyk Prolog vznikl v r. 1972 ve Francii na univerzitě v Luminy (Marseille). První verzi jazyka vytvořili A. Colmerauer a P. Roussel jako prostředek pro efektivní odvozování logických důsledků z určité třídy formulí predikátové logiky.

Teoretické modely

V r. 1974 R. Kowalski analyzoval předpoklady a způsob práce jazyka Prolog a vytvořil teoretický model Prologu na základě procedurální interpretace Hornových klauzulí.

Na tento model navazuje další implementace v Edinburgu a v Londýně, z nichž zejména úspěšná Warrenova implementace z r. 1977 určila do značné míry syntax jazyka. Právě z této implementace vychází první učebnice Prologu od Clocksina a Mellishe.

V 70. letech vznikají další implementace Prologu v Itálii, Portugalsku, Belgii, Maďarsku, Polsku, aj. Většina z nich má ještě charakter interpretu, Warrenova implementace zahajuje éru překladačů Prologu.

Hardwarové implementace

K významnému obratu došlo v r. 1981. Způsobil ho japonský projekt počítačů 5.generace, kde byl Prolog zvolen jako základní jazyk logického procesoru centrální jednotky počítače. Vyvolalo to vlnu zájmu o Prolog a metody logického programování. Prolog se začal rychle prosazovat vedle jiných jazyků pro symbolické výpočty (např. Lisp).

Vedle interpretů Prologu rostla i úloha překladačů. První Warrenův překladač byl následován řadou dalších. Warren v r. 1983 shrnul poznatky z konstrukce různých překladačů do abstraktního modelu překladače Prologu, WAM (Warren Abstract Machine). Z něj vychází další komerčně úspěšné překladače (např. Arity Prolog, Prolog 86, Prolog II, Quintus Prolog).

Novou kapitolu otevírá paralelní zpracování logických programů. Zatím vzniklo několik jazyků pro paralelní výpočty logických programů. Patří k

nim především Parlog od Clarka a Gregoryho, Shapirův Concurrent Prolog a jazyky GHC (Guarded Horn Clauses) a KL1 (Kernel Language One).

9.2. Použití Prologu

Od počátku byl Prolog využíván při zpracování přirozeného jazyka (francouštiny) a pro symbolické výpočty v různých oblastech umělé inteligence. Používá se v databázových a expertních systémech, při počítačové podpoře specializovaných činností, např. při projektování (CAD – Computer Aided Design) nebo výuce (CAI – Computer Aided Instruction), i v klasických úlohách symbolických výpočtů, jako je návrh a konstrukce překladačů, a to nejen jako prostředek vhodný pro reprezentaci a zpracování znalostí, ale i jako nástroj pro řešení úloh.

Program v Prologu definuje konečný počet relací pomocí příkazů, popisující jejich vlastnosti. Můžeme říci, že program deklaruje vše, co se může vypočítat. Problém vytvoření podrobného plánu pro tuto činnost na první pohled ustupuje do pozadí. Tím se Prolog podobá specifikačním jazykům, které dovolují oddělit problém úplné a korektní specifikace od problému efektivnosti průběhu (výpočtu) vytvářeného programu.

Výhoda: je snazší upravit specifikaci než hotový program. Způsob výpočtu logického programu je dán procedurální interpretací jeho příkazů.

Prolog pracuje s relacemi způsobem, který zahrnuje operace používané v relačních databázových modelech a dovoluje definovat dotazovací jazyky pro specifické potřeby uživatele databáze. Je tedy vhodný prostředkem i pro databázové aplikace.

Podmíněné příkazy Prologu mají tvar implikací:

A platí, jsou-li splněny podmínky **P1, P2, ..., Pn**

Prolog je konverzační jazyk nové generace programovacích jazyků, která by měla usnadnit tvorbu programů co nejširšímu okruhu uživatelů. Snaží se odstranit část rutiny při programování a tím si zajistit místo vedle osvědčených jazyků jako Fortran, Cobol a Pascal.

Uživatel Prologu se může více soustředit na popis vlastností relací, tedy na otázku *co* se má vypočítat, aniž by byl na každém kroku nucen řešit detaily spojené s otázkou *jak* to udělat a *kam* uložit získané výsledky.

V Prologu nejsou příkazy pro řízení běhu výpočtu ani příkazy pro řízení toku dat. Chybějí prostředky pro programování cyklů, větvení apod. a nepoužívá se přiřazovací příkaz. S tím souvisí i rozdílná úloha proměnných. *Proměnná* v Prologu označuje po dobu výpočtu objekt, který vyhovuje určitým podmínkám. Jeho vymezení se při výpočtu upřesňuje.

Průběh výpočtu je v Prologu řízen jeho interpretem na základě znění programu. Programátor může chod výpočtu ovlivnit řídícími příkazy v mnohem menší míře než u jiných jazyků.

I když byl původně Prolog navržen jako jazyk specializovaný na symbolické výpočty, moderní implementace směřují k obecnějšímu použití.

9.3. Alternativy k Prologu

ALF (Algebraic Logic Functional programming language, programovací jazyk pracující na základě algebraické logiky) je jazyk, který kombinuje funkční a logické programovací techniky. Základem programu ALF je logika Hornových klauzulí s rovností která se skládá z predikátů a Hornových klauzulí pro logické programování, funkce a rovnostmi pro funkční programování. Abstraktní stroj je postaven na Warrenově abstraktním stroji (Warren Abstract Machine - WAM) s několika rozšířeními k implementaci zužování a přepisování. V současné implementaci programů tohoto abstraktního stroje jsou spouštěny emulátorem napsaným v programovacím jazyku C.

CORAL je deduktivní databázový/logický programovací systém vyvinutý na Univerzitě Wisconsin-Madison. Je to deklarativní jazyk založený na pravidlech Hornových klauzulí s rozšířeními jako příkazy SQL group-by a agregačními operátory. Tento program využívá syntaxi podobnou programovacímu jazyku Prolog.

LOLLI je interpret pro logické programování založený na principech lineární logiky. Lolli, je pojmenovaný po implikačním operátoru lineární logiky, tento operátor vypadá takto ‘‘-o‘‘. Programu se také často říká lollipop což znamená lízátko. Lolli je plná implementace jazyka popsaného v knize "Logic Programming in a Fragment of Intuitionistic Linear Logic" (od autorů Josh Hodas & Dale Miller). Tento programovací jazyk se o trochu liší v syntaxi a má několik extra-logických predikátů a operátorů.

MERCURY je nově čistě deklarativní logický programovací jazyk. Stejně jako Prolog a ostatní existující programovací jazyky pro programování logiky, je tento jazyk velmi vysoce úrovňový což umožňuje soustředit se na problém a ne na nízko úrovňové záležitosti jako je správa paměti. Narozdíl od Prologu, který je orientován směrem průzkumného programování, tak programovací jazyk Mercury je navržen na konstruování velkých, spolehlivých a efektivních systémů tvořený týmy programátorů. Jako následek je to, že programování v Mercury má trochu jinou podobu než programování v Prologu.

Hlavní vlastnosti Mercury:

- Mercury je čistě deklarativní: predikáty v Mercury nemají nelogické vedlejší účinky
- Mercury je jazyk silně založený na typech: Systém typů v Mercury je systém založený na logice mnoha druhů s parametrickým polymorfismem, velmi podobný je typovému systému moderních funkčních jazyků jako je ML a Haskell. Chyby v typech jsou hlášeny při kompilaci.
- Mercury je silně založený na stavech.
- Mercury má silně deterministický systém.
- Mercury má systém modulů.
- Mercury podporuje programování vyššího řádu se závorkami a lambda výrazy.
- Mercury je velmi efektivní (ve srovnání s existujícími logickými programovacími jazyky)
- Silné typy, stavy a determinismus poskytují kompilátoru informace, které potřebuje k vytvoření velmi efektivního kódu.

GOEDEL je považován za nástupce Prologu. Hlavním cílem Goedel je to, aby tento programovací jazyk měl výrazovost podobnou Prologu, ale mnohem lépe řešenou stránku deklarací a sémantiky ve srovnání s Prologem. Tato vylepšená deklarativní sémantika má značný výhody pro konstrukci programu, kontrolu, opravy, transformace a tak dále. Srovnatelný důraz je kladen na meta-logické vybavení, které Prolog postrádá.

Goedel má deklarativní náhrady pro var, nonvar, assert a retract. Goedel je silně typový jazyk, jeho typový systém je založen na logice mnoha druhů s parametrickým polymorfismem. Má systém modulů a podporuje čísla s plovoucí čárkou a čísla integer s nekonečnou přesností. Může vyřešit rozhodování nad konečnými základy čísel integer a také rozhodování o racionálních základech. Podporuje též zpracování konečných množin. Má také pružná pravidla pro výpočet a vkládání operátorů, které zobecňuje vazbu současných programovacích jazyků pro programování logiky.

Nejdůležitější probrané pojmy:

- historie vzniku jazyka Prolog
- alternativní implementace logického programování

Literatura

POLÁK, J.: Prolog. Grada 1992, Praha.
ŠTĚPÁNEK, P. Programování v jazyku Prolog. SNTL 1991, Praha.
Turbo Prolog: User's guide. Borland International 1988, Scotts Valley.
Turbo Prolog: Reference guide. Borland International 1988, Scotts Valley.
MAŘÍK, Vladimír a kol. Umělá inteligence 1. Academia 1993, Praha.
MAŘÍK, Vladimír a kol. Umělá inteligence 2. Academia 1995, Praha.
Internetové odkazy na tutoriály a implementace Prologu jako např.:
Logic Programming (http://www.afm.sbu.ac.uk/logic-prog/)