INFORMATIKA

O programovacím jazyku PROLOG

MIROSLAV KOLAŘÍK

Přírodovědecká fakulta UP, Olomouc

PROLOG je interpretační (neprocedurální) jazyk. Patří mezi deklarativní programovací jazyky – potlačuje imperativní[1)] složku. PROLOG je využíván především v oboru umělé inteligence a v počítačové lingvistice (obzvláště pro zpracování přirozeného jazyka, pro nějž byl původně navržen). PROLOG je založen na predikátové logice. Základními využívanými přístupy jsou unifikace (speciální substituce), rekurze a backtracking (metoda prohledávání do hloubky).

O logickém programování

Program je souborem tvrzení, kterými programátor (uživatel, expert) popisuje určitou část okolního světa. Výpočet nad daným programem, který je iniciován zadáním dotazu, je hledání důkazu dotazu z daného souboru tvrzení.

Logické programování je jedním z paradigmat programování (programovacích stylů). Základními rysy, kterými se logické programování zásadně odlišuje od ostatních programovacích paradigmat jsou:

- programátor specifikuje, co se má vypočítat, a ne jak se to má vypočítat a kam uložit mezivýsledky,
- nejsou potřeba příkazy pro řízení běhu výpočtu ani pro řízení toku dat, nejsou potřeba příkazy cyklů, větvení, ani přiřazovací příkaz,

[1)] Imperativní paradigma popisuje, jak vyřešit problém. Deklarativní paradigma popisuje, co je problém.

- neexistuje rozdělení proměnných na vstupní a výstupní,
- nerozlišuje se mezi daty a programem.

Dále je typické, že logický program je konečná množina logických formulí, přičemž výpočet je zahájen zadáním dotazu, což je logická formule, kterou zadává uživatel. Cílem výpočtu je najít důkaz potvrzující, že dotaz logicky vyplývá z logického programu. Pokud je zjištěno, že dotaz z programu vyplývá, výpočet končí a uživateli je oznámeno `true` s hodnotami případných proměnných, které se v dotazu vyskytují. Pokud není zjištěno, že dotaz z programu vyplývá, výpočet končí a uživateli je oznámeno `false`. Dodejme, že se může stát, že výpočet neskončí, protože uvázne v nekonečném cyklu.

Abychom si osvojili základní principy logického programování, zaměříme se na programovací jazyk PROLOG.

Základem PROLOGu je databáze klauzulí, které lze dále rozdělit na fakta a pravidla, nad kterými je možno klást dotazy formou tvrzení, u kterých PROLOG zhodnocuje jejich pravdivost. Nejjednoduššími klauzulemi jsou fakta, která pouze vypovídají o vlastnostech objektu nebo vztazích mezi objekty. Složitějšími klauzulemi jsou pravidla, která umožňují odvozovat nová fakta. Zapisují se ve tvaru

```
hlava :- telo.
```

kde `hlava` definuje odvozovaný fakt, `telo` podmínky, za nichž je pravdivý. Tělo pravidla obsahuje jeden či více cílů. Pokud se interpretu podaří odvodit, že je tělo pravdivé, ověří tím pravdivost hlavy.

První příklad

Víme, že se dá z Prahy letět přímo do Paříže a do Lyonu. Dále víme, že se dá přímo letět z Paříže do Marseille a z Marseille přímo do Vídně. Letecká spojení mezi městy jsou buď přímá nebo přes nějaká další města.

Uvedené informace nyní přepíšeme tak, aby syntaxe odpovídala PROLOGu. Dostaneme následující logický program:

```
direct(praha,pariz).
direct(praha,lyon).
direct(pariz,marseille).
direct(marseille,viden).
connection(X,Y) :- direct(X,Y).
connection(X,Z) :- direct(X,Y), connection(Y,Z).
```

První čtyři řádky jsou fakta, předposlední a poslední řádek jsou pravidla. Všimněme si, že jak fakta, tak pravidla jsou zakončena tečkou, a že před levou závorkou nikdy není mezera. Jelikož jsou velká počáteční písmena vyhrazena pro proměnné, píšeme zde města s malými počátečními písmeny.

Nainstalujme si nyní SWI-PROLOG. Je to jedna z mnoha aplikací, ve které můžeme vytvářet, překládat a testovat programy psané v PROLOGu. Autorem SWI-PROLOGu je Jan Wielemaker z University of Amsterdam. Zdarma lze stáhnout (verze pro Unix, MS-Windows) ze stránky[2]:

http://www.swi-prolog.org

Do právě staženého programu můžeme načítat programy v PROLOGu (a to přes nabídku: `File` → `Consult ...`)[3]. Ihned po načtení konkrétního souboru můžeme zadávat konkrétní dotazy, na které nám interpret PROLOGu obratem odpoví. Zkopírujte (nebo přepište) výše uvedených šest řádků do (obyčejného) textového souboru a po uložení (typicky s příponou `.pl`) soubor načtěte do aplikace SWI-PROLOG. Nyní je PROLOG připraven na dotazy, které se zapisují za symboly „`?-`“[4]. Nyní vyzkoušíme zadat některé jednoduché dotazy.[5] Níže uvedené dotazy vyzkoušejte na počítači. Nebojte se vyzkoušet i chybně položené dotazy[6], ať víte, jak na ně interpret PROLOGu zareaguje.

```
?- direct(praha,lyon).
```

Jelikož je tento fakt součástí našeho programu, odpoví PROLOG:

```
true.
```

Nyní položíme podobný dotaz:

```
?- direct(lyon,praha).
```

[2] Nebo se podívejte do repozitářů své distribuce.

[3] Pokud máte swipl puštěný v konzoli (a ne v okenním prostředí), tak lze načtení souboru „file.pl“ provézt pomocí „[file]“.

[4] Každý dotaz musí být ukončen tečkou. Klávesou „Enter“ se aktivuje vyvolání odpovědi.

[5] Po vložení dotazu PROLOG spustí inferenční mechanismus a snaží se najít odpověď na dotaz. Odpověď je buďto nalezena (a zodpovězena) nebo nikoliv (například proto, že se interpret dostal do nekonečného cyklu).

[6] Například dotaz se špatným počtem argumentů, nebo s chybějící závorkou, nebo s neznámým relačním symbolem a tak podobně.

Tento dotaz z našeho programu nijak nevyplývá, proto na něj PROLOG odpoví:

```
false.
```

Nyní zadáme dotaz s proměnnou `X`:

```
?- direct(praha,X).
```

Na tento dotaz můžeme v souladu s naším programem dostat dvě různé (pravdivé odpovědi):

```
X=pariz;
X=lyon.
```

Pomocí středníku požadujeme další odpovědi, které můžeme na náš dotaz obdržet. Pokud bychom místo středníku stiskli klávesu „`Enter`“, seznam případných dalších odpovědí nedostaneme; můžeme však položit nový dotaz. Všimněme si, že při zadání dotazu s proměnnou `X` dostáváme jako odpověď možné hodnoty této proměnné.

Zkuste odhadnout a poté si sami ověřte, jaké odpovědi dá PROLOG na dotaz obsahující dvě různé proměnné `X` a `Y`.

```
?- direct(X,Y).
```

Složitější a zajímavější je binární relace `connection`, která je definována pomocí dvou pravidel. Pravidlo

```
connection(X,Y) :- direct(X,Y).
```

nám říká, že spojení mezi místy `X` a `Y` může být přímé a pravidlo

```
connection(X,Z) :- direct(X,Y), connection(Y,Z).
```

nám říká, že spojení mezi místy `X` a `Z` může vést také přes místo `Y`, případně přes další místa, mezi kterými je přímé spojení.

Na dotaz

```
?- connection(praha,viden).
```

tak dostaneme odpověď

```
true.
```

neboť z Prahy vedou přímá spojení (přes jednotlivá města) až do Vídně, konkrétně přes Paříž a Marseille. V případě relace `connection` používáme princip rekurze (silný a v PROLOGu často používaný nástroj), který ještě několikrát výhodně použijeme.

Druhý příklad

PROLOG je programovací jazyk pro symbolické, nenumerické výpočty. Zejména je vhodný pro řešení problémů, které obsahují objekty a vztahy (relace) mezi objekty. V následujícím příkladu budeme popisovat rodinné vztahy (mezi jistými osobami). Fakt, že Tom je rodičem Roberta zapíšeme v PROLOGu takto:

```
rodic(tom,robert).
```

Zde jsme zvolili `rodic` jako jméno relace; `tom` a `robert` jsou jejími argumenty. Jména píšeme s malými počátečními písmeny, aby je překladač PROLOGu nepovažoval za proměnné. Podobně můžeme pomocí faktů definovat další vztahy.

V následujícím příkladu budeme uvažovat jednoduchou databázi faktů zachycující hierarchii příbuzenských vztahů. Tato databáze obsahuje fakta zapsaná pomocí binární relace `rodic`, definující vztah: člověk `X` je rodičem člověka `Y`. V databázi jsou dále použity dvě unární relace `muz` a `zena` definující pohlaví dané osoby.[7)]

```
rodic(pavla,robert).
rodic(tom,robert).
rodic(tom,liza).
rodic(tom,vilem).
rodic(robert,anna).
rodic(robert,patricie).
rodic(patricie,jan).
```

[7)] Unární relací rozumíme relaci s jedním argumentem, tedy konkrétní vlastnost nějakého objektu. Binární relací rozumíme relaci popisující vztah mezi dvěma objekty, tedy relaci se dvěma argumenty. Podobně ternární relací rozumíme relaci se třemi argumenty. Například ternární relace `stred(A,S,B)` může být interpretována tak, že bod `S` je středem úsečky určené body `A` a `B` . Obecně n-ární relací rozumíme relaci s n argumenty, tedy relaci, která popisuje vztah n objektů, kde n je přirozené číslo. Dále doplňme, že každá relace se skládá z identifikátoru, za nímž následuje výraz v kulatých závorkách, který obsahuje příslušný počet argumentů.

```
zena(pavla).
zena(liza).
zena(anna).
zena(patricie).
muz(tom).
muz(robert).
muz(jan).
```

Lze vyčíst, že Pavla je rodičem Roberta. Dále, že Tom je rodičem Roberta, Lízy a Viléma a tak dále. Všimněme si, že databáze například nedefinuje, kdo je druhým rodičem Jana, ačkoliv bychom jeho existenci mohli intuitivně předpokládat.

Následujícím dotazem s proměnnou `X` snadno zjistíme, kdo všechno je potomkem Toma.

```
?- rodic(tom,X).
```

```
X=robert;
X=liza;
X=vilem.
```

Vyzkoušíme si nyní dotaz[8)] složený ze dvou částí, které mají být splněny současně:

```
?- rodic(tom,X), zena(X).
```

Ptáme se opět na potomky Toma, ale jen takové, které mají ženské pohlaví (ptáme se tedy na dcery Toma). Tentokrát dostaneme jedinou odpověď:

```
X=liza.
```

K programu postupně doplníme pravidla. Nejprve program rozšíříme zavedením binární relace `potomek`, jako inverzi k relaci `rodic`. Mohli bychom postupně dodefinovat nová fakta, například

```
potomek(robert,pavla).
```

Mnohem elegantnějším řešením je však využít již zavedené relace `rodic`

[8)] Pokud chcete vyvolat předchozí dotazy, stiskněte několikrát klávesu „šipka nahoru“.

a definovat k ní inverzní relaci `potomek`. V PROLOGu toto pravidlo zapíšeme následovně:

```
potomek(Y,X) :- rodic(X,Y).
```

V pravidlu figurují dvě proměnné; můžeme ho číst takto: pro každé `X` a `Y`, jestliže `X` je rodičem `Y`, pak `Y` je potomkem `X`. Mezi fakty a pravidly si můžeme všimnout podstatného rozdílu. Každý fakt, například

```
rodic(tom,liza).
```

je něco, co je vždy, nepodmíněně, pravdivé. Na druhou stranu, pravidla specifikují věci, které jsou pravdivé, jestliže je splněna nějaká podmínka. Říkáme, že pravidla mají

- podmínkovou část (pravá strana pravidla), tzv. tělo pravidla
- důsledkovou část (levá strana pravidla), tzv. hlava pravidla.

Například, v pravidle

```
potomek(Y,X) :- rodic(X,Y).
```

je hlavou `potomek(Y,X)` a tělem `rodic(X,Y)`.

Pokud je podmínka `rodic(X,Y)` pravdivá, pak je jako logický důsledek pravdivá i relace `potomek(Y,X)`.

S přidaným pravidlem pro potomky nyní na dotaz

```
?- potomek(jan,patricie).
```

obdržíme odpověď `true`, ačkoliv relaci `potomek` mezi fakty nenajdeme.

Dále definujeme jednoduchým pravidlem unární relaci `mitdite(X)`, která bude pravdivá v případě, že proměnná `X` je rodičem libovolného dítěte. Jelikož při definici těla pravidla na daném dítěti nezáleží, použijeme tzv. anonymní proměnnou, kterou značíme podtržítkem[9)], tedy symbolem „_“.

```
mitdite(X) :- rodic(X,_).
```

Unární relaci `mitdite` si sami otestujte.

Definujme další dvě pravidla:

```
matka(X,Y) :- rodic(X,Y), zena(X).
prarodic(X,Z) :- rodic(X,Y), rodic(Y,Z).
```

[9)] Přesněji řečeno, která začíná podtržítkem.

Čárka mezi dvěma podmínkami v těle pravidla znamená konjunkci obou podmínek, tedy to, že obě podmínky musí být současně pravdivé.

Jak vidíme relace `matka` je založena na následujícím tvrzení se dvěma proměnnými: pro každé[10] `X` a `Y`, `X` je matkou `Y`, jestliže `X` je rodičem `Y` a `X` je žena.

Relace `prarodic` je založena na následujícím tvrzení se třemi proměnnými: pro každé `X`, `Y` a `Z`, `X` je prarodičem `Z`, jestliže `X` je rodičem `Y` a zároveň `Y` je rodičem `Z`.

Na dotaz, kdo jsou prarodiče Patricie, obdržíme (v souladu s naším programem) dvě odpovědi:

```
?- prarodic(X,patricie).

X=pavla;
X=tom;
false.
```

Pokud by nás zajímalo, kdo všechno jsou prarodiče a nepotřebovali bychom znát jména vnoučat, využijeme dotaz s anonymní proměnnou:

```
?- prarodic(X,_).

X=pavla;
X=tom;
X=robert;
false.
```

Přidejme nyní k našemu programu další dvě pravidla, která definují binární relaci `predek`. První pravidlo definuje přímé (bezprostřední) předky a můžeme ho popsat takto:

Pro každé `X` a `Z`: `X` je předkem `Z`, jestliže `X` je rodičem `Z`.

Druhé pravidlo definuje nepřímé předky. Řekneme, že `X` je nepřímým předkem některého `Z`, jestliže existuje rodičovský řetězec lidí mezi `X` a `Z`. Toto pravidlo definujeme pomocí sebe sama (využijeme tak rekurzi):

Pro každé `X` a `Z`: `X` je předkem `Z`, jestliže existuje `Y` takové, že

(1) `X` je rodičem `Y`

(2) `Y` je předkem `Z`.

[10] Proměnné jsou vždy brány s obecnými kvantifikátory, tedy v nejširším možném rozsahu.

Následují obě pravidla přepsaná do PROLOGu:

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

Promyslete si, proč je relace `predek` korektně zavedena. Nakreslete si k tomu odpovídající obrázek. Funkčnost pravidla si vyzkoušíme na dotazu, který má za cíl zjistit, komu je Pavla předkem.

```
?- predek(pavla,X).

X=robert;
X=anna;
X=patricie;
X=jan;
false.
```

Poznámka 1. Rekurze znamená sebeopakování. Velmi často se používá v matematice a informatice. V programování rekurze představuje opakované vnořené volání stejné funkce (podprogramu); hovoříme o tzv. rekurzivní funkci. Nedílnou součástí rekurzivní funkce musí být ukončující podmínka určující, kdy se má rekurze zastavit. Po každém kroku volání sebe sama musí dojít ke zjednodušení problému. Pokud nenastane koncová situace, provede se rekurzivní krok. Rekurze je jedním ze základních principů programování v PROLOGu a je v něm často využívána.

Poznámka 2. U delších programů je vhodné využít komentáře. Řádkový komentář začíná symbolem %. Tedy vše, co je za znakem „%“ je až do konce řádku považováno za komentář. Blokový (víceřádkový) komentář začíná dvěma symboly /* a končí dvěma symboly */. Tedy vše, co je mezi symboly „/*“ a „*/“ je bráno jako komentář a při překladu programu je tato část ignorována.

Seznamy a práce s nimi

Dále se budeme věnovat, v PROLOGu velmi často používaným strukturám, tzv. „seznamům“. Seznamy nejprve definujeme a seznámíme se s jejich značením. Poté si na konkrétních ukázkách představíme základní operace se seznamy a přiblížíme si tak práci s nimi.

Seznam je jednoduchá datová struktura široce používaná v nenumerickém programování. Seznamem může být posloupnost libovolného po-

čtu položek, například jaro, léto, podzim, zima. Tento seznam může být v PROLOGu napsán takto:

```
[jaro,leto,podzim,zima]
```

Toto je ale pouze externí vzhled seznamu. Všechny struktury jsou v PROLOGu kořenové stromy[11)] a seznamy nejsou výjimkou.

Jak může být seznam reprezentován jako standardní objekt PROLOGu? Musíme vzít v úvahu dva případy: seznam je buď prázdný, nebo neprázdný. V prvním případě je seznam jednoduše zapsán takto: `[]`. Ve druhém případě seznam sestává z:

- první položky, nazývané hlava seznamu,
- zbývající části seznamu, nazývané ocas.

Pro náš ukázkový seznam

```
[jaro,leto,podzim,zima]
```

je hlavou `jaro` a ocasem seznam:

```
[leto,podzim,zima]
```

Obecně, hlavou seznamu může být cokoliv (libovolný PROLOGovský objekt, například strom, nebo proměnná); ocas musí být vždy seznamem. Hlava a ocas jsou pak spojeny do struktury speciálním funktorem,

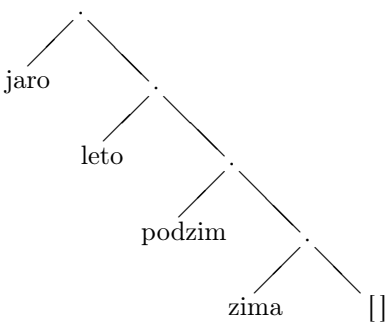
```
.(Hlava, Ocas)
```

Jelikož je ocas opět seznamem, je buď prázdný, nebo má vlastní hlavu a ocas. Proto k reprezentaci seznamu libovolné délky nepotřebujeme žádný další princip. Náš ukázkový seznam je tedy reprezentován jako:

```
.(jaro,.(leto,.(podzim,.(zima,[]))))
```

a následující obrázek ukazuje odpovídající stromovou strukturu.

[11)] Pod pojmem kořenový strom zde máme na mysli souvislý neorientovaný graf bez kružnic s jedním význačným vrcholem (tzv. kořenem).



My budeme nadále používat zejména reprezentaci seznamu pomocí hranatých závorek, přičemž budeme mít na paměti, že interní reprezentace je realizována pomocí binárních[12)] stromů a odpovídá zápisu pomocí teček (tzv. tečkových párů). Poznamenejme ještě, že prvky seznamu mohou být objekty libovolného druhu; tedy i opět seznamy, například

```
[ester,[19,12],simon,[22,12]]
```

Uvedený seznam má čtyři prvky, kde druhým a čtvrtým prvkem je dvouprvkový seznam čísel.

Často je praktické mít možnost zacházet s celým ocasem jako s jediným objektem. Například, nechť

```
L=[a,b,c]
```

Pak můžeme psát `Ocas=[b,c]` a `L=.(a,Ocas)`.

Toto lze v notaci hranatých závorek pro seznamy vyjádřit pomocí svislé čárky (která separuje hlavu a ocas) takto:

```
L=[a|Ocas]
```

Zápis pomocí svislé čárky je ve skutečnosti ještě obecnější. Můžeme uvést libovolný počet prvků, po kterých následuje „`|`“ a seznam zbývajících položek. Alternativní zápisy výše uvedeného seznamu jsou:

```
[a,b,c]=[a|[b,c]]=[a,b|[c]]=[a,b,c|[]]
```

[12)] U binárních stromů má každý vrchol nejvýše dva potomky.

Vybrané operace na seznamech

Nyní se zaměříme na operaci náležení (membership), ověřující, zda je nějaký objekt prvkem seznamu. Implementaci provedeme s binární relací

```
member(X,L)
```

kde `X` je objekt a `L` je seznam. Dotaz `member(X,L)` je pravdivý, jestliže se `X` vyskytuje v `L`. Například

```
member(b,[a,b,c])
```

je pravda,

```
member(b,[a,[b,c]])
```

není pravda a

```
member([b,c],[a,[b,c]])
```

pravda je. Program pro relaci náležení vychází z následujícího pozorování. `X` je prvkem `L`, jestliže

(1) `X` je hlavou `L`, nebo

(2) `X` je prvkem ocasu `L`.

Oba dva body zapíšeme pomocí dvou klauzulí; první je jednoduchý fakt a druhá klauzule je pravidlo:

```
member(X,[X,Ocas]).
member(X,[Hlava|Ocas]) :- member(X,Ocas).
```

Druhou operací se seznamy, kterou se budeme zabývat je spojení (concatenation) dvou seznamů do třetího seznamu. Pro spojení (též zřetězení) seznamů zavedeme ternární relaci:

```
conc(L1,L2,L3)
```

Zde jsou `L1` a `L2` dva seznamy a `L3` jejich spojení. Například

```
conc([a,b],[c,d],[a,b,c,d])
```

je pravda, naproti tomu

```
conc([a,b],[c,d],[a,b,a,c,d])
```

je nepravda. Při definování relace `conc` budeme mít opět dva případy, závislé na prvním argumentu `L1`:

(1) Jestliže je prvním argumentem prázdný seznam, pak musí být druhý a třetí argument stejný seznam `L`.

(2) Jestliže první argument relace `conc` je neprázdný seznam, pak musí mít hlavu a ocas a musí vypadat takto:

```
[X|L1]
```

Výsledkem spojení seznamu `[X|L1]` a nějakého seznamu `L2` je seznam `[X|L3]`, kde `L3` je spojením seznamů `L1` a `L2`.

Oba body vyjádříme v PROLOGu jako jeden fakt a jedno pravidlo:

```
conc([],L,L).
conc([X|L1],L2,[X|L3]) :- conc(L1,L2,L3).
```

Tento program již můžeme použít na spojení dvou daných seznamů, například:

```
?- conc([a,b,c],[1,2,3],L).

L=[a,b,c,1,2,3].
```

Přestože program `conc` vypadá poněkud jednoduše, může být flexibilně využit i jinými způsoby. Například můžeme `conc` využít obráceným způsobem pro rozklad daného seznamu na dva seznamy, například takto:

```
?- conc(L1,L2,[a,b,c]).

L1=[]
L2=[a,b,c];

L1=[a]
L2=[b,c];

L1=[a,b]
L2=[c];

L1=[a,b,c]
L2=[].
```

Je tedy možné seznam `[a,b,c]` rozložit čtyřmi různými způsoby, které všechny nalezl náš program `conc` přes backtracking.

Snadno si rozmyslíme, jak přidat (add) do seznamu položku na první místo. Nová položka se tak stane novou hlavou seznamu. Jestliže `X` je nová položka a `L` je seznam, ke kterému má být `X` přidáno, pak má výsledný seznam tvar:

```
[X|L]
```

Nepotřebujeme tedy proceduru pro přidávání nového prvku na začátek seznamu. Nicméně, kdybychom ji chtěli explicitně definovat, tak ji můžeme vytvořit jako následující fakt:

```
add(X,L,[X|L]).
```

Dále se budeme zabývat vymazáním (delete) prvku `X` ze seznamu `L`. Použijeme k tomu relaci

```
del(X,L,L1)
```

kde `L1` je rovno seznamu `L` bez prvku `X`. Relace `del` bude definována podobně jako relace `member`. Opět máme dva případy:

(1) Jestliže `X` je hlavou seznamu, pak výsledkem po smazání je ocas seznamu.

(2) Jestliže se prvek `X` vyskytuje v ocasu seznamu, pak je odtud smazán.

```
del(X,[X|Ocas],Ocas).
del(X,[Y|Ocas],[Y|Ocas1]) :- del(X,Ocas,Ocas1).
```

Podobně jako u výše zavedené operace náležení (`member`) je relace `del` nedeterministická. Pokud má `X` více výskytů v seznamu, pak je `del` schopna smazat každý z nich pomocí backtrackingu. Samozřejmě každé alternativní provedení smaže pouze jeden výskyt `X`, přičemž ostatní výskyty zůstanou zachovány. Například:

```
?- del(a,[a,b,a,a],L).

L=[b,a,a];
L=[a,b,a];
L=[a,b,a].
```

Relace `del` neuspěje, pokud seznam neobsahuje položku, která má být smazána, například:

```
?- del(x,[a,b,c,d],List).

false.
```

Relace `del` může být použita také jiným způsobem. Například, pokud chceme vložit prvek `a` do seznamu `[1,2,3]`, můžeme to udělat tak, že položíme dotaz: co je `L`, když po vymazání prvku `a` z `L` získáme seznam `[1,2,3]`?

```
?- del(a,L,[1,2,3]).

L=[a,1,2,3];
L=[1,a,2,3];
L=[1,2,a,3];
L=[1,2,3,a].
```

Operaci vložení (insert) prvku `X` na libovolné místo seznamu `Seznam` (dávající `VetsiSeznam`), lze zavést pravidlem:

```
insert(X,Seznam,VetsiSeznam) :-
  del(X,VetsiSeznam,Seznam).
```

Permutace

Občas je užitečné vygenerovat všechny permutace daného seznamu. Za tímto účelem definujeme relaci `perm` se dvěma argumenty. Argumenty jsou dva seznamy, takové, že jeden je permutací druhého. Záměrem je vytvářet permutace seznamu pomocí backtrackingu, jak je naznačeno v následujícím příkladu:

```
?- perm([a,b,c],P).

P=[a,b,c];
P=[a,c,b];
P=[b,a,c];
...
```

Program pro generování permutací bude opět založen na uvážení dvou případů souvisejících s prvním seznamem:

(1) Jestliže je první seznam prázdný, musí být i druhý seznam prázdný.

(2) Je-li první seznam neprázdný, tedy je ve tvaru `[X|L]`, pak nejprve vytvoříme permutace seznamu `L`, čímž obdržíme seznam `L1`, do kterého poté stačí vložit prvek `X` na každou možnou pozici.

Těmto dvěma případům odpovídají tyto dvě PROLOGovské klauzule:

```
perm([],[]).
perm([X|L],P) :- perm(L,L1), insert(X,L1,P).
```

Například pro dotaz

```
?- perm([cervena,modra,bila],P).
```

dostaneme jako výsledek všech šest permutací:

```
P=[cervena,modra,bila];
P=[cervena,bila,modra];
P=[modra,cervena,bila];
P=[modra,bila,cervena];
P=[bila,cervena,modra];
P=[bila,modra,cervena].
```

Pokud ale zadáme dotaz ve tvaru

```
?- perm(L,[a,b,c]).
```

dostane se program do nekonečné smyčky, Je proto na místě nezbytná obezřetnost.

Základy aritmetiky

V této části se krátce zaměříme na základní aritmetické operace a jejich vyhodnocování.

Nejprve si představíme některé předdefinované operátory:

`+`	sčítání
`-`	odčítání
`*`	násobení
`/`	dělení

`**`	mocnina
`//`	celočíselné dělení
`mod`	modulo, zbytek po celočíselném dělení.

Jedná se o výjimečný případ, kdy se operátor může chovat jako operace. V takových případech je však nutná další indikace k provádění aritmetických výpočtů. Následující dotaz je naivním pokusem o aritmetický výpočet:

```
?- X=1+2.
```

PROLOG totiž odpoví

```
X=1+2.
```

a ne `X=3`, jak jsme nejspíše očekávali. Důvod je jednoduchý: výraz `1+2` pouze označuje PROLOGovský term, kde `+` je funktor a `1` a `2` jsou jeho argumenty. V posledním dotazu není nic, co by iniciovalo výpočet (aktivovalo operaci sčítání). Tento problém řeší předdefinovaný operátor `is`. Právě operátor `is` vynucuje vyhodnocení. Správný způsob k vyvolání vyhodnocení aritmetického výrazu je:

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

Nyní bude odpověď

```
X=3.
```

Sčítání zde bylo provedeno zvláštním postupem, pomocí speciální procedury[13)], která je spojena (asociována) s operátorem `is`. Podobně fungují i další výše uvedené předdefinované operátory. Vyzkoušejme některé položením složeného dotazu:

```
?- X is 5/2, Y is 5//2, Z is 5 mod 2.

X=2.5,
Y=2,
Z=1.
```

[13)] Takové procedury nazýváme vestavěné, anglicky built-in procedures.

Levým argumentem operátoru `is` je jednoduchý objekt. Pravým argumentem je aritmetický výraz složený z aritmetických operátorů, čísel a proměnných (jejichž hodnoty musí být při provádění výpočtu známy).

V PROLOGu také můžeme porovnávat aritmetické výrazy, jako například, zda-li je součin čísel `277` a `37` větší než `10000`:

```
?- 277*37>10000.

true.
```

Poznamenejme, že podobně jako `is`, operátor „`>`“ vynutí vyhodnocování.

Představme si operátory pro porovnávání:

`X>Y`	`X` je větší než `Y`
`X<Y`	`X` je menší než `Y`
`X>=Y`	`X` je větší nebo rovno než `Y`
`X<=Y`	`X` je menší nebo rovno než `Y`
`X=:=Y`	hodnoty `X` a `Y` jsou stejné
`X=\=Y`	hodnoty `X` a `Y` nejsou stejné.

Dodejme ještě, že operátor „`=`“ se podstatně liší od operátoru „`=:=`“. Rozdíly si nejprve demonstrujme na konkrétních příkladech:

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

true.

?- 1+2=2+1.

false.

?- 1+A=B+2.

A=2,
B=1.
```

Například, máme-li dotazy ve tvaru „`X=Y`“ a „`X=:=Y`“, pak první z nich způsobuje porovnání objektů `X` a `Y` a může (pokud se `X` a `Y` shodují) konkretizovat hodnoty proměnných, které se v nich vyskytují. Na druhou stranu

druhý operátor „=:=“ způsobuje aritmetické vyhodnocení a nemůže být nikdy použit ke konkretizaci hodnot proměnných.

Dále si ukážeme dva příklady, na kterých demonstrujeme použití aritmetických operací. První bude výpočet faktoriálu, zatímco druhý bude sloužit k určení počtu položek daného seznamu.

Uvažme následující program v PROLOGu:

```
factorial(0,1).
factorial(N,X) :-
  N >= 1,
  M is N-1,
  factorial(M,R),
  X is R*N.
```

Jak snadno ověříte, daný program umožňuje pro dané nezáporné celé číslo n vypočítat jeho faktoriál, tedy hodnotu $n!$. Například na dotaz `factorial(6,X).` vrátí PROLOG odpověď `X=720`.

Vyzkoušejte, zda si PROLOG poradí i s velkou hodnotou, například položením následujícího dotazu:

```
factorial(20000,X).
```

Druhým příkladem je určení délky seznamu. I zde se nám bude hodit aritmetika. Budeme totiž počítat počet položek v seznamu. K tomuto účelu definujme proceduru `delka(Seznam,N)` se dvěma argumenty, která bude počítat prvky v seznamu `Seznam` a v `N` zaznamenávat jejich počet. Jistě bude užitečné uvažovat dva případy:

(1) Jestliže je seznam prázdný, pak je jeho délka `0`.

(2) Je-li seznam neprázdný, tedy je ve tvaru `Seznam=[Hlava|Ocas]`, pak je jeho délka rovna číslu `1` plus délka ocasu `Ocas`.

Tyto dva případy korespondují s následujícím programem:

```
delka([],0).

delka([_|Ocas],N) :-
  delka(Ocas,N1),
  N is 1+N1.
```

Program je hotov. Zkuste si sami rozmyslet, co by se stalo, kdybychom

v něm prohodili poslední dva řádky. My program vyzkoušíme položením jednoho konkrétního dotazu:

```
?- delka([a,b,[c,d],e],N).

N=4.
```

Závěrem

PROLOG je programovací jazyk zvláště vhodný pro problémy, které zahrnují objekty a vztahy mezi nimi. Silným nástrojem pro vytváření programů v PROLOGu je rekurze. Programy v jazyku PROLOG se skládají z výrazů, které tvoří fakta a pravidla. Zvláštní výrazy, které nejsou přímou součástí programů, jsou dotazy, někdy nazývané cíle. Programy v PROLOGu slouží k vyjádření (popisu) naší znalostní báze. Programy píšeme v roli „programátorů“, pomocí cílů aktivujeme výpočet, přičemž cíle zadáváme v roli „uživatelů“ vytvořeného programu.

Seznámili jsme se se seznamy, v PROLOGu často používanými datovými strukturami. Víme, že seznam je buď prázdný, nebo sestává z „hlavy“ a „ocasu“, který je také seznamem. V souvislosti s operacemi se seznamy, umíme naprogramovat relaci pro: náležení do seznamu, spojení dvou seznamů, přidání prvku do seznamu, smazání prvku ze seznamu. Dozvěděli jsme se, že aritmetika je v PROLOGu spjata s vestavěnými procedurami. Vyhodnocení[14] aritmetického výrazu je zajištěno přes proceduru `is` a také pomocí operátorů pro porovnávání: `<`, `<=` atd. Poznali jsme také vestavěné procedury asociované s předdefinovanými operátory: `+`, `-`, `*`, `/` a podobně.

Z prostorových důvodů jsme nepředstavili pokročilejší techniky, které umožňují ovlivňovat výpočetní proces, například tzv. řezy s jejichž pomocí je snadné naprogramovat cykly, podmíněné výrazy a pracovat s negací. Dodejme, že PROLOG je plnohodnotným programovacím jazykem, ve kterém lze naprogramovat veškeré algoritmy (třeba algoritmy třídící či nějaký složitý expertní systém).

Literatura

[1] *Bratko, I.*: PROLOG Programming for Artificial Intelligence (4th Edition). Pearson Education Canada, 2011.

[14] Při vyhodnocování aritmetického výrazu musí mít všechny argumenty číselné hodnoty.