Prolog - úvod a program na rodinné vztahy

Program v prologu je v zásadě databáze faktů a vztahů mezi nimi napsaná pomocí Hornovských klauzulí. Každá klauzule je napsaná ve tvaru `H:-B.`, kde `H` je hlava klauzule a `B` je její tělo. Můžete si to představovat i tak, že `:-` je šipka doleva (tedy implikace zprava doleva), a přesně takový je také význam klauzule. Klauzule mohou mít i prázdné tělo. Potom se píše jen `H`. Například následující 4 klauzule říkají, že (postupně), `honza`, `adam`, `david` a `jiri` jsou muži.

```
muz(honza).
muz(adam).
muz(david).
muz(jiri).
```

Stejným způsobem další klauzule říkají, že `jitka`, `jana` a `lida` jsou ženy.

```
zena(jitka).
zena(jana).
zena(lida).
```

Klauzule mohou mít i více než jeden parametr. Následující klauzule vyjadřuje, ze `honza` je potomek `adam`a. Podobně pro další klauzule.

```
%potomek(X, Y)
%X je potomek Y
potomek(honza, adam).
potomek(adam, david).
potomek(jiri, adam).

potomek(jitka, adam).
potomek(jitka, jana).
```

Pravidla, která platí mezi jednotlivými klauzulemi, se dají napsat pomoci klauzulí s tělem. Například fakt, ze když je `X` potomek `Y`, a zároveň je `X` žena, tak `X` je dcera `Y`. V logice bychom to mohli zapsat jako `zena(X) & potomek(X,Y) -> dcera(X,Y)`. V Prologu napíšeme to samé, jen pomocí prologovské syntaxe, tedy:

```
dcera(X,Y):-zena(X),potomek(X,Y).
syn(X,Y):-muz(X),potomek(X,Y).
```

Všimněte si, ze jména konstant začínají malým písmenem a jména proměnných velkým. Podobně jako jsme právě nadefinovali `dcera` a `syn`, můžeme definovat i další rodinné vztahy.

```
deda(X,Y):-potomek(Y,Z),potomek(Z,X),muz(X).
bratr(X,Y):-potomek(X,Z),potomek(Y,Z),X \= Y,muz(X).
```

K čemu je zápis `X\=Y` v definici `bratr`? `X\=Y` znamená, že `X` a `Y` nejsou unifikovatelné. Bez tohoto atomu, bychom na dotaz `bratr(jiri, X)`, dostali i odpověď, ze `jiri` je bratr sám sebe. A tím se dostáváme k dotazům a k tomu, co se Prologem dá vlastně dělat.

Úkol: Zkuste si pohrát s příklady v tomto textu, přidejte si další lidi, nadefinujte si například predikáty `teta`, `babicka`, atd.

Tento soubor si můžete i stáhnout jako platný kód v Prologu. Pusťte si interpret Prologu, např. SWI-Prolog a zkuste tento soubor načíst (`consult(cv1).` – musíte být ve správném adresáři, pokud v něm nejste, použijte buď `chdir('cesta/s/lomitky/dopredu')`, nebo `working_directory(_,'cesta/s/lomitky/dopredu/')`. Funguje i zkratka `[cv1].` a pokud máte příponu `.pl` asociovanou s Prologem, stačí na soubor normálně kliknout.

Nyní už můžete pokládat dotazy.

Dotazy se píší přímo v interpretu prologu, za prompt `?-` (odpovědi Prologu jsou potom na samostatných řádkách). Nejjednodušší dotaz je například:

```
?- muz(jiri).
true.
```

Prolog odpovídá `true`, tzn. že byl schopen najít v databázi (nebo z ní odvodit), že dotaz je pravdivý. Podobně, když položíte dotaz:

```
?- dcera(jitka, jana).
true.
```

Prolog zase odpoví `true`, protože byl z databáze schopen odvodit, že `jitka` je dcera `jany`.

Můžete ale pokládat i složitější dotazy, které obsahuji proměnné, ty v Prologu musí začínat velkým písmenem:

```
?- dcera(X, jana).
X = jitka ;
false.
```

V tomto případě vám Prolog odpoví, ze `X` je Jana, a čeká, co budete dělat. Když zmáčknete středník, pokusí se najít další `X`, které odpovídá dotazu, když se mu to nepovede, odpoví `false`.

Dotaz nemusí obsahovat vůbec žádné konstanty. V takovém případě Prolog najde všechna možná ohodnoceni proměnných v dotazu:

```
?- syn(X,Y).
X = honza,
Y = adam ;
X = adam,
Y = david ;
X = jiri,
Y = adam.
```

Úkol: Zkuste si trochu rozšířit naši databázi vztahů a napsat predikát `predek(X, Y)`, který bude splněný v případě, že `X` je předek `Y`.

Mgr. Martin Pilát, Ph.D.
Malostranské náměstí 25
118 00 Praha
Martin.Pilat@mff.cuni.cz

Prolog - numerály a seznamy

Dnes si zkusíme napsat trochu složitější programy v Prologu, na kterých si můžeme vyzkoušet různé složitější věci, například si ukázat obousměrnost výpočtů. Poslouží nám k tomu Peanova aritmetika. V té máme jedinou konstantu `0`, a jedinou funkci `S(x)`, která vrací následníka `x`. Dají se v ní potom nadefinovat numerály (čísla) tak, že numerál `n` je `n`-krát aplikovaná operace `S` na `0`.

Napišme napřed proceduru pro součet dvou numerálů.

```
% soucet(X,Y,Z)
% implementuje scitani numeralu, Z = X + Y
% prvni klauzule: 0+Y=Y
% druha klauzule: s(X)+Y=s(X+Y)
soucet(0,Y,Y).
soucet(s(X), Y, s(Z1)):-soucet(X,Y,Z1).
```

Pohrajte si s touto procedurou. Co se stane, když dáte na různá místa proměnnou místo numerálu? Co třeba dělá `soucet(X, s(0), s(s(0)))`? A co `soucet(X,Y,s(s(s(0))))`?

Napište predikát, který vytvoří dvojnásobek zadaného numerálu, a predikát, který rozhodne jestli daný numerál reprezentuje sudé číslo.

```
dvakrat(X,Y):-soucet(X,X,Y).
sude(X):-soucet(Z,Z,X).
```

Zkuste napsat další predikáty pro práci s numerály – součin a mocninu. Napište predikát, který rozhodne, jestli zadaný term je číslo.

```
% soucin(X,Y,Z) -- spocita XY a vrati vysledek v Z
soucin(_,0,0).
soucin(0,_,0).
soucin(X, s(Y), Z2):-soucin(X,Y,Z1),soucet(Z1,X,Z2).
```

Umí předcházející procedura `soucin/3`, rozkládat čísla? Co se stane, když zavoláte `soucin(X,Y,s(s(s(s(s(0))))))`. Proč? Upravte `soucin/3` tak, aby v tomhle případě fungoval.

```
% lt(X,Y) -- vrati true, kdyz X < Y
lt(X,s(X)).
lt(X,s(Y)):-lt(X,Y).

% cislo(X) -- rozhoduje, zda X je cislo.
cislo(s(X)):-cislo(X).
cislo(0).

soucin1(_, 0, 0).
soucin1(0, _, 0).
soucin1(s(X), Y, Z):-soucet(A, Y, Z), soucin1(X, Y, A).
```

Jak byste v prologu reprezentovali datovou strukturu? Co třeba struktura pro datum?

Prolog umí pracovat také se seznamy, ale na rozdíl od jiných jazyků neumí přistupovat rychle na dané místo v seznamu. Přístup trvá lineárně dlouho.

Seznamy se píšou do hranatých závorek, jejich prvky se oddělují čárkou např. `[1,2,3]` je seznam čísel `1`, `2` a `3`; `[1,2,[3,4]]` je seznam, který obsahuje čísla `1`,`2` a seznam `[3,4]`. Prázdný seznam se zapíše jako `[]`. Seznam se dá rozdělit na první prvek a zbytek pomocí `[X|Xs]`, `X` potom obsahuje první prvek seznamu a `Xs` zbytek. Podobně je možné získat i první dva prvky seznamu a zbytek jako `[X1, X2 | Xs]`. Kratší než dvouprvkový seznam se s tímto termem neunifikuje.

Užitečnou procedurou pro práci se seznamy je `prvek(X,S)`, která říká, jestli `X` je prvek seznamu `S`. Ve většině verzí tento predikát najdete jako `member/2`.

```
% prvek(?X, +S)
prvek(X,[X|_]).
prvek(X,[_|S]):-prvek(X,S).
```

Procedura se podívá, jestli je první prvek seznamu unifikovatelný s `X`, pokud ano, uspěje. Druhá řádka potom řeší situaci, kdy `X` není unifikovatelné s prvním prvkem seznamu, v takovém případě se procedura volá rekurzivně na zbytek seznamu.

Můžeme napsat i proceduru, která připojí prvek na začátek seznamu. Je to hodně jednoduché, stačí výsledný seznam unifikovat se seznamem, který má na prvním místě `X` a zbytek je `S`.

```
%pridejz(+X, +S, -Z). Pridava X na zacatek seznamu S a vysledek vraci v Z
pridejz(X, S, [X|S]).
```

Jak je to s přidáním prvku v seznamu na konec? Jak by taková procedura vypadala? A jaká bude její složitost? Je triviální přidat prvek na konec prázdného seznamu, stačí vytvořit jednoprvkový seznam s tímto prvkem. Pro přidání na konec delšího seznamu vezmeme jeho první prvek, zkopírujeme ho do výsledného seznamu a zavoláme proceduru rekurzivně.

```
% pridejk(+X, +S, -Z). Pridava X na konec seznamu S a vysledek vraci v Z.
pridejk(X,[],[X]).
pridejk(X,[Y|Ys],[Y|Z]):-pridejk(X,Ys,Z).
```

Všimněte si, že na začátek seznamu umíme přidávat v konstantním čase, kdežto na konec seznamu v čase lineárním v délce seznamu.

U obou předchozích predikátů jste si mohli všimnout znaků `+/-` v komentáři. Tyto znaky se používají pro rozlišení vstupních a výstupních proměnných – `'+'` označuje vstupní proměnnou, `'-'` výstupní. Při volání se potom očekává, že vstupní proměnné mají konkrétní hodnotu a výstupní jsou volné. Můžete se setkat i se znakem `'?'`, ten znamená, že proměnná může být vstupní i výstupní. Toto je důležité při čtení a psaní dokumentace - pokud zavoláte predikát jinak, než je popsáno, může se zacyklit, nebo vracet nesprávné výsledky. (Celý tento systém notace je mnohem složitější, můžete se podívat do dokumentace, nám bude stačit zjednodušená verze popsaná výše.)

Zkusme si nyní napsat otočení seznamu pozpátku.

```
%otoc(+S,-Z). Otoci seznam S a vrati vysledek v Z
otoc([],[]). % otoceni prazdeho senzamu je prazdny seznam
otoc([X|Xs],Y):-otoc(Xs,Z),pridejk(X,Z,Y). % kdyz chces otocit [H|T] otoc T a pripoj za to H
```

Takhle napsané otočení seznamu je neefektivní, trvá kvadraticky dlouhou dobu (n-krát se volá `pridejk` na seznamy délky $1 \ldots n$). Otočení ale i v Prologu lze napsat v lineárním čase, stačí využít jednoduchou techniku akumulátoru. Akumulátor je proměnná, do které si postupně ukládáme mezivýsledky a (typicky) v posledním kroku rekurze ji překopírujeme do výsledku.

S pomocí akumulátoru můžeme napsat proceduru `otoc2/2`, která seznam otočí v lineárním čase. Použije k tomu pomocnou proceduru `otoc/3`, která navíc používá akumulátor. Ten je na začátku prázdný.

```
%otoc2(+S,-Z). Otoci seznam S a vrati vysledek v Z.
otoc2(S,Z):-otoc2(S,[],Z).

%otoc2(+S, @A, -Z). Otoci seznam S a vrati vysledek v Z, pouziva A jako akumulator.
otoc2([], A, A). % Seznam uz je prazdny, prekopiruj akumulator
otoc2([X|Xs], A, Z):-otoc2(Xs, [X|A], Z).       % Prvni prvek z [X|Xs] pripoj na zacatek
                                                % akumulatoru a rekurzivne se zavolej.
```

Co se děje v tomto případě? Místo, abychom přidávali prvky na konec výsledného seznamu, přidáváme je na začátek akumulátoru. Když se vstupní seznam vyprázdní, zkopírujeme akumulátor do výsledku. Jak tedy probíhá výpočet pro dotaz `otoc2([1,2,3,4], Z)`? Napřed se zavolá pomocná procedura `otoc2([1,2,3,4],[],Z)` a potom postupně `otoc2([2,3,4],[1],Z)`, `otoc2([3,4],[2,1],Z)`, `otoc2([4],[3,2,1],Z)` a `otoc2([], [4,3,2,1], Z)`. V tuto chvíli se zavolá první klauzule procedury `otoc2/3` a `Z` se unifikuje s akumulátorem `[4,3,2,1]`. Potom už se program jen vrátí s unifikací `Z = [4,3,2,1]`. Všimněte si, že v tomto případě program běžel v době, která je lineární v délce seznamu, a že jsme toho dosáhli tím, že jsme přidávání na konec seznamu nahradili přidáváním na začátek akumulátoru, který jsme nakonec překopírovali.

Promyslete si další operace pro práci se seznamy: `sudy`, `lichy` (říkají, jestli má seznam sudou nebo lichou délku), `prefix` a `sufix` (vrací postupně prefixy, nebo sufixy seznamu) atd.

```
sudy([]).
sudy([_,_|S]):-sudy(S).

lichy([_|S]):-sudy(S).

prefix([],_).
prefix([X|Xs], [X|Ys]):-prefix(Xs,Ys).

sufix(X,Y):-otoc2(Y,Y1),prefix(X1,Y1),otoc2(X1,X).
```

Mgr. Martin Pilát, Ph.D.
Malostranské náměstí 25
118 00 Praha
Martin.Pilat@mff.cuni.cz

Prolog - seznamy a aritmetika

Minule jsme se věnovali seznamům, dnes v tom budeme ještě pokračovat, přidáme ale i aritmetiku.

Začněme jednoduchým příkladem na odebrání prvku ze seznamu.

```
% vymaz(+X, +X, -V) :- vymaze vyskyt prvku X ze seznamu S
%                      a vrati vysledny seznam jako V
vymaz(X,[X|Xs],Xs).
vymaz(X,[Z|Xs],[Z|Y]):-vymaz(X,Xs,Y).
```

Občas se hodí, aby procedura uspěla i v případě, že seznam daný prvek neobsahuje

```
% vymaz1(X,Odkud,CoZbude) :- totez co vymaz
%                            jen uspeje i kdyz X neni prvkem seznamu Odkud.
vymaz1(_,[],[]).
vymaz1(X,[X|T],T):-!.
vymaz1(X,[H|T],[H|T1]) :- vymaz1(X,T,T1).
```

Všimněte si, že pokud nenapíšete `!` na konec první klauzule, Prolog vrací i seznamy, ve kterých jsou odebrané jiné výskyty než ten první a nakonec vrátí i původní seznam. `!` je operátor řezu a říká Prologu, že zvolená rozhodnutí má považovat za konečná. Zakazuje mu backtrackovat v daném místě - ořezává všechny větve výpočtu kromě té první, proto ten název “operátor řezu”.

Zkusme si ještě napsat proceduru, která vrátí prostřední prvek ze seznamu. Jak ji napíšete, když nevíte dopředu, jak je seznam dlouhý?

```
prostredni(S,X):- prostredni(S,S,X).
prostredni([_],[X|_],X).
prostredni([_,_],[X|_],X).
prostredni([_,_|T1],[_|T2],X):- prostredni(T1,T2,X).
```

Jak procedura funguje? Všimněte si, že v prvním seznamu se jde vždy po dvou a ve druhém po jednom kroku. Když se tedy první seznam vyprázdni (má max 2 prvky), na začátku toho druhého je přesně prostřední prvek.

Nakonec napišme ještě proceduru, která spočítá průnik dvou seznamů (ta je těžší a potřebujete na ni operátor řezu).

```
prunik([],_,[]).
prunik(_,[],[]).
prunik([X|Xs], Y, [X|Z]):-member(X,Y), prunik(Xs,Y,Z), !.
prunik([_|Xs], Y, Z):-prunik(Xs,Y,Z).
```

Úkol: Zvládli byste napsat sjednocení? Zkuste si to.

Podívejme se nyní na aritmetiku v Prologu a na počítání. Počítáni v Prologu má svá úskalí a liší se od počítání v jazycích, na jaké jste zvyklí. Část těchto problémů plyne z toho, že v Prologu neexistuje operator přiřazení, `=` dělá unifikaci. Místo `=` se v aritmetických výpočtech v Prologu používá operátor `is`.

Rozdíl mezi `=` a `is` je vidět na následujících dotazech a odpovědích Prologu.

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

? X is 1+2
X = 3.
```

V Prologu také existují obvyklé relační operátory: rovná se se zapíše jako `=:=`, nerovná se jako `=\=`, menší než a větší než jako `<` a `>`, případně menší nebo rovno a větší nebo rovno jako `=<` a `>=`. Dávejte si pozor na to, že některé z těchto operátorů se zapisují jinak, než jste asi zvyklí.

U všech relačních operátorů je potřeba, aby obě strany výrazu už měly přiřazenu konkrétní hodnotu. U operátoru `is` stačí, když je přiřazena konkrétní hodnota termům na pravé straně (pokud je přiřazena konkrétní hodnota i termům na levé straně, operátor uspěje, pokud se obě hodnoty rovnají, jinak selže).

S pomocí aritmetiky můžeme napsat několik jednoduchých procedur. Nejjednodušší z nich je výpočet délky seznamu

```
% delka(+S,-L) :- do L dosadi delku seznamu S
delka([], 0).
delka([_|T], N):-delka(T,N1),N is N1 + 1.
```

Všimněte si, že výpočet `N` a rekurzivní volání nejde prohodit, protože jinak `N1` ještě nemá přiřazenou žádnou hodnotu.

Můžeme také napsat jednoduchou proceduru pro výpočet druhé mocniny zadaného čísla.

```
% nadruhou(+X, -Y) :- prirazuje do Y druhou mocninu X
nadruhou(X,Y):-Y is X*X.
```

Zajímavější ale je napsat proceduru, která spočítá libovolnou mocninu libovolného čísla.

```
% mocnina(+Z,+M,-V) :- pocita Z^M a vysledek ulozi do V
mocnina(_,M,_):-M<0,fail.
mocnina(_,0,1).
mocnina(Z,1,Z).
mocnina(Z,2,V):-nadruhou(Z,V).
mocnina(Z,M,V):-M>2, M1 is M - 1, mocnina(Z,M1,V1), V is V1*Z.
```

Za použití aritmetiky se dají napsat i další procedury pro práci se seznamy, například nalezení minima v seznamu. Všimněte si tady i použití akumulátoru.

```
% minseznam(+S, -M) :- najde minimum v seznamu S a ulozi ho do M
minseznam([H|T], M) :- minseznam(T, H, M).
minseznam([], A, A).
minseznam([H|T], A, M):-A=<H,minseznam(T,A,M).
minseznam([H|T], A, M):-H<A,minseznam(T,H,M).
```

Můžeme napsat i přístup k `N`-tému prvku v seznamu. Dejte si ale pozor, přístup trvá lineárně dlouho.

```
% nty(+N, +S, +X) :- do X ulozi N-ty prvek seznamu S
nty(1, [H|_], H).
nty(N, [_|T], X):-N1 is N-1,nty(N1,T,X).
```

Mgr. Martin Pilát, Ph.D.
Malostranské náměstí 25
118 00 Praha
Martin.Pilat@mff.cuni.cz

Prolog - prohledávání, rozdílové seznamy

Většina programů, které jsme zatím v Prologu napsali, se chovala deterministicky, tj. v každém kroku byla jen jedna (rozumná) možnost, co udělat, a tedy jsme se snažili, aby Prolog právě tuto variantu zvolil. V Prologu je ale mnohem běžnější používat prohledávání a programy jsou tedy často nedeterministické - Prolog zkouší různé klauzule z definice procedury dokud nenajde správné řešení.

Prohledávání

Jako příklad nedeterministického programu můžeme vyřešit známý problém loupežníků - vybrat ze seznamu čísla, jejichž součet se rovná zadanému číslu. Jedno z řešení může vypadat například následujícím způsobem.

```
% vyber(S, N, Z) :- vybere ze seznamu S seznam Z jehoz soucet je presne N
vyber([], N, _):-N>0, fail.
vyber([], 0, []).
vyber([H|T], N, [H|Z]):-N1 is N-H,vyber(T,N1,Z).
vyber([_|T],N,Z):-vyber(T,N,Z).
```

Poslední dvě klauzule v tomto řešení říkají, že buď první číslo seznamu máme dát (druhá řádka od konce), nebo nemáme (poslední řádka). Prolog tedy napřed zkusí do seznamu číslo dát, a vyřešit nový problém s kratším seznamem a menším číslem, když se to nepovede, selže a zkusí variantu, kde tam číslo nedá.

Program můžeme i zjednodušit. Poslední dvě řádky v sobě vlastně obsahují predikát `member`. Dají se napsat i pomocí něj, nebo pomocí predikátu `select`, který funguje podobně, ale kromě samotného prvku ze seznamu vrací i seznam bez tohoto prvku. Druhá varianta řešení tedy může vypadat například takto:

```
vyber2(X, N, Z):-vyber2(X, N, 0, [], Z).
vyber2(_, N, N, Z, Z).
vyber2(X, N, A, Z, V):-select(P, X, Xs),
                       B is A + P,
                       B =< N,
                       vyber2(Xs, N, B, [P|Z], V).
```

Třídění

Pro implementaci quicksortu se hodí procedura, která rozdělí seznam na dva, v jednom nich jsou čísla menší než zvolené číslo, ve druhém větší.

```
% rozdel(+S, +Pivot, -Mensi, -Vetsi) :- rozdeli S tak, ze v seznamu Mensi jsou
%       prvky mensi nebo rovny Pivotu a v seznamu vetai jsou prvky vetai nez Pivot

rozdel([], _, [], []).
rozdel([H|T], Pivot, [H|Mensi], Vetsi):-H=<Pivot,rozdel(T,Pivot,Mensi,Vetsi).
rozdel([H|T], Pivot, Mensi, [H|Vetsi]):-H>Pivot,rozdel(T,Pivot,Mensi,Vetsi).
```

Abychom mohli napsat quicksort, potřebujeme ještě napsat spojení dvou seznamů. To by mělo být jednoduché. (V Prologu je spojení také definované jako procedura `append`):

```
% spojeni(?X, ?Y, ?Z) :- do seznamu Z ulozi spojeni seznamu X a Y
spojeni([],X,X).
spojeni([H|T], X, [H|S]):-spojeni(T,X,S).
```

A teď už máme všechno a můžeme klidně napsat quicksort. `T\=[]` v poslední klauzuli zajišťuje, že Prolog nebude vracet stejné odpovědi vícekrát, ale není ho tam nutné mít (všechny vrácené odpovědi jsou správné, dalo by se mu vyhnout i tak, že bychom chtěli, aby seznam měl aspoň dva prvky `[H1, H2|T]`).

```
% qsort(+X,-Y) :- do seznamu Y ulozi setrideny seznam X
qsort([],[]).
qsort([A],[A]).
qsort([H|T],X):-T\=[],rozdel(T,H,Mensi,Vetsi),
                qsort(Mensi,X1),qsort(Vetsi,X2),
                spojeni(X1,[H|X2],X).
```

Rozdílové seznamy

Při implementaci quicksortu jsme potřebovali zřetězovat seznamy. Zřetězení seznamů trvá dlouho (lineárně v délce prvního seznamu), v Prologu ale můžeme použít trik, kterému se říká rozdílový seznam. Spojeni rozdílových seznamů potom jde napsat v konstantním čase.

Jak ten trik funguje? Místo použití pouze samostatného seznamu použijeme seznam a volnou proměnnou. Běžně se tyto dvě části oddělují symbolem `-` (proto se jim asi říká rozdílové seznamy), můžete si je ale klidně oddělit i jinak, případně si nadefinovat vlastní datovou strukturu (binární term).

Rozdílový seznam tedy má dvě části, samotný seznam a volnou proměnnou, např. `S-X`. Tohle ale samo o sobě nestačí, důležité je, že ta proměnná `X` je zároveň použita jako konec seznamu `S`. Takový rozdílový seznam například může vypadat následujícím způsobem `[a,b,c|X]-X`.

Napsat převod seznamu na rozdílový seznam je snadné (v lineárním čase).

```
% narozdil(+S,-RS) :- prevadi seznam S na rozdilovy seznam RS
narozdil([],X-X).
narozdil([H|T],[H|S]-X):-narozdil(T,S-X).
```

Převod z rozdílového seznamu na obyčejný seznam je ještě jednodušší (a dokonce v konstantním čase).

```
% naobyc(+RS,S):- prevadi rozdilovy seznam RS na obycejny seznam S
naobyc(X-[],X).
```

Když nyní v rozdílovém seznamu `[a,b,c|X]-X` unifikujeme `X` s libovolným seznamem (např. `[d,e]`), dostaneme `[a,b,c,d,e]-[d,e]` (v jednom kroku) a vidíte, ze máme spojení seznamu (+ nějaký ocásek navíc). Ještě zajímavější situace by nastala, kdybychom spojili náš původní seznam s rozdílovým seznamem, např. s `[d,e|Y]-Y`, a dostali bychom `[a,b,c,d,e|Y]-Y` - tedy zase rozdílový seznam. Takové spojení se ale dá opět napsat velmi snadno:

```
%spojeniRS(+A,+B,-C):-rozdilovy seznam C je spojeni rozdilovych seznamu A a B
spojeniRS(A-B,B-B1,A-B1).
```

Poslední úkol pro dnešek (víceméně nesouvisející s předcházejícím) je napsání transpozice matice. Matice je zadaná jako seznam seznamů. Pro transpozici (tuhle implementaci) potřebujeme pomocnou proceduru, která ze seznamu seznamů odebere první prvky a vrátí zbytky.

```
%vsePrazdne(+SeznamSeznamu):-vsechny seznamy v SeznamSeznamu jsou prazdne
vsePrazdne([]).
vsePrazdne([[]|Z]):-vsePrazdne(Z).

%hlavyZbytky(+SeznamSeznamu, -Hlavy, -Zbytky):-rozdeli vsechny seznamu v SeznamSeznamu na Hlavy a Zbytky
hlavyZbytky([], [], []).
hlavyZbytky([[H|T] | Z], [H|PP], [T|ZB]):-hlavyZbytky(Z, PP, ZB).

%transpozice(+M, -TM):-TM je transpozice matice M
transpozice(M, []):-vsePrazdne(M).
transpozice(M, [H|TM]):-hlavyZbytky(M, H, Z), transpozice(Z, TM).
```

Mgr. Martin Pilát, Ph.D.
Malostranské náměstí 25
118 00 Praha
Martin.Pilat@mff.cuni.cz

Prolog - stromy

V Prologu lze pracovat i se složitějšími datovými strukturami. Např. s binárními stromy se pracuje tak, že si vytvoříme term `t(LevySyn, Uzel, PravySyn)`, který reprezentuje uzel a jeho dva syny. Prázdný uzel můžeme reprezentovat jako `nil` (`nil` není žádné zvláštní klíčové slovo Prologu, je to jen nulární term, kdybychom použili třeba `nic` bude to fungovat úplně stejně).

Strom, který má kořen 1, a dva syny 2 a 3 se tedy dá zapsat jako `t(t(nil,2,nil),1,t(nil,3,nil))`. Když levého syna (2) nahradíme podstromem `t(t(nil, 4, nil),2, t(nil,5, nil))` zapíšeme výsledek jako `t(t(t(nil,4,nil),2,t(nil,5,nil)),1,t(nil,3,nil))`.

Zkusme si v Prologu práci s binárními stromy. Nejjednodušší jsou asi průchody stromem. Začneme tedy s převedením stromu na seznam v inorder pořadí.

```
% inorderList(+T,-S) :- prevadi strom T na seznam S v inorder poradi
inorderList(nil,[]).
inorderList(t(X,Y,Z), S):-inorderList(X,S1),inorderList(Z,S2),append(S1,[Y|S2],S).

% preorder(+T,-S) :- prevadi strom T na seznam S v preorder poradi
preorderList(nil,[]).
preorderList(t(X,Y,Z), [Y|S]):-preorderList(X,S1),preorderList(Z,S2),append(S1,S2,S).

% postorder(+T,-S) :- prevadi strom T na seznam S v postorder poradi
postorderList(nil,[]).
postorderList(t(X,Y,Z), S):-postorderList(X,S1),postorderList(Z,S2),append(S1,S2,S3),append(S3,[Y],S).
```

Občas se místo seznamu prvků ze stromu hodí další prvek vracet při stisknuti `;` (nebo při selhání výpočtu a hledání dalšího ohodnocení).

```
inorder(t(L,_,_),X):-inorder(L,X).
inorder(t(_,X,_),X).
inorder(t(_,_,R),X):-inorder(R,X).

preorder(t(_,X,_),X).
preorder(t(L,_,_),X):-preorder(L,X).
preorder(t(_,_,R),X):-preorder(R,X).

postorder(t(L,_,_),X):-postorder(L,X).
postorder(t(_,_,R),X):-postorder(R,X).
postorder(t(_,X,_),X).
```

Samozřejmě je možné vytvořit i binární vyhledávací stromy. Pro to potřebujeme proceduru pro přidání do BST.

```
% pridejBST(+T,+X,+S) :- prida prvek X do stromu T a vrati strom S
pridejBST(nil, X, t(nil,X,nil)).
%pridejBST(t(L,X,P),X,t(L,X,P)).  % odkomentovani tehle radky zpusobi, ze se prvky nemohou opakovat
pridejBST(t(L,U,P),X,t(T,U,P)):-X=<U,pridejBST(L,X,T). % tohle je pak treba zmenit na X<U
pridejBST(t(L,U,P),X,t(L,U,T)):-X>U,pridejBST(P,X,T).
```

Když už umíme přidávat do BST, tak samozřejmě umíme i postavit BST ze seznamu (níže jsou dvě možné varianty).

```
% seznamNaBST(+S,-T) :- vytvori BST T ze seznamu S
seznamNaBST(S,T):-seznamNaBST(S,nil,T).
seznamNaBST([],T,T).
seznamNaBST([H|Hs],T1,T):-pridejBST(T1,H,T2),seznamNaBST(Hs,T2,T).

sB([], nil).
sB([X], t(nil,X,nil)).
sB([H|T], V):-sB(T,V1),pridejBST(H,V,V1).
```

Pomocí převedení na seznam a inorder průchodu lze samozřejmě seznam i setřídit:

```
% bstSort(+Seznam,-SetridenySeznam) :- tridi Seznam pomoci prevodu na BST a vraci
%                                                                    SetridenySeznam
bstSort(S,T):-seznamNaBST(S,T1),inorderList(T1,T).
```

Úkol: Pokuste se nyní napsat převod stromu na seznam tak, že v něm budou prvky uloženy po patrech. Tj. vlastně udělat průchod stromem do šířky. Jak byste takový program napsali? Hodí se vám na to rozdílové seznamy z minulého cvičení.

Stromy se ale dají používat i jinak, než jen pro BST. Můžeme jimi například reprezentovat výrazy. V uzlech jsou potom operátory v listech (uzlech, které obsahují dva `nil`) jsou potom hodnoty. Zkusme si napsat vyhodnocení takového stromu:

```
%vyhodnot(Strom, Hodnota):-vyhodnooti vyraz zadany jako Strom a vysledek vrati jako Hodnota
vyhodnot(t(nil, H, nil), H).
vyhodnot(t(LP, *, PP), H):-vyhodnot(LP, H1), vyhodnot(PP, H2), H is H1*H2.
vyhodnot(t(LP, +, PP), H):-vyhodnot(LP, H1), vyhodnot(PP, H2), H is H1+H2.
vyhodnot(t(LP, -, PP), H):-vyhodnot(LP, H1), vyhodnot(PP, H2), H is H1-H2.
vyhodnot(t(LP, /, PP), H):-vyhodnot(LP, H1), vyhodnot(PP, H2), H2 =\= 0, H is H1/H2.
```

Úkol: Uměli byste z výrazu v postfixu takový strom vyrobit? A co kdybyste dostali seznam a úkol mezi prvky seznamu přidat operátory a závorky tak, aby výsledný výraz měl zadanou hodnotu?

Mgr. Martin Pilát, Ph.D.
Malostranské náměstí 25
118 00 Praha
Martin.Pilat@mff.cuni.cz

Prolog - grafy a prohledávání stavového prostoru

Je čas začít v Prologu také dělat něco složitějšího. Uvidíme, že Prolog se dá použít i k prohledávání grafů, hledání cest a podobně.

Jak ale v Prologu graf reprezentovat? Existují asi dva způsoby, které se dají relativně snadno použít. Jeden z nich je seznam následníků, druhý je seznam hran.

Pokud chceme graf reprezentovat jako seznam následníků můžeme si např. vytvořit predikát `grafSN([Vrchol1->[Naslednici1],Vrchol2->[Naslednici2], ...])`. Když se nám to hodí, můžeme si přidat i samostatný seznam vrcholů např. jako `grafSN([Vrcholy], [V1->[N1], V2->[N2], ...])`. (Samozřejmě, na operátoru, který použijete pro oddělení vrcholu od seznamu vůbec nezáleží.).

Druhou možností, jak reprezentovat graf je použít seznam hran. V takovém případě si vytvoříme predikát `grafSH([Vrcholy], [hrana(V1,V2), hrana(V3,V4), ...])`, kde `V1`, `V2`, `V3`, `V4` jsou samozřejmě vrcholy ze seznamu `[Vrcholy]`.

Vytvořme si tedy popis malého grafu o 5 vrcholech: `grafSN([1,2,3,4,5],[1->[2,3,4],2->[5],3->[1,2,4],4->[1],5->[2,3])`

První věc, která se nám hodí jsou procedury na převod mezi oběma reprezentacemi.

```
% gSNnaSH(+GrafSN,-GrafSH) :- GrafSH je GrafSN prevedeny ze seznamu nasledniku na seznam hran.
gSNnaSH(grafSN(V, N), grafSH(V, Hrany)):-bagof(hrana(V1,V2), E^(member(V1->E,N),member(V2,E)), Hrany).

% sSHnaSN(+GrafSH,-GrafSN) :- GrafSN je GrafSH prevedeny ze seznamu hran na seznam nasledniku
gSHnaSN(grafSH(V,          E),          grafSN(V,          Naslednici)):-setof(V1->N,          (member(V1,V),setof(X,
member(hrana(V1,X),E),N)),Naslednici).
```

Používejme teď chvíli grafy reprezentované seznamy hran. Napřed můžeme napsat predikát, který vyjadřuje, že mezi dvěma vrcholy existuje cesta.

```
% cesta(+GrafSH, +Odkud, +-Kam) :- v grafu GrafSH existuje cesta z Odkud do Kam
cesta(grafSH(_,E), O, K) :- member(hrana(O,K), E).
cesta(grafSH(V,E), O, K) :- member(hrana(O,Z), E), cesta(grafSH(V,E), Z, K).
```

Procedura `cesta/3` má několik problémů. Jeden z nich je, že nevrací nalezenou cestu, druhý je, že bude vracet úplně všechny cesty, které v grafu existuji včetně těch, které obsahují cykly. Navíc se může i zacyklit (bude procházet ten samý cyklus v grafu stále kolem dokola). Zkusme tedy napsat proceduru, která bude vracet všechny cesty bez cyklů.

```
cestaBC(grafSH(V,E), O, K, C) :- cestaBC(grafSH(V,E),O,K,[],C).
cestaBC(_, O, O, _, [O]).
cestaBC(grafSH(V,E), O, K, P, [O|C]) :- member(hrana(O,Z), E), \+member(Z, P), cestaBC(grafSH(V,E), Z, K, [O|P],
C).
```

Hledání cesty v grafu je podobné prohledávání stavového prostoru. Jen při prohledávání stavového prostoru nemáme tento prostor zadaný explicitně seznamy následníků jednotlivých vrcholů, ale jen akcemi, které lze v každém stavu provést. Představme si například známý příklad se dvěma nádobami a přeléváním vody mezi nimi. Máme dvě nádoby, jednu o objemu `V1` litrů, druhou o objemu `V2` litrů. Navíc máme k dispozici neomezený zdroj vody a vodu můžeme z nádob libovolné vylévat. Cílem je najít takovou posloupnost přelévání, která povede k tomu, že v druhé nádobě je právě daný objem vody.

Stav můžeme v tomto případě reprezentovat jako dvojici `s(X, Y)`, kde `X` je množství vody v první nádobě a `Y` je množství vody v druhé nádobě. Máme potom několik akcí, které můžeme v každém stavu provést.

```
% akce(+Stav, +MaxV1, +MaxV2, -NovyStav):-NovyStav je Stav, ktery vznikne
% aplikaci jedne akce na Stav, kdyz nadoby maji maximalni objemy MaxV1 a MaxV2
akce(s(_, Y), _, _, s(0, Y)). % vyliti prvni nadoby
akce(s(X, _), _, _, s(X, 0)). % vyliti druhe nadoby
akce(s(_, Y), V1, _, s(V1, Y)). % naplneni prvni nadoby
akce(s(X, _), _, V2, s(X, V2)). % naplneni druhe nadoby
akce(s(X, Y), V1, _, s(X1, Y1)):-X + Y > V1, X1 = V1, Y1 is Y - (V1 - X). % preliti druhe nadoby do prvni, nevejde
se vse
akce(s(X, Y), V1, _, s(X1, Y1)):-X + Y =< V1, X1 is X + Y, Y1 = 0. % preliti druhe do prvni, vse se vejde
akce(s(X, Y), _, V2, s(X1, Y1)):-X + Y > V2, Y1 = V2, X1 is X - (V2 - Y). % preliti prvni do druhe, nevejde se vse
akce(s(X, Y), _, V2, s(X1, Y1)):-X + Y =< V2, Y1 is X + Y, X1 = 0. % preliti prvni do druhe, vejde se vse
```

Teď už tedy umíme ke každému stavu najít všechny sousední stavy. Můžeme se tedy podívat, jak bude stav vypadat po jednom kroku. Akce nám vlastně ukazuje jeden krok, který můžeme provést. Jak bude vypadat stav po provedení `N` kroků?

```
% nkroku(+Stav, +MaxV1, +MaxV2, +N, -NovyStav):-NovyStav je stav, ktery vznikne
% aplikovani N kroku na Stav, MaxV1 a MaxV2 jsou maximalni objemy nadob
nkroku(S, _, _, 0, S):-!. % Musime zariznout, jinak budeme zkouset zaporne pocty kroku
nkroku(S, MaxV1, MaxV2, N, S2):-akce(S, MaxV1, MaxV2, S1),
                                N1 is N - 1,
                                nkroku(S1, MaxV1, MaxV2, N1, S2).
```

Predikát `nkroku` je užitečný a funguje, ale občas zkouší i cesty, na kterých se několikrát opakuje ten samý stav. Když nás bude zajímat nejkratší řešení, tak takové cesty zcela jistě nechceme a můžeme se jich zbavit. Navíc by se nám líbilo získat posloupnost stavů, které vedou k cílovému. K tomu použijeme akumulátor (a navíc jednu výstupní proměnnou, která bude obsahovat posloupnost stavů).

```
% nkroku(+Stav, +MaxV1, +MaxV2, +N, -Posloupnost, -NovyStav):-NovyStav je stav,
% ktery vznikne aplikovani N kroku na Stav, MaxV1 a MaxV2 jsou maximalni objemy
% nadob, Posloupnost je posloupnost stavu ze Stav do NovyStav, ktera vede k
% vytvoreni NovyStav, stavy v Posloupnosti se neopakuji
nkroku(S, MaxV1, MaxV2, N, P, S1):-nkroku(S, MaxV1, MaxV2, N, [], P, S1).
nkroku(S, _, _, 0, A, P, S):-reverse(A, P).
nkroku(S, MaxV1, MaxV2, N, A, P, S2):-N>0, akce(S, MaxV1, MaxV2, S1),
                                      \+member(S1, A),
                                      N1 is N - 1,
                                      nkroku(S1, MaxV1, MaxV2, N1, [S1|A], P, S2).
```

Pomocí predikátu `nkroku/6` už můžeme napsat prohledávání, které nám zajistí, že najdeme nejkratší řešení daného problému. Uděláme tzv. iterované prohlubování. Na začátku nastavíme maximální počet kroků na 1, zkusíme tak problém vyřešit, když se to nepovede, zvýšíme počet kroků na 2 a opakujeme. Po každém neúspěchu zvýšíme počet kroků o 1.

```
% vyresID(+PocatecniStav, +MaxV1, +MaxV2, +KoncovyStav, -P):-P je posloupnost
% kroku, ktera prevadi PocatecniStav na KoncovyStav, MaxVi je objem nadoby i,
% posloupnost se hleda pomoci postupnehoprohlubovani (iterative deepening)
vyresID(PS, MaxV1, MaxV2, KS, P):-vyresID(PS, MaxV1, MaxV2, 1, KS, P).
vyresID(PS, MaxV1, MaxV2, N, KS, P):-nkroku(PS, MaxV1, MaxV2, N, P, KS),!.
vyresID(PS, MaxV1, MaxV2, N, KS, P):-N1 is N + 1, N1 < 50, % max pocet kroku je 50
                                     vyresID(PS, MaxV1, MaxV2, N1, KS, P).
```

Výhodou iterovaného prohlubování je, že má menší prostorovou složitost než prohledávání do šířky (nemusí si pamatovat všechny navštívené stavy, stačí mu aktuální větev) a navíc zajišťuje nalezení nejkratšího řešení (což DFS se stejnou prostorovou složitostí nezaručuje).

Napsat prosté DFS je v Prologu samozřejmě taky možné a dokonce je to o něco jednodušší než postupné prohledávání.

```
% vyresDFS(+PocatecniStav, +MaxV1, +MaxV2, +KoncovyStav, -P):-stejne jako
% predchozi, jen pouziva DFS misto ID
vyresDFS(PS, MaxV1, MaxV2, KS, P):-vyresDFS(PS, MaxV1, MaxV2, KS, [PS], P).
vyresDFS(_, _, _, KS, [KS|A], P):-reverse([KS|A], P), !.
vyresDFS(PS, MaxV1, MaxV2, KS, A, P):-akce(PS, MaxV1, MaxV2, NS),
                                      \+member(NS, A), % jinak se zacyklime
                                      vyresDFS(NS, MaxV1, MaxV2, KS, [NS|A], P).
```

Mgr. Martin Pilát, Ph.D.
Malostranské náměstí 25
118 00 Praha
Martin.Pilat@mff.cuni.cz