NEPROCEDURÁLNÍ PROGRAMOVÁNÍ

ÚVOD DO PROGRAMOVACÍHO JAZYKA

PROLOG

verze 3.03

Rudolf Kryl

KSVI MFF UK Praha

1. Neprocedurální programování

Převážná většina programovacích jazyků dnes běžně užívaných má, přes velké rozdíly, jeden podstatný rys - jde o tzv. **procedurální jazyky**. Popisujeme v nich (víceméně přesně), **jak se má daná úloha vyřešit.** Základem, na kterých tyto jazyky stojí, je pojem proměnné jako označení místa v paměti a přiřazovacího příkazu, který umožní nějak získanou hodnotu uložit do vhodné proměnné. V tom, co je nad těmito základy vystavěno, se Basic, assembler, Fortran, Algol, Pascal, Ada, PL/I, C a C++ (abychom vyjmenovali jen ty nejznámější) liší velmi podstatně, daný rámec však nepřekračují.

Existují však i programovací jazyky založené na jiných principech. V **neprocedurálním programovacím jazyce** (trochu zjednodušeně řečeno) **specifikujeme problém**, který chceme řešit **a ne nutně přesný způsob, jakým se má řešit**. Historicky nacházejí takové jazyky uplatnění především v oblasti tzv. **umělé inteligence**, která je mimo jiné charakterizována převahou úloh nenumerického charakteru. Nejdůležitějšími představiteli těchto jazyků jsou **LISP a Prolog**.

LISP je velmi starý programovací jazyk, který vznikl v USA již v padesátých letech. Slouží ke zpracování seznamů, na které převádí všechny datové struktury. LISP je dodnes populární, má mnoho dialektů (pravděpodobně ještě více než Basic), můžete se s ním setkat např. v kreslicím systému Autocad, většina úplných implementací jazyka LOGO obsahuje i "lispovskou" část. Pěknou a dobře čitelnou knihou o LISPu je kniha [1].

My jsme si vybrali druhý z této dvojice, jazyk **Prolog**. Je představitelem tzv. **logického programování** (i když, jak uvidíme, tento rámec poněkud přesahuje). Jak uvidíme, neskládá se **program** v Prologu z příkazů, ale **je** vlastně **popisem relací** pomocí jednoduchých formulí. Na výpočet programu v Prologu se pak lze dívat jako na hledání důkazu vhodné formule tzv. Hornovy logiky rezoluční metodou. Nemusíte však mít strach z přílišného formalismu, Prolog jde vykládat i bez přímých odkazů na logiku. Takový způsob výkladu jsme zvolili i my.

Ovládnutí myšlenkových základů programování v Prologu Vám pomůže lépe pochopit i programování "klasické", především rekursi, která hraje v Prologu - stejně jako v LISPu zcela zásadní roli. Uvidíte také, že přes všechny rozdíly mezi procedurálními a neprocedurálními programovacími jazyky, je programování přece jen "jen jedno" - většinu skutečně chytrých myšlenek z jednoho světa můžeme použít i ve světě druhém. Ze zkušenosti se studenty MFF vím, že takový zcela odlišný pohled na programování je velmi užitečný pro formování názorů na programování.

Cílem našeho výkladu není tedy pouze Prolog sám, ale především pomoci Vám pochopit, co to programování vlastně je.

Náš postup výkladu je blízký knize [2], která je velmi dobře čitelná, ale hůře dostupná - neexistuje český ani slovenský překlad. Nejdostupnější - a velmi dobrou - je učebnice [3], používá však jinou metodu výkladu, blíže k logice, a obsahuje menší množství příkladů.

2. Jednoduchý příklad

Bez dalších úvodů si ukážeme rovnou jednoduchý program v Prologu. Bude sloužit k zodpovídání otázek o rodinných vztazích malé komunity lidí. Představme si, že Spolek klepen (dále jen SK) vyslal jednu ze svých zasloužilých členek jako vyzvědačku, aby zjistila "kdo co s kým". Zprávu vyzvědačky si můžete přečíst v pravém sloupci, který je oddělen znaky % .

V levém sloupci vidíme program v Prologu, který vystihuje stejnou skutečnost (ve skutečnosti je celý následující text regulérním prologovským programem, protože celý text za znakem % až do konce řádky se v Prologu chápe jako komentář. Čtěte zatím jen vyzvědaččinu zprávu vpravo a na vlastní program koukejte jen po očku.

```
muz(adam).                  % Tam vám byl jeden muž Adam
manz(adam,eva).             % ti dva byli manželé.
rodic(adam,josef).          % Adam byl rodičem Josefa
rodic(adam,hugo).           % a Huga.
rodic(eva,josef).           % Rodičem Josefa byla i Eva,
rodic(eva,hugo).            % zrovna tak byla i rodičem Huga.

muz(karel).                 % Byl tam i další muž, jmenoval se Karel,
zena(helena).               % a žena Helena.
manz(karel,helena).         % Ti také byli manželé.
rodic(karel,zuzana).        % Karel byl rodičem Zuzany,
rodic(helena,zuzana).       % i Helena byla jejím rodičem.
rodic(helena,alfred).       % Helena byla i rodičem Alfreda
rodic(karel,alfred).        % a Karel taky.

zena(klara).                % Byla tam i žena Klára,
rodic(klara,emil).          % která byla rodičem Emila,
muz(emil).                  % což byl muž.
rodic(karel,emil).          % Rodičem Emila byl i Karel

zena(kunhuta).              % Byla tam i žena Kuhnuta,
manz(zibrid,kunhuta).       % jejíž manželem byl Žibrid.
muz(zibrid).                % Byl to muž.
rodic(kunhuta,eva).         % Kunhuta byla rodičem Evy
rodic(kunhuta,katka).       % a Katky.
zena(katka).                % Katka byla žena.
rodic(zibrid,eva).          % Rodičem Evy
rodic(zibrid,katka).        % a Katky byl také Žibrid.

zena(ruzena).               % Žena Růžena
rodic(ruzena,jan).          % byla rodičem Jana.
manz(jan,lucie).            % Jan byl manželem Lucie.
muz(jan).                   % Jan byl muž
zena(lucie).                % a Lucie žena.
rodic(jan,adam).            % Oba byli
rodic(lucie,adam).          % rodiči Adama.

muz(jose).                  % Muž José
manz(jose,katka).           % byl manželem Katky.
```

Zpráva klepny-vyzvědačky je psána poměrně neobratně a rozhodně ne krásným jazykem, přesto však z ní lze kromě trivialit jako, že

- Jan a Lucie jsou manželé
- Helena byla žena

vyčíst i lecos zajímavého. Například, že

- manžel Heleny Karel měl nemanželské dítě Emila s nějakou Klárou,

- Hugova teta Katka byla manželkou Josého, neví se však o tom, že by některý z nich měl nějaké dítě
- Adamova tchýně se jmenovala Kunhuta.

Jistě vám nečiní potíže se o tom sami přesvědčit. Zkuste to !

Cvičení:

> Zjistěte
> - jak se jmenoval manžel Josefovy babičky
> - kolik vnuků měl Žibřid.

To bylo snadné. Pravděpodobně by vás ale přivedla do rozpaků zakázka SK, abyste jim udělali program, který by takové závěry umožnil činit. Jak máme naprogramovat, co to je tchýně ?

Budeme-li předpokládat, že SK chce k zodpovídání dotazů použít počítače, musíme se na to, co klepna-vyzvědačka vlastně zjistila, podívat blíže - a trochu to zformalizovat.

Klepna zjistila, že některé objekty zkoumání jsou muži a jiné ženy. Dále zjistila, že mezi některými objekty je vztah manželství a že některé objekty jsou rodiči jiných.

Použijeme-li jazyka matematické logiky, můžeme říci, že k **popsání situace použila** vyzvědačka **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), které se klepně o dané komunitě podařilo zjistit. Dalším rozdílem proti vyprávění klepny v pravém sloupci je, že 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 š.

Klepna-vyzvědačka **mohla skutečnosti**, které zjistila, **vyjádřit i pomocí jiných predikátů**. Mohla například místo predikátu `rodic(X,Y)` použít dvou predikátů `matka(X,Y)` a `otec(X,Y)`. Tím by si ušetřila nutnost konstatovat o některých objektech, že jsou to muži resp. ženy. Predikáty muz a zena by však při tom zcela odstranit nešlo - bez nich bychom nemohli popsat pohlaví těch, kteří nejsou rodiči.

Jistě byste si dokázali vymyslet i jiné množiny predikátů dostatečných k vyjádření dané skutečnosti. **Zkuste to!**

Jak však s takovými informacemi má pracovat počítač? Klepny (spolu s námi) dobře vědí, co je to tchýně nebo teta. Domyslí se také, že Hugo bude asi muž, i když to ze zprávy vyzvědačky přímo nevyplývá. Bude-li program obsahovat dvojici faktů

rodic(a,b). rodic(b,a).

poznají, že to není pikanterie, ale nesmysl.

Jak však takovým vědomostem "naučit počítač" ? Řešení je překvapivě snadné - vůbec ho to učit nebudeme.

Rezignujeme zcela **na význam predikátů**, se kterými pracujeme, a budeme **s nimi pracovat jen jako s "nápisy", 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. Takovému postupu se říká formální a je jednou ze základních pracovních metod matematiky a logiky.

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**.

Výhody formálního postupu znáte však i z "běžného života". Bylo velkým pokrokem, když (obsahové) hieroglyfické písmo bylo nahrazeno hláskovým. Slovo "slunce" sice cizinci nemusí (narozdíl od hieroglyfu - obrázku sluníčka) být srozumitelné, ale zase jde vytvořit slovník, ve kterém lze najít význam slov, kterým

nerozumíme. Zkuste hledat v čínském slovníku - to je fuška a věda. Formalizace zápisu řeči podstatně zjednodušuje, ne-li dokonce umožňuje, knihtisk a zpracování informace.

Jiným příkladem úspěšné formalizace je (desítková) poziční soustava, která podstatně zjednodušuje početní operace i pro člověka (o počítačích nemluvě). Ale zpátky k Prologu.

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

Základní jednotkou prologovského programu je tzv. **k l a u z u l e** , která *vždy končí tečkou*. Náš program je velmi jednoduchý a obsahuje jen **nejjednodušší typ klauzulí tzv. f a k t y** . Fakt v prologovském programu je konstatováním, že pro nějaké objekty platí nějaké predikát.

Prolog je **interaktivní jazyk**, je-li program (například náš) zaveden v paměti, můžeme se ho ptát. Na obrazovce se **objeví výzva ? - a počítač čeká na náš dotaz**.

Ukažme si příklady takové konverzace. Naše dotazy budeme psát *kursivou*, odpovědi počítače s odsazením a **tučně**. Jako dříve budeme v pravé části řádky psát za znakem "% " význam dotazů a odpovědí. **Přijetí odpovědi** počítače musíme potvrdit **znakem „enter“** - odeslání řádku. Tak dostaneme znovu výzvu ?- a můžeme se ptát znovu.

```
?-manz(jan,lucie).          % Jsou Jan a Lucie manželé?
     yes.                   % Ano, jsou.
?-manz(lucie,jan).          % Jsou Lucie a Jan manželé?
     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. A jak jsme si řekli, Prolog nic neví o tom, že my v některých případech vztahy manz(X,Y) a manz(Y,X) považujeme za ekvivalentní.

Pomocí dotazů tohoto typu si můžeme pouze ověřovat, zda pro přesně specifikované objekty platí dané predikáty. Kdybychom mohli své dotazy formulovat jen takto, mělo by to jen malý užitek. SK by také pravděpodobně nebyl takovým programem nadšen. Ve svých dotazech prologovskému programu můžeme však použít i proměnných (nepředstavujte si však nic podobného proměnné např. v Pascalu) . Proměnná se v Prologu pozná tak, že začíná velkým písmenem.

```
?-manz(X,lucie).            % Kdo je manželem Lucie ?
     X = jan.               % Manželem Lucie je Jan.
```

Jak Prolog vyhodnocoval tento dotaz?

Hledal takový **objekt T**, který by **mohl substituovat** (dosadit) **za proměnnou X, aby platilo tvrzení manz(T,lucie).** A našel jako první substituci X = jan , která vyhovovala**.**

Můžeme se ale také zeptat

```
?-manz(jan,X).              % Kdo je manželkou Jana ?
     X = lucie.
```

Vidíme tedy, že samotný **program v Prologu n e m á u r č e n " s m ě r v ý p o č t u** " tj. co je vstup a co má být výstupem. Tento **směr udává až dotaz**, který uživatel zadá.

P r o g r a m j e t e d y j e n s o u h r n e m f a k t ů, k t e r ý " n e l z e s p u s t i t ". V ý p o č e t s t a r t u j e m e a ž z a d á n í m d o t a z u.

Odpověď na následující dotaz bude pochopitelně záporná, protože lucie není prvním argumentem žádného faktu o predikátu manz.

```
?- manz(lucie,X).           % Kdo je manželkou Lucie ?
     no.                    % Nemá ji.
```

Zkusme zadat tento dotaz

```
?-rodic(X,katka).           % Kdo je rodičem Katky ?
     X = kunhuta            % Kunhuta
```

V programu jsou obsaženy dva relevantní fakty

```
rodic(kunhuta,katka). a rodic(zibrid,katka).
```

Obdrželi jsme odpověď X = kunhuta proto, že tento fakt se vyskytuje v programu dříve. Pokud nás zajímají i další možné substituce, které vedou k pravdivému tvrzení, máme možnost požádat o další pokusy o jejich nalezení tak, že na odpověď nepotvrdíme znakem „enter“ jako jsme činili dosud, ale vyžádáme hledání dalších substitucí znakem středník. Zopakujme tedy poslední konverzaci:

```
?-rodic(X,katka).          % Kdo je rodičem Katky ?
     X = kunhuta           % Kunhuta - chci další možnosti,
         ;                 % proto napíši středník
     X = zibrid            % Prolog našel další možné řešení - Žibrid
         ;                 % chci další možnosti,
     no                    % jiného rodiče katka již nemá
```

Středník způsobí odvolání posledně provedené substituce a vyvolá pokus o hledání další vyhovující substituce. Jistě vám to něco připomíná - **b a c k t r a c k i n g** . Skutečně, jak později uvidíme, jádrem algoritmu interpretace prologovských programů je postup "**ú p l n é h o p r o c h á z e n í d o h l o u b k y s n á v r a t y**", tedy backtrackingu.

Můžeme také zjišťovat všechny manželské dvojice pomocí následující konverzace:

```
?-manz(On,Ona).                    % Která dvojice jsou manželé?
     On = adam , Ona = eva  ;      % Adam a Eva, a další ?
     On = karel , Ona = helena  ;  % Karel a Helena, a další ?
     On = zibrid , Ona = kunhuta ; % Žibřid a Kunhuta, a další ?
     On = jan , Ona = lucie  ;     % Jan a Lucie , a další ?
     On = jose , Ona = katka  ;    % José a Katka , a další ?
     no                            % žádná další manželství
                                   % nejsou v programu registrována
```

Naše dotazy mohou být i složitější. Napíšeme-li v dotazu dva predikáty za sebou, odděleny čárkou, hledáme substituci, která způsobí platnost obou. Například

```
?-rodic(X,katka),muz(X).           % Kdo je rodičem Katky a mužem
                                   % zároveň, tj. jejím otcem ?
X = zibrid ;                       % Otcem je Žibrid,
     no                            % další již není k dispozici.
```

U tohoto dotazu je třeba si opět vysvětlit, jak Prolog dospěl ke své odpovědi. Nejprve se snažil nalézt za proměnnou X takovou substituci, aby splnil predikát rodic(X,katka). Jak již víme, našel nejprve substituci X = kunhuta. Tím byl splněn p r v n í c í l - p l a t n o s t p r e d i k á t u rodic(X,Katka) - a Prolog mohl přejít k pokusu o splnění cíle druhého - muz(kunhuta) (za proměnnou X je již substituováno). Ten se však (pochopitelně) splnit nepodařilo.

Proto Prolog zahájil návrat - odvolal poslední substituci za X (obdobně jako činí po středníku) - a pokusil se splnit cíl rodic(X,katka) jinak. Našel další možnost substituce X = zibrid, která již vedla i ke splnění druhého požadovaného cíle. Po dalším návratu vyvolaném zadáním středníku uživatelem, již nebyla jiná substituce splňující oba cíle nalezena.

Viděli jsme tedy, že **operátor č á r k a má v Prologu význam k o n j u n k c e** - 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).

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

Například do našeho programu bychom mohli přidat pravidlo definující nový predikát otec:

```
otec(Otec,Dite):- rodic(Otec,Dite),muz(Otec).
                                    % otcem Dítěte je jeho rodič Otec, který je také muž.
```

Čárka, stejně jako dříve v dotazech, **znamená** i na pravé straně pravidel **konjunkci**.

Povšimněte si také toho, že „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 otec.

Po přidání této klauzule, definující predikát otec, do programu, bychom mohli nového predikátu použít v dotazu:

```
?-otec(X,katka).                    % Kdo je otcem Katky?
     X = zibrid ;                   % otcem je Žibrid,
     no                             % další již není k dispozici.
```

V rámci rovnoprávnosti přidejme do programu i klauzuli definující, co je to matka.

```
matka(Matka,Dite):- rodic(Matka,Dite), zena(Matka).
                                    % Matka Dítěte je jeho rodič, který je žena.
```

Nyní již také můžeme definovat predikát manzdite(X), který vyjadřuje, že X je manželským dítětem:

```
manzdite(X):- otec(Ot,X),matka(Mat,X),manz(Ot,Mat).
                % Dítě je manželské, jestliže jeho otec a matka jsou manželé
```

Z takového predikátu by již členky SK mohly mít radost ! Asi by jim ji však zkazila skutečnost, že definovat predikát nemanzdite(X), který by je zajímal nepochybně víc, je v Prologu dost těžké a nemáme pro to zatím dostatečně silné prostředky. Zkuste se zamyslet nad tím, proč !

Pro pedanty (či životem zkoušené) připomínáme, že jsme do našeho modelu nezahrnuli rozvody. Vadí-li vám to, zkuste program příslušně modifikovat !

Zastavme se na chvíli u **syntaxe Prologu**. Jak už víme, **začínají jména atomů malým písmenem a proměnné velkým**. Je tedy "matka" jméno predikátu a "Matka" jméno proměnné.

Zkusme na závěr této kapitoly definovat několik dalších nových predikátů:

```
deda(D,V):- rodic(R,V),otec(D,R).
                                    % je-li D otec rodiče R, pak je dědou V
teta(T,X):- rodic(Y,X), sestra(T,Y).
                                    % Je-li Y rodič X a T je jeho sestra, je T tetou X
sestra(Ses,X):- rodic(Y,Ses),rodic(Y,X),zena(Ses).
                                    % Má-li Ses stejného rodiče jako X a je-li to žena,
                                    % pak je to sestra X
```

Jsou tyto predikáty naprogramovány správně ?

N e ! Podle této definice je každá žena (pokud se ví o alespoň jednom z jejích rodičů) svojí sestrou.

Abychom mohli sestru definovat správně, potřebujeme predikát

```
ruzne(X,Y)                          % ruzne(X,Y) platí pokud za X a Y
                                    % nejde substituovat tak, aby X a Y byly stejné
```

Definovat takový predikát je pro nás zatím obtížné. Pravděpodobně se vám nelíbil ani popis jeho významu. Problém není kupodivu příliš jednoduchý.

Předpokládejme, že takový predikát máme k dispozici. Pak není žádný problém "vyrobit si sestru".

```
sestra(Ses,X):- rodic(Y,Ses),rodic(Y,X),zena(Ses),ruzne(X,Ses).
```

Všechny definice nových predikátů, se kterými jsme se dosud setkali byly tvořeny jedinou klauzulí. To je však spíše výjimečný případ.

Definujme predikát vmanzelstvi(X,Y), který o dané dvojici bude platit, když X a Y budou manželé (nepožadujeme, aby X byl muž a Y žena). Příslušná definice může vypadat např. takto:

```
vmanzelstvi(X,Y):- manz(X,Y).          % Je-li X manželem Y, jsou X a Y v manželství
vmanzelstvi(X,Y):- manz(Y,X).          % Je-li Y manželem X, jsou X a Y v manželství
```

Totéž by šlo napsat si pomocí **o p e r á t o r u s t ř e d n í k** , který **znamená d i s j u n k c i - logickou spojku n e b o** , i v jedné klauzuli:

```
vmanzelstvi(X,Y):- manz(X,Y) ; manz(Y,X).
                % Je-li X manželem Y nebo Y manželem X, jsou X a Y v manželství
```

Nyní je již také jasné, proč příkazem pro hledání dalšího alternativního řešení byl právě středník.

Z důvodů přehlednosti programů je, jak uvidíte, **většinou lepší používat tvar s dvěma klauzulemi, než se středníkem**. Proto mu, alespoň dokud nezískáte větší praxi a s ní i vlastní názor, dávejte přednost.

Pokud však přesto chcete se středníky experimentovat již teď, vězte, že

operátor čárka (konjunkce) **v á ž e v í c než operátor středník** (disjunkce).

Například

```
p(X):- a(X,Y),b(Y);c(Y),d(X,Y).
```

je ekvivalentní s

```
p(X):- a(X,Y),b(Y).
p(X):- c(Y),d(X,Y).
```

a ne s

```
p(X):- a(X,Y),b(Y),d(X,Y).
p(X):- a(X,Y),c(Y),d(X,Y).
```

Kdybychom chtěli předchozí definici zapsat pomocí středníku v jedné klauzuli, **musíme použít závorky:**

```
p(X):- a(X,Y),(b(Y);c(Y)),d(X,Y).
```

Přehlednost takových zápisů **můžeme zvýšit vhodnou grafickou úpravou**, např.

```
p(X):-  a(X,Y),
        (b(Y);c(Y)),
        d(X,Y).
```

Výhodou zápisu se středníkem, ve srovnání se zápisem pomocí dvou klauzulí, v tomto případě je, že se při navracení po neúspěchu při snaze o splnění predikátu b(Y) , nemusíme znovu pokoušet o splnění predikátu a(X,Y), ale rovnou se snažíme o splnění alternativního predikátu c(Y). Tato výhoda může nabýt na váze především, je-li vyhodnocení predikátu a časově náročné.

Chceme-li tuto výhodu zachovat a vyhnout se používání středníků ve složitějších klauzulích, můžeme si vzpomenout na staré dobré strukturované programování a jeho zásadu "Rozděl, co můžeš" , a definovat nový predikát:

```
bneboc(Y):- b(Y).
bneboc(Y):- c(Y).
p(X):- a(X,Y),bneboc(Y),d(X,Y).
```

Když už jsme se tak zapletli do "stylistických" diskusí, nesmíme vám zamlčet ještě jeden pojem, který by se vám mohl hodit i v následujícím cvičení, tzv. **a n o n y m n í p r o m ě n n o u .**

Chceme-li vytvořit predikát zenaty(X), který by byl pravdivý pro ty objekty, které mají manželku, můžeme to udělat takto:

```
zenaty(X):-  manz(X,Y).
```

Predikát zenaty(X) bude, jak již víme, splněn, pokud nalezneme takovou substituci za proměnné X a Y, po níž bude platit manz(X,Y). Užití predikátu ukazuje následující konverzace:

```
?-zenaty(X).
     X = adam ;
     X = karel ;
     X = zibrid ;
     X = jan ;
     X = jose ;
     no
```

Vidíme tedy, že v předchozí definici predikátu zenaty mají proměnné X a Y, použité v predikátu manz na pravé straně pravidla, rozdílnou roli. Proměnná X se vyskytuje i jinde v klauzuli (v tomto případě vlevo), zatímco proměnná Y nikde jinde v klauzuli není. Nezáleží tedy vůbec na jménu této proměnné.

Definice

```
zenaty(X):-  manz(X,Z).
```

je zcela ekvivalentní definici původní. Abychom to, že nám na hodnotě druhé proměnné nezáleží, vyjádřili explicitně, píšeme takovou definici obvykle ve tvaru

```
zenaty(X):-  manz(X,_).
```

Znak **podtržítko '_' označuje** tzv. **anonymní proměnou**. Použití anonymní proměnné demonstruje i následující konverzace:

```
?-manz(On,_).
     On = adam  ;
     On = karel  ;
     On = zibrid  ;
     On = jan  ;
     On = jose  ;
     no
```

Srovnejte získané odpovědi s odpověďmi na dotaz ?-manz(On,Ona), které jsme uvedli výše. Vidíme, že použití anonymní proměnné v dotazu mělo za následek potlačení výstupu hodnot substituovaných za tuto proměnnou.

Anonymní proměnné jsou dalším prostředkem, který nám pomáhá psát přehlednější programy. Mnoho překladačů prologu vás pomocí varování upozorňuje na skutečnost, že jste některou z proměnných použili v nějaké klauzuli jen jednou. Jak jsme již viděli není to samo o sobě chyba, ale velmi často se tak projeví chyby vzniklé překlepem. Patří k dobrému stylu užívat na takových místech anonymní proměnnou.

Cvičení:

1. Sestavte predikáty

```
tchyne(Tch,X)            % Tch je tchýní X
manzdite(On,Ona,Dite)    % Dite je manželským dítětem
                         % dvojice On a Ona
vnuk(StRodic,Vnuk)       % Vnuk je vnukem StRodice
```

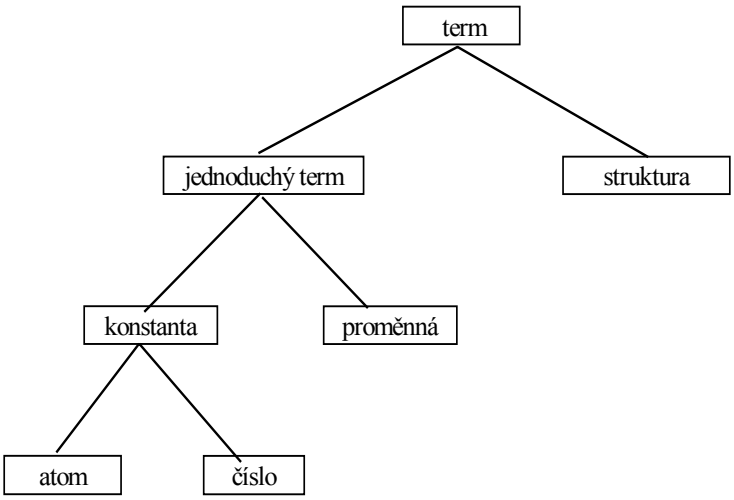
2. Sestavte predikáty vyjadřující další příbuzenské vztahy (např. švagr, újec, bratranec, zeť, dědeček z matčiny strany, manžel vnučky, prababička manželovy tety, a pod.).

3. Tvar prologovského programu

Nebudeme definovat tvar programu v Prologu příliš formálně ze dvou důvodů. Za prvé by nás to odvedlo příliš daleko k čistě technickým otázkám, které pro nás nejsou tak podstatné, a za druhé není v Prologu tak velký rozdíl mezi daty a programem - a jazyk sám obsahuje poměrně bohaté prostředky, kterými programátor může "měnit tvar" programu.

Jak jsme si již řekli, Prolog "počítá" nikoli s obsahy proměnných (proměnné jsou v procedurálních programovacích jazycích de facto jména pro místa v paměti), ale s "nápisy". Universální datovou strukturou, se kterou Prolog pracuje, je tzv. **t e r m -** pojem převzatý z formální logiky.

Klasifikace termů v Prologu je vidět z následujícího obrázku:



obr. 1 Datové typy Prologu

A t o m y mohou být v Prologu konstruovány jedním ze tří způsobů:

- **buď** *začínají malým písmenem* a *obsahují pouze písmena, číslice a znak podtrhávátko*, tedy např.
  rodic emil novy_prvek eMIL atom1
- **nebo** se *skládají pouze ze speciálních znaků*, např.
  , ; < --> <===> :- ?- ! .
- **nebo** jde o *znakové řetězce* uzavřené mezi znaky apostrof (v některých implementacích uvozovky)
  'Adam' 'Horní Počernice' 'kykyryký' .

Č í s l a : V Prologu máme vždy k dispozici celá čísla, někdy však nejdou zpbrazovat příliš velká čísla ("maxint" je malé). Komerční implementace Prologu obsahují většinou i reálná čísla.

P r o m ě n n é : Jak jsme si již řekli, poznají se tak, že *začínají velkým písmenem nebo znakem podtrhovátko, nesmějí však obsahovat speciální znaky,*. např.

X Otec _ Seznam _XXX

S t r u k t u r y jsou *definovány rekursivně*:

Struktura je tvořena **f u n k t o r e m** (který má danou **a r i t u** tj. počet argumentů) **a** příslušným počtem termů, které jsou **j e h o a r g u m e n t y** . Např.

rodic(adam,X)	funktor rodic má aritu 2
datum(23,8,1993)	funktor datum má aritu 3
okamzik(datum(23,8,1993),cas(21,35))	funktor okamžik má aritu 2 funktor cas má aritu 2

strukturou je i klauzule

deda(X,Y) :- rodic(X,Z),rodic(Z,Y).	předdefinovaný funktor **„:-„** je binární a píše se jako infixový operátor

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 ternální), které oba mají jméno ppp.

Víme již, že **program** v Prologu **je posloupnost klauzulí** a že každá klauzule je ukončena tečkou. **Klauzule** mohou být jednoho ze **dvou typů**

fakta	např.	manz(emil,rebeka).
pravidla	např.	otec(Ot,Dite):- rodic(Ot,Dite),muz(Ot).

Levé straně pravidla říkáme **hlava pravidla**, pravé pak **tělo pravidla** (hlava je vlastně prvním argumentem funktoru :- a tělo jeho druhým argumentem).

Na fakty se můžeme podívat také jako na speciální typ pravidel, na jejichž pravé straně stojí standardní nulární predikát true, který je vždy platný

manz(emil,rebeka):- true. ,

resp. jako na pravidla s prázdným tělem. Proto můžeme mluvit o **hlavě i u faktů.**

Ještě jeden termín se nám bude hodit - je to **procedura**. Procedura v prologovském programu je posloupnost klauzulí, jejichž hlavy mají stejný funktor (tj. včetně arity). Uvnitř této posloupnosti je zachováno pořadí klauzulí z programu.

Jak uvidíme, v prologovském programu vůbec nezáleží na pořadí klauzulí, které nepatří do stejné procedury. Proto je běžné psát klauzule patřící k téže proceduře v programu pohromadě.

Ještě něco **o komentářích**. Již umíme psát **řádkové komentáře** uvozené **znakem procento %** - text od tohoto znaku počínaje až do konce řádku se bere jako komentář. Někdy je však opakování % na každém řádku velmi nepříjemné. Proto můžeme používat i **komentářových závorek** /* a */ . **Text mezi "závorkami /* a */ je celý chápán jako komentář** . Tyto komentáře mohou tedy obsahovat i více řádek, resp. mohou končit před koncem řádku:

```
/*
     Program našich klepen (obohacený o některé nové procedury) napsaný
tak, že klauzule patřící k jednotlivým procedurám jsou pohromadě.
Uvnitř jednotlivých procedur jsme pořadí klauzulí nezměnili.
     Stejně jako v Pascalu nezáleží na umístění klauzulí na řádce ani na
počtu mezer, které jednotlivé prvky jazyka oddělují. Programy v Prologu se
tedy píší v tzv. volném formátu, jako dnes už prakticky u všech
programovacích jazyků.
*/

%  Procedura muz
muz(adam).          muz(karel).         muz(emil).
muz(zibrid).        muz(jan).           muz(jose).

%  Procedura zena
zena(lucie).        zena(eva).          zena(helena).
zena(klara).        zena(kunhuta).      zena(katka).
zena(ruzena).

%  Procedura manz
manz(adam,eva).             manz(karel,helena).
manz(zibrid,kunhuta).       manz(jan,lucie).
manz(jose,katka).

%  Procedura rodic
rodic(adam,josef).          rodic(adam,hugo).
rodic(eva,josef).           rodic(eva,hugo).
rodic(karel,zuzana).        rodic(helena,zuzana).
rodic(helena,alfred).       rodic(karel,alfred).
rodic(klara,emil).          rodic(karel,emil).
rodic(kunhuta,eva).         rodic(kunhuta,katka).
```

```
rodic(zibrid,eva).            rodic(zibrid,katka).
rodic(ruzena,jan).            rodic(jan,adam).
rodic(lucie,adam).

%  Procedura otec
otec(Otec,Dite):- rodic(Otec,Dite),muz(Otec).

%  Procedura matka
matka(Matka,Dite):- rodic(Matka,Dite), zena(Matka).

%  Procedura manzdite
manzdite(X):- otec(Ot,X),matka(Mat,X),manz(Ot,Mat).

%  Procedura deda
deda(D,V):- rodic(R,V),otec(D,R).

%  Procedura vmanzelstvi
vmanzelstvi(X,Y):- manz(X,Y).
vmanzelstvi(X,Y):- manz(Y,X).
```

4. Unifikace a backtracking - základ interpretace prologovských programů

V této kapitole se seznámíme podrobněji s tím, jak "vlastně Prolog počítá". Již víme, že interpret prologu hledá metodou backtrackingu v programu vhodnou klauzuli a substituci tak, aby mohl "uspokojit dotaz". S takovým mlhavým a neurčitým popisem jsme se mohli spokojit, dokud "o nic nešlo", ale brzy bychom se dostali do úzkých.

Protože idea backtrackingu (úplného prohledávání do hloubky s návratem) je vám z programování přece jen známa, podíváme se nejdříve na problematiku hledání vhodné substituce.

4.1. Unifikace

Začneme příkladem. Mějme velmi jednoduchý (až prostoduchý) program reprezentující naše znalosti o směru úseček. Bude pracovat se čtyřmi predikáty:

bod(X,Y)	binární predikát bod reprezentuje bod v rovině, jeho argumenty jsou souřadnice příslušného bodu ,
usecka(Zac,Kon)	binární predikát usečka reprezentuje úsečku, argumenty jsou její počáteční a koncový bod ,
vodorovna(Usecka)	unární predikát vodorovna používáme tak, že jeho argumentem je úsečka; o ní pak tvrdí, že má vodorovný směr (tj. je rovnoběžná s osou x) ,
svisla(Usecka)	unární predikát svisla používáme tak, že má za argument úsečku; o té pak predikát tvrdí, že má svislý směr (tj. je rovnoběžná s osou y) .

Zkuste nejprve příslušný program sestavit samostatně !

```
svisla( usecka( bod(X,_),bod(X,_) ) ).
                                % úsečka je svisla, pokud oba její koncové
                                % body mají stejné X-ové souřadnice
vodorovna( usecka( bod(_,Y),bod(_,Y) ) ).
                                % úsečka je vodorovná, pokud oba její koncové
                                % body mají stejné Y-ové souřadnice
```

Pokusme se z tohoto programu "vydolovat, co se dá". Jaké dotazy vlastně můžeme tomuto programu položit?

```
?- vodorovna(usecka(bod(1,3),bod(2,4))).     % Je úsečka s krajními body (1,3)
            no                               % a (2,4) vodorovná ?
?- svisla(usecka(bod(2,3),bod(2,4))).        % Je úsečka s krajními body (2,3)
            yes                              % a (2,4) svislá ?
?- svisla(usecka(bod(1,3),bod(X,4))).        % Jaká musí být x-ová
            X = 1                  % souřadnice bodu, jehož y-ová souřadnice je
                                   % rovna 4, aby úsečka, která spojuje tento
                                   % s bodem (1,3) byla svislá ?
```

To nebylo nic nového, jakou odpověď byste však očekávali na dotaz

?- svisla(usecka(bod(1,Y1),Z)). ? (1)

Je jasné, že za proměnnou Z může být substituována jedině struktura s funktorem bod. Jaké však mají být její argumenty ? Co byste řekli takovéto odpovědi

Y1=7 Z=bod(1,10087) ? (2)

Tato odpověď by byla sice "správná" (vždyť přece úsečka spojující body (1,7) a (1,10087) je svislá), ale současně i velmi podezřelá. Na základě čeho Prolog stanovil substituce za y-ové souřadnice obou bodů ?

Jak jsme si řekli, "neví" Prolog o predikátech našeho programu nic jiného než to, co jsme mu o nich v programu sdělili. Nemůže tedy vědět, že za druhý argument predikátu bod se mají dosazovat čísla, - a i kdyby to věděl, jak by přišel zrovna na hodnoty 7 a 10087 ?

Navíc, uvědomíme-li si skutečný význam dotazu (1), zjistíme, že odpověď (2) vlastně vůbec správná nebyla. Dotazem (1) jsme se totiž ptali:

"V jakých bodech mohou končit svislé úsečky, které začínají v bodě s x-ovou souřadnicí rovnou 1 ?".

A správná odpověď na tuto otázku je:

"V libovolném bodě, jehož x-ová souřadnice je rovna 1." (3)

Přesně vzato ani tato odpověď správná není, neboť, aby šlo o skutečnou úsečku, musí se její koncový bod lišit od počátečního, ale tento poznatek jsme do programu nezabudovali (a ani to, zatím, neumíme).

Požadovanou odpověď (3) však Prolog může nalézt - narozdíl od nepochopitelné odpovědi (2) - velmi snadno. Stačí, aby nesubstituoval zbytečně, když nemusí. Odpověď na dotaz (1) by tedy mohla vypadat takto:

```
Y1 = cokoliv
Z  = bod(1,Y2)                                          (4)
Y2 = cokoliv          ,
```

kde cokoliv by byl předdefinovaný atom, který by nám symbolizoval skutečnost, že na hodnotě dané proměnné splnění daného cíle nezáleží. To bychom však zaváděli zbytečně nový - a nešikovný - formalismus. Zadáte-li dotaz (1) vašemu interpretu Prologu dostanete odpověď v tvaru podobném následujícímu:

```
Y1 =  _323     Z = bod(1,_324)                          (5)
```

Znakem podtrhovátko '_' začínají, jak již víme, jména proměnných. Interpret většinou generuje jména interních proměnných připojením přirozeného čísla za tento znak.

Jaký je z toho všeho závěr ?

Interpret Prologu se snaží nalézt nejobecnější substituci, která splní daný cíl, tzn. *nesubstituuje zbytečně, pokud nemusí.*

Odpověď, ve které se vyskytuje proměnná, znamená, že na dané hodnotě nezáleží (a může být za ní dále substituováno cokoliv).

Uveďme ještě jeden příklad dotazu pro náš program

```
?- horiz(usecka(bod(1,Y1),bod(X,Y2))).
```
(6)

V tomto případě odpověď může být např. takováto:

```
Y1 = _432     X = _433     Y2 = _432     .
```
(7)

Všimněte si, že za proměnné Y1 a Y2 **byla substituována tatáž interní proměnná**.

Došlo tedy ke **ztotožnění proměnných Y1 a Y2 z dotazu**. Bude-li se s odpovědí dále pracovat, musí být za obě dosazeno přesně totéž.

Abychom mohli to, co nyní již velmi pravděpodobně správně tušíte, formulovat přesněji, potřebujeme zavést dva pojmy.

Musíme definovat co to znamená, že si **dva termy odpovídají** (match) a jak probíhá proces **unifikace**, který to demonstruje :

1. Jestliže **T1 i T2 jsou proměnné**, **pak si odpovídají** a *výsledkem jejich unifikace je ztotožnění těchto proměnných.*
2. Jestliže **T1 je proměnná a T2 jakýkoli term (různý od proměnné)**, **pak si termy T1 a T2 odpovídají** a *výsledkem jejich unifikace je substituce termu T2 za proměnnou T1*
   T1 <- T2 .
3. Jestliže **T2 je proměnná a T1 jakýkoli term (různý od proměnné)**, **pak si termy T1 a T2 odpovídají** a *výsledkem jejich unifikace je substituce termu T1 za proměnnou T2*
   T2 <- T1 .
4. Jestliže jsou **T1 i T2 jednoduché termy, různé od proměnných**, pak **si odpovídají jen když jsou identické** a jejich *unifikace je prázdná akce*.
5. Jestliže **T1 i T2 jsou struktury**, pak **si odpovídají**, pokud **jsou tvořeny týmž funktorem** (z definice pak musí být jejich arita stejná) **a** pokud si **jejich parametry navzájem odpovídají** (první prvnímu, druhý druhému, ... , atd.) a *jejich unifikace spočívá v unifikaci odovídajích si dvojic parametrů.*
6. **V žádných jiných případech si dva termy neodpovídají** a *nelze je unifikovat*.

Uveďme několik příkladů unifikace termů:

a) termy

```
f(t(a,X),t(B,b),Y)                          a
f(t(A,b),t(f(t(a,a),t(z,b),Z),b),r)
```

si odpovídají a unifikací získáme substituce:

```
X <- b  ,   A <- a  ,  B <- f(t(a,a),t(z,b),Z)  a   Y <- r  .
```

b) termy

```
g(A,B,a(A,a(x,B)))                          a
g(v,t,a(A,a(x,w)))
```

si neodpovídají (nejde je unifikovat):

- unifikací prvního argumentu dostáváme substituci A <- v,
- unifikací druhého B <- t,
- funktor třetího argumentu v obou termech je a,
- jeho první argumenty si odpovídají,
- druhé argumenty mají (po provedení již vynucené substituce B <- t) tvar:
  a(x,t) a a(x,w) a tedy si neodpovídají.

Poznámka.

S daným algoritmem zjišťování, zda si dva termy odpovídají jsou i (spíše teoretické) problémy. Zkusme zjišťovat, zda lze unifikovat termy

```
X      a        f(X)    .
```

Námi popsaný algoritmus se zacyklí, hledaje substituci tvaru

```
X=f(f(f(f( ........  ))))      .
```

I když my na první pohled vidíme, že vhodná (konečná) substituce nemůže existovat.

Konkrétní implementace Prologu, i když používají popsaný algoritmus, mají takové případy ošetřeny, aby zbytečně nepadaly.

Zdůrazněme na závěr tohoto odstavce **ještě několik důležitých skutečností týkajících se proměnných v Prologu.** (Abychom si rozuměli, budeme se - pro jistotu - vyjadřovat "velmi lidově". Prosím tedy šťouraly, aby mne nyní nechytali za slovo):

1. **Proměnné** v Prologu jsou **"lokální v klauzuli, v níž jsou použity".** To znamená, že je-li v různých klauzulích použito stejného označení proměnné, nemají spolu tyto proměnné (z hlediska programu) vůbec nic společného.
2. Proměnné v Prologu neoznačují místo v paměti počítače (jako je tomu v procedurálních jazycích ala Pascal), ale "jen" **místo v termu, kam lze substituovat** jiný term. To, že se v klauzuli vyskytuje tatáž vícekrát, znamená, že na příslušná místa je nutno substituovat totéž.
3. V Prologu tedy **neexistují "globální proměnné",** do nichž bychom si mohli uložit nějakou hodnotu a někdy později ji odtud znovu "přečíst" pro další použití. **Veškerý přenos informací probíhá předáváním parametrů.**

Nyní již můžeme poměrně přesně popsat algoritmus interpretace prologovských programů.

4.2. Algoritmus interpretace prologovských programů

Jak jsme si již řekli, je základem interpretace prologovských programů algoritmus backtrackingu, tj. **prohledávání do hloubky s návratem v případě neúspěchu.** Nebudeme jej popisovat přesně, zbytečně by to náš text prodloužilo a navíc to lze ponechat čtenáři jako pěkné cvičení (z procedurálního programování).

Připomeňme si velmi volně základní ideu prohledávání s návratem:

Jdi kupředu, dokud můžeš, a pamatuj si všechna rozhodnutí, která při tom děláš. Pokud již nemůžeš dál, vrať se na místo posledního rozhodnutí, rozhodni se jinak, a pokračuj stejným způsobem dál.

Pokud prostor, kterým procházíme, je konečný a nemá cykly, pak popsaným algoritmem projdeme celou jeho část, která je dostupná z místa našeho startu.

Z programátorského hlediska se algoritmus backrackingu realizuje pomocí zásobníku, ve kterém si pamatujeme cestu i rozhodování na jejích jednotlivých místech.

Činnost **interpretu prologovských programů** lze plně popsat pomocí rekursivní procedury **splňování cíle**. Cílem přitom bude term, který je buď dotazem nebo vznikl z některého predikátu zapsaného v programu substitucí za (některé) proměnné v něm obsažené. Přesné zformulování ponecháme čtenáři jako cvičení a spokojíme se s několika poznámkami:

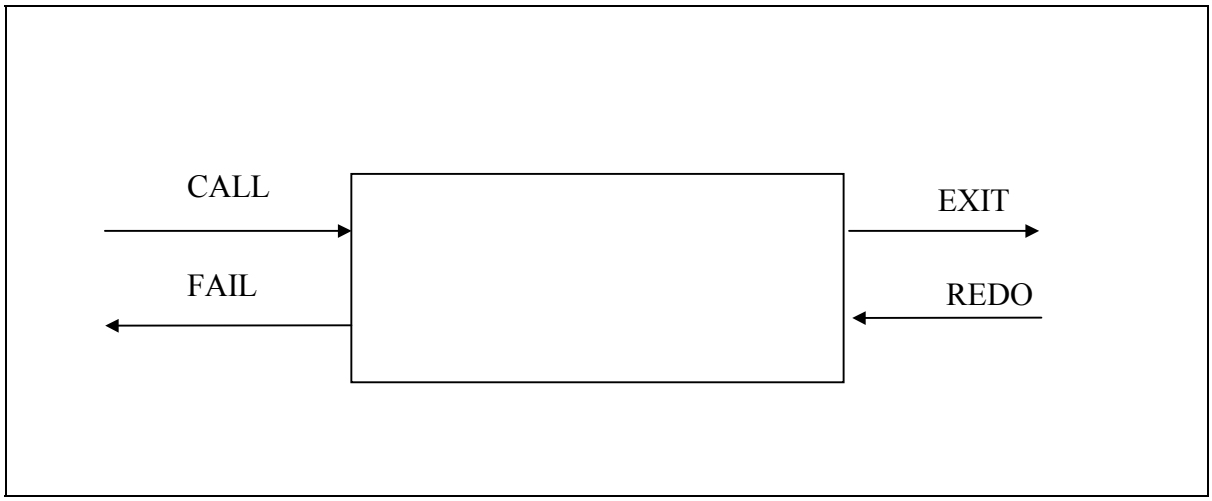
1. Prvním cílem, který se interpret snaží splnit je dotaz zadaný uživatelem.
2. Při splňování cíle interpret hledá první klauzuli, jejíž hlava odpovídá tomuto cíli. Při unifikaci cíle s hlavou této klauzule se provedou jisté substituce (nejen v hlavě, ale v celé klauzuli). Do zásobníku se uloží číslo použité klauzule a substituce, které byly provedeny.
3. Klauzule, jejíž hlava odpovídá právě splňovanému cíli se při cestě výpočtu "kupředu" hledá vždy od počátku programu.
4. Pokud je nalezená odpovídající klauzule faktem, cíl uspěl.
5. Pokud je to pravidlo, pak prolog přejde k postupnému splňování cílů vzniklých provedenými substitucemi z predikátů těla této klauzule. Pokud uspějí všechny cíle těla této klauzule, uspěl i původní cíl. (Při této "cestě kupředu" se zásobník naplňuje).

6. Pokud se některý cíl nepodaří splnit - dojde v souhlase se strategií backtrackingu k navracení výpočtu. Při tom se ze zásobníku vyzvedne poslední položka, prolog odvolá naposledy provedené substituce a snaží se najít další klauzuli, jejíž hlava odpovídá splňovanému cíli. Při tom se program pochopitelně prohledává nikoli od začátku, ale od aktuální pozice v programu (která byla právě vyzvednuta ze zásobníku). Pokud tak dojdeme až na konec programu, tento cíl definitivně nejde splnit (pochopitelně však může být později splněn jiný cíl odvozený jinou substitucí z téhož predikátu v programu) a navracení pokračuje dále stejným způsobem.

4.3. Krabičkový model výpočtu, princip ladění

Pro znázornění průběhu vyhodnocování prologovských cílů se často používá tzv. krabičkový model výpočtu prologovských programů.

Každému volání predikátu odpovídá krabička se čtyřmi bránami. Dvě jsou vstupní a dvě výstupní.



Horní dvě **brány CALL a EXIT** se používají **při** postupu **výpočtu vpřed**.

Vstupní bránou **CALL vstupuje výpočet do splňování daného cíle**. **Podařilo-li se daný cíl splnit** (tj. byla-li nalezena substituce, která převedla daný term na takový, který jde z programu odvodit), **odchází výpočet branou EXIT** k dalšímu cíli, který se má splnit jako další.

Spodní dvě **brány REDO a FAIL** se použijí, když se **výpočet navrací**.

Pokud se **následující cíl nepodařilo splnit, vrací se výpočet** z brány FAIL tohoto cíle **zpět do našeho branou REDO.** To je signálem, že se má **hledat další alternativní substituce** pro splnění aktuálního cíle. **Podaří-li se** [illegible] **výpočet odchází branou EXIT** znovu k následující krabičce, a **když se to nepodaří, výpočet se vrací zpět** [illegible]**ou FAIL** k předcházející krabičce.

Na představě tohoto modelu je velmi často založen **způsob, kterým se prologovské programy obvykle ladí**.

Programátor může předepsat, které predikáty chce při výpočtu v ladicím režimu sledovat (obvykle předdefinovaným predikátem **spy**). Je-li zapnut ladicí režim (obvykle předdefinovaným predikátem **trace**, vypíná jej predikát **notrace**) výpočet se vždy zastavuje při průchodu branou krabičky odpovídající některému ze sledovaných predikátů.

Při každém takovém zastavení si může programátor na obrazovce přečíst aktuální stav termu, který se program právě pokouší splnit resp. byl právě splněn.

Navíc může ovlivnit další průběh výpočtu (např. zrušení ladicího režimu, přidání dalšího predikátu ke sledovaným resp. zrušení sledování jiného, průchod přes některé brány sledovaných predikátů bez zastavení, vyvolání pomocného výpočtu prologovským dotazem - po obdržení odpovědi pokračuje sledovaný výpočet, provedení libovolného příkazu operačního systému atp.).

5. Význam prologovských programů. Rekurse.

Tato kapitola bude pro vás pravděpodobně o něco obtížnější než dosavadní text. Je však klíčová jak pro praktické programování v Prologu, tak i pro pochopení jeho principů a role mezi ostatními programovacími jazyky.

Proto je dobré se k ní občas vrátit. Pak možná objevíte nové významy a další látku k přemýšlení.

5.1. Deklarativní význam prologovského programu

Prolog je praktickým pokusem o realizaci myšlenky logického programování, které se snaží nahradit klasické programy předepisující přesný algoritmus řešení úlohy jejím popisem pomocí prostředků matematické logiky.

Každému **programu** odpovídá jednoznačně formule výrokového počtu, která určuje jeho **(deklarativní[1]) význam**. Tuto formuli budeme definovat postupně. Nejprve budeme pro každou klauzuli definovat formuli, která udává význam této klauzule. Význam programu je pak definován jako konjukce významů jeho klauzulí.

Každé klauzuli prologovského programu **přiřadíme formuli výrokového počtu**, která vyjadřuje význam této klazule následujícím způsobem:

fakt p znamená platnost **tvrzení p**

a

pravidlo p :- r_1 ,r_2 ,r_3 , ... ,r_n .

má význam implikace **r_1 & r_2 & r_3 & ... & r_n => p,**

která tvrdí, že jsou-li platná všechna tvrzení r_1,r_2,r_3,...,r_n , je platné i tvrzení p.

Pokud tato formule **obsahuje proměnné, znamená tvrzení o existenci objektů**, jejichž **substituování** do formule ji převede na **platné tvrzení o těchto objektech**.

Prologovský **program** je vlastně množinou klauzulí, **jeho význam je dán konjunkcí všech formulí**, **které určují význam jednotlivých klauzulí programu**. Tomuto významu se říká **deklarativní význam programu**.

Než pokročíme dále, uvědomme si, že **deklarativní význam nezávisí ani na pořadí klauzulí v programu, ani na pořadí predikátů v těle jednotlivých pravidel.** Proto jsme také mohli na předchozích řádcích o programu mluvit jako o množině klauzulí, a nikoli o posloupnosti klauzulí, jako dosud.

Skutečnost, že programovací jazyk může mít takovou vlastnost, je jistě fascinující. Jenže, jak za chvíli uvidíme, tak jednoduché to zase není. Pouze deklarativní správnost programu totiž nestačí k tomu, aby "počítal správně".

Platí však důležité tvrzení:

> **Je-li program deklarativně správný, nemůžeme se od něj dočkat špatného výsledku.** Nemusíme se však nutně výsledku dočkat vůbec - výpočet programu se může zacyklit.

Můžeme si to představit tak, že program "vymezuje prostor" platných tvrzení . Vyhodnocování dotazů je pak vlastně prohledávání takto vymezeného prostoru.

Je-li program deklarativně správný, pak vymezený prostor obsahuje právě všechny správné odpovědi na libovolný dotaz. Prolog v něm tedy nemůže nalézt žádnou špatnou odpověď. To však bohužel ještě nezaručuje, že výpočet při vyhodnocování dotazu v tomto prostoru "nezabloudí" (zacyklí se) a žádné odpovědi se tedy nedočkáme.

[1] Přívlastku deklarativní užíváme proto, abychom tento význam programu odlišili od významu procedurálního, na který jste zvyklí z jiných programovacích jazyků.

5.2. Rekursivní definice predikátu predek

Zkusme definovat predikát

```
predek(Predek,Potomek)     ,
```

který bude znamenat, že objekt Predek je předkem objektu Potomek.

Uvědomme si, že úloha, se podstatně liší od té, kdy jsme definovali např. dědu, předka z "druhého kolene". Pro to stačilo dvakrát použít na pravé straně pravidla predikát rodic (pro předka z třetího kolene - pradědečka by tam musel být rodic třikrát). Nový predikát však musí vystihnout předky nejen z prvního, druhého a třetího kolene, ale potenciálně z "nekonečného počtu" kolen. Nevystačíme tedy s pravidly, na jejichž pravé straně je jen predikát rodic. Musí přijít ke slovu rekurse (s níž Prolog stojí a padá).

Rekursivní definice předka vám jistě nebude dělat potíže.

Každý předek objektu X je buď jeho rodičem, nebo rodičem jiného předka X.

```
%  ============================================================
%    predek(?Pred,?Pot)   Pred je předkem potomka Pot
%  ============================================================
predek(Rod,Pot):- rodic(Rod,Pot).    % Rod je rodičem potomka Pot
predek(Pred,Pot):-
     rodic(Pred,MlPred),             % Pred je rodičem mladšího
     predek(MlPred,Pot).             % předka Mlpred potomka Pot
```

Dříve než pokročíme dále, musíme si vysvětlit, co vlastně znamenají otazníky před argumenty predikátu předek v „komentářové hlavičce“. Není to součást jazyka (může se tedy vyskytovat jen v komentářích), ale více méně všeobecně akceptovaná **konvence** pro specifikovaní toho, že autor procedury něco předpokládá o dotazech, které budou kladeny. Před argument v komentáři můžeme napsat jeden ze tří znaků + , - , ¨? .

S následujícím významem:

- „**výstupní** argument“ - autor komentáře dává najevo, že v dotazech má na odpovídajícím místě stát nekonkretizovaná proměnná,
+ „**vstupní**“ parametr – autor komentáře dává najevo, že v dotazech má na odpovídajícím místě stát již konkretizovaný term,
? argument může být **jak vstupní, tak i výstupní** tj. v dotazech může na odpovídajícím místě stát „cokoli“ .

Pokud takový znak v komentáři neuvedeme, má to stejný význak jako by tam byl otazník.

Podle této komentářové konvence jsme tedy v definici predikátu predek dali najevo, že jsme při jeho definici neměli na mysli nějaké „specializované“ dotazy.

Zkusme, jaké výsledky dá procedura predek, bude-li doplněna k programu našich klepen z konce kapitoly 3.

```
?- predek(lucie,klara).        % Je Lucie předkem Kláry ?
     no
?- predek(jan,X).              % Kdo je potomkem Jana ?
     X = adam    ;
     X = josef   ;
     X = hugo    ;
     no
?- predek(X,hugo).             % Kdo je předkem Huga ?
     X = adam    ;
     X = eva     ;
     X = kunhuta ;
     X = zibrid  ;
     X = ruzena  ;
     X = jan     ;
```

```
X = lucie   ;
no
```

Cvičení

Neměla by se procedura predek jmenovat raději potomek ?

Pokud jste pochopili, co se po vás v předchozím cvičení vlastně chtělo, udělali jste mi radost.

V jistém smyslu se totiž dá říci, že jméno potomek by bylo pro tuto proceduru o mnoho adekvátnější. Proč ? Procedura predek zjišťuje totiž potomky daleko přirozeněji a rychleji než předky.

Jak se vyhodnocuje dotaz typu

```
?- predek(Pred,konkretni_potomek).
```

kde `konkretni_potomek` je konkrétní atom, pokud nejde o rodiče tohoto potomka ?

Velmi komicky - postupně se procházejí vůbec všechny údaje o dvojicích (Rodič,Dítě) a pak se pro každé takové Dítě zjišťuje, zda náhodou není předkem vstupního (nebo raději vstupujícího?) konkrétního potomka.

Představíte-li si údaje o populaci několika tisíc osob, je to spíš námět na další vtip o policajtech, tentokrát hledajících v archivu, než algoritmus vhodný pro řešení dané úlohy.

Naproti tomu, dotaz typu

```
?- predek(konkretni_predek,Pot).
```

se vyhodnocuje poměrně rozumně. Nestačí-li nám dítě objektu `konkretni_predek`, generujeme pro všechny jeho děti (těch nemůže být příliš mnoho) jejich potomky - opět napřed jejich děti, atd.

"Zdravý selský rozum" nám napovídá, že je lepší když se nejprve snažíme splnit cíle, v nichž alespoň nějaký argument je konkretizován. Budou-li tedy převládat dotazy po předcích nad dotazy po potomcích, bylo by lepší definovat předka jinak:

```
%  ============================================================
%           jiná implementace predka vhodnější pro dotazy
%                     typu pred(-Pred,+Pot)
%    predek2(?Pred,?Pot)   Pred je předkem potomka Pot
%  ============================================================
predek2(Rod,Pot):- rodic(Rod,Pot)    % Rod je rodičem potomka Pot
predek2(Pred,Pot):-                  % jeho předky
     rodic(Rod,Pot),
     predek2(Pred,Rod).              % jsou i předci jeho rodičů
```

5.3. Procedurální význam programu

Predikát predek2 byl založen na (i když z našeho hlediska jen velmi nepatrně) odlišné úvaze, než predikát predek.

Zkusme však zkoumat predikáty, které vzniknou z predikátu predek jen "permutací" jeho klauzulí a cílů v nich. Dostaneme tři další možnosti. Prozkoumejme, zda jsou všechny procedurálně správné, tj. zda dávají vždy správné výsledky a zda vydají (po odmítání středníkem) všechny správné výsledky. Predikát predek je správný i procedurálně. Poměrně snadno se o tom můžete sami přesvědčit. Jak to však bude se správností následující jeho modifikace:

```
%  ============================================================
%     predek3    -     jiná varianta procedury predek
%                   má stejný deklarativní význam jako predek !!
%    predek(?Pred,?Pot)   Pred je předkem potomka Pot
%  ============================================================
predek3(Pred,Pot):-
     predek3(MlPred,Pot),
     rodic(Pred,MlPred).
```

```
Predek3(Rod,Pot):-
     rodic(Rod,Pot).
```

Cvičení Jak bude probíhat konverzace s programem po dotazech

```
?- predek3(jan,X).          ?- predek3(X,hugo).
```

Pokud jste uhodli, že to žádná konverzace nebude - prolog se totiž při vyhodnocování těchto dotazů zacyklí, blahopřejeme vám. Je vidět, že jste dosavadnímu výkladu perfektně rozuměli. Pokud ne, nic si z toho nedělejte, tohle by překvapilo kdekoho.

Proč se program při pokusu splnit cíl predek3 musí zacyklit bez ohledu na to, které argumenty jsou (a zda jsou) konkretizovány?

Splnění cíle predek3(X,Y) je totiž převedeno na splnění cíle predek3(Z,Y) a čehosi jiného potom. Vyhodnocení tohoto cíle si vynucuje vyhodnocení dalšího cíle predek3(W,Z), atd donekonečna.

V teorii gramatik se takovým pravidlům, jako je první klauzule predikátu predek3, říká **levě rekursivní** (definovaný predikát je i prvním cílem vpravo). Pokud je první klauzule nějaké procedury levě rekursivní, pak se výpočet této procedury zacyklí vždy.

Ve skutečnosti to donekonečna nebude, protože interpretu přeteče zásobník a dotaz skončí hláškou podobné této

```
error - stack overflow          .
```

A máme po idyle. Procedura predek3 **je deklarativně správná, nedá však nikdy žádný výsledek**.

Jak to bude s dalšími dvěma "permutacemi" procedury predek? Začněme tou, která nemění klauzule, jen jejich pořadí.

```
%  ============================================================
%               další varianta procedury predek ,
%                také deklarativně ekvivalentní
%    predek4(?Pred,?Pot)   Pred je předkem potomka Pot
%  ============================================================
predek4(Pred,Pot):-
     rodic(Pred,MlPred),
     predek4(MlPred,Pot).
Predek4(Rod,Pot):-
     rodic(Rod,Pot).
```

Zkusme opět naše dvě otázky.

```
?- predek4(jan,X).       % Kdo je potomkem Jana ?
     X = josef   ;
     X = hugo    ;
     X = adam    ;
     no

?- predek4(X,hugo).      % Kdo je předkem Huga ?
     X = kunhuta ;
     X = zibrid  ;
     X = ruzena  ;
     X = jan     ;
     X = lucie   ;
     X = adam    ;
     X = eva     ;
     no
```

Vidíme, že jsme dostali stejné výsledky, ale v jiném pořadí.

Cvičení

1. Pokuste se charakterizovat pořadí v jakém vydávají řešení procedury predek a predek4 ! Odůvodněte svoje tvrzení.
2. Pokuste se dokázat, že procedura predek4 je procedurálně správná (tj. dá vždy správný výsledek).

A jak to bude vypadat se čtvrtou modifikací naší procedury ?

```
%  ============================================================
%               poslední deklarativně ekvivalentní varianta
%                         procedury predek
%    predek5(?Pred,?Pot)   Pred je předkem potomka Pot
%  ============================================================
predek5(Rod,Pot):-
     rodic(Rod,Pot).
Predek5(Pred,Pot):-
     predek5(MlPred,Pot),
     rodic(Pred,MlPred).
```

Reakce bude následující:

```
?- predek5(jan,X).            % Kdo je potomkem Jana ?
     X = adam    ;
     X = josef   ;
     X = hugo    ;
     error - stack overflow

?- predek5(X,hugo).           % Kdo je předkem Huga ?
     X = adam    ;
     X = eva     ;
     X = jan     ;
     X = lucie   ;
     X = kunhuta ;
     X = zibrid  ;
     X = ruzena  ;
     error - stack overflow
```

Program tedy najde pro tyto dotazy všechna řešení, ale pak zabloudí - a není již schopen oznámit, že jsou řešení všechna.

Cvičení

> Může se stát, že procedura predek5 nenalezne všechna řešení pro některý dotaz (případně v jiné "rodině") ?

I když se to nezdá, dělají začátečníci větší chyby v deklarativním významu jimi konstruovaných procedur, než v procedurálním. Vždy proto **vždy začínejte s úvahami o tom, zda je váš program alespoň deklarativně správný.** Věříme, že vás tato kapitola neodradila. Pojďme však již programovat!

6. Seznamy

Jak jsme si již řekli, pracuje prologovský program s obecnými termy. Programátor si tedy může (a musí) pomocí nich vytvářet vlastní datové struktury sám. Jedinou výjimkou je předdefinovaná datová struktura **seznam**, která slouží k reprezentaci posloupnosti (termů).

Prázdný seznam je označen **atomem [ ],** k reprezentaci neprázdného seznamu slouží binární funktor tečka '.' . **Neprázdný seznam je** tedy tzv. **tečka-dvojice** (terminologie pochází z jazyka LISP)

.(Hlava,Tělo),

kde **Hlava je první prvek seznamu** a **Tělo je seznam tvořený zbývajícími prvky seznamu**.

Tedy například

```
.(a,.(b,.(c,[ ])))                                    (1)
```

je správný zápis seznamu obsahujícího tři prvky a,b,c . Tento zápis je však poněkud těžkopádný, proto je zaveden zjednodušený zápis pomocí výčtu prvků v hranatých závorkách. Uvedený seznam můžeme tedy (ekvivalentně!) zapsat také jako

```
[a,b,c] .                                             (2)
```

Operátor "svislá čárka" '|' , který se používá v tomto typu zápisu, umožňuje vyznačit začátek seznamu. Můžeme tedy ekvivalentně psát také

```
[a|[b,c]] , [a,b|[c]]    .                    (3)
```

Zápis **seznamu s použitím operátoru |** má tvar

[Začátek | Tělo] , kde

Začátek je výčet (ne seznam) prvků stojících na začátku definovaného seznamu

a

Tělo je **seznam** tvořící zbytek definovaného seznamu (přitom je-li tento zbytek prázdný, není třeba jej uvádět). Protože Tělo je také seznam, můžeme pro jeho zápis použít tutéž konvenci.

Pro náš seznam o třech prvcích můžeme tedy kromě výše uvedených zápisů použít i následující (poněkud těžkopádné):

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

Všechny tyto zápisy jsou "zkratkami" pro zápis (1).

Pozor však na zápisy, které vzniknou záměnou operátoru | a čárky ! Těmto chybám, vzniklým většinou přepisem, se ubrání málokdo. Např.

`[ [a,b] , c]`	je dvouprvkový seznam o prvcích `[a,b]` a `c`
`[ [a,b] | c]`	není seznam, protože za operátorem | nenásleduje seznam
`[a,b,[c]]`	je seznam o třech prvcích a,b a [c] (jednoprvkový seznam)
`[a,[b,c]]`	je dvouprvkový seznam o prvcích a, [b,c]
`[a,b,c,[]]`	je čtyřprvkový seznam o prvcích a, b , c a [ ] (tj. prázdný seznam) .

U výkladu způsobu zápisu seznamů jsme se zastavili na tak dlouho proto, že jeho zvládnutí vám ušetří mnoho času a problémů.

6.1. Predikáty prvek a vypust

Prvním predikátem pro práci se seznamy, který vytvoříme, bude bude predikát

```
prvek(Prvek,Seznam),                          (1)
```

který uspěje, je-li Prvek prvkem seznamu Seznam. Definice predikátu prvek je poměrně snadná

```
%  ============================================================
%   prvek(X,Sez)          X je prvkem seznamu Sez
%  ============================================================
prvek(X,[X|_]).                  % X je prvkem seznamu, je-li jeho hlavou
prvek(X,[_|T]):- prvek(X,T).     % X je prvkem seznamu, je-li prvkem jeho těla
```

Díky algoritmu interpretace prologovských programů se druhá klauzule použije až tehdy, nepodaří-li se použít klauzuli první. Specifikace predikátu nám naznačuje, že jej lze použít třemi způsoby:

- jako test (% prvek(+A,+B))

```
?- prvek(b,[a,b,c]).
     yes.
```

Nejprve se Prolog neúspěšně pokusí použít první klauzuli, a tedy substituovat X <- a. Po neúspěchu použije klauzuli druhou - a hned úspěšně, neboť prvek b je hlavou těla seznamu [a,b,c].

- pro hledání prvků seznamu (% prvek(-A,+B))

```
?- prvek(X,[a,b,c]).
     X=a ;          % pomocí první klauzule, po odmítnutí středníkem se použije druhá klauzule
     X=b ;          % a opět první, která uspěje
     X=c ;          % ... atd
     no
```

- pro hledání seznamu (% prvek(+A,-B))

```
?- prvek(a,L).
     L=[a]   ;              % řešení podle první klauzule
     L=[_1|[a]]   ;         % po odmítnutí nové řešení
     L=[_2,_3|[a]]   ;
     L=[_4,_5,_6|[a]]       % atd.
```

Při stálém odmítání výsledků středníkem výpočet nikdy neskončí (ve skutečnosti přeteče paměť). Toto použití predikátu prvek nemá samo o sobě význam, může však být, jak ještě uvidíme, s výhodou použito k definování složitějších predikátů.

Jinou velmi potřebnou úlohou je vypouštění prvku ze seznamu. Zkuste si jej naprogramovat !

Jistě se vám to povedlo a i když ne, získali jste lepší předpoklady ke čtení dalšího textu. Sestrojme příslušnou proceduru společně.

Vypuštění hlavy je velmi snadné - získáme tím tělo seznamu. Jinak hlava přejde do výstupního seznamu a převedeme úlohu vypouštění z celého seznamu na vypouštění z jeho těla.

Uvědomme si, že takto pojatá procedura vypouštění uspěje jen tehdy, když vypouštěný objekt skutečně je prvkem seznamu. Vyskytuje-li se objekt v seznamu vícekrát, je ze seznamu vypuštěn pouze první výskyt tohoto objektu.

```
%  ============================================================
%  vypust(?X,?S,?L)         Seznam L vznikne vypuštěním prvku X
%                               ze seznamu S
%      ?- vypust(b,[a,b,c],L).        L=[a,c]
%      ?- vypust(X,[a,b,c],L).        X=a  L=[b,c]  ;
%                                     X=b  L=[a,c]  ;
%                                     X=c  L=[a,b]  ;
%                                     no
%      ?- vypust(d,L,[a,b]).          L=[d,a,b]     ;
%                                     L=[a,d,b]     ;
%                                     L=[a,b,d]     ;
%                                     no
%  ============================================================
vypust(X,[X|T],T).                  % vypuštěním hlavy seznamu
                                    % získáme jeho tělo
vypust(X,[Y|T],[Y|L]):-             % nevypouštíme-li hlavu,
               vypust(X,T,L).       % vypustíme z těla
```

Vzhledem ke vlastnostem sestrojeného predikátu vypust jej lze použít jako alternativu k predikátu prvek, můžeme tedy definovat:

```
prvek1(X,Seznam):- vypust(X,Seznam,_).
```

Někdy potřebujeme vypouštění poněkud jiných vlastností.

První takovou modifikací je predikát vypust1, který uspěje i v případě, že vypouštěný objekt v seznamu není.

```
%  ============================================================
%  vypust1(?X,?S,?L)    Seznam L vznikne ze seznamu S případným
%                         vypuštěním objektu X
%    ?- vypust1(a,[b,c,d],L).            L=[b,c,d]
%    ?- vypust1(a,[b,a,c,d],L).          L=[b,c,d]
%    ?- vypust1(a,[a,b,a,c,d],L).        L=[b,a,c,d]
%  ============================================================
vypust1(X,[X|T],T).
vypust1(X,[Y|T],[Y|L]):- vypust1(X,T,L).
vypust1(X,[],[]).
```

Druhou modifikací je predikát vypustvsechny, který vypustí ze seznamu všechny výskyty daného prvku.

```
%  ============================================================
%   vypustvsechny(+X,+S,-L)    vypuštěním všech výskytů X ze
%                              seznamu S vznikne seznam L
%   ?- vypustvsechny(d,[a,b,c],L).        L=[a,b,c]
%   ?- vypustvsechny(d,[a,d,b,d,c,d],L).  L=[a,b,c]
%  ============================================================
vypustvsechny(X,[X|T],L):-          % je-li X hlavou, nepřejde do
          vypustvsechny(X,T,L).     % výsledku, pokračuji ve
                                    % vypouštění z těla seznamu
vypustvsechny(X,[Y|T],[Y|L]):-      % jestliže se použila tato
          vypustvsechny(X,T,L).     % klauzule, není X hlavou
                                    % seznamu v druhém argumentu,
                                    % stačí vypouštět z těla
vypustvsechny(X,[],[]).             % vypuštěním z prázdného seznamu
                                    % získáme opět prázdný seznam
```

6.2. Další jednoduché predikáty pro práci se seznamy

V tomto odstavci si ukážeme konstrukci několika dalších predikátů pracujících se seznamy. Půjde však spíše o programátorská cvičení. Kromě specifikujícího komentáře uvádíme vždy i jedno typické volání daného predikátu.

a) poslední prvek seznamu

```
%  ============================================================
%     posl(+S,-X)   X je posledním prvkem seznamu S
%      ?-posl([a,b,c,d],X).   X=d.
%  ============================================================
      posl([X],X).
      posl([X|T],Y):- posl(T,Y).

%  ============================================================
%     poslzbyt(+S,-Zbyt,-X)   X je posledním prvkem seznamu S,
%                               Zbyt příslušný začátek seznamu
%    ?- poslzbyt([a,b,c,d],Z,X).   X=d, Z=[a,b,c].
%  ============================================================
      poslzbyt([X],[],X).
      poslzbyt([X|T],[X|T1],Y):- poslzbyt(T,T1,Y).
```

b) první prvek seznamu

```
%  ============================================================
%     prv(+S,-X)              X je prvním prvkem seznamu S
%      ?-prv([a,b,c,d],X).   X=a.
%  ============================================================
      prv([X|_],X).

%  ============================================================
%     prvzbyt(+S,-ZBYT,-X)    X je prvním prvkem seznamu S,
%                               Zbyt příslušný konec seznamu
%     prvzbyt([a,b,c,d],Z,X).   X=a, Z=[b,c,d].
%  ============================================================
      prvzbyt([X|T],T,X).
```

c) prostřední prvek seznamu

```
%  ============================================================
%     prostr(+S,-X)          X je prostředním prvkem seznamu S
%                              je-li S sudé délky, bere se první
%                              "ze dvou prostředních prvků"
%       ?-prostr([a],X).                X = a.
%       ?-prostr([a,b,c,d,e],X).        X = c.
%       ?-prostr([a,b,c,d,e,f],X)       X = c.
%  ============================================================
```

Sestavení příslušného programu není tak snadné. Každého jistě napadne řešení používající délku seznamu a spočítání kolikátý prvek je prostřední. Tuto ideu však nemůžeme s našimi dosavadními znalostmi jazyka v Prologu realizovat, protože dosud nemáme k dispozici vůbec žádné prostředky pro počítání s čísly. Přesto však můžeme úlohu vyřešit. Musíme však naši proceduru založit na jiném - a elegantnějším - principu:

Vyšleme od počátku seznamu směrem k jeho konci dva signály: jeden rychlý a druhý dvakrát pomalejší. V okamžiku, kdy první signál dojde na konec seznamu, bude druhý signál právě v prostředku seznamu.

```
%  ============================================================
%     prostr(+S,-X):- prost(+S,+S,-X).
%                 První argument se bude zkracovat dvakrát rychleji
%                  než argument druhý. V okamžiku, kdy první seznam
%                  v prvním argumentu je jeden nebo dva prvky,
%                  bude hlava druhého argumentu prostředním prvkem
%                  původního seznamu
%       ?-prost([a],[a],X).                        X = a.
%       ?-prost([a,b],[a,b],X).                    X = a.
%       ?-prost([a,b,c,d,e],[a,b,c,d,e],X).        X = c.
%       ?-prost([a,b,c,d,e,f],[a,b,c,d,e,f],X)     X = c.
%  ============================================================
prost([U],[X|T],X).          % původní seznam byl liché délky
prost([U,V],[X|T],X).        % původní seznam byl sudé délky
prost([U,V|T1],[W|T2],X):-   % první argument se zkracuje
     prost(T1,T2,X).         % dvakrát rychleji než druhý
```

d) rozdělení seznamu na seznamy lichých a sudých členů

```
%  ============================================================
%     rozdel(+S,-L1,-L2)   rozděluje seznam S na dva :
%                            L1 je seznam prvků na lichých místech
%                              seznamu S
%                            L2 je seznam prvků na sudých místech
%                              seznamu S
%     ?-rozdel([a,b,c,d,e,f,g],S1,S2)   S1=[a,c,e,g] , S2=[b,d,f]
%  ============================================================
rozdel([X,Y|T],[X|T1],[Y|T2]):- rozdel(T,T1,T2).
Rozdel([X],[X],[]).
rozdel([],[],[]).
```

Predikát rozděl jde používat plnohodnotně „oběma směry“, tedy i pro slévání dvou seznamů. Zkuste si rozmyslet, zda to platí i pro jiné z předcházejících predikátů pro práci se seznamy.

6.3. Zřetězení a obracení seznamu. Použití akumulátoru.

V tomto odstavci sestrojíme několik predikátů, které mají pro práci se seznamy klíčový význam. Mimoto se budeme zabývat i otázkou výpočetní složitosti jednotlivých predikátů.

Prvním z predikátů bude predikát **conc, který realizuje zřetězení dvou seznamů**.

```
%  ============================================================
%  conc(?L1,?L2,?L)   seznam L vznikne zřetězením seznamů L1 a L2
%    ?- conc([a,b,c],[1,2,3,4],L).       L=[a,b,c,1,2,3,4]
%    ?- conc([a,b,c],L,[a,b,c,1,2]).     L=[1,2]
%    ?- conc(L,[a,b,c],[1,2,a,b,c]).     L=[1,2]
%  ============================================================
conc([],L,L).              % přiřetězení seznamu L za prázdný seznam dá seznam L
conc([X|T],L,[X|S]):-      % hlavou zřetězeného seznamu je hlava prvního seznamu,
             conc(T,L,S).  % jeho tělo vznikne přiřetězením druhého seznamu za
                           % tělo prvního
```

Velmi často potřebujeme **seznamy**, se kterými pracujeme, **otáčet**. Nejjednodušším způsobem, jak seznamy otáčet, je následující predikát obrat, který používá predikát conc:

```
%  ============================================================
%  obrat(+L1,-L2)    seznam L2 vznikne otočením seznamu L1
%    ?- obrat([a,b,c],L).           L=[c,b,a]
%  ============================================================
obrat([],[]).                  % Obrácený prázdný seznam je opět prázdný
obrat([X|T],L):-               % Obrácený seznam vznikne tak, že
     obrat(T,T1),              % obrátíme tělo seznamu
     conc(T1,[X],L).           % a za něj přiřetězíme jednoprvkový seznam
                               % obsahující hlavu posledního seznamu
```

Zamysleme se nad výpočetní složitostí predikátů conc a obrat. Je zřejmé, že čas výpočtu predikátu conc roste lineárně s délkou prvního seznamu. Pravidlo v druhé klauzuli se použije tolikrát, jak dlouhý je první seznam.

Zkuste si rozmyslet, zda by se zřetězení seznamů dalo realizovat rychleji!

Protože predikát obrat volá predikát conc tolikrát, kolik činí délka obraceného seznamu, je složitost výpočtu podle predikátu obrat kvadratická.

Založíme-li obracení seznamu na jiném principu, než je použití predikátu conc, **můžeme jej realizovat v lineárním čase**. Tento postup je speciálním případem tzv. **techniky akumulátoru**, což je jeden z nejvděčnějších obratů programování v Prologu (i LISPu). K obracení seznamu použijeme ternální predikát concrev:

```
%  ============================================================
%  concrev(+L,+T,-S)  S je zřetězení obráceného seznamu L
%                       se seznamem T
%      tedy predikát ot(+L,-S):- concrev(L,[],S) otáčí seznam
%      ?-concrev([a,b,c],[1,2],S).    S=[c,b,a,1,2]
%  ============================================================
concrev([],L,L).
concrev([X|T],L,P):- concrev(T,[X|L],P).
```

První argument predikátu concrev je vstupní, druhý je také vstupní, hraje však speciální úlohu, vytváří se v něm postupně v jednotlivých voláních predikátu konečná hodnota, která se do výstupního parametru zkopíruje, je-li splněna koncová podmínka - v tomto případě, že je první vstupní argument již prázdný.

Protože je tento postup velmi důležitý, zkusme trasovat výpočet dotazu

```
?- concrev([a,b,c,d],[1,2],L).
```

první argument	**druhý argument -akumulátor**	**výstupní argument**
[a,b,c,d]	[1,2]	nekokretizováno
[b,c,d]	[a,1,2]	nekokretizováno
[c,d]	[b,a,1,2]	nekokretizováno
[d]	[c,b,a,1,2]	nekokretizováno
[]	[d,c,b,a,1,2]	[d,c,b,a,1,2]

S použitím predikátu concrev můžeme pak definovat nový predikát pro obracení seznamů:

```
obrat1(S,ObrS):- concrev(S,[],ObrS)  ,
```

který obrací seznam v lineárním čase**.**

I když se tím dopouštíme velmi velkého zjednodušování, můžeme vám dát jednu radu:

> *Máte-li problémy s konstrukcí nějakého prologovského predikátu, často vám může pomoci hledat jiný predikát o více parametrech, ze kterého potřebný získáte voláním pomocného predikátu se speciální iniciální hodnotou některých z nich.*

7.Aritmetika

I programovací jazyky, které neslouží k programování numerických výpočtů, se neobejdou bez prostředků pro práci s čísly. Výjimkou není ani Prolog. Práce s čísly v něm však má svoje specifika. Začněme příkladem.

Jaká bude odpověď na dotaz

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

Pokud jste tipovali, že X=3, pak jste na dosavadní výklad nedávali příliš pozor. Funktor = uspěje, pokud se oba argumenty povede unifikovat. Odpověď tedy bude

```
X = 1+2                                  ,
```

neboť k unifikaci vede substituce X <- 1+2 .

To moc nepotěší. Pokud chceme, aby se provedl výpočet a ne unifikace, musíme použít speciálního operátoru is. Tedy

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

Cíl s použitím operátoru is může mít tvar:

```
X is V ,
```

kde X je proměnná a V je aritmetický výraz. Všechny proměnné, které se (případně) vyskytují ve výrazu V, musí být v okamžiku splňování cíle konkretizovány na číselnou hodnotu, jinak dojde k běhové chybě. Obdobně dojde k běhové chybě, pokud proměnná X je již konkretizována, avšak na nečíselnou hodnotu.

Při splňování daného cíle se vyčíslí hodnota aritmetického výrazu V - nechť je to z - a potom

- pokud proměnná X není konkretizována,
  provede se substituce X < z a cíl uspěje
- pokud proměnná X je konkretizována na číselnou hodnotu (označme ji y),
  porovnají se hodnoty z a y. Pokud jsou stejné, cíl uspěje, jinak selže.

Aritmetické **vyhodnocení vynucují i** aritmetické **relační operátory**:

=:=	aritmetická rovnost,
=\=	aritmetická nerovnost,
<	je menší,
>	je větší,
=<	je menší nebo rovno,
>=	je větší nebo rovno.

Na obou stranách od relačního operátoru musí stát aritmetický výraz, jehož všechny proměnné jsou v okamžiku splňování cíle již konkretizovány na číselnou hodnotu. Není-li tomu tak, dojde k běhové chybě.

Při pokusu o splnění cíle konstruovaného z aritmetických relačních operátorů se vyhodnotí oba výrazy a provede se příslušné porovnání získaných číselných hodnot - jsou-li v daném vztahu, cíl uspěje, nejsou-li, selže.

Ilustrujme právě vyloženou látku na krátké konverzaci:

```
?- X is Y+1.                    % běhová chyba - nedefinovaný aritmetický výraz

?- 2+3 = 3+2.
     no                         % tyto dva termy nelze unifikovat

?- 2+3 =:= 3+2.
     yes.                       % oba aritmetické výrazy mají stejnou hodnotu

?- X is 2+3, Y is 2*X , Z is 12-3, Y is Z .
     no                         % za jednotlivé proměnné se postupně substituovala čísla
                                % X=5 , Y=10  a Z = 9,  při vyhodnocování predikátu Y is Z
                                % byla tedy proměnná jY již konkretizována  a její hodnota
                                % různá od hodnoty proměnné Z

?- X<5.                         % běhová chyba - výraz vlevo nelze vyčíslit

?- X is 5 , X is X+1 .
     no                         % za X substituovalo 5, což není rovno 6
```

Viděli jsme tedy, že **operátor is není přesnou analogií "přiřazovátka" :=** , které znáte z Pascalu. Jestliže již jednou byla proměnná X konkretizována, není již možno později tuto substituci změnit (při postupu výpočtu vpřed). K takové změně může dojít jen tak, že je při navracení výpočtu zpět tato substituce odvolána.

Z významu standardního operátoru is a relačních operátorů

(=:= , =\= , < , > , =< , >=)

je zřejmé, že nemá dobrý význam, aby se je pokoušel interpret splňovat znovu hledáním jiných hodnot příslušných aritmetických výrazu. V tomto smyslu mluvíme o tom, že **aritmetické operátory nebacktrackují.**

Vytvořme s pomocí aritmetických operátorů několik velmi jednoduchých predikátů.

```
%  ============================================================
%   soucet(+Sezn,?Souc)   uspěje, je-li Sezn seznam utvořený
%                           z čísel a spočítá  do proměnné Souc
%                           součet prvků tohoto seznamu
%  ============================================================
soucet([],0).                       % součtem prázdného seznamu je nula
soucet([N|L],S) :-                  % součet seznamu vznikne přičtením
     soucet(L,S1), S is S1+N.       % součtu jeho těla k hlavě
```

Cvičení. Co "budou počítat" následující modifikace predikátu soucet ?

```
soucet1([],0).
soucet1([N|L],S) :-  S = S1 + N , soucet1(L,S1).

soucet2([],0).
soucet2([N|L],S) :-  S is S1 + N , soucet2(L,S1), .

soucet3([],0).
soucet3([N|L],S) :-  soucet3(L,S1), S = S1 + N .
```

V prologu nemáme k dispozici datovou strukturu pole, na kterou jsme zvyklí z jiných programovacích jazyků. Nahradit je mohou seznamy, musíme si však uvědomit, že nemáme k dispozici mechanismus indexování a přístup k jednotlivým prvků seznamu trvá rozdílně dlouho. **Přístup k n-tému prvku** seznamu zprostředkuje následující procedura.

```
%  ============================================================
%   nty(+N,?Sezn,?K)      K je N-tým prvkem seznamu Sezn
%     ?- nty(3,[a,b,c,d],X).        X=c
%  ============================================================
nty(1,[H|_],H).
nty(N,[_|L],K):- N1 is N-1, nty(N1,L,K).
```

Cvičení

> Sestavte další procedury umožňující provádět na seznamech operace, které používáme u polí:
>
> ```
> dosad_za_nty(+N,?K,?Sezn,?NSezn)
>       % seznam NSezn vznikne ze seznamu Sezn tak, že změníme N-tý prvek
>       % na hodnotu K
>
> vypust_nty(+N,?Sezn,?NSezn)
>       % seznam NSezn vznikne ze seznamu Sezn vypuštěním N-tého prvku
>
> vloz_nty(+N,?K,?Sezn,?Nsezn)
>       % seznam NSezn vznikne ze seznamu Sezn vložením prvku K na N-té místo
> ```

Často potřebujeme zjistit **délku seznamu**, to zprostředkuje následující procedura:

```
%  ============================================================
%   delka(+Sezn,?Del)     uspěje, je-li Sezn seznam, Del je
%                           počet jeho prvků
%     ?-delka([a,b,c],N).           N = 3
%  ============================================================
delka([],0).
delka([H|L],N):- delka(L,N1), N is N1+1.
```

Jistě nejpopulárnějším číselným algoritmem je Eukleidův algoritmus, zjišťující nejmenšího společného dělitele dvou čísel. Uvádíme dvě verze (z pedagogických důvodů bez komentářů - doplňte si je !).

```
nsd(A,A,A).
nsd(A,B,D):- A>B, A1 is A-B, nsd(A1,B,D).
nsd(A,B,D):- A<B, B1 is B-A, nsd(A,B1,D).

nsd1(A,A,A).
nsd1(A,B,D):- A>B, A1 is A-B, nsd1(A1,B,D).
nsd1(A,B,D):- nsd(B,A,D).
```

Cvičení

> Sestavte predikát `nsdz(+Sezn,?N)` , který pro seznam čísel Sezn spočítá největšího společného dělitele jeho prvků.

Základem algoritmů vnějšího třídění je algoritmus **slévání dvou uspořádaných posloupností do jedné**. Naprogramujme ho:

```
%  ============================================================
%   slej(+S1,+S2,-S)       uspořádaný seznam S vznikne slitím
%                          uspořádaných seznamů S1 a S2
%     ?-slej([2,4,8],[1,2,3,5],L).   L=[1,2,2,3,4,5,8]
%  ============================================================
slej([X|T],[Y|S],[X|L]):- X<Y, slej(T,[Y|S],L).
slej([X|T],[Y|S],[Y|L]):- Y<X, slej([X|T],S,L).
slej([X|T],[X|S],[X,X|L]):- slej(T,S,L).
Slej([X|T],[],[X|T]).
Slej([],[X|T],[X|T]).
slej([],[],[]).
```

Procedura slej má jednu zajímavou vlastnost - v každém případě lze použít **právě jednu** z jejích klauzulí. To velmi pomáhá zejména při ladění programu. Takovým procedurám se někdy říká **deterministické**. Pokud bychom o tuto vlastnost nestáli, mohli bychom poslední tři klauzule nahradit následujícími dvěma:

```
slej(T,[],T).
slej([],T,T).
```

7.1. Třídící algoritmus quicksort

Na závěr této kapitoly si ukážeme, jak snadno se v Prologu napíše algoritmus třídění **quicksort**.

Quicksort je rekursivní algoritmus, který na základě čísla, kterému se obvykle říká **pivot**, rozdělí tříděný seznam na dvě části. V jedné jsou čísla menší než pivot, v druhé zbylá. Obě tyto části pak setřídí stejným postupem (část délky jedna pochopitelně již třídit netřeba).

Rozdělení seznamu podle pivota realizuje následující predikát:

```
%  ============================================================
%   split(+Pivot,+Vstup,-Male,-Velke)   rozdělí vstupní seznam
%                        Vstup na dva seznamy :
%                Male  je tvořen prvky menšími než Pivot
%                Velke je tvořen prvky většími nebo rovnými než Pivot
%      ?- split(6,[12,4,2,87,10,3],U1,U2).
%                          U1=[4,2,3]   U2=[12,87,10]]
%  ============================================================
split(_,[],[],[]).                    % rozdělením prázdného seznamu
                                      % dostaneme dva prázdné seznamy
split(Pivot,[Y|Zbytek],[Y|Male],Velke) :-
               Pivot>Y,                          % Y patří mezi malé
               split(Pivot,Zbytek,Male,Velke).   % rozdělení zbytku
split(Pivot,[Y|Zbytek],Male,[Y|Velke]) :-
               Pivot=<Y,                         % Y patří mezi velké
               split(Pivot,Zbytek,Male,Velke).   % rozdělení zbytku
```

S použitím obvyklého predikátu pro zřetězení seznamů conc již snadno zkonstruujeme třídící predikát quicksort (za pivota jsme vzali první prvek seznamu).

```
%  ============================================================
%   quicksort(+Vstup,-Setrideno)
%                          Setrideno je setříděný seznam Vstup
%  ============================================================
quicksort([],[]).
quicksort([Pivot|Zbytek],Setrideno) :-
           split(Pivot,Zbytek,Mala,Velka),
           quicksort(Mala,SetridenaMala),
           quicksort(Velka,SetridenaVelka),
           conc(SetridenaMala,[Pivot|SetridenaVelka],Setrideno).
Conc([],L,L).
conc([X|L1],L2,[X|L3]) :-
          conc(L1,L2,L3).
```

K tomuto programu se ještě vrátíme v deváté kapitole, při diskusi o efektivitě prologovských programů.

8. Eratosthenovo síto

V našem výkladu Prologu jsme již pokročili natolik, abychom mohli sestavit program řešící reálnou úlohu z aritmetiky. Vybereme si implementaci jednoho z nejstarších a nejhezčích postupů, jak zjišťovat prvočísla - tzv.

algoritmu Eratosthenova síta. Tento algoritmus hledá prvočísla tak, že ze seznamu kandidátů systematicky eliminuje čísla, která prvočísly nejsou. Půvab algoritmu spočívá především v tom, že k jeho provádění nemusíme vůbec umět dělit.

Připomeňme si postup Eratosthenova síta:

Máme dáno číslo N a chceme zjistit všechna prvočísla menší nebo rovna než N. Stačí se jistě omezit na hledání lichých prvočísel, jediným sudým prvočíslem je číslo 2.

Vyjdeme ze seznamu kandidátů, který budou na počátku tvořit všechna lichá čísla menší než N. Odtrhneme ze seznamu první prvek (nyní je to 3), který je jistě prvočíslem, a upravíme seznam tak, že z něj vyškrtáme všechny násobky tohoto prvního prvku (k tomu stačí umět jen přičítat). Po této úpravě dostáváme nový seznam kandidátů. Jeho první prvek je opět prvočíslem. Celý postup budeme opakovat, dokud neprojdeme celý seznam kandidátů. (Přesvědčte se sami, že navržený algoritmus skutečně řeší danou úlohu).

Navržený výpočet můžeme urychlit, pokud si uvědomíme, že vyškrtávání můžeme ukončit v okamžiku, kdy první číslo v seznamu kandidátů je větší než druhá odmocnina z N.

Má-li totiž číslo K (<=N) jednoho dělitele většího než $\sqrt{N}$, musí mít i jiného dělitele menšího než $\sqrt{N}$.

Následující program realizující Eratosthenovo síto je velmi podrobně komentován, přesto si však zaslouží několik poznámek, protože jsme v něm použili několik technických "fíglů".

Hlavním predikátem je nulární predikát **prvoc**, který se nejprve uživatele zeptá na číslo N, po té vytvoří pomocí volání predikátu **seznam** seznam kandidátů na prvočísla, po té provede vlastní algoritmus Eratosthenova síta a naonec zavolá predikát **vypis**, který způsobí výstup požadovaného seznamu prvočísel (po 10 prvočíslech na řádce).

Vlastní algoritmus Eratosthenova síta realizuje predikát **udelej**, který má tři argumenty

```
udelej(N,S,List):- vytvor(N,S,[],List)
```

S je seznam kandidátů, List je hledaný seznam (ovšem v obráceném tj. klesajícím pořadí). Důvodem pro takovou konstrukci je skutečnost, že v Eratosthenově sítu zjišťujeme prvočísla v pořadí od nejmenšímu k největšímu. Kdybychom chtěli mít seznam prvočísel stále rostoucí, museli bychom nové prvočíslo přidávat vždy na konec aktuálního seznamu, což je v Prologu zbytečně pomalé. Přidáváme tedy raději na začátek seznamu prvočísel - a dostáváme tak seznam v opačném pořadí. Ten pak na závěr (algoritmem s lineární složitostí) otočíme.

Jak vidíme, provádí celou práci predikátu udelej predikát vytvor. Jeho třetí argument je použit jako akumulátor, ze kterého se hodnota zkopíruje do čtvrtého (výstupního) argumentu v okamžiku, kdy se druhý argument (seznam kandidátů) stane prázdným. K tomu však většinou nedojde (dojde k tomu vůbec ?), protože pokud se zjistí, že první prvek seznamu kandidátů je větší než $\sqrt{N}$, pak již víme, že všechna čísla v seznamu kandidátů jsou prvočísly a můžeme jej rovnou přidat k hodnotě akumulátoru (pomocí predikátu concrev, který provede i nutné otočení).

Pokud je první kandidát ještě menší než N, pak se pomocí predikátu vyh provede vyškrtání násobků tohoto kandidáta ze seznamu kandidátů. Predikát vyh dostává v prvním argumentu hodnotu prvního kandidáta, v druhém číslo od kterého se má ve třetím argumentu vyškrtávat. Druhý argument je inicializován na dvojnásobek prvního kandidáta.

Cvičení

> Predikát vyh je v programu zkonstruován tak, že se snaží vypouštět i sudé násobky prvního kandidáta. To je však zbytečné, neboť v seznamu kandidátů nejsou žádná sudá čísla. Modifikujte program tak, aby vyškrtával jen liché násobky.

Program Eratosthenovo síto

```
%  ============================================================
%  prvoc    hlavní   procedura,  která   obsluhuje  i   komunikaci
%           s uživatelem - vyžádá si vstup přirozeného čísla N -
%           a pak spočítá a vytiskne všechna prvočísla menší nebo
%           rovná tomuto číslu
%  ============================================================
prvoc:- write($Zadej cislo:$),
      read(N), N>1,
      seznam(N,S),          % S je seznam lichých čísel menších nebo rovných než N
      udelej(N,S,ObrPrvoc),          % vlastní Eukleidův algoritmus
      concrev(ObrPrvoc,[ ],Prvoc),   % seznam musíme otočit
      vypis(10,[2|Prvoc]).           % výstup zjistěných prvočísel

%  ============================================================
%  seznam(+N,-S)  vygeneruje do S seznam lichých čísel od 3 do N
%  ============================================================
seznam(N,S):- sezn(3,N,S).

%  ============================================================
%  sezn(+K,+N,-T)  T seznam čísel K, K+2, K+4, ... do N
%  ============================================================
sezn(N,N,[N]).                      % ukončení seznamu
sezn(K,N,[ ])    :- K>N.            % K již do seznamu nepřijde
sezn(K,N,[K|L]):- K<N,              % K bude hlavou
                  K1 is K+2,        % další prvek bude o 2 větší
                  sezn(K1,N,L).

%  ============================================================
%  vypis(+Max,+List)    vypisuje prvky seznamu List, na každé
%                       řádce bude Max prvků
%  ============================================================
vypis(Max,List):- vypis(Max,0,List).

%  ============================================================
%  vypis(+Max,+K,+List) vypisuje prvky seznamu List, na každé
%                       řádce bude Max prvků, K je počet prvků,
%                       které již vystoupili na aktuální řádce
%  ============================================================
vypis(_,_,[ ]):-nl.                 % výpis prázdného seznamu na daném
vypis(Max,Max,L):- nl,              % řádku již vystoupilo Max čísel
            vypis(Max,0,L).         % výstup na další řádek
vypis(Max,K,[H|L]):- K<Max,
            write(H), write(' '),   % výstup čísla H
            K1 is K+1,              % zvětšení počítadla
            vypis(Max,K1,L).

%  ============================================================
%  udelej(+N,+S,-List)  pokud je S rostoucí seznam všech lichých
%          čísel menších nebo rovných N (tj. kandidátů na prvo-
%          čísla), bude List klesající seznam všech prvočísel
%          menších než N
%  ============================================================
udelej(N,S,List):- vytvor(N,S,[ ],List).
```

```
%  ============================================================
%  vytvor(+N,+S,+Acum,-List)  realizace predikátu udelej pomocí
%                             akumulátoru ve třetím argumentu
%                             druhý argument vstup, čtvrtý výstup
%  ============================================================
vytvor(N,[ ],L,L).    % je-li druhý argument prázdný seznam,
                      % zkopíruje se třetí argument (akumulátor) do čtvrtého argumentu
vytvor(N,[H|T],L,P):- H*H=<N,      % H je dost malé
            H2 is H+H,             % H2 je dvojnásobek H
            vyh(H,H2,T,T1),        % T1 vznikne vyhozením násobků čísla H z T
            !,                     % zařízntutí pro jistotu, viz kap. 11
            vytvor(N,T1,[H|L],P).% vstupem ve druhém argumentu se stal
                                   % vyškrtaný seznam kandidátů T1,
                                   % H bylo přidáno jako hlava do akumulátoru
vytvor(N,[H|T],L,P):- H*H>N,       % hlava H vstupního seznamu je již příliš velká
            concrev(T,[H|L],P).    % před nashromážděný seznam [H|L] v
                                   % akumulátoru je nutné přidat otočený zbytek
                                   % vstupního seznamu T

%  ============================================================
%   vyh(+H,+P,+T,-T1)  Seznam T1 vzniká z rostoucího seznamu T
%                        vypuštění všech čísel tvaru
%                        P+k*H pro k=0,1,.....
%  ============================================================
vyh(_,_,[],[]).                              % vyhozením z [ ] vznikne [ ]
vyh(H,Co,[Co|T],L):- Co1 is Co+H,            % vyhodil jsem hlavu Co
                     vyh(H,Co1,T,L).         % budu vyhazovat prvek Co1
vyh(H,Co,[X|T],L):-  Co<X, Co1 is Co+H,      % Co není v seznamu
                     vyh(H,Co1,[X|T],L).     % budu vyhazovat Co1
vyh(H,Co,[X|T],[X|L]):- Co>X,                % hlava X zůstává
                      vyh(H,Co,T,L).         % v seznamu

%  ============================================================
%  concrev(+L,+T,-S)  S je zřetězení obráceného seznamu L
%                       se seznamem T
%  ============================================================
concrev([],L,L).
concrev([X|T],L,P):- concrev(T,[X|L],P).
```

9. Efektivita prologovských programů

V této kapitole se budeme věnovat několika ukázkám efektivních implementací algoritmů v Prologu. K této tématice se ještě vrátíme v kapitole 11 pojednávající o operátoru řezu.

9.1. Fibonacciova posloupnost

S tzv. Fibonacciovou posloupností,

1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 , 55 , 89 , 144 , ...

definovanou rekursivními vztahy

f(1) = 1
f(2) = 1
f(n+2) = f(n) + f(n+1) ,

se můžete setkat velmi často nejen při analýze složitosti algoritmů, ale i v mnoha jiných překvapivých souvislostech (např. tzv. zlatý řez - "dokonalý" poměr stran obrazu, tak ctěný renesančními malíři - jde odvodit z této posloupnosti). O to nám však nejde. Program, který vznikne doslovným přepisem této definice, má exponenciální složitost a může tedy sloužit jako školní příklad, jak se rekurse užívat nemá. V Prologu tento doslovný přepis vypadá takto:

```
fib(1,1).                fib(2,1).
Fib(N,F) :- N>2,
            N1 is N-1, fib(N1,F1),
            N2 is N-2, fib(N2,F2),
            F is F1+F2.
```

V učebnicích programování najdete závěr, že v takovýchto případech je lepší používat místo rekurse iteraci. Co však v Prologu, který je na rekursi "bytostně" založen ? I v něm jde pochopitelně efektivní (lineární) výpočet Fibonacciovy posloupnosti naprogramovat.

```
% ============================================================
%  fibiter(+K,+N,+Predch,+Aktual,-Hodnota)
%           počítá Fibonacciovu posloupnost iterační metodou
%           předpokládá,že v argumentu Aktual je K-tý a v argumentu
%           Predch (k-1)-ní člen Fibonacciovy posloupnosti
%           jsou-li tyto předpoklady při volání splněny,
%           spočte do výstupního parametru Hodnota N-tý člen
%           Fibonacciovy posloupnosti,můžeme tedy definovat
%   fib3(N,F):- fibiter(2,N,1,1,F).
% ============================================================
fibiter(N,N,_,F2,F2).                % Pokud je čítač v prvním argumentu roven N, pak již je
                                     % hodnota N-tého členu ve čtvrtém argumentu
fibiter(M,N,F1,F2,F):- M<N,          % Pokud je čítač ještě menší než N,
         NextM is M+1,               % pak třeba zvětšit čítač o 1
         NextF2 is F1+F2,            % a spočíst novou hodnotu
         fibiter(NextM,N,F2,NextF2,F).   % a celý proces opakovat
                                         % se zvětšeným čítačem
fib3(N,F):-                          % výpočet Fibonacciovy posloupnosti
   fibiter(2,N,1,1,F).               % se převede na volání iteračního predikátu fibiter
                                     % s příslušnou inicializací vstupních parametrů
```

Cvičení

> Naprogramujte iterativní výpočet Fibonacciovy posloupnosti, ve kterém se v každém kroku spočítají dvě nové hodnoty posloupnosti.

9.2. Zřetězení seznamů a rozdílové seznamy

Tento odstavec je poněkud obtížnější, proto vám doporučujeme vrátit se k němu později ještě jednou.

Podařilo se vám realizovat zřetězení seznamů rychleji než lineárně ? Asi ne, ale přesto to jde. Musíme ovšem přejít k jiné reprezentaci seznamů, k **rozdílovým seznamům**, které jsou speciálním případem **neúplně definovaných datových struktur**. Neúplná datová struktura je term, který obsahuje nějakou "volnou" proměnnou tj. proměnnou, za kterou ještě nebylo substituováno.

Budeme seznamy reprezentovat jako "rozdíly", kde menšenec je seznam, ve kterém nahradíme atom [] proměnnou, a menšitel je tatáž proměnná. Seznam [a,b,c] bude tedy reprezentován termem

```
[a,b,c|L]-L   .
```

Ke **zřetězení rozdílových seznamů** stačí překvapivě jediný fakt. Zřetězení tedy lze realizovat **v konstantním čase**. Stačí k tomu fakt:

```
conc1(A-B,B-C,A-C).
```

Ukažme si, jak probíhá zřetězení seznamů [a, b, c | T1] - T1 a [e, f | T2] - T2 pomocí predikátu conc1. Dojde k substitucím:

A <- [a, b, c | T1] ,
B <- T1 , B <- [e, f | T2] ,

tedy musí platit T1 <- [e, f | T2] a A <- [a, b, c | [e, f | T2]] , což není nic jiného než
A <- [a, b, c, e, f | T2] .

Výsledkem tedy bude [a, b, c, e, f | T2] - T2 .

Uvědomte si, že pokud za volnou proměnnou v rozdílovém seznamu substituujeme seznam, např.
`[a,b,c|[d,e]]-[d,e]` ,
není daný term již rozdílovým seznamem (neobsahuje volnou proměnou) .

Sestavme ještě proceduru pro převod seznamu na rozdílový a zpět.

```
%  ============================================================
%   prevod(?ObycSezn,?RozdSezn)
%            převádí seznam ObycSezn na reprezentaci pomocí rozdílových
%               seznamů RozdSezn T1 a zpět
%            ?-prevod([a,b,c],RS).          RS=[a,b,c|T]-T
%            ?-prevod(S,[a,b,c|T]-T).       S=[a,b,c]
%  ============================================================
prevod([],T-T):- var(T).      % predikat var(T) uspěje, pokud T je
                              % dosud volná proměnná viz. kapitola 16
prevod([X|L],[X|T1]-T):- var(T),prevod(L,T1-T).
```

Pro převod ze seznamu na rozdílový bychom mohli volání predikátu var vynechat.

V příštím paragrafu použijeme rozdílových seznamů k odstranění použití predikátu conc z procedury quicksort.

9.3. Efektivnější implementace quicksortu

V odstavci 7.1. jsme sestrojili prologovskou proceduru, která realizuje algoritmus quicksortu. Použití predikátu conc v této proceduře způsobuje její zbytečnou složitost. Použití predikátu conc je však v tomto algoritmu zbytečné, pokud použijeme neúplně definované datové struktury.

```
%  ============================================================
%           Quicksort bez použití predikátu conc
%  ============================================================
%  quick(+Vstup,-Usporadany,+Pomocny)
%       Relace "quick" má tento význam:
%           quick(A,B,C)=sort(A,P),conc(P,C,B)
%               tedy seznam B vznikne připojením seznamu C za
%               seznam P, který  vznikl uspořádáním seznamu A.
%    ?-quick([4,2,7,1],B,[5,6,9]).   B=[1,2,4,7,5,6,9]
%  Definujeme-li tedy predikát
%               quicksort1(A,B):-quick(A,B,[]).
%  dostáváme třídící algoritmus
%  ============================================================
quick([Pivot|Zbytek],S,Konec):-
            split(Pivot,Zbytek,Mala,Velka),    % Predikát split je stejný jako v odstavci 7.1
            quick(Mala,S,[Pivot|Y]),
            quick(Velka,Y,Konec).
quick([],Konec,Konec).

quicksort1(Vstup,Setrideno) :-
            quick(Vstup,Setrideno,[])..
```

Stejnou ideu můžeme popsat i v řeči rozdílových seznamů:

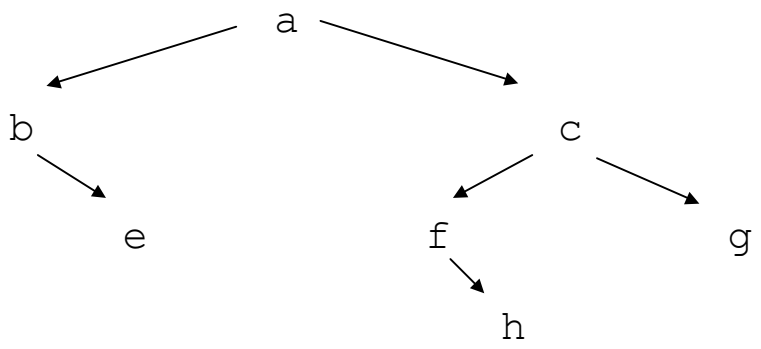
```
%  ============================================================
%    Quicksort bez použití predikátu conc s rozdílovými seznamy
%  ============================================================
quick2([Pivot|Zbytek],A1-Z2) :-
               split(Pivot,Zbytek,Mala,Velka),
               quick2(Mala,A1-[Pivot|A2]),
               quick2(Velka,A2-Z2).
quick2([],Z-Z).
```

10. Stromy

Rekursivní charakter jazyka a to, že "počítá přímo s termy", dělá z Prologu velmi vhodný nástroj pro práci s datovými strukturami, které jsou definovány rekursivně. 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)) )  ).
```

10.1. Průchody stromem

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, 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]).
```

```
?- nas_strom(T),projdi(T,L).          %  po dotazu  dostaneme
T = .......
L = [a,b,e,c,f,h,g]                   %  průchod do hloubky
```

Přidáváme-li podstromy na konec pomocného seznamu, chová se tento jako f r o n t a , 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).   .
```

```
?- nas_strom(T),projdi(T,L).          %   po dotazu
                                      %   dostaneme
T = .......
L = [a,b,c,e,f,g,h]                   %   tedy průchod do šířky   .
```

Cvičení

a) Modifikujte dané procedury pro procházení stromem tak , aby místo vydávání obsahu jednotlivých uzlů stromu do seznamu, volali v každém nějakou proceduru akce.
b) Sestavte procedury na procházení grafu do hloubky a do šířky.

Vhodnou reprezentaci grafu prologovským termem si zvolte sami. Pozor v grafu mohou být cykly !

10.2. 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
```

Cvičení

Modifikujte proceduru tak, aby při odmítání odpovědi na dotaz prvek(-X,+Strom) vydala všechny prvky stromu v rostoucím pořadí.

```
%  ============================================================
%  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
```

Cvičení

> Lze proceduru vloz použít ve tvaru vloz(+X,-Strom,+Strom1) k vypouštění prvků z binárního vyhledávacího stromu? Svoji odpověď odůvodněte!

Cvičení

> Sestavte procedury
> `vylejrost(+Strom,-Sezn)` `vylejkles(+Strom,-Sezn)` ,
> které vydávají v seznamu Sezn prvky obsažené v binárním vyhledávacím stromě Strom
> a) uspořádané od nejmenšího k největšímu resp.
> b) uspořádané od největšího k nejmenšímu

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
```

```
%  ============================================================
%              pomocná procedura pro vypust
%  nejprav(+Vstrom,-Zbytek,-Nejpr)  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
```

Cvičení

Co se stane, odmítneme-li výsledný strom středníkem ? Odstraňte tuto závadu procedury.

11. Operátor řezu

Všechny součásti programovacího jazyka, které jsme dosud probrali, patří (možná s výjimkou aritmetiky a jejího is) do tzv. **čistého Prologu**. Nesnažíme se v něm nijak ovlivňovat algoritmus vyhodnocování dotazů. Za to máme jistotu o poměrně těsné souvislosti deklarativního a procedurálního významu programu. Bohužel však - jak tomu už v programování bývá - musíme často při řešení konkrétních úloh dělat kompromisy a z původních idejí slevovat. Nevystačíme proto jen s "čistým" Prologem.

Ke zvýšení efektivity prologovských programů a k vyjádření některých predikátů, které jsme dosud neuměli reprezentovat, slouží standardní prologovský **predikát řezu**. Značí se znakem vykřičník ! .

Predikát řezu okamžitě uspěje, ale při tom zakáže backtrackování přes sebe zpět.

Např. mějme následující proceduru:

```
a(X,Y):- b(X,Z),!,c(Z,Y).
a(X,Y):- d(X,Y).
```

Splní-li se cíl b(X,Z), splní se okamžitě i následující řez. Když nyní dojde k selhání cíle c(Z,Y) výpočet celé procedury skončí neúspěchem. Kdyby v proceduře řez nebyl, zkoušela by se alternativní unifikace cíle b(X,Z) a po jejím případném selhání by se zkoušel cíl d(X,Y) z druhé klauzule procedury.

V odstavci 6.1 jsme sestrojili proceduru prvek(X,Sezn), která při odpovídání na dotaz

```
?- prvek(-X,+Sezn)
```

v případě odmítání výsledku středníkem (nebo při návratu přes ní, je-li součástí těla nějaké klauzule) vydávala postupně všechny prvky seznamu (pokud se v seznamu opakovaly, pak vícekrát). To nám může někdy vadit. Sestrojme její obměnu, která vydá pouze první prvek, s použitím operátoru řezu:

```
%  ============================================================
%     prvek1(-X,+L)   vydá první výskyt X v L
%                     není-li X v L selže
%  ============================================================
prvek1(X,[X|L]):- !.      % je-li prvek nalezen, je zakázán návrat
prvek1(X,[_|L]):-         % nebyl-li nalezen X v hlavě (jinak se sem výpočet nedostane),
          prvek1(X,L).    % budeme ho hledat v těle
```

Sestrojme proceduru, která vloží do seznamu prvek jen v tom případě, že tam již nebyl:

```
%  ============================================================
%     pridej(+X,+L,-NL)   seznam NL vznikne ze seznamu L
%                    přidání prvku X na jeho začátek ovšem jen
%                    v tom případě, že X v L již není
%  ============================================================
pridej(X,L,L):-
            prvek(X,L),       % je-li X již prvkem L, nepřidám ho
            ! .               % a zakáži návrat
pridej(X,L,[X|L]).            % X není prvkem L (jinak bych se
                              % sem nedostal), mohu ho tedy přidat
```

Cvičení

1) Jak se predikát pridej bude chovat při jiných dotazech, např.

```
pridej(-X,+L,-NL)   , atp.
```

2) Zkuste predikát pridej naprogramovat bez pomoci řezu !

11.1. Negace

Již v druhé kapitole jsme narazili na potřebu binárního predikátu ruzne, který by uspěl, když jeho argumenty nejdou unifikovat. Pomocí řezu ho již sestavíme.

Pokusme se vyjádřit v Prologu tvrzení

"Jana má ráda muže, ale ne plešaté" .

Bez operátoru řezu to nejde. S ním a se standardním predikátem fail, který, je-li volán, okamžitě selže, ji sestavíme poměrně snadno:

```
marada(jana,X):-
            plesaty(X),       % je-li X plešaté uspěje,
            !,                % zakáže návrat
            fail.             % a selže.
Marada(jana,X):-              % k této klauzuli se výpočet dostane, pokud X není plešaté,
            muz(X).           % je-li to muz, má ho Jana ráda
```

Obdobně můžeme **definovat požadovaný predikát ruzne**.

```
%  ============================================================
%       ruzne(X,Y)      uspěje, nejdou-li X a Y unifikovat
%  ============================================================
ruzne(X,Y):-
          X=Y,              % jdou-li X a Y unifikovat
          !, fail.          % zakáži návrat a selžu
ruzne(X,Y).                 % Sem se dostanu jen, nejdou-li X a Y
                            % unifikovat
```

Ve většině implementací Prologu je takový predikát standardní a jmenuje se \= .

Obecně můžeme definovat **predikát negace not** následovně:

```
%  ============================================================
%     not(P)       uspěje, pokud se nepodaří cíl P splnit
%  ============================================================
not(P) :- P, ! , fail.
not(P).
```

Označení predikátu negace není v jednotlivých implementacích Prologu stejné. V některých se zapisuje jako predikát not(P), v jiných **jako unární operátor bez závorek - not P** (tak ji budeme psát v dalším i my).

V některých implementacích Prologu není přípustné psát argument některého predikátu jako cíl, musí být proto předchozí definice negace zapsána takto:

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

kde standardní predikát call zprostředkuje "volání argumentu P jako cíle".

Je lepší používat pro vyjádření negace operátor not, než ji opisovat pomocí operátoru řezu. Lépe se tak v našich programech vyznáme.

Výjimkou z tohoto pravidla jsou však konstrukce typu "podmiňovacího příkazu", např.:

```
a(X,Y):- not b(X),          % pokud b(X) neplatí
         c(X,Y).            % pak c(X,Y)
a(X,Y):- b(X),              % pokud b(X) platí
         d(X,Y).            % pak d(X,Y)
```

V nich se, v případě, že b(X) platí, vyhodnocuje tento cíl zbytečně dvakrát, jak je vidět z rozepsání definice našeho predikátu a:

```
a(X,Y):- neb(x),c(X,Y).
a(X,Y):- b(X),d(X,Y).
neb(X):- b(X),!,fail.
neb(X).
```

Proto je efektivnější (i přehlednější) tento tvar definice predikátu a(X,Y):

```
a(X,Y):- b(X),!,            % platí-li b(X),
         d(X,Y).            % pak d(X,Y)
a(X,Y):- c(X,Y).            % jinak, tj. neplatí-li b(X), c(X,Y).
```

Cvičení

> Jaký je vlastně význam negace, jak je definována v Prologu? Odpovídá negaci, jak ji známe z běžného života, logiky a matematiky ?

Pokud jste odpověděli, že ne, z jiných než psychologických důvodů - Proč by se jinak ptal? , je to úspěch.

Rozdíl je následující:

V běžné řeči, a především **v klasické** (jsou i jiné) **logice , jsou**

výrok A i jeho negace **not A**

výroky o témže světě. Výroky "je hezká" a "není hezká" jsou hodnocením nějakého objektu.

Naproti tomu **v Prologu**

jehezka(X) je konstatováním vlastnosti objektu X,

zatímco jeho negace

not jehezka(X) znamená jen, že z programu **nelze odvodit platnost predikátu jehezka(X).**

Je tedy negace výrok o programu, ne o objektu X, ten (ta) X může být hezká - ba překrásná , ale někdo to zapomněl zanést do programu - a už platí not hezka(X), a pro program X hezká není.

S negací ve smyslu Prologu se setkáváme v demokratických státech u soudu:

Komu jsme nedokázali, že vrah je, ten vrahem (ve smyslu práva) není.

11.2. Důsledky použití řezu

Jak jsme viděli, **umožňuje nám operátor řezu**

- ovlivňovat efektivitu prologovských programů
  (tím, že se nehledá zbytečně v některých větvích prostoru úlohy),
- definovat vzájemně se vylučující použití jednotlivých klauzulí procedury,
- definovat negaci,
- ... .

Přes tyto přínosy se doporučuje **užívat ho "s rozumem" tj. málo**. Většinou totiž ničí deklarativní význam programu.

Uvažujme dvě procedury

```
p :- a, b.          q :- a, !, b.
p :- c.             q :- c.
```

Deklarativní význam procedury p je p <==> (a & b) v c ,
(kde v je logická spojka "vel", "or" - nebo) .

Deklarativní význam procedury q je q <==> (a & b) v (not a & c) .

Zaměníme-li v obou procedurách pořadí klauzulí, dostáváme procedury:

```
p1 :- c.            q1 :- c.
p1 :- a, b.         q1 :- a, !, b.
```

Deklarativní význam procedury p1 je stejný jako procedury p.

Deklarativní význam procedury q1 je však jiný q1 <==> c v (a & b) .

Někdy se rozlišují dva typy řezu:

- **zelený řez** - je takové použití řezu, jehož odstranění nemá vliv na deklarativní význam programu (odřezává jen některé větve výpočtu)
- **červený řez** - je takové použití řezu, které deklarativní význam programu mění.

Řez v procedurách q a q1 je červený.

Červeným řezům bychom se raději měli vyhýbat (ne vždy to však jde).

12. TŘÍDĚNÍ SEZNAMŮ

V předchozím textu jsme si ukázali implementaci třídícího algoritmu quicksort v Prologu. Protože jde o rekursivní algoritmus, není obtížné jej "přepsat do Prologu". Pokud jste si již zkusili naprogramovat i jiné třídící algoritmy, jistě jste zjistili, že je to o něco obtížnější.

V tomto odstavci si ukážeme implementace několika dalších třídících algoritmů a vrátíme se i ke quicksortu. Povšimněte si, jakou roli v našich programech hraje operátor řezu.

Naivní třídění

Tento algoritmus je tak prostoduchý (a tedy i pomalý), že jej uvádíme jen jako **odstrašující případ**. Vychází z myšlenky, systematického generování všech permutací vstupního seznamu tak dlouho, dokud nenajdeme tu, která je uspořádaná.

```
/* (velmi) naivní třídění
   naive_sort(+L,-S)      postupně generuje backtrackováním
                            všechny permutace seznamu L
                            tak dlouho, dokud nevygeneruje
                            uspořádanou
   perm(+L,-P)            P je permutace seznamu L
   ins(+X,+L,-S)          S vznikne vložením prvku X do seznamu L
   sorted(+L)             testuje, zda seznam L je rostoucí
*/

naive_sort(L,S):- perm(L,S),sorted(S).

perm([],[]).
perm([X|T],S):- perm(T,S1),ins(X,S1,S).

ins(X,T,[X|T]).
ins(X,[Y|T],[Y|T1]):- ins(X,T,T1).

sorted([X,Y|Z]):- X<Y, sorted([Y|Z]).
sorted([X]).
sorted([]).
```

Třídění vkládáním

Tento algoritmus třídí rekursivně tak, že do setříděného těla vstupního seznamu vloží na správné místo hlavu tohoto seznamu.

```
/* třídění vkládáním
   insert(+X,+L,-S)       seznam S vznikne vložením prvku X na
                          správné místo do uspořádaného seznamu L
   insert_sort(+[X|T],-S)    nejprve se setřídí tělo T vstupního
                      seznamu, pak se do něj vloží na správné
                      místo hlava X vstupního seznamu
*/

insert_sort([],[]).
insert_sort([X|Tail],Sorted) :-
          insert_sort(Tail,SortedTail),
          insert(X,SortedTail,Sorted).

insert(X,[Y|Sorted],[Y|Sorted1]) :-
          X>Y ,                    % je-li X>Y musí být vložen do těla,
          !,                       % jiná možnost není,proto řez
          insert(X,Sorted,Sorted1).
insert(X,Sorted,[X|Sorted]).
```

Bublinkové třídění

Pravděpodobně nejpopulárnějším třídícím algoritmem je bublinkové třídění, které je založeno na odstraňování inverzí, tj. dvojic sousedních prvků, které nejsou správně uspořádány.

```
/*  bublinkové třídění
    swap(+L,-S)     uspěje pokud je v seznamu L inverze,
                      seznam S potom vznikne odstraněním
                      první z těchto inverzí

    bubble_sort(+L,-S)  pokud je v seznamu L inverze,
                          je odstraněna, je zakázáno vracení
                          a na nový seznam je opět zavolán
                          tentýž predikát
                        pokud inverze v L není, pak tento seznam
                          je již setříděný
*/

bubble_sort(List,Sorted) :-
     swap(List,List1),              % ve vstupním seznamu byla odstraněna jedna inverze
     !,                             % při navracení by byla obnovena, proto řez
     bubble_sort(List1,Sorted).
bubble_sort(Sorted,Sorted).         % v seznamu není žádná inverze, je tedy setříděn

swap([X,Y|Rest],[Y,X|Rest]) :- X>Y. % odstranění inverze na začátku seznamu
swap([Z|Rest],[Z|Rest1]) :-
     swap(Rest,Rest1).                                  % hledání a případné
odstranění inverze v těle seznamu
```

Třídění výběrem

Tento algoritmus je založen na tom, že se vybere nejmenší prvek a umístí se na první místo setříděného seznamu. Po té se totéž opakuje pro zbytek vstupního seznamu.

```
/*  třídění výběrem
   min(+L,-Min)         Min je minimálním prvkem seznamu L
   min(+L,+X,-Min)      Min je minimum z čísla X a prvků seznamu L
   select_sort(+L,-S)   Hlavou seznamu se stane minimum seznamu L,
                        jeho tělem setříděný zbytek L (po odebrání
                        tohoto minima)
*/

select_sort([],[]).
select_sort(T,[X|S]):- min(T,X),
                       del(X,T,T1),
                       select_sort(T1,S).

min([Z|T],X):- min(T,Z,X).
min([Y|T],Z,X):- Y>=Z, min(T,Z,X).
min([Y|T],Z,X):- Y<Z, min(T,Y,X).
min([],X,X).

del(X,[X|T],T):- !.                 % vypuštěním X výpočet okamžitě končí
del(X,[Y|T],[Y|T1]):- del(X,T,T1).
```

13. Reprezentace množin pomocí seznamů

Poměrně často potřebujeme v programech pracovat s (konečnými) množinami. Množiny můžeme reprezentovat mnoha různými způsoby. Při porovnávání jejich vhodnosti záleží především na tom, jaké prvky množiny mohou mít a jaké operace s množinami chceme provádět.

Nejjednodušším způsobem, jak reprezentovat v Prologu množinu, je pracovat s prostým seznamem jejích prvků. Přívlastek prostý znamená, že se žádný prvek nesmí v seznamu vyskytovat vícekrát.

Sestavíme jako programátorské cvičení některé jednoduché predikáty pracující s touto reprezentací množin.

13.1. Množiny reprezentované jako prosté seznamy prvků

```
/*  Množiny reprezentované jako prosté seznamy prvků
 uprava(+Sezn,-Mnoz)       uspěje, je-li Sezn seznam, Mnoz je
                             pak ekvivalentní prostý seznam tj.
                             množina
 vyhod(+H,+Sezn,-Sezn1)    uspěje, je-li Sezn seznam, Sezn1 z něj
                             vznikne vypuštěním všech výskytů
                             prvku H
 vyhod1(+H,+Mnoz,-Mnoz1)   uspěje, je-li Mnoz seznam.
                             Je-li H prvkem seznamu Mnoz, vznikne
                               seznam Mnoz1 ze seznamu Mnoz
                               vynecháním prvního výskytu prvku H,
                             není-li H prvkem Mnoz, je Mnoz1
                               totožné s Mnoz.
                             Je-li tedy Mnoz množina, vznikne
                             množina Mnoz1 případným vypuštěním
                             (jediného) výskytu H
 p_member(+H,+Mnoz,-Mnoz1)   uspěje, je-li H prvkem seznamu Mnoz.
                               Seznam Mnoz1 vznikne ze seznamu
                                 Mnoz vypuštěním prvního výskytu
                                 prvku H;
                               je-li tedy Mnoz množina, pak Mnoz1
                                 je tato množina bez prvku H
 je_mnoz(+Sezn)            uspěje, je-li Seznam množinou (prostý)
 rovny(+Mnoz1,+Mnoz2)      uspěje, pokud jsou Mnoz1 a Mnoz2
                              rovny jako množiny
 sjednoc(+M1,+M2,-Sjedn)   Sjedn je sjednocením množin M1 a M2
                             počítá se jako  zřetězení a následná
                             úprava
 uprava1(+Sezn,-Mnoz)      uspěje, je-li Sezn seznam.
                             V novém seznamu Mnoz má každý prvek
                               o jeden výskyt méně než v původním
                               seznamu Sezn.
                             Pokud je v seznamu  Sezn každý prvek
                               nejvýše dvakrát, je pak Mnoz
                               ekvivalentní prostý seznam
 sjednoc1(+M1,+M2,-Sjedn)    Sjedn je sjednocením množin M1 a M2
                               počítá rekursí přes množinu M1
 prunik(+M1,+M2,-Prunik)     Průnik množin M1 a M2
 rozdil(+M1,+M2,-Rozdil)     Rozdíl množin M1 a M2
*/

 uprava([H|T],[H|T1]):- vyhod(H,T,T2),uprava(T2,T1).
 uprava([],[]).
```

```
vyhod(H,[H|T],T1):- !, vyhod(H,T,T1).
vyhod(H,[K|T],[K|T1]):- vyhod(H,T,T1).
vyhod(H,[],[]).

vyhod1(H,[H|T],T):- ! .
vyhod1(H,[K|T],[K|T1]):- vyhod1(H,T,T1).
vyhod1(H,[],[]).

p_member(H,[H|T],T):- !.
p_member(H,[K|T],[K|T1]):- p_member(H,T,T1).

je_mnoz([X|T]):- not p_member(X,T,_), je_mnoz(T).
je_mnoz([]).

rovny([X|T],M):- p_member(X,M,M1),rovny(T,M1).
rovny([],[]).

sjednoc(M1,M2,S):- conc(M1,M2,S1), uprava1(S1,S).

uprava1([H|T],[H|T1]):- vyhod1(H,T,T2),uprava1(T2,T1).
uprava1([],[]).

sjednoc1([H|T],M,[H|T1]):- vyhod1(H,M,M1), sjednoc1(T,M1,T1).
sjednoc1([],M,M).

prunik([H|T],M,[H|T1]):- p_member(H,M,M1), !, prunik(T,M1,T1).
prunik([H|T],M,T1):- prunik(T,M,T1).
prunik([],_,[]).

rozdil([H|T],M,T1):- p_member(H,M,M1),!,rozdil(T,M1,T1).
rozdil([H|T],M,[H|T1]):- rozdil(T,M,T1).
rozdil([],_,[]).

conc([H|T],S,[H|U]):- conc(T,S,U).
conc([],S,S).
```

Cvičení

> Vytvořte predikát `podmn(Mnoz1,Mnoz2)`, který uspěje právě když je Mnoz1 podmnožinou množiny Mnoz2.

13.2. Množiny reprezentované jako rostoucí seznamy prvků

Asi vás napadlo, že by se práce s množinami zjednodušila, kdybychom o jejich reprezentacích předpokládali víc, například to, že prvky v seznamech jsou uspořádány od nejmenšího k největšímu. Největší výhodou bude skutečnost, že **pak má každá množina právě jednu reprezentaci**.

```
/*  Množiny reprezentované jako rostoucí seznamy
 je_mnozr(+Sezn)              uspěje, je-li Sezn množinou,
                                tj. je rostoucí
 upravar(+Sezn,-Mnoz)         uspěje, je-li Sezn seznam, Mnoz je
                                pak ekvivalentní rostoucí seznam
 rovnyr(+Mnoz1,Mnoz2)         uspěje, pokud jsou množiny Mnoz1
                                a Mnoz2 rovny (jako množiny)
 p_memberr(+H,+Mnoz,-Mnoz1)   uspěje, je-li H pvrkem množiny Mnoz,
                                Mnoz1 je tato množina bez prvku H
 sjednocr(+M1,+M2,-Sjedn)     Sjedn je sjednocením množin M1 a M2
 prunikr(+M1,+M2,-Prunik)     Průnik množin M1 a M2
 rozdilr(+M1,+M2,-Rozdil)     Rozdíl množin M1-M2           */
```

```
je_mnozr([X,Y|T]):- X<Y, je_mnozr([Y|T]).
je_mnozr([X]).
je_mnozr([]).

upravar(Sezn,Mnoz):- uprava(Sezn,PrSezn), %  z minulého příkladu
                     sort(PrSezn,Mnoz).   %  libovolné třídění

rovnyr(M1,M2):- M1=M2.

prunikr([X|T],[X|S],[X|U]):- prunikr(T,S,U).
prunikr([X|T],[Y|S],U):- X<Y, prunikr(T,[Y|S],U).
prunikr([X|T],[Y|S],U):- X>Y, prunikr([X|T],S,U).
prunikr([],_,[]).
prunikr(_,[],[]).

sjednocr([X|T],[X|S],[X|U]):- sjednocr(T,S,U).
sjednocr([X|T],[Y|S],[X|U]):- X<Y, sjednocr(T,[Y|S],U).
sjednocr([X|T],[Y|S],[Y|U]):- X>Y, sjednocr([X|T],S,U).
sjednocr([],S,S).
sjednocr(S,[],S).

p_memberr(H,[H|T],T):- !.
p_memberr(H,[K|T],[K|T1]):- K<H, p_memberr(H,T,T1).

rozdilr([X|T],[X|S],U):- rozdilr(T,S,U).
rozdilr([X|T],[Y|S],[X|U]):- X<Y, rozdilr(T,[Y|S],U).
rozdilr([X|T],[Y|S],U):- X>Y, rozdilr([X|T],S,U).
rozdilr([],_,[]).
rozdilr(T,[],T).
```

14. Vstup a výstup

Vstupy a výstupy jsou v programování velmi důležité. Přesto můžeme pozorovat tendenci, že při výkladu programování se jim věnuje čím dál tím menší pozornost (srovnej např. FORTRAN a Pascal). Pro výklad Prologu i experimentování s ním v jistém smyslu vstup a výstup vůbec nepotřebujeme, protože máme s programem interaktivní kontakt bez programování vstupů a výstupů.

V Prologu je však možno - jako v každém plnoprávném programovacím jazyku - naprogramovat i klasický vstup ze souboru a výstup do souboru.

Při startu programu je **aktuální vstup** nastaven z klávesnice, **aktuální výstup** na obrazovku. Tyto vstupy a výstupy označuje atom user.

Ke změně aktuálního vstupu a výstupu slouží predikáty:

see(F)	nastaví aktuální vstup ze souboru F
seen(F)	uzavře vstupní soubor F a obnoví aktuální vstup z klávesnice (jako by zavolal see(user))
seeing(-F)	dotaz na aktuální vstupní soubor
tell(F)	nastaví aktuální výstup do souboru F
told(F)	uzavře výstupní soubor F a obnoví aktuální výstup na obrazovku (jako by zavolal tell(user))
telling(-F)	dotaz na aktuální výstupní soubor

Základním typem vstupu (resp. výstupu) v Prologu je **vstup (resp. výstup) termů**. Slouží k němu predikáty:

read(?T)	přečte z aktuálního vstupu jeden term (ukončený tečkou) a unifikuje jej s argumentem T
write(+T)	vystoupí instance termu T s právě platnými hodnotami substitucí za proměnné v termu T obsažené

Dalším typem je **vstup a výstup znaků**. Slouží k němu např. tyto predikáty:

get0(?Z)	přečte jeden znak z aktuálního vstupu, uspěje pokud jej lze unifikovat s termem Z
get(Z)	chová se stejně, pouze ignoruje (přeskakuje), řídící znaky (ASCII < 32)
put(Z)	výstup znaku do aktuálního výstupu (ne řídící znaky)
tab(N)	vystoupí N mezer (N přirozené číslo)
nl	přechod na novou řádku (pascalské writeln) .

Dříve než uvedeme několik příkladů, dovolte malé intermezzo.

14.1. Programování cyklů

Tento odstavec vlastně do kapitoly o vstupech a výstupech nepatří, zařadili jsme ho sem proto, že se nám cykly budou při programování vstupů a výstupů hodit.

I když je Prologu vlastní rekursivní programování, někdy potřebujeme naprogramovat i cyklus. Nejjednodušším způsobem je použití **standardního** nulárního **predikátu repeat**.

```
%  ============================================================
%     repeat          nulární predikát okamžitě uspěje
%                       a uspěje vždy i při návratu
%  ============================================================
repeat.
repeat:- repeat.
```

a) cyklus repeat-until

obdobu tohoto druhu cyklu můžeme naprogramovat takto:

repeat, C_1, C_2, C_3, ... ,C_n ,Podm , ! .

Splňování posloupnosti cílů $C_1, C_2, C_3, \ldots, C_n$ se opakuje tak dlouho, dokud není splněn cíl Podm.

```
%  ============================================================
%    vstup     přečte ze vstupu jeden term
%              pokud to není přirozené číslo menší než 100,
%              opakuje výzvu a čtení
%  ============================================================
vstup:- repeat,
        write('Zadej přizené číslo menší než 100:'),  % výzva
        read(N),                 % přečtení termu ze vstupu
        integer(N),              % uspěje je-li term N celé číslo (standardní predikát)
        N>0,                     % uspěje, je-li číslo, které vstoupilo
        N<100,                   % větší než 0 a menší než 100
        ! .
```

b) nekonečný cyklus repeat-fail

Splňování posloupnosti cílů

repeat, C_1, C_2, C_3, ... ,C_n , fail .

vede k nekonečnému opakování splňování posloupnosti cílů

C_1, C_2, C_3, ... ,C_n .

Odůvodněte proč.

c) for cyklus

Cyklus se vzrůstající hodnotou parametru jde poměrně přirozeně naprogramovat pomocí rekurse. Příkladem může být procedura seznam(N,S) z kapitoly 8.

Přímá konstrukce for cyklu je poněkud násilná, asi je lepší používat rekursi. Poslouží nám však jako pěkné cvičení.

Cvičení

Uvažujme predikát for definovaný takto:

```
for(I,I,J):- I>J,!,fail.
for(I,I,J).
for(I,J,K):- J1 is J+1, for(I,J1,K).
```

Dokažte, že procedura

```
p(I,Dolni,Horni,Telo):- for(I,Dolni,Horni), call(Telo), fail.
p(_,_,_,_).
```

způsobí opakovaná splňování cíle Telo s rostoucí hodnotou proměnné I počínaje od hodnoty Dolni až po hodnotu Horni.

Cvičení Modifikujte predikát for, aby šlo zadat i krok.

14.2. Příklady

a) predikát tab

Naprogramujme standardní predikát tab

```
%  ============================================================
%    tab(+N)  předpokládá, že parametr N je přirozené číslo
%               na standardní výstup vystoupí N mezer
%  ============================================================
tab(0).
tab(N):- put(' '), N1 is N-1, tab(N1).
```

b) vizualizace binárního stromu na tiskárně

```
%  ============================================================
%      ukaz(+Strom)    vytiskne binární strom Strom,
%                "položený na bok" : kořen vlevo, listy vpravo
%      uk(+Strom,+Odsaz,+D)  vytiskne strom Strom s kořenem
%                odsazeným o Odsaz mezer od levého kraje,
%                podstormy budou odsazeny o D mezer dál
%  ============================================================
ukaz(Strom):- uk(Strom,0,2).
uk(nil,K,_):-
     tab(K),write(nil),nl.
uk(t(L,V,R),K,D):-
          K2 is D+K,               %   odsazení podstromů
          uk(R,K2,D),              %   pravý podstorm
          tab(K),write(V),nl,      %   kořen
          uk(L,K2,D).              %   levý podstrom
```

c) kopírování souboru F1 do souboru F2

```
%  ============================================================
%     kop(+F1,+F2)  zkopíruje soubor F1 do souboru F2
%                      obnoví původní nastavení vstupních souborů
%  ============================================================
kop(F1,F2):-
     seeing(In),              % zjištění a uschování aktuálních
     telling(Out),            % I/O souborů
     see(F1),                 % otevření zdrojového souboru
     tell(F2),                % otevření cílového souboru
     repeat,                  % opakuj
     read(Term),              % přečti Term
     ((                       % druhá závorka není nutná
       Term=end_of_file,      % test konce souboru
       !,                     % ukončení, je-li konec souboru
       told,seen,             % uzavření souborů (je často zbytečné,
                              % provede se automaticky při otevření nových)
       see(In),tell(Out)      % obnovení původních I/O souborů
      )                       % končí akce na konci souboru
      ;                       % alternativa - soubor F1 nekončí
      (                       % závorka opět není nutná
       write(Term),           % výpis Termu do F2
       fail                   % navracení k počátku cyklu
      )
     ).
```

d) Přečtení řádky z klávesnice a její výpis na obrazovku

```
%  ============================================================
%     ctiradku(-S)  přečte z klávesnice do seznamu S jednu řádku
%                     ukončenou znakem CR (ASCII=13)
%     pisradku(+S)  vypíše na obrazovku seznam znaků S
%  ============================================================
ctiradku(Sez):-
     get0(X),          % přečtení znaku X
     (X=13,            % je-li X CR
      Sez=[],          % ukonči tvorbu seznamu
      !                % zařízni
      ;                % alternativa - x není CR
      ctiradku(S),     % přečti tělo seznamu
      Sez=[X|S]        % X bude hlavou výsledného seznamu
     ).
pisradku([]):- !,nl.   % výstup prázdného seznamu -
                       %   odřádkování a konec
pisradku([X|Z]):-      % výstup neprázdného seznamu -
     put(X),           % výstup prvního znaku
     pisradku(Z).      % výstup zbylých znaků
```

14.3. Definování operátorů

V syntaxi Prologu jsme se kromě obvyklého predikátového zápisu setkali na několika místech i s použitím operátorů, například aritmetických, s operátorem unifikace =, operátorovým zápisem negace a pod. Operátory jsou i symboly :- , čárka či středník.

Operátorový zápis je v Prologu vlastně jen záležitostí vstupu a výstupu. Proto nečiní problém dát programátorovi **možnost definovat si vlastní operátory**. To může značně pomoci čitelnosti programů, protože jsme z řady oblastí na užívání operátorů zvyklí. Ukažme si, jak se to dělá.

K definování operátorů slouží standardní ternální predikát op:

op (?Priorita, ?Asociativita, ?Jmeno) .

Začneme posledním třetím argumentem **Jmeno** , jehož hodnotou je **atom označující příslušný operátor.**

První argument **Priorita** je „číselný“ a **udává prioritu daného operátoru**. Číselná vyjádření priorit se mohou u různých implementací lišit, vždy však **platí, že čím větší číslo, tím operátor váže méně** - má nižší prioritu. Abychom tento argument mohli rozumně využít, musíme znát číselná vyjádření priorit, která naše implementace užívá.

Hodnotou druhého argumentu **Asociativita** může být jeden z následujících, na první pohled tajemných atomů:

xfx	xfy	yfx	pro binární infixové operátory
fx	fy		pro unární prefixové operátory
xf	yf		pro unární postfixové operátory.

Tímto způsobem udáváme počet argumentů (jeden nebo dva) a u unárních argumentů i pozici operátoru. **Symbol f** v těchto atomech **reprezentuje operátor**, **symboly x a y** jeho **operandy**. Mezi x a y je následující rozdíl: na místě symbolu y může stát jakýkoli operand, na místě x pouze operand, který není vytvořen pomocí operátoru se stejnou nebo vyšší prioritou (připomínám vyšší priorita - nižší číslo) než má právě definovaný operátor. Asi Vám to stále není jasné, ukažme si tedy několik příkladů.

Aritmetické operátory by mohly být definovány například takto:

```
op(500, yfx, +).   op(500, yfx, -).     % binární operátory typu sčítání,
op(450, fx, +).    op(450, fx, -).      % unární plus a minus,
op(400, yfx, *).   op(400, yfx, /).     % binární operátory typu násobení.
```

Z těchto definic je jasné, že

term	znamená	protože
a + b + c	(a + b) + c	první argument smí mít stejnou prioritu, druhý musí mít větší
a + b * c	a + (b * c)	operátor * má větší prioritu než operátor +
+ a - - b	(+ a) - (- b)	unární znaménka jsme definovali s větší prioritou než binární -
+ + a	je nepřípustné	znaménko má asociativitu fx, kdyby byla fy, bylo by přípustné

Příklad nám jen posloužil jako ilustrace. Většinou mají znaménka stejnou prioritu jako binární plus a minus.

Pomocí klauzulí

```
op(506,fx,p).   op(505,fy,q).   op(503,yf,r).   op(504,xf,s).
```

jsme definovali čtyři unární operátory a, b, c, d. První dva, p a q, jsou prefixové, druhé dva postfixové. Můžeme tedy psát termy

```
p arg , q arg  , arg r , arg s  ,
```

v nichž arg je atom, který je argumentem. Můžeme psát i následující termy:

```
q q arg        % operátor q na výsledek operátoru q na argument arg,
p p arg        % není přípustné, p má asociativitu fx
p q arg        % operátor p na výsledek operátoru q na argument arg
p arg r r      % operátor p na výsledek dvojnásobného použití operátoru r na argument arg
```

Standardní operátory jsou definovány naprosto stejným způsobem. Potřebujete-li zjistit, jaké operátory jsou právě definovány, můžete se to dovědět dotazem

?- op (Priorita, Asociativita, Jmeno).

musíme pochopitelně získané odpovědi odmítat středníkem, abychom dostali postupně všechny definované operátory (elegantněji by to šlo pomocí operátoru bagof, se kterým se seznámíme v další kapitole).

Kdybychom k programu našich klepen přidali klauzuli

op(vhodné_číslo, xfx, rodic). ,

mohli bychom místo predikátového zápisu `rodic(X,Y)` používat operátorový `X rodic Y.`

15. Natažení programu do paměti, program jako "databáze"

Zatím jsme si neřekli, **jak program** v Prologu **"načteme do paměti",** abychom s ním mohli pracovat.

Slouží nám k tomu dva standardní predikáty:

consult(F)	načte program ze souboru F do paměti, pokud je již v paměti program uložen, přidá přečtené klauzule na konec programu
reconsult(F)	načte program ze souboru F do paměti pokud je již v paměti program uložen, vymaže z něj ty procedury, které jsou obsaženy v souboru a nahradí je jejich verzemi ze souboru

Je tedy třeba **používat reconsult vždy po opravě souboru**. Načteme-li opravený soubor pomocí consult, přidají se konzultované klauzule za stávající - získáme tedy v paměti zcela nesmyslný program. A nejen to - tento program se bude většinou chovat stejně jako program před opravou, neboť se nové verze změněných procedur neuplatní, protože budou v programu až za původními. Tato chyba se začátečníkovi špatně odhaluje - v souboru je všechno v pořádku.

Některé překladače nemají oba predikáty consult a reconsult. Obvykle pak predikát consult pracuje jako reconsult.

Program v Prologu obsahuje typicky také data - jako v našem programu SK z kapitoly 2, je dobrým zvykem uchovávat je v jiném souboru než "vlastní program" a během výpočtu je „nakonzultovat“ pomocí predikátu consult (resp.reconsult). Usnadní to práci s jinými daty.

Velmi často lze místo consult(F) napsat jen [F], a dokonce můžeme napsat stručné [F1,F2,F3] ve významu posupného provádění tří cílů consult(F1), consult(F2), consult(F3).

Soubor, který konzultujeme nemusí obsahovat jen klauzule tvarů, které jsme se dosud naučili, ale také tzv. **„příkazy“.** Formálně jsou to **klauzule s „prázdnou hlavou“.** Mají tvar

```
:- tělo .                    ,
```

kde tělo je posloupnost klauzulí. Tyto příkazy se při konzultování hned provádějí. Velmi často to jsou příkazy consult nebo jiné inicializační akce.

Skládá-li se například náš program z několika částí umístěných v různých souborech, dejme tomu ZDROJ1.PL , ZDROJ2.PL a ZDROJ3.PL, můžeme do prvního z nich přidat klauzuli

```
:- consult('ZDROJ2.PL'), consult('ZDROJ3.PL').
```

Tím se při konzultování programu z prvního souboru „natáhnou“ do paměti i zbylé dva. Chceme-li některé jiné, máme snadnou práci s opravou.

Protože na program V Prologu se jde dívat jako na data obecného interpretu prologovských programů, říká se mu někdy "databáze". A Prolog nám poskytuje i prostředky, jak do této databáze - programu **přidávat nové klauzule, či jiné vyřazovat.** Při tom přidávané klauzule mohou být vytvořeny až při výpočtu programu. Mohou tedy hrát i roli jakýchsi "globálních proměnných", ve kterých si můžeme pamatovat mezivýsledky.

I když se v obecnosti bez těchto prostředků obejít nelze, je dobré snažit se jim vyhýbat. Nejen, že kazí význam programu (modifikace programu během jeho výpočtu se dnes již nenosí ani v asembleru) a ztěžují jeho ladění, ale v Prologu mají navíc negativní vliv na rychlost výpočtu - paradoxně tím více, čím rychlejší kompilátor máme.

V našem textu mnoho příkladů na užití těchto prostředků úmyslně neuvádíme, protože umožňují programovat v Prologu "jako v Pascalu" s použitím "globálních proměnných" a obcházet tak to, co jsme vás chtěli naučit.

Prostředky pro aktualizaci databáze jsou:

assert(+K)	přidá klauzuli K na konec programu v paměti
assertz(+K)	ekvivalentní s assert(K)

asserta(+K)	přidá klauzuli K na začátek programu v paměti
retract(?K)	odstraní z programu v paměti první výskyt klauzule, která jde unifikovat s argumentem K
retractall(?H)	odstraní z programu v paměti všechny klauzule, jejichž hlava unifikuje s termem H

16. Několik standardních predikátů a operátorů

V této kapitole uvedeme bez velkých komentářů několik další standardních predikátů a operátorů, které můžete při programování potřebovat. Na další, ale i na přesný význam těchto, se musíte podívat do manuálu implementace, kterou budete používat.

Hned po informaci, jak nějaký program spustit, je nejdůležitější **informace, jak z něj odejít**. Na prolog většinou neplatí žádné exit, quit či end, ale německé halt.

halt	ukončení práce s Prologem

Většina překladačů má celou unárních standardních predikátů, které **testují, zda argument má danou vlastnost**.

atom(X)	uspěje, je-li X atom
atomic(X)	uspěje, je-li X atom nebo číslo
integer(X)	uspěje, je-li X integer
var(X)	uspěje, je-li X dosud nekonkretizovaná proměnná
nonvar(X)	uspěje, není-li X proměnná nebo je-li to proměnná, která byla již konkretizována

Další dva predikáty umožňují **testovat, zda dva termy jsou ekvivalentní** (před unifikací, ne po ní)

X==Y	uspěje, jsou-li X a Y ekvivalentní termy
X\==Y	uspěje, nejsou-li X a Y ekvivalentní termy (při těchto predikátech nedochází k unifikaci)

např.

```
?- X==Y.
     no                    %  X a Y jsou dvě různé proměnné
?- X=Y, X==Y.
     X=_12   Y=_12    %  unifikací při splňování cíle X=Y byly  proměnné X a Y ztotožněny
```

Další predikát umožňuje **přístup k argumentům struktury a naopak konstrukci struktury z argumentů**:

Term =.. L	uspěje, má-li seznam L jako hlavu hlavní funktor F termu Term a jako tělo seznam argumentů funktoru F v termu T (může se používat obousměrně)

Příklad

Sestrojme predikát , který provede **substituci jednoho termu druhým za všechny výskyty v daném termu.**

```
/* subst( Vstup, Co, Cim, Vystup)
  provede v termu Vstup substituci za všechny výskyty (pod)termu Co termem Cim ; výsledkem je term Vystup
*/

subst(Vstup,Vstup,Vystup,Vystup).          % Substituuje se za celý term
subst(Vstup, _ , _ ,Vstup):-
             atomic(Vstup).                % Jinak, není-li Vstup struktura, substituovat nelze
```

```
subst(Vstup,Co,Cim,Vystup):-
            Vstup =.. [Funktor|Argumenty] ,          % rozebereme strukturu
            subst_v_seznamu(Argumenty,Co,Cim,Vyst_Argumenty),
                                        % provedeme substituce na všech argumentech
            Vystup =..[Funktor|Vyst_Argumenty].      % a zase ji složíme zpět

subst_v_seznamu([Hlava|Ocas],Co,Cim,[NovaHlava|NovyOcas]) :-
            subst(Hlava,Co,Cim,NovaHlava),
            subst_v_seznamu(Ocas,Co,Cim,NovyOcas).
subst_v_seznamu([ ],_ ,_ ,[ ]).
```

Obdobnou roli hrají i další dva standardní predikáty

functor(?Term,?F,?Arita)	uspěje, je-li F hlavní funktor termu Term o aritě Arita
arg(+N,?Term,?Arg)	uspěje, je-li Arg N-tým argumentem termu Term

Další predikát umožňuje **vytvářet z atomů znakové řetězce a naopak**:

name(?Atom,?List)	uspěje, je-li List seznam ASCII kódů znaků atomu Atom

Uvědomme si, že znakové řetězce jsou v Prologu ekvivalentní se seznamy ASCII kódů jednotlivých znaků.

Příklad

Předpokládejme, implementace, se kterou pracujeme dává k dispozici má k dispozici standardní predikát dos, který zavolá příkaz operačního systému (vpodstatě každý překladač nějaký takový predikát má). Sestavíme predikát e(F), který zavolá váš oblíbený editor (dejme tomu, že se volá příkazem dosu „turbo <jmeno souboru>“) na soubor F a po provedení oprav nakonzultuje novou verzi programu.

```
e(F):-  name(F,RetF),                          % F atom, RetF znakový řetězec
        conc(RetF,".PL",RetFsPrip),            % RetFsPrip je úplný název souboru s příponou PL
        conc("turbo ",RetFsPrip,RetPrikaz),    % konstrukce řetězce příkazu dosu
        name(Prikaz,RetPrikaz),                % převod řetězce na atom
        dos(Prikaz),                           % volání editoru příkazem operačního systému
                    % zde pracuje editor, po jeho ukončení se bude splňovat další predikát
        reconsult(F).                          % rekonzultování opraveného souboru
```

Jistě jste si uvědomili, že musíte na konci práce s editorem opravovaný soubor uložit.

Nakonec jsme si nechali dva velmi užitečné predikáty, které umožňují jednoduše **„nasbírat“ všechny termy mající nějakou vlastnost.** Tato vlastnost může být velmi složitá. Začneme prvním z nich, druhý predikát je jen jeho modifikací.

bagof(+Term,+Cil,-Seznam)	Seznam je seznam všech instancí termu Term, pro něž je splněn cíl Cil

S naším programem SK (Spolku Klepen - podle nového pravopisu už "evropsky" Klepen s velkým K) bychom dostali následující odpovědi:

```
?- bagof(X,rodic(karel,X),L).
    X=_0038  L=[zuzana,alfred,emil]

?- bagof(X,manzdite(X),L).
    X=_0038  L=[josef,hugo,zuzana,alfred,eva,katka,emil]

?- bagof(dvoj(X,Y),vmanzelstvi(X,Y),L).
     X=_0038  L=[dvoj(adam,eva),dvoj(karel,helena),
                 dvoj(zibrid,kunhuta),dvoj(jan,lucie),
                 dvoj(jose,katka),dvoj(eva,adam),
                 dvoj(helena,karel),dvoj(kunhuta,zibrid),
                 dvoj(lucie,jan),dvoj(katka,jose)]
```

Nyní již lze uvést i druhý predikát

setof(+Term,+Cil,-Seznam) Totéž jako bagof, jen v seznamu Seznam je každý prvek právě jednou (jde o množinu reprezentovanou prostým seznamem)

Ještě na jednu skutečnost musíme upozornit. Oba predikáty se totiž chovají **v případě, že neexistuje instance prvního argumentu, pro kterou je cíl v druhém argumentu splněn**. V tomto případě není „výsledkem“ ve třetím argumentu prázdný seznam, jak byste možná očekávali, ale **oba predikáty** v tomto případě **neuspějí**..

17. Závěrem - aneb co všechno jsme nestihli

Proti původnímu záměru se rozsah textu neúměrně zvětšil a stal se z něho začátek ucelené učebnice Prologu. Přesto v něm mnoho chybí. Namátkou jmenuji

- práce s datovými strukturami
- ukázky typických úloh, které se pomocí Prologu řeší
- více o tom, jak se Prolog překládá
- více o logickém programování
- alespoň jedna větší úloha
- .....

Proč je pro vás dobré se s Prologem seznámit, i když ho třeba nebudete nikdy používat nebo učit ?

- Pomůže vám pochopit, co je to rekurse.
- Dá vám nový pohled na programování. Co je podstatné a co jen technika.
- Snad i víc

Literatura:

[1] Ivan Kalaš : Iné programovanie - Stretnutie s jazykom LISP

ALFA Bratislava 1990 ISBN 80-05-00866-X

[2] Ivan Bratko : Prolog Programming for Artificial Intelligence

Addison-Wesley 1986 ISBN 0-201-14224-4

[3] Petr Jirků a kol: Programování v jazyku Prolog

SNTL Praha 1991 ISBN 80-03-00609-0

OBSAH