Programování v Prologu

7. 10. – 21. 10. 2014

Imperativní a deklarativní programování

Jelikož **Prolog** patří mezi deklarativní jazyky, zatímco běžně užívané programovací jazyky se řadí mezi imperativní, měli bychom si nejprve ujasnit, jak se tato dvě *programovací paradigmata* liší.

Velice jednoduše a přitom poměrně výstižně lze říci, že v **imperativním** jazyce popisujeme, *jak něco udělat* (definujeme postup), zatímco v **deklarativním** popisujeme, *co udělat* (definujeme výsledek).

Rozdělení pouze na imperativní a deklarativní jazyky je ovšem zjednodušené – ve skutečnosti existuje více programovacích paradigmat, která se různě prolínají a navazují na sebe.

Programovací jazyky také často umožňují používat a kombinovat různá paradigmata, takže přesné rozdělení je téměř nemožné. Dokonce i jazyky označované jako deklarativní často obsahují prvky imperativního programování a naopak.

Do imperativní skupiny patří procedurální programování, zatímco do deklarativní řadíme *funkcionální programování* (např. Lisp), *programování s omezujícími podmínkami* (Constraint programming, např. CLR(P)), a *logické programování* (sem patří **Prolog**).

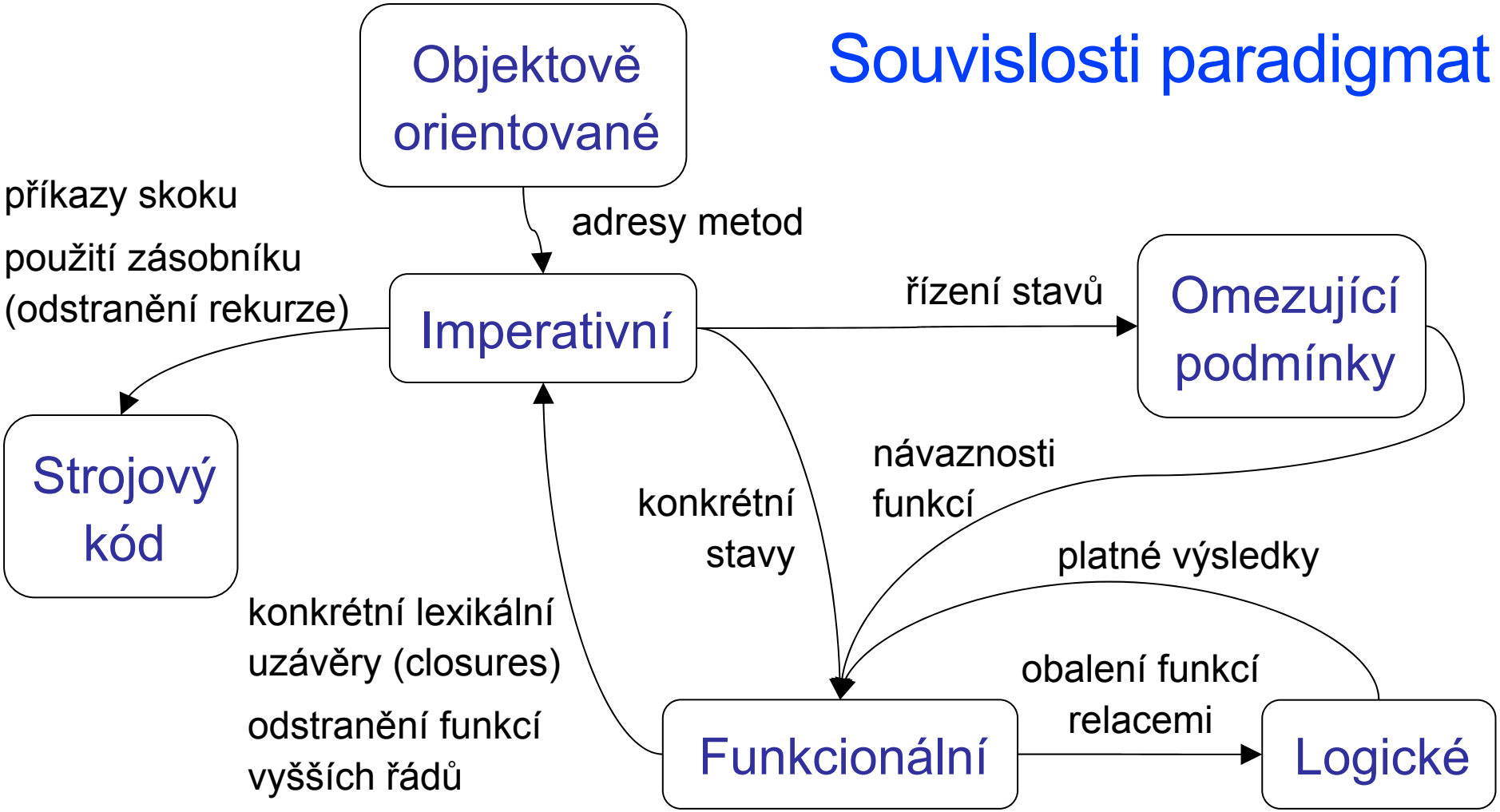
Některá paradigmata, jako objektově orientované programování, nelze jednoznačně zařadit ani do jedné skupiny, respektive, jeho znaky se objevují v obou.

Graf na následujícím snímku považujme jen za ilustrativní, nebude naším cílem přesně rozumnět naznačeným vztahům.

Nicméně, je z něj patrné, že deklarativní jazyky nás více *oddalují od hardwaru*. To na jednu stranu šetří práci, na druhou stranu nás omezují – to, co v jazyce C intuitivně snadno vyřešíme, může v deklarativním jazyce vyžadovat složité konstrukce.

Z toho lze vyvodit, že deklarativní jazyky jsou vhodné jen pro *určité úlohy*, které ale řeší rychle a *efektivně*. My můžeme těžit z toho, že takovou úlohu dokážeme poznat a umíme ji deklarativním jazykem řešit.

Souvislosti paradigmat



Jak již bylo řečeno, i imperativní programovací jazyky obsahují prvky deklarativního programování, i když pochopitelně v omezené míře a nelze je srovnávat s deklarativními jazyky nebo je dokonce použít jako jejich účinnou náhradu.

Nicméně, můžeme se alespoň pokusit v následujících příkladech nastínit vlastnosti deklarativního programování známějšími (imperativními) jazyky.

Příklad 1: Programujeme v javascriptu. Máme pole celých čísel, chceme vypočítat jejich druhé mocniny a uložit je do nového pole.

Ryze imperativním přístupem můžeme úkol splnit následovně:

```
var numbers = [1,2,3,4,5];
var squares = [];

for (var i = 0; i < numbers.length; i++) {
  squares.push(numbers[i] * numbers[i]);
}
console.log(squares); // [1,4,9,16,25]
```

Pokud bychom měli popsat, jak jsme řešení navrhli, bylo by to:

„Projdu všechny položky pole od začátku do konce, každou z nich umocním a přidám na konec nového pole.“

Můžeme ovšem také využít javascriptovou konstrukci Array.map, která na každou položku pole aplikuje funkci a uloží návratovou hodnotu do nového pole:

```
var numbers = [1,2,3,4,5];
var squares = numbers.map(function(n) {
  return n * n;
});
```

Popis toho, co jsme v příkladu předvedli, by vypadal spíše takto:

„Vytvořím nové pole, kde každá položka bude druhou mocninou původní položky.“

Vůbec se tedy nezabýváme tím, jak přesně budou položky z pole získány a jak z nich bude sestaveno nové pole, pouze tím, co chceme provést. To je v podstatě deklarativní přístup.

Lze očekávat, že ve funkci `map` se stane totéž nebo něco velice podobného jako v prvním případě; podstatné ale je, že nás toto nezajímá.

Příklad 2: Programujeme v Javě. Chceme (jedno jak) opět zpracovat pole čísel.

Řešení, které pravděpodobně většinu programátorů napadne jako první, je následující:

```
int[] pole = {1,2,3,4,5,6,7,8,9,10};
for (int i = 0; i < pole.length; i++) {
  //zpracuj pole[i]
}
```

Alternativa je tentokrát méně odlišná, přesto nese stejné znaky, jako v příkladu 1:

```
int[] pole = {1,2,3,4,5,6,7,8,9,10};
for (int hodnota : pole) {
  //zpracuj "hodnota"
}
```

I když víme, že se jedná jen o zkrácený zápis iterace a pole se bude opět procházet od začátku do konce, v podstatě jsme jen řekli, že chceme nějak zpracovat všechny prvky.

Můžeme ještě podotknout, že pole lze v tomto případě zaměnit třeba za množinu a kód pro zpracování nemusíme měnit:

```
Collection<Integer> kolekce
    = new HashSet(Arrays.asList(pole));
for (int hodnota : kolekce) {
  //zpracuj "hodnota"
```

To však už „zavání“ spíše rozebíráním vlastností Javy a polymorfismu, což není směr, kterým bychom se chtěli vydat.

Deklarativní jazyky zpravidla neznají pojem cyklus – je třeba nahradit ho rekurzí.

Pro připomenutí a pro úplnost můžeme proti sobě postavit tradiční příklad výpočtu faktoriálu v iterativní a rekurzivní podobě:

```
int faktorial_iterace(int n) {
  int f = 1;
  for (int i = 2; i <= n; i++)
  {
    f *= n;
  }
  return f;  }
```

```
int faktorial_rekurze(int n) {
  if (n <= 1) {
    return 1;
  }
  return n *
      faktorial_rekurze(n-1);
}
```

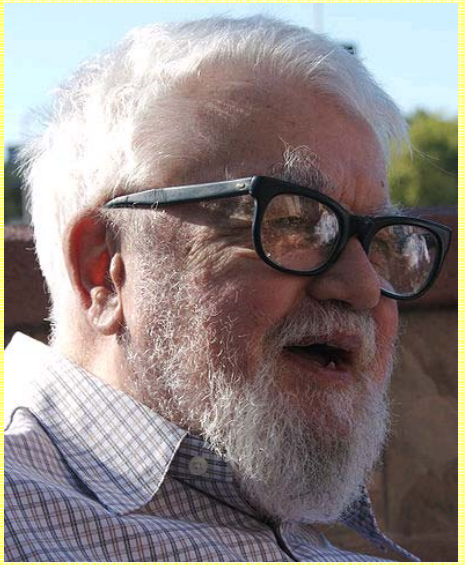
Další deklarativní jazyky

1) Lisp

Lisp je funkcionální programovací jazyk. Byl navržen v roce 1958, což jej dělá druhým nejstarším vyšším jazykem (o rok mladší, než FORTRAN).

Lisp je populární především pro řešení úloh umělé inteligence.

John McCarthy



1927 – 2011

tvůrce Lispu

Program zapsaný v Lispu má stromovou strukturu a používá prefixovou notaci (na prvním místě stojí operátor, další jsou argumenty). Každý příkaz je uzávorkován, což kódu v Lispu dává charakteristický vzhled příkazů s mnoha závorkami.

Příklad 3: (rekurzivní) funkce pro výpočet faktoriálu v Lispu:

```
(defun factorial (n)
   (if (<= n 1)
       1
       (* n (factorial (- n 1)))))
```

2) SQL

SQL možná patří mezi nejznámější deklarativní jazyky. Řadí se do skupiny jazyků pro speciální účely (v tomto případě pro dotazování v databázích).

Protože tyto jazyky nejsou určeny pro obecné použití, nemusejí být ani Turingovsky úplné (nemusí být možné zapsat v nich libovolný program). Lze nicméně dokázat, že SQL je Turingovsky úplný jazyk, i když to nemusí být na první pohled patrné.

Donald Chamberlin

*1944

tvůrce SQL

Příklad 4: Mějme v databázi tabulky klientů a jejich bankovních účtů. Chceme získat tabulku jmen klientů, společně s celkovým obnosem každého z nich.

Raymond Boyce

1947 – 1974

spolutvůrce SQL

```
SELECT `jmeno`, SUM(`zustatek`) AS `penize`
FROM `klienti` LEFT JOIN `ucty`
    ON `klienti`.`id` = `ucty`.`klient`
GROUP BY `klienti`.`id`
```

SQL dotazem říkáme pouze to, že chceme vzít tabulku klientů, připojit k nim jejich účty, seskupit záznamy podle klientů (`GROUP BY`) a v každé skupině sečíst účetní zůstatky (`SUM`).

Historie Prologu

První verze Prologu byla vytvořena v roce 1972 francouzským informatikem Alainem Colmerauerem ve spolupráci Philippem Rousselem na univerzitě v Marseille.

Název Prolog je zkratka z francouzského "PROgrammation en LOGique".

Alain Colmerauer

*1941

tvůrce Prologu

Základem při tvorbě Prologu se stala procedurální interpretace Hornových klauzulí, kterou vymyslel britský informatik (původem z USA) Robert Kowalski.

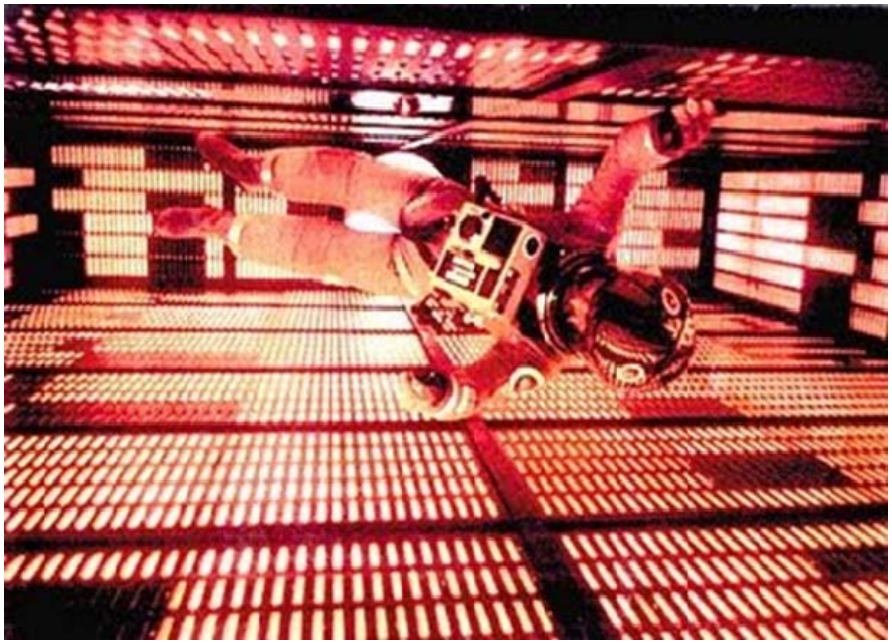
Původně byl Prolog zamýšlen jako samotný základ počítače schopného přirozeně komunikovat s člověkem. Taková myšlenka fascinovala komunitu oblasti umělé inteligence – zvláště proto, že se objevila už v 70. letech.

Robert Kowalski

*1941

autor procedurální interpretace Hornových klauzulí

Nejambicióznějším projektem v tomto ohledu byl japonský program pro vývoj počítače páté generace. Měl být založen plně na logickém programování a ke zvýšení výkonu měl používat masivní paralelizaci.



Dave v Halově "mozku"

Nejlépe takové představě patrně odpovídají superpočítače z dobových sci-fi knih a filmů.

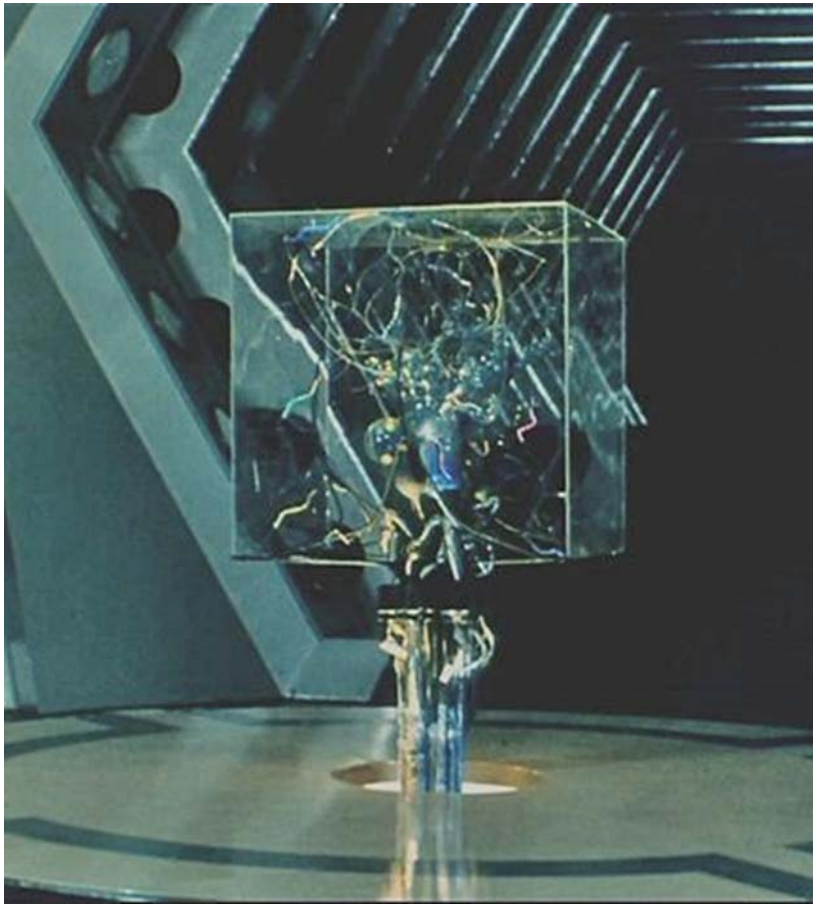
Nutno podotknout, že tehdejší počítače jsou označovány za počítače třetí generace. O tom, jaký bude počítač čtvrté generace, byly zatím vesměs pouze matné představy.

Počítač Colossus z filmu Colossus: The Forbin Project

Označení "počítač páté generace" mělo zdůraznit, že projekt předbíhá svoji dobu.

Nicméně, po investování spousty peněz a deseti letech vývoje byl projekt nakonec zrušen, přičemž nedosáhl zamýšleného cíle.

Problémem byl nedostatek výpočetní síly, obtížné řešení paralelizace a fakt, že tyto specializované systémy byly výkonově brzy překonány běžnými univerzálními procesory.



Centrální mozek lidstva

Projekt byl částečně příliš nadčasový, částečně zkrátka špatně navržený a špatně odhadl vývoj v nejbližších letech – předpokládalo se například, že jednotlivé procesory brzy dosáhnou mezního výkonu, zatímco ve skutečnosti bylo třeba přikročit k paralelizaci až o nějakých 20 let později.

Jediným výsledkem tak zůstala popularizace umělé inteligence a dvou jazyků – Lispu a Prologu, které zůstaly jako jedna z alternativ pro programování běžných počítačů.

Mezitím byl Prolog rozvíjen paralelně univerzitou v Marseille a univerzitou v Edinburghu.

Verze Prologu vyvinutá v Edinburghu (Edinburgh Prolog) je v současnosti považována víceméně za standard a vychází z ní mnoho v současnosti existujících implementací Prologu.

Alternativním standardem je ISO Prolog, udržovaný technologickou komisí v New Yorku.

Použití Prologu v současnosti

Přestože japonský projekt Prolog zpopularizoval, dost možná mu zároveň uškodil, protože skončil neúspěchem – na veřejnosti se tak objevily pochybnosti o samotné použitelnosti Prologu.

Jak je to tedy s Prologem dnes?

- **Otázka**: *Používá se Prolog v komerční sféře?*
- **Odpověď**: Ano, ale moc se o tom nemluví.

- **Otázka:** *Proč ne?*
- **Odpověď:** Není k tomu důvod, zákazníky programovací jazyk obvykle nezajímá.
- **Otázka:** *Tak proč se mluví o Javě a Céčku?*
- **Odpověď:** Prolog patří do skupiny deklarativních jazyků, které se nepoužívají tak často a v takové míře, proto o něm ani není tolik slyšet.
- **Otázka**: *Kde se Prolog vyskytuje?*
- **Odpověď:** Prolog bude obvykle součástí většího systému, jehož další části jsou vytvořeny v jiných jazycích.

V důsledku toho Prolog může v celém systému poněkud "zapadnout". Úkolem Prologu je obvykle řešit problémy z oblasti **umělé inteligence** a **lingvistiky**. Prolog se tak používá k prototypování.

- **Otázka:** *Proč k prototypování?*
- **Odpověď:** Při vytváření prototypu nám jde o co nejrychlejší vytvoření funkční implementace zamýšleného řešení, abychom ověřili použitelnost. Programátor znalý Prologu v něm obvykle dokáže rychle napsat program, třeba i neefektivní. Efektivitu potom můžeme vylepšit tím, že ověřené řešení implementujeme ve vhodnějším jazyce.

- **Otázka:** *Co firmy vede k nepoužívání Prologu?*
- **Odpověď:** Zavedení dalšího jazyka a propojení s již používanými je nákladné. Firmy proto zpravidla váhají, i když v jiném jazyce je řešení problému třeba řádově jednodušší. Nevýhodou Prologu je také nedostatek moderních vývojových prostředí (IDE).

Řečeno o Prologu

"…Comparing the results of this unfinished C++ project and (successful) Prolog project … development time was reduced by factor between 20 to 50."

"…Java solver needed more time to create domain variables only than Prolog needed to solve the complete problem."

"We have about 200K lines of Prolog."

"At this moment, I don't see any other platform than Prolog that could be used efficiently for large scale real time constraint programming."

"…during last 10 years of interviewing poeple I was able to dectect only one guy who knew the word "Prolog". He was from Egypt".

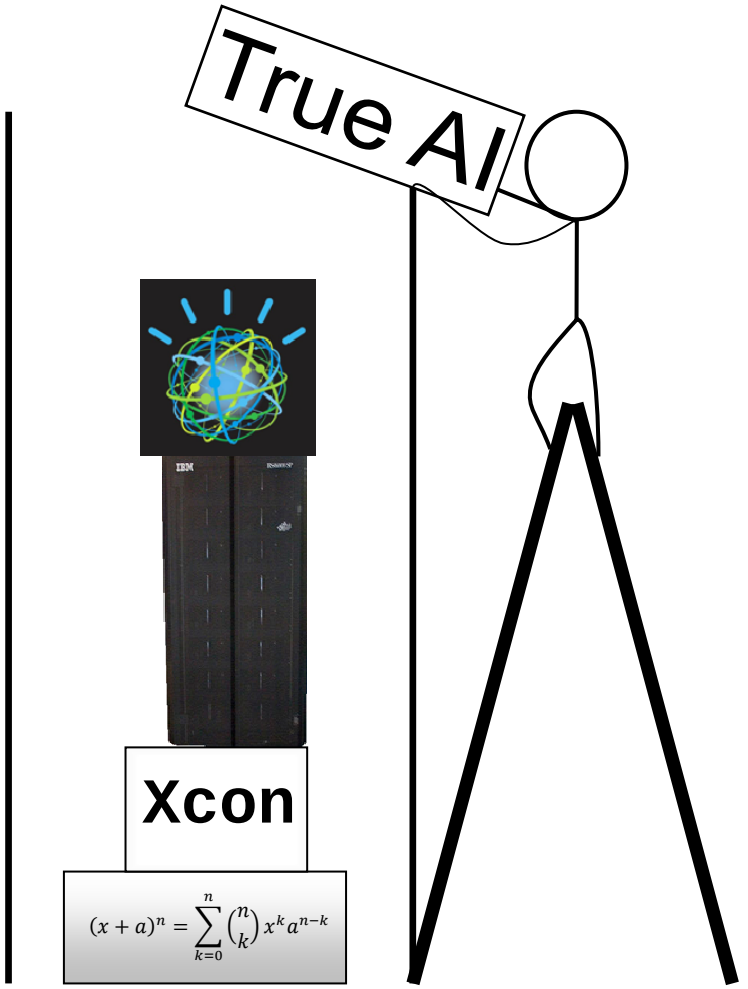
compgroups.net/comp.lang.prolog/real-world-applications/115396

groups.google.com/forum/#!msg/comp.lang.prolog/Wx_ipFUVWfM/5Hz47Oqu_2MJ

Zajímavost:

Fenomén pronásledující vývoj umělé inteligence:

"**Zvedání laťky**": Pokaždé, když je nějaký úkol vyřešen použitím umělé inteligence, brzy někdo prohlásí, že na tom přece nebylo nic složitého a o žádnou umělou inteligenci se ve skutečnosti nejedná.



Příklady použití Prologu

Raketoplán Buran (1988)

Prolog byl použit k řízení systémů sovětského raketoplánu Buran a jeho nosné rakety Energia.

Informace pochází z odtajněné zprávy CIA, kde je toto uváděno jako příklad úspěchu sovětského inženýrství, jinak v oblasti výpočetní techniky zaostávajícího.

Projekt Buran byl zrušen po prvním (úspěšném) testovacím letu, vzhledem k politické situaci doprovázející rozpad SSSR.

Clarissa (2005)

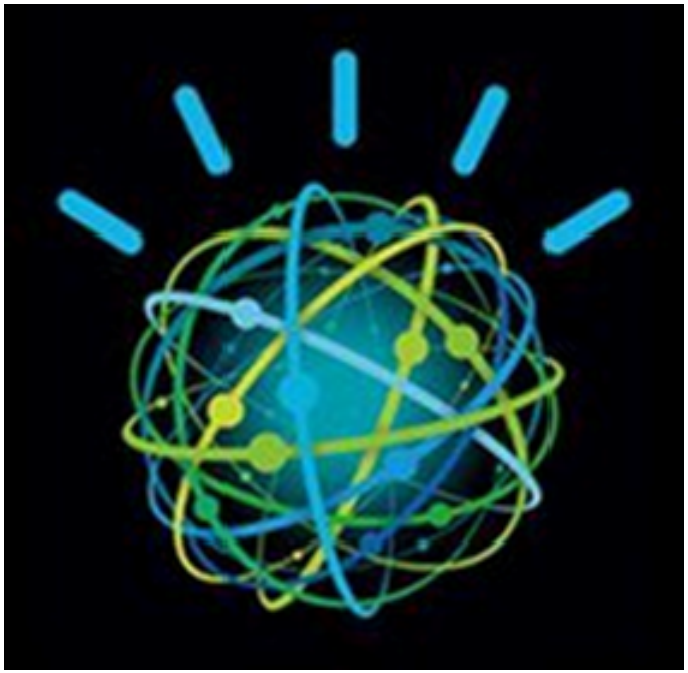
Clarissa je interaktivní hlasem ovládaný systém pro ISS. Poskytuje astronautům rady při provádění 12000 různých řídicích povelů a ušetří je tak náročného sledování manuálů.

Clarissa není nijak náročný program, běží na obyčejném PC.

Experimentální použití na Zemi (headset má zamezit hluku na stanici).

IBM Watson (2011)

Watson je superpočítač vyvinutý firmou IBM. Funguje tak, že předzpracuje velké množství textu a ve vytvořené databázi pak podle asociací hledá odpovědi na otázky. Oproti tradičnímu vyhledávání by měl umět sám rozpoznat relevatní informace a poskytnout nejlepší odpověď.



Watsonův avatar

Watson sestává z 2880 procesorových jader a používá 15 TB RAM. Maximální výkon je 80 teraflops (počet operací v provoucí řádové čárce za sekundu). Běží pod Linuxem a je naprogramován především v Javě, C++ a v Prologu.

Watson v zákulisí (10 racků po 10 serverech IBM Power 750)

Krátce o Prologu ve Watsonu: http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/

Zajímavost: Ve finále prvního klání bylo téma "Americká města" a nápověda zněla: “Its largest airport is named for a World War II hero; its second largest for a World War II battle.” Správná odpověď byla “What is Chicago?”, ale Watson velmi nejistě odpověděl “What is Toronto?”

Hledáním v mapě lze zjistit, že Toronto je americké město na severu Západní Virginie, nicméně je málo známé a nemá letiště. Odpověď tedy nebyla zcela nemožná, ale přesto podivná.

Watson ovšem do finále předem vsadil velice malou částku, takže chyba jeho vítězství nijak neohrozila.


Wisconsin
Michigan
Maine
Vermont
New Hampshire
New York
Massachusetts
Connecticut
Rhode Island
A
Pennsylvania
Ohio
Illinois
Indiana
Maryland
New Jersey
West Virginia
Delaware
District of Columbia
Kentucky
Virginia
Tennessee
North Carolina

WADNES

je systém pro řízení distribuce vody. Vedoucí projektu zdůrazňoval jednoduchost a přehlednost programů v Prologu ve srovnání s C++, jen zanedbatelnou ztrátu výkonu a paměťovou nenáročnost.

COPLEX

je systém pro optimalizaci dopravy v reálném čase.

CASEy

je expertní systém pro společnost Boeing usnadňující práci pozemního personálu.

Syntax Prologu a postup hledání řešení

Prolog je založen na formulích *predikátové logiky 1. řádu*. Algoritmus pro určení pravdivostní hodnoty formule známe – je to *rezoluční metoda*. Ta však má při hledání řešení obecných formulí superexponenciální složitost.

Výkonné dokazovací algoritmy používají heuristiky založené na rozsáhlých znalostech z dané oblasti. Takový postup však není použitelný k vytvoření programovacího jazyka. Autoři Prologu proto přesně vymezili tvar formulí, vstupujících do rezoluce.

Jako nejlepšího kandidáta zvolili *Hornovy klauzule*, na nichž založili syntax Prologu. Než se dostaneme k samotnému Prologu, připomeňme krátce použité termíny z logiky:

Definice 1: **Klauzule** je *disjunkce literálů*.

Příklad 1: Formule $a \vee \neg b \vee c \vee d$ je klauzule.

Formule $(a \wedge \neg b) \Leftrightarrow (c \vee d)$ není klauzule.

Formule $\neg(a \vee \neg b \vee c \vee d)$ není klauzule.

Definice 2: **Hornova klauzule** je klauzule, která obsahuje *nejvýše jeden pozitivní literál*, ostatní jsou negované.

Příklad 2: Mějme Hornovu klauzuli $\neg a \vee \neg b \vee \neg c \vee d$, což je ekvivalentní zápis implikace: $(a \wedge b \wedge c) \Rightarrow d$.

(Pro připomenutí: $f \Rightarrow g \quad \Leftrightarrow \quad \neg f \vee g$ *)*

Jinak řečeno, platnost d plyne z platnosti všech literálů a, b, c.

Pro zápis v Prologu musíme však zápis obrátit: $d \Leftarrow (a \wedge b \wedge c)$.

Platí tedy, že Hornovy klauzule lze obecně zapsat jako **implikace**.

V predikátové logice můžeme kromě prostých faktů používat i *proměnné*, *predikáty* a *kvantifikátory* proměnných – ∀ a ∃.

Příklad 3: Máme tyto formule predikátové logiky a chceme prozkoumat jejich pravdivost:

$$(\forall X)\big(počet\_hran(X,3) \Rightarrow trojúhelník(X)\big)$$

$$(\forall X)\big(počet\_hran(X,4) \Rightarrow čtverec(X)\big)$$

Predikáty $počet\_hran$, $trojúhelník$ a $čtverec$ by měly být sebevysvětlující.

Formule $(\forall X)\big(počet\_hran(X,3) \Rightarrow trojúhelník(X)\big)$ je zřejmě pravdivá – každý geometrický útvar, který má 3 hrany, je trojúhelník.

S formulí $(\forall X)\big(počet\_hran(X,4) \Rightarrow čtverec(X)\big)$ je to jinak – určitě ne každý čtverhraný útvar je čtverec. Formule v této podobě je tedy nepravdivá.

Pravdivá by byla, pokud by namísto všeobecného kvantifikátoru obsahovala existenční: $(\exists X)\big(počet\_hran(X,4) \Rightarrow čtverec(X)\big)$

Hornova klauzule v predikátové logice tedy může vypadat třeba takto:

$$p(X,Y) \Leftarrow q(X) \wedge r(X,Y,a) \wedge s(Y,b) \ ,$$

kde X, Y jsou proměnné, a, b jsou běžné atomické formule.

Kvantifikátory neuvádíme, protože formuli můžeme prohlásit za klauzuli až tehdy, pokud kvantifikátory převedeme na všeobecné a přesuneme na začátek – klauzule tedy platí vždy a pro libovolné hodnoty proměnných.

Dohodněme se, že disjunkci literálů lze zapsat jako množinu literálů, pro jednoduchost bez množinové závorky:

$$p(X,Y) \Leftarrow q(X),\ r(X,Y,a),\ s(Y,b)$$

Obecně:

$$b \Leftarrow a_1,\ a_2,\ \ldots,\ a_{n-1},\ a_n$$

Při *procedurální interpretaci* Hornovy klauzule chápeme b jako proceduru, jejíž tělo sestává z postupného volání procedur $a_1, \ldots, a_n$. Tato jednoduchá úvaha přímo vede na postup zpracování v počítači a v Prologu.

V Prologu Hornovu klauzuli zapíšeme následovně:

```
b :- a1, …, an.
```

Takovou klauzuli v Prologu nazýváme **pravidlo**. Jako symbol implikace je použita dvojtečka bezprostředně následovaná znaménkem mínus. Dále nesmíme zapomenout ukončit klauzuli *tečkou* (jí odpovídá středník např. v C nebo Javě).

Pokud na pravé straně implikace nestojí žádné formule (klauzule je automaticky splněna), píšeme `b :- true.` nebo lépe jen `b.` (stále nutno psát tečku!). Taková klauzule se nazývá **fakt**.

Pokud na levé straně implikace není identifikátor formule, jejíž pravdivost zkoumáme, píšeme `?- a1, …, an.`

Poznámka: Některé verze Prologu umožňují také zápis ve tvaru

`:- a1, …, an.`

Taková klauzule se nazývá **cílová** (goal). Cíle Prolog začne *automaticky vyhodnocovat* po spuštění programu – existuje tedy podobnost s funkcí (metodou) `main` v C nebo Javě.

Teď se podíváme na jednoduchý program v Prologu.

Příklad 4: Následující řádky definují několik vztahů mezi lidmi a alkoholickými nápoji:

```
miluje('Petr',pivo).
miluje('Petr',vino).
miluje('Aneta',vino).
miluje('Aneta','Petr').
```

`miluje` je zde predikát, `'Petr'`, `'Aneta'`, `pivo` a `vino` jsou atomické formule (atomy) a všechny čtyři klauzule reprezentují fakta (platí bezpodmínečně).

Možná máte pocit, že jsme nic nenaprogramovali, nicméně, tyto čtyři řádky jsou skutečně program v Prologu, i když samozřejmě opravdu jednoduchý.

Zadaná fakta jsou při spuštění programu uložena do databáze Prologu. My nyní můžeme zadávat cíle a Prolog zjistí, zda jsou splněny.

Na dotaz `?- miluje('Petr',pivo).` Prolog odpoví `yes`. Dospěl k tomu tak, že ve své databázi nalezl první klauzuli s predikátem `miluje` a porovnal jeho argumenty se zadanými. První je fakt `miluje('Petr',pivo).` – cíl je tedy splněn.

Při dotazu `?- miluje('Aneta','Petr').` Prolog postupně prohledá celou databázi a u posledního faktu nalezne shodu – odpoví tedy znovu `yes`.

Na dotaz `?- miluje('Aneta',pivo).` Prolog odpoví `no`, protože nic v databázi jeho platnosti nenasvědčuje.

Prolog předpokládá *úplný popis světa* – tedy, že všechny platné vztahy budou popsány v jeho databázi. Nezná žádnou odpověď typu "nevím" a pokud nedokáže daný vztah ze svých znalostí potvrdit, předpokládá, že neexistuje.

Na dotaz `?- miluje('Petr','Aneta').` Prolog odpoví `no`, přestože databáze obsahuje fakt `miluje('Aneta','Petr').`, protože Prolog důsledně respektuje pořadí argumentů, resp. vyhodnocované relace jsou àpriori nesymetrické.

Ke spokojenosti Anety bychom tedy museli do databáze přidat fakt `miluje('Petr','Aneta')`, čímž zajistíme symetrii tohoto vztahu.

Následuje snímek obrazovky, na níž je znázorněno, jak takové "ruční" dotazování přesně vypadá (byl použit B-Prolog).

```
Type 'help' for usage.
Compiling::prvni_priklad.pl
compiled in 45 milliseconds
loading::prvni_priklad.out
| ?- miluje('Petr',pivo).

yes
| ?- miluje('Aneta','Petr').

yes
| ?- miluje('Aneta',pivo).
no
| ?- miluje('Petr','Aneta').
no
| ?- _
```

I s tak jednoduchým programem však můžeme provádět o něco složitější dotazy.

Pokud zadáme `?- miluje('Aneta',X)`, Prolog pro nás zjistí, koho nebo co všechno miluje Aneta, neboli všechny hodnoty, které lze dosadit za `X`, aby klauzule byla pravdivá.

Konkrétně Prolog vypíše `X = vino ?` jakožto první nalezenou možnost. My však můžeme zadáním *středníku* navržené řešení odmítnout a Prolog nabídne další: `X = 'Petr'`. Bezprostředně potom vypíše `yes`, protože další řešení neexistují.

Zajímavost: Podobnost Prologu a SQL:

SQL jsme v souvislosti s Prologem zmínili především proto, že oba jazyky jsou formou zápisu predikátové logiky 1. řádu.

Pokud znáte SQL, uvažte toto:

- Fakta a data (řádky tabulek) jsou totéž.
- Relace v relační teorii je totéž, co tabulka.
- Pohled na tabulku (View) je totéž, co pravidlo v Prologu.

Společné znaky Prologu a SQL:

- založeny na logice,
- mohou ukládat a používat relace,
- mohou vyjadřovat složité logické podmínky,
- mohou ukládat fakta a vyvozovat z nich závěry,
- jsou to programovací jazyky a
- jsou turingovsky úplné.

Nicméně, každý jazyk je určen pro zcela jiné účely, považovat je za ekvivalenty by tedy bylo nanejvýš nevhodné.

Z čeho se skládá program

Celý program se skládá z **termů**, které dělíme na:

- **konstanty**
  - **číselné** – zapisujeme je běžným způsobem – například `42` nebo `3.14`.
  - **atomy** – musejí začínat malým písmenem a mohou obsahovat číslice a podtržítka, např. `pivo`, `a1` nebo `chleba_s_maslem`.
    Pokud jsou ohraničeny *apostrofy*, mohou obsahovat i mezery a další znaky, např. `'Petr'`.

- **proměnné** – začínají velkým písmenem nebo podtržítkem, např. `X`, `Akumulator` nebo `_N`. Samotné podtržítko (`_`) je *anonymní* proměnná. Proměnné začínající podtržítkem nás nezajímají a *nevypisují se* na výstup.
- **složené termy**
  - **struktury** – *predikáty* (např. `student('Jan')` a `pise(student('Jan'),program(X,'Prolog'))`, – *operátory a operandy* – jde v podstatě také o predikáty, avšak musejí být zapsány infixově.
  - **seznamy** – sem patří také řetězce, probereme je později.

Další pojmy, se kterými se můžeme v Prologu setkat:

Predikát tvoří **funktor** a **argumenty** – `funktor(arg1,…,argN)`.
V literatuře se lze take setkat se zápisem `funktor/N`, kde N je celé nezáporné číslo a značí počet argumentů (*aritu*) predikátu.

Term na *levé straně* pravidla se nazývá **hlavička** pravidla, termy na *pravé straně* tvoří **tělo** pravidla.

B-Prolog

B-Prolog je zavedená implementace standardního Prologu (existuje od roku 1994) doplněná o několik rozšíření. Jako jedna z jejích hlavních výhod je uváděna velká výkonnost.

Jedná se o komerční implementaci poskytovanou zdarma pro výukové a nekomerční účely, nově také pro komerční použití jednotlivci.

Logo z domovské stránky www.probp.com

Ceník z roku 2014

Academic license	Free
Individual license	Free
Non-commercial site license	Free
Commercial site license	$2,980 USD
C source code license + technical support	$4,980 USD

B-Prolog lze stáhnout z jeho domovské stránky www.probp.com. Instalace probíhá jednoduše rozbalením archivu do zvolené složky a připojením do systémové proměnné PATH (Windows), nebo nastavením aliasu (Linux, OS X).

Ze současné neplacené verze 8.1 však zřejmě byly vypuštěny některé doplňky, zejména ".jar" soubory pro spolupráci s Javou (informace z dubna 2014).

Pokud si je budete chtít vyzkoušet, doporučuji upravit stahovací odkaz, aby směřoval na starší verzi Prologu (pro Windows http://www.probp.com/download/bp78_win.zip), nebo stáhnout tuto verzi B-Prologu ze stránek předmětu.

Ke spouštění B-Prologu v této starší verzi slouží skript `bp.bat` (Windows) nebo jen `bp` (Unix). Ve skriptu je proměnná `BPDIR`, nastavena na preferované umístění (`c:\BProlog` nebo `$HOME/BProlog`), které musíte upravit, pokud zvolíte jiné.

V archivu je i kvalitní podrobná dokumentace (přestože ne vždy zcela aktuální), na kterou se doporučuji obrátit, pokud se budete chtít dovědět něco nad rámec těchto přednášek.

Dále archiv obsahuje ukázkové příklady. Další příklady a dokumentace v online podobě je na stránkách B-Prologu.

Zdrojové kódy programů v B-Prologu mají příponu ".pl", standardní přípona programu zkompilovaného do bajtkódu (bytecode) je ".out" .

Po spuštění B-Prologu v konzoli se objeví příkazová řádka uvozená `| ?-`, která značí, že je možno zadávat cíle Prologu. Ukončit B-Prolog lze zadáním predikátu (příkazu) `halt` nebo ukončením vstupu (CTRL+D) nebo DELETE při prázdné řádce. V současné verzi B-Prologu 8.1 však toto, zdá se, příliš nefunguje).

```
c:\BProlog>bp
B-Prolog Version 8.1, All rights reserved, (C) Afany Software 1994-2014.
| ?- halt

c:\BProlog>_
```

Použitím parametru `-g` lze zadat cíl, který se má splnit ihned po spuštění B-Prologu (`-g "cíl"`). Ke kompilaci a spuštění programu zároveň slouží predikát `cl` – spustit program ihned po spuštění B-Prologu tedy můžete takto:

```
bp -g "cl(program)"
```

(Program je název zdrojového souboru bez přípony ".pl". Cíl raději zapisujte do uvozovek, aby byl chápán jako jeden parametr.)

```
c:\BProlog>bp -g cl(priklad)
B-Prolog Version 8.1, All rights reserved, (C) Afany Software 1994-2014.
Compiling::priklad.pl
compiled in 2 milliseconds
loading...
| ?-
```

Pamatujte, že zadaný název programu je atomický term Prologu. Pokud tedy zadáváte název včetně cesty, musíte použít apostrofy (kvůli lomítkům):

```
bp -g "cl('cesta/program')"
```

Pokud váš program sestává z více souborů, lze predikátu `cl` zadat seznam parametrů nebo predikát použít vícekrát:

```
bp -g "cl([soubor1,soubor2])"
```

nebo

```
bp -g "cl(soubor1),cl(soubor2)"
```

V obou případech B-Prolog obsah souborů automaticky zkombinuje. Pro větší počet souborů je evidentně možné vytvořit jeden hlavní soubor, ze kterého se vše spustí.

Pokud chceme program pouze zkompilovat, použijeme příkaz `compile(program)`. Zkompilovaný kód bude uložen v souboru se stejným názvem, ale s příponou ".out".

Pokud chceme specifikovat vlastní název a příponu, použijeme `compile(program,vysledek)`, kde "vysledek" je název výsledného souboru (včetně přípony).

Pro spuštění již zkompilovaného programu pak použijeme příkaz `load(vysledek)`.

V obou případech lze také zadat seznam souborů.

Pokud v programu nejsou definovány žádné cíle (klauzule bez hlavičky začínající `?-` nebo `:-`, vyhodnocované ihned), Prolog pochopitelně sám od sebe neudělá nic, dokud nebude zadán dotaz.

V případě cílů zadaných s parametrem `-g` nebo přímo v programu však B-Prolog po jejich provedení nevypisuje automaticky žádné výsledky – jakýkoliv výstup musí naprogramovat sám autor programu.

Dokud provádíme pouze jednoduché pokusy, je výhodnější zadat cíl ručně do příkazové řádky Prologu a automaticky získat informaci o úspěchu/neúspěchu, případně další výsledky.

Začínáme se složitějším programem

Program uvedený na následujícím snímku není o mnoho složitější než úvodní ukázka.

Postupně si ho však upravíme do podoby, kdy bude sice stále jednoduchý, ale dokáže najít cestu z místa A do místa B podle údajů, které mu o světě vložíme do databáze.

Využijeme při tom něco z poznatků o různých možnostech Prologu, které si mezitím ukážeme.

Program definuje čtyři fakta (sousednost několika států) a další dvě pravidla, která popisují možnosti cestování mezi státy "suchou nohou":

```
sousedi('Kanada','USA').
sousedi('USA','Mexiko').
sousedi('Rusko','Cina').
sousedi('Cina','Indie').

cesta(X,Y) :- sousedi(X,Y).
cesta(X,Y) :- sousedi(X,Z), cesta(Z,Y).
```

V tomto programu je již využita *rekurze* – tu snadno vidíme z toho, že predikát z hlavičky pravidla se vyskytuje v těle pravidla.

`cesta(X,Y) :- sousedi(X,Y).` – toto pravidlo definuje, že ze státu X lze cestovat do státu Y, pokud s ním sousedí. Slouží jako *zastavovací podmínka* rekurze.

`cesta(X,Y) :- sousedi(X,Z), cesta(Z,Y).` – pravidlo definuje, že ze státu X lze cestovat do Y, pokud sousedí s jiným státem Z, ze kterého lze do Y cestovat.

Řekněme, že zadáme cíl `?- cesta('Kanada','Mexiko').` a prozkoumáme, co se stane.

První odpovídající klauzule `cesta(X,Y) :- sousedi(X,Y).` má argumenty (proměnné) `X` a `Y`. Prolog provede **unifikaci** proměnných a konstant – *navázání* proměnných na termy `'Kanada'` a `'Mexiko'`. Důsledek se podobá přiřazení do proměnné, ale existují rozdíly (viz později).

Je důležité vědět, že proměnné platí vždy jen *v rozsahu právě zpracovávané formule* – jako lokální proměnné ve funkci. V Prologu neexistuje *nic jako globální proměnné* – ty jsou zcela proti principům deklarativního programování.

Prolog pokračuje vyhodnocením těla pravidla `cesta(X,Y) :- sousedi(X,Y).`, přičemž platí `X = 'Kanada'` `Y = 'Mexiko'` Musí proto najít predikát odpovídající `sousedi('Kanada', 'Mexiko')`. Protože ani jeden z faktů v databázi neodpovídá, aplikuje *mechanismus návratu* (backtracking) – zruší současný postup a zkusí nejbližší další vyhovující klauzuli:

```
cesta(X,Y) :- sousedi(X,Z), cesta(Z,Y).
```

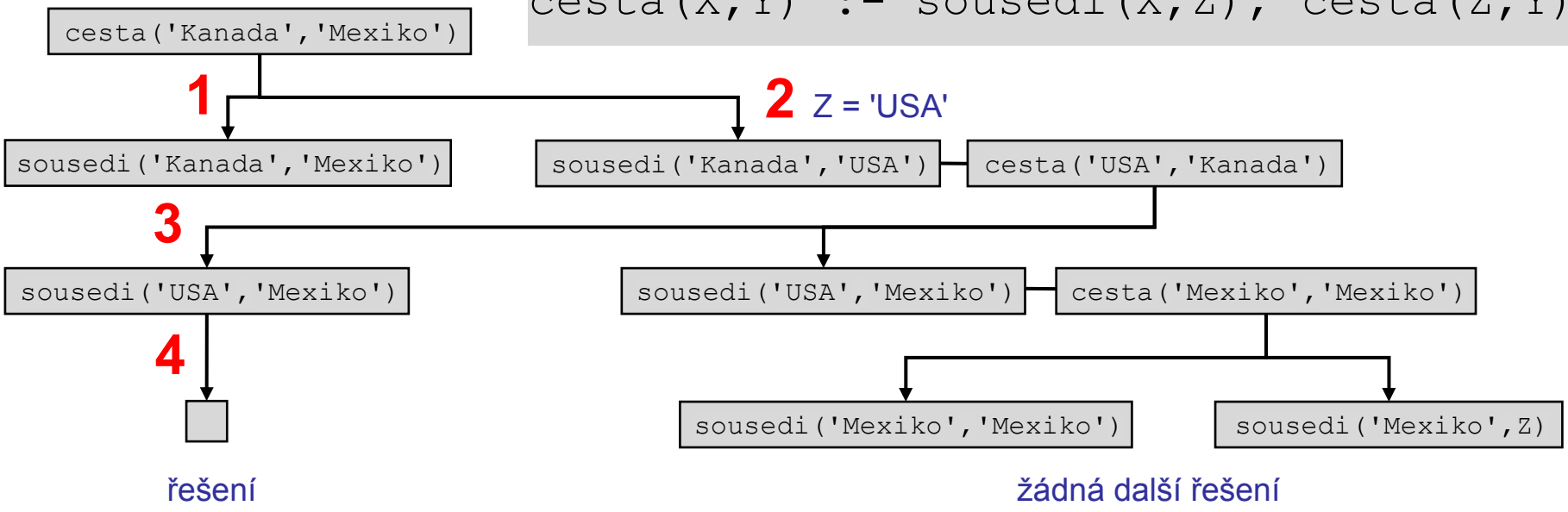
Předchozí navázání proměnných je zrušeno, protože Prolog opustil pravidlo, ale pro toto nové pravidlo ho stejným způsobem provede znovu.

Tělo pravidla `cesta(X,Y) :- sousedi(X,Z), cesta(Z,Y).` Prolog vyhodnocuje klasicky zleva doprava, tudíž hledá stát sousedící se státem `X = 'Kanada'`, který může být libovolný – proměnná `Z` ještě není navázána. V databázi má fakt `sousedi('Kanada','USA').`, tudíž naváže `Z = 'USA'` a první predikát je splněn.

Druhý predikát má díky navázání proměnných podobu `cesta('USA', 'Mexiko')` – Prolog pro něj znovu zkusí pravidlo `cesta(X,Y) :- sousedi(X,Y).` a tentokrát najde vyhovující fakt v databázi. Celé pravidlo je tedy splněno a protože to byl náš původní dotaz, Prolog odpoví `yes`.

Tento strom naznačuje možné cesty při hledání řešení. Červené číslice značí, jak Prolog prochází jednotlivé větve.

```
cesta(X,Y) :- sousedi(X,Y).
cesta(X,Y) :- sousedi(X,Z), cesta(Z,Y).
```



Pravděpodobně odhadnete, že pokud zadáme dotaz `?- cesta('Rusko','USA').`, Prolog provede podobný průzkum "na druhé straně oceánu", ale v Indii se dostane do slepé uličky a musí odpovědět `no`.

Samozřejmě se opět můžeme ptát i jinak, například na dotaz `?- cesta('USA',X).` nám Prolog odpoví `X = 'Mexiko' ?` – protože se neptáme jen na jednoduchá fakta, nemůže hned určit, zda existují další možnosti, ale po zadání středníku odpoví `no` – žádná další možnost není.

Tím jsme narazili na zjevný nedostatek programu – sousednost je v databázi definována jen "od severu k jihu", protože uvedené vztahy nejsou považovány za symetrické – takové Mexiko nebo Indie teď podle Prologu nesousedí s ničím:

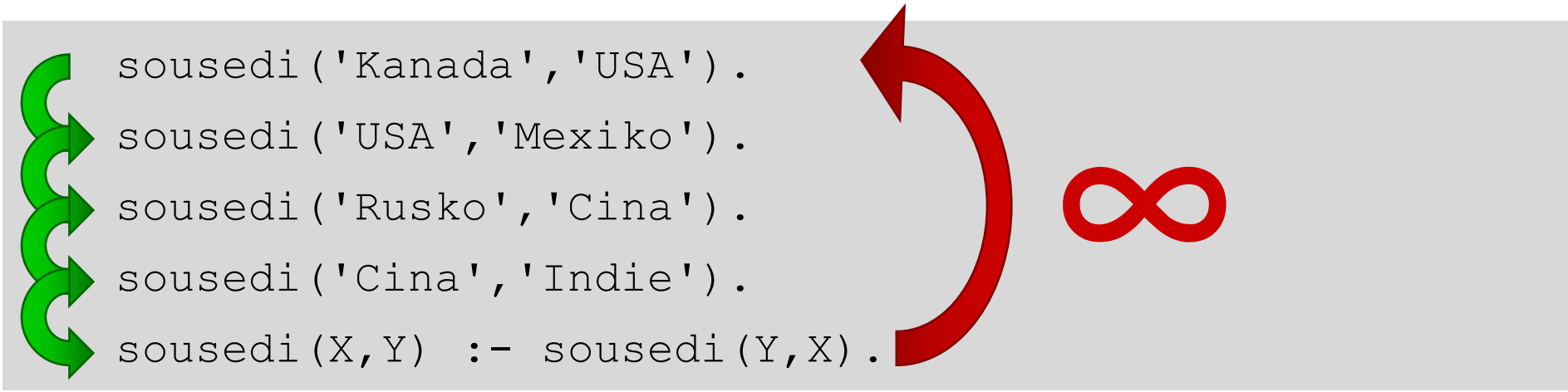
```
sousedi('Kanada','USA').
sousedi('USA','Mexiko').
sousedi('Rusko','Cina').
sousedi('Cina','Indie').
```

Nejjednodušší by bylo přidat mezi fakta všechny obrácené vztahy – při větším množství by to však byla spousta práce navíc a program by nám zbytečně narostl.

Patrně vás napadá i lepší řešení – můžeme přidat pravidlo `sousedi(X,Y) :- sousedi(Y,X).` – tím vlastně definujeme symetrii relace.

Ta má však jeden *zásadní problém*. Jakmile se zeptáme na sousednost států, které spolu nesousedí (což při hledání cesty děláme běžně), Prolog projde všechna fakta, skončí u tohoto pravidla a zkusí ho splnit. Projde tedy fakta znovu s přehozenými parametry, ale ani teď sousednost nenajde – protože neexistuje. Opět skončí u tohoto pravidla a nikde není řečeno, že se ho nemá pokusit splnit. Je zřejmé, že výpočet uvázne v *rekurentní smyčce* a ta dříve či později skončí přetečením zásobníku.

Naznačená rekurentní smyčka:



Problém rekurze lze snadno vyřešit zavedením nového predikátu:

```
prechod(X,Y) :- sousedi(X,Y).
prechod(X,Y) :- sousedi(Y,X).
```

Pokud nyní v dalších dotazech budeme důsledně používat predikát `prechod` namísto `sousedi`, Prolog vždy pouze prověří sousednost v jednom nebo případně v druhém směru. Pokud ani jedno pravidlo nevyhoví, odpověď je zkrátka `no` – není zde žádná rekurze, která by způsobila zacyklení.

Predikát cesta tedy v reakci na změny upravíme následovně:

```
cesta(X,X).
cesta(X,Y) :- prechod(X,Z), cesta(Z,Y).
```

Všimněte si, že abychom nedělali jen prosté přejmenování, zjednodušili jsme také první klauzuli – rekurzi ukončíme tím, že už jsme v cíli.

Pouze tehdy může být první argument (současná poloha) a druhý argument (cílová poloha) stejný a tudíž použita stejná proměnná.

Celý program teď vypadá takto:

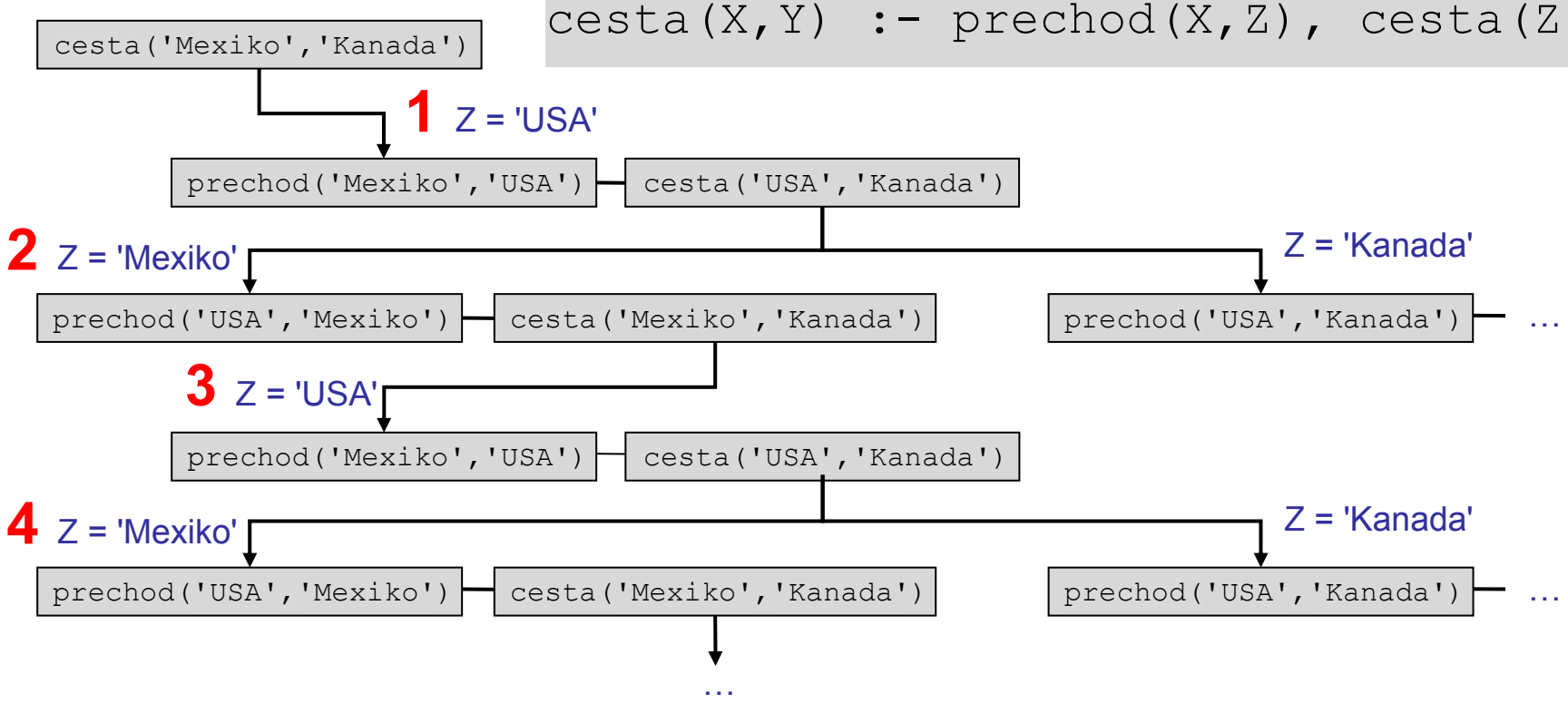
```
sousedi('Kanada','USA').
sousedi('USA','Mexiko').
sousedi('Rusko','Cina').
sousedi('Cina','Indie').

prechod(X,Y) :- sousedi(X,Y).
prechod(X,Y) :- sousedi(Y,X).

cesta(X,X).
cesta(X,Y) :- prechod(X,Z), cesta(Z,Y).
```

Zacyklení se nám však stále nepodařilo odstranit, pouze se přesunulo do predikátu `cesta`. Při hledání například cesty z Mexika do Kanady nebo třeba do Ruska totiž Prolog začne donekonečna přecházet mezi Mexikem a USA.

Tentokrát je problém v tom, že prohledáváme neorientovaný graf, přičemž si nepamatujeme, kde už jsme byli.

Jinak řečeno – podle pravidel, pokud Prolog nedosáhl cílového státu, má zkusit libovolný sousední a vzhledem k obousměrné sousednosti nejen nikdy nevyčerpá možnosti přechodů, ale také se může zacyklit třeba jen mezi dvěma státy.

```
cesta(X,Y) :- prechod(X,Z), cesta(Z,Y).
```



Prolog si Kanadu nikdy nevybere, protože přechod do Mexika pokaždé najde v databázi první, ať je v rekurzi libovolně hluboko.

Na takové případy, kdy Prologu umožníme "vrátit se" bez toho, že by aplikoval backtracking, si musíme obecně dávat pozor – může tak snadno dojít k tomu, že Prolog "přešlapuje na místě", zatímco ve skutečnosti se ve svém odvozovacím procesu zanořuje hlouběji a hlouběji.

Poznámka: *Mimochodem, pokud bychom ponechali původní ukončovací podmínku (* `cesta(X,Y) :- prechod(X,Y)` *), cesta z Mexika do Kanady by nalezena byla (po přechodu do USA by se již nevyhodnocovalo rekurzivní pravidlo), ale při pokusu dostat se do Ruska nebo pokud by cesta byla delší, pak by k zacyklení opět došlo.*

Prolog bohužel neumí žádná kouzla, kterými by se takovému zacyklení ubránil, musíme se zacyklení ubránit sami.

Zkusíme si tedy pamatovat, odkud jsme přišli. To můžeme udělat přidáním dvou argumentů predikátu cesta:

```
cesta(_,_,X,X). %V cíli už nás nezajímá, kde jsme byli.
cesta(B,A,X,Y) :- prechod(X,Z), cesta(A,X,Z,Y).
```

- Argument `A` popisuje stát, odkud jsme právě přišli.
- Argument `B` popisuje stát, kde jsme byli předtím.

Pamatujeme si tedy poslední tři navštívené státy (včetně toho, kde jsme), takže poznáme, když se začneme vracet.

Teď ještě musíme použít tuto znalost k ukončení rekurze. Na to nám ovšem nestačí, co jsme si zatím ukázali, potřebujeme probrat tzv. vestavěné (build-ins) predikáty – viz dále.

Cut & fail

Jedna z možností, jak utnout nekonečnou rekurzi, je kombinace predikátu "cut" (značí se vykřičníkem – `!`) a predikátu `fail`.

Predikát "cut" (`!/0`, v češtině ho najdeme pod názvem řez) Prologu *zabrání změnit jakékoliv rozhodnutí*, které učinil *od výběru pravidla*, které právě vyhodnocuje, *do predikátu řezu*. V podstatě odřízne příslušné větve rozhodovacího stromu a pokud pravidlo nevyhoví, nutí přejít na vyšší úroveň stromu.

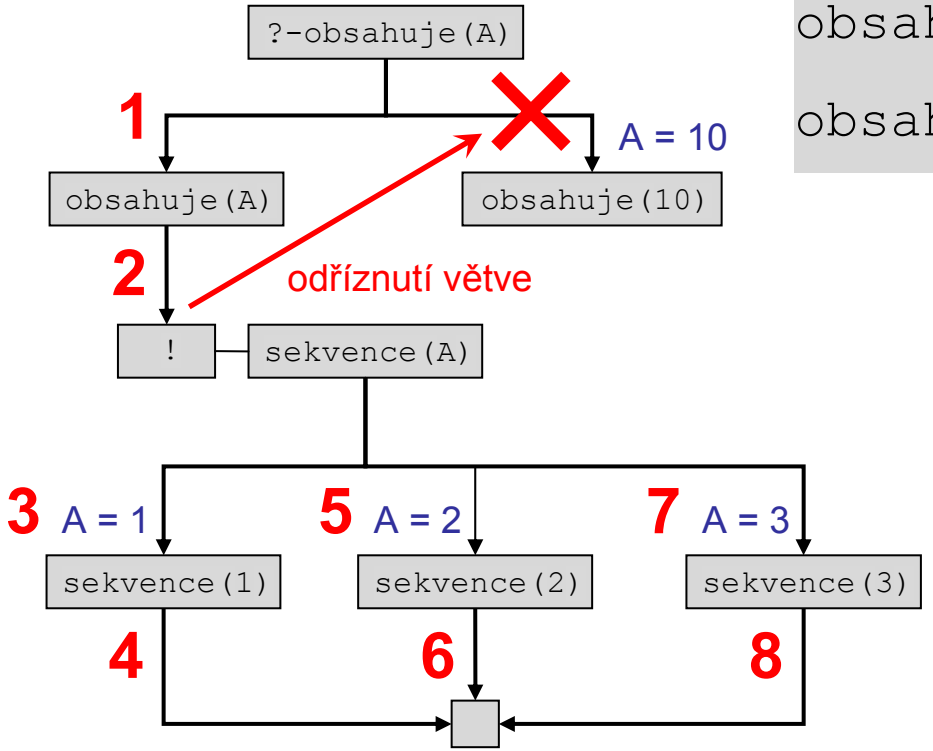
Poznámka: *Číslo v závorce udává aritu predikátu – v tomto případě je predikát bez argumentů.*

Příklad 5: Podívejme se, co se stane, pokud v následujícím programu budou nebo nebudou označeny predikáty "cut":

```
sekvence(1). sekvence(2). sekvence(3).
obsahuje(A) :- !/*1*/, sekvence(A), !/*2*/.
obsahuje(10).
```

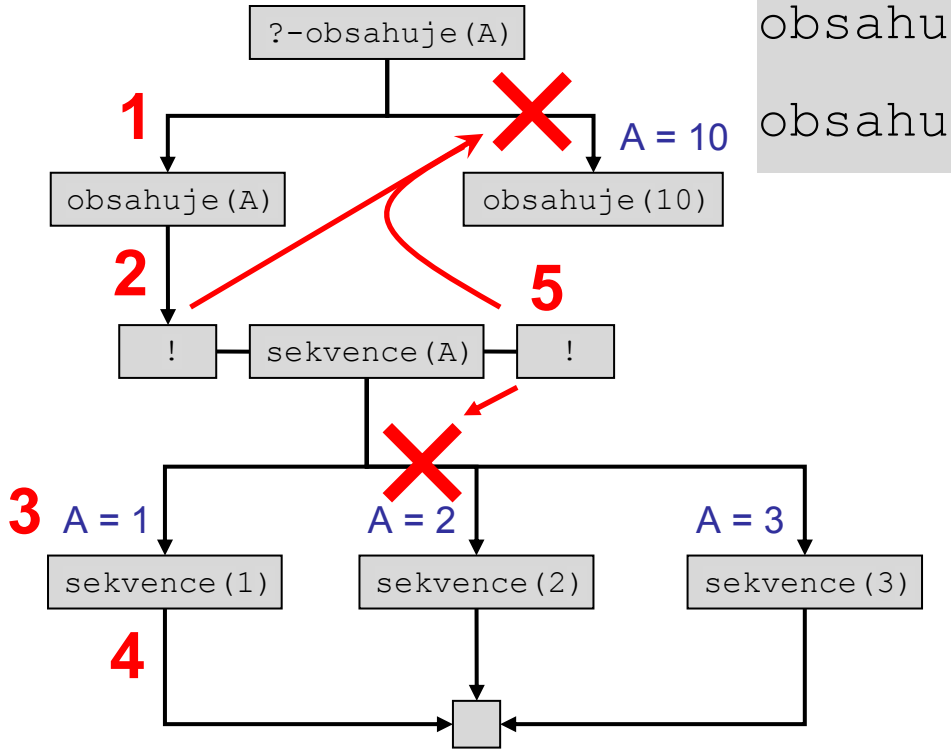
Pokud oba řezy vyškrtneme a zadáme dotaz `?-obsahuje(A)`, patrně zjistíme, že Prolog nám nabídne po řadě všechny možnosti – `A = 1`, `A = 2`, `A = 3`, `A = 10`.

Pokud ponecháme pouze řez č. 1, možné hodnoty budou `A = 1`, `A = 2`, `A = 3`. Vlivem řezu Prolog poslední klauzuli nevyhodnotí.




```
obsahuje(A) :- !, sekvence(A).
obsahuje(10).
```

Pokud ponecháme pouze řez č. 2, jediná možná hodnota proměnné bude `A = 1`. Stejně tak v případě použití obou řezů:




```
obsahuje(A) :- !, sekvence(A), !.

obsahuje(10).
```

Predikát `fail/0` má jedinou funkci – není nikdy splněn a lze jím snadno a přehledně zajistit nesplnění pravidla.

Do predikátu `cesta` přidáme pravidlo k utnutí rekurze:

```
cesta(_,_,X,X).
cesta(X,_,X,_) :- !, fail. %Z nějakého jiného státu jsme
                           %se vrátili do předcházejícího.
cesta(B,A,X,Y) :- prechod(X,Z), cesta(A,X,Z,Y).
```

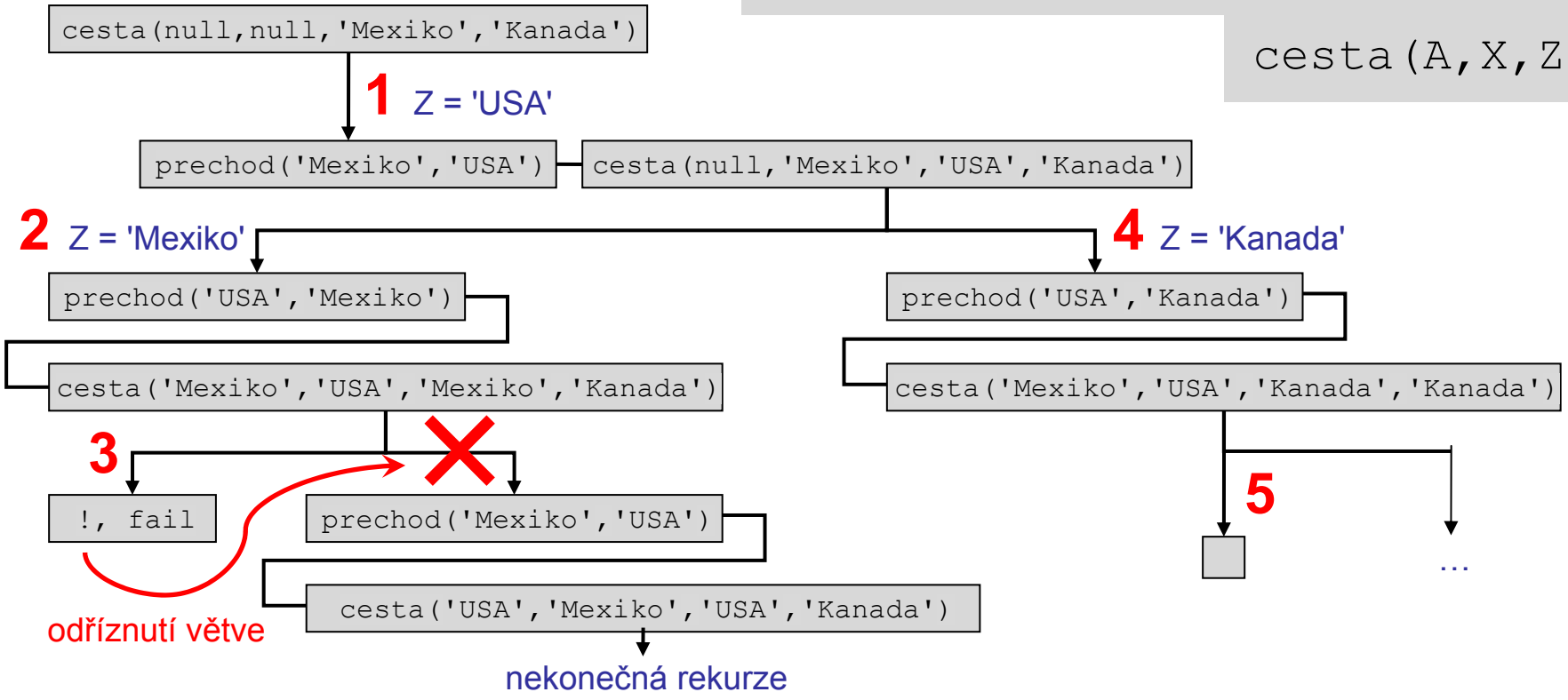
Poznámka: *Toto řešení je sice funkční, avšak z hlediska přehlednosti programu není vhodné. Později se k příkladu vrátíme a vysvětlíme, co s ním můžeme dále udělat.*

Nyní ještě zkusíme dodefinovat nový predikát `cesta/2` se dvěma argumenty pro snažší použití:

```
cesta(X,Y) :- cesta(null,null,X,Y). %Null je obyčejný
  %vymyšlený atom a můžeme ho nahradit kterýmkoliv jiným,
  %pokud se neshoduje s jedním ze států.
```

Poznámka: *Jak je z příkladu vidět, dva predikáty s různým počtem argumentů mohou být stejně pojmenovány a typicky je pak odlišujeme právě uvedenou aritou.*

```
cesta(X,_,X,_) :- !, fail.
cesta(B,A,X,Y) :- prechod(X,Z),
                  cesta(A,X,Z,Y)
```



Výpis na standardní výstupní zařízení

Predikát `write/1` je vždy splněn a vypíše term na standardní výstupní zařízení. Nemusíme se tak spoléhat pouze na odpovědi, které nám Prolog poskytne automaticky. Také je vhodný pro kontrolní výpisy.

- `write(2.718)` vypíše `2.718`
- `write(pivo)` vypíše `pivo`
- `write('Pilsner Urquell')` vypíše `Pilsner Urquell`
- `write(1,1,2,3,5)` je chyba (příliš mnoho argumentů)

Nicméně lze vypsat i složený term, takže omezení na jeden argument lze částečně obejít:

- `write(fibonacci(1,1,2,3,5))` vypíše `fibonacci(1,1,2,3,5)`

V případě proměnných je vypsán term, na který jsou navázány (brzy probereme podrobněji), nebo jejich vnitřní označení:

- `write(X)` vypíše něco na způsob `_238` za předpokladu, že proměnná X dosud nebyla navázána.

Predikát `nl/0` je vždy splněn a vypíše odřádkování.

Predikát `cesta/2` můžeme upravit, aby byly také vypsány státy, přes které budeme cestovat:

```
cesta(_,_,X,X).
cesta(X,_,X,_) :- !, fail.
cesta(B,A,X,Y) :- prechod(X,Z), cesta(A,X,Z,Y), write(X), nl.
```

Umístění predikátu způsobí, že státy budou vypsány v opačném pořadí (nejprve poslední navštívený), protože predikát `write` bude vyhodnocen až při návratu z rekurze.

Zato však budou vypsány jen státy, které jsme skutečně museli projít – při jiném umístění by se ve výpisu objevily i státy, kam se Prolog zkusil vydat a zjistil, že tudy cesta nevede.

Proměnné

Jak je patrné z již uvedených kódů, proměnné v Prologu nijak nedeklarujeme. Stačí napsat nový term začínající velkým písmenem (nebo podtržítkem) a Prolog s ním automaticky začne zacházet jako s novou proměnnou.

Ztotožňování (matching)

Ztotožnění je operace, kterou lze provést (tj. výsledkem je "yes"), pokud jsou ztotožňované *termy shodné*. Obecně lze ztotožnit libovolné termy, ale zpravidla to nemá smysl, pokud alespoň jeden z nich není proměnná.

Ztotožňování proměnných jsme již v tichosti používali − dochází k němu při vyhodnocování klauzulí. Teď si ho probereme podrobněji a zejména si ukážeme operace, které nám umožní lépe s ním pracovat.

Proměnná v Prologu se může nacházet ve dvou stavech:

- **volná proměnná** – to je výchozí stav všech proměnných.
- **vázaná proměnná** – do tohoto stavu může proměnná přejít navázáním na libovolný term.

Ztotožnění může být *úspěšné* v těchto případech:

- pokud je stejný term ztotožňován sám se sebou,
- pokud se ho účastní proměnné vázané na *stejný term*,
- pokud se ho účastní *jedna volná proměnná* a jedna vázaná proměnná nebo jiný term – pak je volná proměnná navázána na term,
- pokud se ho účastní *dvě volné proměnné*; proměnné, které byly úspěšně ztotožněny, se potom chovají jako stejná proměnná – navázáním jedné z nich je na stejný term navázána i druhá.

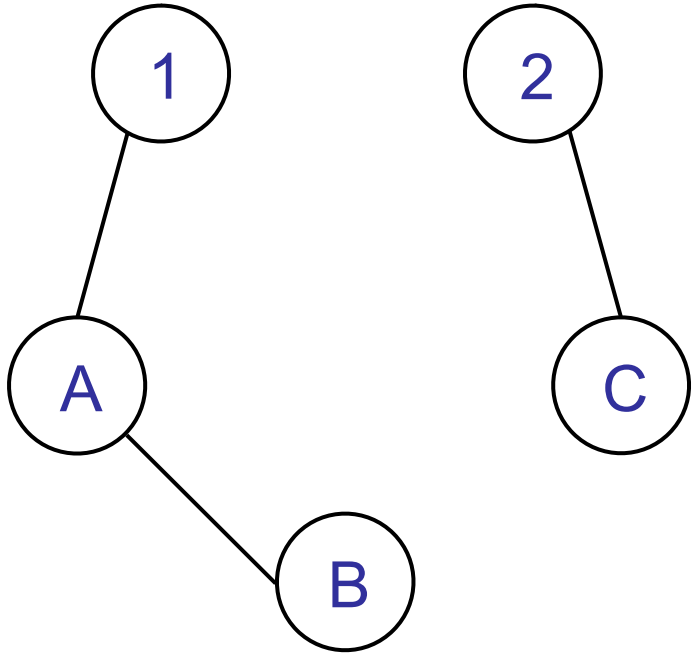
Pokud je proměnná jednou navázána, *navázání nelze zrušit* nebo změnit, *kromě backtrackingu* – při jeho aplikaci je zrušeno vše, co se stalo za bodem, kam jsme se vrátili, jako kdyby se nic nestalo.

Proměnná tedy existuje jako volná proměnná jen do chvíle, než je *poprvé ztotožněna* s termem, který není volnou proměnnou, a potom už je *trvale vázána* na tento term.

V jednoduchém případě můžeme toto chování znázornit v grafu:

Vrcholy grafu zastupují použité termy – dvě konstanty a tři proměnné. Pokud jsou dva termy úspěšně ztotožněny, vložíme mezi ně hranu.

Navázání proměnné na term poznáme podle toho, že mezi nimi vede cesta.



V případě na obrázku byla proměnná A ztotožněna s proměnnou B a také s konstantou 1, proměnná C s konstantou 2. Ztotožnění mezi "levou" a "pravou" komponentou grafu již není možné.

Operátor `=` (rovnítko) provádí *ztotožnění* termů a je splněn, pokud ztotožnění bylo úspěšné. Snadno můžeme napsat predikát se stejnout funkcí: `ztotozni(A,A).`

Příklad 6:

```
?- A = 1, %Navázání volné proměnné A na číslo 1.
   A = B, %Porovnáním s volnou proměnnou B je i B
          %navázána na číslo 1.
   write(B). %Vypíše 1, odpověď je yes.
```

Příklad 7:

```
?- A = 1, %Navázání volné proměnné A na číslo 1.
   A = 2. %Číslo 1 se neshoduje s číslem 2.
          %Odpověď je no.
```

Příklad 8:

```
?- A = 1,
   B = 1,
   A = B. %Obě proměnné jsou navázány na číslo 1.
          %Odpověď je yes.
```

Příklad 9:

```
?- A = 1,
   B = 2,
   A = B. %Odpověď je no.
```

Příklad 10:

```
?- A = B, %Nejprve navážeme volné proměnné.
   A = 1, %Obě proměnné jsou navázány na číslo.
   write(B). %Vypíše 1.
```

Datové typy

Prolog sice rozlišuje datové typy termů, ale jedná se o *slabě typovaný jazyk*. Nemusíme tedy žádným způsobem udávat, s jakým datovým typem zamýšlíme pracovat, a často se ani nemusíme zabývat tím, jaký typ Prolog termu přisoudil.

Pokud nás typ termu přece jen zajímá a nejsme si jisti, můžeme použít vestavěné predikáty k jeho ověření:

`atom(X)`, `atomic(X)` (zahrnuje čísla), `float(X)` je totéž, co `real(X)`, `integer(X)`, `number(X)`, `nonvar(X)`, `var(X)`, `compound(X)`, `ground(X)`, `callable(X)`.

Pokud se vám zdá, že mezi predikáty *chybí* něco jako `boolean(X)`, máte pravdu a není to opomenutí. Zároveň jste si možná už všimli, že pokud lze u predikátů hovořit o návratové hodnotě, je jí právě pravdivostní hodnota.

Pravdivostní hodnotu jako takovou *nelze* v Prologu *uložit*. Pokud zapíšeme `A = true`, neztotožňujeme proměnnou s pravdivostní hodnotou, ale s termem, který je vždy splněn.

Také zápisem `A = write(ahoj)` neztotožňujeme proměnnou s pravdivostní hodnotou predikátu `write/1` (kterou má jako každý jiný predikát), ale se samotným predikátem, který lze kdykoli vyhodnotit. Zápisem `A,A,A` potom lze vypsat `ahojahojahoj`.

Omezující podmínky

Dosud jsme při tvorbě programu využívali velice základní prostředky Prologu, které odpovídají jeho nejrannějším verzím.

Prolog však poskytuje mnoho predikátů a operátorů, které nám umožňují *podmínit další vyhodnocování pravidla* – jejich splnění nebo nesplnění je něčím podmíněno, tudíž vložením na správné místo můžeme zajistit, že celé pravidlo nemůže být splněno a Prolog jeho vyhodnocování ihned ukončí.

Jedná se vlastně o kombinaci dvou deklarativních subparadigmat – čistě logického programování (logic programing) a programování s omezeními (constraint programming). Výsledkem je logické programování s použitím omezení (*constraint logic programming*).

Poznámka: *Nabízí se použití zkrácených označení jako "logické programování s omezeními" nebo "omezené logické programování", ale ta jsou poněkud zavádějící.*

K vytvoření omezujících podmínek samozřejmě můžeme použít i vlastní predikáty (například predikát **`spojeni`** v programu pro hledání cesty je vlastně použit jako omezující podmínka).

Nicméně, použití předdefinovaných operátorů je obvykle jednodušší, přehlednější a umožňuje definovat také složitější podmínky, které s vlastními predikáty definujeme jen obtížně.

První z těchto operátorů `=` jsme si již uvedli v souvislosti se ztotožňováním proměnných, ale evidentně ho lze využít i k podmíněnému provádění pravidla.

Predikát `not/1` je splněn, pokud jeho *argument nebyl splněn*. Má široké možnosti – v podstatě jen s ním můžeme zapsat mnoho omezujících podmínek. Pro jednoduchost a přehlednost však často použijeme některý předdefinovaný operátor.

Operátor `\=` testuje termy na neshodu. Je splněn, pokud ani jeden z termů *není volná proměnná* a termy *nelze ztotožnit*. Operátor je ekvivalentem zápisu `not(A=B)`.

Příklad 11:

```
?- A = 1,
   B = 2,
   A \= B. %Odpověď je yes.
```

Operátor `?=` je splněn, pokud termy *lze ztotožnit*, ale *nenavazuje* volné proměnné. Ekvivalent je `not(not(A=B))`.

Příklad 12:

```
?- A = 1,
   A ?= B,
   write(B). %B je volná proměnná, odpověď je yes.
```

Operátor `==` je splněn, pokud ani jeden z termů *není volná proměnná* a termy je *možno ztotožnit*. Volné proměnné operace *nenavazuje*.

Operátor `\==` je splněn, pokud alespoň jeden z termů *je volná proměnná* nebo termy *nelze ztotožnit*. Volné proměnné operace *nenavazuje*.

Kromě čárky (`,`) jako operátoru konjunkce můžeme používat i **středník** (`;`) jako *operátor disjunkce*, přestože v Hornově klauzuli se podle definice nevyskytuje – je to jedno z pozdějších rozšíření Prologu. Potom se hodí i *závorky* pro určení priority.

Nicméně, než zapisovat komplikovaná pravidla s několika disjunkcemi, může být přehlednější rozdělit náš záměr na více pravidel.

Příklad 13: Dva ekvivalentní zápisy predikátu:

```
sekvence(A) :- A = 1; A = 2; A = 3.
```

```
sekvence(A) :- A = 1.
sekvence(A) :- A = 2.
sekvence(A) :- A = 3.
```

Varování kompilátoru

Varovná hlášení lze zpravidla potlačit určitým nastavením, avšak nejen v B-Prologu, ale obecně, doporučuji toto nedělat.

Hlavně v benevolentnějších jazycích (slabě typovaných, bez deklarace proměnných apod.) mohou varování ve skutečnosti představovat poměrně závažné problémy, přestože z hlediska sémantiky jazyka nejde o fatální chybu.

Asi nejběžnější varovné hlášky B-Prologu jsou "singleton variable" a "predicate is not defined contiguously".

Varování "*singleton variable*" je vyvoláno, pokud se proměnná v klauzuli nachází jen na jediném místě. To prakticky znamená, že je navázána a dále nevyužita nebo testována bez navázání. Tato situace přirozeně vzniká, pokud máme predikát sestávající z více klauzulí, ale ne každá zpracovává všechny argumenty. Ostatní výskyty jsou obvykle *překlepy* v názvu proměnné. Vypnutím nebo ignorováním těchto varování ale na druhé straně riskujete, že budete překlepy hledat mnohem obtížněji.

Mnohem lepší řešení je použít *anonymní proměnnou* `_` nebo *podtržítko před názvem*, např. `_Promenna`, pokud chcete zachovat jméno pro větší přehlednost.

Varování "*predicate is not defined contiguously*" je vyvoláno tehdy, pokud predikát sestává z více klauzulí, které nejsou v programu uvedeny za sebou.

Pokud logická struktura vašeho programu nevyžaduje jiné uspořádání, doporučuji toto varování taktéž neignorovat a raději predikáty definovat souvisle – zvláště, pokud máte ve zvyku během vývoje zakomentovávat odložené části kódu a vedle nich vytvářet nové varianty. Toto varování vás pak může upozornit, že se do programu vloudila část kódu, kterou momentálně nechcete použít.

Správné použití řezu

Vraťme se nyní k programu na hledání cesty, konkrétně k použití predikátu "cut". Jádro programu vypadalo následovně:

```
cesta(_,_,X,X).
cesta(X,_,X,_) :- !, fail.
cesta(B,A,X,Y) :- prechod(X,Z), cesta(A,X,Z,Y).
```

Řez je zde *pro program velmi důležitý* – bez něj nebude mít druhé pravidlo žádný účinek (nebude nikdy splněno) a program "spadne" do rekurentní smyčky.

Tomuto použití řezu se někdy říká **červený řez** (red cut). Červená obvykle značí, že něco *není v pořádku*, a stejně je tomu i tady.

Problémem je, že predikát "cut" byl do Prologu zaveden pouze za účelem *zefektivnění programů* – má sloužit k tomu, abychom Prologu *zabránili zbytečně procházet větve*, pokud víme, že v nich nenalezne řešení.

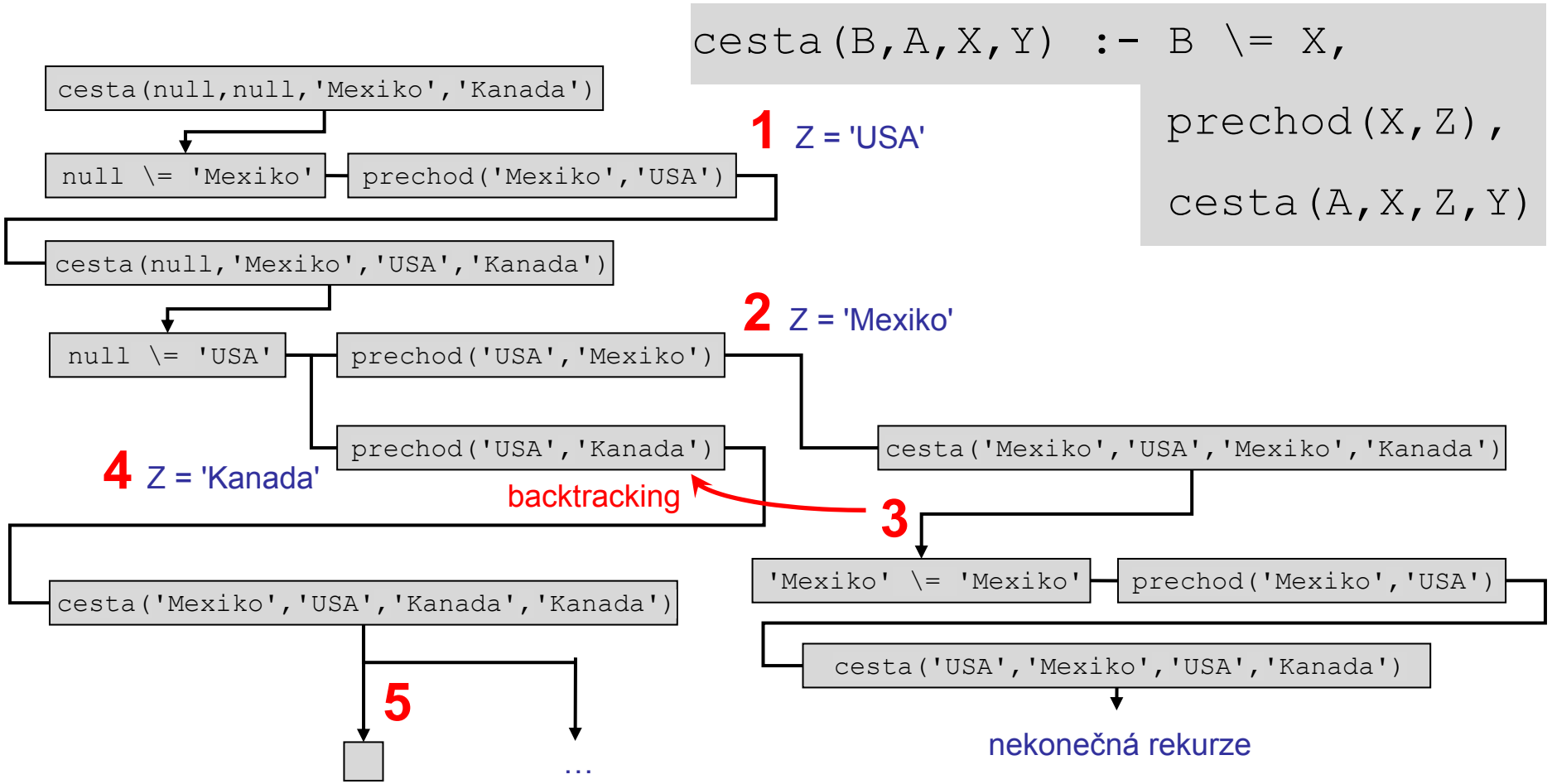
Takovému použití se říká **zelený řez** (green cut). Poznáme ho snadno podle toho, že *odstraněním řezů nezměníme chování* programu (jen možná poběží o něco déle), respektive tyto řezy do programu dodáváme až po jeho odladění (pro urychlení).

Červeným řezům se obvykle chceme *vyhnout*, protože značně narušují deklarativní charakter Prologu a *znepřehledňují* program – musíme dávat pozor, kde se vyskytuje predikát řezu. V podstatě je nutno sledovat tok programu od začátku do konce jako u imperativního programování.

Co tedy udělat s naším programem? Naštěstí jsme si ukázali spoustu operátorů, díky kterým můžeme rekurentní cyklus rozbít omezující podmínkou. Nabízí se třeba operátor `\=` :

```
cesta(_,_,X,X).
cesta(B,A,X,Y) :- B \= X, prechod(X,Z), cesta(A,X,Z,Y).
```

Jak je z ukázky vidět, nepěkné pravidlo s řezem můžeme úplně zahodit – viz následující obrázek:


cesta(B,A,X,Y) :- B \= X,
prechod(X,Z),
cesta(A,X,Z,Y)
cesta(null,null,'Mexiko','Kanada')
null \= 'Mexiko'
prechod('Mexiko','USA')
1 Z = 'USA'
cesta(null,'Mexiko','USA','Kanada')
null \= 'USA'
prechod('USA','Mexiko')
2 Z = 'Mexiko'
prechod('USA','Kanada')
4 Z = 'Kanada'
backtracking
cesta('Mexiko','USA','Mexiko','Kanada')
3
'Mexiko' \= 'Mexiko'
prechod('Mexiko','USA')
cesta('USA','Mexiko','USA','Kanada')
nekonečná rekurze
cesta('Mexiko','USA','Kanada','Kanada')
5
…

Aritmetické operace

V Prologu můžeme používat řadu obvyklých aritmetických operací: `+`, `-`, `*`, `/`, `//` nebo `div` pro celočíselné dělení, `mod` nebo `rem` pro zbytek po celočíselném dělení atd.

Také existují predikáty pro běžné aritmetické funkce, jako `abs`, `sin`, `cos` nebo `log`.

Nicméně, pokud se pro aritmetické operace pokusíte použít operátor ztotožnění `=` (případně jeho další varianty jako `\=`), narazíte pravděpodobně na překvapivé výsledky (viz dále).

Ztotožnění totiž zachází s celými termy tak, jak jsou zadány, bez ohledu na jejich složitost. Term `1+1` *nelze ztotožnit* s termem `2`, protože `1+1` je evidentně nějaký složený term, který rozhodně nevypadá stejně jako číselný atom `2`.

Musíme si uvědomit, že *aritmetické operátory* jako `+` nebo `-` nejsou stejné jako třeba `=`, protože *nevracejí pravdivostní hodnotu*. Jsou to jen znaky, dokud se nezeptáme na *aritmetickou hodnotu* výrazu. Pokud položíme dotaz `?-1=1`, Prolog odpoví `yes`, ale pokud zadáme `?-1+1`, Prolog nedokáže odpovědět (dojde k chybě) !

Pochopitelně nemá smysl zkoumat aritmetickou hodnotu termů, které nejsou aritmetickými výrazy. Aby term byl *aritmetický výraz*, musí se skládat pouze z *čísel*, *aritmetických operátorů* a *aritmetických funkcí*.

Operátor `is` provádí *ztotožnění aritmetických hodnot* termů. Pokud je levý term volná proměnná, je tato navázána na aritmetickou hodnotu výrazu napravo. Jinak je operator splněn, pokud se shodují aritmetické hodnoty výrazů.

Pravý argument nesmí být volná proměnná (operace tedy nefunguje obousměrně jako v případě `=`).

Příklad 14:

```
?- A = 1,
   B = A + 1,
   B = 2. %Odpověď je no (1+1 není totéž, co 2).
```

Příklad 15:

```
?- A is 1,
   B is A + 1,
   B is 2. %Odpověď je yes (1+1 má stejnou
hodnotu,
```

Příklad 16:

```
?- A is 1,
   B is 2. %Odpověď je no. Jakkoliv se operátor
           %”is” podobá klasickému přiřazení,
           %nemůže změnit navázání promněnné.
```

Příklad 17: Když už se zabýváme aritmetikou, můžeme si konečně i v Prologu zapsat oblíbený příklad s faktoriálem:

```
faktorial(0,1) :- !.
faktorial(N,F) :- N1 is N-1, faktorial(N1,F1), F is F1*N.
```

Nejprve rekurentně snižujeme N až na 0, potom při návratu z rekurze používáme proměnnou F, ve které násobením dáme dohromady výsledek.

Pozor, *nelze použít* zápis `faktorial(N is N-1,F1)`, to je operace s výsledkem ano/ne jako každá jiná v Prologu a nikoli term, který můžeme použít jako argument.

Zákeřnější chybou může být zápis `faktorial(N-1,F1)` – tím vznikne platný term, takže Prolog si sám od sebe nebude stěžovat, ale rozhodně nikoli očekávaná aritmetická hodnota, takže nedostaneme očekávaný výsledek.

Jistě vám neuniklo *použití řezu* v příkladu. Bez jeho použití by Prolog nalezl správné řešení, ale na požádání by byl ochoten pokračovat ve snižování N do záporných hodnot, což by vedlo k zacyklení. Řez lze tedy označit za červený řez, protože ovlivňuje chování programu.

K odstranění potřeby řezu by dále bylo možno vložit omezující podmínku do druhého pravidla, byla by to ovšem další podmínka, vyhodnocovaná v každém kroku.

Také lze program upravit na jediné pravidlo a vyhnout se řezu i dodatečné podmínce, sníží se ovšem přehlednost programu:

```
faktorial(N,F) :- (N = 0, F = 1);
                  (N1 is N-1, faktorial(N1,F1), F is F1*N).
```

Jako u všech pravidel, legitimita řezu tedy není dána jednoznačně. O jeho použití je třeba rozhodnout také na základě okolností.

Další operátory pro zjištění shody/neshody aritmetických hodnot jsou vhodné, pokud chcete mít jistotu, že *nedopatřením nenavážete* volnou proměnnou, tj. pro větší *robustnost kódu*.

Operátor `=:=` je splněn, pokud se *shodují aritmetické hodnoty* termů. Ani jeden z termů *nesmí být volná proměnná*.

Operátor `=\=` je splněn, pokud termy mají *rozdílné aritmetické hodnoty*. Ani jeden z termů *nesmí být volná proměnná*.

Porovnávací operátory

Pomocí standardních operátorů můžeme v Prologu porovnávat termy dvěma způsoby:

Operátory `<`, `>`, `=<`, `>=` porovnávají *aritmetické hodnoty* výrazů. Jde tedy o běžné porovnání, jaké očekáváme. *Neexistují* však operátory `<=` a `=>`, snad aby se předešlo záměně s implikací.

Operátory `@<`, `@>`, `@=<`, `@>=` porovnávají podle "standardního uspořádání termů", což v B-Prologu znamená podle abecedy, resp. podle hodnot jednotlivých znaků.

V podstatě tyto operátory odpovídají operátorům pro porovnávání řetězců, nicméně je třeba si uvědomit, že termy nejsou řetězce a rovněž nelze očekávat, že budou správně řadit termy např. podle pravidel češtiny.

Příklad 18:

```
?- (1+3 > 2+1), %Tato podmínka je splněna
   (1+3 @< 2+1). %Tato také
                 %Odpověď je yes.
```

Seznamy

Standardním prostředkem pro uložení libovolného množství prvků jsou v Prologu seznamy.

Prvkem seznamu může být libovolný term. Syntax Prologu se seznamy počítá, tudíž vlastní seznam si můžeme vytvořit jednoduše: `A = [1,2,3]` – proměnná A je navázána na seznam tří číselných hodnot.

Seznamy zapisujeme do *hraných závorek* a k oddělení prvků slouží čárky. Zápis `[]` znamená *prázdný seznam*.

Seznamy mohou být i víceúrovňové – `[[1,2,3],[4,5]]` je dvouprvkový seznam seznamů, `[[]]` je seznam obsahující prázdný seznam. Nakonec, seznam je také "jen" term.

Seznamy v Prologu (ať už je daná implementace řeší jakkoliv) lze nejlépe přirovnat k *jednosměrným spojovým seznamům* (bez ukazatele na poslední prvek) – můžeme snadno přistoupit na jejich začátek a libovolně je rozšiřovat, ale dostat se doprostřed nebo nakonec seznamu je poměrně náročné.

Vytvořit seznam je sice pěkné, ale nepříliš užitečné, pokud s ním nedokážeme manipulovat.

Základním nástrojem k tomuto účelu je syntaktická konstrukce `[a,b,c|S]`, která značí *seznam*, jehož *první tři prvky* jsou *a, b, c*, a prvky, které obsahuje *seznam S*, budou *zbývající prvky*.

Prvky na začátku (často jen jeden term) jsou označovány **hlavička** seznamu (*head*), seznam *S* označíme jako **zbytek** seznamu. (Doslovný překlad anglického *tail* není v české terminologii zaveden, takže se mu raději vyhneme.)

Pokud je jako hlavička seznamu uvedena volná proměnná, např. `[H|T]`, je proměnná *H* ztotožněna s *prvním prvkem* seznamu.

Přestože se to možná nezdá, tento nástroj umožňuje dělat se seznamy cokoliv.

Příklad 19: Operátor lze použít k odebrání prvku ze seznamu:

```
?- L = [a,b,c],
   L = [_|M],
   M = [b,c].
```

Příklad 20: Obdobně lze operátor použít i k přidání prvku:

```
?- M = [b,c],
   L = [a|M],
   L = [a,b,c].
```

Protože dělat vše touto jedinou operací je přece jen trochu nešikovné, různé implementace Prologu poskytují předdefinované predikáty pro práci se seznamy.

Poznámka: *Žádný z uvedených predikátů sice nedefinuje ISO norma Prologu, ta je však poměrně slabá a specifikuje prakticky jen základní nezbytnosti. Pro pokročilejší rozšíření, kterými se budeme dále zabývat, je tedy poznámka "not in ISO" v dokumentaci běžný jev.*

Nicméně, alespoň ty běžnější z nich najdeme ve stejné podobě minimálně ve známějších implementacích Prologu, takže si s jejich použitím nemusíme dělat těžkou hlavu.

Predikát `append(A,B,C)` představuje *spojení* dvou *seznamů*. Všechny tři argumenty tedy musejí být seznamy – seznam *B* je připojen *na konec* seznamu *A* a následuje *výsledek C*.

Tímto predikátem lze přidat prvek na konec seznamu, ale u delšího seznamu to trvá déle (hned si ukážeme), takže to nemusí být nejlepší nápad.

Příklad 21: Běžné použití ke spojení dvou seznamů:

```
?- append([a,b,c],[d,e],L),
   L = [a,b,c,d,e].
```

Příklad 22: Predikát lze použít i opačně – zjistit, co je třeba přidat, abychom získali požadovaný seznam:

```
?- append([a,b,c],L,[a,b,c,d,e]),
   L = [d,e].
```

Příklad 23: Můžeme se i zeptat, z jakých všech seznamů lze požadovaný seznam složit. Zde už dostaneme i pro takto krátký seznam celkem 4 možnosti, na ukázku je uvedena jen ta první:

```
?- append(M,N,[a,b,c]),
   M = [], N = [a,b,c].
```

Příklad 24: Pokud by snad předdefinovaný predikát náhodou nebyl k dispozici, můžeme si ho snadno napsat sami:

```
pripoj([],L,L). %Druhý seznam je základ pro výsledek
pripoj([H1|T1],L2,[H1|L3]) :- pripoj(T1,L2,L3).
  %Prvky z prvního seznamu postupně přidává na začátek
  %výsledného seznamu
```

Tento predikát ale v podstatě funguje opačně, než jak predikát `append` intuitivně popíšeme – na začátek druhého seznamu přidá prvky prvního seznamu. Mít výrazně *kratší druhý seznam* (např. jednoprvkový) se tedy *nevyplatí*.

Poznámka: Pokud nevěříte, že predikát sestávající ze dvou krátkých klauzulí má všechny funkce predikátu `append`, můžete si to sami vyzkoušet.

Vzhledem k tomu, že postup Prologu při vyhodnocování predikátu přece jen nemusí být na první pohled jasný, zkusme si rozebrat, co se bude dít při spojování seznamů:

Příklad 25: Zadáme cíl:

```
?- pripoj([a,b,c],[d,e],S).
```

Aby vyhovoval pravidlu

```
pripoj([H1|T1],L2,[H1|L3]) :- pripoj(T1,L2,L3). ,
```

musíme argumenty chápat následovně (L3 zatím volná prom.):

```
pripoj([a|[b,c]],[d,e],[a|L3]) :- pripoj([b,c],[d,e],L3).
```

Rekurzivně aplikujeme pravidla

```
pripoj([b|[c]],[d,e],[b|L3]) :- pripoj([c],[d,e],L3).
```

```
pripoj([c|[]],[d,e],[c|L3]) :- pripoj([],[d,e],L3).
```

a aplikujeme ukončovací podmínku

```
pripoj([],[d,e],[d,e]).
```

A nyní sestavení výsledku při návratu z rekurze:

```
pripoj([c|[]],[d,e],[c|[d,e]]) :- pripoj([],[d,e],[d,e]).
pripoj([b|[c]],[d,e],[b|[c,d,e]]) :-
                                    pripoj([c],[d,e],[c,d,e]).
pripoj([a|[b,c]],[d,e],[a|[b,c,d,e]]) :-
                                  pripoj([b,c],[d,e],[b,c,d,e]).
?- pripoj([a,b,c],[d,e],[a,b,c,d,e]).
```

Jak je na tom náš predikát s efektivitou? Docela dobře. Doba běhu se prakticky shoduje s předdefinovaným predikátem, přičemž u obou predikátů je přímo úměrná pouze délce prvního seznamu. Na druhém seznamu doba zpracování nezáleží.

Predikát `append/4` funguje obdobně jako `append/3`, ale místo dvou seznamů spojuje hned tři seznamy. *Efektivita* tohoto predikátu je *srovnatelná* se dvěma použitími `append/3` po sobě.

Příklad 26: Ke splnění uvedeného cíle vede 10 kombinací, počínaje `L=[a,b,c]`, `M=[]`, `N=[]` a konče `L=[]`, `M=[]`, `N=[a,b,c]`:

```
?- append(L,M,N,[a,b,c]).
```

Predikát `member(T,S)` najde při práci se seznamy časté využití – je splněn v případě, že se term *T* nachází *v seznamu*, který je zadán jako *S*.

Predikát seznam prohledává od začátku do konce v *O(n)* čase.
Doba hledání pro tisíce položek: < 1 ms.
Doba hledání pro miliony položek: x10 – x100 ms.

Příklad 27:
```
?- member(b,[a,b,c]). %Odpověď je yes.
```

Příklad 28:
```
?- member(b,[[a,b,c],[d,e]]). %Odpověď je no.
```

Příklad 29:
```
?- member([d,e],[[a,b,c],[d,e]]). %Odpověď je yes.
```

Příklad 30: Pochopitelně také predikát `member` lze použít opačně, tedy k průchodu seznamem:

```
?- member(X,[a,b,c]),
   write(X), write(' '), fail. %Vypíše a b c
```

Příklad 31: Pokud je term v seznamu vícekrát, bude vícekrát vypsán:

```
?- member(X,[a,b,b,a,b,b,a,b,c,a,b,d]),
   write(X), fail. %Vypíše abbabbabcabd
```

Predikát `delete(S,T,V)` *odstraní ze seznamu S* všechny výskyty termu *T* a výsledný seznam představuje V.

Příklad 32: Použití pro odstranění prvků ze seznamu:

```
?- append([a,b,b,a,b,b,a,b,c,a,b,d],b,L),
   L = [a,a,a,c,a,d].
```

Příklad 33: Zjevně nelze zrekonstruovat původní seznam – není možné říct, odkud a kolik výskytů bylo odstraněno:

```
?- append(L,b,[a,a,a,c,a,d]). %chyba
   %L = ???
```

Příklad 34: Test, zda lze z jednoho seznamu vytvořit druhý odstraněním jediného termu:

```
?- delete([a,b,c],_,[b,c]). %Odpověď je yes.
```

Poznámka: *Jak je patrné z dosavadních příkladů, manipulace se seznamem "nezničí" původní seznam – změněný seznam musíme vždy navázat na jinou proměnnou, jinak by neuspělo ztotožnění.*

Pravidlo, že navázání proměnné nelze dodatečně změnit neporušujeme ani v tom smyslu, že změníme seznam, na který je navázána.

Predikát `length(S,L)` ztotožňuje *seznam S* a *délku* tohoto seznamu *L*.

Pokud chceme *zjistit délku* seznamu (L je volná proměnná), predikát sice pracuje v čase *O(n)*, ale je velice *rychlý* (pro miliony položek řádově milisekundy).

Pokud nás však zajímá, zda má seznam *určitou délku* (*L* je vázaná proměnná), je *mnohem pomalejší* – rychlost odpovídá predikátu `member` (pro miliony položek až stovky milisekund). Predikát pro tento účel zjevně není optimalizován.

Predikát `closetail/1` uzavře seznam tak, že odřízne zbytek seznamu (tail), pokud se jedná o volnou proměnnou. Pokud však seznam není zadán v podobě `[Hlavička|Zbytek]` nebo zbytek není volná proměnná, nedělá nic a je splněn.

Odříznutí provede jednoduše tak, že volnou proměnnou naváže na prázdný seznam `[]` – tím jakoby zbytek seznamu zmizel.

Příklad 35:

```
?- L = [a,b,c|Zbytek],
   closetail(L),
   L = [a,b,c]. %Odpoveď je yes.
```

Příklad 36:

```
?- L = [a,b,c,Zbytek],
   closetail(L),
   L = [a,b,c,Zbytek]. %Seznam není otevřený -
                       %beze změny.
```

Poznámka: *Možná se zdá, že uvedené pozorování o "neničení" seznamu je tímto predikátem nabouráno, ale nikoli – pokud jsou v seznamu volné proměnné, můžeme je pochopitelně navázat, ale samotný seznam se tím nemění.*

Rozdílové seznamy

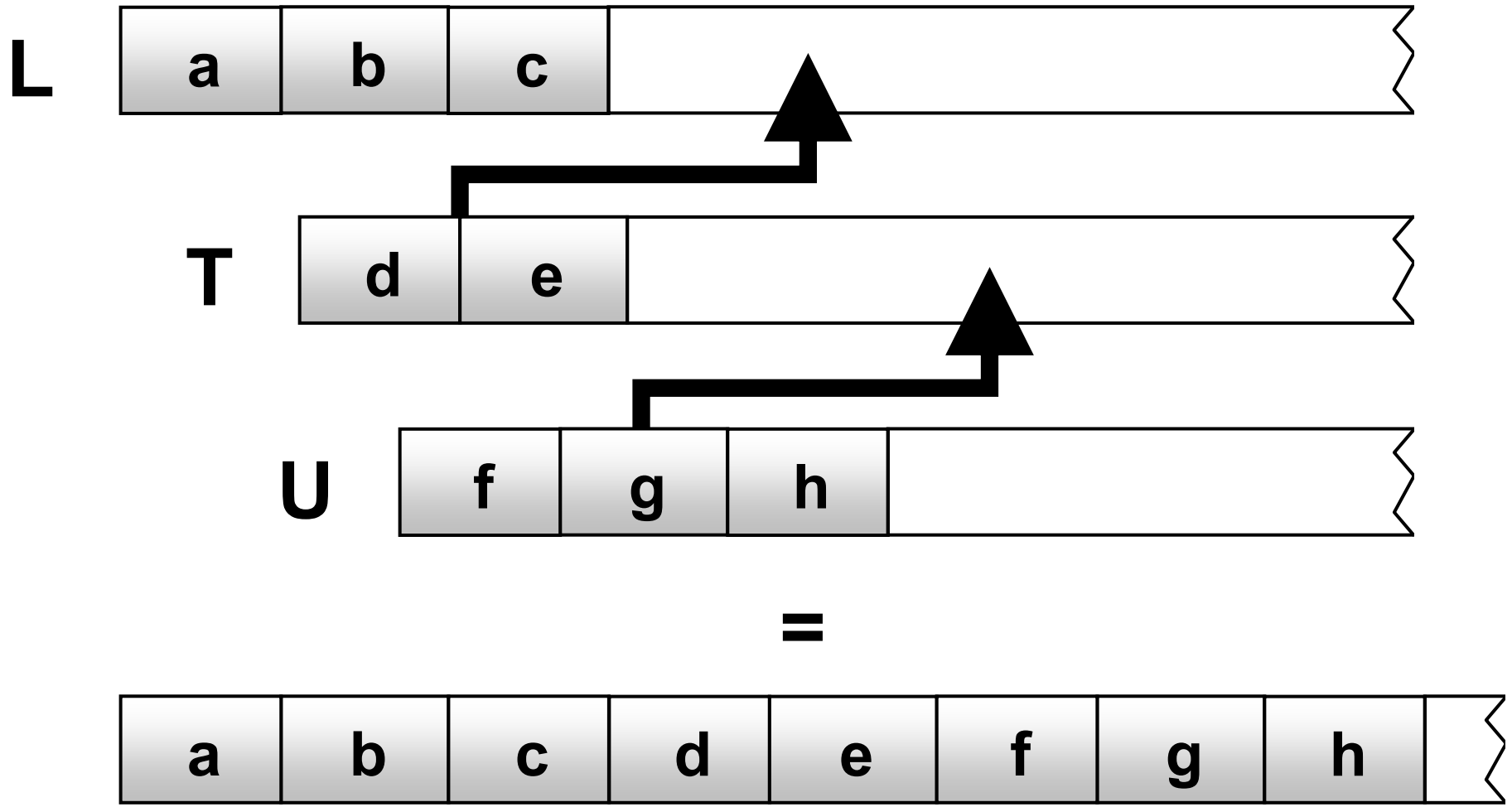
Přes všechno, co jsme již definovali, jsme se stále ještě nedostali k tomu, jak efektivně přidávat prvky na konec seznamu. Řekli jsme si, že predikát `append` je pro dlouhé seznamy *pomalý*, ale co s tím?

Predikát `closetail` se možná zdá nezajímavý a neužitečný, ale samotná jeho existence nás přivádí k technice zvané *rozdílové seznamy* (difference lists), která možná vyřeší náš problém. Myšlenka je zcela prostá – *neuzavírejme seznam*.

Příklad 37: Přidávání do otevřeného (rozdílového) seznamu:

```
?- L = [a,b,c|T], %Vytvoření rozdílového seznamu
   T = [d,e|U], %Přidání 2 prvků
   U = [f,g,h|V], %Přidání dalších 3 prvků
   V = [], %Uzavření seznamu (nebo použít closetail na L,
           %pokud to nemůžeme udělat takto přímo).
   L = [a,b,c,d,e,f,g,h]. %Do seznamu přidáno bez rekurze
                          %a opakovaného procházení – v čase O(1).
```

Nejde o nic nového – jen o uvědomění si možnosti této konstrukce, nebo faktu, že seznam můžeme nechat otevřený, jak dlouho chceme.


L
a
b
c
T
d
e
U
f
g
h
=
a
b
c
d
e
f
g
h

Příklad 38: Znovuotevření seznamu pro přidávání:

```
?- L = [a,b,c],
   append(L,[d|T],M),
   M = [a,b,c,d,T].
```

Uzavřený seznam můžeme predikátem "append" znovu otevřít (společně s přidáním dalšího prvku), pokud víme, že chceme přidat více prvků. Není to náročnější než pouhé přidání prvku a můžeme ušetřit hodně času při přidávání prvků dalších.

Pokud se podíváme zpět na naši vlastní implementaci predikátu "append" a pokusíme se ji napsat se znalostí rozdílových seznamů, dobereme se ke stejnému výsledku (je zde však použita k přidávání prvků po jednom, proto ta neefektivita) – tuto techniku tedy někdy použijeme jen intuitivně, nicméně, její znalost pomůže vždy zvážit, zda je použité řešení opravdu efektivní.

S rozdílovými seznamy si příliš nerozumějí predikáty pro práci se seznamy – můžeme dostávat podivné výsledky (např. test na přítomnost termu pomocí "member" nikdy neuspěje). Nejlepší řešení tedy může být vybudovat seznam jako rozdílový, potom ho uzavřít a dále s ním pracovat. Samozřejmě záleží na situaci.

Řetězce

Řetězce jsou v Prologu reprezentovány seznamem znaků, resp. jejich číselných kódů. Řetězec můžeme definovat jednoduše tak, že text uzavřeme do uvozovek.

Predikát `write_string/1` *vypíše* na standardní výstup *řetězec* v čitelné podobě. Argumentem musí být *seznam číselných kódů*. Pro řetězce je potřeba, protože `write` vypíše seznam čísel tak, jak skutečně vypadá, a „kolem“ řetězce vypisuje uvozovky.

Příklad 39: Výpis atomu a výpis řetězce.

```
?-Atom = 'Toto je atom, ktery je dale nedelitelnou jednotkou',
  write(Atom), nl, %Vypíše atom tak, jak je pojmenován
  Retezec = "Toto je retezec, reprezentovany seznamem cisel",
  write(Retezec), nl, %Vypíše [84,111,116,111,32,106,101,32,…]
  write_string(Retezec). %Vypíše "Toto je retezec, …"
```

Predikát `format(P,H)` vypíše *řetězec P*, přičemž nahradí *řídicí značky* v něm obsažené. *H* je seznam obsahující *hodnoty* potřebné k nahrazení řídicích značek a vypisuje se bez uvozovek.

Některé řídicí značky (modifikátor N je nepovinný)

~a	Vloží atom.
~Nc	N-krát vloží znak (podle zadaného kódu).
~Nn	N-krát vloží odřádkování.
~Nd	Vloží celé číslo délky minimálně N (doplní mezery zleva)
~Nf	Vloží desetinné číslo na N desetinných míst.
~NR	Převede celé číslo na základ N (2 až 36) a vloží do řetězce.
~Ns	Vloží řetězec délky N (doplní mezery zprava).

Příklad 40:

```
?- format("~5c5 mezer",32).
%Vypíše "     5 mezer" (bez uvozovek)
```

Příklad 41:

```
?- format("~6d~n~6d",[4294967296,256]). %Vypíše:
%4294967296
%   256
```

Příklad 42:

```
?- format("~36R",[44027]). %Vypíše: XYZ
```

Příklad 43:

```
?- format("~16R ~R",[16,8]). %Vypíše: 10 10
```

Predikát `atom_chars(A,Z)` rozloží *atom* A na *znaky* a ztotožní je se *seznamem* atomů Z. Stále se však jedná o seznam obecných atomů, nejde o řetězec!

Predikát `atom_codes(A,R)` rozloží atom A na znaky a kódy těchto znaků ztotožní se seznamem atomů R. *Převádí* tedy mezi *atom* A na *řetězec* R nebo opačně.

Příklad 44:

```
?- atom_chars(pivo,Znaky),
   write_string(Znaky), nl, %Chyba
   atom_codes(pivo,Retezec),
   write_string(Retezec). %Vypíše pivo
```

Predikát `number_chars/2` dělá totéž, co `atom_chars`, ale pracuje s numerickou hodnotou.

Predikát `number_codes/2` dělá totéž, co `atom_codes`, ale pracuje s numerickou hodnotou. *Převádí* tak *číselnou hodnotu arg1* na *řetězec arg2* nebo opačně.

Příklad 45:

```
?- atom_chars(3.14,Znaky),
   write_string(Znaky), nl, %Chyba
   write(Znaky), nl, %Vypíše ['3','.','1','4']
   atom_codes(pivo,Retezec),
   write_string(Retezec). %Vypíše □3.14□
```

Predikát `sub_atom(A,S,L,R,B)` "vyřízne" z atomu část, která se stane novým atomem. A je původní *atom*, S je *začátek* subatomu, L je *délka* subatomu, R je *zbývající délka* atomu za subatomem, B je *subatom*.

Příklad 46:

```
?- sub_atom(nejneobhospodarovavatelnejsi,7,7,_,B).
   %B = hospoda
   sub_atom(nejneobhospodarovavatelnejsi,S,L,R,B)
   %S = 7, L = 7, R = 14
```

Predikát `name/2` funguje jako `atom_codes` či `number_codes` (pro čísla i ostatní atomy), ale ne zpětně (pouze *atom na řetězec*).

Procházení cyklického grafu

Nadešel čas vrátit se k našemu programu pro hledání cesty. Opustili jsme ho sice v utěšeném stavu, ale jen do chvíle, než se rozhodneme trochu rozšířit naši mapu světa.

Dosud jsme v našem grafu neměli žádný cyklus. Stačilo nám tedy zabránit vracení a nehrozilo nám, že se dostaneme někam, kde jsme už byli. Taková situace však těžko bude někdy odpovídat kompletní mapě světa.

Na následujícím snímku je uveden celý dosavadní program.

```
sousedi('Kanada','USA').
sousedi('USA','Mexiko').
sousedi('Rusko','Cina').
sousedi('Cina','Indie').

prechod(X,Y) :- sousedi(X,Y).
prechod(X,Y) :- sousedi(Y,X).

cesta(_,_,X,X).
cesta(B,A,X,Y) :- B \= X, prechod(X,Z), cesta(A,X,Z,Y).
cesta(X,Y) :- cesta(null,null,X,Y).
```

Stačí pouze lehce doplnit mapu Asie, aby v grafu vznikl cyklus:

```
sousedi('Kanada','USA').
sousedi('USA','Mexiko').
sousedi('Rusko','Cina').
sousedi('Rusko','Mongolsko').
sousedi('Mongolsko','Cina').
sousedi('Cina','Kazachstan').
sousedi('Kazachstan','Rusko').
sousedi('Cina','Indie').
```

Jedním „správným“ dotazem lze náš program opět zacyklit:

```
?- cesta('Rusko', 'Indie').
```

Zřejmě potřebujeme trochu dlouhodobější paměť, abychom se vyhnuli „chození v kruhu“ – nejlépe trvalou. Dobře, že už známe seznamy.

Provedeme dvě jednoduché úpravy:

1. Zrušíme proměnné pro předcházející a před-předcházející stát a místo nich přidáme jedinou proměnnou P pro seznam. Do seznamu budeme přidávat každý navštívený stát.
2. Dosavadní podmínku nahradíme predikátem member, abychom zjistili, zda jsme stát nenavštívili kdykoli předtím.

Dostaneme novou podobu predikátů `cesta/3` a `cesta/2`:

```
cesta(X,X,_).
cesta(X,Y,P) :- not(member(X,P)),
                prechod(X,Z), cesta(Z,Y,[X|P]).
cesta(X,Y) :- cesta(X,Y,[]).
```

Teď už se nikdy nestane, že by se hledání cesty zacyklilo. Zjistíme však pouze, zda cesta existuje a ne, jaká opravdu je. Kompletní cesta existuje jen na nejvyšší úrovni zanoření, navíc v opačném pořadí.

Upravíme tedy program tak, aby při návratu z rekurze sestavil výslednou cestu a poskytl nám ji "ven":

```
cesta(X,X,_,[X]). %Cíl na konec výsledné cesty
cesta(X,Y,CP,[X|FP]) :- not(member(X,CP)),
                        prechod(X,Z), cesta(Z,Y,[X|CP],FP).
cesta(X,Y,P) :- cesta(X,Y,[],P).
```

Proměnná „CP“ představuje dosavadní prošlou cestu, „FP“ představuje výslednou nalezenou cestu.

Pokud se nyní zeptáme na cestu z Indie do Ruska, můžeme si postupně vyžádat následující sadu možných cest:

```
P = ['Indie','Cina','Kazachstan','Rusko']
P = ['Indie','Cina','Kazachstan','Rusko','Mongolsko','Rusko']
P = ['Indie','Cina','Rusko']
P = ['Indie','Cina','Rusko','Mongolsko','Rusko']
P = ['Indie','Cina','Rusko','Kazachstan','Rusko']
P = ['Indie','Cina','Mongolsko','Rusko']
P = ['Indie','Cina','Mongolsko','Rusko','Kazachstan','Rusko']
```

Jak je vidět, některé cesty jsou kratší, některé delší, v některých případech se dokonce v Rusku "usadíme" až na druhý pokus (to proto, že Prolog po splnění první klauzule zkouší alternativně splnit i druhou). My se ale nechceme probírat spoustou možných cest, zajímala by nás jen ta nejkratší, respektive cesta přes nejméně států.

Poznámka: *Pochopitelně, posuzovat délku cesty podle počtu projetých státu je trochu naivní. To by rozumně fungovalo možná v končinách, kde překonání hranice zabere celý den. To teď ale necháme stranou. Kdybychom chtěli program dále rozvíjet, mohli bychom nahradit státy silnicemi, celé silnice silničními úseky, přidat ohodnocení podle délky a kvality. To už se ale dostáváme do zcela jiných poměrů.*

Pokud nás nějaké varianty cesty určitě nezajímají, jsou to ty, kde cíl navštívíme vícekrát. Těch se zbavíme naším ospravedlněným použitím řezu v první klauzuli.

V tomto případě už jde vlastně o zelený řez – Prolog díky němu neztrácí čas hledáním nesmyslných řešení, kterých bychom se později stejně zbavili:

```
cesta(X,X,_,[X]) :- !. %Už nic nezkoušet.
cesta(X,Y,CP,[X|FP]) :- not(member(X,CP)),
                        prechod(X,Z), cesta(Z,Y,[X|CP],FP).
cesta(X,Y,P) :- cesta(X,Y,[],P).
```

```
P = ['Indie','Cina','Kazachstan','Rusko']
P = ['Indie','Cina','Rusko']
P = ['Indie','Cina','Mongolsko','Rusko']
```

Zde máme novou množinu cest. Ale jak tedy najít nejkratší cestu? Naštěstí v Prologu umíme počítat a i posoudit, které číslo je větší.

Mohli bychom zkusit najít všechny možné cesty a následně si vybrat. S rostoucím počtem států a rozsahem mapy však obrovsky vzroste také počet možných cest, přičemž většina z nich bude absurdně zdlouhavá a komplikovaná, zatímco vhodná cesta bude nalezitelná v poměrně krátkém čase.

Prologovská metoda hledání řešení je vlastně *nativní hledání do hloubky* (deep-first-search). Její dobře známou nevýhodou je neefektivita, pokud se řešení nachází relativně „mělko“ ve stromě.

Hledání do šířky (breadth-first search) je oproti tomu paměťově náročnější – v podstatě si musí pamatovat všechny zkoumané cesty najednou. Navíc, v případě jeho použití, musíme *zahodit* přirozené *chování Prologu* a vyvinout *vlastní postup* hledání řešení.

Jako kompromis se tak nabízí metoda *iterativního prohlubování* či iterativního zanořování (iterative deepening) – česká terminologie zde není jednoznačná.

Metoda spočívá v prohledávání grafu do hloubky s omezenou maximální hloubkou, přičemž v případě neúspěchu se maximální hloubka zvýší a vyhledávání se opakuje.

Takový postup vypadá jako neefektivní opakování výpočtů, ale pokud opět uvážíme, že řešení leží *mělko ve stromě*, snadno může být *mnohem rychlejší než* prohledání jediné větve až na konec metodou *DFS*.

Zkusíme tedy prohlubovat: Predikát `cesta/4` změníme na `cesta/5` – přidáme čítač, který když klesne na nulu, hledání ukončíme:

```
cesta(X,X,_,[X],_) :- !.
cesta(X,Y,CP,[X|FP],L) :- not(member(X,CP)),
    L > 0, L1 is L-1, %Snížení čítače za navštívený stát
    prechod(X,Z), cesta(Z,Y,[X|CP],FP,L1).
```

Nyní ještě musíme upravit predikát `cesta/3`, kterým hledání zahajujeme, aby ho spouštěl opakovaně.

Pokud prohlubování implementujeme tak, že za každý pokus se maximální hloubka zvýší o 1, určitě jako první dostaneme nejkratší cestu (nebo jednu z nich).

Určitě lze zapsat predikát `sekvence/1`, který nejprve poskytne hodnotu 1 a za každý návrat („backtrack“) a další pokus hodnotu o 1 zvýší. Použijeme ho tedy:

```
cesta(X,Y,P) :- sekvence(L), cesta(X,Y,[],P,L), !.
```

Na konec je ještě přidán řez, aby Prolog nepokračoval hledáním delších cest.

Počet cest z Indie do Ruska se nám tak zredukoval na jedinou:

```
P = ['Indie','Cina','Rusko']
```

Zároveň jsme však oživili již vyřešený problém. Pokud se teď zeptáme na cestu z Ruska do USA, dostaneme nekonečnou rekurzi – Prolog v případě neúspěchu zvýší hloubku hledání o 1 a to dělá do nekonečna, bez ohledu na to, že maxima nebylo dosaženo.

V praxi asi budeme očekávat, že nějakou cestu na mapě vždy najdeme (pokud zahrneme i námořní a leteckou dopravu).

V takovém případě nebude příliš bolestivé řešit problém podmínkou na maximální hloubku. Tu vytvoříme úpravou predikátu `sekvence`, aby bylo možno zadat konečnou hodnotu:

```
cesta(X,Y,P) :- sekvence(L,8), cesta(X,Y,[],P,L), !.
```

Pokud chceme mít jistotu, že hledání neskončí předčasně, určitě ho můžeme zajistit ukončením, když hloubka dosáhne počtu vrcholů v grafu. A to jsme právě udělali.

Pokud chceme za každou cenu přesně určit, kdy nemá smysl pokračovat, vyžádá si to trochu složitější úpravy.

Potřebujeme detekovat, zda při hledání řešení bylo dosaženo maximální hloubky, a tedy, zda má smysl učinit další pokus.

Přidáme další argument RL, jehož jediným účelem bude předání nevyužitého počtu kroků ven z rekurze:

```
cesta(X,X,_,[X],L,L) :- !.
cesta(X,Y,CP,[X|FP],L,RL) :- …
```

To je ale samo o sobě k ničemu, protože v případě nenalezení cesty je užit backtracking a veškeré údaje o průběhu výpočtu jsou tím zrušeny.

Proto, i když to zní divně, přidáme do predikátu třetí klauzuli, která zajistí, že predikát bude pokaždé splněn:

```
cesta(X,X,_,[X],L,L) :- !.
cesta(X,Y,CP,[X|FP],L,RL) :- …
cesta(X,_,_,[X],L,L).
```

Hlavička se shoduje s první klauzulí, jediný rozdíl je v tom, že nevyžaduje dojít do cíle.

Díky jeho umístění na konci ho Prolog zkouší až tehdy, nemá-li jinou možnost, takže úspěšné nalezení cesty není ovlivněno.

O úspěšném nalezení cesty musíme ale rozhodnout mimo tento predikát. Proto vložíme ještě jeden argument D (jako arg5), který opět bude sloužit pouze pro předání státu, kde hledání skončilo, ven z rekurze:

```
cesta(X,X,_,[X],X,L,L) :- !.
cesta(X,Y,CP,[X|FP],D,L,RL) :- …
cesta(X,_,_,[X],X,L,L).
```

Díky tomu můžeme potom rychle rozhodnout o úspěchu hledání.

Pro náš záměr však budeme ještě potřebovat predikát, který nám umožní získat celou množinu řešení a dále s nimi pracovat.

Vytváření seznamů řešení

Predikát `findall(T,C,V)` ztotožňuje *V* se *seznamem* všech *termů*, které lze ztotožnit s T, aby byl splněn cíl C.

Lepší představu o použití predikátu dává následující příklad:

Příklad 47:

```
?- findall(X,member(X,[(1,a),(2,b),(3,c)]),Xs),
   Xs = [(1,a),(2,b),(3,c)].
```

Příklad sice prakticky nic nedělá, ale je přehledný a snadno pochopitelný.

Predikát `bagof(T,C,V)` se podobá predikátu `findall` a v předchozím příkladu by se i choval stejně. Liší se však v zacházení s volnými proměnnými, které se nevyskytují v termu T, ale jsou v cíli C.

Zatímco `findall` tyto proměnné v podstatě ignoruje (chová se, jakoby byly anonymní proměnná), `bagof` je postupně ztotožňuje se všemi možnými hodnotami a pro každý z případů zároveň ztotožňuje seznam V s hodnotami ostatních proměnných, aby byl cíl splněn. Predikát tedy poskytuje celou množinu řešení.

Šanci na pochopení predikátu `bagof` doufejme dává následující příklad, kde je také uvedeno srovnání s predikátem `findall`.

Příklad 48: Porovnání chování predikátu `bagof` a `findall`:

```
?- bagof(Y,member((X,Y),[(1,a),(2,b),(3,c)]),Xs),
   ((X=1, Xs=[a]);
    (X=2, Xs=[b]);
    (X=3, Xs=[c])).
```

`bagof` poskytne množinu tří dvojic hodnot proměnných `X` a k nim všech hodnot `Y` v seznamu `Xs`, které jsou možná řešení.

```
?- findall(Y,member((X,Y),[(1,a),(2,b),(3,c)]),Xs),
   Xs = [a,b,c].
```

Findall poskytne prostý seznam termů, se kterými lze ztotožnit `Y`.

Teď se můžeme vrátit k problému hledání cesty. Nemusíte se obávat, že bychom použili predikát `bagof`, nicméně použijeme `findall` k úpravě výchozího predikátu `cesta/3`:

```
cesta(X,Y,P) :- %Začátek hledání
  sekvence(L),
  findall(result(Pi,Di,RLi),cesta(X,Y,[],Pi,Di,L,RLi),Ress),
  ( %Nalezena cesta vedoucí k cíli - úspěch
    (member(result(Pi,Y,_),Ress), !, P = Pi);
    %Anebo žádné hledání nevyčerpalo kroky - fail
    (not(member(result(_,_,0),Ress)), !, fail)
  ).
```

Není to tak složité, jak to může vypadat. Predikát `findall` nám poskytne údaje o všech pokusech najít cestu (včetně těch neúspěšných – predikát `cesta` jsme upravili, aby byl vždy splněn).

My si vybereme, co nás zajímá (`Pi` – skutečně prošlá cesta, `Di` – kde skončila, `RLi` – kolik nevyužitých kroků zbylo), "zabalíme" to do složeného termu `result` a přidáme do seznamu `Ress`.

Nakonec zjistíme, zda platí nějaká z podmínek pro utnutí hledání:

a) Jsme v cíli.

b) Už se nemůžeme nikam dál dostat (ale nevyčerpány kroky).

```
cesta(X,X,_,[X],X,L,L) :- !. %Nalezeno řešení
cesta(X,Y,CP,[X|FP],D,L,RL) :- not(member(X,CP)),
  L > 0, L1 is L-1, %Kontrola a snížení zbýv. kroků
  prechod(X,Z), cesta(Z,Y,[X|CP],FP,D,L1,RL).
cesta(X,_,_,[X],X,L,L). %Slepá ulička
cesta(X,Y,P) :- sekvence(L), %Prohlubování hledání
  findall(result(Pi,Di,RLi),cesta(X,Y,[],Pi,Di,L,RLi),Ress),
  ( %Nalezena cesta vedoucí k cíli - úspěch
    (member(result(Pi,Y,_),Ress), !, P = Pi);
    %Anebo žádné hledání nevyčerpalo kroky - fail
    (not(member(result(_,_,0),Ress)), !, fail)
  ).
```

V této podobě ale program spotřebuje hodně paměti, když hromadí všechny varianty cesty. V podstatě je to jako hledání do šířky, kdy si pamatuje všechny možnosti pro další práci.

Můžeme se s tím tedy smířit, pokusit se program optimalizovat, aby ukládal méně dat, použít minulé řešení s prostým pevným omezením (ať už hloubkou, nebo třeba časem hledání), anebo navrhnout něco jiného. Rozhodnout se samozřejmě musíme podle okolností.

Než hledání cesty opustíme, podíváme se ještě na deklarativní cyklus v B-Prologu a s jeho pomocí na hledání do šířky; ale to už skutečně začíná být docela "ošklivé".

Deklarativní cyklus

Prolog k vytváření cyklů standardně používá rekurzi, ale vytvářet rekurzivní predikáty pro každý jednoduchý cyklus vede ke zbytečně komplikovaným a nepřehledným programům.

B-Prolog proto poskytuje konstrukci `foreach`, tedy cyklus, který jsme na začátku zmiňovali jako ukázku deklarativního přístupu v iterativních jazycích.

Tento cyklus lze použít při zpracování seznamů nebo lze zadat číselnou sekvenci, stejně jako u běžného for cyklu.

Základní forma cyklu vypadá následovně:

```
foreach(A1 in S1, …, An in Sn, LokalProm, Cil)
```

- `A1`–`An` jsou hodnoty ze seznamů `S1`–`Sn`. Můžeme tedy najednou procházet libovolné množství seznamů – Prolog projde všechny možné kombinace.
- Předposlední argument `LokalProm` se seznam lokálních proměnných (kromě `A1` až `An`, ty jsou lokální automaticky). Ostatní proměnné uvedené v „cíli“ jsou platné v rámci celého pravidla, ve kterém je cyklus použit. Nejde tedy o proměnné o nic globálnější, než ostatní proměnné mimo cyklus.

Příklad 49:

```
?- foreach(I in [1,2,3], write(I)). %Vypíše 123
```

Příklad 50:

```
?- foreach(I in [1,2,3], A = I).
   %Odpověď je "no", protože globální proměnná A
   %je navázána 1 a poté ztotožněna s číslem 2.
```

Při použití globální proměnné B-Prolog také vypíše varování, že se jedná o globální proměnnou – to je známka, že bychom je měli používat s rozmyslem.

Příklad 51:

```
?- foreach(I in [1,2,3], [A], A = I).
   %Odpověď je "yes", protože A je lokální
   %proměnná platná pouze v rámci jedné iterace.
```

Příklad 52: Pokud pracujeme s čísly, můžeme je také zadat jako číselnou sekvenci:

```
?- foreach(I in 1..3, format("~d ",I)). %Vypíše 1 2 3
```

Příklad 53: Můžeme také definovat krok pro změnu hodnoty:

```
?- foreach(I in 3..-1..1, format("~d ",I)). %Vypíše 3 2 1
```

Příklad 54: Lze pracovat i s desetinnými čísly:

```
?- foreach(F in 1.0..0.2..1.5, format("~1f ",F)).
                                    %Vypíše 1.0 1.2 1.4
```

Základní cyklus `foreach` se nehodí k agregaci výsledků. Proto je k dispozici i rozšířená varianta s akumulátorem:

```
foreach(A1 in S1, …, Akumulátor, LokálProm, Cíl)
```

nebo

```
foreach(A1 in S1, …, LokálProm, Akumulátor, Cíl)
```

Můžeme použít dvě formy akumulátoru

- `ac(Acc, PočátečníHodnota)` – Počáteční hodnota je uložena do akumulátoru před započtením iterace.
- `ac1(Acc, PosledníHodnota)` – Počáteční hodnota je v tomto případě volná proměnná.

Konkrétní způsob přidávání do zásobníku řídíme v "Cíli". Předcházející hodnota akumulátoru je zde značena *Acc^0*, nová hodnota akumulátoru *Acc^1* (s ohledem na to, jak si akumulátor pojmenujeme).

Příklad 55:

```
?- foreach(I in [1,2,3], ac(S,0), S^1 is S^0+I).
   S = 6.
```

Příklad 56:

```
?- foreach(I in [1,2,3], ac(L,[]), L^1=[I|L^0]).
   L = [3,2,1].
```

Příklad 57:

```
?- foreach(I in [1,2,3], ac1(L,[]), L^0=[I|L^1]).
   L = [1,2,3].
```

Na následujícím snímku je uvedena verze predikátu pro hledání cesty, která prohledává do šířky.

Nepracuje s jedním stavem (státem, ve kterém se nachází), ale s celým seznamem cest, které prošel. V podstatě v cyklu projde všechny "rozdělané cesty", najde predikátem `findall` všechna možná prodloužení každé z nich o jeden krok a z nich sestavuje nový seznam cest. Rekurze je použita jen pro iteraci, nedochází k žádnému větvení.

Program zde není podrobněji rozebrán. Pokud vás řešení zajímá, doporučuji stáhnout příslušný příklad a prozkoumat ho podrobně.

```
bfscesta(Y,CPs,FP) :- %Konec rekurze v cíli
  member(CP,CPs), CP=[Y|_], reverse(CP,FP), !.
bfscesta(Y,CPs,FP) :- foreach(
    CP in CPs, ac(NPs,[]), %Akum. Pro cesty s dalším krokem
    [X,Z,Ps,NPsi,_T], %Lokální proměnné
    ( CP = [X|_T], %X je současná pozice na cestě
      %Najdi všechna možná prodloužení cesty
      findall([Z|CP],(prechod(X,Z), not(member(Z,CP))),Ps),
      foreach(P in Ps,ac(NPsi,NPs^0),NPsi^1 = [P|NPsi^0]),
      NPs^1 = NPsi ) %Přidej všechna prodl. cesty do nových
  ), NPs \= [], bfscesta(Y,NPs,FP). %Další iterace
cesta(X,Y,P) :- bfscesta(Y,[[X]],P).
```

Toto řešení nelze považovat za vhodné pro Prolog – v podstatě úplně "zahazuje" normální rezoluční metodu implementovanou v Prologu a k řešení používá vlastní algoritmus, což je spíše imperativní přístup.

Pokud je obdobné řešení opravdu to jediné, které vás napadá, a přitom jste přesvědčeni, že dokážete myslet deklarativně, pak Prolog zřejmě není vhodný programovací jazyk pro řešení vašeho problému.

Více o rekurzi

Levá a pravá rekurze

Levá rekurze znamená, že rekurzivní volání se v těle pravidla nachází *nalevo* – podle vzoru `rekurze :- rekurze, term1, …, termN.`

Pravá rekurze znamená, že rekurzivní volání je v těle pravidla *napravo* – `rekurze :- term1, …, termN, rekurze.`

Levá rekurze je obvykle *nežádoucí*, protože s sebou přináší určité nevýhody.

Lze se setkat i s alternativním označením *koncová rekurze* (tail recursion) pro pravou rekurzi (tedy, když je rekurzivní volání v pravidle až jako poslední).

Kterákoliv jiná rekurze by pak byla označena jako *nekoncová rekurze* (non-tail recursion).

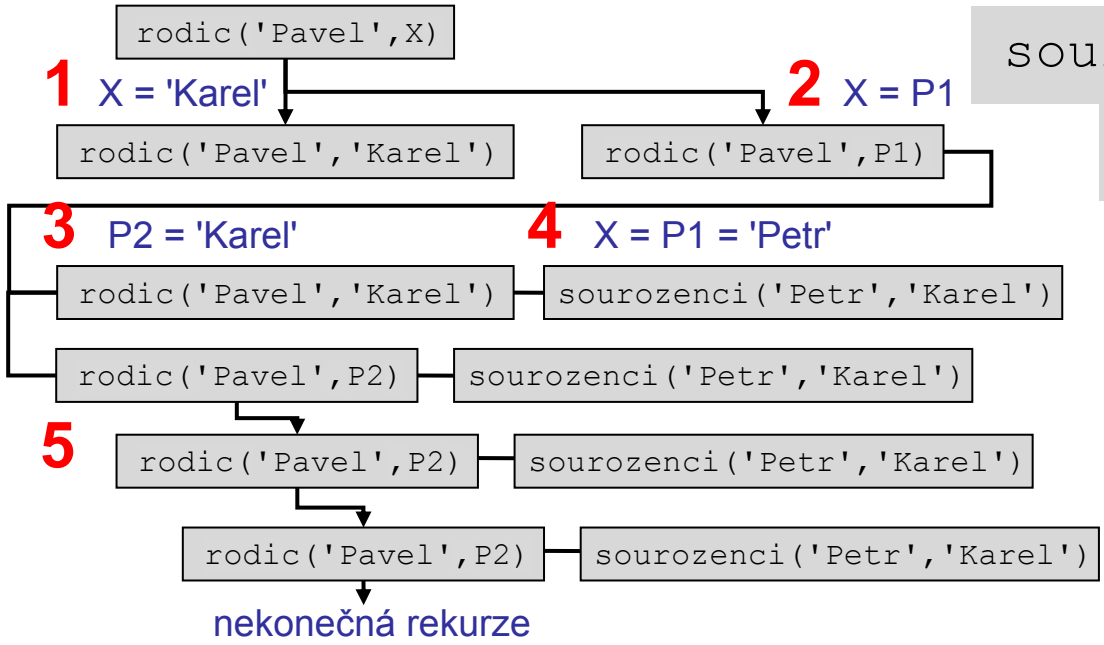
Někdy může levá rekurze způsobit zacyklení, které při použití pravé rekurze nevznikne.

Příklad 58: Mějme tento jednoduchý program s levou rekurzí:

```
sourozenci('Petr','Karel').
rodic('Pavel','Karel').
rodic(R,P1) :- rodic(R,P2), sourozenci(P1,P2).
```

Pokud zadáme jednoduchý dotaz na rodičovství, kde zadáme hodnoty obou proměnných (`?- rodic('Pavel','Petr')`), všechno je v pořádku.

Pokud ale chceme zjistit všechny Pavlovy děti dotazem `?-rodic('Pavel',X)`, Prolog nalezne dvě platné varianty `X = 'Karel'` a `X = 'Petr'`, načež pokračuje nekonečnou rekurzí ve snaze najít nějakou další variantu.

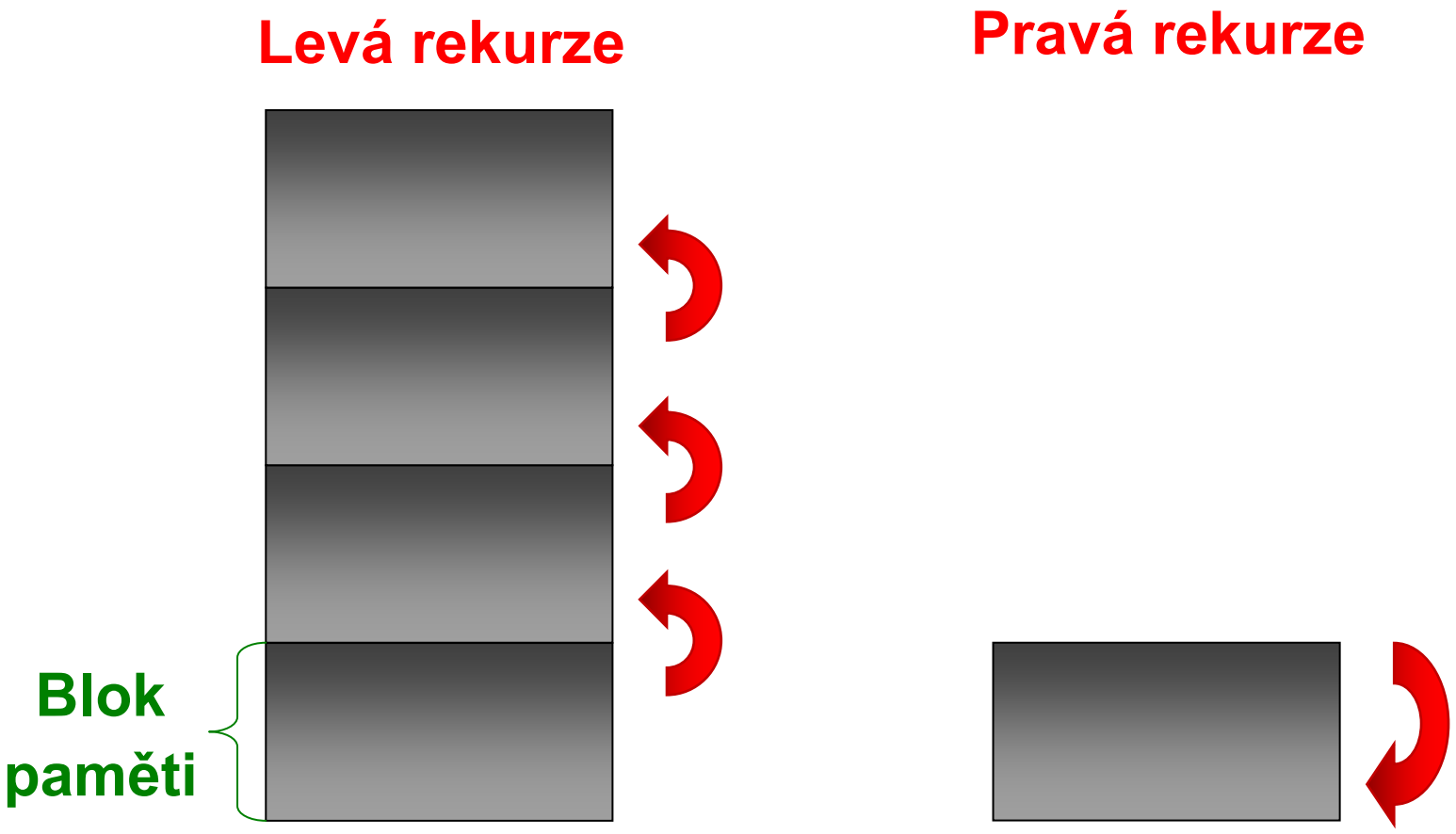



```
sourozenci('Petr','Karel').
rodic('Pavel','Karel').
rodic(R,P1) :-
    rodic(R,P2),
    sourozenci(P1,P2).
```

Hlavní výhodou pravé rekurze je **úspora paměti v zásobníku**. Tak jako jiné jazyky, založené na rekurzi, i Prolog musí rekurzi řešit efektivně.

Normálně je při každém rekurzivním volání nutno vyhradit v zásobníku další blok paměti pro nová data. Pokud však používáme *pravou rekurzi*, Prolog může při rekurzivním volání *znovu použít* již *alokovanou paměť*, kterou nebude dále potřebovat, a *vyhnout se* tak další alokaci a rychlému *vyčerpání paměti zásobníku*.

Ilustrace použití zásobníku při levé a pravé rekurzi



Technika akumulátoru

Příklad 59: Řekneme, že máme seznam číselných hodnot v počtu 1 milion a chceme získat jejich součet. Intuitivně, ve snaze napsat co nejjednodušší program, problém nejspíš vyřešíte následujícím způsobem:

```
sum([],0).
sum([H|T],S) :- sum(T,S0), S is H + S0.
```

Musíme použít levou rekurzi, protože potřebujeme znát dosavadní součet, abychom k němu mohli přičíst další hodnotu.

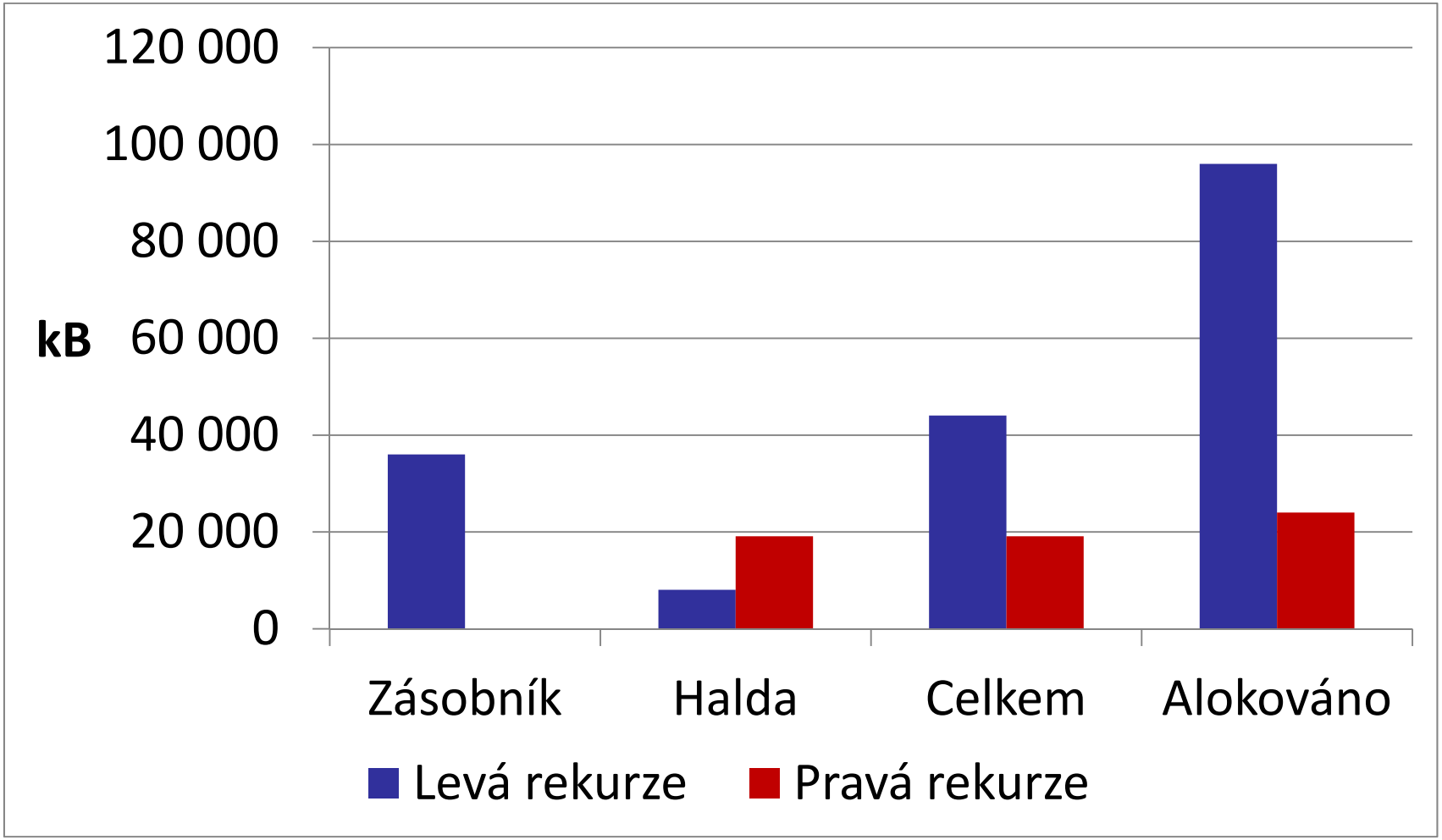
Pokud si však na začátku vyhradíme proměnnou pro součet, můžeme hodnoty ze seznamu přičítat rovnou a rekurzivní volání lze přesunout na konec:

```
sum([],S,S). %Hodnota z akumulátoru do výsledku
sum([H|T],Acc,S) :- Acc1 is Acc + H, sum(T,Acc1,S).
sum(L,S) :- sum(L,0,S). %Interface vypadá stejně
```

Prostřední argument nazýváme **akumulátor**. Slouží k nashromáždění (*akumulaci*) *výsledků* – v tomto případě jen jednoho součtu – který je při návratu z rekurze pouze *předán* "ven" *prostřednictvím arg3*.

Porovnání spotřeby paměti v době max. zanoření rekurze

	Levá rekurze	Pravá rekurze
Zásobník	36 002 540 bajtů	2 540 bajtů
Halda	8 055 196 bajtů	19 116 436 bajtů
Celkem Z+H	44 057 736 bajtů	19 118 976 bajtů
Alokováno Z+H	96 000 000 bajtů	24 000 000 bajtů


120 000
100 000
80 000
kB
60 000
40 000
20 000
0
Zásobník
Halda
Celkem
Alokováno
Levá rekurze
Pravá rekurze

Porovnání doby výpočtu sumy

Levá rekurze	Pravá rekurze
350−400 ms	250−300 ms

Programy sice obsahují víceméně stejné příkazy, tudíž výpočetní složitost by mohla být zhruba stejná. Když už něco, řešení s akumulátorem vypadá o něco složitější. Ve skutečnosti je však průměrná doba výpočtu kratší.

Základy profilace kódu

B-Prolog nabízí predikáty, které můžeme použít kdekoli v programu, pokud nás zajímá, jak moc náš program "mrhá" pamětí nebo co mu trvá tak dlouho.

Predikát `statistics/0` vypíše podrobné hlášení o využití paměti a další údaje o dosavadním běhu programu (počet spuštění garbage collectoru, počet spuštění backtrackingu).

Na následujícím snímku je ukázka výstupu:

```
Stack+Heap:     96,000,000 bytes
  Stack in use:36,002,540 bytes
  Heap in use:   8,055,160 bytes

Program:         8,000,000 bytes
  In use:        1,244,687 bytes
  Symbols:           5,917

Trail:           8,000,000 bytes
  In use:                40 bytes

Memory manager:
  GC:            Calls(8), Time(3469 ms)
  Expansions:    Stack+Heap(3), Program(0), Trail(0), Table(0)
```

Predikát `statistics(Klic,Hodnota)` můžeme použít v případě, že se nechceme probírat celým výpisem nebo chceme s výsledky dále pracovat.

- `statistics(runtime,[Spusteni,Volani])` – doba běhu programu od spuštění a od posledního volání;
- `statistics(gc,Pocet)` – počet spuštění gb collectoru;
- `statistics(backtracks,Pocet)` – počet volání backtrackingu;
- `statistics(gc_time,Ms)` – doba strávená během garbage collectoru.

- `statistics(OddilPameti,[Obsazeno,Volno])` – `OddilPameti` může být `program`, `heap`, `control`, `trail`, `table`.

Predikát `cputime/1` zjišťuje dobu uplynulou od posledního volání. Používá predikát `statistics/2`; jejich užití jsou tedy zaměnitelná z hlediska naměřeného času.

Predikát `time(C)` zjišťuje, jak dlouho *trvá* provést *cíl* C a na výstup vypíše hlášení oznamující naměřený čas.

Další rozšíření B-Prologu

Množinové operace

Množinu v programování obvykle chápeme jako strukturu, ve které musí být každý prvek unikátní − nesmí tedy obsahovat duplicitní hodnoty.

B-Prolog nemá speciální strukturu pro množiny, ale nabízí predikáty, které umožňují zacházet se seznamy jako s množinami.

Predikát `is_set/1` je splněn, pokud je argument seznam bez duplicitních termů.

Příklad 60:
```
?- A = [1,2,3,4], is_set(A). %Odpověď je yes
```

Příklad 61:
```
?- A = [1,2,3,2,3,4], is_set(A). %Odpověď je no
```

Příklad 62: Pokud seznam obsahuje vnořené seznamy, jsou jako termy uvažovány celé seznamy. V podseznamech tedy mohou být stejné termy, ale vnější seznam je přesto množina:

```
?- A = [[1,2,3],[2,3,4]], is_set(A). %Odpověď je yes
```

Příklad 63: Seznamy stejných termů jsou samy považovány za stejné termy:

```
?- L = [1,2,3], M = [1,2,3], N = [1,2,3],
   A = [L,M,N], is_set(A). %Odpověď je no
```

Predikát `eliminate_duplicate(Seznam,Mnozina)` převádí seznam na množinu. Ztotožňuje proměnnou `Mnozina` a seznam, který vznikl odstraněním všech duplikátů ze `Seznam`.

Predikát `intersection(A,B,C)` ztotožňuje množinu C s průnikem množin A a B.

Predikát `union(A,B,C)` ztotožňuje množinu C se sjednocením množin A a B.

Predikát `subset(Podmnozina,Mnozina)`

Predikát `subtract(A,B,C)` ztotožňuje množinu C s prvky, které obsahuje množina A a neobsahuje množina B.

Pole

Pole v B-Prologu jsou založena na strukturách, ale nelze je s nimi zaměňovat.

Predikát `new_array(A,Dimenze)`, respektive `new_array(A,[D1,D2,…,DN])` slouží k vytvoření *pole* A o N dimenzích délky D1, D2, …, DN. Pole tedy mají pevnou velikost. Všechny prvky jsou inicializovány jako volné proměnné.

Predikát `is_array(A)` je splněn, pokud A je pole.

Příklad 64:

```
?- new_array(A,[3,3]),
   is_array(A). %Odpověď je yes
```

Příklad 65:

```
?- A =
((A11,A12,A13),(A21,A22,A23),(A31,A32,A33)),
   is_array(A). %Odpověď je no
```

Predikát `arg(I,A,T)` ztotožňuje I-tý prvek pole A s termem T. Lze ho tedy použít pro přístup do pole. V případě vícerozměrného pole je třeba použít predikát vícekrát.

Příklad 66:

```
?- new_array(A,[3]),
   arg(1,A,a), arg(2,A,b), arg(3,A,c),
       %A = [](a,b,c)
   arg(2,A,X).
       %X = b
```

Příklad 67:

```
?- new_array(A,[1,1]),
   arg(1,A1,A2), arg(1,A2,a). %A = [](a)
```

Predikáty `a2_get(A,I,J,T)`, `a3_get(A,I,J,K,T)` umožňují snazší přístup do dvourozměrného a trojrozměrného pole.

Příklad 68: Pole lze inicializovat i tak, že ho přímo zapíšeme – takto lze definovat i nepravidelné pole:

```
?-  A = [](
    [](
      [](2,3,5),
      [](7,11,13,17),
      [](19,23)
    ),
    [](29,31,37)
   ).
```

Protože přístup k poli pomocí predikátů může být poněkud neohrabaný, umožňuje B-Prolog i zavedený přístup `A[I,J,K,…]`.

Tento zápis však lze použít pouze v aritmetických výrazech, pravdivostních výrazech, nebo společně se speciálním operátorem ztotožnění `@=`, který existuje k tomuto účelu.

Příklad 69:

```
?- new_array(A,[3]),
   A[1] @= a, A[2] @= b, A[3] @= c.
       %A = [](a,b,c)
```

Zápis `A[I,J,K,…]` je před kompilací programu nahrazen výrazem `A^[I,J,K,…]` – proto je nutno používat speciální operátor přiřazení, běžné přiřazení výrazy nevyhodnocuje, ale ztotožňuje je tak, jak jsou.

Zápis `A^length` je aritmetický výraz s hodnotou délky pole.

Příklad 70:

```
?- A = [](1,2,3,4,5,6,7,8,9),
   L is A^length. %L = 9
```

Operátor `^` lze v obou uvedených významech užít i pro seznamy.

Závěr

Nakonec **shrnutí** základních **faktů** o Prologu:

- Prolog je *slabě typovaný* jazyk.
- *Proměnné* jsou v Prologu *trvale navazovány*.
- *Návratovou hodnotou* je v Prologu vždy *pravdivostní hodnota*.
- Proměnné v Prologu nelze navázat na pravdivostní hodnotu.
- Prolog nativně hledá řešením prohledáváním do hloubky.

- Pravdivostní operátory pracují s termy, aritmetické operátory s jejich aritmetickými hodnotami.

- Prolog nachází *uplatnění* především v *rozpoznávání přirozeného jazyka* (nezaměňujte s převodem mluveného slova na text) a v *optimalizačních úlohách*.
- Prolog pravděpodobně nebude dobrou volbou pro úlohy zahrnující především rozsáhlé matematické výpočty.

Shrnutí Prologovské terminologie:

konstanta, proměnná, struktura, seznam = term

číslo, atom = konstanta

funktor(argumenty) = predikát

hlavička :- tělo. = pravidlo

hlavička. = fakt

hlavička :- true. = pravidlo fungující stejně jako fakt

?- tělo. = cíl

fakt, pravidlo, cíl = klauzule

běžné značení predikátů: funktor/arita, arita = počet argumentů

[hlavička|zbytek] = seznam; hlavička = termy oddělené čárkou; zbytek = seznam termů

"Toto je řetězec reprezentovaný seznamem znaků"

'Toto je term, který je dále nedělitelnou jednotkou'

Pokud se budete zajímat o B-Prolog do větších podrobností, doporučuji především oficiální dokumentaci (ke stažení na www.probp.com, v online podobě tamtéž).

Použitá literatura:

Neng-Fa Zhou: B-Prolog User's Manual, www.probp.com

Attila Csenki: Prolog Techniques

Attila Csenki: Applications of Prolog

Patrick Blackburn, Johan Bos, Kristina Striegnitz: Learn Prolog Now!

Karel Ježek: Logické programování – Prolog

Rudolf Kryl: Úvod do programovacího jazyka PROLOG, MFF UK Praha, 2000

Paul Brna: Prolog Programming – A First Course, 2001