PROGRAMOVÉ STRUKTURY: LOGICKÉ PROGRAMOVÁNÍ

Prolog, predikát, pravidlo, rekurze, princip rezoluce, unifikace, seznamy, princip návratu, typové úlohy

KIV/PGS

© Karel Ježek & Martin Zíma

Prolog

- PROgrammation en LOGique
  - programování v logice
- Program popisuje svět Prologu
  - nevyjadřuje tok řízení = vývojový diagram výpočtu
  - tvoří jej databáze faktů a pravidel (tzv. klauzulí)
- Známe tři základní typy klausulí
  - fakta `predikat(arg1, arg2, …, arg3).`
  - pravidla `hlava :- telo.`
  - cíl (dotaz) `?- predikat(arg1, arg2, …, argN).`
- Výpočet spustíme zadáním dotazu (cíle)
  - výsledkem jsou hodnoty proměnných dotazu

Elementy programu

- Term – jednoduchá datová struktura
  - struktura

    `cte(student(jan, novy), kniha(Autor, Nazev))`
  - jednoduchý term
    - proměnná
      - běžná proměnná `A  Rodic  _X`
      - anonymní proměnná `_`
    - konstanta
      - číslo `12  -5  -2.15  1e2`
      - atom `a  ty_a_ja  'Ty a ja'  b1  ,  ?`
- Struktury mohou být také tvořeny
  - operátory a operandy `2 + 2` nebo `+(2, 2)`

    `X < 2` nebo `<(X, 2)`
  - seznamy

© Karel Ježek & Martin Zíma

Strukturovaný term = struktura

- Je složen z
  - atomu
    - zde pojmenován jako funktor
  - argumentů
    - mají podobu termu
- Definuje predikát, který je dán
  - funktorem, tj. názvem predikátu
  - aritou, tj. počtem argumentů predikátu
    - atom je funktor s nulovou aritou
- Každý predikát je definován množinou
  - faktů
  - pravidel
- Predikáty nabývají hodnot
  - `true`
  - `false`

Implementace Prologu

- Komerční
  - Amzi! Prolog
  - LPA-Prolog
  - SICStus Prolog
  - Visual Prolog
- Volné nebo open-source
  - B-Prolog – jen pro nekomerční použití
  - GNU Prolog (též gprolog)
  - SWI-Prolog – ten budeme používat při výuce
  - XSB Prolog

Spuštění programu v Prologu

- Z nabídky *Start* spustíme SWI-Prolog
  - spustí se běhové prostředí listener
- Z nabídky *File/Edit* otevřeme soubor `dum.pro`
- Otevře se okno editoru s programem
- Vybráním nabídky *File/Consult* vybereme soubor `dum.pro`, aby se nahrál do prostředí listeneru
- Nyní v okně listeneru můžeme pokládat dotazy
- Na odpověď můžeme reagovat
  - stiskem *Enter* – odpověď přijímáme
  - stiskem *;* - odpověď odmítáme, dostaneme novou

 © Karel Ježek & Martin Zíma

Program dum.pro


```
ma(dum, dvere).
ma(dum, okno).
ma(dvere, klika).

rozbite(dvere).

opravit(X) :- ma(dum, X), rozbite(X).

?- ma(dum, dvere).   % true
?- ma(Neco, klika).  % Neco = dvere
?- ma(dum, Neco).    % Neco = dvere, Neco = okno
?- opravit(X).       % X = dvere
```


© Karel Ježek & Martin Zíma

Pravidla

- Základní tvar

```
cil(argumenty) :- formule z podcílů.
```

- Podcíle jsou oddělovány
  - čárkou – konjunkce
  - středníkem – disjunkce
  - skupiny podcílů lze závorkovat – změna priority
- Disjunkci můžeme také zapsat jako množinu pravidel
  - všechna tato pravidla mají
    - stejnou hlavu
    - různá těla
- Matematický zápis pravidla

```
p ← q1 & q2 & … & qn.
```

Program rodina.pro


```
rodic(eva, petr).        % Eva je rodicem Petra
rodic(eva, jan).         % Eva je rodicem Jana
rodic(jindrich, jan).    % Jindrich je rodicem Jana

zena(eva).               % Eva je zena

muz(petr).               % Petr je muz
muz(jan).                % Jan je muz
muz(jindrich).           % Jindrich je muz

matka(M, D):-rodic(M, D), zena(M).
otec(O, D):- rodic(O, D), muz(O).
```

Výklad pravidel

- Pravidlo

```
matka(M, D):-rodic(M, D), zena(M).
```

  - říká, že M je matkou dítěte D, jestliže M je rodičem D a zároveň M je žena.
- Pravidlo

```
otec(O, D):- rodic(O, D), muz(O).
```

  - říká, že O je otcem dítěte D, jestliže O je rodičem D a zároveň O je muž.
- Jak definovat rodinné vztahy? Zkuste vyložit pravidlo

```
deda(D, V) :- rodic(D, X), rodic(X, D).
```

 © Karel Ježek & Martin Zíma

Arita predikátu a anonymní proměnná

- Máme pravidla

```
otec(O, D):- rodic(O, D), muz(O).
otec(O):- rodic(O, _), muz(O).
```

- Predikát `otec(O, D)` je jiný než predikát `otec(O)`
  - první vyjadřuje binární relaci mezi dvěma osobami
    - arita predikátu je 2 (2 argumenty)
  - druhý je unární relací říkající pouze, že zda nějaká osoba je otcem
    - arita predikátu je 1 (1 argument)
    - symbol _ reprezentuje anonymní proměnnou
      - nechceme znát její hodnotu
      - každý její výskyt vyjadřuje jinou hodnotu (nesouvisí mezi sebou)
- Zkuste definovat např. predikát `syn/2`

Rekurze

- Ukázána nejlépe na příkladu
- Definice českých panovníků – přemyslovců:
  - přemyslovec je `premysl_orac`
  - přemyslovec je také syn přemyslovce
- Zapsáno v Prologu

```
premyslovec(premysl_orac).
premyslovec(P) :- syn(P, R), premyslovec(R).
```

- Alternativně to lze zapsat jedinou klausulí
  - přemyslovec je buď `premysl_orac`, nebo je to syn nějakého `R`, který je přemyslovcem

```
premyslovec(P) :-
   (P = premysl_orac); (syn(P, R), premyslovec(R)).
```

- Pokud nám odpověď nestačí, odmítneme ji `;` (středníkem)

 © Karel Ježek & Martin Zíma

Rekurze (2)

- Nabízí se tyto dotazy:
  1. jak vytvořit dotaz pro jména všech přemyslovců?
  2. jak definovat potomka po meči?
  3. jak definovat příbuznost osob X a Y po meči?
- Předpokládejme, že v databázi prologu máme fakta o potomcích tvaru `syn(nezamysl, kresomysl).`
  - dotazem `?- premyslovec(P).` získáme odmítáním (`;`) všechny hodnoty, kdy je tento predikát pravdivý, až dostaneme odpověď `no.`
- Stejný výsledek obdržíme pohodlněji dotazem

  `?- premyslovec(P), write(P), fail.`
  - váže konjunkcí tři podcíle
    - nalezení přemyslovce `P`
    - tisk `P`
    - vždy nesplnění predikátu způsobí hledání nového řešení

 © Karel Ježek & Martin Zíma

Rekurze (3)

- Definice potomka po meči
  - mužský potomek
- Rekurzivní predikát `potomek/2`
  - nerekurzivní pravidlo
    - poskytuje pouze přímé potomky
- Zápis v Prologu

```
potomek(X, Y) :- syn(X, Y).
potomek(X, Y) :- syn(X, Z), potomek(Z, Y).
```

- Dotaz

```
?- potomek(premysl_orac, P), write(P), fail.
```

  - vypíše všechny přemyslovce, kteří jsou přímými i nepřímými mužskými potomky přemyslovce `premysl_orac`.

© Karel Ježek & Martin Zíma

Rekurze (4)

- Příbuznost osob X a Y po meči
  - tyto osoby mají stejného mužského předka (předchůdce)
- Rekurzivní predikát `predek/2`
  - nerekurzivní pravidlo určuje otce dané osoby

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

- Pravidlo pro zjištění příbuznosti osob X a Y

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

- Dotaz

```
?- pribuzni(nezamysl, kresomysl).
```

  - odpoví `yes`, pokud jsou `nezamysl` a `kresomysl` příbuzní, jinak odpoví `no`.

Princip rezoluce

- Postup, jak Prolog hledá řešení dotazu
- Předpokládejme pravidla a, b tvaru

  $a \text{ :- } a_1, a_2, \ldots, a_n.$

  $b \text{ :- } b_1, b_2, \ldots, b_m.$
- Nechť b_i je a
  - pak **rezolucí** je

  $b \text{ :- } b_1, \ldots, b_{i-1}, a_1, \ldots, a_n, b_{i+1}, \ldots, b_m.$

- Když se při plnění cílů z těla pravidla b narazí na zpracování cíle b_i (tj. a), začnou se zpracovávat cíle těla pravidla a
  - jednotlivé cíle jsou predikáty, které Prolog porovnává s klauzulemi v jeho databázi
  - proces porovnávání, když dopadne úspěšně, se nazývá **unifikace**

Unifikace

- Porovnává-li se
  - proměnná s konstantou, naváže se na tuto konstantu
  - dvě volné (neinstalované) proměnné, stanou se synonymy
  - volná proměnná s termem, naváže se na tento term
  - termy, které nejsou volnými proměnnými, musí být pro úspěšné porovnání stejné
- Příklad unifikace v dotazu

```
?- X = Y, Y = a.
```

  - pak `X` a `Y` mají stejnou hodnotu `a`
  - operátorem `=` unifikujeme, nepřiřazujeme
    - pro přiřazení složí operátor `is`

 © Karel Ježek & Martin Zíma

Př. největší společný dělitel

- Největší společný dělitel
  - čísel `A` a `A` je `A`
  - čísel `A` a `B` je
    - největší společný dělitel čísel `A - B` a `B`
      - jestliže `A` je větší jak `B`
    - největší společný dělitel čísel `B - A` a `A`
      - jestliže `A` je menší než `B`
- Řešení vede na rekurentní výpočet
  - rekurzivní predikát `nsd/3`
  - program `nsd.pro`

Program nsd.pro


```
nsd(A, A, A).

nsd(A, B, NSD) :- A > B,
                  A1 is A - B,
                  nsd(A1, B, NSD).
nsd(A, B, NSD) :- A < B,
                  B1 is B - A,
                  nsd(B1, A, NSD).
?- nsd(16, 12, X).
```

Př. výpočet faktoriálu

- Faktoriál 0 je 1
- Faktoriál 1 je 1
- Faktoriál `N` je `F`
  - jestliže platí, že nějaké `M` má hodnotu `N - 1`
  - a současně faktoriál `M` je `G`
  - a současně `F` má hodnotu `G * N`
- Řešení opět vede na rekurentní výpočet
  - rekurzivní predikát `fakt/2`
  - program `fakt.pro`

Program fakt.pro


```
fakt(0, 1).
fakt(1, 1).

fakt(N, F) :- M is N - 1,
              fakt(M, G),
              F is G * N.
?- fakt(3, X).
```

- Proč při odmítnutí výsledku vznikne chyba přetečení zásobníku?

Zásady při plnění cílů

- Dotaz může být složen z několika cílů
- Při konjunkci cílů jsou cíle plněny postupně zleva
- Pro každý cíl je při jeho plnění prohledávána databáze od začátku
- Při úspěšném porovnání klausule s cílem je její místo v databázi označeno ukazatelem
  - každý z cílů má svůj ukazatel
- Při úspěšném porovnání cíle s hlavou pravidla, pokračuje výpočet plněním cílů zadaných v těle pravidla

 © Karel Ježek & Martin Zíma

Zásady při plnění cílů (2)

- Cíl je splněn
  - je-li úspěšně porovnán s faktem databáze
  - nebo s hlavou pravidla databáze
    - a jsou splněny podcíle těla pravidla
- Není-li během exekuce některý cíl splněn ani po prohlédnutí celé databáze
  - je aktivován **mechanismus návratu**
- Splněním jednotlivých cílů dotazu je splněn globální cíl
  - systém vypíše hodnoty proměnných zadaných v dotazu
- Zjistí-li se při výpočtu, že globální cíl nelze splnit
  - je výsledkem `no`.

Mechanismus návratu

- Exekuce se vrací k předchozímu splněnému cíli, zruší se instalace (navázání) proměnných a pokouší se opětovně splnit tento předchozí cíl prohledáváním databáze dále od ukazatele pro tento cíl
- Splní-li se opětovně tento cíl, pokračuje se plněním dalšího (předtím nesplněného) vpravo stojícího cíle
- Nesplní-li se předchozí cíl, vrací se výpočet ještě více zpět (na vlevo stojící cíl)

Shrnutí základních principů

- **Program** specifikujeme množinou klausulí
  - klausule mají podobu faktů, pravidel a dotazu
  - Prolog zná pouze to, co je definované programem
- **Fakt** je jméno relace a argumenty (objekty) v daném uspořádání
  - uspořádání je důležité
- **Pravidlo** vyjadřuje vztahy, které platí, jsou-li splněny podmínky z těla (cíle)
  - hlavu tvoří vždy jen jeden predikát
- **Dotaz** může tvořit jeden nebo více cílů
  - cíle mohou obsahovat proměnné i konstanty
  - Prolog najde tolik odpovědí, kolik je požadováno (pokud existují)

Shrnutí základních principů (2)

- **Proměnná** je v klausuli obecně kvantifikována
  - její platnost je omezena na klausuli
- **Definice predikátu** je posloupnost klausulí pro jednu relaci
  - predikát může určovat vztah, databázovou relaci, typ, vlastnost nebo funkci
  - jméno predikátu musí být atomem.
- **Plnění cíle** provádí Prolog pro nový cíl prohledáváním databáze od začátku
  - při opakovaném pokusu prohledáváním od naposledy použité klausule.

Shrnutí základních principů (3)

- **Rekurzivní definice predikátu** musí obsahovat ukončovací podmínku
- **Typ termu** je rozpoznatelný syntaxí
  - atomy a čísla jsou konstanty
  - atomy a proměnné jsou jednoduchými termy
  - anonymní proměnná představuje neznámý objekt, který nás nezajímá
  - struktury jsou složené typy dat
  - pravidlo je strukturou s funktorem :–
- **Funktor** je určen jménem a aritou

Unifikace termů podrobněji

- Dva termy jsou úspěšně porovnány (lze říci, že si jsou podobné), pokud:
  - jsou totožné nebo
  - proměnné v termech lze navázat na objekty tak, že po navázání proměnných jsou tyto termy totožné
- Ukázkové příklady:
  - termy `datum(D, M, 2016)` a `datum(X, 12, R)`
    - jsou unifikovatelné
      - `D` je `X`, `M` je `12` a `2016` je `R`
  - termy `datum(D, M, 2016)` a `datum(X, 12, 2004)`
    - nejsou unifikovatelné

Unifikace termů podrobněji (2)

- Ukázkové příklady (pokračování):
  - termy `bod(X, Y, Z)` a `datum(D, M, 2016)`
    - nejsou unifikovatelné
  - termy `datum(D, M, 2016)` a `datum(X, 12)`
    - nejsou unifikovatelné
- Prolog vybere vždy nejobecnější možnost porovnání
  - porovnání vyjadřuje operátor =

Unifikace termů podrobněji (3)

- Při porovnání proměnné se strukturou je třeba vyloučit případ, kdy se tato proměnná vyskytuje ve výrazu
- Příklad:
  - `?- X = f(X).`
  - neproveditelné porovnání
  - způsobí navázání
    - `X` na `f(X)` ⟶ na `f(X)` ⟶ na `f(X)` ...
  - způsobí *stack overflow*
    - přetečení zásobníku

 © Karel Ježek & Martin Zíma

Aritmetické výrazy jsou termy

- Unifikace
  - symbol + je názvem struktury

```
?- X = +(2, 2).
X = 2 + 2
?- X = 2 + 2.
X = 2 + 2
```

- Vyhodnocení
  - výraz vyhodnotíme predikátem `is`

```
?- X is +(2, 2).
X = 4
?- X is 2 + 2.
X = 4
```

Seznamy

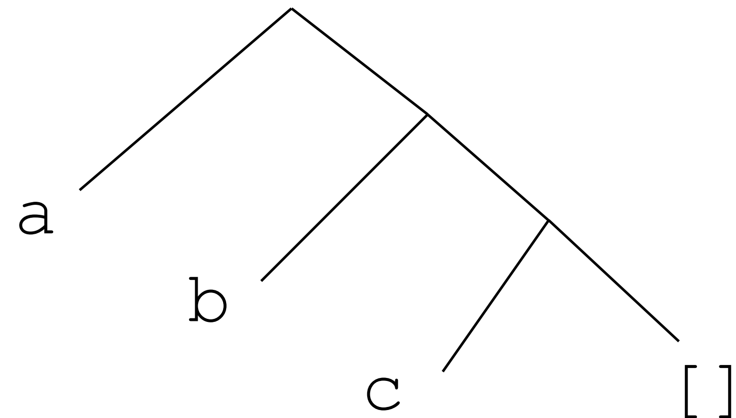
- Rekurzivní datová struktura tvaru
  - `[e1, e2, …, en]`
  - `ei` jsou elementy seznamu
- Elementy seznamů jsou často opět seznamy
- Symbolem | (svislítko) lze seznam rozdělit na
  - prvý element (také se nazývá hlava)
  - a zbytek seznamu

  `[ Hlava | Zbytek ]`
- Prázdný seznam označíme `[ ]`

 © Karel Ježek & Martin Zíma

Příklady stejného seznamu

- Ukázkový seznam
  - `[ a, b, c ]`
  - `[ a | [ b, c ] ]`
  - `[ a, b | [c] ]`
  - `[ a, b, c | [] ]`
- Grafická reprezentace
  - zbytek seznamu je opět seznam
    - případně prázdný seznam



Porovnání seznamů


```
?- [X] = [a, b, c, d].
no.

?- [X | Y] = [a, b, c, d].
X = a
Y = [b, c, d]

?- [X, Y | Z] = [a, b, c, d].
X = a
Y = b
Z = [c, d]
```

```
?- [X] = [[a, b, c, d]].
X = [a, b, c, d]

?- [X | Y] = [[a, [b, c]], d].
X = [a, [b, c]]
Y = d

?- [X | Y] = [].
no.

?- [X | Y] = [d].
X = d
Y = []
```

Test existence prvku v seznamu

- Zjištění, zda v nejvyšší úrovni seznamu existuje prvek
  - predikát `member(Prvek, Seznam)`
  - slovně: `member` platí, jestliže:
    - je-li prvek na začátku seznamu
    - jinak `member` platí, pokud prvek je ve zbytku seznamu

```
member(X, [X, _]).
member(X, [_ | Y]) :- member(X, Y).
```

- Standardní predikát, nemusíme ho definovat

Příklady použití member


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

?- member(a, [b, c, [d, a]]).
no
```

Nalezení posledního prvku seznamu

- Predikát `delete(Seznam, Prvek)`
  - slovně:
    - je-li v seznamu jen jeden prvek, tak je tím posledním
    - jinak je to poslední prvek zbytku seznamu

```
last([X], X).
last([_ | T], X) :- last(T, X).
```

- Ukázka použití

```
?- last([a, [b, c]], X).
X = [b, c]
```

 © Karel Ježek & Martin Zíma

Odstranění prvku ze seznamu

- Predikát `delete(PuvodniS, VyslednyS, Prvek)`

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

- Slovně:
  - je-li prvek prvním v seznamu, je výsledkem zbytek
  - jinak je výsledkem seznam, se stejným prvním prvkem, ale se zbytkem, v němž je vynechán uvažovaný prvek
- Ukázka použití

```
?- delete([a, b, a], L, a).
L = [a, b];
L = [b, a];
no
```

Připojení seznamu k seznamu

- **Predikát** `append(Seznam1, Seznam2, Vysledek)`

```
append([], X, X).
append([A | B], X, [A | C]) :- append(B, X, C).
```

- Slovně:
  - když přidáme k prázdnému seznamu seznam `X`, výsledkem bude seznam `X`
  - když přidáme k seznamu (jehož prvý prvek je `A` a jeho zbytek je `B`) seznam `X`, bude výsledný seznam mít prvý prvek `A` a jeho zbytkem bude seznam `C`, který vznikne spojením seznamů `B` a `X`
- Ukázka použití

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

Pozoruhodnosti append

- Zeptáme se, jaké dva seznamy musíme spojit, aby vznikl seznam
  - `[a, b, c]`
- Prolog nám je najde a pokud budeme chtít, najde nám všechny možnosti
- Námi definovaný `append` je obousměrný
  - lze zaměnit co je vstup a co výstup

```
?- append(X, Y, [a, b, c]).
X = []
Y = [a, b, c];

X = [a]
Y = [b, c];

X = [a, b]
Y = [c];

X = [a, b, c]
Y = [];
no
```

Vázané a volné argumenty

- Argumenty funkcí (tj. prologovské proměnné) mohou být
  - bound (vázané) – zkráceně `b`
  - free (volné) – zkráceně `f`
- Příklady

```
bbb        ?- append([a, b], [c], [a,b,c]).
           yes
bbf        ?- append([a, b], [c], S3).
           S3 = [a, b, c];
           no
```

Vázané a volné argumenty (2)

- Příklady (pokračování)

```
bfb        ?- append([a, b], S2, [a, b, c]).
           S2 = [c];
           no
bff        ?- append([a, b], S2, S3).
           S2 = H159
           S3 = [a, b | H159];
           no
fbb        ?- append(S1, [c], [a, b, c]).
           S1 = [a, b];
           no
```

 © Karel Ježek & Martin Zíma

Vázané a volné argumenty (3)

- Příklady (pokračování)

```
fbf          ?- append(S1, [c], S3).
             S1 = []
             S3 = [c];
             S1 = [H277]
             S3 = [H277, c];
             atd.
ffb          ?- append(S1, S2, [a, b]).
             S1 = []
             S2 = [a, b];
             S1 = [a]
             S2 = [b];
             S1 = [a, b]
             S2 = [];
             no
```

Vázané a volné argumenty (4)

- Příklady (pokračování)

```
fff          ?- append(S1, S2, S3).
             S1 = []
             S2 = H253
             S3 = H253;
             S1 = [H323]
             S2 = H253
             S3 = [H323 | H253];
             S1 = [H323, H349]
             S2 = H253
             S3 = [H323, H349 | H253];
             atd.
```

Vícesměrnost dalších predikátů

- Predikát `member`

```
?- member(X, [a, [b, c], d]).
X = a;
X = [b, c];
X = d;
no
```

- Predikát `delete`

```
?- delete(X, [a, b], c).
X = [c, a, b];
X = [a, c, b];
X = [a, b, c];
no
```

 © Karel Ježek & Martin Zíma

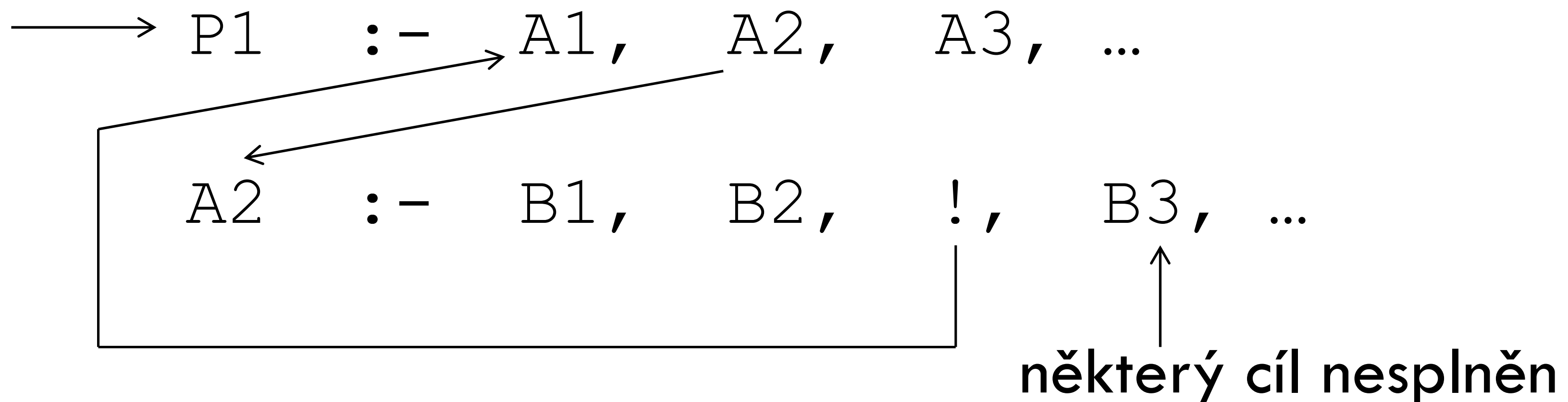
Řetězce

- Seznamy znaků jsou řetězce
  - uzavírají se do řetězcových závorek – uvozovky

```
?- [X, Y | Z] = "abcd".
X = 97
Y = 98
Z = [99, 100];
yes
```

Ovlivnění mechanismu návratu

- Mechanismus navracení lze ovlivnit tzv. predikátem řezu
  - označovaný jako !
  - predikát řezu je vždy splněn
- způsobí při nesplnění některého cíle za ním přeskok až na nové plnění cíle A1, tj. nesplnění cíle A2



Predikát řezu

- Použijeme jej, když chceme zabránit hledání jiné alternativy
- Odřízne další provádění cílů z pravidla, ve kterém je uveden
- Je bezprostředně splnitelným cílem
- Projeví se pouze tehdy, když má přes něj dojít k návratu
- Změní mechanismus návratu tím, že znepřístupní ukazatele vlevo od něj ležících cílů
  - přesune je na konec databáze

Příklad použití predikátu řezu

- Modifikace programu `fakt.pro`
- Výskyt predikátu řezu zabrání výpočtu faktoriálu pro záporný argument při odmítnutí výsledku

```
fakt(N, 1) :- N = 0, !.

fakt(N, F) :- M is N - 1,
              fakt(M, G),
              F is G * N.
```

Hanojské věže

- Jsou dány tři trny A, B a C
- Na trnu A je navléknuto N kotoučů s různými průměry tak, že vytváří tvar pagody (menší je nad větším)
- Přemístěte kotouče z trnu A na trn C s využitím trnu B jako pomocného trnu tak, aby stále platilo, že nikdy není položen větší kotouč na menší
- Přemisťování probíhá vždy po jednom kotouči
  - nelze přemisťovat dva a více kotoučů naráz

Ukázka Hanojských věží

- Ukázkové video řeší věže o velikosti 5
- https://www.youtube.com/watch?v=EkjucFnNZoA

Program hanoi.pro


```
veze :- repeat,
        write('Pocet kruhu (nebo zaporne pro ukonceni): '),
        read(X), hanoi(X), fail.

hanoi(N) :- N < 0, halt.
hanoi(N) :-  presun(N, levy, stred, pravy), !.

presun(0, _, _, _) :- !.
presun(N, A, B, C) :- M is N - 1,
                      presun(M, A, C, B), inform(A, B),
                      presun(M, C, B, A).

inform(A, B) :- write([presun, disk, z, A, na, B]), nl.
```

Standardní predikáty

- Zahrnují skupiny predikátů
  - I/O operace
  - řídící predikáty a testy
  - predikáty pro práci s aritmetickými výrazy
  - predikáty pro manipulaci s databází
  - predikáty pro práci se strukturami
- Jazyk Prolog je tvořen
  - resolučním mechanismem a
  - skupinou standardních predikátů
    - z nichž si většinu ukážeme
    - ale nemusíte si všechny pamatovat

I/O operace

- Vždy splnitelné
  - `write(X)`
    - zápis termu do výstupního proudu
  - `nl`
    - odřádkování
  - `tab(X)`
    - výstup X mezer do výstupního proudu
- Jednou splnitelné
  - `read(X)`
    - čtení termu ze vstupního proudu

 © Karel Ježek & Martin Zíma

Řídící predikáty a testy

Predikát	Popis
`true`	vždy splněný cíl
`fail`	vždy nesplněný cíl
`var(X)`	splněno, je-li `X` proměnnou
`nonvar(X)`	splněno, neplatí-li `var(X)`
`atom(X)`	splněno, je-li `X` instalováno na atom
`integer(X)`	splněno, je-li `X` instalováno na integer
`atomic(X)`	splněno, je-li `X` instalováno na atom nebo integer
`not(X)`	`X` musí být interpretovatelné jako cíl. Uspěje, pokud `X` není splněn.
`call(X)`	`X` musí být interpretovatelné jako cíl. Uspěje, pokud `X` je splněn
`halt`	ukončí výpočet

 © Karel Ježek & Martin Zíma

Řídící predikáty a testy (2)

Predikát	Popis
`X = Y`	pokus o porovnání X s Y
`X \= Y`	opak X = Y
`X == Y`	striktní rovnost
`X \== Y`	úspěšně splněn, neplatí-li X == Y
`!`	změna mechanismu návratu
`repeat`	nekonečněkrát splnitelný cíl
`X, Y`	konjunkce cílů
`X; Y`	disjunkce cílů

 © Karel Ježek & Martin Zíma

Práce s aritmetickými výrazy

Predikát	Popis
`X is E`	`E` musí být aritmetický výraz, který se vyhodnotí a porovná s `X`
`E1 + E2`	při instalovaných argumentech (podobně `-, *, /, mod`)
`E1 > E2`	při instalovaných argumentech (podobně `>=, <, =<, \=, =`)
`E1 =:= E2`	uspěje, jsou-li hodnoty `E1` a `E2` rovny
`E1 =\= E2`	uspěje, nejsou-li hodnoty `E1` a `E2` rovny

 © Karel Ježek & Martin Zíma

Manipulace s databází a klausulemi

Predikát	Popis
`listing(X)`	výpis všech klausulí, na jejichž jméno je X instalováno
`listing`	výpis celého programu
`clause(X, Y)`	porovnání X a Y s hlavou a s tělem klausule
`asserta(X)`	přidání klausule instalované na X na začátek databáze
`assertz(X)`	přidání klausule instalované na X na konec databáze
`retract(X)`	odstranění prvního výskytu klausule X z databáze
`findall(X, Y, Z)`	všechny výskyty termu X v databázi, které splňují cíl Y vloženy do seznamu Z

Závody gymnastek

- Typická úloha ze sobotní přílohy novin
- Určete jména vítězek disciplín z těchto informací:
  1. Dvořáková ani Sobotková nevyhrály přeskok ani bradla.
  2. V přeskoku nezvítězila Věra ani Ludmila.
  3. Sobotková se nejmenuje ani Věra ani Jiřina a kamarádí s Bečkovou.
  4. Junková není ani Monika ani Jiřina.
  5. Věra nevyhrála na bradlech, Monika nevyhrála v prostných.
  6. Jednou z disciplín byla kladina.
  7. V každé ze čtyř disciplín zvítězila jiná závodnice.

 © Karel Ježek & Martin Zíma

Řešení úlohy

- Úlohám tohoto typu se říká "zebra"
  - jedenou z prvních přišel Albert Einstein
  - https://cs.wikipedia.org/wiki/Einsteinova_h%C3%A1danka

- Klasické řešení
  - program `zebra.pro`
- Řešení pomocí permutací
  - program `permutace.pro`

 © Karel Ježek & Martin Zíma