

## Vestavěné predikáty (pokračování)

## Testování typu termu

<code>var(X)</code>	X je volná proměnná
<code>nonvar(X)</code>	X není proměnná
<code>atom(X)</code>	X je atom ( <code>pavel</code> , <code>'Pavel Novák'</code> , <code>&lt;--&gt;</code> )
<code>integer(X)</code>	X je integer
<code>float(X)</code>	X je float
<code>atomic(X)</code>	X je atom nebo číslo
<code>compound(X)</code>	X je struktura

## Určení počtu výskytů prvku v seznamu

## Konstrukce a dekompozice atomu

```
count( X, S, N ) :- count( X, S, 0, N ).
```

```
count( _, [], N, N ).
```

```
count( X, [X|S], N0, N ) :- !, N1 is N0 + 1, count( X, S, N1, N ).
```

```
count( X, [_|S], N0, N ) :- count( X, S, N0, N ).
```

```
:-? count( a, [a,b,a,a], N )      :-? count( a, [a,b,X,Y], N ).
```

```
N=3                               N=3
```

```
count( _, [], N, N ).
```

```
count( X, [Y|S], N0, N ) :- nonvar(Y), X = Y, !,  
                           N1 is N0 + 1, count( X, S, N1, N ).
```

```
count( X, [_|S], N0, N ) :- count( X, S, N0, N ).
```

### Atom (opakování)

- řetězce písmen, čísel, „\_“ začínající malým písmenem: `pavel`, `pavel_novak`, `x2`, `x4_34`

- řetězce speciálních znaků: `+`, `<->`, `===>`

- řetězce v apostrofech: `'Pavel'`, `Pavel Novák'`, `'prší'`, `'ano'`

```
?- 'ano'=A.      A = ano
```

### Řetězec znaků v uvozovkách

- př. `"ano"`, `"Pavel"`

```
?- A="Pavel".
```

```
A = [80,97,118,101,108]
```

```
?- A="ano".
```

```
A=[97,110,111]
```

- př. použití: konstrukce a dekompozice atomu na znaky, vstup a výstup do souboru

### Konstrukce atomu ze znaků, rozložení atomu na znaky

```
name( Atom, SeznamASCIIKodu )
```

```
name( ano, [97,110,111] )
```

```
name( ano, "ano" )
```

## Konstrukce a dekompozice termu

- Konstrukce a dekompozice termu

Term =.. [ Funktor | SeznamArgumentu ]

a(9,e) =.. [a,9,e]

Call =.. [ Funktor | SeznamArgumentu ], call( Call )

atom =.. X ⇒ X = [atom]

- Pokud chci znát pouze funktor nebo některé argumenty, pak je efektivnější:

```
functor( Term, Funktor, Arita )      functor( a(9,e), a, 2 )
                                     functor( atom, atom, 0)   functor( 1, 1, 0)
arg( N, Term, Argument )            arg( 2, a(9,e), e)
```

## Příklad: dekompozice termu I.

- count\_term( Integer, Term, N ) určí počet výskytů celého čísla v termu

▪ ?- count\_term( 1, a(1,2,b(x,z(a,b,1))),Y), N ). N=2

▪ count\_term( X, T, N ) :- count\_term( X, T, 0, N ).

count\_term( X, T, N0, N ) :- integer(T), X = T, !, N is N0 + 1.

count\_term( \_, T, N, N ) :- atomic(T), !.

count\_term( \_, T, N, N ) :- var(T), !.

count\_term( X, T, N0, N ) :- T =.. [ \_ | Argumenty ],  
count\_arg( X, Argumenty, N0, N ).

count\_arg( \_, [], N, N ).

count\_arg( X, [ H | T ], N0, N ) :- count\_term( X, H, 0, N1 ),  
N2 is N0 + N1,  
count\_arg( X, T, N2, N ).

▪ ?- count\_term( 1, [a,2,[b,c],[d,[e,f],Y]], N ).

count\_term( X, T, N0, N ) :- T = [\_|\_], !, count\_arg( X, T, N0, N ).

klauzuli přidáme **před** poslední klauzuli count\_term/4

## Rekurzivní rozklad termu

- Term je proměnná (var/1), atom nebo číslo (atomic/1) ⇒ konec rozkladu
- Term je seznam ([\_|\_]) ⇒  
procházení seznamu a rozklad každého prvku seznamu
- Term je složený (=./2, functor/3) ⇒  
procházení seznamu argumentů a rozklad každého argumentu
- Příklad: ground/1 uspěje, pokud v termu nejsou proměnné; jinak neuspěje

ground(Term) :- atomic(Term), !.

ground(Term) :- var(Term), !, fail.

ground([H|T]) :- !, ground(H), ground(T).

ground(Term) :- Term =.. [ \_Funktor | Argumenty ],  
ground( Argumenty ).

?- ground(s(2,[a(1,3),b,c],X)).

?- ground(s(2,[a(1,3),b,c])).

no

yes

## Cvičení: dekompozice termu

- Napište predikát substitute( Podterm, Term, Podterm1, Term1), který nahradí všechny výskyty Podterm v Term termem Podterm1 a výsledek vrátí v Term1
- Předpokládejte, že Term a Podterm bez proměnných
- ?- substitute( sin(x), 2\*sin(x)\*f(sin(x)), t, F ). F=2\*t\*f(t)

# Technika a styl programování v Prologu

- Styl programování v Prologu
  - některá pravidla správného stylu
  - správný vs. špatný styl
  - komentáře
- Ladění
- Efektivita

## Technika a styl programování v Prologu

### Styl programování v Prologu I.

- Cílem stylistických konvencí je
  - redukce nebezpečí programovacích chyb
  - psaní čitelných a srozumitelných programů, které se dobře ladí a modifikují
- Některá pravidla správného stylu
  - krátké klauzule
  - krátké procedury; dlouhé procedury pouze s uniformní strukturou (tabulka)
  - klauzule se základními (hraničními) případy psát před rekurzivními klauzulemi
  - vhodná jména procedur a proměnných
    - nepoužívat seznamy ([...]) nebo závorky ({...}, (...)) pro termy pevné arity
  - vstupní argumenty psát před výstupními
  - **struktura programu – jednotné konvence** v rámci celého programu, např.
    - mezery, prázdné řádky, odsazení
    - klauzule stejné procedury na jednom místě; prázdné řádky mezi klauzulemi; každý cíl na zvláštním řádku

### Správný styl programování

- konstrukce setříděného seznamu Seznam3 ze setříděných seznamů Seznam1, Seznam2: `merge( Seznam1, Seznam2, Seznam3 )`
- `merge( [2,4,7], [1,3,4,8], [1,2,3,4,4,7,8] )`
- `merge( [], Seznam, Seznam ) :-`  
`!.` % prevence redundantních řešení
- `merge( Seznam, [], Seznam ).`
- `merge( [X|Telo1], [Y|Telo2], [X|Telo3] ) :-`  
`X < Y, !,`  
`merge( Telo1, [Y|Telo2], Telo3 ).`
- `merge( Seznam1, [Y|Telo2], [Y|Telo3] ) :-`  
`merge( Seznam1, Telo2, Telo3 ).`

## Špatný styl programování

```
merge( S1, S2, S3 ) :-
  S1 = [], !, S3 = S2;           % první seznam je prázdný
  S2 = [], !, S3 = S1;           % druhý seznam je prázdný
  S1 = [X|T1],
  S2 = [Y|T2],
  ( X < Y, !,
    Z = X,                       % Z je hlava seznamu S3
    merge( T1, S2, T3 );
    Z = Y,
    merge( S1, T2, T3 ) ),
  S3 = [ Z | T3 ].
```

## Dokumentace a komentáře

- co program dělá, jak ho používat (jaký cíl spustit a jaké jsou očekávané výsledky), příklad použití
- které predikáty jsou hlavní (*top-level*)
- jak jsou hlavní koncepty (objekty) reprezentovány
- doba výpočtu a paměťové nároky
- jaké jsou limity programu
- zda jsou použity nějaké speciální rysy závislé na systému
- jaký je význam predikátů v programu, jaké jsou jejich argumenty, které jsou vstupní a které výstupní (pokud víme)
  - vstupní argumenty „+“, výstupní „-“      `merge( +Seznam1, +Seznam2, -Seznam3 )`
  - `JmenoPredikatu/Arit`      `merge/3`
- algoritmické a implementační podrobnosti

## Styl programování v Prologu II.

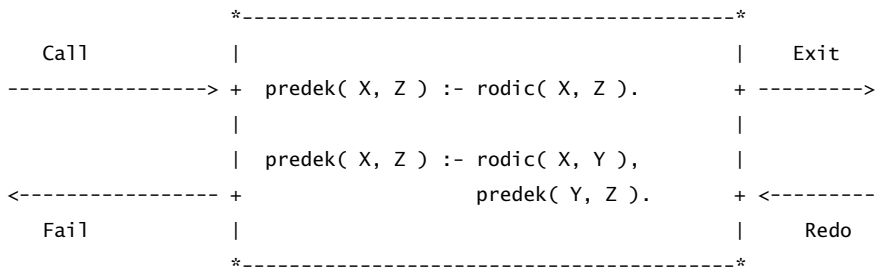
- **Středník „;“** může způsobit nesrozumitelnost klauzule
  - nedávat středník na konec řádku, používat závorky
  - v některých případech: rozdělení klauzule se středníkem do více klauzul
- **Opatrné používání operátoru řezu**
  - preferovat použití zeleného řezu (nemění deklarativní sémantiku)
  - červený řez používat v jasně definovaných konstruktech
    - negace: `P, !, fail; true`      `\+ P`
    - alternativy: `Podminka, !, Ci11 ; Ci12`      `Podminka -> Ci11 ; Ci12`
- **Opatrné používání negace „\+“**
  - negace jako neúspěch: negace není ekvivalentní negaci v matematické logice
- **Pozor na `assert` a `retract`**: snižují transparentnost chování programu

## Ladění

- Přepínače na trasování: `trace/0`, `notrace/0`
- Trasování specifického predikátu: `spy/1`, `nosp/1`
  - `spy( merge/3 )`
- `debug/0`, `nodebug/0`: pro trasování pouze predikátů zadaných `spy/1`
- Libovolná část programu může být spuštěna zadáním vhodného dotazu: **trasování cíle**
  - vstupní informace: jméno predikátu, hodnoty argumentů při volání
  - výstupní informace
    - při úspěchu hodnoty argumentů splňující cíl
    - při neúspěchu indikace chyby
  - nové vyvolání přes `;`: stejný cíl je volán při backtrackingu

## Krabičkový (4-branový) model

- Vizualizace řídicího toku (backtrackingu) na úrovni predikátu
  - Call: volání cíle
  - Exit: úspěšné ukončení volání cíle
  - Fail: volání cíle neuspělo
  - Redo: jeden z následujících cílů neuspěl a systém backtrackuje, aby našel alternativy k předchozímu řešení



## Efektivita

- Čas výpočtu, paměťové nároky, a také časové nároky na vývoj programu
  - u Prologu můžeme častěji narazit na problémy s časem výpočtu a paměti
  - Prologovské aplikace redukují čas na vývoj
  - vhodnost pro symbolické, nenumerické výpočty se strukturovanými objekty a relacemi mezi nimi
- Pro zvýšení efektivity je nutno se zabývat **procedurálními aspekty**
  - zlepšení efektivity při prohledávání**
    - odstranění zbytečného backtrackingu
    - zrušení provádění zbytečných alternativ co nejdříve
  - návrh **vhodnějších datových struktur**, které umožní efektivnější operace s objekty

## Příklad: trasování

```

a(X) :- nonvar(X).
a(X) :- c(X).
a(X) :- d(X).
c(1).
d(2).

*-----*
Call |                               | Exit
-----> + a(X) :- nonvar(X). | ----->
          | a(X) :- c(X).      |
<----- + a(X) :- d(X).      + <-----
Fail  |                               | Redo
*-----*

| ?- a(X).
      1      1 Call: a(_463) ?
      2      2 Call: nonvar(_463) ?
      2      2 Fail: nonvar(_463) ?
      3      2 Call: c(_463) ?
      3      2 Exit: c(1) ?
      ?      1      1 Exit: a(1) ?
X = 1 ? ;
      1      1 Redo: a(1) ?
      4      2 Call: d(_463) ?
      4      2 Exit: d(2) ?
      1      1 Exit: a(2) ?
X = 2 ? ;
no
% trace
| ?-
    
```

## Zlepšení efektivity: základní techniky

- Optimalizace posledního volání (LCO) a akumulátory**
- Rozdílové seznamy** při spojování seznamů
- Caching:** uložení vypočítaných výsledků do programové databáze
- Indexace** podle prvního argumentu
  - např. v SICStus Prologu
  - při volání predikátu s prvním nainstanciováním argumentem se používá hašovací tabulka zpřístupňující pouze odpovídající klauzule
  - zamestnanec( Prijmeni, KrestniJmeno, Oddezeni, ...)
- Determinismus:**
  - rozhodnout, které klauzule mají uspět vícekrát, ověřit požadovaný determinismus