

Vestavěné predikáty (pokračování)

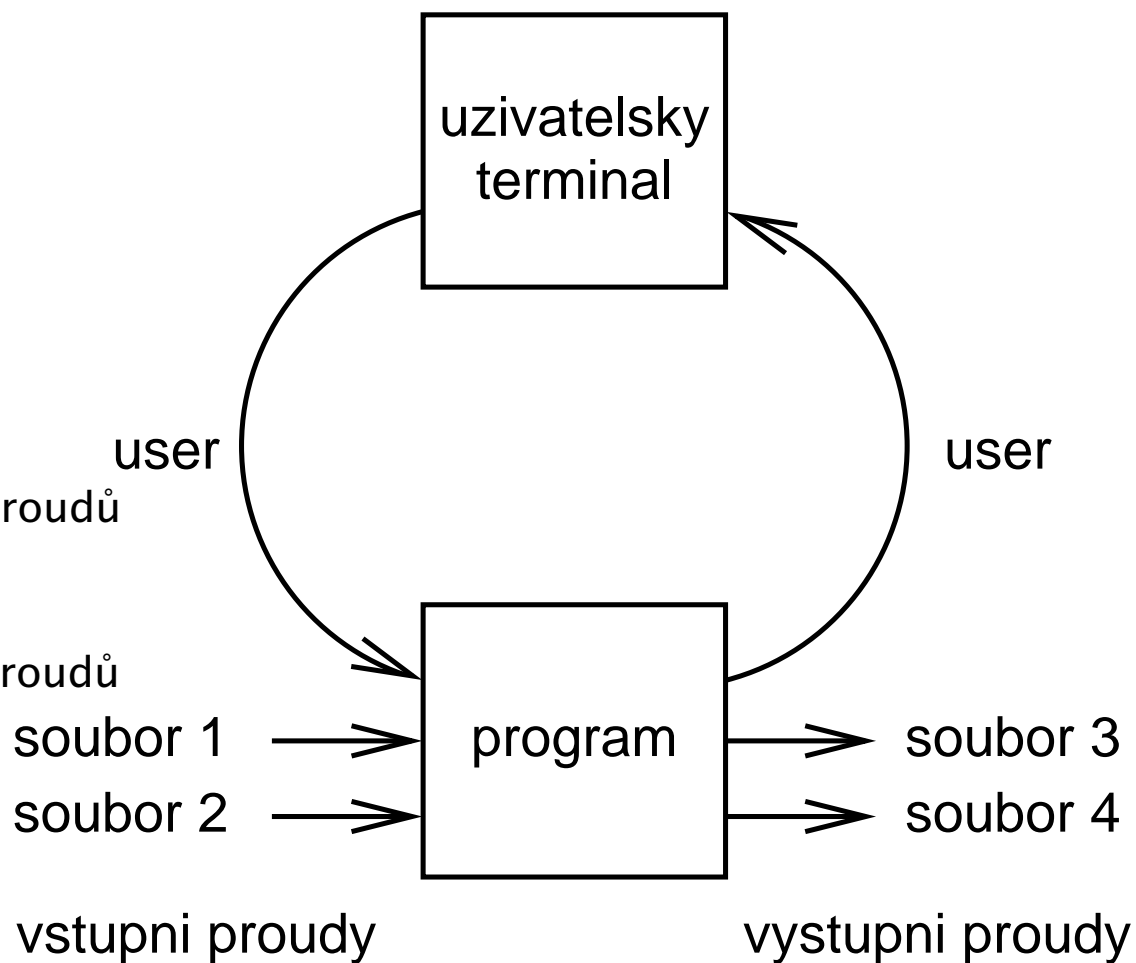
Vstup a výstup

- program může číst data ze **vstupního proudu** (*input stream*)
- program může zapisovat data do **výstupního proudu** (*output stream*)
- dva **aktivní proudy**

- aktivní vstupní proud
- aktivní výstupní proud

- uživatelský terminál – user**

- datový vstup z terminálu
chápán jako jeden ze vstupních proudů
- datový výstup na terminál
chápán jako jeden z výstupních proudů



Vstupní a výstupní proudy: vestavěné predikáty

- změna (**otevření**) aktivního vstupního/výstupního proudu: `see(S)/tell(S)`

```
cteni( Soubor ) :- see( Soubor ),  
                  cteni_ze_souboru( Informace ),  
                  see( user ).
```

- **uzavření** aktivního vstupního/výstupního proudu: `seen/told`

Vstupní a výstupní proudy: vestavěné predikáty

- změna (**otevření**) aktivního vstupního/výstupního proudu: see(S)/tell(S)

```
cteni( Soubor ) :- see( Soubor ),  
                  cteni_ze_souboru( Informace ),  
                  see( user ).
```

- **uzavření** aktivního vstupního/výstupního proudu: seen/told

- **zjištění** aktivního vstupního/výstupního proudu: seeing(S)/telling(S)

```
cteni( Soubor ) :- seeing( StarySoubor ),  
                  see( Soubor ),  
                  cteni_ze_souboru( Informace ),  
                  seen,  
                  see( StarySoubor ).
```

Sekvenční přístup k textovým souborům

- **čtení** dalšího **termu**: `read(Term)`

- při čtení jsou termy odděleny tečkou

- | `?- read(A), read(ahoj(B)), read([C,D]).`

Sekvenční přístup k textovým souborům

- **čtení** dalšího **termu**: `read(Term)`

- při čtení jsou termy odděleny tečkou

```
| ?- read(A), read( ahoj(B) ), read( [C,D] ).
```

```
|: ahoj. ahoj( petre ). [ ahoj( 'Petre!' ), jdeme ].
```

```
A = ahoj, B = petre, C = ahoj('Petre!'), D = jdeme
```

Sekvenční přístup k textovým souborům

● čtení dalšího **termu**: `read(Term)`

- při čtení jsou termy odděleny tečkou

```
| ?- read(A), read( ahoj(B) ), read( [C,D] ).
```

```
|: ahoj. ahoj( petre ). [ ahoj( 'Petre!' ), jdeme ].
```

A = ahoj, B = petre, C = ahoj('Petre!'), D = jdeme

- po dosažení konce souboru je vrácen atom `end_of_file`

● zápis dalšího **termu**: `write(Term)`

```
?- write( ahoj ).      ?- write( 'Ahoj Petre!' ).
```

nový řádek na výstup: `n\`

N mezer na výstup: `tab(N)`

Sekvenční přístup k textovým souborům

● **čtení** dalšího **termu**: `read(Term)`

- při čtení jsou termy odděleny tečkou

```
| ?- read(A), read( ahoj(B) ), read( [C,D] ).
```

```
|: ahoj. ahoj( petre ). [ ahoj( 'Petre!' ), jdeme ].
```

A = ahoj, B = petre, C = ahoj('Petre!'), D = jdeme

- po dosažení konce souboru je vrácen atom `end_of_file`

● **zápis** dalšího **termu**: `write(Term)`

```
?- write( ahoj ).      ?- write( 'Ahoj Petre!' ).
```

nový řádek na výstup: `n`

N mezer na výstup: `tab(N)`

● **čtení/zápis** dalšího **znaku**: `get0(Znak)`, `get(NeprazdnyZnak)/put(Znak)`

- po dosažení konce souboru je vrácena `-1`

Příklad čtení ze souboru

```
process_file( Soubor ) :-
    seeing( StarySoubor ),           % zjištění aktivního proudu
    see( Soubor ),                   % otevření souboru Soubor
    repeat,
        read( Term ),                % čtení termu Term
        process_term( Term ),        % manipulace s termem
        Term == end_of_file,         % je konec souboru?
    !,
    seen,                             % uzavření souboru
    see( StarySoubor ).              % aktivace původního proudu

repeat.                               % opakování
repeat :- repeat.
```

Čtení programu ze souboru

● Interpretování kódu programu

- ?- consult(program).
- ?- consult('program.pl').
- ?- consult([program1, 'program2.pl']).

● Kompilace kódu programu

- ?- compile([program1, 'program2.pl']).
- ?- [program].
- ?- [user]. **zadávání kódu ze vstupu** ukončené CTRL+D
- další varianty podobně jako u interpretování
- typické zrychlení: 5 až 10 krát

Všechna řešení

- Backtracking vrací pouze jedno řešení po druhém
- Všechna řešení dostupná najednou: [bagof/3](#), [setof/3](#), [findall/3](#)
- [bagof\(X, P, S \)](#): vrátí seznam S, všech objektů X takových, že P je splněno

`vek(petr, 7).`

`vek(anna, 5).`

`vek(tomas, 5).`

?- `bagof(Dite, vek(Dite, 5), Seznam).`

Všechna řešení

- Backtracking vrací pouze jedno řešení po druhém
- Všechna řešení dostupná najednou: [bagof/3](#), [setof/3](#), [findall/3](#)
- `bagof(X, P, S)`: vrátí seznam S, všech objektů X takových, že P je splněno

`vek(petr, 7).`

`vek(anna, 5).`

`vek(tomas, 5).`

?- `bagof(Dite, vek(Dite, 5), Seznam).`

`Seznam = [anna, tomas]`

Všechna řešení

- Backtracking vrací pouze jedno řešení po druhém
- Všechna řešení dostupná najednou: [bagof/3](#), [setof/3](#), [findall/3](#)
- [bagof\(X, P, S \)](#): vrátí seznam S, všech objektů X takových, že P je splněno

`vek(petr, 7).`

`vek(anna, 5).`

`vek(tomas, 5).`

?- `bagof(Dite, vek(Dite, 5), Seznam).`

`Seznam = [anna, tomas]`

- Volné proměnné v cíli P jsou **všeobecně kvantifikovány**

?- `bagof(Dite, vek(Dite, Vek), Seznam).`

Všechna řešení

- Backtracking vrací pouze jedno řešení po druhém
- Všechna řešení dostupná najednou: `bagof/3`, `setof/3`, `findall/3`
- `bagof(X, P, S)`: vrátí seznam S, všech objektů X takových, že P je splněno

`vek(petr, 7).`

`vek(anna, 5).`

`vek(tomas, 5).`

?- `bagof(Dite, vek(Dite, 5), Seznam).`

`Seznam = [anna, tomas]`

- Volné proměnné v cíli P jsou **všeobecně kvantifikovány**

?- `bagof(Dite, vek(Dite, Vek), Seznam).`

`Vek = 7, Seznam = [petr];`

`Vek = 5, Seznam = [anna, tomas]`

Všetchna řešení II.

- Pokud neexistuje řešení `bagof(X,P,S)` neuspěje
- `bagof`: pokud nějaké řešení existuje několikrát, pak `S` obsahuje duplicity
- `bagof`, `setof`, `findall`:

`P` je libovolný cíl

```
vek( petr, 7 ).
```

```
vek( anna, 5 ).
```

```
vek( tomas, 5 ).
```

```
?- bagof( Dite, ( vek( Dite, 5 ), Dite \= anna ), Seznam ).
```

```
Seznam = [ tomas ]
```

Všetchna řešení II.

- Pokud neexistuje řešení `bagof(X,P,S)` neuspěje
- `bagof`: pokud nějaké řešení existuje několikrát, pak `S` obsahuje duplicity
- `bagof`, `setof`, `findall`:

`P` je libovolný cíl

```
vek( petr, 7 ).
```

```
vek( anna, 5 ).
```

```
vek( tomas, 5 ).
```

```
?- bagof( Dite, ( vek( Dite, 5 ), Dite \= anna ), Seznam ).
```

```
Seznam = [ tomas ]
```

- `bagof`, `setof`, `findall`:

na objekty shromažďované v `X` nejsou žádná omezení: `X` je term

```
?- bagof( Dite-Vek, vek( Dite, Vek ), Seznam ).
```

```
Seznam = [petr-7,anna-5,tomas-5]
```


Existenční kvantifikátor „ \exists ”

- Přidání **existenčního kvantifikátoru** „ \exists ” \Rightarrow hodnota proměnné nemá význam

?- bagof(Dite, Vek \exists vek(Dite, Vek), Seznam).

Existenční kvantifikátor „ \exists ”

- Přidání **existenčního kvantifikátoru** „ \exists ” \Rightarrow hodnota proměnné nemá význam

?- bagof(Dite, Vek \exists vek(Dite, Vek), Seznam).

Seznam = [petr,anna,tomas]

Existenční kvantifikátor „ \exists ”

- Přidání **existenčního kvantifikátoru** „ \exists ” \Rightarrow hodnota proměnné nemá význam

?- bagof(Dite, Vek \exists vek(Dite, Vek), Seznam).

Seznam = [petr,anna,tomas]

- Anonymní proměnné jsou všeobecně kvantifikovány, i když jejich hodnota není (jako vždy) vracena na výstup

?- bagof(Dite, vek(Dite, _Vek), Seznam).

Seznam = [petr] ;

Seznam = [anna,tomas]

Existenční kvantifikátor „ \exists ”

- Přidání **existenčního kvantifikátoru** „ \exists ” \Rightarrow hodnota proměnné nemá význam

?- bagof(Dite, Vek \exists vek(Dite, Vek), Seznam).

Seznam = [petr,anna,tomas]

- Anonymní proměnné jsou všeobecně kvantifikovány, i když jejich hodnota není (jako vždy) vracena na výstup

?- bagof(Dite, vek(Dite, _Vek), Seznam).

Seznam = [petr] ;

Seznam = [anna,tomas]

- Před operátorem „ \exists ” může být i seznam

?- bagof(Vek ,[Jmeno,Prijmeni] \exists vek(Jmeno, Prijmeni, Vek), Seznam).

Seznam = [7,5,5]

Všetchna řešení III.

- `setof(X, P, S)`: rozdíly od `bagof`
 - S je uspořádaný podle `@<`
 - případné duplicity v S jsou eliminovány

Všechna řešení III.

● `setof(X, P, S)`: rozdíl od `bagof`

- S je uspořádaný podle $@<$

- případné duplicity v S jsou eliminovány

● `findall(X, P, S)`: rozdíl od `bagof`

- všechny proměnné jsou existenčně kvantifikovány

?- `findall(Dite, vek(Dite, Vek), Seznam)`.

Všechna řešení III.

● `setof(X, P, S)`: rozdíl od `bagof`

● S je uspořádaný podle $@<$

● případné duplicity v S jsou eliminovány

● `findall(X, P, S)`: rozdíl od `bagof`

● všechny proměnné jsou existenčně kvantifikovány

?- `findall(Dite, vek(Dite, Vek), Seznam)`.

⇒ v S jsou shromažďovány všechny možnosti i pro různá řešení

⇒ `findall` uspěje přesně jednou

Všetchna řešení III.

● `setof(X, P, S)`: rozdíl od `bagof`

- S je uspořádaný podle $@<$
- případné duplicity v S jsou eliminovány

● `findall(X, P, S)`: rozdíl od `bagof`

- všechny proměnné jsou existenčně kvantifikovány

?- `findall(Dite, vek(Dite, Vek), Seznam)`.

⇒ v S jsou shromažďovány všechny možnosti i pro různá řešení

⇒ `findall` uspěje přesně jednou

- výsledný seznam může být prázdný ⇒ pokud neexistuje řešení, uspěje a vrátí `S = []`

Všetchna řešení III.

● `setof(X, P, S)`: rozdíl od `bagof`

- S je uspořádaný podle `@<`
- případné duplicity v S jsou eliminovány

● `findall(X, P, S)`: rozdíl od `bagof`

- všechny proměnné jsou existenčně kvantifikovány

?- `findall(Dite, vek(Dite, Vek), Seznam)`.

⇒ v S jsou shromažďovány všechny možnosti i pro různá řešení

⇒ `findall` uspěje přesně jednou

- výsledný seznam může být prázdný ⇒ pokud neexistuje řešení, uspěje a vrátí `S = []`

- ?- `bagof(Dite, vek(Dite, Vek), Seznam)`.

Vek = 7, Seznam = [petr];

Vek = 5, Seznam = [anna, tomas]

?- `findall(Dite, vek(Dite, Vek), Seznam)`.

Všechna řešení III.

● `setof(X, P, S)`: rozdíl od `bagof`

- S je uspořádaný podle `@<`
- případné duplicity v S jsou eliminovány

● `findall(X, P, S)`: rozdíl od `bagof`

- všechny proměnné jsou existenčně kvantifikovány

?- `findall(Dite, vek(Dite, Vek), Seznam)`.

⇒ v S jsou shromažďovány všechny možnosti i pro různá řešení

⇒ `findall` uspěje přesně jednou

- výsledný seznam může být prázdný ⇒ pokud neexistuje řešení, uspěje a vrátí `S = []`

- ?- `bagof(Dite, vek(Dite, Vek), Seznam)`.

Vek = 7, Seznam = [petr];

Vek = 5, Seznam = [anna, tomas]

?- `findall(Dite, vek(Dite, Vek), Seznam)`.

Seznam = [petr,anna,tomas]

Testování typu termu

`var(X)`

X je volná proměnná

`nonvar(X)`

X není proměnná

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 (pavel, 'Pavel Novák', <-->)
<code>integer(X)</code>	X je integer
<code>float(X)</code>	X je float
<code>atomic(X)</code>	X je atom nebo číslo

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 (pavel, 'Pavel Novák', <-->)
<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

`count(X, S, N)`

Určení počtu výskytů prvku v seznamu

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

Určení počtu výskytů prvku v seznamu

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

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

Určení počtu výskytů prvku v seznamu

`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).`

Určení počtu výskytů prvku v seznamu

```
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).
```

Určení počtu výskytů prvku v seznamu

```
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 )
```

```
N=3
```

Určení počtu výskytů prvku v seznamu

```
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

Určení počtu výskytů prvku v seznamu

```
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 )
```

N=3

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

N=3

Určení počtu výskytů prvku v seznamu

```
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 ).
```

Konstrukce a dekompozice atomu

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`

Konstrukce a dekompozice atomu

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 a dekompozice atomu

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]

Konstrukce a dekompozice termu

● Konstrukce a dekompozice termu

Term =.. [Funktor | SeznamArgumentu]

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

Ci1 =.. [Funktor | SeznamArgumentu], call(Ci1)

Konstrukce a dekompozice termu

● Konstrukce a dekompozice termu

Term =.. [Funktor | SeznamArgumentu]

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

Ci1 =.. [Funktor | SeznamArgumentu], call(Ci1)

atom =.. X

Konstrukce a dekompozice termu

● Konstrukce a dekompozice termu

Term =.. [Funktor | SeznamArgumentu]

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

$Ci1 =.. [Funktor | SeznamArgumentu], call(Ci1)$

atom =.. $X \Rightarrow X = [atom]$

Konstrukce a dekompozice termu

- Konstrukce a dekompozice termu

Term =.. [Funktor | SeznamArgumentu]

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

$Ci1 =.. [Funktor | SeznamArgumentu], call(Ci1)$

atom =.. $X \Rightarrow X = [atom]$

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

`functor(Term, Funktor, Arita)`

`functor(a(9,e), a, 2)`

Konstrukce a dekompozice termu

- Konstrukce a dekompozice termu

Term =.. [Funktor | SeznamArgumentu]

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

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

atom =.. X \Rightarrow 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)

Konstrukce a dekompozice termu

● Konstrukce a dekompozice termu

Term =.. [Funktor | SeznamArgumentu]

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

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

atom =.. X \Rightarrow 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)

Rekurzivní rozklad termu

- Term je proměnná (`var/1`), atom nebo číslo (`atomic/1`) \Rightarrow konec rozkladu

Rekurzivní rozklad termu

- Term je proměnná (var/1), atom nebo číslo (atomic/1) \Rightarrow konec rozkladu
- Term je složený (= ./2, functor/3) \Rightarrow
procházení seznamu argumentů a rozklad každého argumentu

Rekurzivní rozklad termu

- Term je proměnná (var/1), atom nebo číslo (atomic/1) \Rightarrow konec rozkladu
- Term je seznam ([_ | _]) \Rightarrow [] ... řešen výše jako atomic
procházení seznamu a rozklad každého prvku seznamu
- Term je složený (= . ./2, functor/3) \Rightarrow
procházení seznamu argumentů a rozklad každého argumentu

Rekurzivní rozklad termu

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

Rekurzivní rozklad termu

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

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

Rekurzivní rozklad termu

- Term je proměnná (`var/1`), atom nebo číslo (`atomic/1`) \Rightarrow konec rozkladu
- Term je seznam (`[_|_]`) \Rightarrow `[]` ... řešen výše jako `atomic`
procházení seznamu a rozklad každého prvku seznamu
- Term je složený (`=./2`, `functor/3`) \Rightarrow
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.
```

Rekurzivní rozklad termu

- Term je proměnná (`var/1`), atom nebo číslo (`atomic/1`) \Rightarrow konec rozkladu
- Term je seznam (`[_|_]`) \Rightarrow `[]` ... řešen výše jako `atomic`
procházení seznamu a rozklad každého prvku seznamu
- Term je složený (`=./2`, `functor/3`) \Rightarrow
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).
```

Rekurzivní rozklad termu

- Term je proměnná (`var/1`), atom nebo číslo (`atomic/1`) \Rightarrow konec rozkladu
- Term je seznam (`[_|_]`) \Rightarrow [] ... řešen výše jako `atomic`
procházení seznamu a rozklad každého prvku seznamu
- Term je složený (`=.. /2`, `functor/3`) \Rightarrow
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 ).
```


Rekurzivní rozklad termu

- Term je proměnná (`var/1`), atom nebo číslo (`atomic/1`) \Rightarrow konec rozkladu
- Term je seznam (`[_|_]`) \Rightarrow [] ... řešen výše jako `atomic`
procházení seznamu a rozklad každého prvku seznamu
- Term je složený (`=.. /2`, `functor/3`) \Rightarrow
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)).
```

no

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

yes

Příklad: dekompozice termu I.

- `count_term(Integer, Term, N)` určí počet výskytů celého čísla v termu

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).` $N=2$

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).`

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.`

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), !.`

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), !.`

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],`

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).`

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).`

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),`

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,`

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).`

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).`

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`

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` jsou termy bez proměnných
- ?- `substitute(sin(x), 2*sin(x)*f(sin(x)), t, F)`. $F=2*t*f(t)$