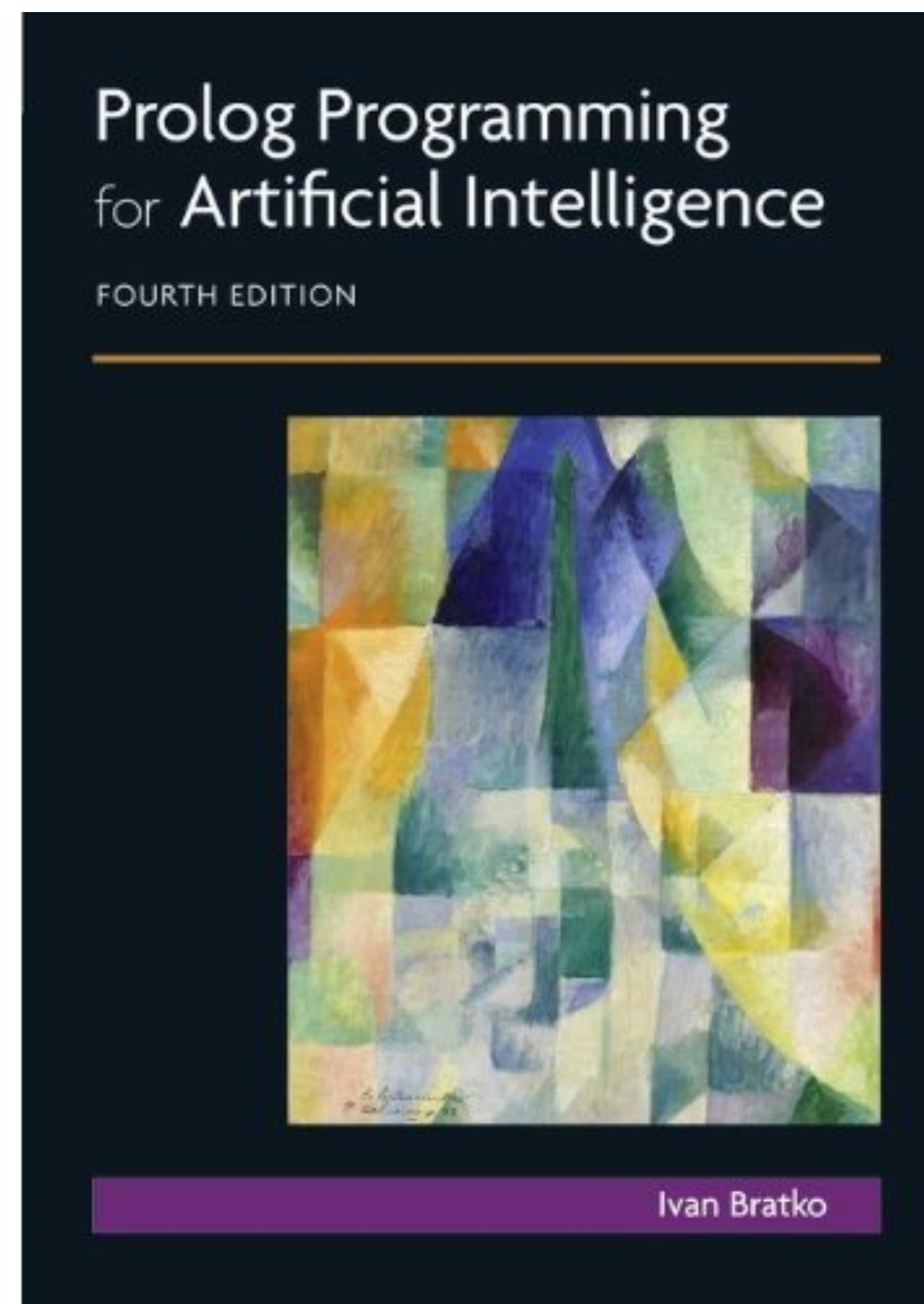








Neprocedurální programování

Prolog 4





Řez a negace



Co bylo minule

-  Koncová rekurze
-  Ladění
-  Deklarativní a procedurální význam programu
-  Aritmetika
-  Binární stromy v Prologu
-  Predikáty vyšších řádů

Osnova

-  Seznamy seznamů
-  Neúplně definované datové struktury
-  Řez a negace
-  Vestavěné predikáty pro testování struktury termu

Seznamy seznamů : Matice

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

`[[1, 2],
[3, 4],
[5, 6]]`

matice



seznam řádků

řádek
matice



seznam prvků
na řádku

Poslední sloupec matice

```
% posledni(+Matice,?Sloupec):- vrátí poslední  
%                               Sloupec Matice.  
  
posledni(Matice, Sloupec):-  
    maplist(last, Matice, Sloupec).  
  
?- posledni([ [a,b,c,d],  
              [e,f,g,h],  
              [i,j,k,l] ], S).  
  
S = [d,h,l].
```

Transpozice matice

```
% tsp(+M,-TM) :- Matice TM je transpozicí  
%               matice M.
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

```
?- tsp [[1,2],[3,4],[5,6]],T).  
    T = [[1,3,5],[2,4,6]].
```

Transpozice matice pomocí `maplist/3`

```
seznam(X,Xs,[X|Xs]).    % oddělí hlavu | tělo

tsp(Xss,[ ]):- prazdna_matice(Xss). % báze
tsp(Xss,[Xs|Zss]):-
    maplist(seznam,Xs,Yss,Xss), % oddělí 1.sloupec
    tsp(Yss,Zss). % rekurzivně transponuje ostatní
```

 **Problém:** Jak definovat `prazdna_matice/1`?

Neúplně definované datové struktury

$[a, b, c]$
seznam



$[a, b, c | S] - S$
rozdílový seznam

volná
proměnná

Zřetězení rozdílových seznamů

- v konstantním čase
- `zretez(A-B, B-C, A-C)`.

?- `zretez([a,b,c|X]-X, [d,e|Y]-Y, Z)`.

`X = [d,e|Y],`

`Z = [a,b,c,d,e|Y] - Y`

Rozdílové seznamy

obyčejný seznam \leftrightarrow rozdílový seznam

```
% prevod1(+Os,-Rs) :- Rs je rozdílová reprezentace  
%                      obyčejného seznamu Os.
```

```
?- prevod1([a,b,c],Rs).
```

```
Rs = [a,b,c | S] - S.
```

```
prevod1([],S-S).
```

```
prevod1([X|Xs],[X|S]-T):-prevod1(Xs,S-T).
```

```
% prevod2(-Os,+Rs).
```

```
?- prevod2(Os,[a,b,c|S]-S).
```

```
Os = [a,b,c].
```

```
prevod2(Xs,Xs-[]).
```

Quicksort efektivně

```
quicksort([ ], S-S).
```

```
quicksort([X|Xs], -) :- split(X,Xs,Ys,Zs),  
                        quicksort(Ys, -),  
                        quicksort(Zs, -),  
                        append( , ).
```

Problém

- navrhnete efektivní verzi třídění quicksortem
- odstráňte explicitní volání predikátu `append/3`
- ke zřetězení využijte rozdílové seznamy

Predikát $!/0$

- vždy uspěje
- při pokusu o návrat při backtrackingu způsobí okamžité selhání splňovaného cíle

$c_1 :- p_1, \dots, p_i, !, p_j, \dots, p_k.$

$c_2 :- p_m, \dots, p_n.$

- c_1 a c_2 jsou termy s hlavním funktorem c
- p_i uspěje \Rightarrow uspěje i $!$
- p_j selže \Rightarrow selže cíl c

☀ Příklad řezu

```
% prvek(X, Xs) :- X je prvkem seznamu Xs.
```

```
prvek(X, [X|_]).
```

```
prvek(X, [_|Xs]) :- prvek(X, Xs).
```

```
% prvek_det(X, Xs) :- prvek/1 deterministicky.
```

```
prvek_det(X, [X|_]) :- !.
```

```
prvek_det(X, [_|Xs]) :- prvek_det(X, Xs).
```

- `prvek_det(-X, +S)` vrátí **první** prvek X v S

Červený řez

Mění deklarativní význam programu

$p :- a, b.$

$p :- c.$

$p \Leftarrow (a \wedge b) \vee c$

$p :- a, !, b.$

$p :- c.$

$p \Leftarrow (a \wedge b) \vee (\neg a \wedge c)$

☀ Příklad: $\text{prvek}/2$ vs. $\text{prvek_det}/2$

Zelený řez

Nemění deklarativní význam programu

- pouze “odřezává” neperspektivní větve výpočtu

```
max(X, Y, X) :- X >= Y, !.
```

```
max(X, Y, Y) :- X < Y.
```

☞ Ale pozor

```
max(X, Y, X) :- X >= Y, !.
```

```
max(X, Y, Y).
```

- ?- max(2, 1, 1).

true

chyba !!!

Negace neúspěchem

```
jazyk(c).
```

```
jazyk(python).
```

```
jazyk(prolog).
```

```
jazyk(haskell).
```

```
proc(c).
```

```
proc(python).
```

Ada má ráda programovací jazyky :

```
ma_rada(ada, X) :- jazyk(X).
```

Negace neúspěchem

```
jazyk(c).  
jazyk(python).  
jazyk(prolog).  
jazyk(haskell).
```

```
proc(c).  
proc(python).
```

Ada má ráda programovací jazyky, ale ne ty procedurální:

```
ma_rada(ada, X) :- proc(X), !, fail.
```

```
ma_rada(ada, X) :- jazyk(X).
```


Negace: definice pomocí řezu

```
% \+(C) :- Cíl C nelze splnit.
```

```
\+(C) :- call(C), !, fail.
```

```
\+(C) .
```

Doporučená notace pro negaci

- operátor `\+`
- `:- op(900, fy, \+)`.
- `not\1` definován jen kvůli kompatibilitě

```
ma_rada(ada, X) :- jazyk(X), \+ proc(X) .
```

Prolog: negace neúspěchem

\+

- neodpovídá negaci v matematické logice
- negace neúspěchem
- předpoklad uzavřeného světa

☀ Příklad negace neúspěchem

```
jazyk(c).
```

```
jazyk(python).
```

```
jazyk(prolog).
```

```
jazyk(haskell).
```

```
proc(c).
```

```
proc(python).
```

```
?- proc(X).
```

```
    X = c ;
```

```
    X = python
```

☀ Příklad negace neúspěchem

```
jazyk(c).  
jazyk(python).  
jazyk(prolog).  
jazyk(haskell).
```

```
proc(c).  
proc(python).
```

```
?- \+ proc(X).
```

```
    false
```

```
?- jazyk(X), \+ proc(X).
```

```
    X = prolog ;
```

```
    X = haskell
```

☀ Příklad negace neúspěchem

```
jazyk(c).
```

```
jazyk(python).
```

```
jazyk(prolog).
```

```
jazyk(haskell).
```

```
proc(c).
```

```
proc(python).
```

```
?- \+ proc(X).
```

```
false
```

```
?- \+ proc(X), jazyk(X).
```

```
false
```

Negace: volné proměnné

`\+` `C`

- `C` může obsahovat volné proměnné

Možné řešení

- definovat negaci (`not/1`) pouze pro konstantní termy
- SWI Prolog
 - » `not/1` ekvivalentní `\+`
 - » norma (ISO) doporučuje používat `\+`

Negace nebo řez?

```
porazil(smid, panatta).  
porazil(lendl, barazzutti).  
porazil(barazzutti, smid).
```

Definujme predikat

`kategorie(+Hrac, -Trida)`

pro tridy

- vitez
- bojovnik
- sportovec

Negace ... ?

```
% kategorie(+Hrac,-Trida)
```

```
kategorie(X,vitez):- porazil(X,_),  
                      \+ porazil(_,X).
```

```
kategorie(X,bojovnik):- porazil(X,_),  
                        porazil(_,X).
```

```
kategorie(X,sportovec):- porazil(_,X),  
                          \+ porazil(X,_).
```

✗ Nevýhoda

- opakované vyhodnocení téhož cíle

... nebo řez?

```
kategorie(X,bojovník):- porazil(X,_),  
                        porazil(_,X),!.  
kategorie(X,vitez):-   porazil(X,_),!.  
kategorie(X,sportovec):-porazil(_,X).
```

Idiom

```
p :- test1, !, tělo1.  
p :- test2, !, tělo2.  
p :- tělo3.
```

... nebo řez?

```
kategorie(X,bojovnik):- porazil(X,_),  
                        porazil(_,X),!.  
kategorie(X,vitez):-   porazil(X,_),!.  
kategorie(X,sportovec):-porazil(_,X).
```

 **Problém:** Jak dopadnou dotazy typu

- `kategorie(+Hrac,+Trida)`
- `kategorie(-Hrac,+Trida)?`

Neunifikovatelné ...

```
% neunif(X,Y) :- X a Y nelze unifikovat.
```

```
neunif(X,Y) :- X = Y, !, fail.
```

```
neunif(X,Y).
```

Předdefinovaný operátor $\backslash=$.

```
% alternativní definice
```

```
neunif(X,Y) :- \+ (X = Y) .
```

... a různé

```
% ruzne(X,Y) :- X a Y jsou ruzne.
```

```
ruzne(X,Y) :- X == Y, !, fail.
```

```
ruzne(X,Y) .
```

Předdefinovaný operátor `\==` .

```
ruzne(X,Y) :- \+ (X == Y) .
```

ekvivalence termů

Zkrocení řezu

```
% once(Cíl) :- vrátí první řešení,  
%               které splní Cíl.
```

```
once(C) :- C, !.
```

```
% forall(+Podminka, +Cil):-  
%               uspěje, pokud Cil lze splnit  
%               pro všechny hodnoty proměnných  
%               pro než lze splnit Podminku
```

```
forall(Podminka, Cil) :-  
    \+ (Podminka, \+ Cil).
```

If -> Then ; Else

If -> Then ; _ :- If, !, Then.

If -> _ ; Else :- !, Else.

If -> Then :- If, !, Then.

✓ Podmínka **If** se vyhodnocuje jen jednou

✓ Uvnitř “větví” **Then** a **Else** možný backtracking

☀ Příklad

```
% rozdíl(+Xs,+Ys,-Zs):- seznam Zs je rozdílem  
%                               množin, reprezentovaných  
%                               seznamy Xs a Ys.
```

Rozdíl pomocí negace

```
rozdil([ ],_,[ ]).
```

```
rozdil([X|Xs],Ys,Zs):- member(X,Ys),  
                        rozdil(Xs,Ys,Zs).
```

```
rozdil([X|Xs],Ys,[X|Zs]):- \+ member(X,Ys),  
                           rozdil(Xs,Ys,Zs).
```

Rozdíl pomocí řezu

```
rozdil([ ],_,[ ]).
```

```
rozdil([X|Xs],Ys,Zs):- member(X,Ys),!,  
                        rozdil(Xs,Ys,Zs).
```

```
rozdil([X|Xs],Ys,[X|Zs]):- rozdil(Xs,Ys,Zs).
```


Rozdíl pomocí if-then-else

```
rozdil([ ],_,[ ]).
```

```
rozdil([X|Xs],Ys,Zs):-
```

```
    (member(X,Ys)-> Zs=Zs1; Zs=[X|Zs1]),
```

```
    rozdil(Xs,Ys,Zs1).
```

Predikát lze nalézt ve standardní knihovně jako

`subtract/3`

Predikáty pro řízení výpočtu

Predikáty pro řízení výpočtu

- ✓ nabízejí idiomy imperativního programování
- může existovat elegantnější řešení v neprocedurálním duchu

Predikáty vyšších řádů: `filter`

```
% filter(+P, +Xs, ?Ys) :- seznam Ys obsahuje právě  
%                          ty prvky X seznamu Xs  
%                          pro něž call(P, X) uspěje.  
  
filter(_, [], []).  
  
filter(P, [X|Xs], [X|Ys]) :- call(P, X), !, filter(P, Xs, Ys).  
filter(P, [_|Xs], Ys) :- filter(P, Xs, Ys).  
  
?- filter(muz, [adam, eva, kain, abel]).  
   Ys = [adam, kain, abel].
```

Viz knihovní predikát `include/3` resp. `exclude/3`.

Vestavěné predikáty: test typu termu

`atom/1` argumentem je atom

`atomic/1` argumentem je konstanta

`number/1` `integer/1` `float/1`

`var/1` argumentem je volná proměnná

`nonvar/1` argumentem není volná proměnná

`ground/1` argumentem je konstantní term

- bez volných proměnných

`compound/1` argumentem je složený term

Rozbor struktury termu: univ

Vestavěný operátor $=..$

- univ
- Term $=..$ Seznam
 - » Seznam = [HlavniFunktor | SeznamArgumentu]
 - » $+Term =.. -Seznam$
 - » $-Term =.. +Seznam$

?- $f(a,b) =.. S.$

$S = [f,a,b]$

?- $T =.. [p,X,f(X,Y)].$

$T = p(X,f(X,Y))$

Rozbor struktury termu: functor

Specifičtější vestavěné predikáty

`functor(Term, F, A) :- Term` má hlavní funktor `F`
a aritu `A`.

- k termu určí funktor a aritu: `(+, ?, ?)`
- k funktoru a aritě vytvoří term: `(?, +, +)`

?- `functor(f(a,b), F, A).`

`F = f`

`A = 2`

?- `functor(Term, f, 2).`

`Term = f(_, _)`

Rozbor struktury termu: arg

`arg(?N,+Term,?A) :- A je N-tým argumentem Termu.`

`?- arg(2,f(X,t(a),t(b)),A).`

`A = t(a)`

☀ Příklad

`?- functor(D,datum,3),`

`arg(1,D,10),`

`arg(2,D,brezen),`

`arg(3,D,2025).`

`D = datum(10,brezen,2025)`

☀ Příklad zjednodušování výrazů

```
s(*, X, 1, X).
```

```
s(*, 1, X, X).
```

```
s(*, X, Y, Z) :- number(X), number(Y), Z is X*Y.
```

```
s(*, X, Y, X*Y). % zarážka pro *
```

Podobná tabulka pro další operátory

```
simp(V, V) :- atomic(V), !.
```

```
simp(V, ZV) :- V =.. [Op, La, Pa],  
               simp(La, ZL), simp(Pa, ZP),  
               s(Op, ZL, ZP, ZV).
```


Zjednodušování výrazů

?- `simp(2*3*a,Z).`

`Z = 6*a`

?- `simp(a*2*3,Z).`

`Z = a*2*3`

 **Problém:** Co s tím?

`s(*,X*Y,Z,W*X) :- number(Y), number(Z), W is Y*Z.`

☀ Příklad: symbolické derivování

```
?- der(x^3,x,D).
```

```
    D = 3*x^2
```

```
% der(+Vyráz,+X,-Der):- Der je derivací Vyrázu  
%                          vzhledem k proměnné X  
%                          Vyráz a X jsou konstantní termy
```

```
der(X,X,1).
```

```
der(Y,X,0) :- atomic(Y), X\=Y.
```

☀ Příklad: symbolické derivování

% derivace elementárních funkcí

der(sin(X), X, cos(X)).

der(cos(X), X, -sin(X)).

der(e^X, X, e^X).

der(ln(X), X, 1/X).

% derivace mocniny

der(X^N, X, N*X^N1) :- number(N), N1 is N-1.

☀ Příklad: symbolické derivování

% pravidla pro různé operátory

$\text{der}(F+G, X, DF+DG) :- \text{der}(F, X, DF), \text{der}(G, X, DG).$

$\text{der}(F-G, X, DF-DG) :- \text{der}(F, X, DF), \text{der}(G, X, DG).$

$\text{der}(F \cdot G, X, F \cdot DG + DF \cdot G) :- \text{der}(F, X, DF), \text{der}(G, X, DG).$

$\text{der}(F/G, X, (G \cdot DF - F \cdot DG) / (G \cdot G)) :- \text{der}(F, X, DF),$
 $\text{der}(G, X, DG).$

☀ Příklad: symbolické derivování

```
?- der(sin(cos(x)), x, D).
```

```
D = cos(cos(x))* -sin(x)
```

```
% derivace složené funkce
```

```
der(F_G_X, X, DF*DG) :- F_G_X =.. [_ , G_X],  
                        G_X \= X,  
                        der(F_G_X, G_X, DF),  
                        der(G_X, X, DG).
```

✎ **Problém:** neumí zjednodušit výsledek

```
?- der(x*x, x, D).
```

```
D = x*1+1*x
```