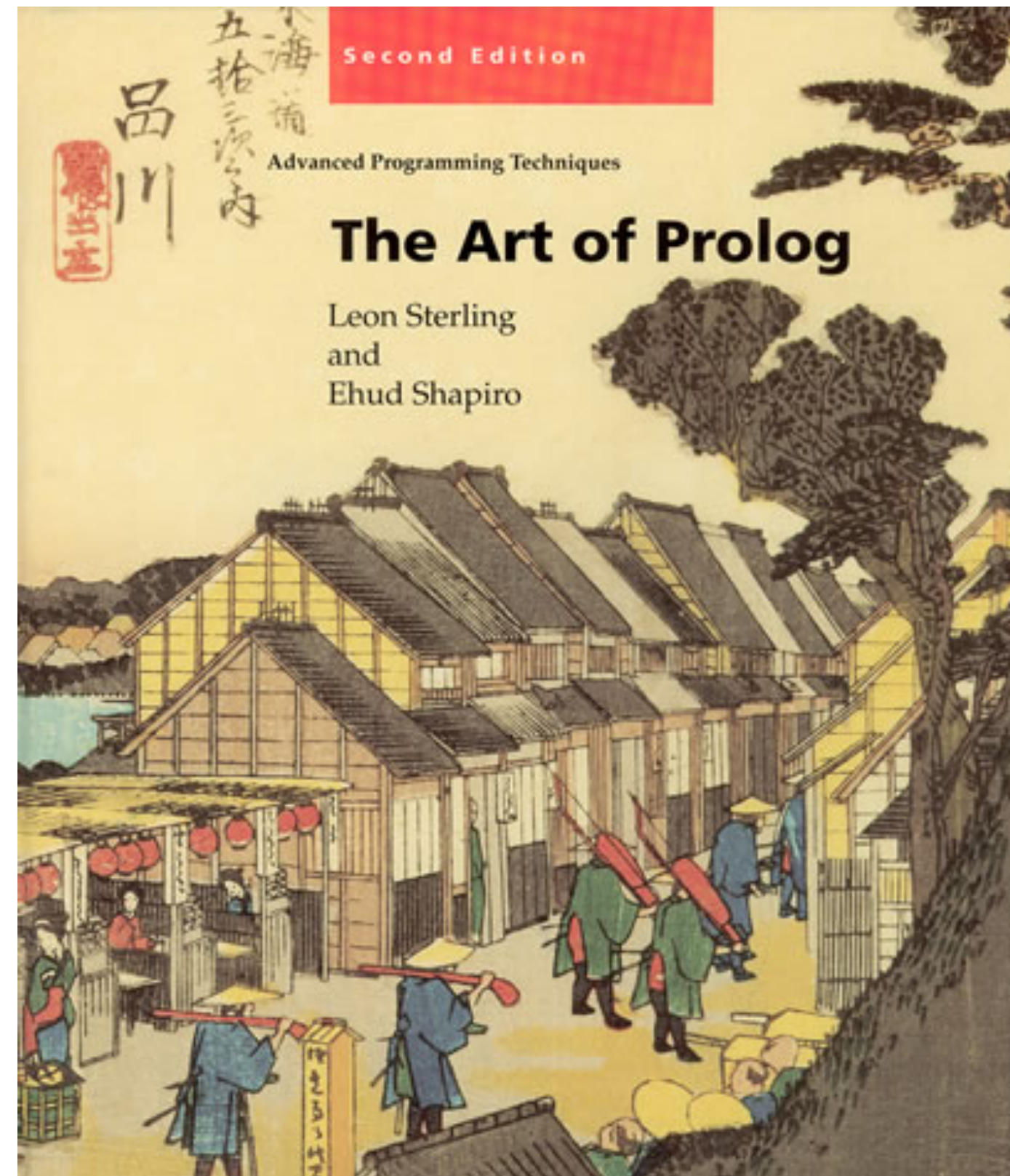






Neprocedurální programování

Prolog 3


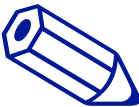




Aritmetika



Co bylo minule

-  Algoritmus splňování cíle
-  Rekurze
-  Směr výpočtu, nedeterminismus
-  Seznamy

Osnova

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

Koncová rekurze

$p(\dots) :- \underbrace{\dots}, p(\dots).$

zde se p nevyskytuje
pouze deterministické cíle
(žádný backtracking)

procedura p
je deterministická

Koncová rekurze

- návrat z každého rekurzivního volání je triviální
 - » úspora paměti
 - ✓ konstatní prostor na zásobníku
 - » rychlost
- rekurzi lze nahradit iterací
 - » překladač provádí automaticky
 - » Tail Recursion Optimization / Last Call Optimization

☀ Příklad (ne)koncové rekurze

`% otoc(+Xs,-Ys):- Ys je otočením seznamu Xs.`

`otoc([],[]).`

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

Procedura `otoc/2` **není** koncově rekurzivní

☀ Příklad koncové rekurze

`otocAk (Xs, Ys) :- otocAk (Xs, [], Ys) .`

`otocAk ([], A, A) .`

`otocAk ([X | Xs], A, Ys) :- otocAk (Xs, [X | A], Ys) .`

Procedura `otocAk/2` je **koncově rekurzivní**

- pouze konstatní prostor na zásobníku
- náhrada rekurze iterací

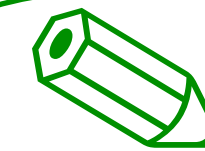
☀ Příklad (ne)koncové rekurze

% fak(+N,?F):- F je N faktoriál.

fak(0,s(0)).

fak(s(N),F) :- fak(N,F1), soucin(F1,s(N),F).

Procedura fak/2 **není** koncově rekurzivní



☀ Příklad koncové rekurze

```
% fak2(+N,?F):- F je N faktoriál.
```

```
fak2(N,F) :- fak2(N,s(0),F).
```

```
% fak2(N,A,F) :- F = N! · A .
```

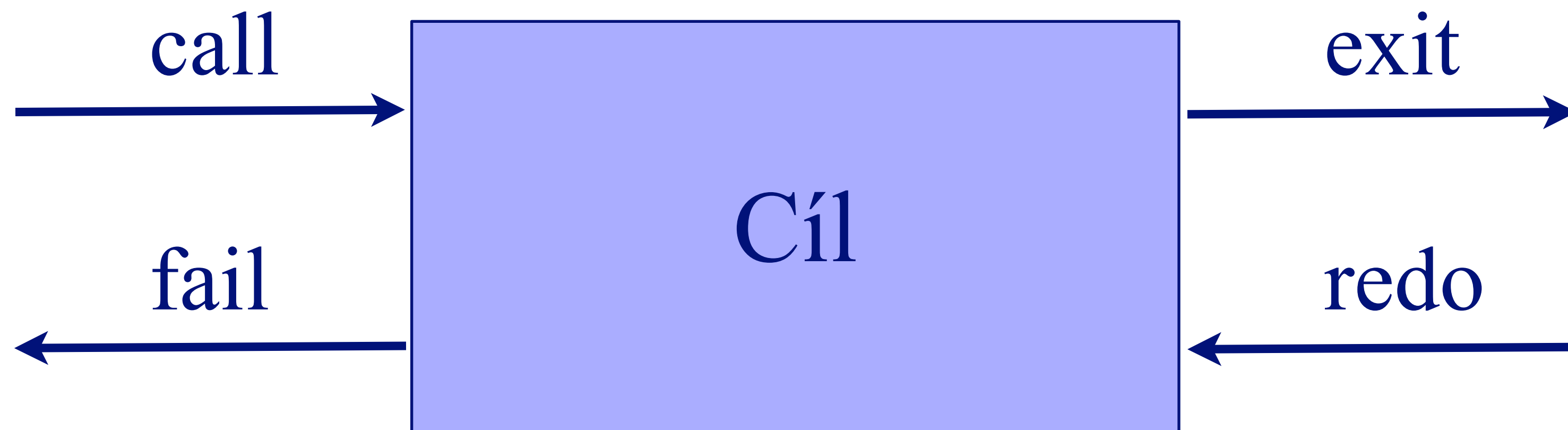
```
fak2(0,A,A).
```

```
fak2(s(N),A,F) :- soucin(s(N),A,A1), fak2(N,A1,F).
```

Procedura `fak2/2` je **koncově rekurzivní**

- dodatečný argument `A` hraje roli **akumulátoru**

Krabičkový model výpočtu



Ladění

Trasování výpočtu

- `trace/0` zapne, `notrace/0` vypne
- zastaví na každé bráně
- vypíše bránu, hloubku rekurze, cíl
- možnost zadávat příkazy
 - » výpis info nebo ovlivnění průběhu výpočtu

Grafický režim (SWI Prolog)

- `guitracer/0` zapne
- `noguitracer/0` vypne

Úplné trasování

- → nadměrný objem informací → selektivní trasování

Ladění

Sledování vybraných predikátů (“spypoint”)

- `spy(Pred)` nastaví sledování predikátu `Pred`
 - » `?- spy(mensi/2).`
 - » zapne režim ladění
 - » `[debug] ?-`
 - » pustí trasování až poté, když poprvé narazí na `mensi/2`
- `nospy(Pred)` zruší sledování `Pred`
- `nospyall/0` zruší vše
- `nodebug/0` vypne režim ladění
 - » “spypoint” bude ignorován (ale ne zapomenut)
- `debug/0` zapne ladění

Deklarativní význam programu

Pravidla

- $p :- q, r.$ $p :- q; r.$

mají význam formulí

- $q \wedge r \Rightarrow p$ $q \vee r \Rightarrow p$

Deklarativní význam programu

- množina formulí, které určují význam klauzulí programu
- nezávisí na pořadí klauzulí programu
- nezávisí na pořadí termů v těle pravidel
 - » \wedge a \vee jsou komutativní

Procedurální význam programu

Pravidlo

- $p :- q, r.$

lze interpretovat i takto

- pro splnění cíle p je třeba nejprve splnit podcíl q a potom podcíl r

Procedurální význam

- = procedura splňování cíle vzhledem k danému programu
- závisí na pořadí klauzulí programu i termů v těle pravidel

Deklarativní / procedurální správnost programu

Program může být

- **správný deklarativně**
 - » odpověď na dotaz existuje
- **leč nesprávný procedurálně**
 - » odpověď nelze nalézt procedurou splňování cíle
 - » popsaná procedura splňování cíle není úplná

Je-li program **správný deklarativně**

- nemůže dát chybný výsledek
- nemusí však dát vůbec žádný výsledek
 - » je-li procedurálně nesprávný, může dojít k zacyklení

☀ Příklad genealogický: předek/2

```
% predek(?Predek,?Potomek) :- Predek je předkem  
%                               Potomka.
```

```
predek(X,Y) :- rodic(X,Y).
```

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

predek/2 je deklarativně i procedurálně správný

```
% predek2(?Predek,?Potomek) :- jiná varianta.
```

```
predek2(X,Z) :- predek2(Y,Z), rodic(X,Y).
```

```
predek2(X,Y) :- rodic(X,Y).
```

predek2/2 se zacyklí

☀ Příklad genealogický: předek/2

```
predek3(X,Z) :- rodic(X,Y), predek3(Y,Z).
```

```
predek3(X,Y) :- rodic(X,Y).
```

predek3/2 je deklarativně i procedurálně správný

```
predek4(X,Y) :- rodic(X,Y).
```

```
predek4(X,Z) :- predek4(Y,Z), rodic(X,Y).
```

predek4/2 najde všechna řešení, pak se zacyklí

Aritmetika

?- $X = 1+1.$

?- $X \text{ is } 1+1.$

Vestavěný predikát **is** / 2

- definovaný jako operátor
- ?- $\text{display}(X \text{ is } 1+1).$
- $\text{is}(X, +(1, 1))$

Operátor is/2

S is T

- term T je vázán na aritmetický výraz
- hodnota T je vyhodnocena jako aritmetický výraz
- výsledek je unifikován s termem S
- term T **není** vázán na aritmetický výraz \Rightarrow **chyba**

Vestavěný systémový predikát

- nepatří do čistého Prologu

Aritmetické operátory

$+$, $-$, $*$, $/$, $//$, $^$, mod

- ? – $x \text{ is } 2^3 \text{ mod } 5$.

Relační: $>/2$, $</2$, $>=/2$, $=</2$

Rovnost a nerovnost: $==$, \neq

- vyhodnocení operandů, porovnání výsledků
- operand **není** vázán na aritmetickou hodnotu \Rightarrow **chyba**
- ? – $1+2 == 2+1$.

Aritmetické operátory

- jsou deterministické
- “nebacktrackují”

Aritmetické funkce

Lze použít v aritmetických výrazech

- $\text{max}/2$, $\text{min}/2$, $\text{abs}/1$...
- $\text{sin}/1$, $\text{cos}/1$, $\text{tan}/1$, $\text{sqrt}/1$, $\text{log}/1$...
» ?- X is $4*\text{asin}(\text{sqrt}(2)/2)$.
- bitové operace: $\wedge/2$, $\vee/2$, $\text{xor}/2$, $\gg/2$, $\ll/2$...
- vrací aritmetickou (nikoliv logickou) hodnotu

Jednoduché aritmetické predikáty

`% max(+X,+Y,?Max) :- Max je maximum z čísel X a Y.`

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

`max(X,Y,Y).`

Jaké budou odpovědi na následující dotazy?

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

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

`true.`

chyba !!!

Korektní verze

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

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

Jednoduché aritmetické predikáty

```
% mezi(+X,+Y,-Z):- Postupně vrátí celá čísla  
%                       splňující  $X \leq Z \leq Y$ .
```

```
mezi(X,Y,X) :- X =< Y.
```

```
mezi(X,Y,Z) :- X<Y, NoveX is X+1, mezi(NoveX,Y,Z).
```

```
?- mezi(1,3,Z).
```

- $Z = 1$;
- $Z = 2$;
- $Z = 3$.

Předdefinován jako standardní predikát `between/3`

Délka seznamu

`% delka(+Xs,?N) :- N je počet prvků seznamu Xs.`

`delka([],0).`

`delka([_|Xs],N) :- delka(Xs,N1), N is N1+1.`

☞ Alternativní řešení


- koncová rekurze
- akumulátor

Délka seznamu s akumulátorem

```
delkaAk (Xs, N) :- delkaAk (Xs, 0, N) .
```

```
delkaAk ([ ], A, A) .
```

```
delkaAk ([_ | Xs], A, N) :- A1 is A+1, delkaAk (Xs, A1, N) .
```

 Jaké odpovědi obdržíme na dotazy

- `?- delkaAk (Xs, 3) .`
- `?- delkaAk (Xs, N) .`

 **Problém:** Jak vygenerovat seznam dané délky ?

Předdefinován jako standardní predikát

```
length ( ?Xs, ?N )
```

Třídění sléváním: Mergesort

```
% mergesort(+Xs,-Ys):- Ys je vzestupně seřazený  
% seznam Xs.
```

```
mergesort([],[]).
```

```
mergesort([X],[X]).
```

```
mergesort(Xs,Ys) :- Xs = [_,_|_],  
                    split(Xs,Xs1,Xs2),  
                    mergesort(Xs1,Ys1),  
                    mergesort(Xs2,Ys2),  
                    merge(Ys1,Ys2,Ys).
```

Třídění sléváním: rozdělení

```
% split(+Xs,-Ys,-Zs):- Ys a Zs jsou seznamy lichých  
%                       a sudých prvků seznamu Xs.  
  
split([],[],[]).  
split([X],[X],[]).  
split([X,Y|Zs],[X|Xs],[Y|Ys]) :- split(Zs,Xs,Ys).
```


Třídění sléváním: slévání

```
% merge(+Xs,+Ys,-Zs) :- sloučí uspořádané seznamy  
%.                Xs a Ys do uspořádaného seznamu Zs.  
  
merge([ ],Ys,Ys).  
merge([X|Xs],[ ],[X|Xs]).  
merge([X|Xs],[Y|Ys],[X|Zs]) :- X <= Y,  
                                merge(Xs,[Y|Ys],Zs).  
merge([X|Xs],[Y|Ys],[Y|Zs]) :- X > Y,  
                                merge([X|Xs],Ys,Zs).
```

Třídění: Quicksort

```
% quicksort(+Xs,-Ys):- Ys je vzestupně setříděný  
% seznam Xs.
```

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

```
quicksort([X|Xs],S) :- qsplit(X,Xs,Ys,Zs),  
                        quicksort(Ys,YsS),  
                        quicksort(Zs,ZsS),  
                        append(YsS,[X|ZsS],S).
```

Quicksort: rozdělení

```
% qsplit(+Pivot,+Xs,-Ys,-Zs):- Ys a Zs jsou seznamy  
%                               prvků <=Pivot a >Pivot seznamu Xs.  
  
qsplit(_,[],[],[]).  
qsplit(P,[X|Xs],[X|Ys],Zs):- X <= P,  
                               qsplit(P,Xs,Ys,Zs).  
qsplit(P,[X|Xs],Ys,[X|Zs]):- X > P,  
                               qsplit(P,Xs,Ys,Zs).
```

Třídění termů

Standardní predikát `sort/2`

- `sort(+Seznam, ?UsporadanySeznam)`
- `Seznam` je seznam libovolných termů
- setřídí a odstraní duplicity
- `msort/2` setřídí, duplicity zachová

Standardní pořadí termů

- `proměnné` < `čísla` < `atomy` < `složené termy`
- `proměnné`: dle adresy
- `atomy`: lexikograficky
- `složené termy`: arita, funktor, argumenty zleva doprava
- standardní operátory `@</2`, `@>/2`, `@=</2`, `@>=/2`

Programování s omezujícími podmínkami

`?- use_module(library(clpfd)).`

- Constraint Logic Programming over Finite Domains

Aritmetické operátory

- `#=` `#\=` `#<` `#>` `#=<` `#>=`

Příklady

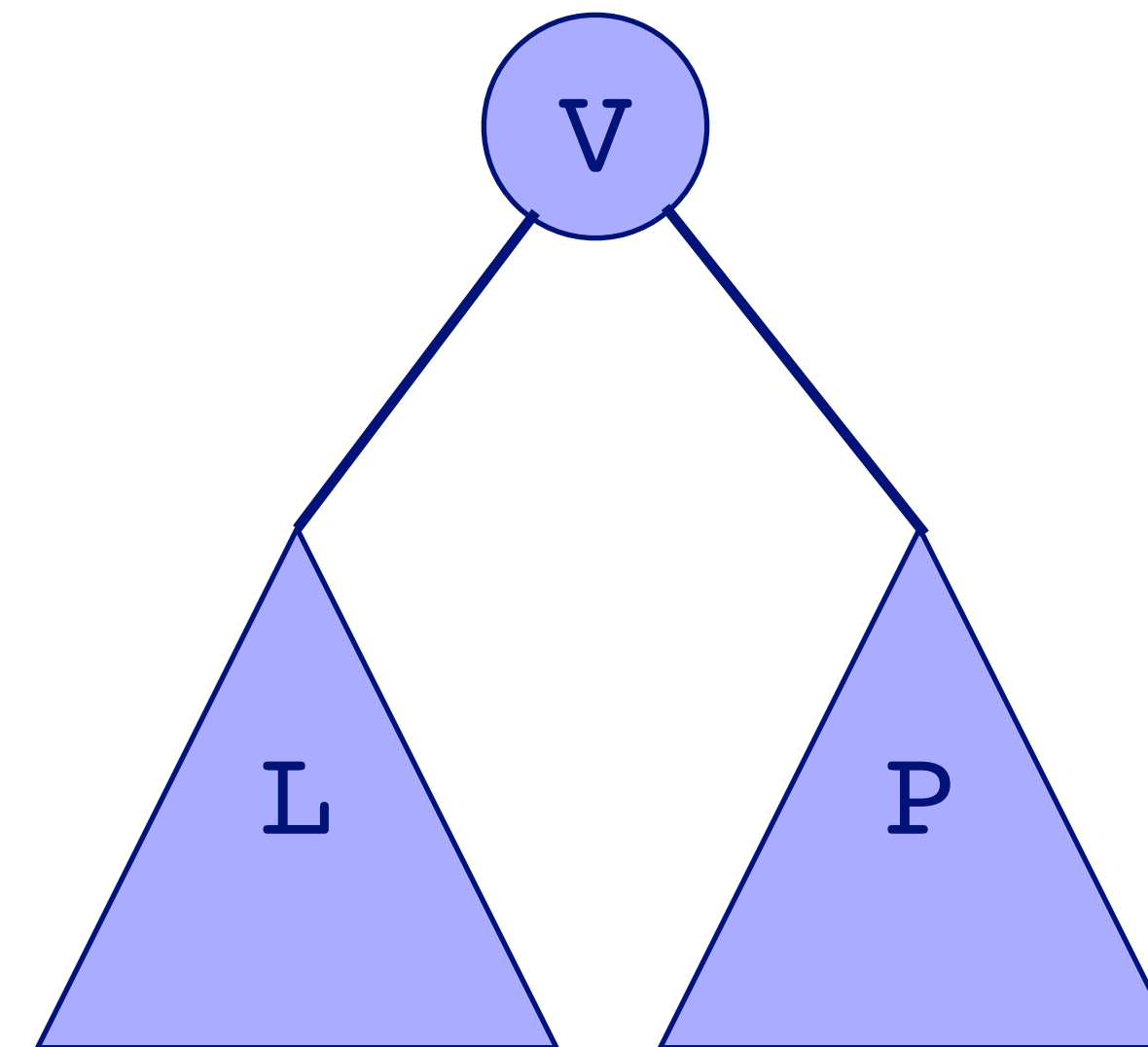
- `?- X #= 3+2.` `?- 3+2 #= X.`
- `?- X*2 #= 9.` `?- X #\= 3.`
- `?- X in 1..10, X^2 #= 9.`
- `?- 2 #= X+3.` `?- Y #= X+3, Y+1 #= 1.`
- `?- Y #= X+3, Y+X #= 1.` `?- X^2 - X #= 2.`

Binární stromy

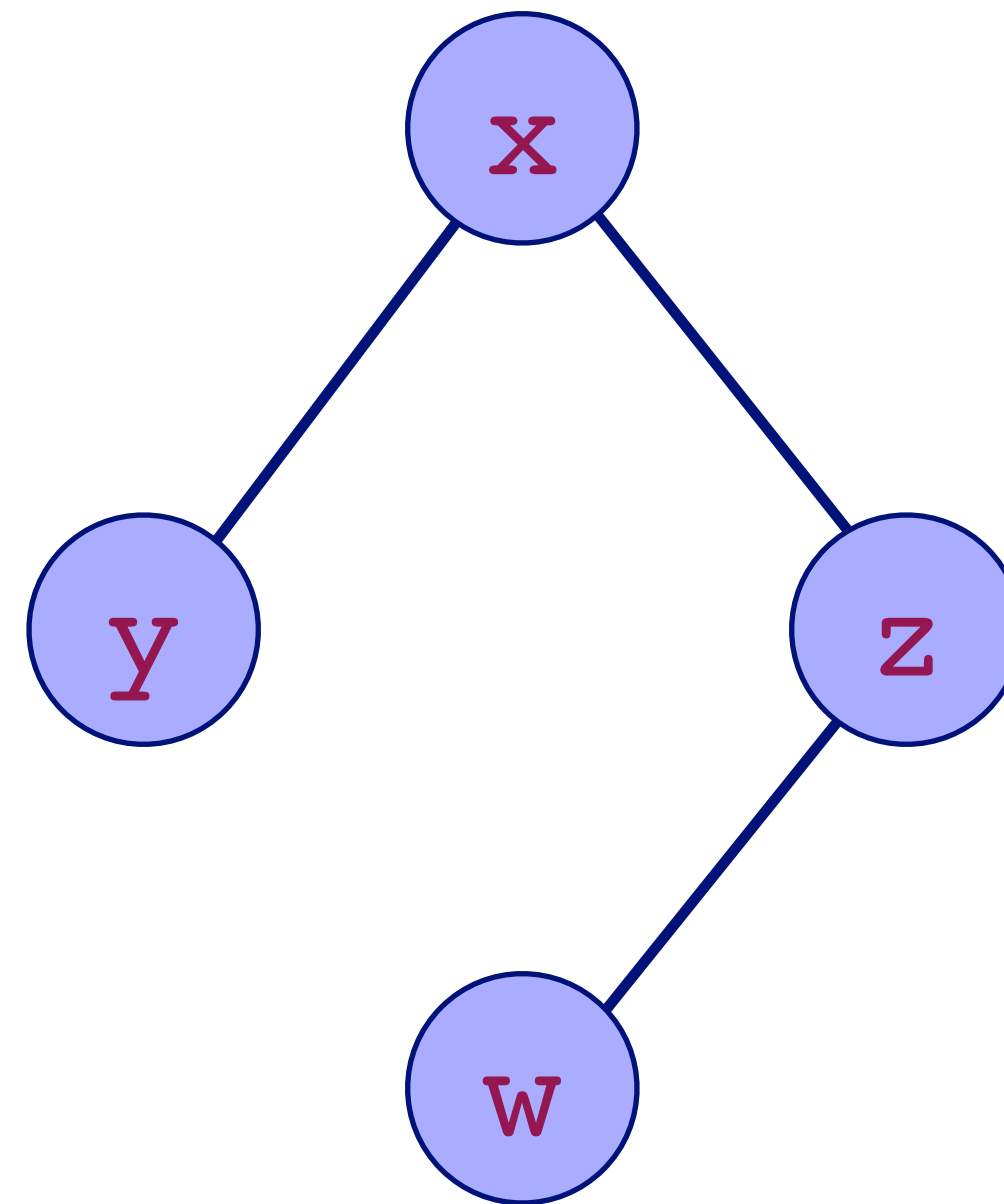
Prázdný strom: atom `nil`

Neprázdný strom: složený term `b(L, V, P)`

- `L` je levý podstrom
- `V` je vrchol, který je kořenem
- `P` je pravý podstrom

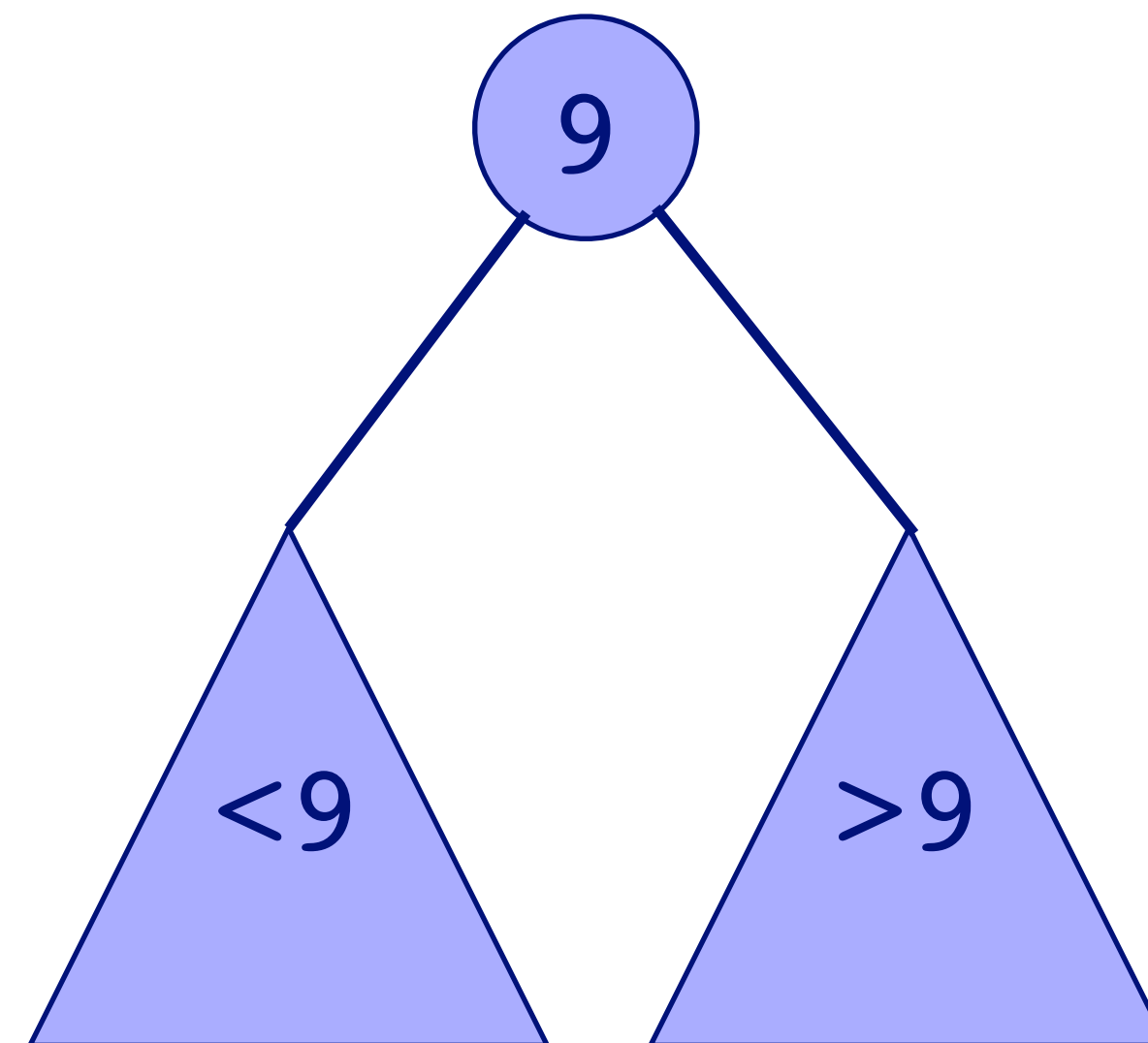


☀ Příklad binárního stromu

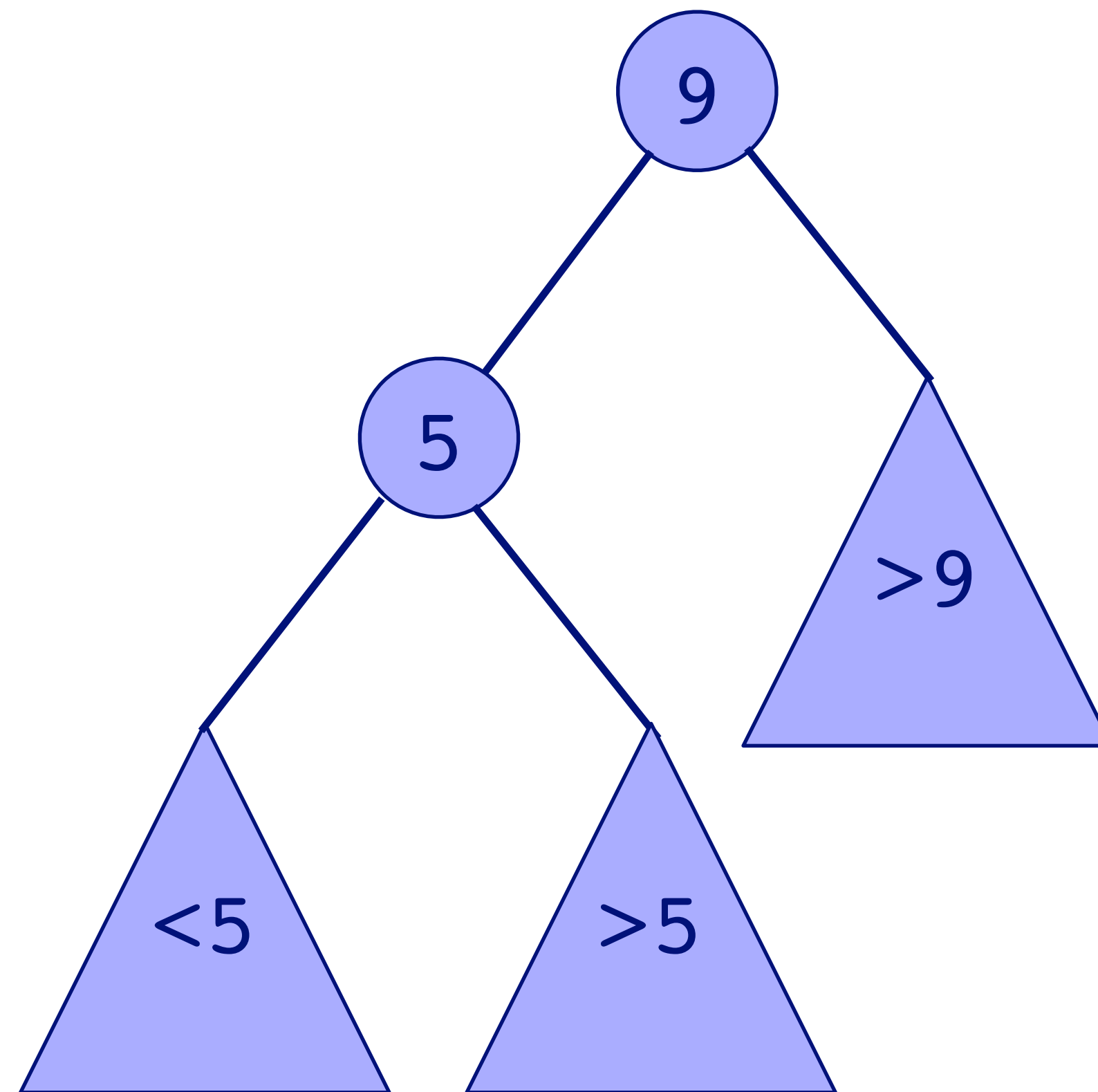


```
b(b(nil, y, nil),  
  x,  
  b(b(nil, w, nil), z, nil)  
)
```

Binární vyhledávací stromy



Binární vyhledávací stromy



Binární vyhledávací stromy: je prvkem?

```
% search(+X,+B) :- X je prvkem binárního  
%                  vyhledávacího stromu B.  
  
search(X, b(_,X,_)).  
search(X, b(L,K,_)) :- X<K, search(X,L).  
search(X, b(_,K,P)) :- X>K, search(X,P).
```

Binární vyhledávací stromy: vlož

```
% insert(+X,+B,-B1) :- B1 vznikne vložení X  
%                               do BVS B.
```

```
insert(X, nil, b(nil,X,nil)).
```

```
insert(X, b(L,X,P), b(L,X,P)).
```

bez duplicit

```
insert(X, b(L,V,P), b(L1,V,P)) :- X < V,  
                                insert(X,L,L1).
```

```
insert(X, b(L,V,P), b(L,V,P1)) :- X > V,  
                                insert(X,P,P1).
```

Problém

- lze proceduru využít též k vypuštění prvku **X** z **B**
- dotazem typu **insert(+X,-B1,+B)** ?

Binární vyhledávací stromy: vypust'

```
% del(+X,+B,-B1) :- B1 vznikne vypuštěním X z BVS B.  
del(X, b(nil,X,P), P).  
del(X, b(L,V,P), b(L1,V,P)) :- X < V,  
                                del(X, L, L1).  
del(X, b(L,V,P), b(L,V,P1)) :- X > V,  
                                del(X, P, P1).  
del(X, b(L,X,P), b(L1,Y,P)) :- L \= nil,  
                                delmax(L, L1, Y).
```

Binární vyhledávací stromy: vypust'

```
% delmax(+B,-B1,-Y) :- B1 vznikne vypuštěním  
%                       maximálního prvku Y z BVS B.  
  
delmax(b(L,X,nil),L,X).  
delmax(b(L,X,P),b(L,X,P1),Y):- P \= nil,  
                                delmax(P,P1,Y).
```


Problémy se stromy

- ❶ Definujte základní operace nad **binární haldou**, tj.
 - vložení prvku
 - odstranění kořene
- ❷ Navrhněte reprezentaci pro obecné (kořenové) stromy.

Predikáty vyšších řádů

`% call(Cil,X,...):- zavolá Cil s argumenty X,...`

`call(member,b,[a,b,c])`

- `member(b,[a,b,c])`

`call(member(b),[a,b,c])`

`call(plus,1,2,Z)`

- `plus(1,2,Z)`

`call(plus(1),2,Z)`

`call(plus(1,2),Z)`

Predikáty vyšších řádů: maplist/2

```
% maplist(+Predikat,?Xs) :- uspěje, pokud (unární)
%      Predikat uspěje na všech prvcích seznamu Xs.

maplist(_, []).
maplist(P, [X|Xs]) :- call(P,X), maplist(P,Xs).

?- maplist(muz, [adam,kain,abel]).
true.

?- length(Xs,2), maplist(muz,Xs).
```

maplist/3

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

```
maplist(P,[X|Xs],[Y|Ys]):- call(P,X,Y),  
                             maplist(P,Xs,Ys).
```

```
?- maplist(<,[1,2,3],[4,5,6]).  
true.
```

```
?- maplist(length,[[1,2,3],[a,b]],Ls).  
Ls = [3,2].
```

```
?- maplist(length,Xss,[3,2]).  
Xss = [[_,_,_],[_,_]].
```

```
?- maplist(plus(1),[1,2,3],Ys).  
Y = [2,3,4].
```

maplist/4

```
?- maplist(plus,[1,2,3],[10,20,30],Zs).
```

```
    Zs = [11,22,33].
```

```
?- help(maplist).
```

Predikáty vyšších řádů: foldl/4

```
% foldl(P, [X1, ..., Xn], V0, V) :-    úspěje, pokud úspěje  
%                                     P(X1, V0, V1),  
%                                     P(X2, V1, V2),  
%                                     ... ,  
%                                     P(Xn, Vn-1, V) .
```

```
foldl(_, [], V, V) .
```

```
foldl(P, [X|Xs], V0, V) :- call(P, X, V0, V1),  
                           foldl(P, Xs, V1, V) .
```

```
?- foldl(plus, [1, 2, 3, 4, 5, 6], 0, V) .  
V = 21 .
```

☀ Příklad: reverse/2 pomocí foldl

```
seznam(X,Xs,[X|Xs]).    % vytvoří seznam
```

```
obrat(Xs,Ys):- foldl(seznam,Xs,[],Ys).
```

✎ **Problém:** Definujte predikát pro výpočet **délky seznamu** pomocí **foldl**.