FUNKCIONÁLNÍ A LOGICKÉ PROGRAMOVÁNÍ
10. PROGRAMOVÁNÍ V JAZYKU PROLOG – POSTUP VYHODNOCOVÁNÍ DOTAZŮ, ARITMETIKA, ŘÍZENÍ VÝPOČTU, SLOŽITĚJŠÍ PŘÍKLADY.

2011 Jan Janoušek
MI-FLP

Basic rules for writing Prolog programs

Query evaluation in Prolog

Program in Prolog can be seen as a database of facts and rules.

Two sources of nondeterminism during searching and unification :

- Which rule is to be used? Depth-first search.
- Which subgoal is to be used? From left to right.

Recursive definitions

- Nonrecursive:

dědeček(U,Z):-muž(U), rodič(U,V), rodič(V,Z).

pradědeček(A,B):-muž(A), rodič(A,C), rodič(C,D), rodič(D,B).

rod_linie(P,PP):-rodič(P,PP).

rod_linie(P,PP):-dědeček(P,PP).

rod_linie(P,PP):-pradědeček(P,PP)...

- Recursive:

rod_linie(P,PP):-rodič(P,PP).

rod_linie(P,PP):-rodič(P,P1),rod_linie(P1,PP).

Example – finding a way in graph

hrana(a,b). hrana(a,c). hrana(c,d). hrana(c,e).

4 possibilities how to find a way between two nodes in the graph:

```
c1(Y,Y).
c1(Z,K):-hrana(Z,M),c1(M,K).
c2(Y,Y).
c2(Z,K):-c2(Z,M),hrana(M,K).
c3(Z,K):-hrana(Z,M),c3(M,K).
c3(Y,Y).
c4(Z,K):-c4(Z,M),hrana(M,K).
c4(Y,Y).
```

Which is the best?

General rules

Y.6

- We go from the simplest to the most complicated:
  - Facts.
  - Rules without recursion.
  - Rules with recursion (tail recursion is preferred).

```
c1(Y,Y).
c1(Z,K):-hrana(Z,M),c1(M,K).
```

Comments

- Line comments:

% this is a line comment.

- Block comments:

/* This is
a block comment */

- Comments of predicates:

% predek(?predek,?dite).

  - +X – input parameters
  - -X – output parameters
  - ?X – both input and output parameters

Parameters can be used both as input and output parameters

- % matka(?X,?Y). X is mother of Y.

?-matka(ludmila,X).

?-matka(X,vaclav).

- % spojeni(?S1,?S2, ?Vysledek). Lists S1 and S2 are joined to Vysledek.

?-spojeni([1,2,3],[4,5],X).Yes, X=[1,2,3,4,5]

?-spojeni([1,2,3],Z,[1,2,3,4,5]) Yes, Z=[4,5].

?-spojeni(X, Z, [1,2,3,4,5]). Yes, X=[], Y=[1,2,3,4,5]

Yes, X=[1], Y=[2,3,4,5]

Yes, X=[1,2], Y=[3,4,5], ...

Arithmetics

```
?- X=1+2.
X = 1+2
```

```
?- X is 1+2.
X = 3
```

number comparison `>=   =<   >   <   =:=   =\=`

aritmetic operations `+   -   *   /   //   mod` ...

`length(S, N)`	list length
`gcd(X, Y, D)`	greatest common divisor
`max_elm(S, Max)`	maximal element
`pack(List, Sum, Sel)`	select from List into Sel to give Sum

Operator cut (řez)

Cut - !

- Non-logical predicate !
- ! is always succeeded. The evaluation goes through.
- ! does not allow backtracking.

- Mějme program obsahující následující 3 klauzule

```
1.b(X,Y):-e(X,Y), f(Y), !, g(X), h(Y,Z).
2.b(X,Y):-k(X,Y).
3.p(X):-a(X), b(X,Y), c(Y), d.
```

- Řez fixuje přijaté „částečné řešení"– omezuje splnění podcílů vlevo od řezu (e & f) na jedinou možnost.
- Překročení řezu zamezí využití ostatních pravidel. Uvažme např., že platí (e & f) pro nějaké hodnoty X a Y. V takovém případě se při dotazu ?-b. nikdy nedostaneme ke zjišťování, zda platí k.
- Řez neovlivňuje zpětný chod vpravo do svého výskytu, t.j. mezi cíli g a h.

- Sestrojme proceduru, která vloží do seznamu prvek jen v tom případě, že tam již nebyl:

```
%  pridej(+X,+L,-NL) seznam NL vznikne ze seznamu L
% přidání prvku X na jeho začátek ovšem jen
% v tom případě, že X v L již není %

pridej(X,L,L):- prvek(X,L),      % je-li X již prvkem L, nepřidám ho
!.                                % a zakáži návrat
pridej(X,L,[X|L]).               % X není prvkem L (jinak bych se
                                  % sem nedostal), mohu ho tedy přidat
```

Operators of negation fail, not

Operator fail

"Jana má ráda muže, ale ne plešaté" .

Bez operátoru řezu to nejde. S ním a se standardním predikátem fail, který, je-li volán, okamžitě selže, ji sestavíme poměrně snadno:

```
marada(jana,X):- plesaty(X),  % je-li X plešaté uspěje,
!,                            % zakáže návrat
fail.                         % a selže.

marada(jana,X):- % k této klauzuli se výpočet dostane, pokud X není plešaté,
        muz(X).      % je-li to muz, má ho Jana ráda
```

Operator not

```
% not(P) uspěje, pokud se nepodaří cíl P splnit

not(P) :- P, ! , fail.
not(P).
```

Some predicate definitions

Some predicate definitions - 1


```
%member(Elm, List)                % test for element, top level

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

%memberall(Elm, List)             % test on all levels

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

Some predicate definitions - 2


```
%reverse(List, RevL)   % reversing a list (using tail recursion)
reverse(X,RX) :- rev1(X,[],RX).
rev1([],RX,RX).
rev1([X|Rest],Acc,RX) :- rev1(Rest,[X|Acc],RX).

delete(Elm,List,Result)       % delete first occurrence
delete(X,[X|Rest],Rest).
delete(X,[Y|Rest],[Y|DRest]) :- delete(X,Rest,DRest).

insert(Elm,List,Result)       % insert an element
insert(Elm,List,Result) :- delete(Elm,Result,List).

% or directly mimicking the definition of delete
insert(X,L,[X|L]).
insert(X,[Y|L],[Y|LX]) :- insert(X,L,LX).
```

Some predicate definitions - 3


```
%perm(S,P)                          % generate a list permutation

% a nice example of declarative thinking

perm([],[]).
perm([X|Rest],P) :- perm(Rest,PRest), insert(X,PRest,P).

% or another way round ...

perm([],[]).
perm(S,[X|P]) :- delete(X,S,Rest), perm(Rest,P).

length(S, N)                        % list length
length([],0).
length([X|Rest],N) :- length(Rest,N1), N is N1+1.

gcd(X, Y, D)                        % greatest common divisor
gcd(0,Y,Y).
gcd(X,Y,R) :- X >= Y, !, X1 is X-Y, gcd(X1,Y,R).
gcd(X,Y,R) :- X < Y, !, Y1 is Y-X, gcd(Y1,X,R).
```

Some predicate definitions - 4


```
%max_elm(S, Max)                % maximal element
max_elm([X],X).
max_elm([X,Y|R],M) :-
  max_elm([Y|R],Mx), (X > Mx, !, M=X ; M=Mx).

%pack(List, Sum, Sel)  % select from List into Sel to give Sum
pack(_,0,[]).
pack([X|Rest],S,[X|R]) :- S >= X, S1 is S-X, pack(Rest,S1,R).
pack([X|Rest],S,R) :- pack(Rest,S,R).

% or if we permit repetitive selection from List
pack([X|Rest],S,[X|R]) :-
 S >= X, S1 is S-X, pack([X|Rest],S1,R).
```

Some built-in predicates

Some Built-in Prolog Predicates

- **Loading Prolog programs**

consult(F) - loads program from the file F

reconsult(F) - like consult except that each predicate already defined has its definition replaced by the new defintion being loaded

- **Debugging tools**

help(S) - gives help on a symbolic atom, e.g., help(see)

halt - stops Prolog

trace, notrace - turns tracing on and off, resp.

- **Controls**

true, fail - always succeeds/fails as a goal

call(P) - forces P to be a goal; succeeds if P does, else fails

! **-** Prolog cut

repeat - succeeds any number of times

not(Q), \+Q - negation as failure og Q

- **Testing**

atom(X) - succeeds if X is bound to a symbolic atom

integer(X) - succeeds if X is bound to an integer

atomic(X) - succeeds if X is bound to a symbolic atom or number

compound(X) - succeeds if X is bound to a compound term

float(X) - succeeds if X is bound to a real number

string(X) - succeeds if X is bound to a string

ground(G) - succeeds if G has unbound variables

var(X) - succeeds if X is an uninstantiated variable

nonvar(X) - succeeds if X is an instantiated variable

- **Input/output**

seeing(X) - succeeds if X is (or can be) bound to current read port. X=user is keybord input

see(X) - opens port for input file bound to X so that input for 'read' is then taken from that port

seen - closes any selected input port/file, and causes 'read' to look at user

read(X) - reads into XProlog type expression from current port

telling(X) - succeeds if X is (or can be) bound to current output port X=user is screen

tell(X) - opens port for output file bound to X so that output from 'write' or 'display' is sent to that port

told - closes any selected output port/file and reverts to screen output

write(E) - writes Prolog expression bound to E into current output port

nl - next line (line feed).

tab(N) - write N spaces to selected output port

- **Terms and clauses**

clause(H,B) - retrieves clauses in memory whose head matches H and body matches B

functor(E,F,N) **-** E must be bound to a functor expression of the form 'f(...)'. F will be bound to 'f', and N will be bound to the number of arguments that f has

arg(N,E,A) - E must be bound to a functor expression, N is a whole number, and A will be bound to the Nth argument of E (or fail)

=.. - 'univ' converts between term and list

asserta(C) - assert clause C into database above other clauses with the same predicate

assertz(C), assert(C) - assert clause C into database below other clauses with the same predicate.

retract(C) - retract clause C from the database

- **Special**

findall(T,G,L)- finds all solutions of G, instantiates variables of T to the values they have in that solution and adds that instantiation of T to L

bagof(T,G,L) - like findall, but with free variables in G existentially quantified

setof(T,G,L) - like bagof but terms in L sorted alphabetically and duplicates removed

Simulation of iterative cycles

Repeat


```
% repeat nulární predikát okamžitě uspěje
% a uspěje vždy i při návratu

repeat.
repeat:- repeat.
```

```
% vstup přečte ze vstupu jeden term
% pokud to není přirozené číslo menší než 100,
% opakuje výzvu a čtení

vstup:- repeat,
write('Zadej přizené číslo menší než 100:'), % výzva
read(N),                    % přečtení termu ze vstupu
integer(N), % uspěje je-li term N celé číslo (standardní predikát)
N>0,                        % uspěje, je-li číslo, které vstoupilo
N<100,                      % větší než 0 a menší než 100
! .
```

The conclusion: our motivating example and its solution in Prolog

Einstein riddle

1. In a street there are five houses, painted five different colours.

2. In each house lives a person of different nationality.

3. These five homeowners each drink a different kind of beverage, smoke different brand of cigar and keep a different pet.

Question: **Who owns the fish ?**

Hints:

1.The Brit lives in a red house.

2.The Swede keeps dogs as pets.

3.The Dane drinks tea.

4.The Green house is next to, and on the left of the White house.

5.The owner of the Green house drinks coffee.

6.The person who smokes Pall Mall rears birds.

7.The owner of the Yellow house smokes Dunhill.

8.The man living in the centre house drinks milk.

9.The Norwegian lives in the first house.

10.The man who smokes Blends lives next to the one who keeps cats.

11.The man who keeps horses lives next to the man who smokes Dunhill.

12.The man who smokes Blue Master drinks beer.

13.The German smokes Prince.

14.The Norwegian lives next to the blue house.

15.The man who smokes Blends has a neighbour who drinks water.

Program in Prolog

- The Nationalities are: brit, swede, dane, norwegian, german.
- The Colors are: red, green, white, yellow, blue.
- The Beverages are: tea, coffee, milk, beer, water.
- The Cigars are: pallmall, dunhill, blend, bluemaster, prince.
- The Pets are: fish, dog, bird, cat, horse.

As the main data structure we use a list of five elements for the particular houses. Each of these lists contains five values (nationality,color,drink,smoke,animal).

- Constructor of the initial list:

```
% persons(+N,-R). Creating a list of N lists of 5 elements.

persons(0, []) :- !.
persons(N, [(_Men,_Color,_Drink,_Smoke,_Animal)|T]) :-
                                        N1 is N-1, persons(N1,T).

? – persons(5,R).
R = [(_,_,_,_,_),(_,_,_,_,_),(_,_,_,_,_),(_,_,_,_,_),(_,_,_,_,_)]
```

- Auxilliary selector of the n-th element:

```
% person(+N,+L,-R). Returns the N-th element form the list L.

person(1, [H|_], H) :- !.
person(N, [_|T], R) :- N1 is N-1, person(N1, T, R).

?- person(2,[a,b,c],V).
V = b
```

- The hints are translated into predicates:

```
% The Brit lives in a red house
hint1([(brit,red,_, _, _)|_]).  % the valid value
hint1([_|T]) :- hint1(T).    % the iteration to iterate over the list elements.
% the predicate is true when the the list contains the valid value

% The Swede keeps dogs as pets
hint2([(swede,_,_,_,dog)|_]).
hint2([_|T]) :- hint2(T).

% The Dane drinks tea
hint3([(dane,_,tea,_,_)|_]).
hint3([_|T]) :- hint3(T).
```

```
% The Green house is on the left of the White house
hint4([(_,green,_,_,_),(_,white,_,_,_)|_]).
hint4([_|T]) :- hint4(T).

% The owner of the Green house drinks coffee.
hint5([(_,green,coffee,_,_)|_]).
hint5([_|T]) :- hint5(T).

% The person who smokes Pall Mall rears birds
hint6([(_,_,_,pallmall,bird)|_]).
hint6([_|T]) :- hint6(T).

% The owner of the Yellow house smokes Dunhill
hint7([(_,yellow,_,dunhill,_)|_]).
hint7([_|T]) :- hint7(T).
```

```
% The man living in the centre house drinks milk
hint8(Persons) :- person(3, Persons, (_,_,milk,_,_)).

% The Norwegian lives in the first house
hint9(Persons) :- person(1, Persons, (norwegian,_,_,_,_)).

% The man who smokes Blends lives next to the one who keeps cats
hint10([(_,_,_,blend,_),(_,_,_,_,cat)|_]).
hint10([(_,_,_,_,cat),(_,_,_,blend,_)|_]).
hint10([_|T]) :- hint10(T).
```

% The man who keeps horses lives next to the man who smokes Dunhill

```
hint11([(_,_,_,dunhill,_),(_,_,_,_,horse)|_]).
hint11([(_,_,_,_,horse),(_,_,_,dunhill,_)|_]).
hint11([_|T]) :- hint11(T).
```

% The man who smokes Blue Master drinks beer

```
hint12([(_,_,beer,bluemaster,_)|_]).
hint12([_|T]) :- hint12(T).
```

% The German smokes Prince

```
hint13([(german,_,_,prince,_)|_]).
hint13([_|T]) :- hint13(T).
```

```
% The Norwegian lives next to the blue house
hint14([(norwegian,_,_,_,_),(_,blue,_,_,_)|_]).
hint14([(_,blue,_,_,_),(norwegian,_,_,_,_)|_]).
hint14([_|T]) :- hint14(T).

% The man who smokes Blends has a neighbour who drinks water
hint15([(_,_,_,blend,_),(_,_,water,_,_)|_]).
hint15([(_,_,water,_,_),(_,_,_,blend,_)|_]).
hint15([_|T]) :- hint15(T).
```

- The question:

% We just iterate the list, specifying that there is a man with a fish.

```
question([(_,_,_,_,fish)|_]).
question([_|T]) :- question(T).
```

- % solution(-L). The solution must validate all the created predicates.

```
solution(Persons) :-
  persons(5, Persons),
  hint1(Persons),
  hint2(Persons),
  hint3(Persons),
  hint4(Persons),
  hint5(Persons),
  hint6(Persons),
  hint7(Persons),
  hint8(Persons),
  hint9(Persons),
  hint10(Persons),
  hint11(Persons),
  hint12(Persons),
  hint13(Persons),
  hint14(Persons),
  hint15(Persons),
  question(Persons).
```

Run of our program

```
?- solution(Persons).

Persons=[(norwegian,yellow,water,dunhill,cat),(dane,blue,tea,blen
d,horse),(brit,red,milk,pallmall,bird),(german,green,coffee,prince,fi
sh),(swede,white,beer,bluemaster,dog)] ? ;

no
```