http://www.dcs.fmph.uniba.sk/~plachetk

/TEACHING/RLDB2006

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Leto 2006–2007

Literatúra

- Leon Sterling, Ehud Shapiro: The Art of Prolog, MIT Press, 1986
- Pierre M. Nugues: An Introduction to Language Processing with Perl and Prolog, Springer Verlag, 2006, Appendix A, “An Introduction to Prolog”, http://www.cs.lth.se/home/Pierre?Nugues/ilppp/chapters/appA.pdf

- WWW: “+prolog +tutorial”
- Ukážková aplikácia: http://www.zillions-of-games.com

História

- **Herbrand (1930), Turing (1940), Robinson (1965): automatické dokazovanie teorém**
- **Colmerauer (1970): prvá implementácia Prologu**
- **Deransart, ISO standard (1995)**

… a mnohí ďalší

- **Logické (deklaratívne) programovanie, založené na logike prvého rádu**

Základné súčasti Prologu: fakty (EDB predikáty)

```
character(achilles, illiad).
character(menelaus, illiad).
character(helen, illiad).
character(menelaus, odyssey).
character(helen, odyssey).
male(achilles).
male(menelaus).
female(helen).

uinverse(_).
```

predikát / arita.

character/2

male/1

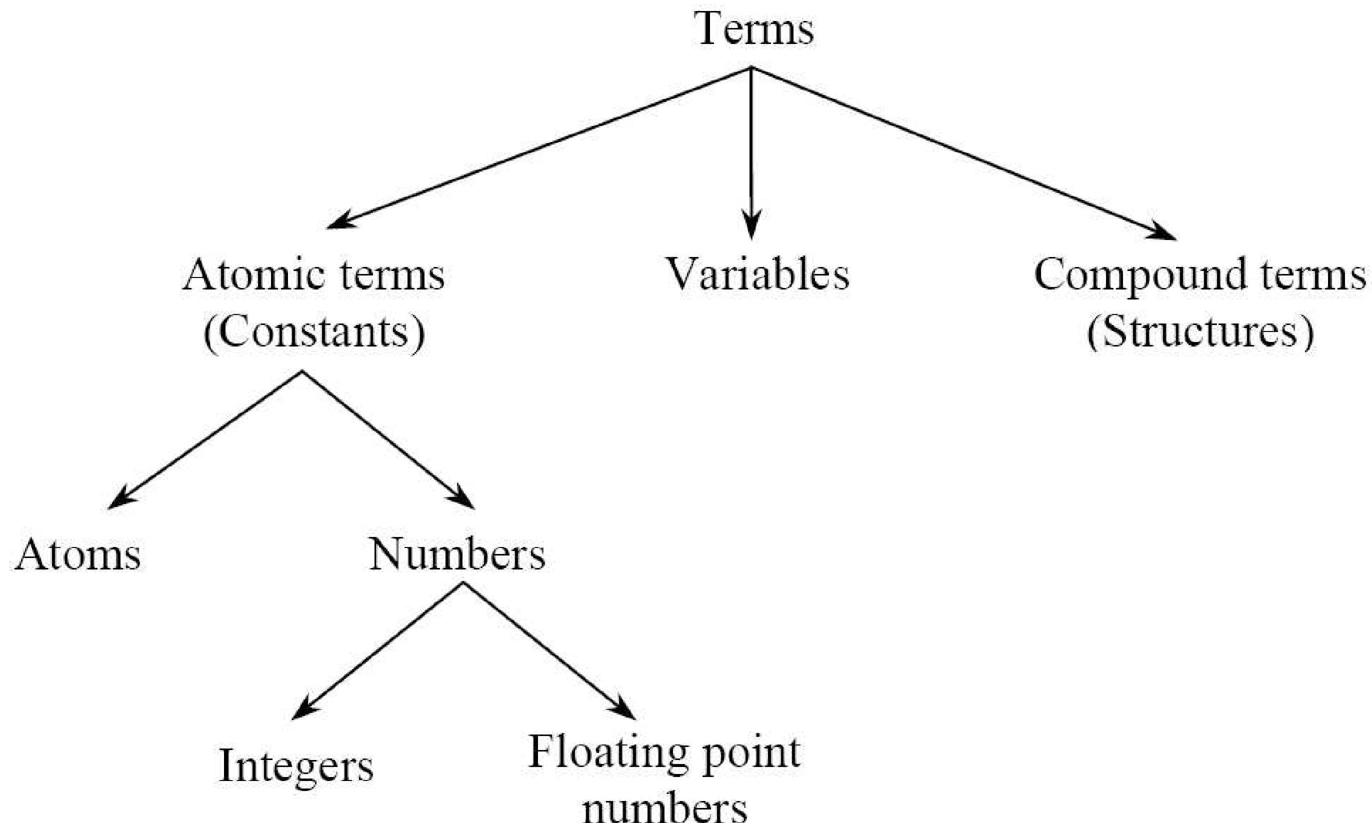
female/1

Atómy sú predikáty arity 0

Základné súčasti Prologu: všetky dáta sú termy

role(menelaus, iliad, king(sparta, menelean)).

Na skladanie termov do štruktúr sa používajú funkčné symboly (functor). Predikáty, resp. funktory rôznej arity sú rôzne, aj keď majú rovnaké meno: overloading



Základné súčasti Prologu: pravidlá (clauses)

HEAD :- $GOAL_1$, $GOAL_2$, ..., $GOAL_N$.

Pravá strana (body) je konjunkciou podcieľov

V jednom pravidle sa dá zapísať aj disjunkcia, ale nedporučuje sa (disjunkcia sa zapíše ako viac pravidiel s rovnakou hlavou):

HEAD :- $GOAL_1$; $GOAL_2$.

Sú aj pravidlá s prázdnou hlavou, tzv. directives, ktoré sa vykonajú práve raz, pri parsovaní, napr.

:- dynamic(dodava / 2).

Dotazy sú syntakticky identické s pravou stranou pravidiel („anonymné pravidlá“)

Interpretácia programov: unifikácia

Kľúčovým algoritmom je unifikácia dvoch termov (Herbrand)

Initialization step

Initialize σ to $\{\}$

Initialize `failure` to `false`

Push the equation $T_1 = T_2$ on the stack

Loop

repeat {

pop $x = y$ from the stack

if x is a constant and $x == y$. Continue.

~~else if x is a variable and x does not appear in y.~~ no occurs-check

~~Replace x with y in the stack and in σ. Add the substitution $\{x = y\}$ to σ.~~

else if x is a variable and $x == y$. Continue.

else if y is a variable and x is not a variable.

Push $y = x$ on the stack.

else if x and y are compounds with $x = f(x_1, ..., x_n)$ and $y = f(y_1, ..., y_n)$.

Push on the stack $x_i = y_i$ for i ranging from 1 to n.

else Set `failure` to `true`, and σ to $\{\}$. Break.

} until (stack $\neq \emptyset$)

Interpretácia programov: unifikácia

Absencia occurs-check nie je príliš citeľná, ale occurs-check sa dá efektívne implementovať (aj keď kedysi sa verilo že nie), takže tam byť má. Súčasný Prolog ponúka aj správnu unifikáciu, no interne ju nepoužíva:

```
? unify_with_occurs_check(X, f(X)).
No
? unify_with_occurs_check(X, f(a)).
X=f(a)
```

Interpretácia programov: SLD rezolúcia

Odvádzanie (dokazovanie) faktov: modus ponens

Socrates is a man.

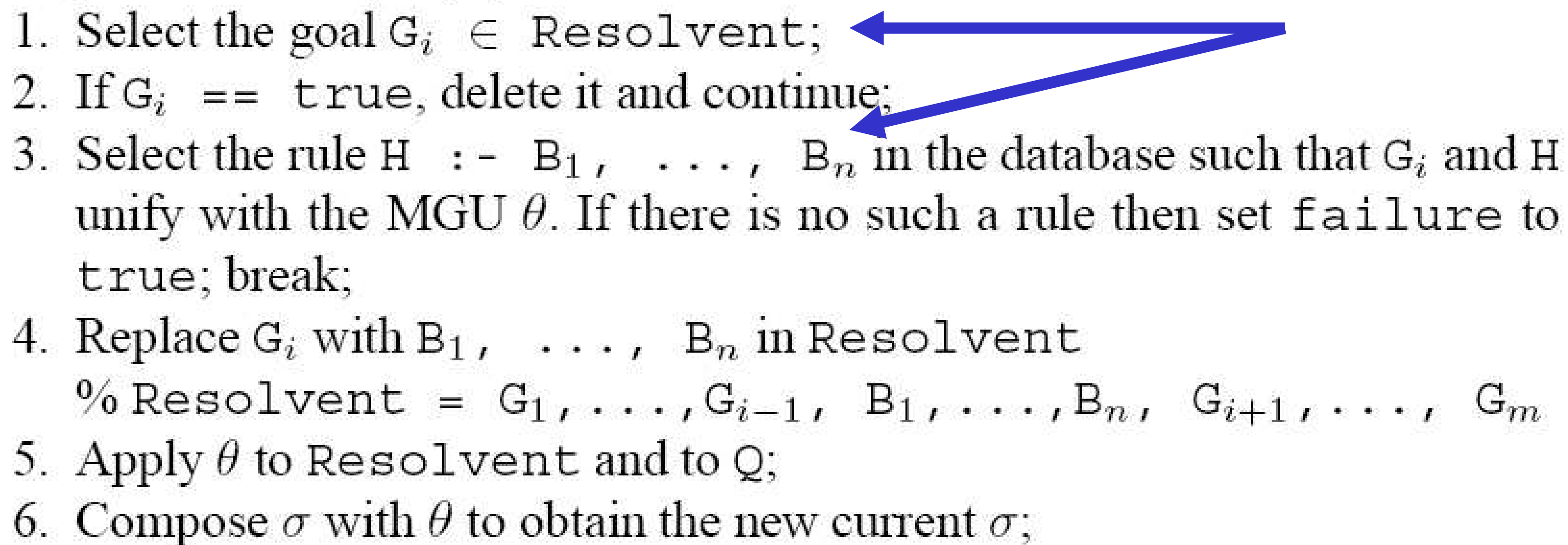
Man is mortal.

Socrates is mortal.

	Formal notation	Prolog notation
Facts	α	`man('Socrates').`
Rules	$\underline{\alpha \Rightarrow \beta}$	`mortal(X) :- man(X).`
Conclusion	β	`mortal('Socrates').`

Prolog používa modus ponens opačne. Snaží sa dokázať β, a keď sa to podarí, snaží sa dokázať α. Toto je vždy konštruktívny dôkaz

Interpretácia programov: SLD rezolúcia



Initialization

Initialize `Resolvent` to `Q`, the initial goal of the resolution algorithm.

Initialize σ to {}

Initialize `failure` to `false`

Loop with `Resolvent = ` G_1, G_2, ..., G_i, ..., G_m

while (`Resolvent` $\neq \emptyset$) {

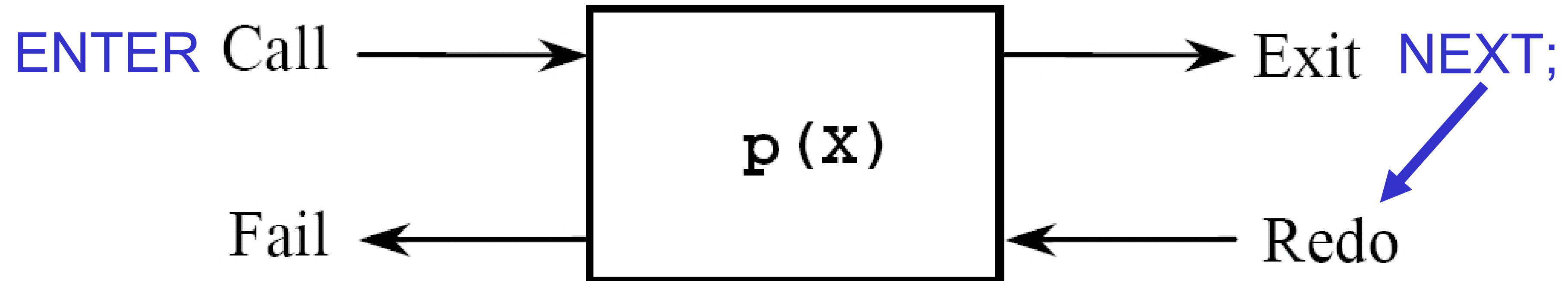
1. Select the goal G_i $\in$ `Resolvent`;
2. If G_i `== true`, delete it and continue;
3. Select the rule `H :-` B_1, ..., B_n in the database such that G_i and `H` unify with the MGU θ. If there is no such a rule then set `failure` to `true`; break;
4. Replace G_i with B_1, ..., B_n in `Resolvent`
   % `Resolvent =` $G_1, \ldots, G_{i-1}$, $B_1, \ldots, B_n$, $G_{i+1}, \ldots$, G_m
5. Apply θ to `Resolvent` and to `Q`;
6. Compose σ with θ to obtain the new current σ;

}

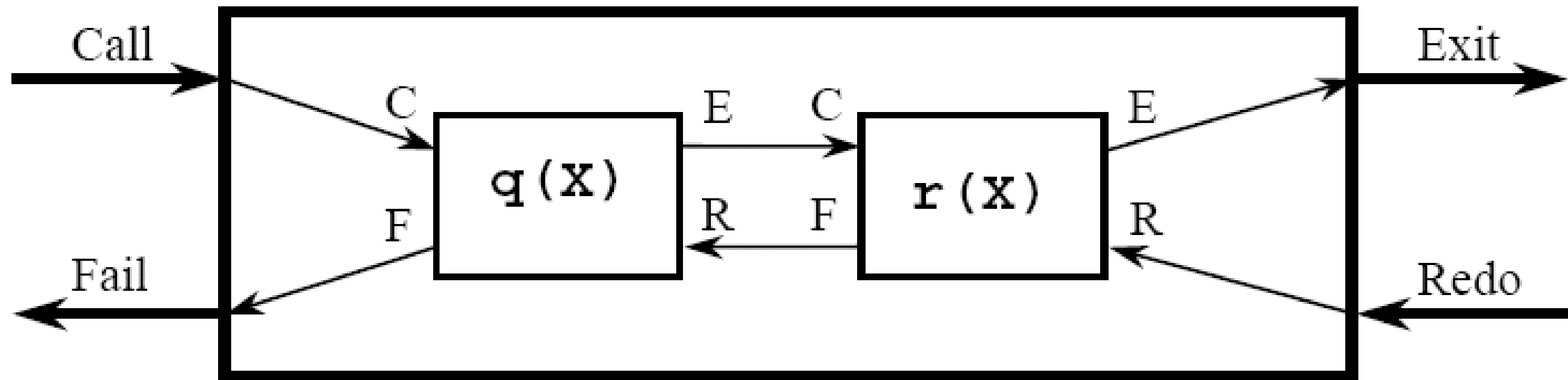
Tento algoritmus nehovorí v akom poradí vyberať podciele, resp. pravidlá. Prolog vždy ide deterministicky zľava doprava

Interpretácia programov: SLD rezolúcia (4-port model)

? p(X).



p(X) :- q(X), r(X).



Interpretácia programov: SLD rezolúcia (4-port model)

To, čo Prolog predvádza pri zapnutom „trace“, je 4-port model dotazu, pričom sa sleduje pohyb z portu do portu

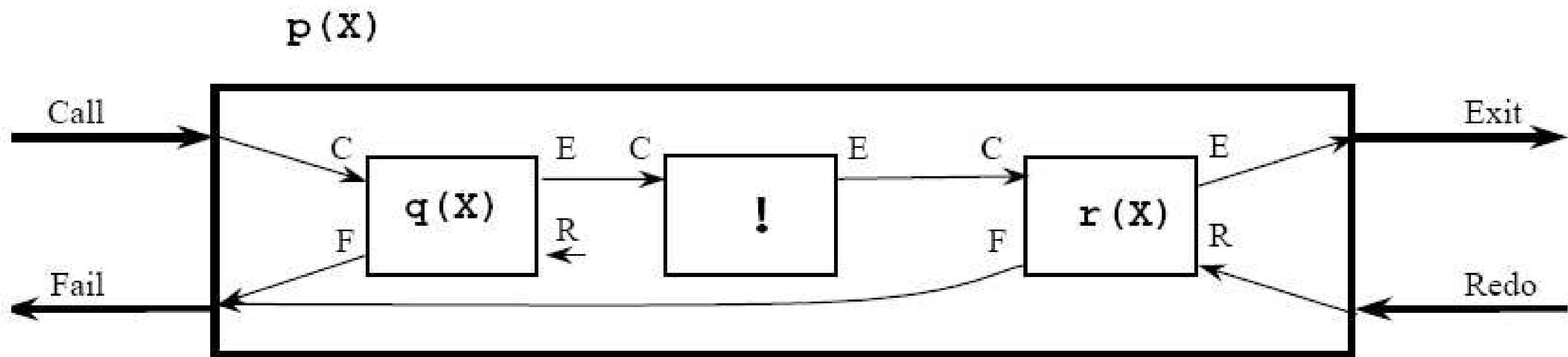
```
mortal(X) :- man(X).
man(socrates).

2 ?- trace.
Yes
[trace] 2 ?- mortal(socrates).
   Call: (7) mortal(socrates) ? creep
   Call: (8) man(socrates) ? creep
   Exit: (8) man(socrates) ? creep
   Exit: (7) mortal(socrates) ? creep
Yes
```

Interpretácia programov: cut!

Cut (!) je špeciálny predikát, ktorý je vždy TRUE, a spôsobí odstránenie backtrackingu (redo) zo všetkých podcieľov naľavo od seba, lokálne v jednom pravidle. Používa sa na optimalizáciu (orezanie vetiev, ktoré určite nevedú k výsledku). Pozor, cut je často zdrojom programátorských chýb!

p(X) :- q(X), !, r(X).



Interpretácia programov: cut!

Let us suppose that a predicate P consists of three clauses:

P :- A_1, ..., A_i, !, A_{i+1}, ..., A_n.
P :- B_1, ..., B_m.
P :- C_1, ..., C_p.

Executing the cut in the first clause has the following consequences:

1. All other clauses of the predicate below the clause containing the cut are pruned. That is, here the two remaining clauses of P will not be tried.
2. All the goals to the left of the cut are also pruned. That is, A_1, ..., A_i will no longer be tried.
3. However, it will be possible to backtrack on goals to the right of the cut.

P :- ~~A_1, ..., A_i,~~ !, A_{i+1}, ..., A_n.
~~P :- B_1, ..., B_m.~~
~~P :- C_1, ..., C_p.~~

Interpretácia programov: minimum bez cut

```
min(X, Y, X) :- X < Y.
min(X, Y, Y) :- X >= Y.

[trace] 3 ?- min(1, 2, X).
   Call: (7) min(1, 2, _G443) ? creep
   Call: (8) 1<2 ? creep
   Exit: (8) 1<2 ? creep
   Exit: (7) min(1, 2, 1) ? creep
X = 1 ;
   Redo: (7) min(1, 2, _G443) ? creep
   Call: (8) 1>=2 ? creep
   Fail: (8) 1>=2 ? creep
No
```

Interpretácia programov: minimum s „green“ cut

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

[trace] 2 ?- min(1, 2, X).
   Call: (7) min(1, 2, _G443) ? creep
   Call: (8) 1<2 ? creep
   Exit: (8) 1<2 ? creep
   Exit: (7) min(1, 2, 1) ? creep
X = 1 ;
No
```

Interpretácia programov: minimum s „red“ cut

```
min(X, Y, X) :- X < Y, !.
min(X, Y, Y).

[trace] 1 ?- min(1,2,X).
   Call: (7) min(1, 2, _G431) ? creep
   Call: (8) 1<2 ? creep
   Exit: (8) 1<2 ? creep
   Exit: (7) min(1, 2, 1) ? creep
X = 1 ;
No
```

Priebeh výpočtu je síce taký istý, ale závisí nielen od výberu podcieľov zľava doprava, ale aj od výberu pravidiel zľava doprava (resp. zhora nadol). Green cuts sa považujú za zdravé, red cuts sa treba radšej vyhýbať (nie vždy sa dá)

Negácia s „red“ cut

Aj negácia sa v Prologu chápe v „pozitívnom“ zmysle (well-founded model):

Ak P platí, tak potom neplatí \+(P).

Ak P neplatí, tak potom platí \+(P).

Toto sa dá vyjadriť pomocou cut, ktorý musí byť „red“ (fail/0 je predikát, ktorý vždy spôsobí fail):

```
\+(P) :- P, !, fail.
\+(P).
```

if-then-else s „red“ cut

If-then-else je predikát ->/3. ‘->’(condition, action1, action2)

zodpovedá konštrukcii

if condition then action1 else action2

Toto sa dá vyjadriť pomocou cut:

‘->’(condition, action1, action2) :- condition, !, action1.

‘->’(condition, action1, action2) :- !, action2.

Zoznamy (lists)

[] je prázdny zoznam

[a] je zoznam, ktorý obsahuje atóm a

[a, X] je zoznam, ktorý obsahuje atóm a, a premennú X

[[a], [X]] je zoznam, ktorý obsahuje 2 zoznamy (je odlišný od predošlého zoznamu)

Hoci sa to možno nezdá, zoznam je tiež predikát, ./2, .(HEAD, TAIL). Syntax [a, X] je len skratka pre .(a, X)

Notácia [HEAD | TAIL] je len krajším (ekvivalentným) zápisom .(HEAD, TAIL).

Zoznamy (lists)

```
?- [a, b] = [H | T].
H = a, T = [b]

?- [a] = [H | T].
H = a, T = []

?- [a, [b]] = [H | T].
H = a, T = [[b]]

?- [a, b, c, d] = [X, Y | T].
X = a, Y = b, T = [c, d]

?- [[a, b, c], d, e] = [H | T].
H = [a, b, c], T = [d, e]
```

The empty list cannot be split:

```
?- [] = [H | T].
No
```

Zoznamy (lists)

Príklady predikátov nad zoznamami:

```
member(X, [X | _]). % nerekurzivne pravidlo
member(X, [_ | T]) :- member(X, T). % rekurzivne pravidlo

append([], L, L). % nerekurzivne pravidlo
append([H | T1], L, [H | T2]) :- append(T1, L, T2). % rekurzia

reverse([], []).
reverse([H | T], R) :- reverse(T, RT), append(RT, [H], R).
```

Toto funguje, ale je náročné na pamäť (hĺbka stacku je priamo úmerná dĺžke zoznamu)

Zoznamy (lists)

Optimalizovaný reverse, ktorý používa akumulátor (častý trik):

```
reverse(L1, L2) :- reverse(L1, [], L2).
reverse([], L, L).
reverse([H | T], Acc, L) :- reverse(T, [H | Acc], L).
```

Dobrý programátorský štýl

V Prologovských helpoch nájdete informáciu o tom, ktoré argumenty predikátu sú (obyčajne) vstupné (+), ktoré výstupné (-) a ktoré vstupno-výstupné, napr. between(+Low, +High, ?Value), append(?List1, ?List2, ?List3)

Pravidlá sa píšu tak, aby každý podcieľ bol v samostatnom riadku:

```
reverse([H | T], R) :-
        reverse(T, RT),
        append(RT, [H], R).
```

findall, bagof, setof

`findall(+Variable, +Goal, ?Solution)` unifies `Solution` with the list of all the possible values of `Variable` when querying `Goal`.

```
?- findall(X, character(X, iliad), B).
B = [ulysses, hector, achilles]

?- findall(X, character(X, Y), B).
B = [ulysses, hector, achilles, ulysses, penelope,
telemachus]
```

findall, bagof, setof

The predicate bagof(+Variable, +Goal, ?Solution) is similar to findall/3, except that it backtracks on the free variables of Goal:

```
?- bagof(X, character(X, iliad), Bag).
Bag = [ulysses, hector, achilles]

?- bagof(X, character(X, Y), Bag).
Bag =[ ulysses, hector, achilles], Y = iliad ;
Bag = [ulysses, penelope, telemachus], Y = odyssey ;
No.
```

Variables in Goal are not considered free if they are existentially quantified. The existential quantifier uses the infix operator "^". Let X be a variable in Goal. X^Goal means that there exists X such that Goal is true. bagof/3 does not backtrack on it. For example:

```
?- bagof(X, Y^character(X, Y), Bag).
Bag = [ulysses, hector, achilles, ulysses,
penelope, telemachus]

?- bagof(X, Y^(character(X, Y), female(X)), Bag).
Bag = [penelope]
```

findall, bagof, setof

The predicate `setof(+Variable, +Goal, ?Solution)` does the same thing as `bagof/3`, except that the `Solution` list is sorted and duplicates are removed from it:

```
?- setof(X, Y^character(X, Y), Bag).
Bag = [achilles, hector, penelope, telemachus,
ulysses]
```

Ďalšie optimalizačné techniky: dynamické predikáty

Táto implementácia používa dvojitú rekurziu (neefektívne).

```
fibonacci(1, 1).
fibonacci(2, 1).
fibonacci(M, N) :-
  M > 2,
  M1 is M - 1, fibonacci(M1, N1),
  M2 is M - 2, fibonacci(M2, N2),
  N is N1 + N2.
```

Táto tiež, ale efektívnejšie (dynamický update relácie fibonacci):

```
:- dynamic(fibonacci / 2).
fibonacci(1, 1).
fibonacci(2, 1).
fibonacci(M, N) :-
  M > 2,
  M1 is M - 1, fibonacci(M1, N1),
  M2 is M - 2, fibonacci(M2, N2),
  N is N1 + N2,
  asserta(fibonacci(M, N)).
```

Ďalšie optimalizačné techniky: tail recursion

Pravidlo je chvostovo rekurzívne, ak sa rekurzia objavuje len v poslednom podcieli pravidla. Chvostová rekurzia sa dá jednoducho transformovať na iteráciu

Nesprávne:

```
f(X) :- g(X, Y), f(Y).
f(X) :- fact(X).
```

Správne:

```
f(X) :- fact(X).
f(X) :- g(X, Y), f(Y).
```

Tiež správne:

```
f(X) :- g(X, Y), !, f(Y).
f(X) :- fact(X).
```