

<http://www.dcs.fmph.uniba.sk/~plachetk/TEACHING/DB2>

Tomáš Plachetka, Ján Šturc

Faculty of mathematics, physics and informatics,
Comenius University, Bratislava

Summer 2023–2024

- 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”
- Related project: <http://www.zillions-of-games.com>

- **Herbrand (1930), Turing (1940), Robinson (1965): automatic proving of theorems**
 - **Colmerauer (1970): first implementation of Prolog**
 - **Deransart, ISO standard (1995)**
 - **... and many others (Prívvara, Ružička, ...)**
-
- **Logic (declarative) programming, based on the first-order logic**

Prolog language components: facts (EDB predicates)

predicate / arity.

character(achilles, illiad).

character/2

character(menelaus, illiad).

character(helen, illiad).

character(menelaus, odyssey).

character(helen, odyssey).

male(achilles).

male/1

male(menelaus).

female(helen).

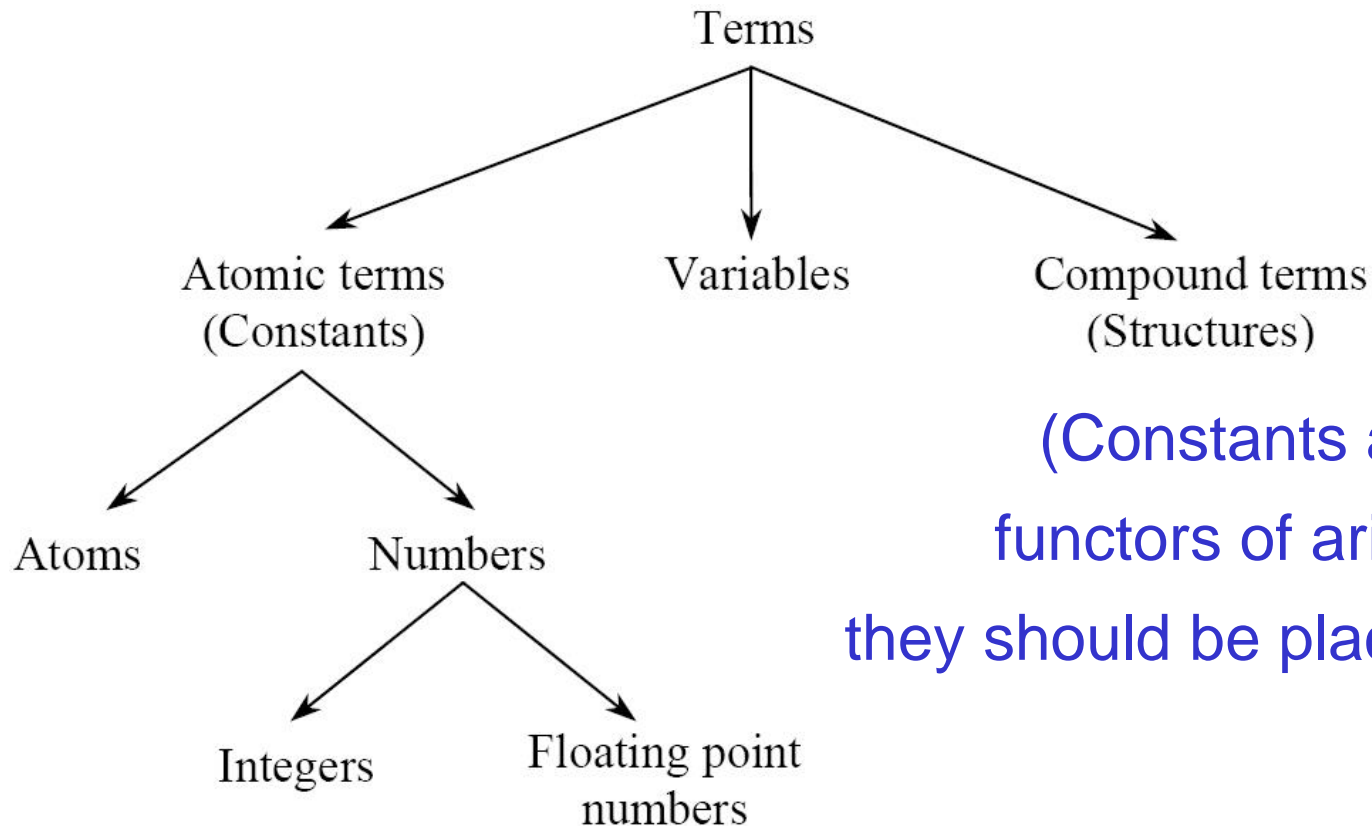
female/1

universe(_).

Prolog language components: terms

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

Function symbols (functors) combine simple terms into more complex structures. Functors with the same name but different arity are different (overloading)



(Constants are functors of arity 0, they should be placed here.)

Prolog language components: rules

HEAD :- GOAL₁, GOAL₂, ..., GOAL_N.

Right side of the rule (body) is a conjunction of subgoals

It is possible to express disjunction in one rule, using ‘;’, but it is recommended to write more rules with the same head instead of that:

HEAD :- GOAL₁; GOAL₂.

Rules with an empty head are called *directives*, which are executed only once, e.g.

:- dynamic(dodava / 2).

Queries are syntactically identical with rules’ bodies („anonymous rules“)

The key algorithm is unification of two terms (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 $x = f(x_1, \dots, x_n)$ and $y = f(y_1, \dots, y_n)$.

 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 $\neq \emptyset$)

Unification, =

Prolog implementations omit occurs-check in unification, although it can be implemented efficiently (it was long believed that it is responsible for Prolog being slow)

Contemporary Prologs offer a correct unification, but they do not use it internally:

? unify_with_occurs_check(X, f(X)).

No

? unify_with_occurs_check(X, f(a)).

X=f(a)

Computation of queries: SLD resolution

Derivation (proof) of truths: modus ponens

Socrates is a man.

Man is mortal.

Socrates is mortal.

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

Prolog uses modus ponens the other way around. It attempts to prove β , and begins with an assumption that it holds. Then it constructs a proof for β by attempting to prove α . The proof is constructed from the consequence to axioms. When the proof of α fails, then Prolog concludes that β does not hold, i.e. $\neg\beta$ holds (**negation as failure**)

Computation of queries: SLD resolution

Initialization


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

Initialize σ to $\{\}$

Initialize failure to false

Loop with Resolvent = $G_1, G_2, \dots, G_i, \dots, G_m$

while (Resolvent $\neq \emptyset$) {

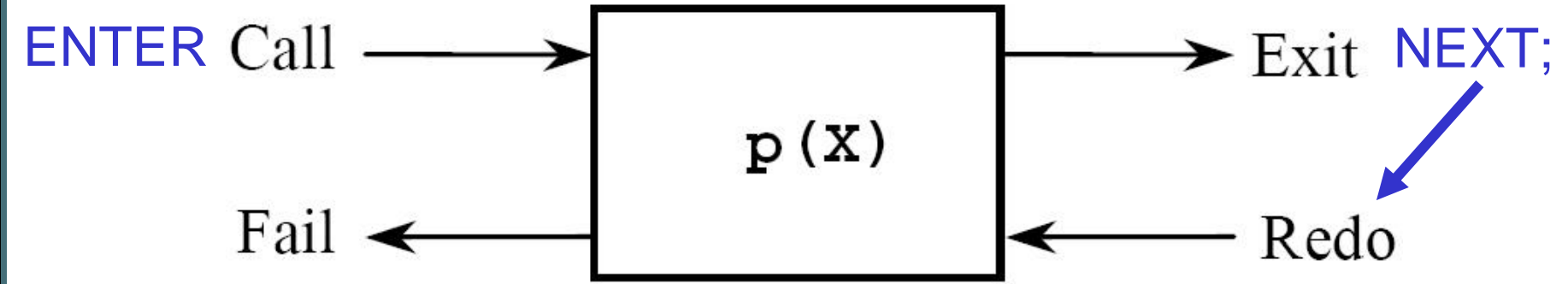
1. Select the goal $G_i \in \text{Resolvent}$; 
2. If $G_i == \text{true}$, delete it and continue;
3. Select the rule $H :- B_1, \dots, B_n$ in the database such that G_i and H unify with the MGU θ . If there is no such a rule then set failure to true; break;
4. Replace G_i with B_1, \dots, B_n in Resolvent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

}

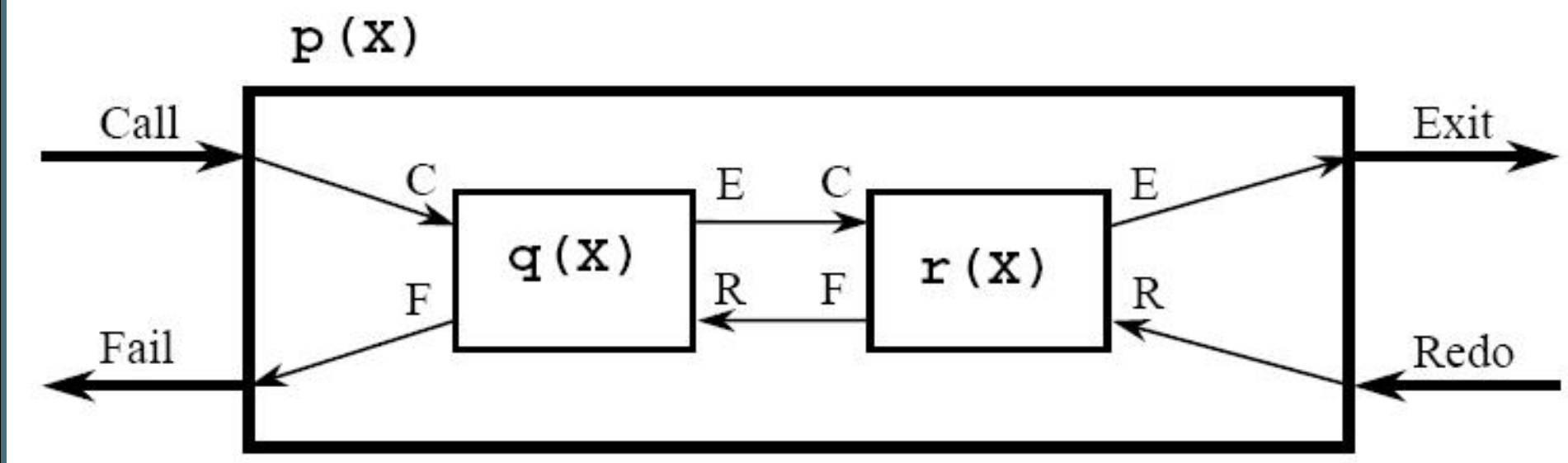
This algorithm does not specify the order for choice of goals and rules, the choice is non-deterministic. **Prolog always makes the choice deterministically, from left to right (a stable model)**

Computation of queries: SLD resolution, 4-port model

? p(X).



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



Computation of queries: SLD resolution, 4-port model

Prolog's "tracing" (debugging) mode reports the transitions in the 4-port model

```
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
```

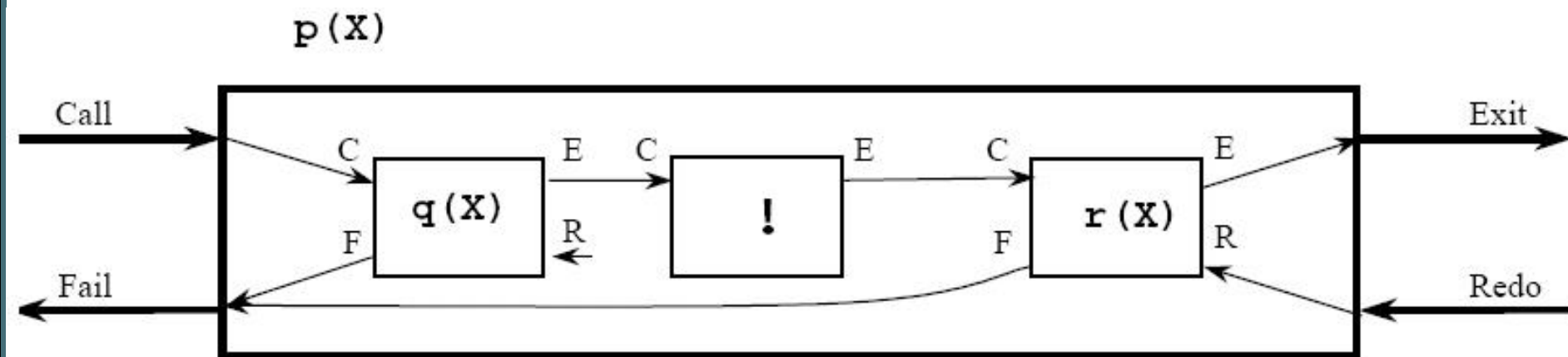
```
Exit: (7) mortal(socrates) ? creep
```

```
Yes
```

Computation of queries: SLD resolution, cut, !

Cut (!) is a special predicate, which always evaluates as TRUE, but it also removes the backtracking (redo) from all subgoals to the left of itself, within one rule. It is used for optimisation (pruning of branches which do not lead to the result). Beware, cut is often a source of programming errors!

$p(X) \text{ :- } q(X), !, r(X).$



Computation of queries: SLD resolution, cut, !

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

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

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, \dots, A_i will no longer be tried.
3. However, it will be possible to backtrack on goals to the right of the cut.

$$P \text{ :- } \overline{A_1, \dots, A_i}, !, A_{i+1}, \dots, A_n.$$
$$\overline{P \text{ :- } B_1, \dots, B_m.}$$
$$\overline{P \text{ :- } C_1, \dots, C_p.}$$

Computation of queries: minimum without cut

$\text{min}(X, Y, X) :- X < Y.$

$\text{min}(X, Y, Y) :- X \geq Y.$

[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 \geq 2 ? creep

Fail: (8) 1 \geq 2 ? creep

No

Computation of queries: minimum with “green cut”

$\text{min}(X, Y, X) :- X < Y, !.$

$\text{min}(X, Y, Y) :- X \geq Y, !.$

[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

Computation of queries: minimum with “red cut”

$\text{min}(X, Y, X) :- X < Y, !.$

$\text{min}(X, Y, Y).$

[trace] 1 ?- $\text{min}(1, 2, X).$

Call: (7) $\text{min}(1, 2, _G431) ?$ creep

Call: (8) $1 < 2 ?$ creep

Exit: (8) $1 < 2 ?$ creep

Exit: (7) $\text{min}(1, 2, 1) ?$ creep

$X = 1 ;$

No

The computation is the same, but it depends not only on the choice of goals from left to right, but also on the choice of rules from left to right. Green cuts are considered “harmless”. Red cuts should be avoided

Computation of queries: negation with “red cut”

Negation in Prolog is „positive“ (negation as failure):

If P holds, then not(P) does not hold.

If P does not hold, then not(P) holds

This can be expressed using a cut, which must be „red“ (fail/0 is a predicate, which is false, i.e. always fails):

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

or not(P) :- call(P), !, fail.

not(P).

(call(P) forces Prolog to evaluate P)

Equivalently, with if-then-else: “if P succeeds then fail, otherwise succeed”

not(P) :- call(P) -> fail ; true.

Computation of queries: if-then-else with “red cut”

If-then-else is a predicate $\rightarrow/3$. ‘ \rightarrow ’(condition, action1, action2)
which corresponds to
if condition then action1 else action2

This can be expressed using cut:

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

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

[] is an empty list

[a] is a list with an atom a

[a, X] is a list with an atom a and a variable X

[[a], [X]] is a list with 2 lists (it differs from the previous list)

Although it is perhaps not evident, a list is a functor, `./2`,
`.(HEAD, TAIL)`. Syntax `[a, X]` is an abbreviation for `.(a, .(X, nil))`.

Notation `[HEAD | TAIL]` is equivalent with `.(HEAD, TAIL)`

```
?- [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
```

Examples of predicates over lists:

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

```
member(X, [_ | T]) :- member(X, T). % recursion
```

```
append([], L, L).
```

```
append([H | T1], L, [H | T2]) :- append(T1, L, T2). % recursion
```

```
reverse([], []).
```

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

This works, but is memory consuming (the depth of the stack is proportional to the length of the list)

Optimised reverse which uses an accumulator (a frequent trick):

```
reverse(L1, L2) :- reverse(L1, [], L2).
```

```
reverse([], L, L).
```

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

Prolog manuals usually provide an information on which arguments of a predicate are (usually) input (+), i.e. bound; and which are output (-), i.e. free; and which are input/output (?), i.e. bound or free, e.g.

`between(+Low, +High, ?Value), append(?List1, ?List2, ?List3)`

The goals are usually written one in a line:

```
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]
```

This (inefficient) program uses a double recursion:

```
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.
```

This too, but more efficiently (dynamic update of the predicate):

`:- 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)). ←
```

Other optimisation techniques: tail recursion

A rule is tail-recursive, when the recursion appears only in the last subgoal of the rule. A tail recursion can be easily transformed to an iteration

Inefficient:

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

```
f (X) :- fact (X) . /* fact is an EDB predicate, not factorial */
```

More efficient:

```
f (X) :- fact (X) .
```

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

Also more efficient, but less comprehensible:

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

```
f (X) :- fact (X) .
```