FUNKCIONÁLNÍ A LOGICKÉ PROGRAMOVÁNÍ
9. ÚVOD DO LOGICKÉHO PROGRAMOVÁNÍ A PROGRAMOVACÍHO JAZYKA PROLOG.

2011 Jan Janoušek
MI-FLP

Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

Characteristics of logic programming

- Non-imperative programming.
- Non-procedural programming (**procedures (functions)** as in imperative or functional languages **do not exist**).
- Declarative programming.
- **Lambda calculus** is a theoretical basis for **functional programming**.
- **First-order logic calculus** is a theoretical basis for **logic programming**.
- Functional languages have more properties than lambda calculus; logic languages have less properties than first-order logic calculus.

Motivating examples

Now only definitions of problems, later their elegant and direct solutions written in Prolog language.

Einsteinova hádanka (Einstein riddle)

1. Je pět domů, každý jiné barvy. (1,2,3,4,5)
2. V každém domě bydlí jedna osoba jiné národnosti.
3. Každý obyvatel domu upřednostňuje určitý nápoj, určitou značku cigaret a chová určité zvíře.
4. Žádná z těchto pěti osob nepije stejný nápoj, nekouří stejné cigarety ani nechová stejné zvíře jako někdo jiný.

Otázka: Komu patří ryba, víme-li, že

1. Brit žije v červeném domě.
2. Švéd chová psa.
3. Dán pije čaj.
4. Němec kouří cigarety Rothmans.
5. Nor bydlí v prvním domku.
6. Obyvatel zeleného domku pije kávu.
7. Kuřák cigaret Winfield pije rád pivo. Obyvatel žlutého domku kouří cigarety Dunhill.
8. Osoba, která kouří Pall Mall, chová papouška.
9. Obyvatel prostředního domku pije mléko.
10. Kuřák cigaret Marlboro bydlí vedle toho, kdo chová kočku.
11. Muž, který chová koně, bydlí vedle toho, kdo kouří Dunhill.
12. Nor bydlí vedle modrého domu.
13. Kuřák cigaret Marlboro má souseda, který pije vodu.
14. Zelený dům stojí nalevo od bílého domu.

Pozn. Einstein předpokládal, že bez tužky a papíru hádanku vyřeší 2% lidí.

PROLOG

Introduction

- Based on first-order predicate logic.
- The name comes from “**PRO**gramming in **LOG**ic".
- Developed at the University of in 1972.
- First implementation was in FORTRAN and written by Alain Colmeraurer.
- Originally intended as a tool for working with natural languages and mechanical theorem proving.

Introduction

- Used by Japan in 1981 as the core programming language for their "fifth generation computers" project.
- Currently used in artificial intelligence, databases, expert systems.
- Prolog is a commercially successful language. Many companies have made a business of supplying Prolog implementations, Prolog consulting, and/or applications in Prolog.

Prolog Programs, Horn Clauses

- Program consists of Horn clauses (axioms). ("If B1 is true and B2 is true and ...and Bn is true then H is true".)

- **Axioms** can be **facts, rules,** and **queries.**

- Axioms:

$H \leftarrow B1, B2,\ldots,Bn$

head body are in Prolog written:

H :- B1, B2,....,Bn.

Prolog Execution

- Run your Prolog program by:
  - Enter a series of axioms
  - Enter a query
  - System tries to prove your query by finding a series of inference steps

Facts

Facts - clause with head and no body.

Example:

```
parent(josef, jan).
parent(josef, sarka).
parent(jana, jan).
parent(jana, sarka).
parent(anna, josef).
male(josef).
male(jan).
female(sarka).
female(jana).
female(anna).
```

Rules

Rules - have both head and body.

Example (X,Y, and Z are variables):

```
mother(X,Y) :- parent(X,Y), female(X).
father(X,Y) :- parent(X,Y), male(X).
grandmother(X,Y) :- mother(X,Z), parent(Z,Y).
grandfather(X,Y) :- father(X,Z), parent(Z,Y).
brother(X,Y) :- parent(Z,X), parent(Z,Y), male(X).

predecessor(X,Y) :- parent(X,Y).
predecessor(X,Y) :- parent(X,Z), predecessor(Z,Y). % recursion
% In Prolog, recursion is used to express cycles!!!
```

Queries

Query – can be thought of as a clause with no body.

Example:

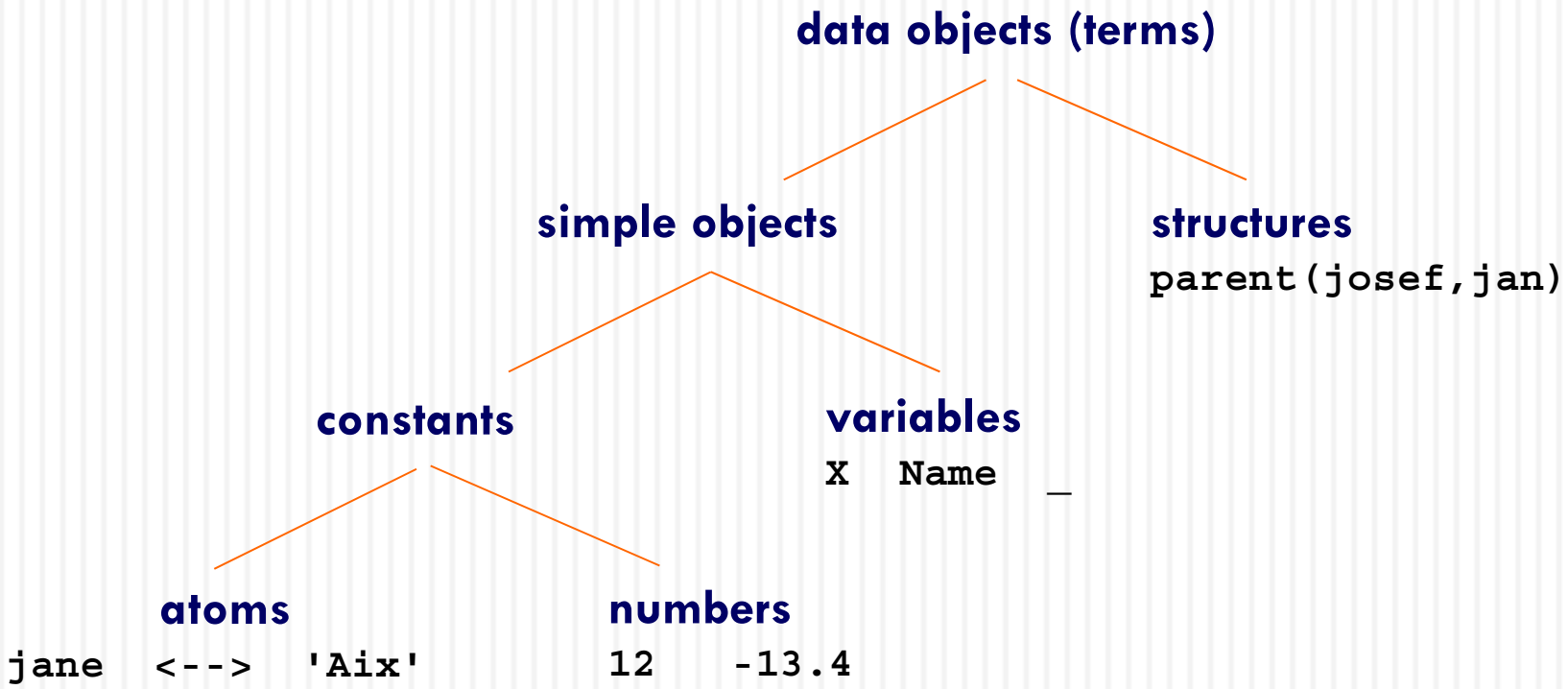
?- parent(josef,sarka).
yes

?- predecessor(X,sarka).
X=jana
X=josef
X=anna
no

Terms

- Heads and bodies are **terms.**
- Terms are of four kinds:
  - **Atoms** - begin with lowercase letters: x, y, z, fred
  - **Numbers**: integers, reals
  - **Variables** - begin with capital letters: X, Y, Z, Alist
  - **Structures**: consist of an atom called a **functor**, and a list of arguments. ex. edge(a,b). line(1,2,4).

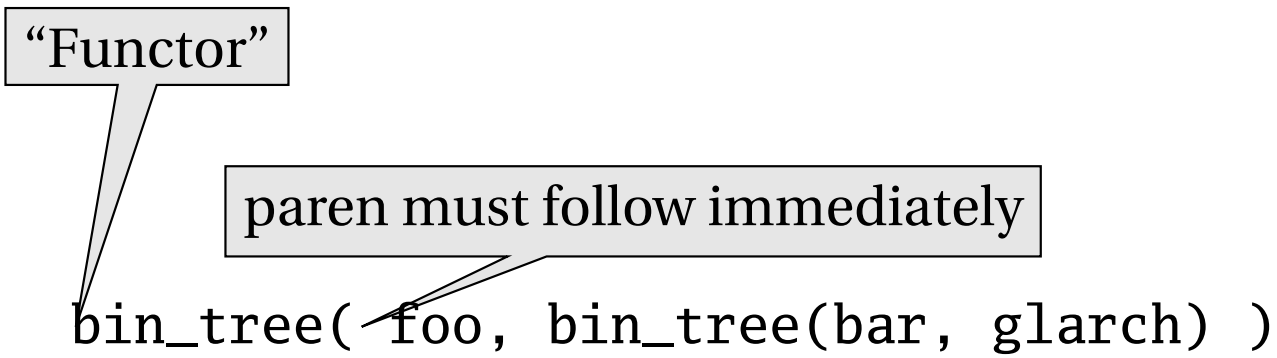
Classification of terms once more



Similar to Lisp - data and programs have the same syntax!

Structures and Functors

A structure consists of a functor followed by an open parenthesis, a list of comma-separated terms, and a close parenthesis:



What's a structure? Whatever you like.

A predicate `nerd(stephen)`
A relationship `teaches(edwards, cs4115)`
A data structure `bin(+, bin(-, 1, 3), 4)`

UNIFICATION – the way how interpreters of Prolog work.

General possibilities of searching the database of Horn clauses. This database (ie. Prolog program) can represented by a tree.

- **DEPTH FIRST** - finds a complete sequence of propositions for the first subgoal before working on the others. **This is what Prolog uses in general!**
- **BREADTH FIRST** - works on all subgoals in parallel. (Can also be done if specific operators are in Prolog program.)
- The implementers of Prolog choose depth first because it can be done with a stack (expected to use fewer memory resources than breadth first).
- Note. The search order can be controlled by **Cut operator** (operátor řezu) – see the next lecture.

Unification

Part of the search procedure that matches patterns.

The search attempts to match a goal with a rule in the database by unifying them.

Recursive rules:

- A constant only unifies with itself
- Two structures unify if they have the same functor, the same number of arguments, and the corresponding arguments unify
- A variable unifies with anything but forces an equivalence

Unification Examples

The = operator checks whether two structures unify:

```
| ?- a = a.
yes                                % Constant unifies with itself
| ?- a = b.
no                                 % Mismatched constants
| ?- 5.3 = a.
no                                 % Mismatched constants
| ?- 5.3 = X.
X = 5.3 ? ;                        % Variables unify
yes
| ?- foo(a,X) = foo(X,b).
no                                 % X=a required, but inconsistent
| ?- foo(a,X) = foo(X,a).
X = a                              % X=a is consistent
yes
| ?- foo(X,b) = foo(a,Y).
X = a
Y = b                              % X=a, then b=Y
yes
| ?- foo(X,a,X) = foo(b,a,c).
no                                 % X=b required, but inconsistent
```

The Searching Algorithm

in the order they appear

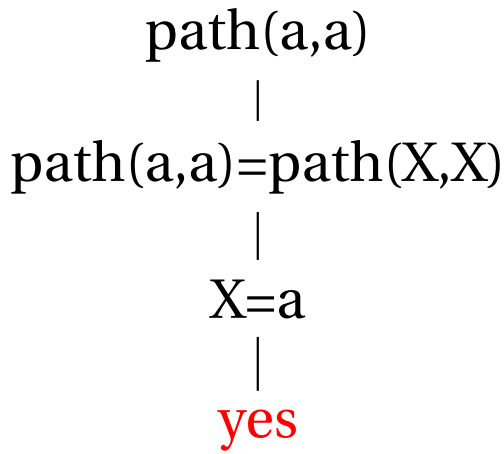
search(goal g, variables e)
    for each clause h :- $t_1, \ldots, t_n$ in the database
        e = unify(g, h, e)
        if successful,
            for each term $t_1, \ldots, t_n$,
                e = search(t_k, e)
        if all successful, return e
    return no

in the order they appear

Note: This pseudo-code ignores one very important part of the searching process!

Order Affects Efficiency

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).

path(X, X).

path(X, Y) :-
    edge(X, Z), path(Z, Y).
```



Consider the query

```
| ?- path(a, a).
```

Good programming practice: Put the easily-satisfied clauses first.

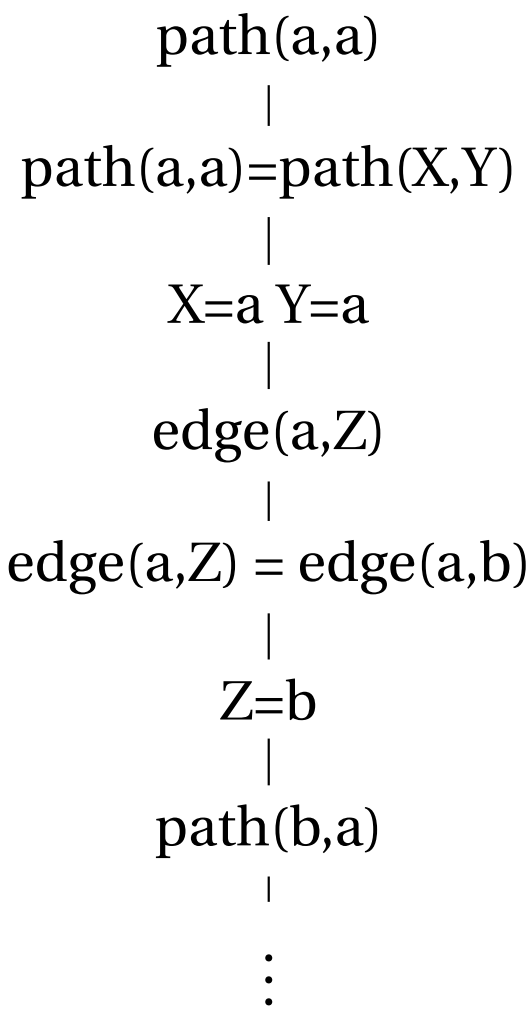
Order Affects Efficiency

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).

path(X, Y) :-
    edge(X, Z), path(Z, Y).

path(X, X).
```

Consider the query

```
| ?- path(a, a).
```

Will eventually produce the right answer, but will spend much more time doing so.

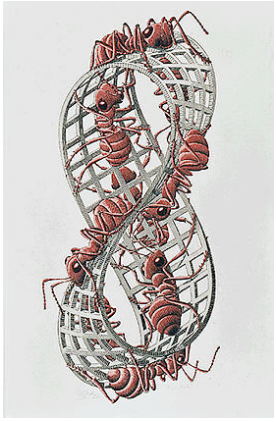


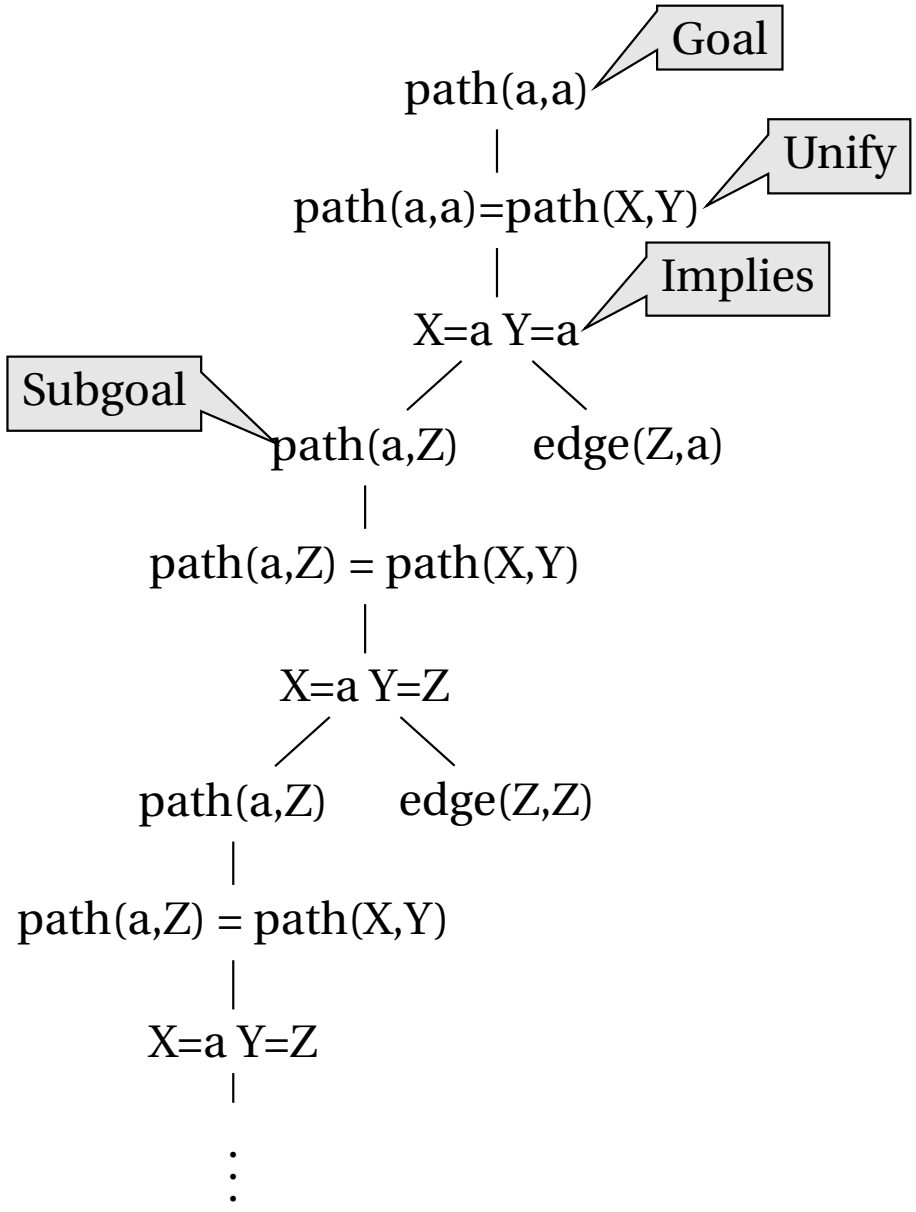
Order Can Cause Infinite Recursion

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).

path(X, Y) :-
    path(X, Z), edge(Z, Y).

path(X, X).
```

Consider the query

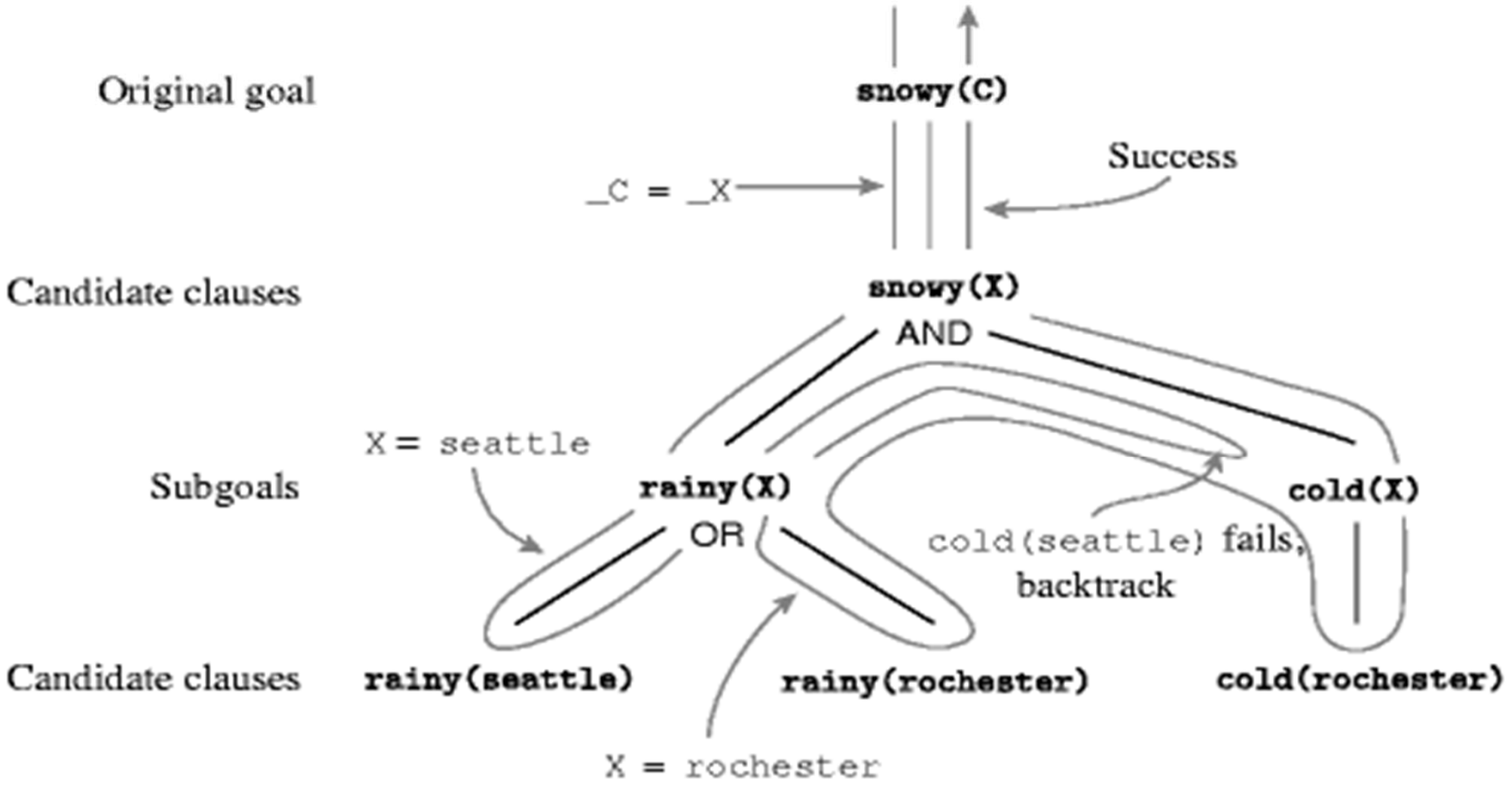
```
| ?- path(a, a).
```





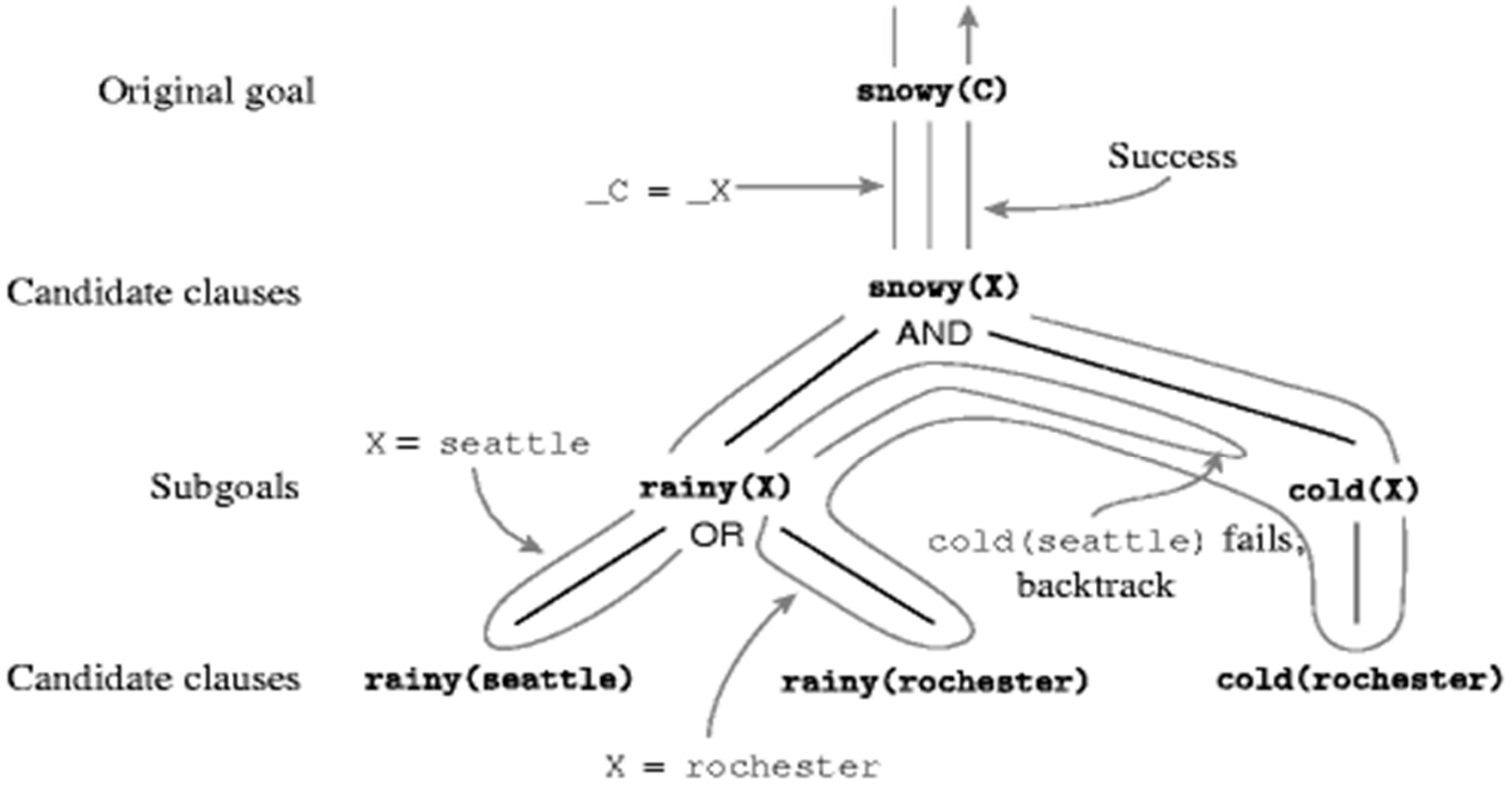
Backtracking search


```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X)
```



Backtracking search


```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X)
```



Types of parameters

- Input, output, and input/output denoted by +, -, and ?, respectively.

- Example:

```
%=============================================
% predek(?Pred,?Pot) Pred je předkem potomka Pot
%=============================================
```

More syntax – arithmetics, and, or

- ARITHMETICS assign statement: is

Example:

Z is Z+1

- Operators , and ;
  - , - conjunction (logical and)
  - ; - disjunction (logical or)

Example:

p(X):- a(X,Y),b(Y);c(Y),d(X,Y).

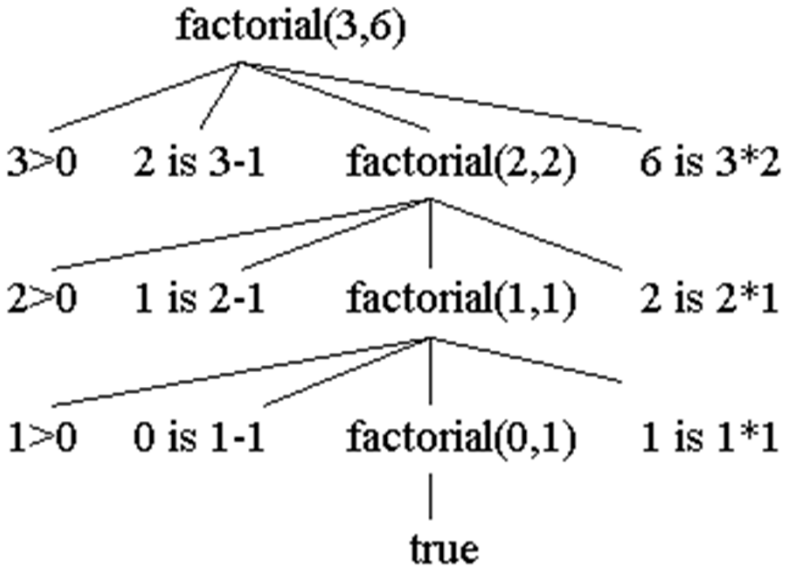
is equivalent to

p(X):- a(X,Y),b(Y).

p(X):- c(Y),d(X,Y).

Example: factorial


```
factorial(0,1).

factorial(N,F) :-
   N>0,
   N1 is N-1,
   factorial(N1,F1),
   F is N * F1.


?-  factorial(3,W).
W=6
```



Lists in Prolog

- [] % the empty list
- [1]
- [1,2,3]
- [[1,2], 3] % can be heterogeneous.

The | separates the head and tail of a list:

is [a | [b,c]]

Other example Programs

Testing membership for lists - `member(Elm,List)`

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

Joining two lists together - `append(L1, L2, Result)`

```
append([], L, L).
append([A | L1], L2, [A | Rest]) :- append(L1, L2, Rest).
```

Adding list elements – `sum_list(L, S)`

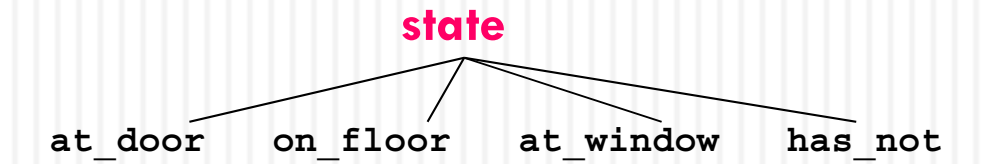
```
sum_list([], 0).
sum_list([X | Rest], S) :- sum_list(Rest, SR), S is SR + X.
```

Yet another example – monkey and banana

We have a room with a door and a window, a banana is hanging in the center on a rope. A monkey is standing at the door, a box is placed at the window. The monkey cannot get banana from the floor, only when staying on top of the box. The monkey wants to get banana.

Initial state

1. monkey is at the door
2. monkey is on the floor
3. box is at the window
4. monkey has no banana




```
state(at_door, on_floor, at_window, has_not)
```

Final state:

```
state(_, _, _, has)
```

- **How to change the state? What can do the monkey?**
- grasp the banana `grasp`
- climb on top of the box `climb`
- push the box `push(A,B)`
- walk to other place `walk(A,B)`

- **State changes via action execution**
  - `change(State1, Action, State2)`
    - previous state
    - following state

How to define the changes?


```
change(state(in_center, on_box, in_center, hasnot),
            grasp,
            state(in_center, on_box, in_center, has)).
change(state(...),
            climb,
            state(...)).
change(state(...),
            push(...),
            state(...)).
change(state(P1, on_floor, C, T),
            walk(P1, P2),
            state(P2, on_floor, C, T)).
```

Can the monkey get the banana?

- `can_get(S)` ... an extra predicate
- the monkey has it already:
  - `can_get(state(_,_,_,has)).`
- not yet having, but can reach a state and get it from there:
  - `can_get(S1):- change(S1, A, S2), can_get(S2).`