Logic and Constraint Programming

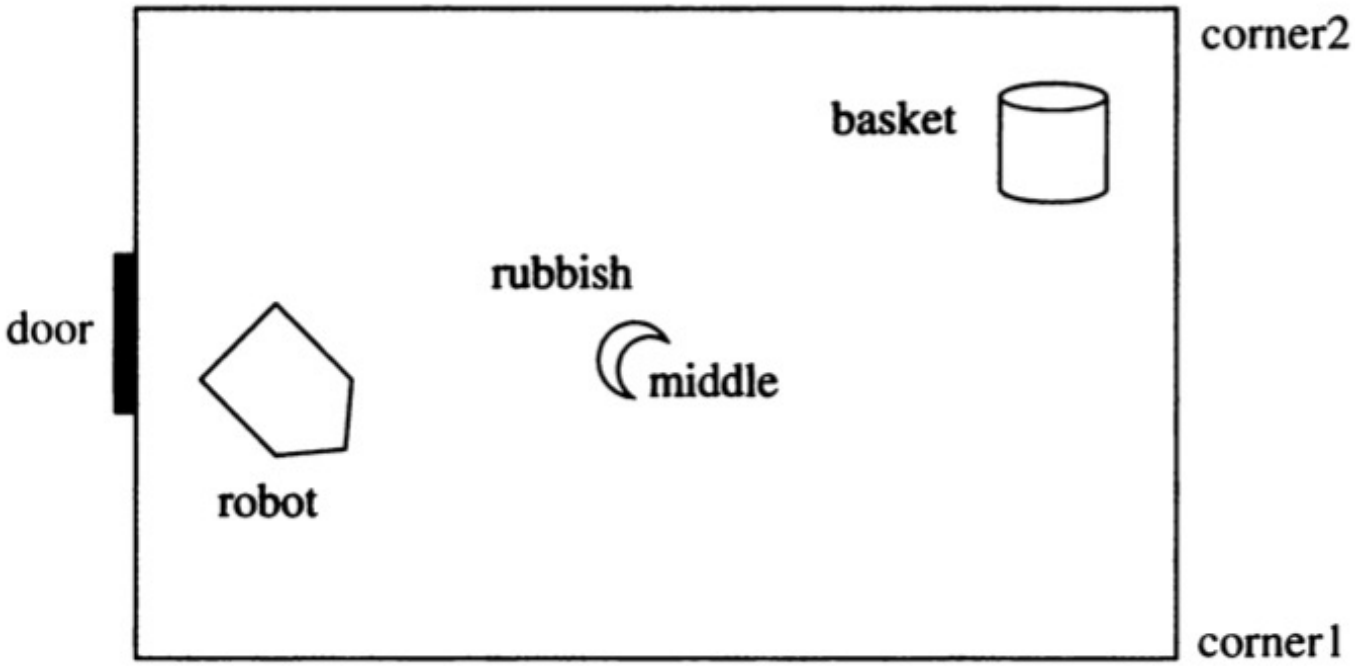
PROLOG

Prof. Fabrizio Fornari

June 7, 2022

Robot Task Planning

Let us consider a mobile robot that has the task of cleaning a room.

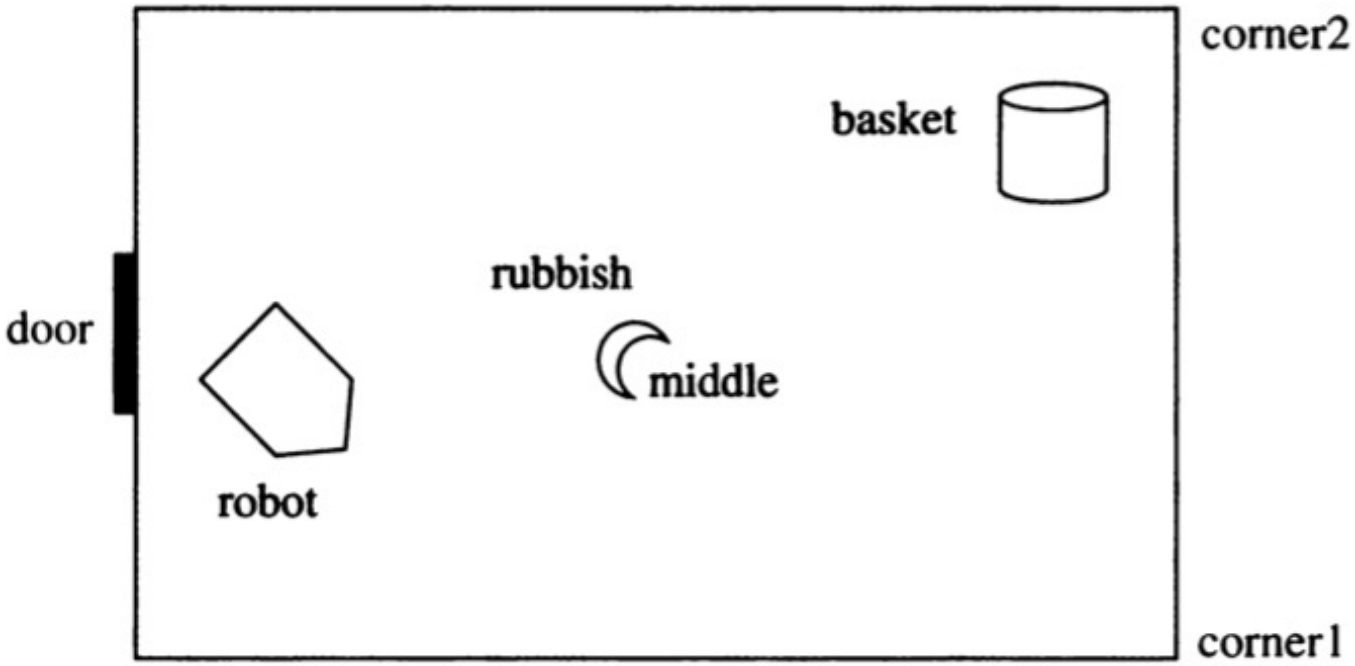


The **robot** is at the **door**, and there is **a piece of rubbbish in the middle of the room** and a **waste basket** in one of the **corners** of the room.

We assume the robot knows how to execute basic commands such as:
go(door,middle)
pick up %pick up the rubbish
drop %drop whatever the robot is holding into the basket
push %push the basket

Robot Task Planning

Let us consider a mobile robot that has the task of cleaning a room.



The robot, to execute commands may use its vision system, which recognizes objects and their locations.

A concrete task for the robot is specified by **goals** the robots is to achieve. A goal may be: **rubbish in the basket**.

The task planning problem is to find a sqeunce of robot actions such that the goal is achieved after executing this sequence.

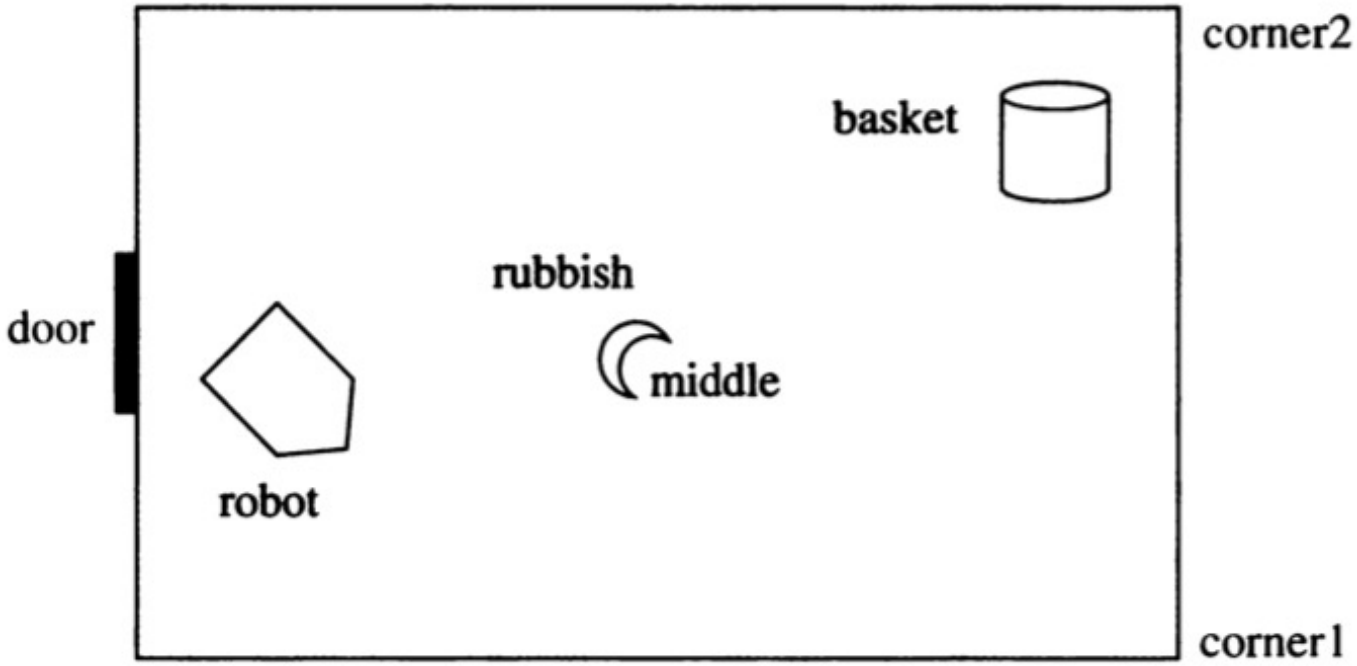
Robot Task Planning

Let us consider a mobile robot that has the task of cleaning a room.

Define the current state of the robot's world

Three things:
the position of the robot,
the position of the rubbish,
the position of the basket.

In our case:
1. Robot at door.
2. Basket in corner 2.
3. Rubbish in the middle of room.

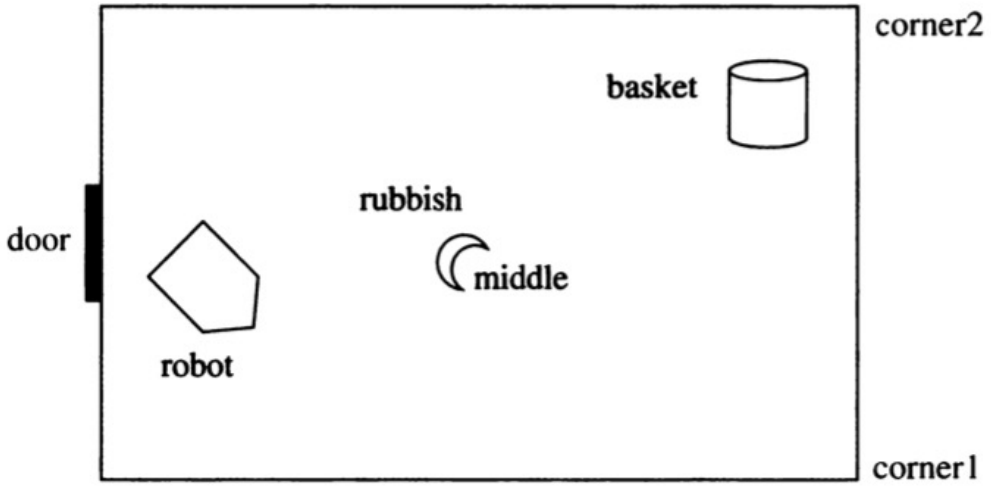


Robot Task Planning

We can combine all the three pieces of location information into one structured object.

state(door, corner2, floor(middle))

robot basket rubbish



The goal '**rubbish in basket**' can be specified by stating that the robot's plan has to bring the world into a state of the form:

state(_, _, in_basket)

Robot Task Planning

Four actions:
pickup, pick up rubbish from floor
drop, drop rubbish into basket
push(Pos1, Pos2), push basket from position **Pos1** to **Pos2**
go(Pos1, Pos2), go from **Pos1** to **Pos2**

Not all actions are possible in every state of the world.
The action ‘drop’ is only possible if the robot is holding rubbish next to the basket.

Such rules can be formalized in Prolog as a three-place relation named **action**:
action(State1, Action, State2)

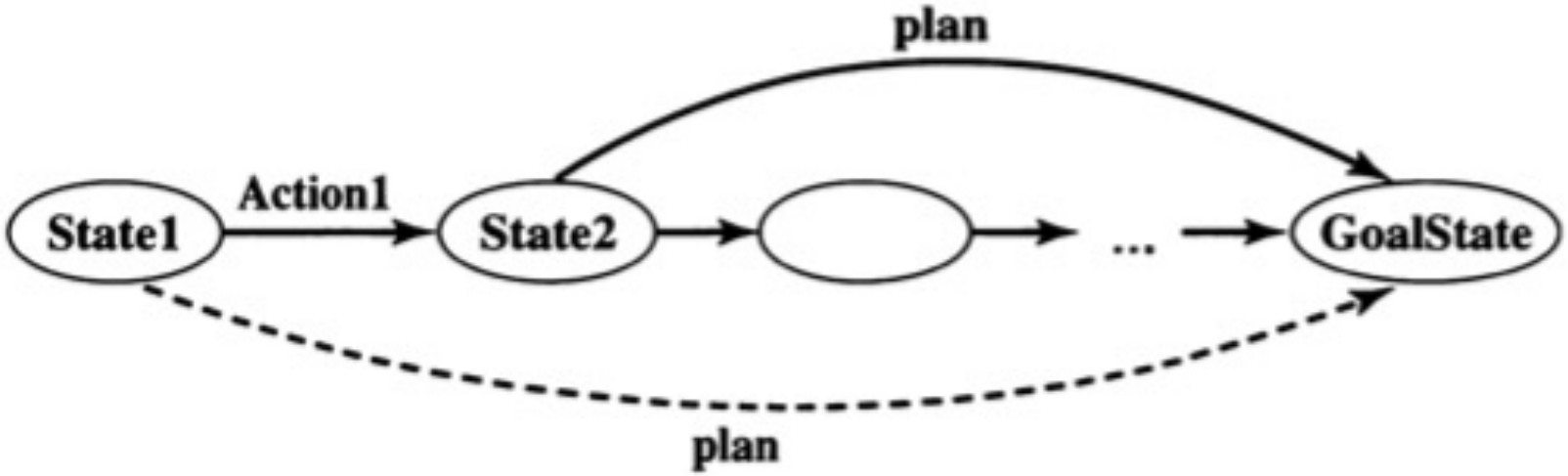
The three arguments of the relation specify an action thus:

State1 is the state before the action is executed
State2 is the state after the action.

Robot Task Planning

We said robot task planning is like finding a **path** in a graph (with nodes and links)

Nodes are states and links are actions



Robot Task Planning

The action **drop** can be defined by the following Prolog fact:

```
action( state( Pos, Pos, held),      % Robot and basket both at Pos, rubbish held by robot
        drop,                        % Action drop
        state( Pos, Pos, in_basket) ).      % After action: rubbish in basket
```

In the same way we defined **drop** we can define that the robot can move from any position **Pos1** to any position **Pos2** by the action **go**:

```
action( state( Pos1, Pos2, Pos3),
        go( Pos1, NewPos1),          % Go from Pos1 to NewPos1
        state( NewPos1, Pos2, Pos3)).
```

Robot Task Planning

When does the robot can pick the rubbish? How can we defien the **pickup** action?

```
action( state( Pos1, Pos2, floor(Pos1)),
        pickup,
        state( Pos1, Pos2, held)).
```

How can we specify the move '**push**'?

```
action( state( Pos1, Pos1, Pos2),          % Robot and basket both at Pos1
        push( Pos1, NewPos),               % Push basket from Pos1 to NewPos
        state( NewPos, NewPos, Pos2)).     % Robot and basket now at NewPos
```

Robot Task Planning

```
plan( State, State, [ ]).          %Start state and goal state are equal, nothing to do

plan( State1, GoalState, [ Action1 | RestOfPlan]) :-
   action( State1, Action1, State2),          % Make first action resulting in State2
   plan( State2, GoalState, RestOfPlan).      % Find rest of plan from State2
```

```
?- plan( state(door, corner2, floor(middle)), state( Rob, Bas, in_basket), Plan).

Rob = Bas, Bas = corner2,
Plan = [ go(door, middle), pickup, go(middle, corner2), drop]
```

Robot Task Planning

How does the planning program finds such plans?

Let us use the example situation where the robot can grasp the rubbish

How can the robot grasp the rubbish?

?- S0 = state(door, corner2, floor(middle)), plan(S0, state(_, _, held), Plan).

Which are the actions the robot can perform?

The only action possible in the initial state is **go**, which brings the robot to the new position **Pos**, where **Pos** is a *variable*. This means that the robot can go anywhere.

After **go** the state is: **state(Pos, corner2, floor(middle))**

action(state(Pos, corner2, floor(middle)), Action1, SecondState)

Can we apply any fact?

Robot Task Planning

The desired position for the robot is simply found through matching between the current goal and the Prolog fact about action **pickup**:

action(state(Pos, corner2, floor(middle)), Action1, SecondState) =
action(state(Pos1, Pos2, floor(Pos1)), pickup, state(Pos1, Pos2, held)).

This matching requires, among other things, that:

What does it mean?

Pos = Pos1, floor(middle) = floor(Pos1)

The rule requires that both the robot and the rubbish are at the same position. This causes the matching **Pos = middle.**

It is like if we ask Prolog
?- S1 = state(Pos, corner2, floor(middle)), plan(S1, state(_, _, held), Plan).

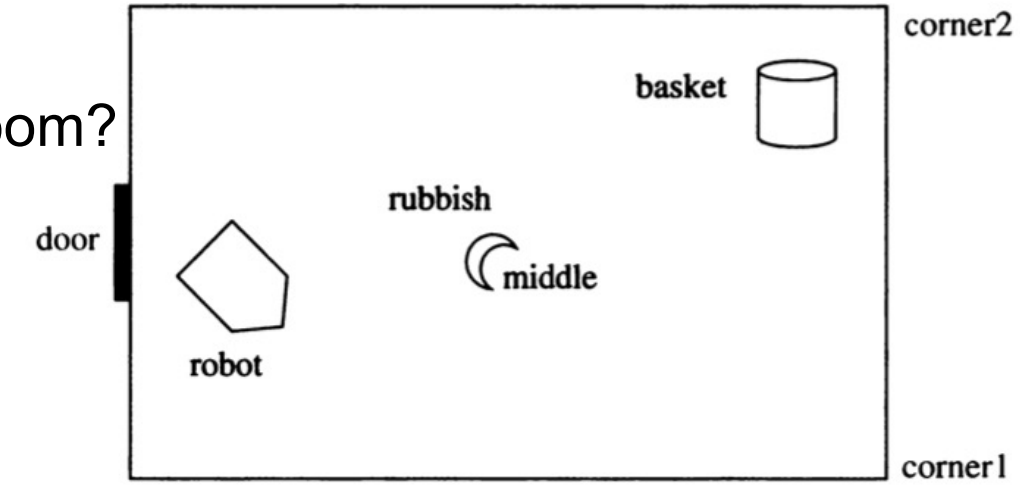
So which is the second resulting state?

SecondState = state(middle, corner2, held).

?- S0 = state(door, corner2, floor(middle)), plan(S0, state(_, _, held), Plan).
Plan = [go(door,middle), pickup]

Robot Task Planning

How can we find alternative plans for cleaning the room?



?- S0 = state(door, corner2, floor(middle)), plan(S0, state(_, _, in_basket), Plan).

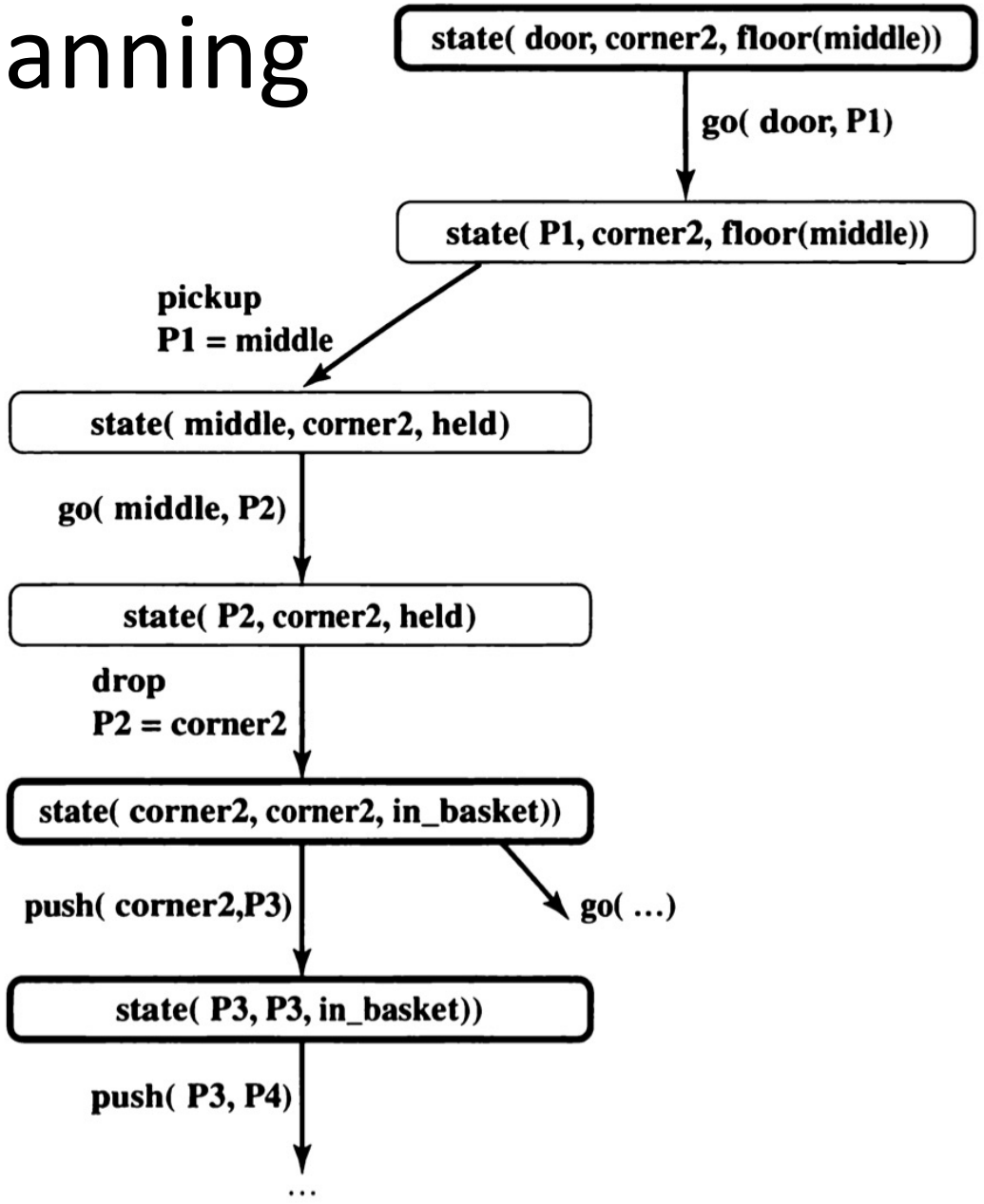
Plan = [go(door, middle), pickup, go(middle, corner2), drop] ;
Plan = [go(door, middle), pickup, go(middle, corner2), drop, push(corner2, _)] ;
Plan = [go(door, middle), pickup, go(middle, corner2), drop, push(corner2, _A), push(_A, _)] ;
Plan = [go(door, middle), pickup, go(middle, corner2), drop, push(corner2, _A), push(_A, _B), push(_B, _)]

The alternative plans are generated by simply appending to the first plan increasingly many pushing actions.

This is the same *depth-first* search we have introduced precendently.

Robot Task Planning

Execution Trace

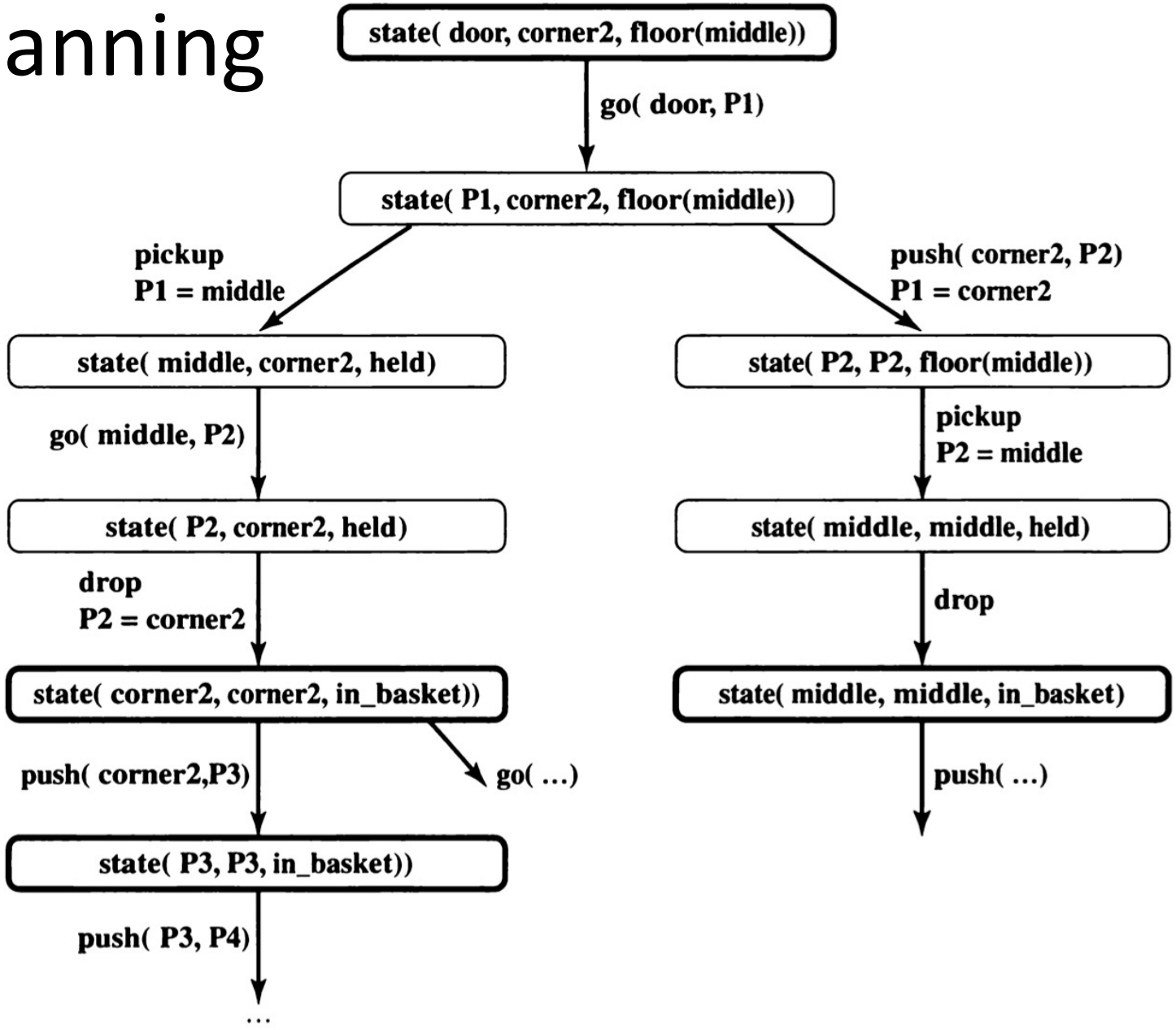


Robot Task Planning

Execution Trace

At the beginning we have two possibilities:

- picking up the rubbish
- pushing the basket



Robot Task Planning

Can we make Prolog generate alternative solutions so that all alternative shorter plans are generated before the longer ones?

We can use the same simple tachnique that we used precedently.

Using **conc** as such a generator

How can we formulate the question?

```
?- S0 = state(door, corner2, floor(middle)),      % Initial state
conc( Plan, _, _),                                % Generate plan templates, short first
plan( S0, state( _, _, in_basket), Plan).
```

```
Plan = [go(door, middle), pickup, go(middle, corner2), drop] ;
Plan = [go(door, corner2), push(corner2, middle), pickup, drop] ;
Plan = [go(door, middle), pickup, go(middle, corner2), drop, push(corner2, _)]
...
```

Robot Task Planning

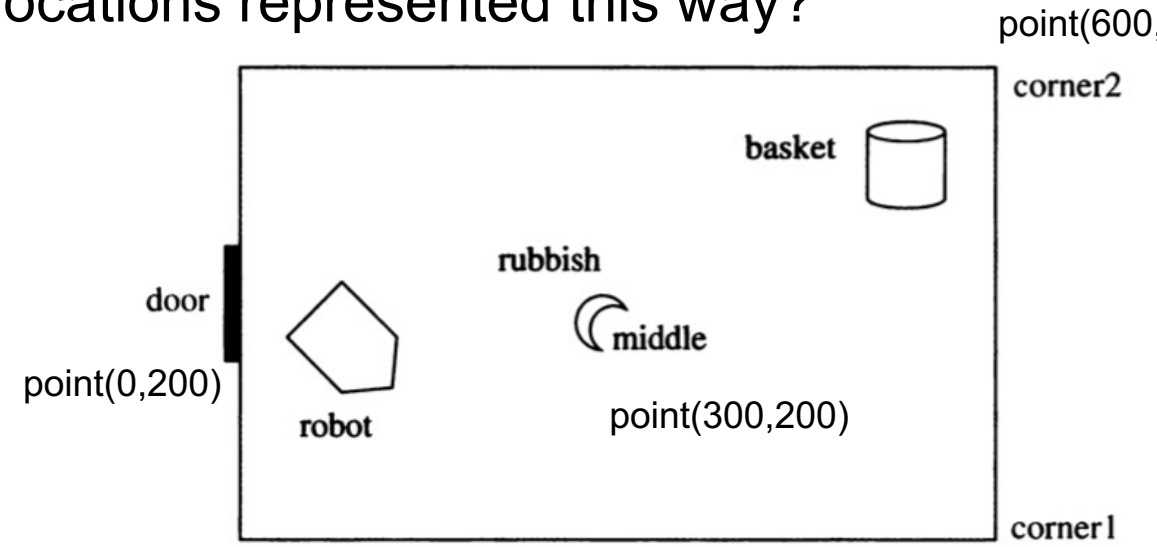
We denoted locations in our program by symbols like **middle** for ‘the middle of the room’.

Suppose that we wanted to work with numerical *x-y* coordinates in cm. Let the middle be at the point (300,200).

Can we use our planning program also with locations represented this way?

How will be the question?

?- S0 = state(point(0,200), point(600,400), floor(point(300,200))),
plan(S0, state(_, _, in_basket), Plan).



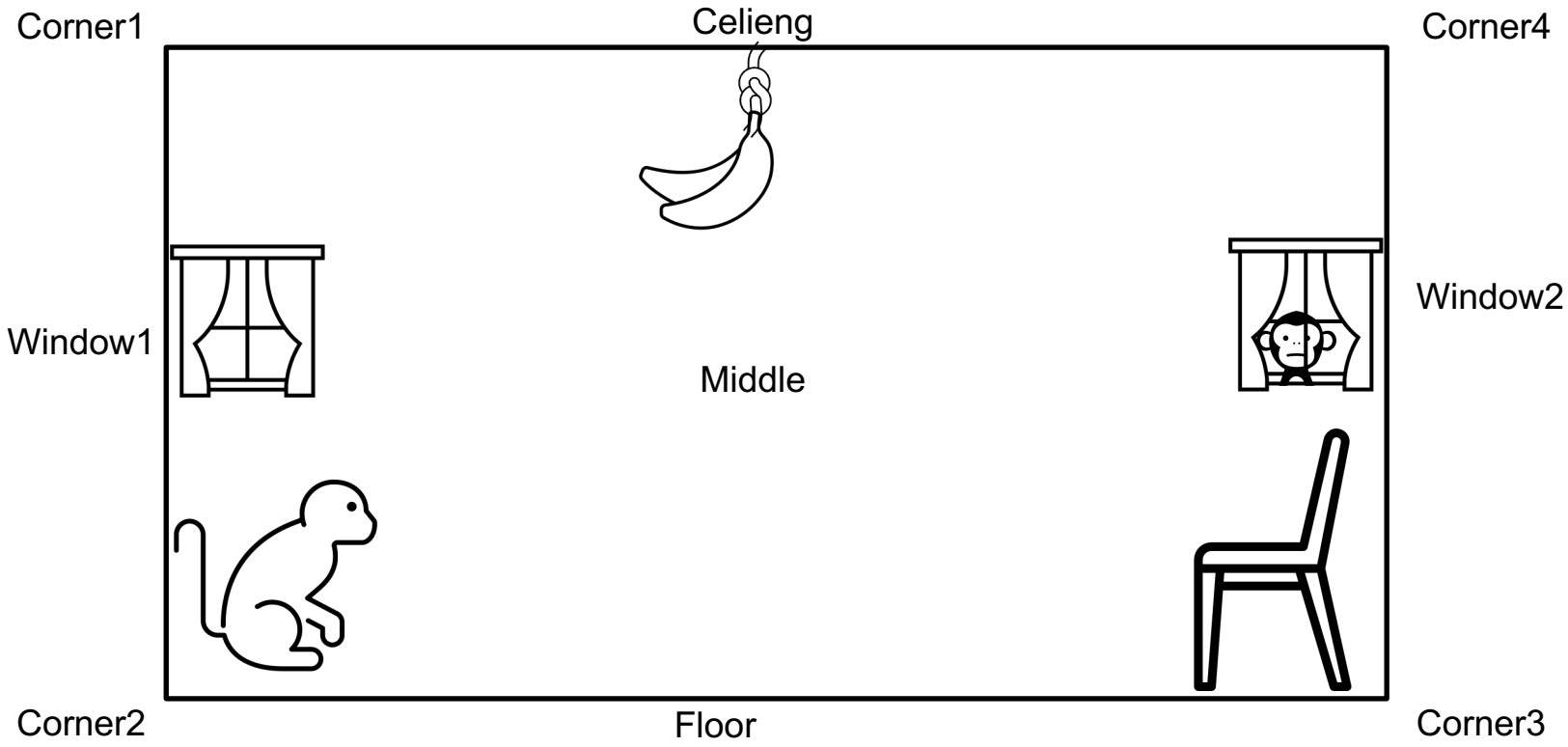
Plan = [go(point(0, 200), point(300, 200)), pickup, go(point(300, 200), point(600, 400)), drop] ;

Assignment

A variation of the Robot Task Planning

Actions:

- Go
- Push
- Climb
- Grasp
- Separate
- Eat
- Open
- Give



- Which are the actions that the monkey must perform to eat bananas? **Mandatory**
- Is there a sequence of actions that allows both monkeys to eat? **Optional**

Constraint Logic Programming

Constraint programming is a powerful paradigm for formulating and solving problems that can be naturally defined in terms of constraints among a set of variables.

Constraint satisfaction. Solving such problems consists of finding such combinations of calues of the variables that satisfy the constraints.

Constraint Logic Programming (CLP) combines the constraint approach with logic programming.

Programming with constraints is a kind of declarative programming and constraint satisfaction is embedded into logic programming languages such as Prolog.

Constraint Logic Programming

Initial example:

?- X + 1 = 5.

In Prolog this matching fails, so Prolog's answer is "false".

If the user's intention is that X is a number and "+" is arithmetic addition, then the answer X = 4 would be more desirable.

Using the built-in predicate **is** instead of "=" does not quite achieve this interpretation. But constraint logic programming (CLP) does.

Constraint Logic Programming

To use CLP in swipl we need to import an external library:
?- use_module(library(clpr)).

Initial example:

syntactic equality

?- X + 1 = 5.
false

?- {X + 1 = 5}.
X = 4.0

Constraint **L**ogic **P**rogramming over *Reals* ***CLP(R)***

Use curly brackets to define a numerical constraint

The curly brackets tell Prolog that equalities are to be handled as constraints by the constraint solver, and not by Prolog's own built-in predicate "=".

https://www.swi-prolog.org/pldoc/man?section=clpqr

Constraint Logic Programming

Consider the conversion of temperature from Fahrenheit to Centigrade.

$$\mathbf{C} = (\mathbf{F} - 32) \times 5/9$$

convert(Centigrade, Fahrenheit) :-

Centigrade is (Fahrenheit - 32)*5/9.

?- convert(C, 95).

C = 35

?- convert(35, F).

ERROR: Arguments are not sufficiently instantiated

To make the procedure work in both directions, we can test which of the arguments is instantiated to a number, and then use the conversion formula properly rewritten for each case.

Constraint Logic Programming

If we interpret the same formula as a numerical constraint we can easily obtain the desired result

convert(Centigrade, Fahrenheit) :-

{ Centigrade = (Fahrenheit - 32)*5/9 }.

?- convert(C, 95).

C = 35

?- convert(35, F).

F = 95

?- convert(C, F).

{ F = 32.0 + 1.8*C }

As the numerical calculation in this case is not possible, the answer is a general formula, meaning: the solution is a set of all F and C that satisfy this formula.

Constraint Satisfaction

A constraint satisfaction problem is stated as follows:

Given:

1. a set of *variables*,
2. the *domains* from which the variables can take values, and
3. *constraints* that the variables have to satisfy.

Find:
An assignment of values to the variables, so that these values satisfy all the given constraints.

Constraint Satisfaction

The constraint satisfaction approach in combination with logic programming has proved to be a very successful tool for a large variety of problems.

Typical esamples:
- scheduling,
- logistics,
- resource management in production,
- transportation
- placement.

These problems involve assigning resources to activities, like machines to job, people to rosters, crew to trains or planes, doctors and nurses to duties and wards, etc.

Constraint Satisfaction

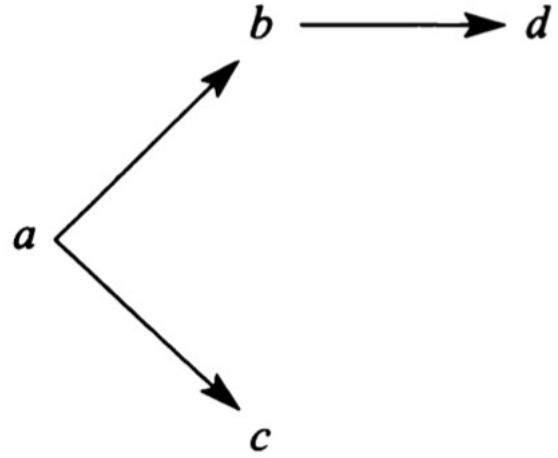
A typical example from scheduling. Let there be 4 tasks

Tasks	*Durations (hours)*
a	*2*
b	*3*
c	*5*
d	4

Precedence constraints

a has to precede *b* and *c*

b has to precede *d*



The problem is to find the start times *Ta*, *Tb*, *Tc*, and *Td* so that the finishing time *Tf* of the schedule is minimal. The earliest start time is 0.

Constraint Satisfaction

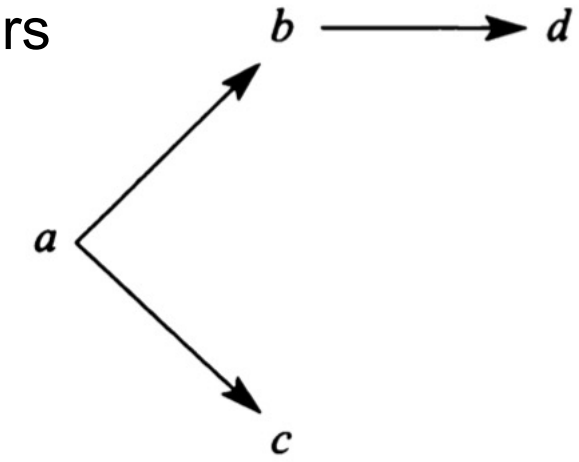
Variables: *Ta, Tb, Tc, Td, Tf*

Domains: All the variables' domains are non-negative real numbers

Constraints:

Ta >= 0	(task a cannot start before time 0)
Ta + 2 <= *Tb*	(task *a* which takes 2 hours precedes *b)*
Ta + 2 <= *Tc*	(*a* precedes *c)*
Tb + 3 <= *Td*	(*b* precedes *d)*
Tc + 5 <= *Tf*	*(c* finished by *Tf)*
Td + *4* <= Tf	*(d* finished by *Tf)*

Criterion: minimize *Tf*



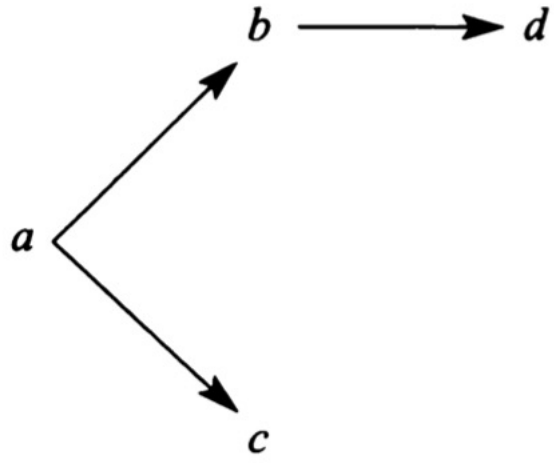
Constraint Satisfaction

In prolog we just need to load the clp library:
?- use_module(library(clpr)).

```
?- {Ta >= 0,
Ta + 2 =< Tb,
Ta + 2 =< Tc,
Tb + 3 =< Td,
Tc + 5 =< Tf,
Td + 4 =< Tf},
minimize( Tf).
```

```
Ta = 0.0,
Tb = 2.0,
Td = 5.0,
Tf = 9.0,
{_A>=0.0, _=2.0-_A, _A=<2.0, Tc=4.0-_A}

{ Tc =< 4.0}
{ Tc >= 2.0}
```



Constraint Satisfaction

CLP systems differ in the domains and types of constraints they can process. Families of CLP techniques appear under names of the form CLP(X) where X stands for the domain.

CLP(R) → the domains of the variables are real numbers, and constraints are arithmetic equalities, inequialities and disequalities over **real numbers**

CLP(Z) → Integers

CLP(Q) → Rational numbers

CLP(B) → Boolean domains

CLP(FD) → User-defined finite domain

Constraint Satisfaction

Consider the predicate **fib(N,F)** for computing **F** as the Nth Fibonacci number:

F(0) = 1, F(1) = 1, F(2) = 2, F(3) = 3, F(4) = 5, etc.

In general, for N > 1, F(N) = F(N – 1) + F(N – 2).

```
fib( N,F) :-
   N = 0, F = 1
   ;
   N = 1, F = 1
   ;
   N>=2,
   N1 is N -1, fib(N1,F1),
   N2 is N -2, fib(N2,F2),
   F is F1 + F2.
```

?- fib(4, F). What are we asking?

?- fib(5, F).

?- fib(6, F).

?- fib(N, 13). What are we asking?

Constraint Satisfaction

Consider the predicate **fib(N,F)** for computing **F** as the Nth Fibonacci number:

F(0) = 1, F(1) = 1, F(2) = 2, F(3) = 3, F(4) = 5, etc.

In general, for N > 1, F(N) = F(N – 1) + F(N – 2).

```
fib( N,F) :-
   { N = 0, F = 1 }
   ;
   { N = 1, F = 1 }
   ;
   { N>=2, N1 = N -1, N2 = N -2, F = F1 + F2},
   fib(N2,F2),
   fib(N1,F1).
```

?- fib(N, 4). ERROR: Stack limit

The program keeps trying to find two consecutive Fibonacci numbers F1 and F2 such that F1 + F2 = 4.

It does not realize that once their sum has exceeded 4, it will only be increasing and it can never become equal to 4.

Constraint Satisfaction

For all N, the N-th Fibonacci number F(N) >= N.

Therefore N1, F1, N2, F2 in our program must always satisfy the constraints:
F1 >= N1, F2 >= N2.

```
fib( N,F) :-
   { N = 0, F = 1 }
   ;
   { N = 1, F = 1 }
   ;
   { N>=2, N1 = N -1, N2 = N -2, F = F1 + F2,
      F1 >= N1, F2 >= N2},
   fib(N2,F2),
   fib(N1,F1).
```

?- fib(N, 4). false

Constraint Logic Programming

In **CLP(R)** real numbers are approximated by floating point numbers, whereas in **CLP(Q)**, Q is rational numbers, that is quotients between two integers.

Some constraints can be stated *exactly* as quotients of integers, whereas floating point numbers are only approximations.

```
?- { X = 2*Y, Y = 1-X }.
```

```
:- use_module(library(clpq)).
X = 2r3,          % X = 2/3
Y = 1r3.          % Y = 1/3
```

```
:- use_module(library(clpr)).
X = 0.6666666666666666,
Y = 0.3333333333333333.
```

Constraint Logic Programming

To use CLP(FD) in swipl we need to import an external library:
use_module(library(clpfd)).

Initial example:

syntactic equality

```
?- X + 1 = 5.
false
```

```
?- X + 1 #= 5.
X = 4.
```

Constraint **L**ogic **P**rogramming over *finite domains* ***CLP(FD)***

Constraints Operators as Prolog Predicate:

#=	equality
#\=	disequality
#>=	less equal to
#=<	greater equal to
#>	greater than
#<	less than

Triska, M. (2012, May). The finite domain constraint solver of SWI-Prolog. In *International Symposium on Functional and Logic Programming* (pp. 307-316). Springer, Berlin, Heidelberg. (https://116.203.249.179/swiclpfd.pdf) https://www.metalevel.at/drt.pdf

Constraint Logic Programming

Other examples:

?- 1 + 2 #= 7 – 4.

true.

?- 1 + Y #= 3.

Y = 2.

?- 2*X #= 5.

false.

?- X + Y #= 1 + 2.

X + Y #= 3. % a conditional solution; there is a solution only if X and Y satisty this conditi

Constraint Logic Programming

Let's define n_factorial(N, F), relating N to its factorial F.

```
n_factorial(0, 1).
n_factorial(N, F) :-
        N #> 0,
        F #= N * F1,
        N1 #= N -1,
        n_factorial(N1, F1).
```

```
?- n_factorial(42, F).

?- n_factorial(N, 120).

?- n_factorial(N, F).
```

Constraint Logic Programming

in – used to express that something is in a set

?- 4 in 1..5.

?- 4 in -4\/4. % A set containing -4 another set containing 4, 4 is in the union of those set

ins – Same as **in** but assigns all elements in a list to a range

[X,Y] ins 0..1.
X in 0..1,
Y in 0..1.

[1,2,3] ins 1..3.
?

[1,2,3] ins 1..20.
?

[1,2,3] ins 1..2.
?

Constraint Logic Programming

in – used to express that something is in a set

?- 4 in 1..5.

?- 4 in -4\/4. % A set containing -4 another set containing 4, 4 is in the union of those set

ins – Same as **in** but assigns all elements in a list to a range

[X,Y] ins 0..1.
X in 0..1,
Y in 0..1.

[1,2,3] ins 1..3.
true

[1,2,3] ins 1..20.
true

[1,2,3] ins 1..2.
false

Constraint Logic Programming

Every finite combinatorial task can be mapped to integers.

Example:

A chicken farmer also has some cows for a total of 30 animals, and the animals have 74 legs in total.

How many chickens does the farmer have?

```
?- Chickens + Cows #= 30,
   Chickens*2 + Cows*4 #= 74.
```

Chickens+2*Cows#=37,
Chickens+Cows#=30.

Constraint Logic Programming

sup is the supremum of the current domain of *Var*.

```
?- Chickens + Cows #= 30,
   Chickens*2 + Cows*4 #= 74,
   Chickens in 0..sup,
   Cows in 0..sup.
```

Chickens = 23, Cows = 7.

Constraint Logic Programming

Other predicates:

domain(L, Min, Max) % all the variables in L must have domains Min..Max.

```
domain(List, Min, Max) :-
        List ins Min..Max.
```

all_different(L) % all the variables in L must have different values.

labelling(Options, L) % generates concrete possible values of the variables in list L.

Constraint Logic Programming

?- domain([X,Y], 1, 2), labeling([ ], [X,Y]).

X = Y, Y = 1 ;
X = 1,
Y = 2 ;
X = 2,
Y = 1 ;
X = Y, Y = 2.

Constraint Logic Programming

“7-11 problem”

The total price of 4 items is $ 7.11.
The product of their prices is $ 7.11 as well.
What are the prices of the 4 items, and how many solutions are there?

```
?− Vs = [A,B,C,D], Vs ins 0..711,
        A * B * C * D #= 711*100^3,
        A + B + C + D #= 711,
        labeling([ff], Vs).
```

```
?− Vs = [A,B,C,D], Vs ins 0..711,
        A * B * C * D #= 711*100^3,
        A + B + C + D #= 711,
        A #>= B, B #>= C, C #>= D,
        labeling([ff], Vs).
```

Paul Pritchard and David Gries, The Seven-Eleven Problem, TR (1994)

Constraint Logic Programming

"N-Queens"

The task is to place N queens on an NxN chessboard such that none of the queens is under attack. This means that no two queens share the same row, column or diagonal.

```
n_queens(N, Qs) :-
   length(Qs, N),
   Qs ins 1..N,
   safe_queens(Qs).

safe_queens([]).
safe_queens([Q|Qs]) :-
   safe_queens(Qs, Q, 1),
   safe_queens(Qs).
```

```
safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-
   Q0 #\= Q,
   abs(Q0 - Q) #\= D0,
   D1 #= D0 + 1,
   safe_queens(Qs, Q0, D1).
```

```
?- n_queens(1,Qs),label(Qs).
?- n_queens(N,Qs),label(Qs).
```

https://www.swi-prolog.org/pldoc/man?section=clpfd-n-queens

https://www.metalevel.at/queens/ % To visualize an **animation** of the constraint solving process.

Questionnaire

You can start filling the course evaluation questionnaire.

It is mandatory to fill the questionnaire before registering to an exam.

https://www.unicam.it/studente/didattica/questionari-sulla-didattica

Next Lecture

Maude Overview

The goals of the Maude project are supporting formal executable specification, declarative programming, and a wide range of formal methods as means to achieve high-quality systems in areas much as: software engineering, networks, distributed computing, bioinformatics, and formal tool development.

http://maude.cs.illinois.edu/w/index.php/The_Maude_System