UNICAM
Università di Camerino
1336

# Logic and Constraint Programming

# PROLOG

Prof. Fabrizio Fornari

June 1, 2022

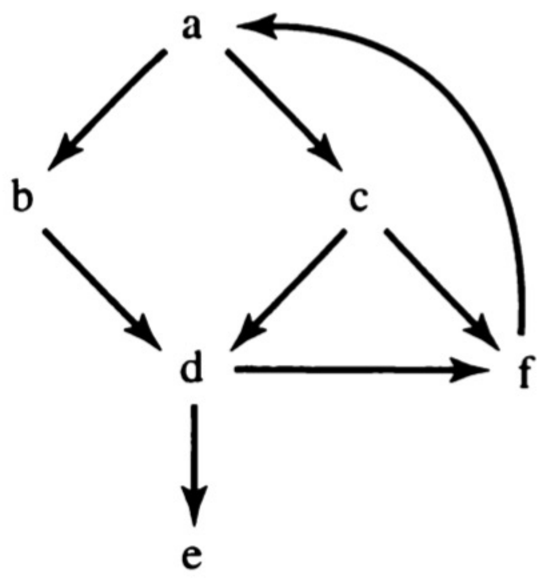
# Programming Examples

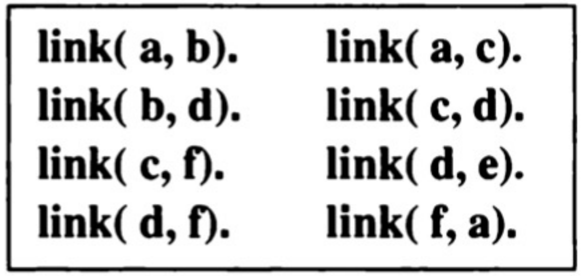
Graph structures are useful abstract representation for many problems.

Directed graph

The most common type of question concerning a graph is:

Is there a path from node X to node Y? And if there is, show the path.





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

# Programming Examples

Concrete practical questions are:

- How can I travel from Camerino to Rome?
- How can a user navigate from web page X to page Y?
- How many links of the type ‘P1 knows P2’ in a social network are needed to get from any person in the world to any other person, by following a chain of ‘knows’ links between people?

All these concrete questions can be answered by the same algorithms that work on abstract graphs where nodes and links have no concrete meaning

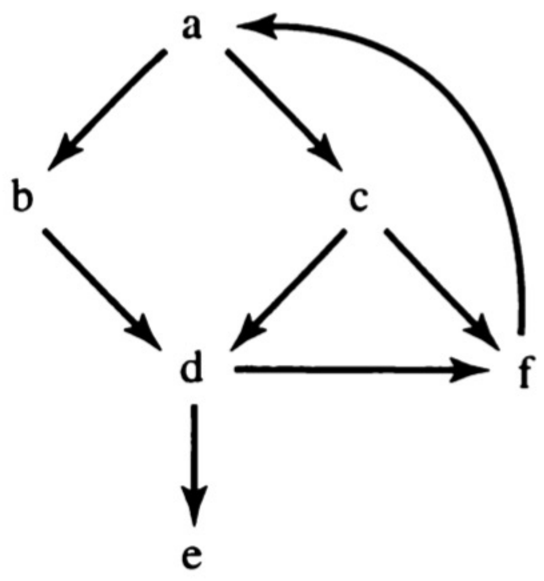
start
North Dakota
South Dakota
Nebraska
Kansas
Oklahoma
Texas
Minnesota
Iowa
Missouri
Arkansas
Louisiana
Wisconsin
Illinois
Mississippi
Michigan
Indiana
Kentucky
Tennessee
Alabama
Ohio
Georgia
Florida
South Carolina
North Carolina
Virginia
Pennsylvania
New York
New Hampshire
Vermont
Massachusetts
Maine
Rhode Island
Connecticu
New Jerse
Delawar
Marylan
West Virgini

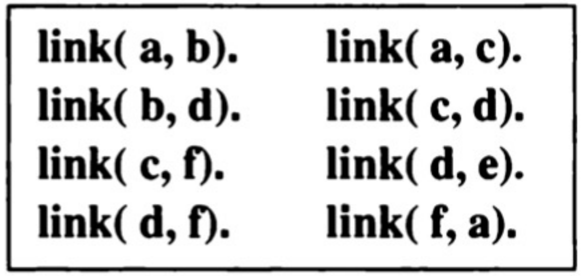
Milgram, S. (1967). The small world problem. *Psychology today*, *2*(1), 60-67.
The average path length for social networks of people in the United States.

# Programming Examples

Graph structures are useful abstract representation for many problems.

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





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

# Programming Examples

**path( StartNode, EndNode)**

it is true if there exists a path from StartNode to EndNote in the given graph.

**When does a path exists from StartNode to EndNote?**

A path exists if
1. StartNode and EndNode are both the same node, or
2. there is a link from StartNode to NextNode, and there is a path from NextNode to EndNode

How can we write these two rules in Prolog?

```
path( Node, Node).   % StartNode and EndNode are both the same node

path( StartNode, EndNode) :-
  link( StartNode, NextNode),
  path( NextNode, EndNode).
```

## Programming Examples

```
  path( Node, Node).     % StartNode and EndNode are both the same node

  path( StartNode, EndNode) :-
     link( StartNode, NextNode),
     path( NextNode, EndNode).
```

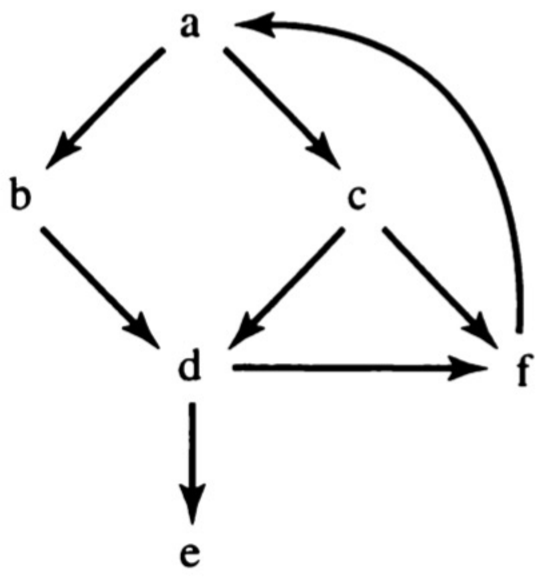
```
?- path(a,e).
true

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

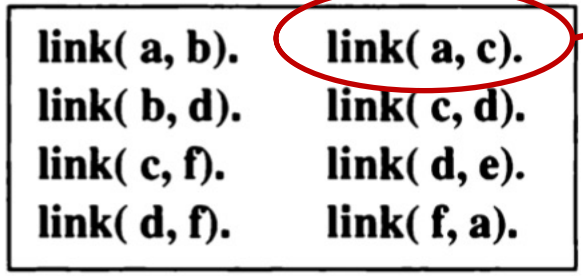
Which are the nodes X reachable from **a**?

# Programming Examples

?- path(a,X).

Which are the nodes X reachable from **a**?





| | |
|---|---|
| link( a, b). | link( a, c). |
| link( b, d). | link( c, d). |
| link( c, f). | link( d, e). |
| link( d, f). | link( f, a). |

X = a ;
X = b ;
X = d ;
X = e ;
X = f ;
X = a ;
X = b ;
…

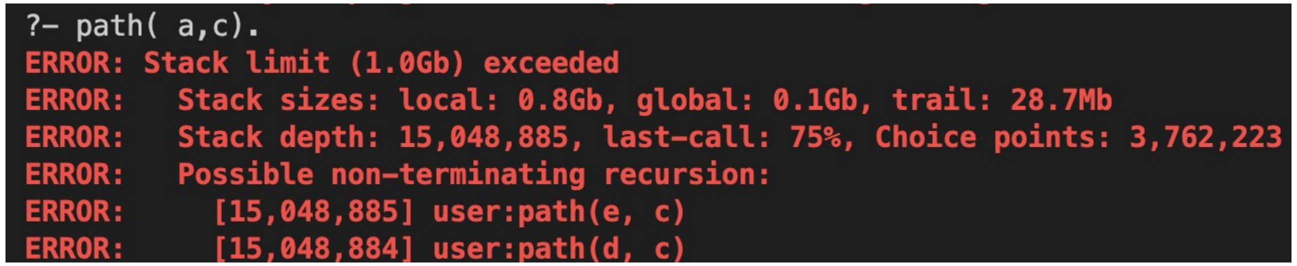
What happens?

Prolog has found that **a** is connected to **a** itself, **b**, **d**, **e**, **f**. Then it is back to **a** restarting the cycle.

It will never find that **a** is linked to **c, link( a,c)**.

Let us ask
**?- path( a,c).**

Why does it happen?



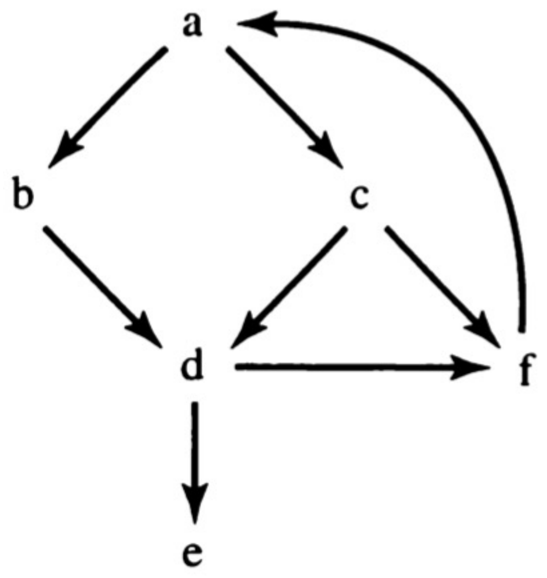
```
?- path( a,c).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.8Gb, global: 0.1Gb, trail: 28.7Mb
ERROR:   Stack depth: 15,048,885, last-call: 75%, Choice points: 3,762,223
ERROR:   Possible non-terminating recursion:
ERROR:     [15,048,885] user:path(e, c)
ERROR:     [15,048,884] user:path(d, c)
```

# Programming Examples

Which is the execution trace?

**?- path( a,c).**

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

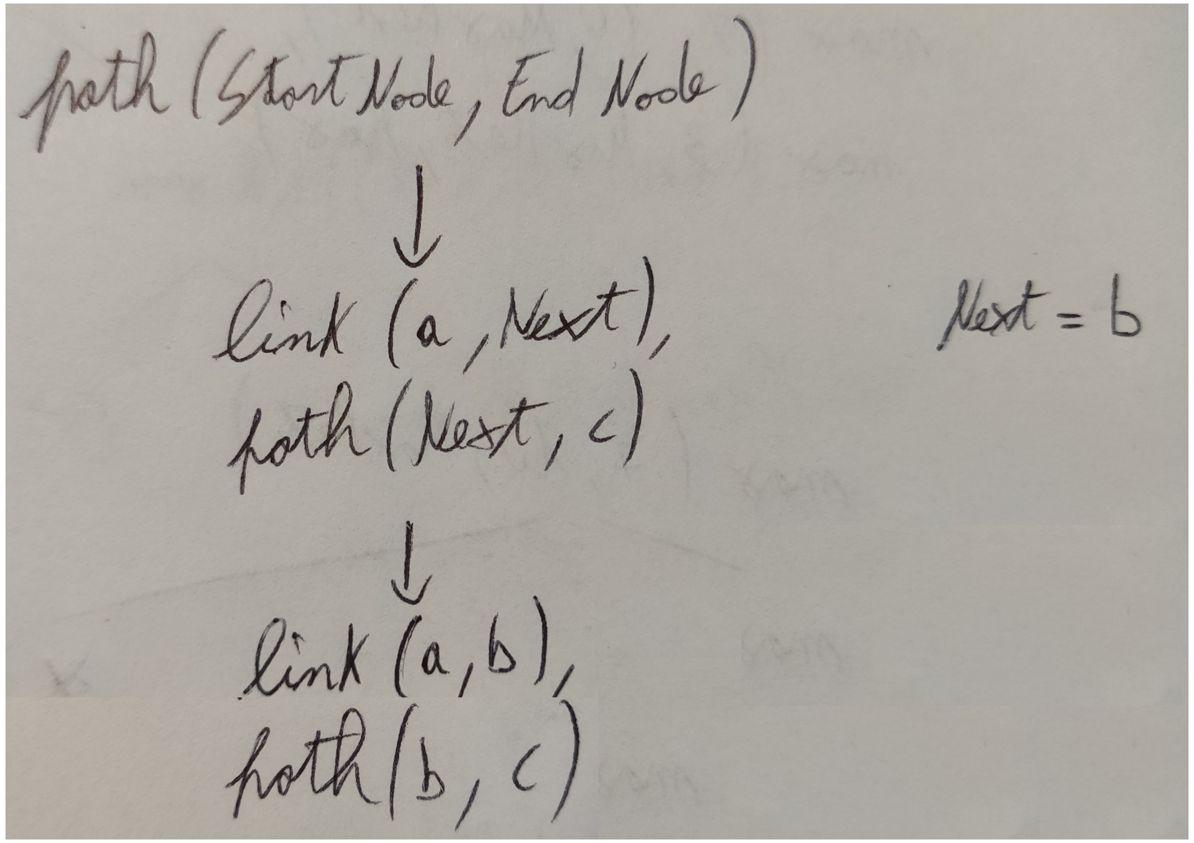


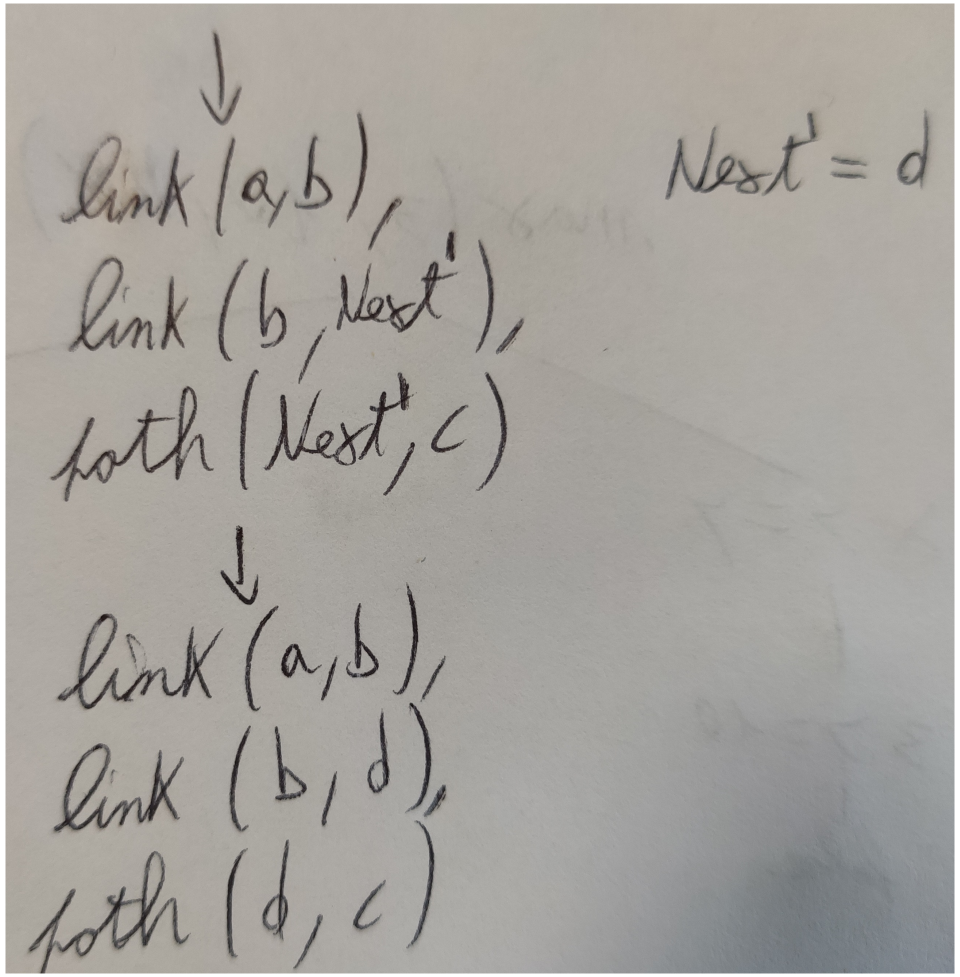
$$\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & \\
b \arrow[dr] & & c \arrow[dl] \arrow[dr] \\
 & d \arrow[d] \arrow[rr] & & f \arrow[ull, bend right] \\
 & e & &
\end{tikzcd}$$

```
path( Node, Node).

path( StartNode, EndNode) :-
   link( StartNode, NextNode),
   path( NextNode, EndNode).
```

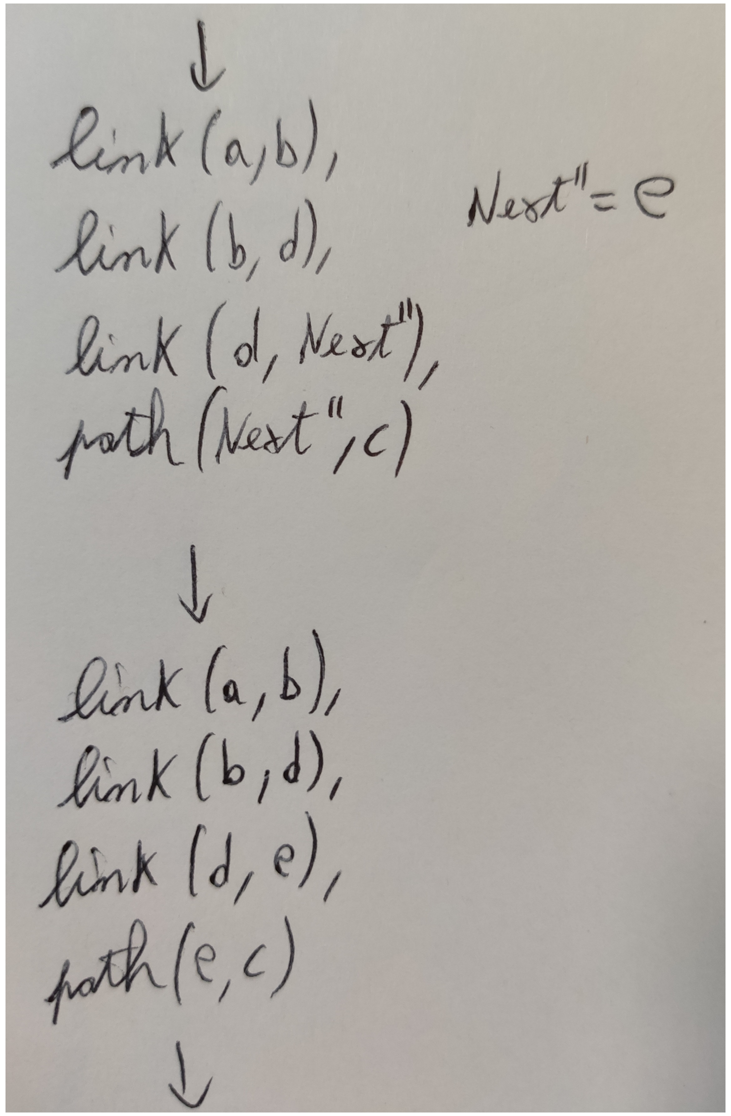
# Programming Examples

The execution trace of **?- path( a,c).**





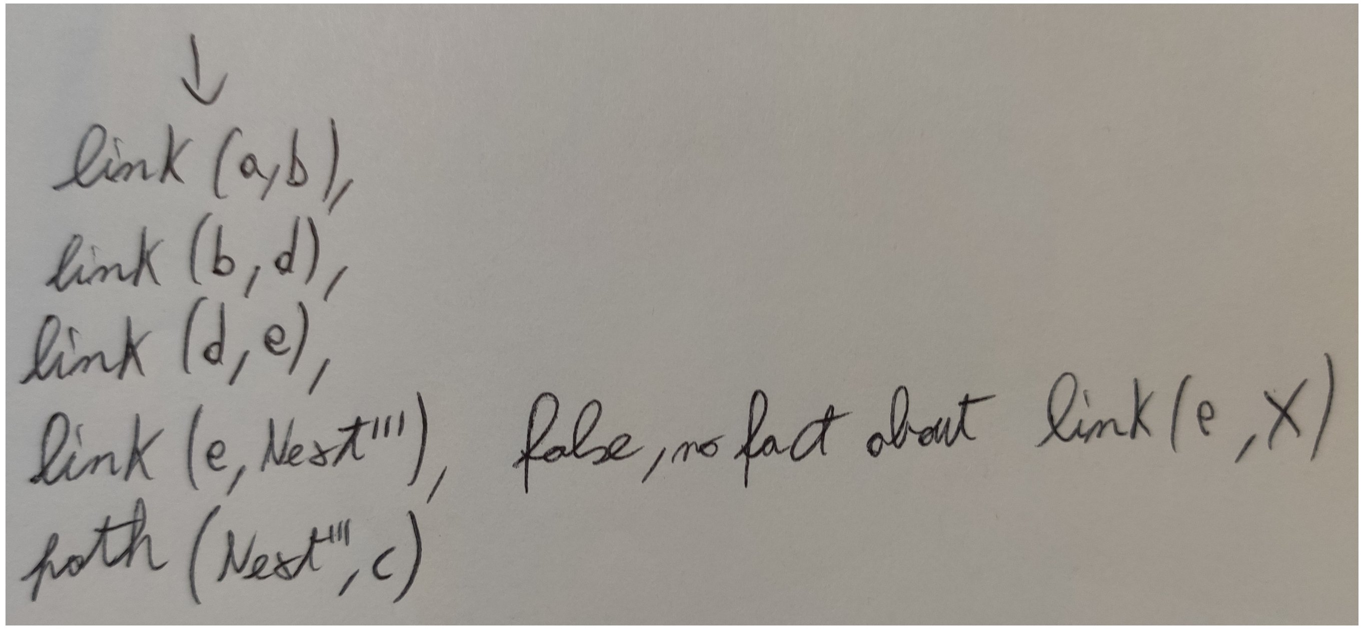
# Programming Examples



```
↓
link(a,b),
link(b,d),
link(d,Next''),
path(Next'',c)
```

Next'' = e

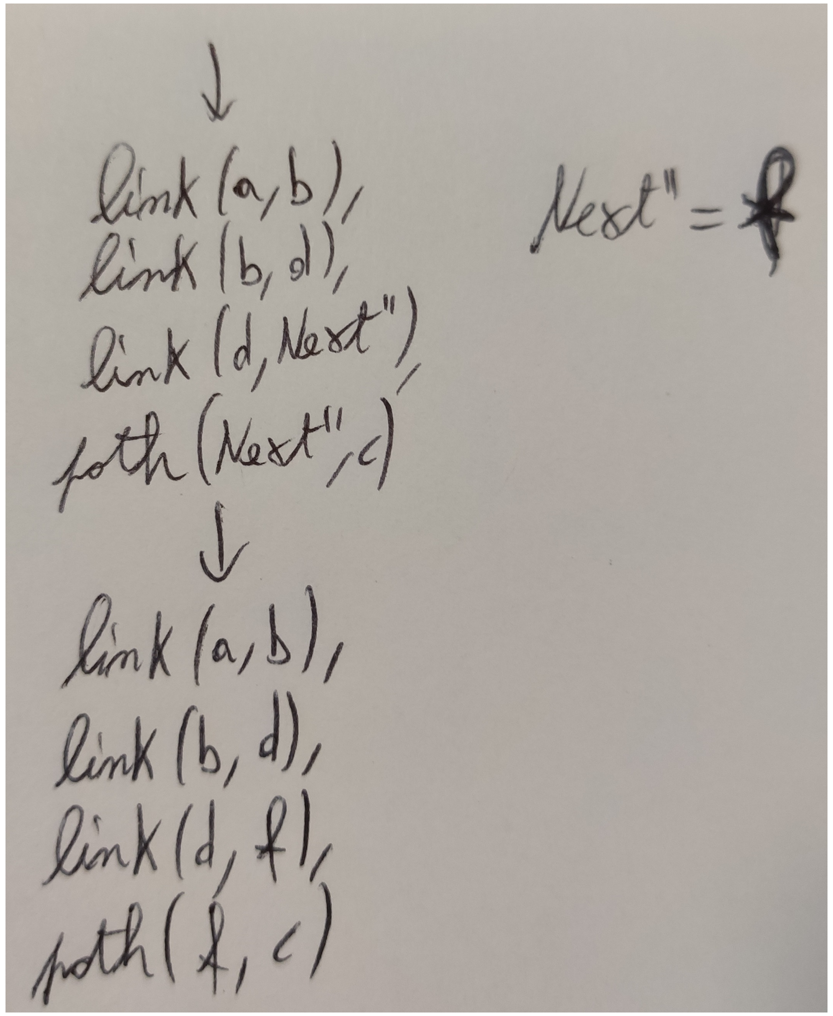
```
↓
link(a,b),
link(b,d),
link(d,e),
path(e,c)
↓
```



```
↓
link(a,b),
link(b,d),
link(d,e),
link(e,Next'''),
path(Next''',c)
```

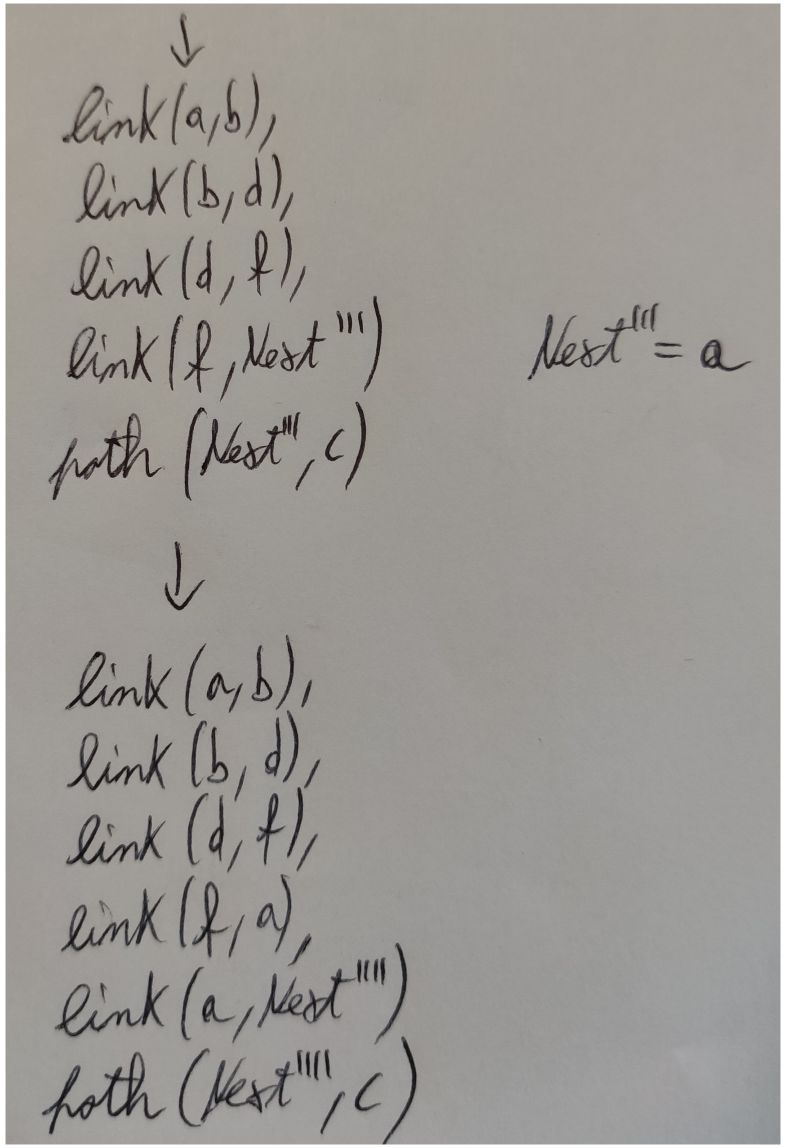
false, no fact about link(e,X)

# Programming Examples



```
↓
link(a,b),
link(b,d),
link(d,Next''),
path(Next'',c)
↓
link(a,b),
link(b,d),
link(d,f),
path(f,c)
```

Next'' = f



```
↓
link(a,b),
link(b,d),
link(d,f),
link(f,Next''')
path(Next''',c)
↓
link(a,b),
link(b,d),
link(d,f),
link(f,a),
link(a,Next'''')
path(Next'''',c)
```

Next''' = a

## Programming Examples

The style in which our program searches a graph is called ***depht-first***. Whenever there is a choice between alternatives where to continue the search, the program chooses a current deepest alternative.

**Each recursive call takes some memory**, that is why Prolog eventually runs out of memory. Because **Prolog has to remember where to return in the event that backtracking occurs**.

Our simple program with DFS (Depth-First Search), has a problem. **The problem does not occur when the graph to be searched is finite and has no cyclical path.**

The problem can be fixed in many ways, for example by **limiting the depth of search**, or by **checking for node repetition** on the currently expanded path.

## Programming Examples

Let us consider a more interesting version of the path predicate with a third argument added:

```
path( Start, End, Path)
```

This is true if **Path** is a path between nodes **Start** and **End** in the graph. **Path** is represented as a list of nodes.

```
?- path( a, e, Path).
Path = [ a, b, d, e]
```

# Programming Examples

```
?- path( a, e, Path).
Path = [ a, b, d, e]
```

How can we define this new path predicate?

This new path predicate can be defined by adding a third argument to our previous path program:

From

```
path( Node, Node).

path( StartNode, EndNode) :-
   link( StartNode, NextNode),
   path( NextNode, EndNode).
```

To

```
path( Node, Node, [Node]).

path( StartNode, EndNode, [StartNode | Rest]) :-
   link( StartNode, NextNode),
   path( NextNode, EndNode, Rest).
```

Remember, there is no confusion between the tow **path** predicates. Why?

# Programming Examples

```
path( Node, Node, [Node]).

path( StartNode, EndNode, [StartNode | Rest]) :-
   link( StartNode, NextNode),
   path( NextNode, EndNode, Rest).
```

Let us try the question, what are all the nodes reachable from node a?

**?- path( a, End, Path).**

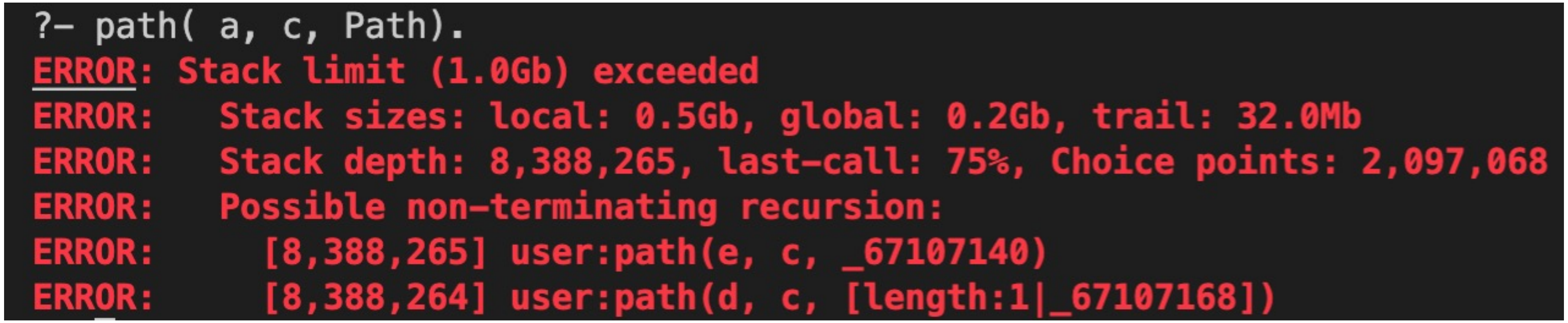
```
End = a, Path = [a] ;
End = b, Path = [a, b] ;
End = d, Path = [a, b, d] ;
End = e, Path = [a, b, d, e] ;
End = f, Path = [a, b, d, f] ;
End = a, Path = [a, b, d, f, a] ;
End = b, Path = [a, b, d, f, a, b] ;
End = d, Path = [a, b, d, f, a, b, d] ;
End = e, Path = [a, b, d, f, a, b, d, e] ;
End = f, Path = [a, b, d, f, a, b, d, f] ;
End = a, Path = [a, b, d, f, a, b, d, f, a] ;
End = b, Path = [a, b, d, f, a, b, d, f, a|...] ;
…
```

# Programming Examples

```
path( Node, Node, [Node]).

path( StartNode, EndNode, [StartNode | Rest]) :-
   link( StartNode, NextNode),
   path( NextNode, EndNode, Rest).
```

Let us try the question, what are all the nodes reachable from node a?

**?- path( a, c, Path).**



```
?- path( a, c, Path).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.5Gb, global: 0.2Gb, trail: 32.0Mb
ERROR:   Stack depth: 8,388,265, last-call: 75%, Choice points: 2,097,068
ERROR:   Possible non-terminating recursion:
ERROR:     [8,388,265] user:path(e, c, _67107140)
ERROR:     [8,388,264] user:path(d, c, [length:1|_67107168])
```

# Programming Examples

This time we can apply a trick to avoid getting stack into a loop.

What can we do?

We can limit the length of the searched path so that when the length has been reached, the search does not continue.

We can do this by means of the **conc** predicate that we previously defined

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

conc( [X | L1], L2, [X | L3] ) :-
conc( L1, L2, L3).
```

```
?- conc( L, _, _).
 L = [ ];
 L = [ _ ];
 L = [ _, _ ];
 L = [ _, _, _ ];
 …
```

Prolog generates through backtracking general lists of increasing length.

## Programming Examples

This time we can apply a trick to avoid getting stack into a loop.

What can we do?

We can limit the length of the searched path so that when the length has been reached, the search does not continue.

We can do this by means of the **conc** predicate that we previously defined

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

conc( [X | L1], L2, [X | L3] ) :-
conc( L1, L2, L3).
```

```
?- conc(Path, _, _), path(a, c, Path).
```

The trick is to first construct a general path (with all the elements uninstantiated) and then ask Prolog to find a path according to this template.

## Programming Examples

```
?- conc(Path, _, _), path(a, c, Path).
```

The trick is to first construct a general path (with all the elements uninstantiated) and then ask Prolog to find a path according to this template.

```
?- conc(Path, _, _), path(a, c, Path).
Path = [ a, c];
Path = [ a, c, f, a, c];
Path = [ a, c, d, f, a, c];
…
```

This trick forces Prolog to search the graph in ***breadth-first*** manner.

Prolog tries all the ways of finding a path of length 1, then all the paths of lengths 2, etc., until a path between the given nodes is found.

# Programming Examples

```
?- conc(Path, _, _), path(a, c, Path).
```

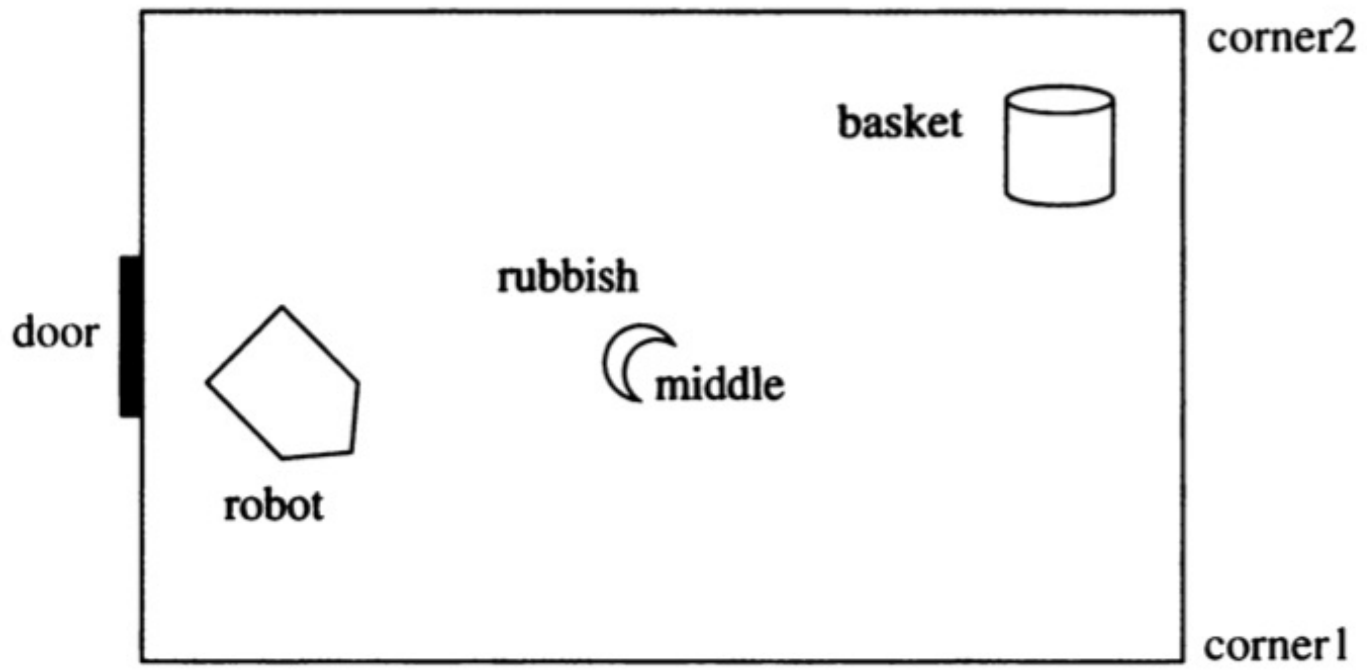
We used **conc** since we have already defined it.

A more elegant way of achievieng the same effect would be to define a special predicate **list(L)**, which is true if its argument L is a list. It can also be used to generate general lists of increasing length:

```
list( [ ]).                % Empty list is a list

list( [ _ | L ]) :-        % This is also a list if
   list( L).               % L is a list

?- list(Path, _, _), path(a, c, Path).
```

## Robot Task Planning

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

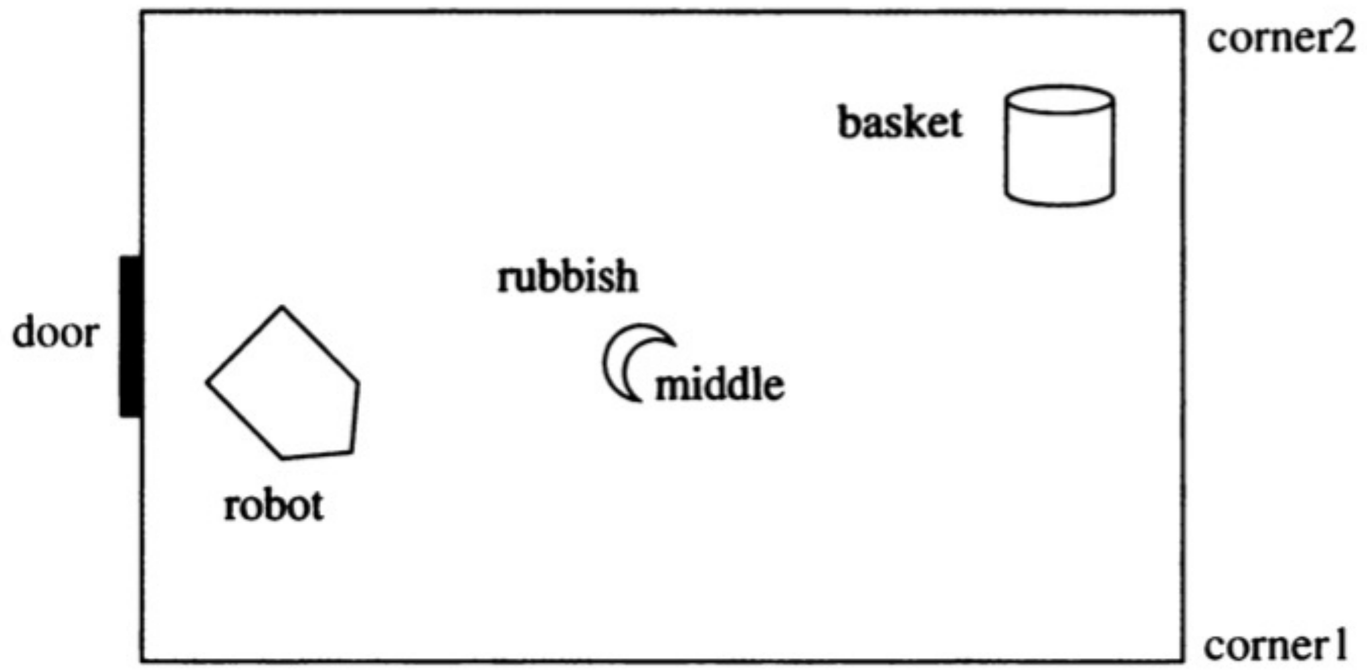


The **robot** is at the **door**, and there is **a piece of rubbbish in the middle of the room** and a **waste basket** in one of the **corners** of the room.

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.



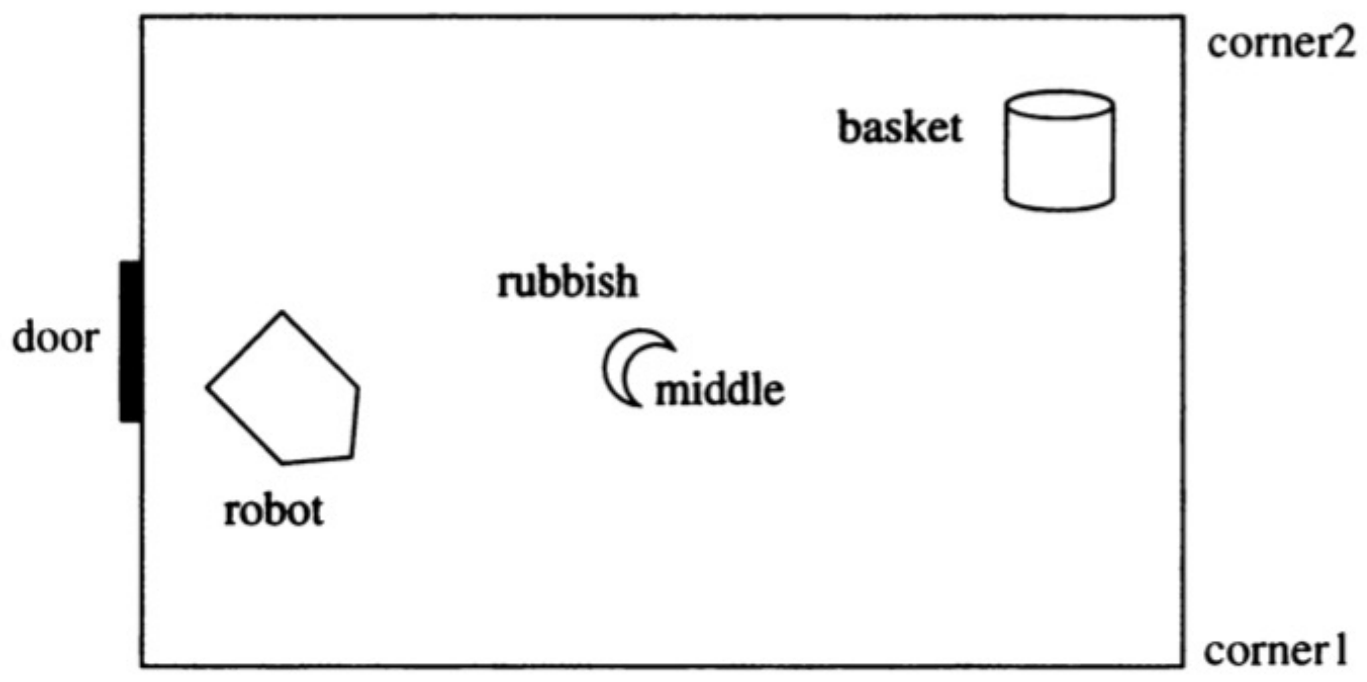
The robot, to execute commands may use its vision system, which recognizes objects and their locations.

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.



Which can be a plan to clean the room?

A plan could be:
go to the middle of the room,
pick up rubbish,
go to corner 2,
drop rubbish into basket

To develop such a planning program, we first have to design a representation of the robot’s world, and define the possible actions the robot can perform.

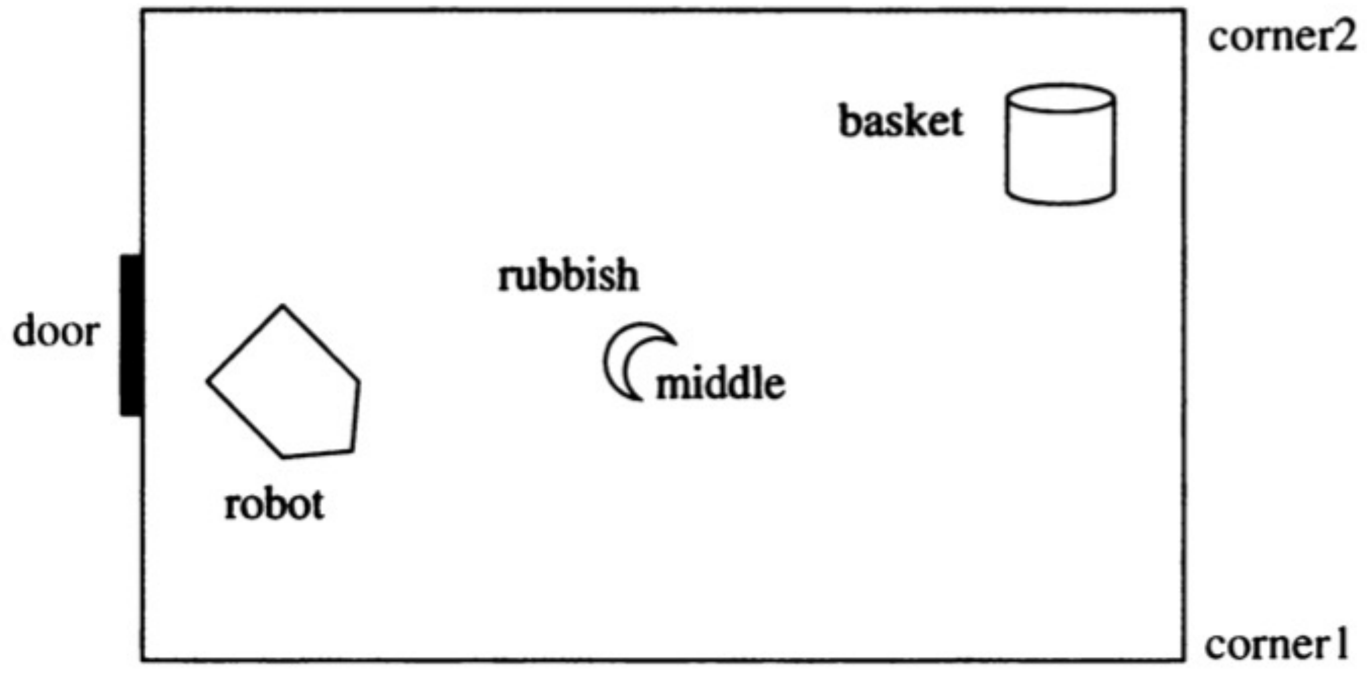
# Robot Task Planning

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

What does 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.



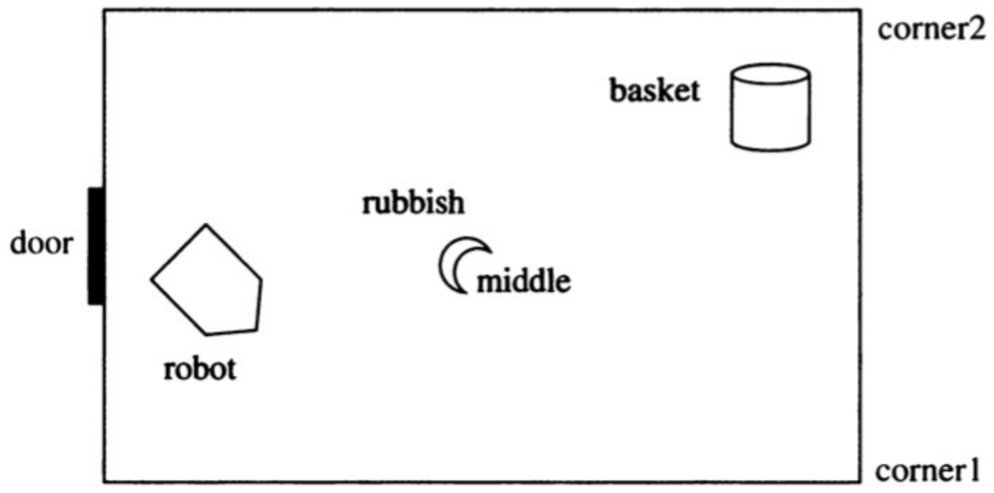
## Robot Task Planning

What about the rubbish? Where can it be?

The location of the rubbish can be **in_basket** or **held** (when the robot holds it).

We can also explicitly state that some locations are **on the floor**.

The rubbish is at **floor(middle)** and not just **middle**. This will allow the robot only to pick up rubbish from the floor and not from the basket.



# Robot Task Planning

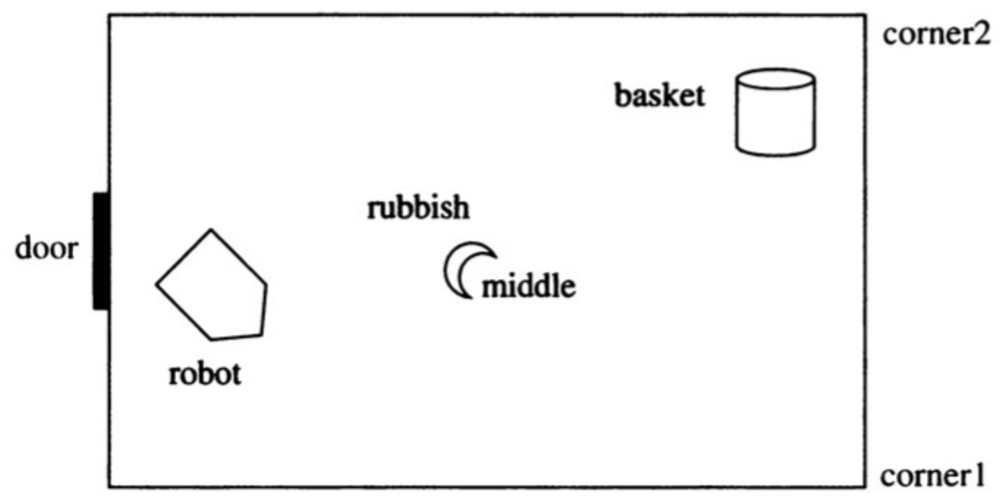
We can combine all the three pieces of location information into one structured object.

$$\textbf{state(}\ \underbrace{\textbf{door}}_{\text{robot}}\textbf{,}\ \underbrace{\textbf{corner2}}_{\text{basket}}\textbf{,}\ \underbrace{\textbf{floor(middle)}}_{\text{rubbish}}\textbf{)}$$

How can we specify the goal of our robot?

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)**

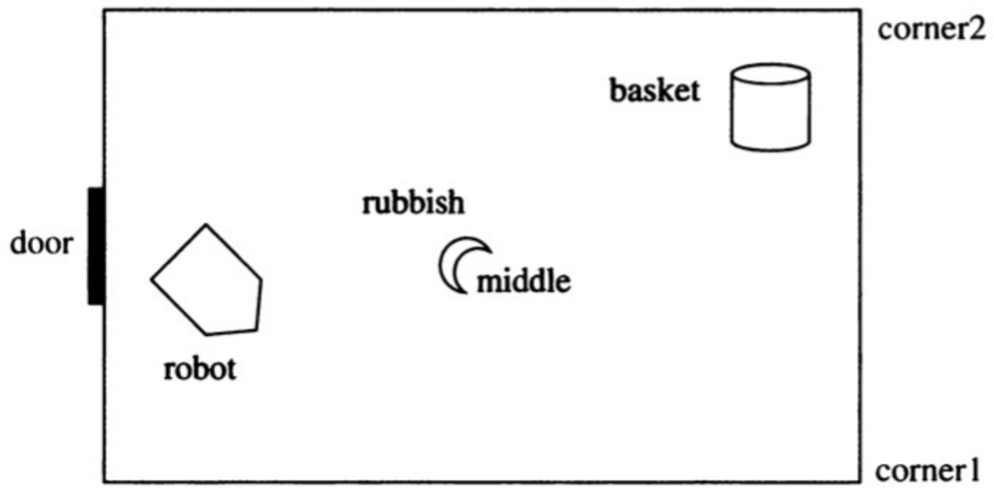


# Robot Task Planning

Now we specify the possible actions the robot can perform. These actions change the world from one state to another.

We will assume a ‘well-behaved’ robot that never drops rubbish on the floor, and never pushes rubbish around.

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**



# 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:

$$\begin{tikzcd} \textbf{State1} \arrow[r, "\textbf{Action}"'] & \textbf{State2} \end{tikzcd}$$

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

## Robot Task Planning

Based on the **action( State1, Action, State2)** relation, how can we define the action **drop**?

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
```

This clause says that after the action, both the robot and the basket remained at position **Pos**, and rubbish ended **in_basket**.

The **defined action is applicable to any situation that matches the specified state before the action**. Such a specification is also called an ***action schema***.

# Robot Task Planning

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)).
```

What happens to the robot?
What happens to the basket?
What happens to the rubbish?

- the robot goes from some position **Pos1** to **NewPos1**
- the locations of basket and rubbish **Pos2** and **Pos3** remain unchanged.

# 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)).
```

What happens to the robot?
What happens to the basket?
What happens to the rubbish?

- the location of robot and basket remains the same, respectively **Pos1**, and **Pos2**
- the locations of the rubbish changes from **floor(Pos1)** to **held**

## Robot Task Planning

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
```

Now that we defined several action. Which are the questions that our program can aswer?

Can the robot, starting from some initial state, clean rubbish into basket? And if yes, what is the plan, i.e. sequence of actions, to do that.

## Robot Task Planning

How can we formulate a predicate to express such a question?

**plan( StartState, GoalState, Plan)**

such a predicate is true if there exists a sequence of possible actions **Plan** that change **StartState** into **GoalState**.

This is like finding a **path** in a graph (as we have seen before).

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

Our program should answer with:

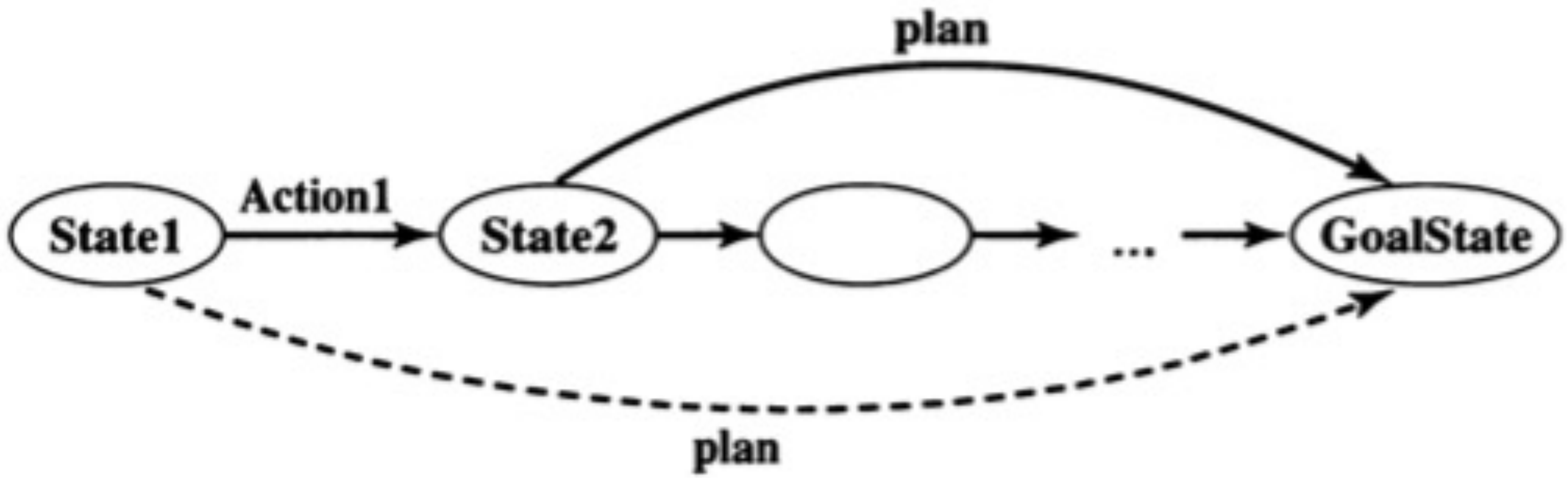
**Plan = [ go(door,middle), pickup, go(middle,corner2), drop]**

# Robot Task Planning

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

In the case of the robot, what are nodes and links?

Nodes are states and links are actions



# Robot Task Planning

Analogously to our program **path(StartNde, GoalNode, Path)** the definition of predicate **plan** can be based on two observations:

1. If goal state is equal to the start state then the goal is trivially achieved, no action is needed. We say this in Prolog by the clause:

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

2. In other cases, one or more actions are necessary. The robot can achieve the goal state from any state **State1**, if there is some action **Action1** from **State1** to some further actions **RestOfPlan**.

Try to define the second observation by yourself, taking inspiration from the second observation we specified for **path**

## Robot Task Planning

Analogously to our program **path(StartNde, GoalNode, Path)** the definition of predicate **plan** can be based on two observations:

1. If goal state is equal to the start state then the goal is trivially achieved, no action is needed. We say this in Prolog by the clause:

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

2. In other cases, one or more actions are necessary. The robot can achieve the goal state from any state **State1**, if there is some action **Action1** from **State1** to some further actions **RestOfPlan**.

```
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
```

## Robot Task Planning

```
plan( State, 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).
```

This question specifies completely the initial state, and partially the goal state. In the goal state, we insist that the location of the rubbish is **in_basket**, but we leave the position **Rob** of the robot and **Bas** of the basket unspecified.

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

Recursive formulation of **plan**.

## 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).
Plan = [ go( door,middle), pickup]
```

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))**

# 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:

**Pos = Pos1, floor( middle) = floor( Pos1)** What does it mean?

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

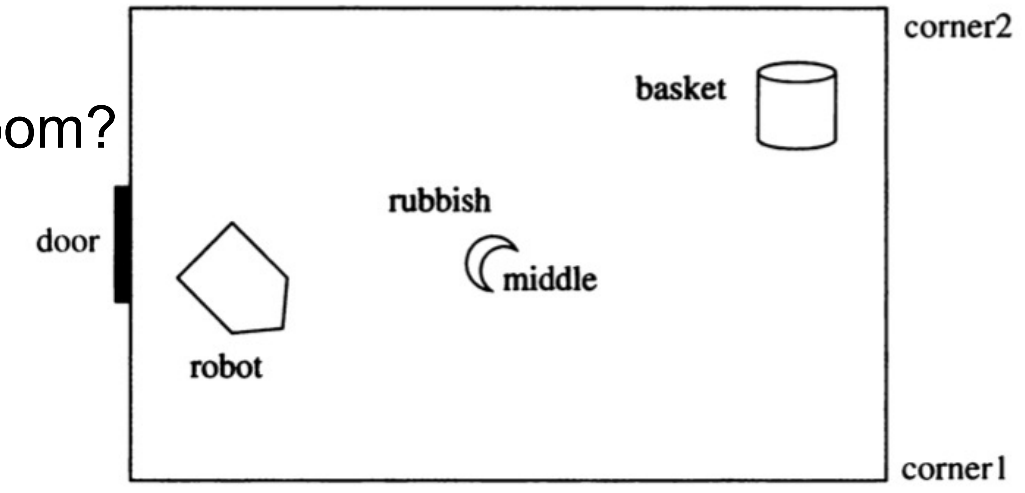
It is like if we ask Prolog
**?- S1 = state(Pos, corner2, floor(middle)), plan( S1, state( _, _, held), Plan).**

So which is the second resulting state?

**SecondState = state( middle, corner2, held).**

# 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.

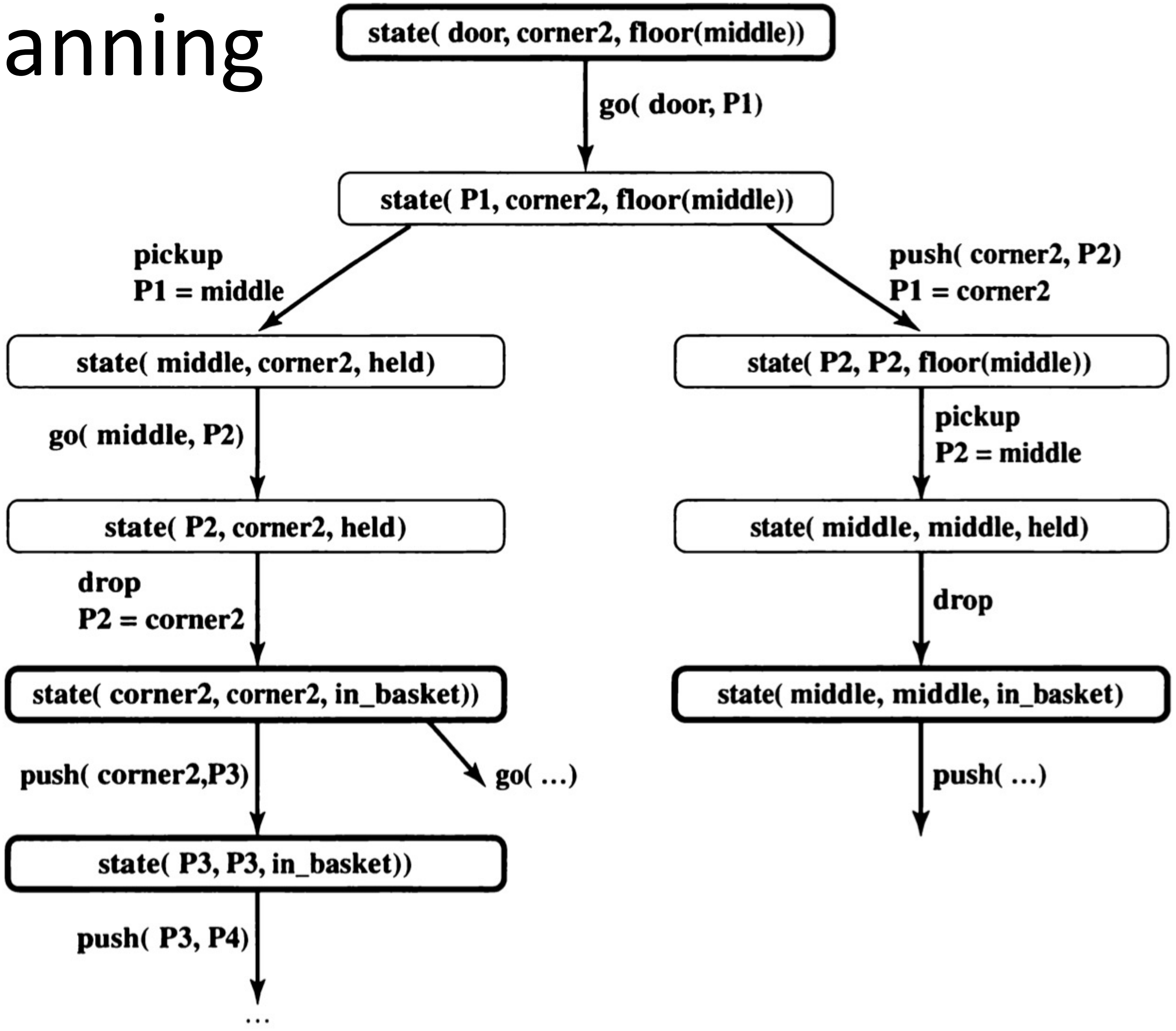
This is the same *depth-first* search we have introduced precendently.

## Robot Task Planning

Execution Trace

At the beginning we have two possibilities:

- picking up the rubbish
- pushing the basket


state( door, corner2, floor(middle))
go( door, P1)
state( P1, corner2, floor(middle))
pickup
P1 = middle
push( corner2, P2)
P1 = corner2
state( middle, corner2, held)
state( P2, P2, floor(middle))
go( middle, P2)
pickup
P2 = middle
state( P2, corner2, held)
state( middle, middle, held)
drop
P2 = corner2
drop
state( corner2, corner2, in_basket))
state( middle, middle, in_basket)
push( corner2,P3)
go( ...)
push( ...)
state( P3, P3, in_basket))
push( P3, P4)
...

# 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

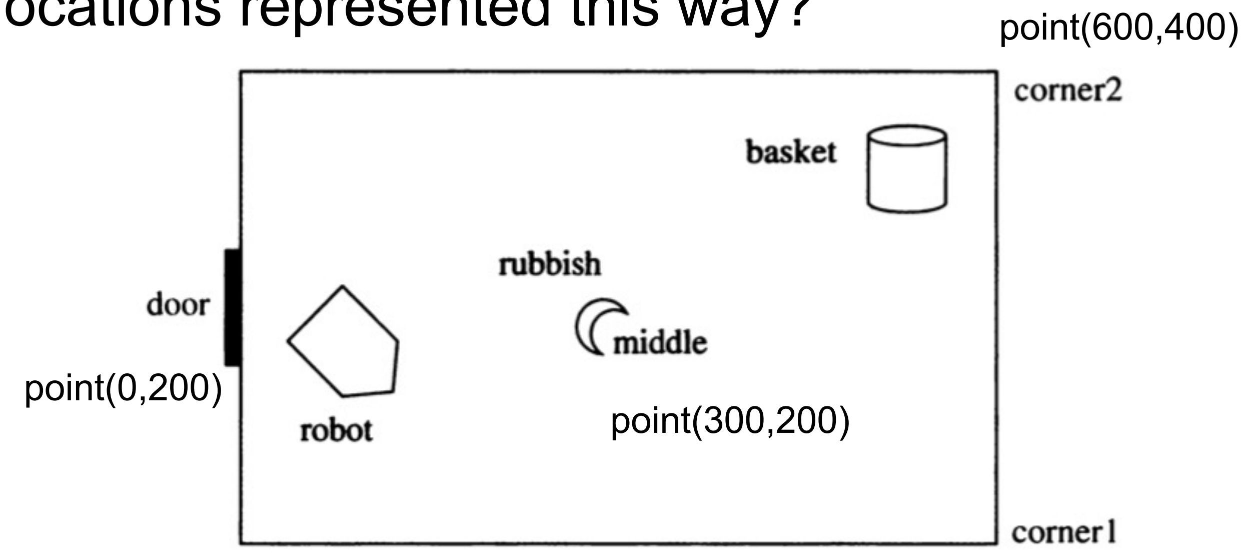
We denoted locations in our program by symbols like **middle** for ‘the middle of the room’.

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] ;

## 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

## Assignment

A variation of the Robot Task Planning

I will upload it on the wiki in the next days.