Introduction to Logic Programming and Prolog

Module of Logics and Artificial Intelligence course

Logic Programming

The Idea

Instead of writing down an algorithm to solve a specific problem, we can **describe** the environment and the problem and let **the program find** the solution, if exists, for us.

Logic Programming

Key features:

- Logic Programming is a **Programming Paradigm** based on **formal logic**;
- Logic is considered formal when its sentences are represented in the formal grammar, syntax and semantics of a logical language, suitable for **formal inference**;
- In a logic program, data and goals are expressed through clauses, in a declarative form.

Differences between Procedural and Declarative Languages

- In **Procedural languages** a sequence of instructions is defined in order to implement an algorithm. A typical procedural language is C99.
  The focus is on how the computation has to be performed.

- **Declarative languages** programs consist in a formal description of the wanted information. A typical declarative language is SQL.
  The focus is on what computation has to be performed.

A logic program consists in a sequence of facts (the true facts about the world), and rules, that let the program to infer implicit knowledge from the facts.

- The **problem** to solve (the **goal**) is **represented** in a **logical language**
- **The solving** of the problem is **realized** by **deductive reasoning**

A classical example

All men are mortal (a **rule**)
Socrates is a man (a **fact**)
It can be **inferred** that Socrates is mortal

Defining relations by facts

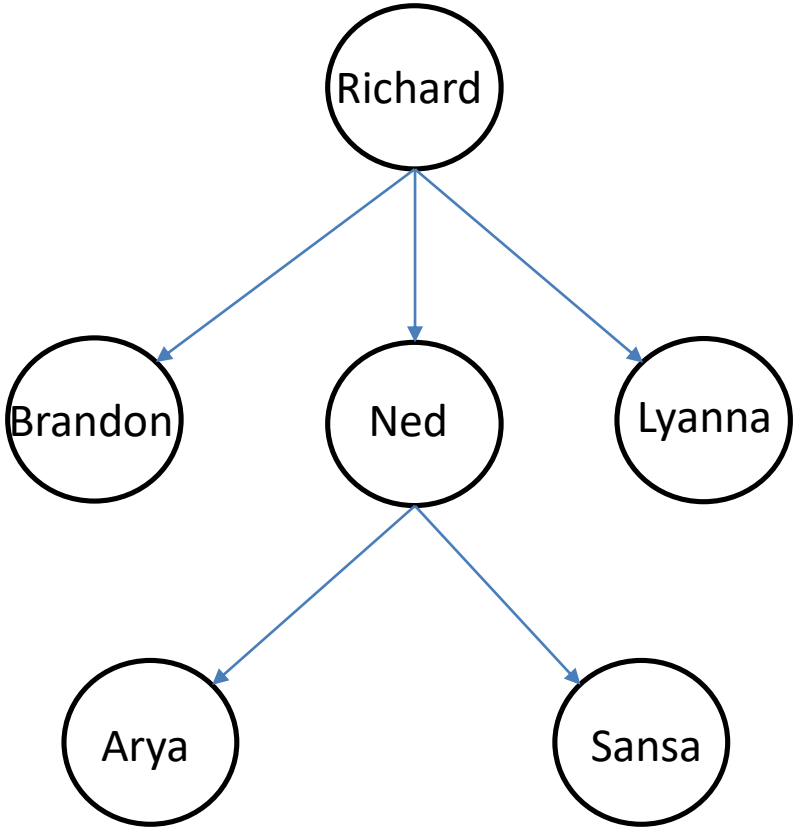
Prolog is specially well suited for solving problems that involve objects and relations between objects.

Let's see an example:
The fact that **Ned is a parent of Arya** can be written in Prolog as:

parent(ned,arya).

parent: name of the relation
ned, **arya**: arguments of the relation

A family tree example:




```
parent(richard,ned).
parent(richard,brandon).
parent(richard,lyanna).
parent(ned,arya).
parent(ned,sansa).
```

A closer view:

```
parent(richard,ned).
parent(richard,brandon).
parent(richard,lyanna).
parent(ned,arya).
parent(ned,sansa).
```

- The program consists of five clauses.

Each clause declares one fact about the parent relation, it is **a particular instance of the relation**.

- Order matters.

The **order of the clauses is important**.

- Each Prolog **program represents the knowledge base**

- Closed World Assumption

It is assumed that **whatever is neither explicitly nor implicitly expressed is false**

Query

A Prolog program can be run executing a query, the goal, against its knowledge base.

The query must be written in the same formal logic.

The program will answer true if it finds the clause as an asserted fact in the program

?- parent(richard,ned).
true

?- parent(arya,ned).
false

Query

In query clauses variables can be used:

Who is Ned's father?
?- parent(X,ned).
X = richard

The variables begins with a capital letter
The strings with an initial lower case letter are constants.

Who is Ned's child?
?- parent(ned,X).
X = arya

The program returned the first fact that makes the query true. If another solution is requested, it is possible to type a semicolon(;):
X = sansa

Query

It can be asked a broader question:
Find X and Y such that X is a parent of Y

?- parent(X,Y).

X= richard
Y= ned;
X= richard
Y= ned;
X= richard
Y= ned;
X= ned
Y= arya;
X= ned
Y= sansa.

Query

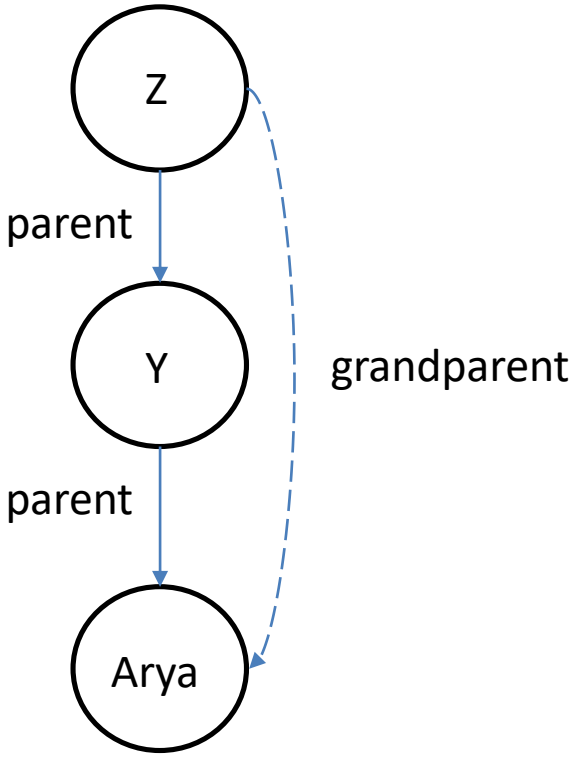
More complex questions can be posed: Who is the grandfather of Arya?
This query can be expressed by two simpler queries:

1) Find Y such that Y is parent of Arya.
2) Find Z such that Z is parent of Y

?- parent(Y,arya),parent(Z,Y).

The answer will be:

Z = richard
Y = ned.



Query

Other examples:

Who are Richard's grandchildren?

Or

Do Arya and Bran have a common parent?

Query

Other examples:

Who are Richard's grandchildren?

?- parent(richard,Y),parent(Y,X).

Or

Do Arya and Bran have a common parent?

?- parent(Z,arya),parent(Z,bran).

Recap

- A **relation** such as parent relation can be **easily defined** in Prolog
- A **prolog program consists of clauses**, each clause terminates with a full stop.
- The arguments of a relation can be:
  - concrete objects or **constants** (**atoms**)
  - general objects as X,Y,Z (**variables**)
  - other **relations**
- It is easy to query Prolog system about relations
- **Queries** to Prolog consists of one or more **goals**
- An **answer** can be positive or negative. If the answer is positive we say that the **goal** was **satisfiable** and the goal succeded. Otherwise the goal was **unsatisfiable** and the goal failed.
- If semicolon is typed, Prolog will find for further satisfying answers

Defining relations by rules

Consider this logical statement:

For all X and Y,
X is the father of Y
if X is a parent of Y, and X is a male.

This statement can be written in Prolog as:

father(X,Y) :- parent(X,Y),male(X).

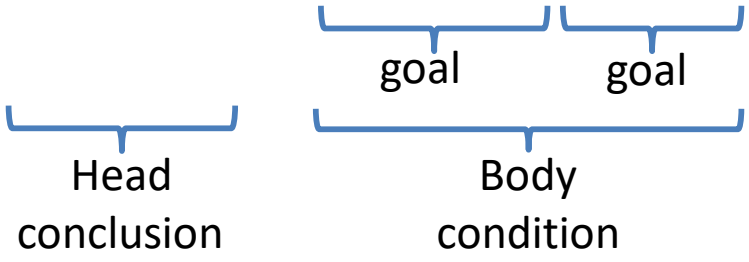
This kind of clauses are called **rules**.

The rules

A fact like parent(ned,arya) or male(ned) **is always and unconditionally true**

In a **rule**, instead, **a fact is true if some condition is satisfied**.

father(X,Y) :- parent(X,Y),male(X).



If the **goals** of the **body** (the **condition**) are true then the **head** (the **conclusion**) is a logical consequence of this and is true.

An example

?- father(ned,arya).
There are no facts about father in the program so applying the rule about father is needed.

The rule can be applied to any X and Y, in this case X has to be substituted with ned and Y with arya.
X = ned
Y = arya

The rule with instantiated variables becomes:
father(ned,arya):-parent(ned,arya),male(ned).
with
parent(ned,arya),male(ned) condition part
father(ned,arya) conclusion part

An example

The goal
father(ned,arya)

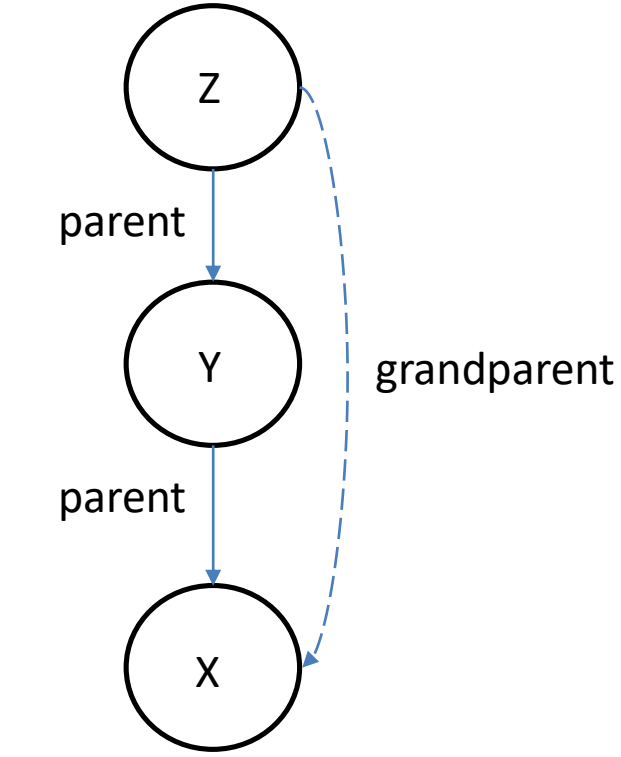
becomes
parent(ned,arya),male(ned)

?- parent(ned,arya)
True
and
?-male(ned)
True

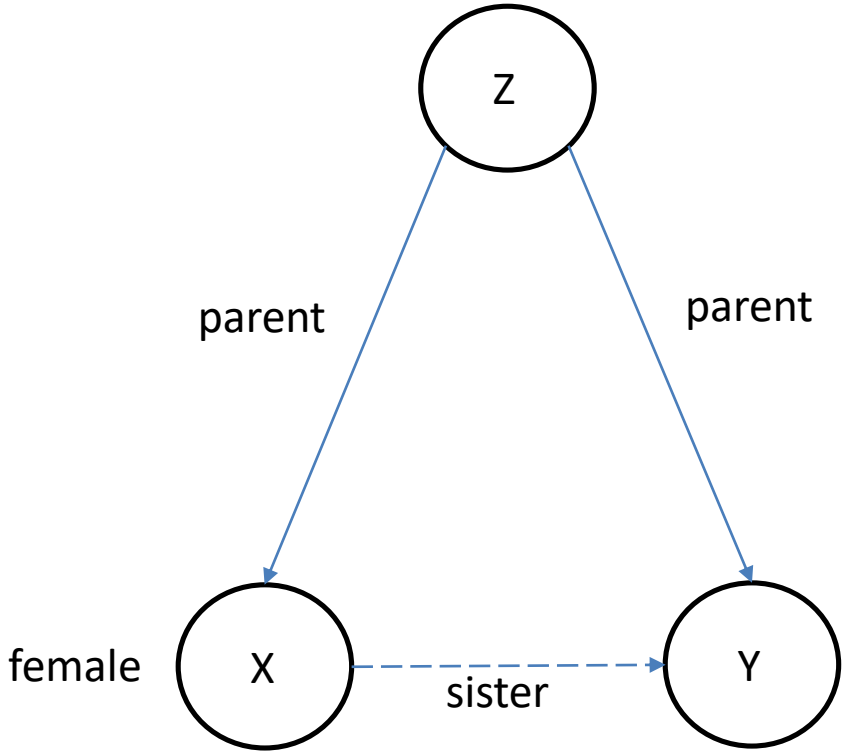
These goals can be easily found as facts in the program, hence Prolog can answer **true**

An example

Similarly we can express the **grandparent relation** and the **sibling relation**.



grandparent relation



sibling relation

Recap

- **Facts** declare things that **are unconditionally true**
- **Rules** declare things that **are true** depending **on a given condition**
- A Prolog **clause consists** of **head** and **body**
- The **body can be a composition of** comma separated **goals** (a conjuction of goals)
- This kind of clauses are called **Horn clauses**

What are Horn clauses?

A **Horn clause** is a clause with **exactly one positive literal**

A **clause** is a **disjunction of literals**

A **literal** is an **atomic formula** or its negation

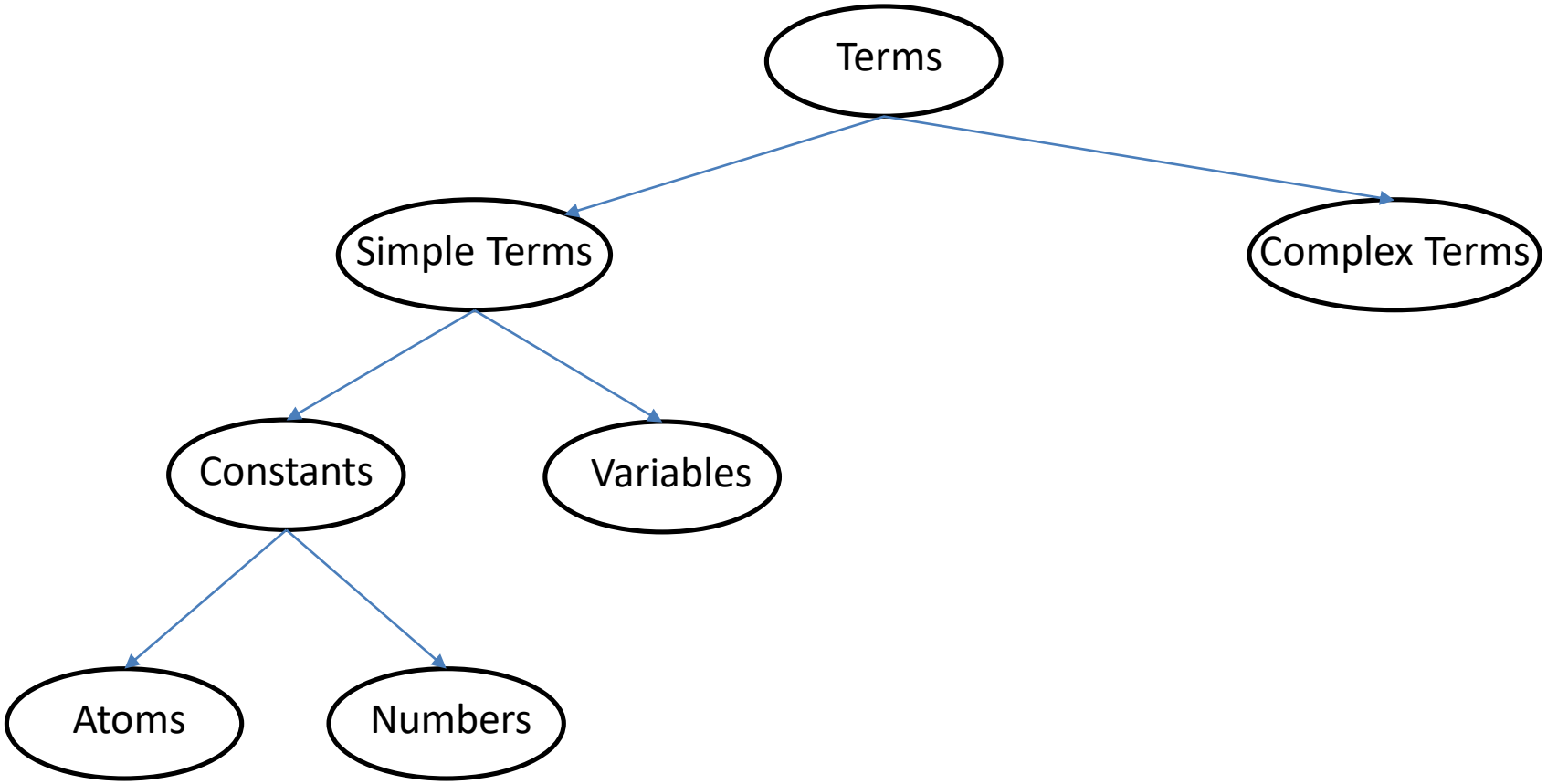
A **Horn formula** is a **conjunctive normal form formula** whose **clauses are all Horn**

Prolog: c:- a, b.
a.
b.

Horn formula: [c∨¬a∨¬ b] ∧ a ∧ b

Prolog Syntax

Facts, rules and queries are **built on Prolog terms**.



Atoms

- A sequence of characters of upper-case letters, lower-case letters, digits, or underscore, starting with a lowercase letter
  Examples: ned, arya, parent

- An arbitrary sequence of characters enclosed in single quotes
  Examples: ‘X', Y', '@$%‘

- A sequence of special characters
  Examples: : , ; . :-

Numbers

- **Integers**: 12, -34, 22342
- **Floats**: 34573.3234, 0.3435

Variables

A sequence of characters of uppercase letters, lower-case letters, digits, or underscore, starting with either an uppercase letter or an underscore

Examples:
X, Y, Variable, Vincent, _tag

Complex Terms

Atoms, numbers and variables are building blocks for complex terms

Complex terms are built out of a **functor** directly followed by a sequence of arguments

- Arguments are put in **round brackets**, separated by commas
- The functor must be an atom

Examples:

- male(ned)
- parent(ned, arya)
- granchild(arya, Z)
- father(X,father(father(father(arya))))

Arity

The number of arguments a complex term has is called its **arity**

Examples:

female(sansa)	has arity 1
sibling(arya,bran)	has arity 2
father(X,father(ned,Z))	has arity 2

Several predicates can be defined with the same functor but **with different arity**, and Prolog would treat this as **different predicates**

Arity of a predicate is usually indicated with the suffix "**/**" followed by a number to indicate the arity

female/1
sibling/2
father/2

Unification and Proof Search in Prolog

Module of Logics and Artificial Intelligence course

Unification – Starting example recap

The knowledge base contains only one fact:

male(ned).

the simplest query is:

male(X).

Prolog **unifies** the two complex terms by **instantiating** the variable **X** with **ned.**

Unification

Unification is possible between two terms:

- if they are the same term
- if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal

Examples:

ned and **ned** unify
42 and **42** unify
male(ned) and **male(ned)** unify
ned and **arya** do not unify
male(ned) and **male(richard)** do not unify

Instantiations

When Prolog unifies two terms, it instantiates any needed variable and and that instantiations are used afterwards.

What Prolog responds to:

?-X=ned.

?-male(X)=male(ned).

?-father(ned,X)=father(X,arya).

?-father(ned,X)=father(Z,arya).

?-X=ned,X=arya.

Instantiations

When Prolog unifies two terms, it instantiates any needed variable and and that instantiations are used afterwards.

What Prolog responds to:

?-X=ned.

X = ned.

?-male(X)=male(ned).

X = ned.

?-father(ned,X)=father(X,arya).

false.

?-father(ned,X)=father(Z,arya).

X = arya.

Z= ned.

?-X=ned,X=arya.

false.

Recap

- **If T1 and T2 are constants**, then T1 and T2 **unify** if they are the **same atom**, or the same number

- **If T1 is a variable and T2 is any type of term**, then T1 and T2 **unify**, and **T1 is instantiated to T2** (and vice versa)

- **If T1 and T2 are complex terms** then they **unify if**:
  1. They have the **same functor and arity**, and
  2. all their **corresponding arguments unify**, and
  3. the **variable instantiations are compatible**.

More examples

What Prolog responds to:

?-k(s(g),Y) = k(X,t(k)).

?-k(s(g),t(k)) = k(X,t(Y)).

?-father(X,X)=father(ned,arya).

More examples

What Prolog responds to:

?-k(s(g),Y) = k(X,t(k)).
X=s(g)
Y=t(k)

?-k(s(g),t(k)) = k(X,t(Y)).
X=s(g)
Y=k

?-father(X,X)=father(ned,arya).
False.

Prolog and unifications

What Prolog responds to:
?-male(X)=X.

Do these terms unify or not?

Prolog and unifications

What Prolog responds to:
?-male(X)=X.

Do these terms unify or not?

X=male(male(male(male(male(male(male(male(male(male(male(mal
e(male(male(male(male(male(male(male(male(male(male(male(mal
e(male(male(male(male(male(male(male(male(male(male(male(mal
e(male(male(...
Infinite terms..

In Prolog:
X = male(X).

Prolog unifies without **occurs check**

Unification without occurs check

Unification without occurs check can lead to **unsound inference**.

Example
A resolution proof can be found for the non-theorem:

$$\forall x \exists y\ parent(y, x) \Rightarrow \exists y \forall x\ parent(y, x)$$

because of the lack of occurs check make **X** unifiable with **f(X)** after skolemization.

Prolog – unification with occurs check:
?- unify_with_occurs_check(male(X), X).
False.

SLD resolution

Selective **L**inear **D**efinite clause resolution is a **sound** and **complete inference rule** for **Horn clauses**.

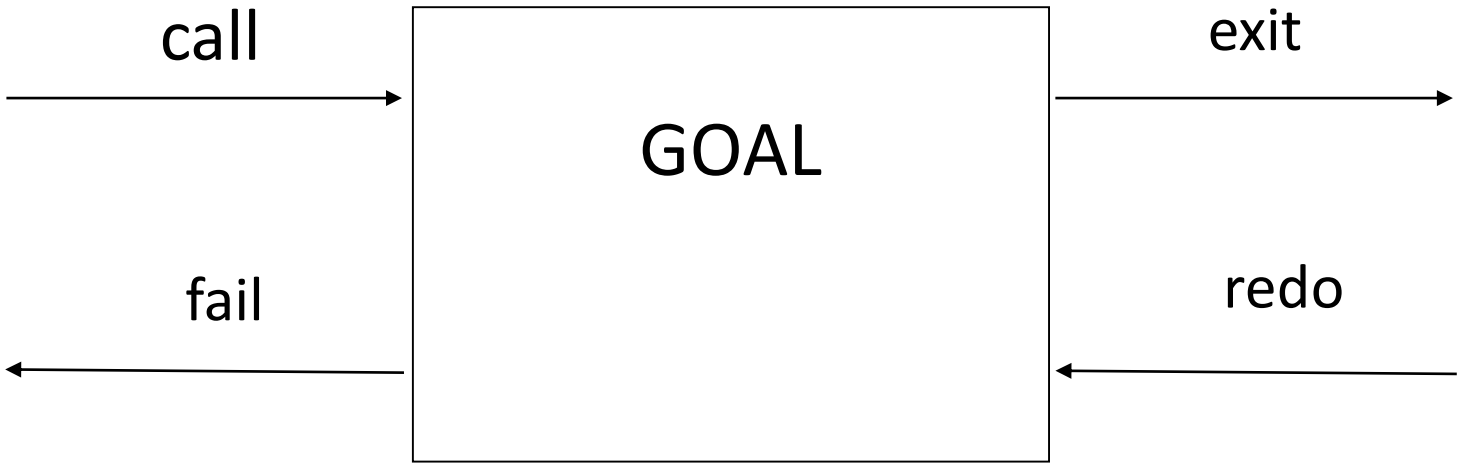
New clauses are derived via **backward reasoning**, using an input clause as goal-reduction procedure.

The unifying substitution steps:

- **Input from the subgoal to the body** of the procedure,
- **Output from the head to remaining subgoals**.

Goal representation

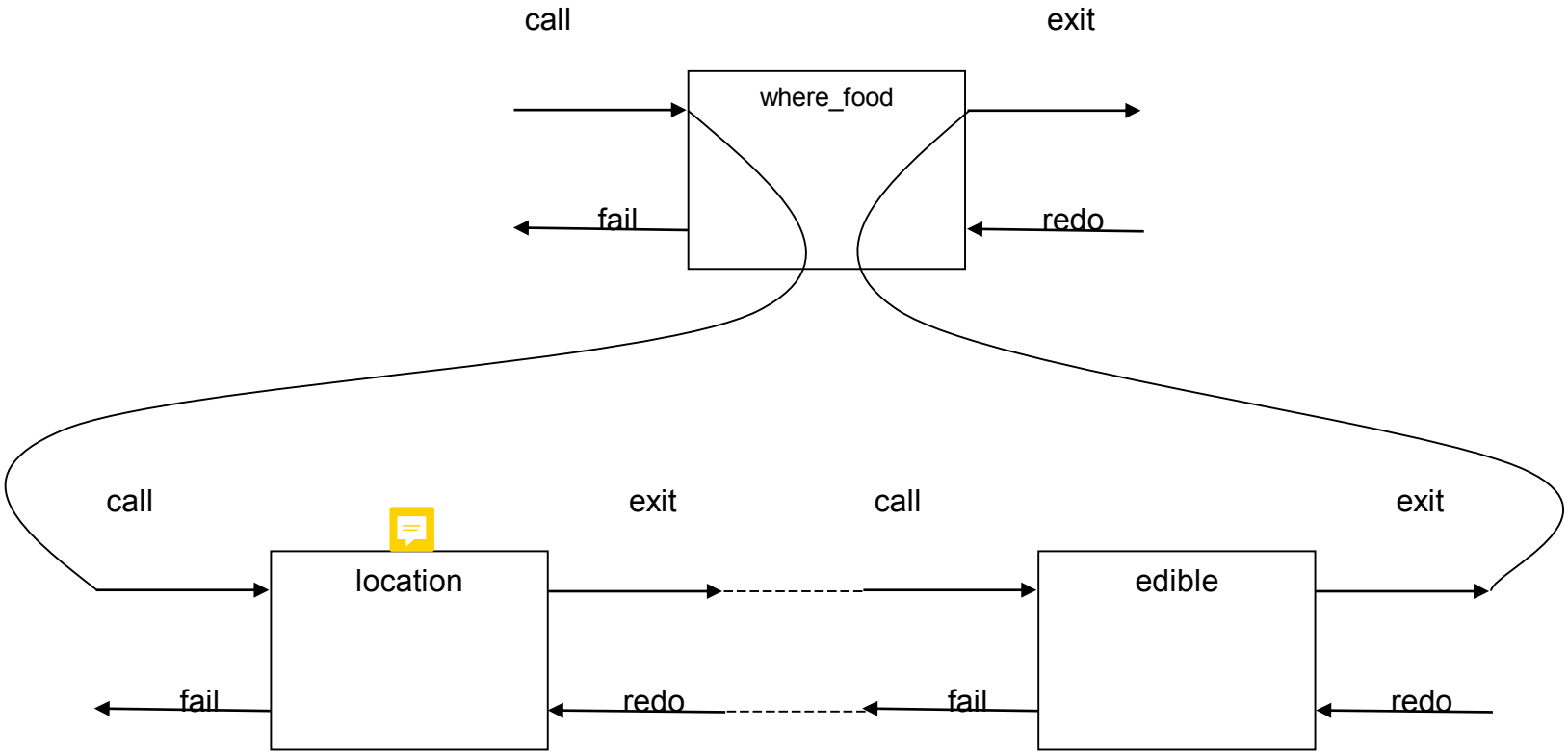
The query can be observed as an input-output black box:



A rule call

What Prolog does when a rule is called:

where_food(X,Y) :- location(X,Y),edible(X).



Trace

The trace command can be useful in order to analyze Prolog steps within the query execution.

8 ?- trace.

[trace] 8 ?- where_food(X,Y).

Call: (7) where_food(_G457, _G458) ? creep

Call: (8) location(_G457, _G458) ? creep

Exit: (8) location(desk, office) ? creep

Call: (8) edible(desk) ? Creep

…

X and **Y** are instantiated with internal anon variables **_G457** and **_G458.**

The query execution

When a query is submitted to the Prolog interpreter, **it tries to prove that the query can be logically derived from the program**.
This consists in **finding a substitution** such that the query can be derived from the knowledge base.

parent(richard,ned).
parent(ned,sansa).
child(Y,X):-parent(X,Y).

?-child(X,Y).

The unification **X/richard Y/ned** let the query to be derived from the program.

Finding for a solution

When a Prolog program is executed, a goal and a rule are involved, the search of a solution can be represented through a tree in which the Prolog interpreter looks for the solution among all alternatives. This **AND-OR** tree is named **search tree**.

The Prolog interpreter uses a **depth first search** strategy with **backtracking**.

The search tree is built evaluating alternatives **one by one** and exploring the sub-goals **from left to right**, analyzing the following sub-goal only when the previous one is satisfied.

Backtracking

When a goal fails, the interpreter uses the **backtracking** technique and **explores the alternative branches**.
When the navigation goes back the **instantiations are annulled** following an inverted order**, from the right to the left**.

The interpreter looks for further matches to satisfy the goal, using different clauses because unification algorithm does not contemplate alternatives.

Example

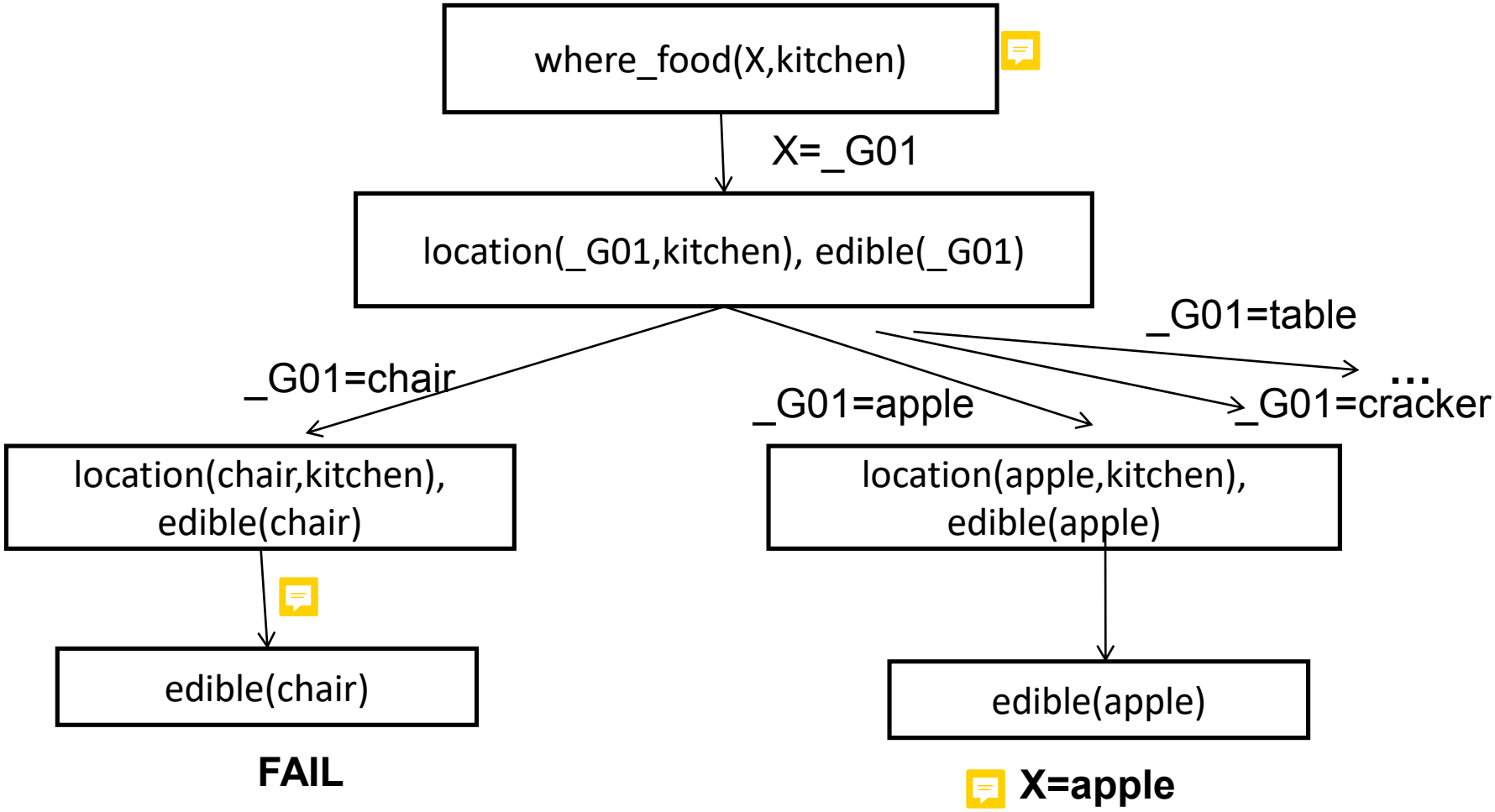
Knowledge base:

```
location(chair,kitchen).
location(apple,kitchen).
location(crackers,kitchen).
location(table,kitchen).
edible(apple).
edible(crackers).
where_food(X,Y)  :-  location(X,Y),edible(X).
```

Query:

```
?- where_food(X,kitchen).
```

Example


where_food(X,kitchen)
X=_G01
location(_G01,kitchen), edible(_G01)
_G01=table
_G01=chair
_G01=apple
...
_G01=cracker
location(chair,kitchen), edible(chair)
location(apple,kitchen), edible(apple)
edible(chair)
edible(apple)
FAIL
X=apple

Example

If the **cardinality** of the instances of the relation **is** previously **known** we could **evaluate** the **efficiency** of the program.
If the number of edible objects is lower than the total number of objects we could switch the subgoals from

where_food(X,Y) :- location(X,Y),edible(X).

to

where_food(X,Y) :- edible(X), location(X,Y).

Using a first fail heuristic to explore a potentially smaller tree.

What search tree is built in this case?

Example

Knowledge base:

f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X),g(X),h(X).

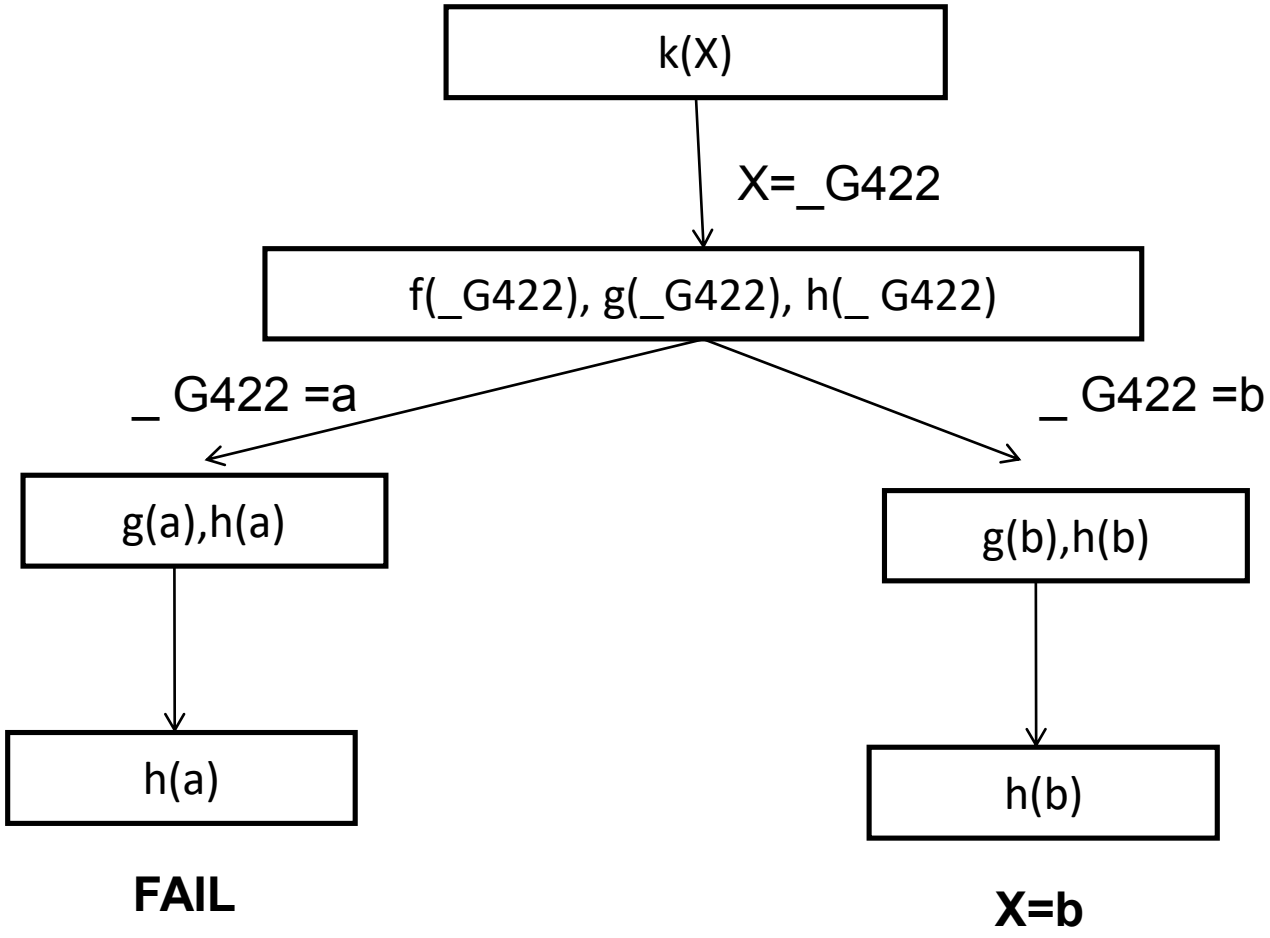
Query:

?- k(X).

Example

```
[trace] 12 ?- k(X).
   Call: (6) k(_G422) ? creep
   Call: (7) f(_G422) ? creep
   Exit: (7) f(a) ? creep
   Call: (7) g(a) ? creep
   Exit: (7) g(a) ? creep
   Call: (7) h(a) ? creep
   Fail: (7) h(a) ? creep
   Redo: (7) f(_G422) ? creep
   Exit: (7) f(b) ? creep
   Call: (7) g(b) ? creep
   Exit: (7) g(b) ? creep
   Call: (7) h(b) ? creep
   Exit: (7) h(b) ? creep
   Exit: (6) k(b) ? creep
X = b.
```

Example


k(X)
X=_G422
f(_G422), g(_G422), h(_ G422)
_ G422 =a
_ G422 =b
g(a),h(a)
g(b),h(b)
h(a)
h(b)
FAIL
X=b

Example

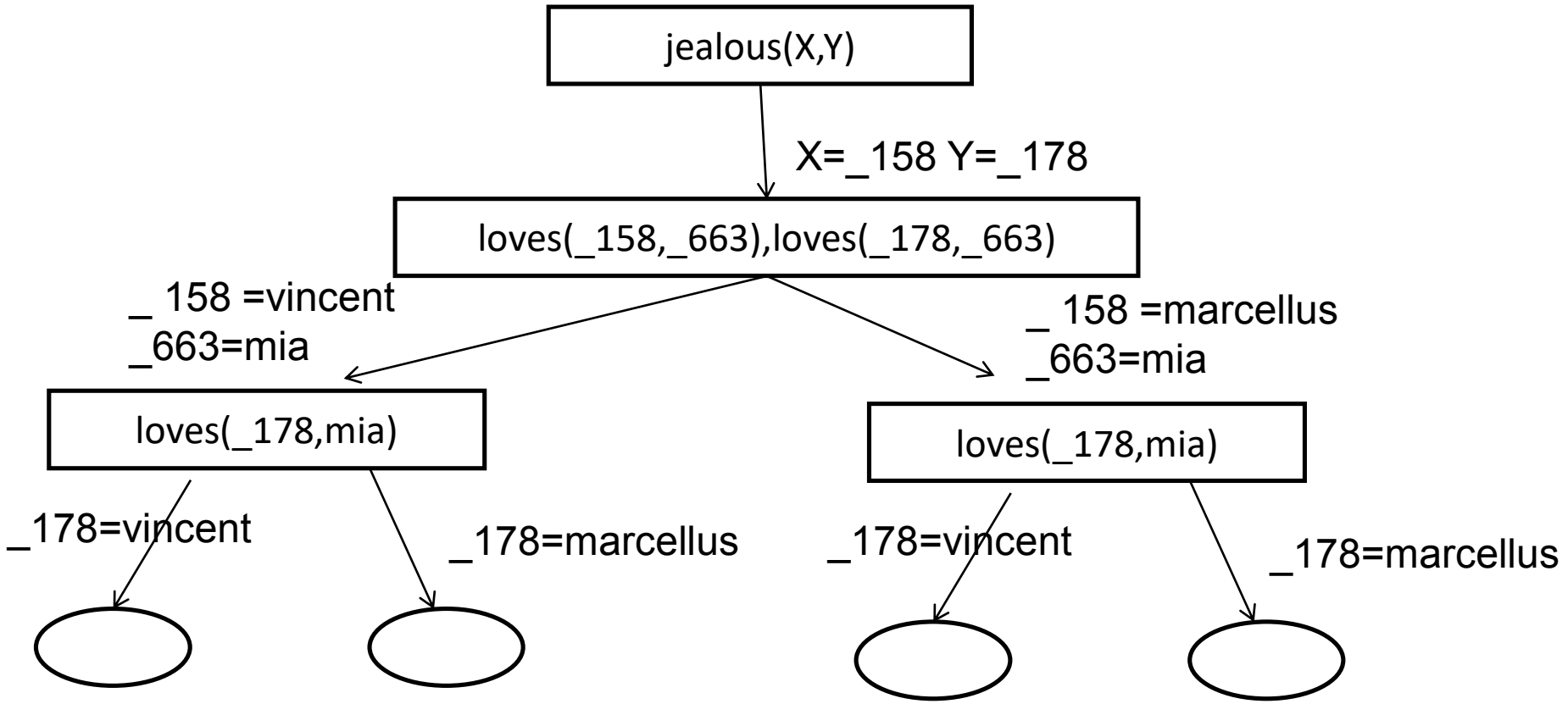
Knowledge base:

loves(vincent,mia).
loves(marcellus,mia).
jealous(X,Y) :-loves(X,Z),loves(Y,Z).

Query:

?- jealous(X,Y).

Example


jealous(X,Y)
X=_158 Y=_178
loves(_158,_663),loves(_178,_663)
_ 158 =vincent
_663=mia
_ 158 =marcellus
_663=mia
loves(_178,mia)
loves(_178,mia)
_178=vincent
_178=marcellus
_178=vincent
_178=marcellus

Recursion in Prolog

Module of Logics and Artificial Intelligence course

Recursion

Recursion is usually used when:

- Follow **relationships chains** among objects
- Efficiently manipulate **lists**

Let us take a look at the former case, a classical example of this kind is the **transitive use of a relation**:

- If Arya is a sibling of Sansa, and Sansa is a sibling of Bran, hence Arya is a sibling of Bran.
- If point A is connected to point B and point B is connected to point C hence point A is connected to point C.
- If the key is in the drawer and the drawer is in the bedside hence the key is in the bedside.

In order to prove these assertions we need to use recursion to explore the relations, keeping in mind we want to represent the transitivity of the relations.

Example 1 – The office

In this knowledge base the position of some objects within an office is modelled:

location(desk,office).
location(computer,office).
location(flashlight,desk).
location(envelope,desk).
location(stamp,envelope).
location(key,envelope).

Our goal is to check if the **key** is in the office

Example 1 – The office

In this knowledge base the position of some objects within an office is modelled:

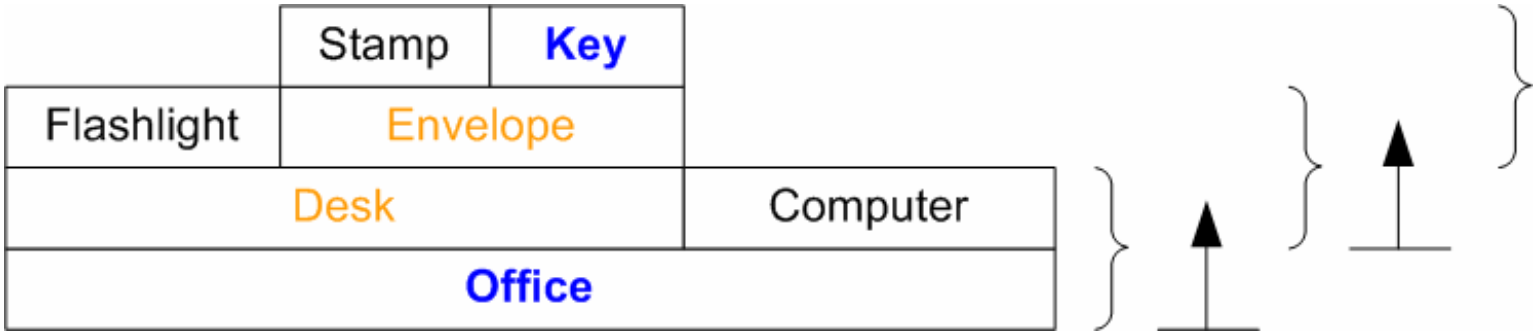
```
is_contained_in(T1,T2):-
        location(T1,T2).
is_contained_in(T1,T2):-
        location(X,T2),
        is_contained_in(T1,X).
```

boundary condition: it determines the stopping condition (IF) for the recursion. **It is true when the direct relation is satisfied**. In other cases (ELSE) Prolog will go over to the second definition of is_contained_in, the recursive one.

It finds an object **X** contained in **T2** and calls **is_contained_in** on **T1** and **X**, transitivelly looking for an object that directly contains **T1**.

Example 1 – The office

The behavior of the query **is_contained_in(key,office)** that will return **true** is shown below:

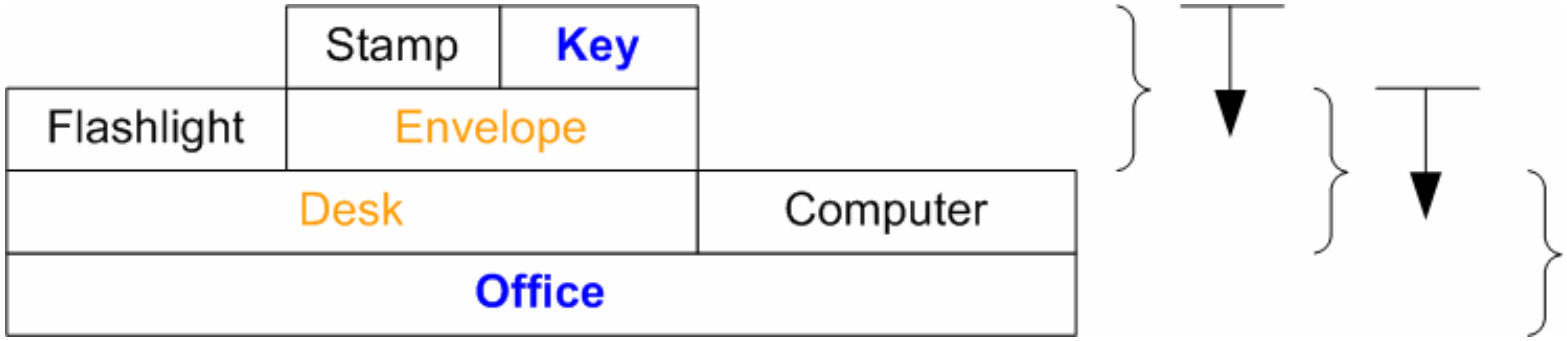


It finds the objects contained in the office and the query is repeated on these. The search moves, step by step, from **T2** towards **T1**. These rules are efficient for queries like **is_contained_in(X,office)**.

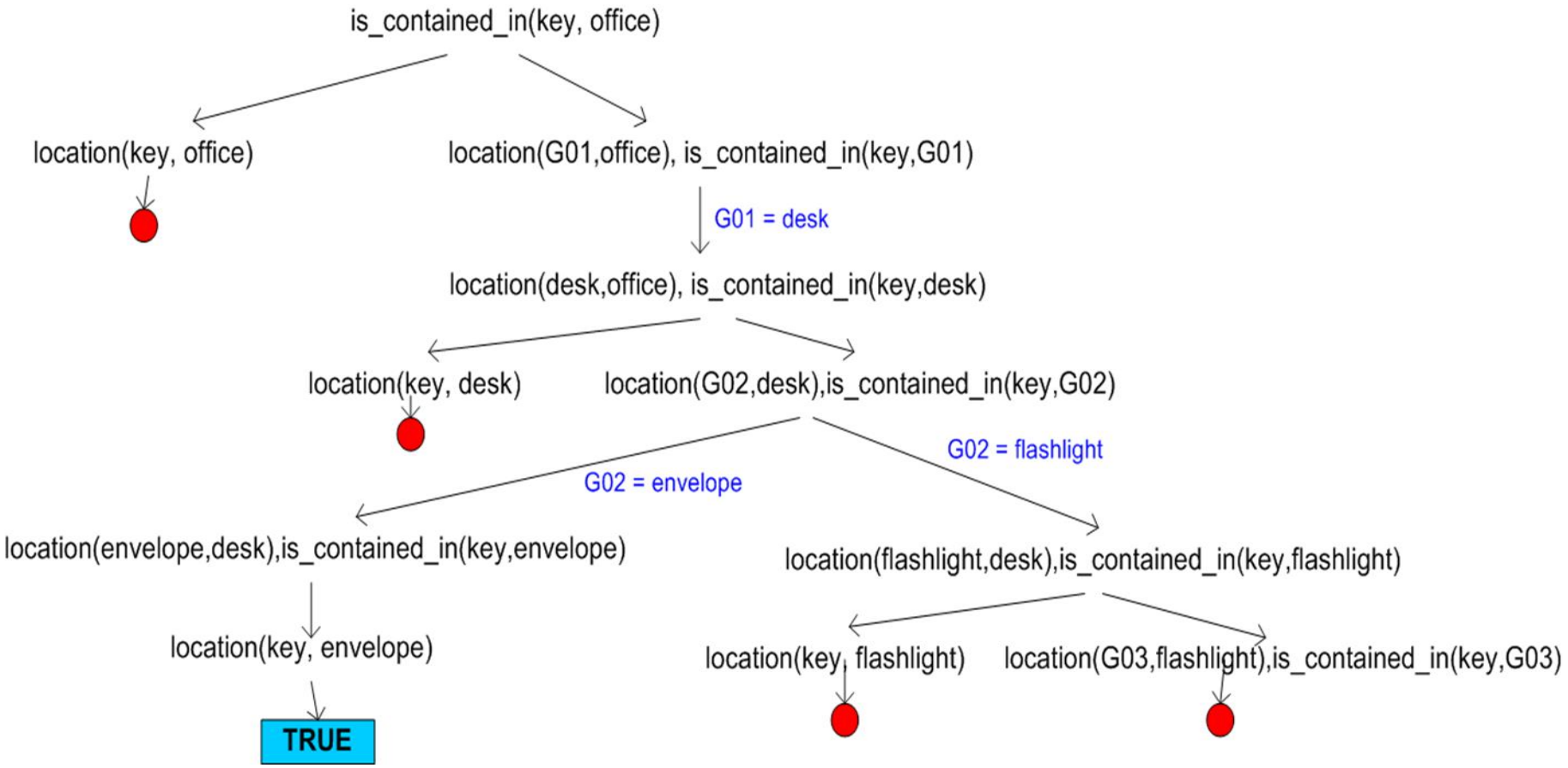
Example 1 – The office

We can write down rules that can efficiently face queries like **is_contained_in(key,X)**, designed to retrieve the objects that contains the **key (envelope, desk and office)**.

is_contained_in(T1,T2):- location(T1,T2).
is_contained_in(T1,T2):- location(T1,X), is_contained_in(X,T2).



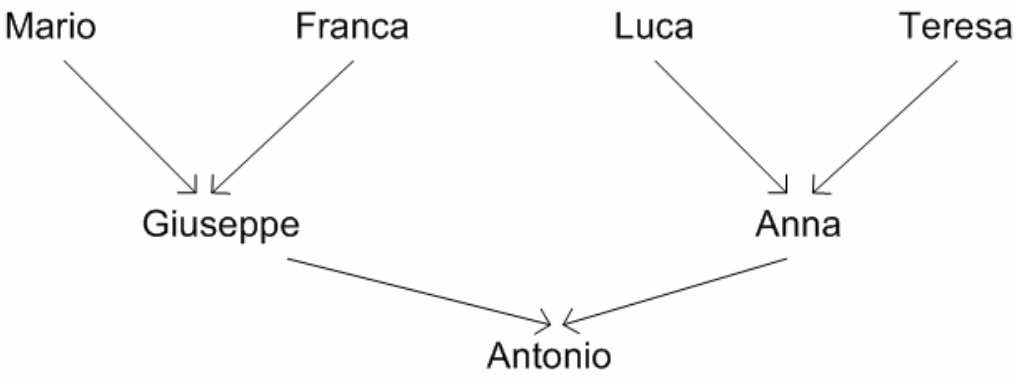
Example 1 – The office



Example 2 – The descendants

The knowledge base contains facts about the family tree shown in figure:

parent(giuseppe,antonio).
parent(anna,antonio).
parent(mario,giuseppe).
parent(franca,giuseppe).
parent(luca,anna).
parent(teresa,anna).



Create two recursive rules, the first one to check if **X** is a descendant of **Y** and the second one to check if **X** is an ancestor of **Y.**

descendant(X,Y).
ancestor(X,Y).

Example 2 – The descendants

```
descendant(X,Y) :-
        parent(Y,X).
descendant(X,Y) :-
        parent(Y,F),
        descendant(X,F).
```

OR

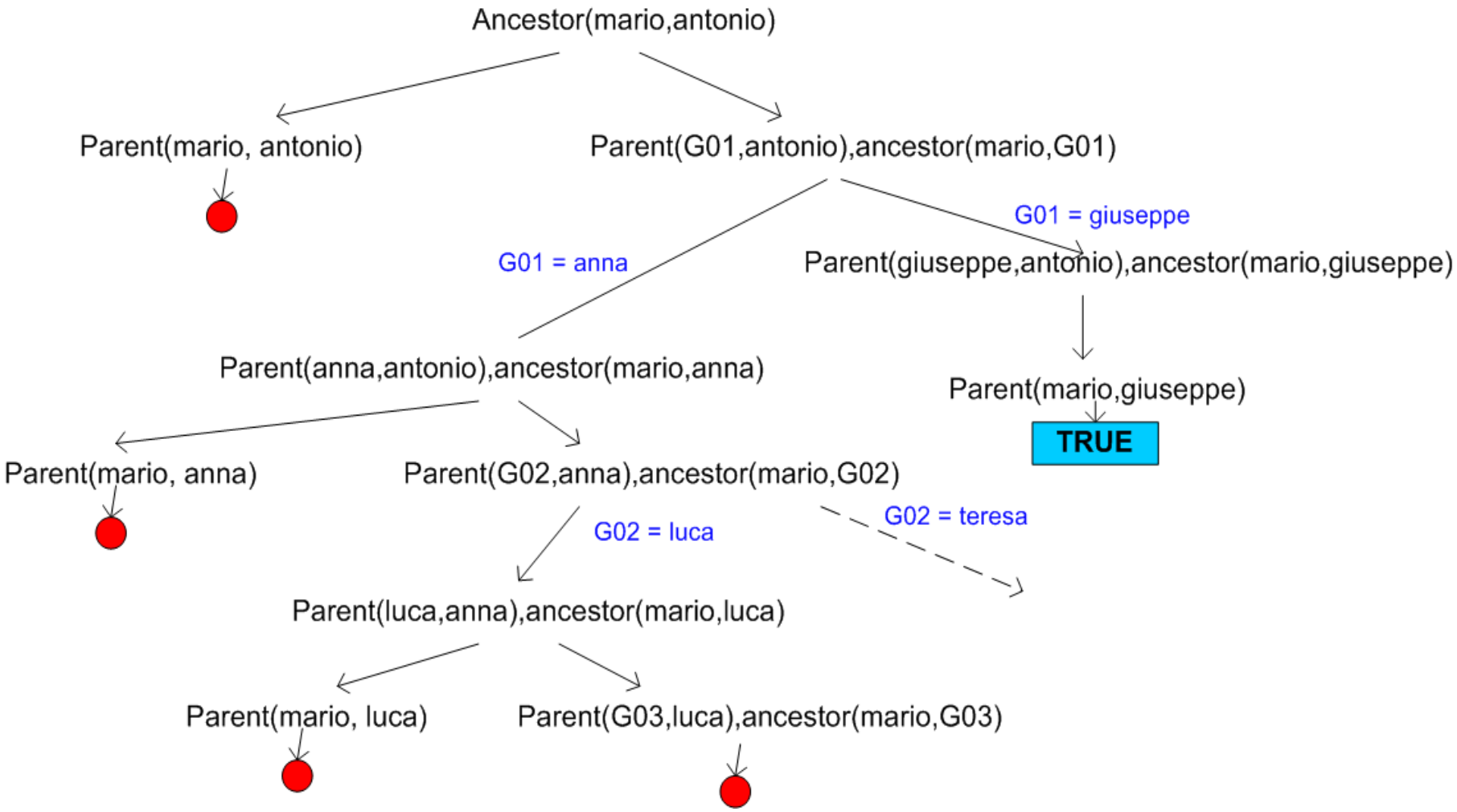
```
descendant(X,Y) :-
        parent(Y,X).
descendant(X,Y) :-
        parent(F,X),
        descendant(F,Y).
```

```
ancestor(X,Y) :-
        parent(X,Y).
ancestor(X,Y) :-
        parent(G,Y),
        ancestor(X,G).
```

OR

```
ancestor(X,Y) :-
        parent(X,Y).
ancestor(X,Y) :-
        parent(X,F),
        ancestor(F,Y).
```

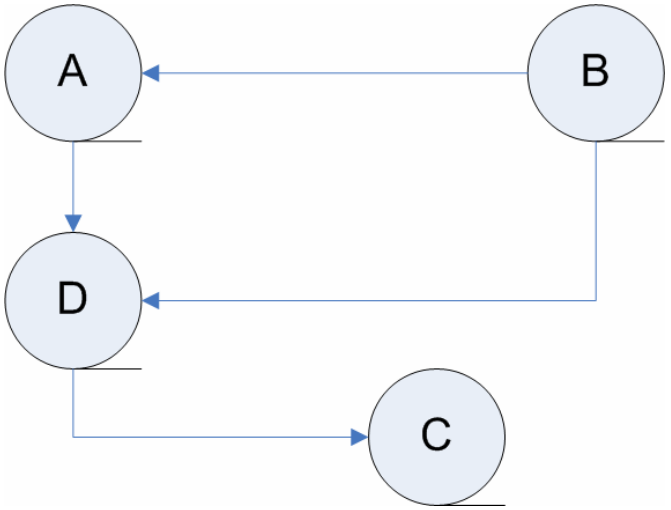
Example 2 – The descendants



Example 3 – The path

The knowledge base models the connections among four points (A,B,C,D):

arc(d,c).
arc(a,d).
arc(b,a).
arc(b,d).



Write a rule that checks if a path exists that connects via one or more arc a point to another. If one of the arguments is a variable, the program must return all the connected points:

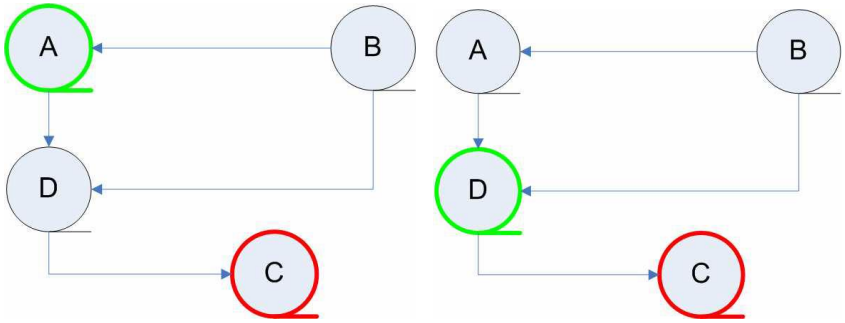
path(X,Y).

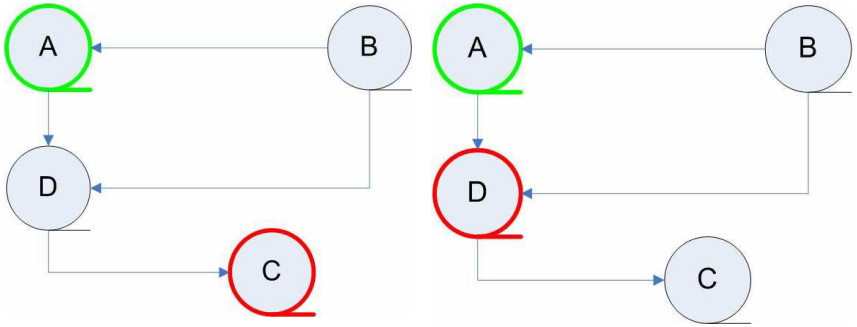
Example 3 – The path

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

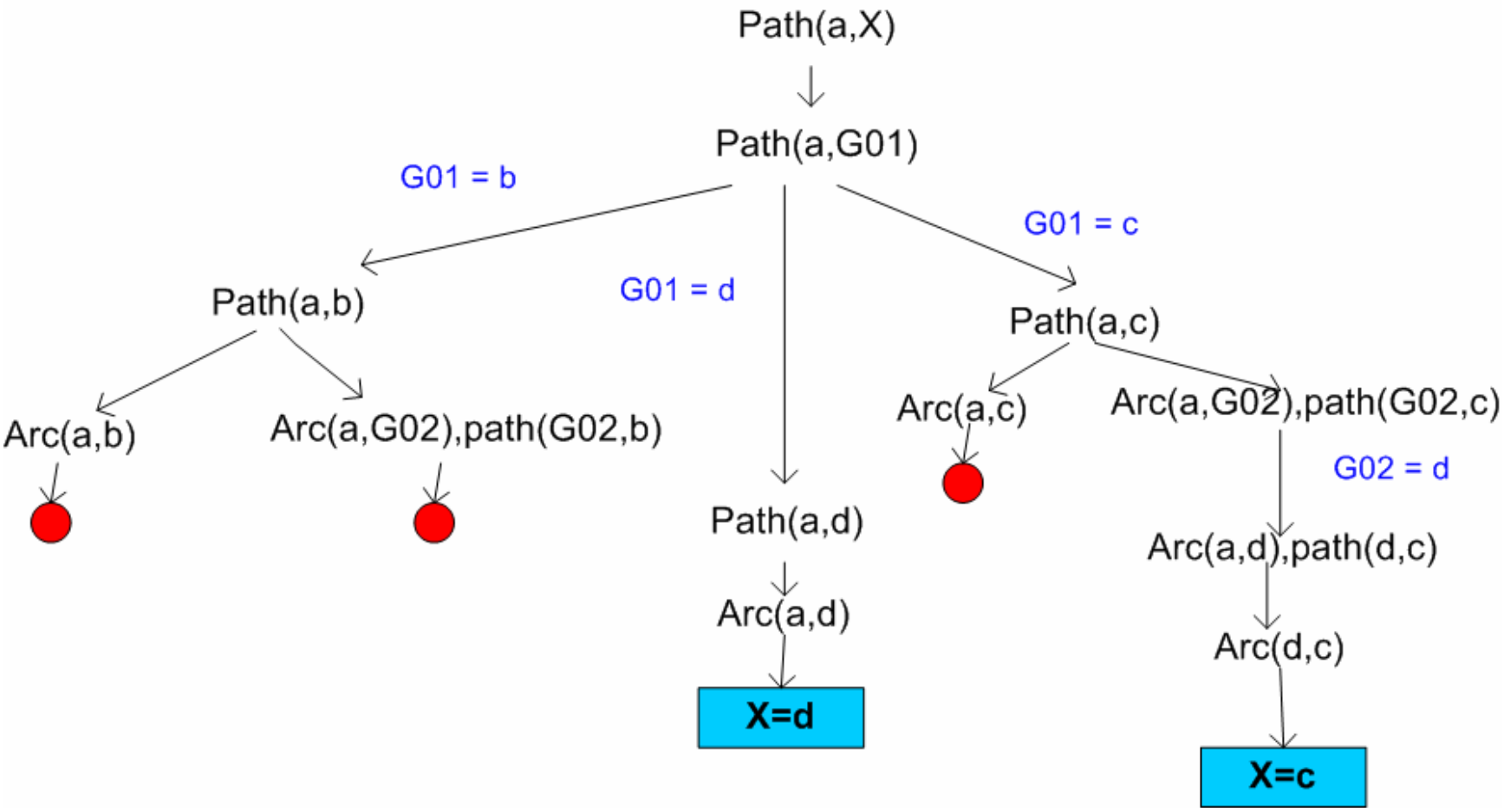
OR

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





Example 3 – The path



Example 4 – The factorial

Write a rule that computes the factorial function of a number:

Example 4 – The factorial

Write a rule that computes the factorial function of a number:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0 \end{cases}.$$

Example 4 – The factorial

Write a rule that computes the factorial function of a number:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0 \end{cases}.$$

```
fact(0,1).
fact(N,Ans):-
        N1 is N - 1,
        fact(N1,A1),
        Ans is N * A1.
```

The boundary condition set the factorial of **0 =1**

It computes before the factorial of **N – 1**, then it is multiplied by **N**. In this approach we go to 0 and coming back we compute the factorial of 1,2..N

Roughly we can say that we compute the factorial of 1 knowing the factorial of 0, hence we compute the factorial of 2 knowing the factorial of 1 and so on.

Example 4 – The factorial

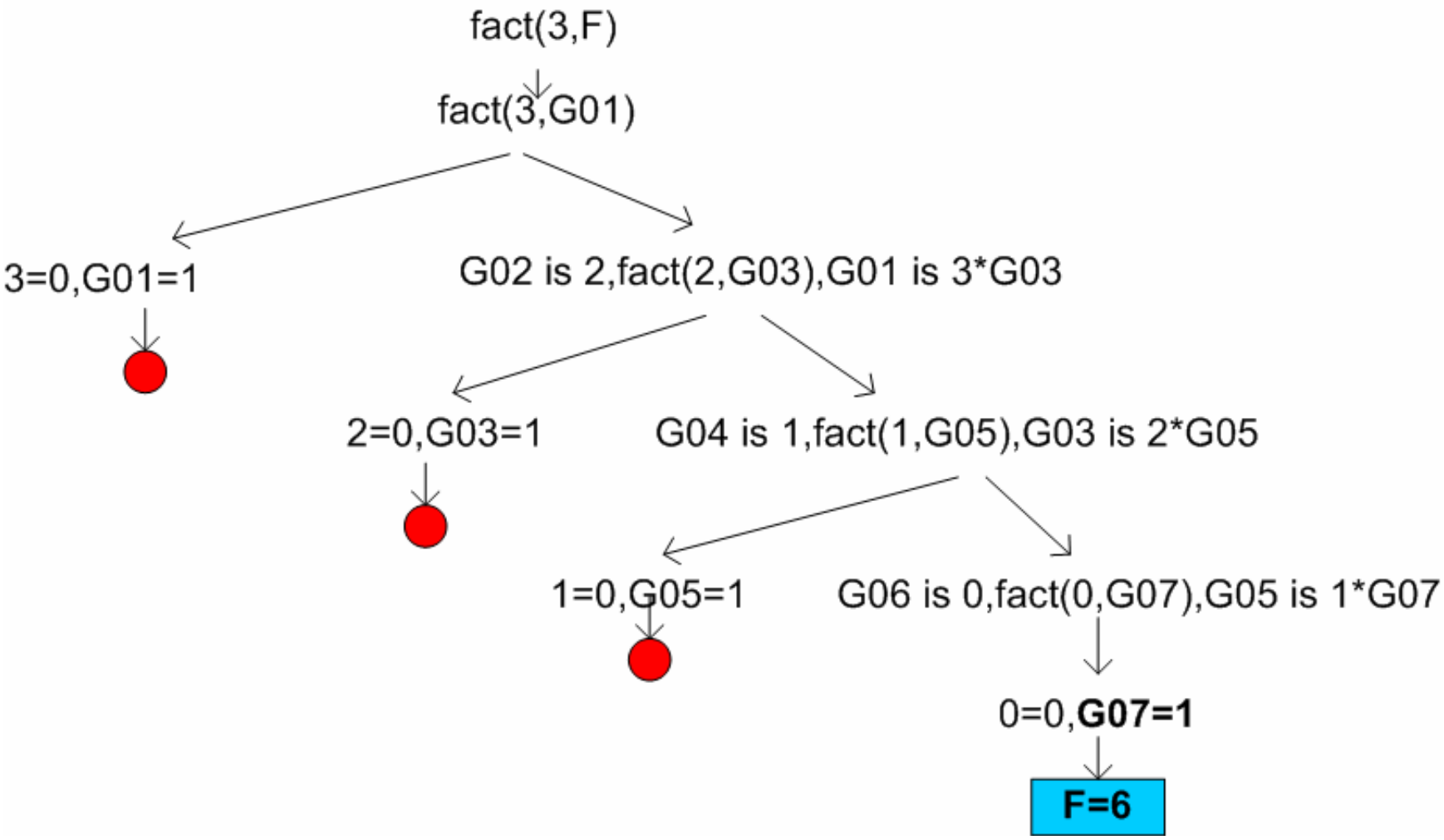
We could think to write down a (wrong) alternative to the previous code:

```
fact(0,1).
fact(N,F):-
        N1 is N - 1,
        F is F * N,
        fact(N1,F).
```

This is wrong! When the program tries to compute **F** that is not instantiated yet! So the Prolog Interpreter answer:

ERROR: is/2: Arguments are not sufficiently instantiated

Example 4 – The factorial



Example 5 – Even and Odd numbers

Write two rules, one to check if a number is even and another one to check if the number is odd:

Example 5 – Even and Odd numbers

Write two rules, one to check if a number is even and another one to check if the number is odd:

```
even(0).
even(N):-
        N1 is N-2,
        N1>=0,
        even (N1).
```

```
odd(1).
odd (N):-
        N1 is N-2,
        N1>=1,
        odd (N1).
```

Examples:

even(4) → 4-2=2 → 2-2=0 → *true*
odd(4) → 4-2=2 → 2-2=0 → 0-2=-2 → *false*

Example 6 – The power function

Write a Prolog program that computes $Z = X^Y$:

Example 6 – The power function

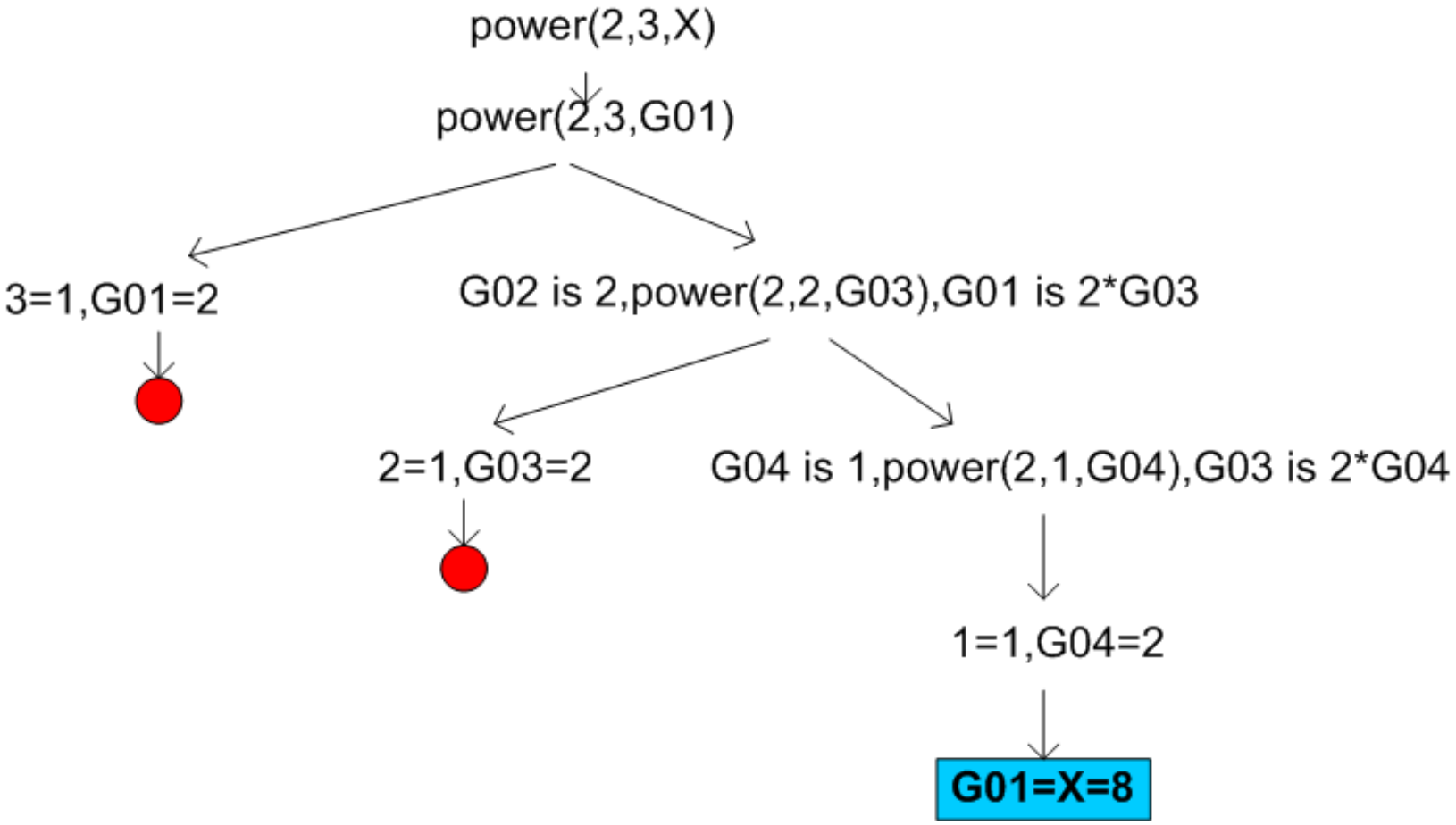
Write a Prolog program that computes $Z = X^Y$:

```
power(X,1,X).
power(X,Y,Z):-
        I is Y-1,
        power(X,I,Z1),
        Z is Z1*X.
```

The rule is applied over descending values until it gets to 1. In that moment the boundary condition set $Z = X^1 = Z$.

This is then multiplied by X, Y-1 times.

Example 6 – The power function



Lists in Prolog

Module of Logics and Artificial Intelligence course

Lists

Lists can be represented as:

Closed Lists: these lists are defined by a finite number of elements explicitly inserted in the list
Example: **[a,b,c]**

Open Lists: the number of the elements is not explicitly defined. They are represented in the form **[H|T]**, in which the **H** is the **HEAD**, the first element, and **T** is the **TAIL**, the list of the remaining elements (it can be empty).
N.B. the tail can be expressed as a list itself, hence the form **[H|[H2|T]]** is valid and useful in some situations.

Unification in Lists

Examples:

[X,c,b] = [a,Z,Y]
[X,c,b(c)] = [a,Y,Z]
[X,a,b] = [c,Y,d]
[X,a,b] = [c,Y,b]
[X|Y] = [1,2,3]
[X|Y] = [1]
[X,Y|Z] = [1,2]
[X|Y] = []
[a,b|X] = [1,2]

Unification in Lists

Examples:

[X,c,b] = [a,Z,Y]	X=a, c=Z, b=Y
[X,c,b(c)] = [a,Y,Z]	X=a, c=Y, b(c)=Z
[X,a,b] = [c,Y,d]	fail
[X,a,b] = [c,Y,b]	X=c, Y=a
[X|Y] = [1,2,3]	X=1, Y=[2,3]
[X|Y] = [1]	X=1, Y=[]
[X,Y|Z] = [1,2]	X=1, Y=2, Z=[]
[X|Y] = []	fail
[a,b|X] = [1,2]	fail

Example 1: Member

We want to write a rule that **checks if a stated element is a member of a list**. The element and the list are parameters passed to the function: **member2(Elem,List)**.

Let us focus on the needed steps:

a) We should **check if the first element of the list is equal to the searched element**. If the check is positive it returns true (this will be our boundary condition).
b) We **exclude the head of the list** and we **call the rule again**.

We will use member2 instead of member as name of the function because of the built-in predicate meber already exists.

Example 1: Member

```
member2(E,[H|_]):-
        E = H.
member2(X,[_|T]):-
        member2(X,T).
```

We are not interested in some variables hence we use anon variables (with "_" underscore) to denote them.

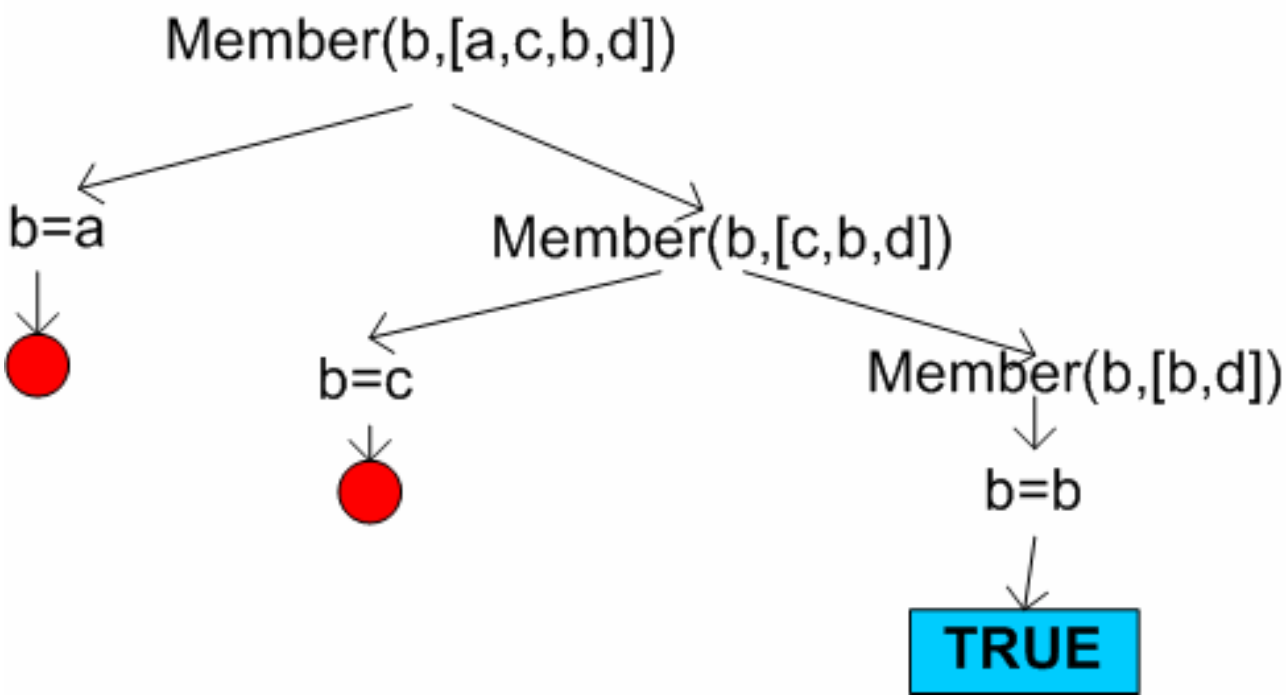
As expected after the check of the boundary condition the rule call recursively itself to check the remaining part of the list.
N.B. The variables names can be different in the rules.

Let us write down a more compact form of the rules using unification:

```
member2(E,[E|_]).
member2(X,[_|T]):-
        member2(X,T).
```

The equality check between E and the head of the list is performed implicitly.

Example 1: Member



Example 2: Make a copy of a list

Write a rule to copy a list in another one.
The simplest way is directly use unification:

List1 = List2.

Or with a simple rule

copy(Lista1, Lista2) :-
Lista1 = Lista2.

OR

copy(X,X).

Now we can write a rule to copy a list in another one, element by element.

Example 2: Make a copy of a list

Write a rule to copy a list in another one.
The simplest way is directly use unification:

List1 = List2.

Or with a simple rule

```
copy(Lista1, Lista2) :-
        Lista1 = Lista2.
```

OR

```
copy(X,X).
```

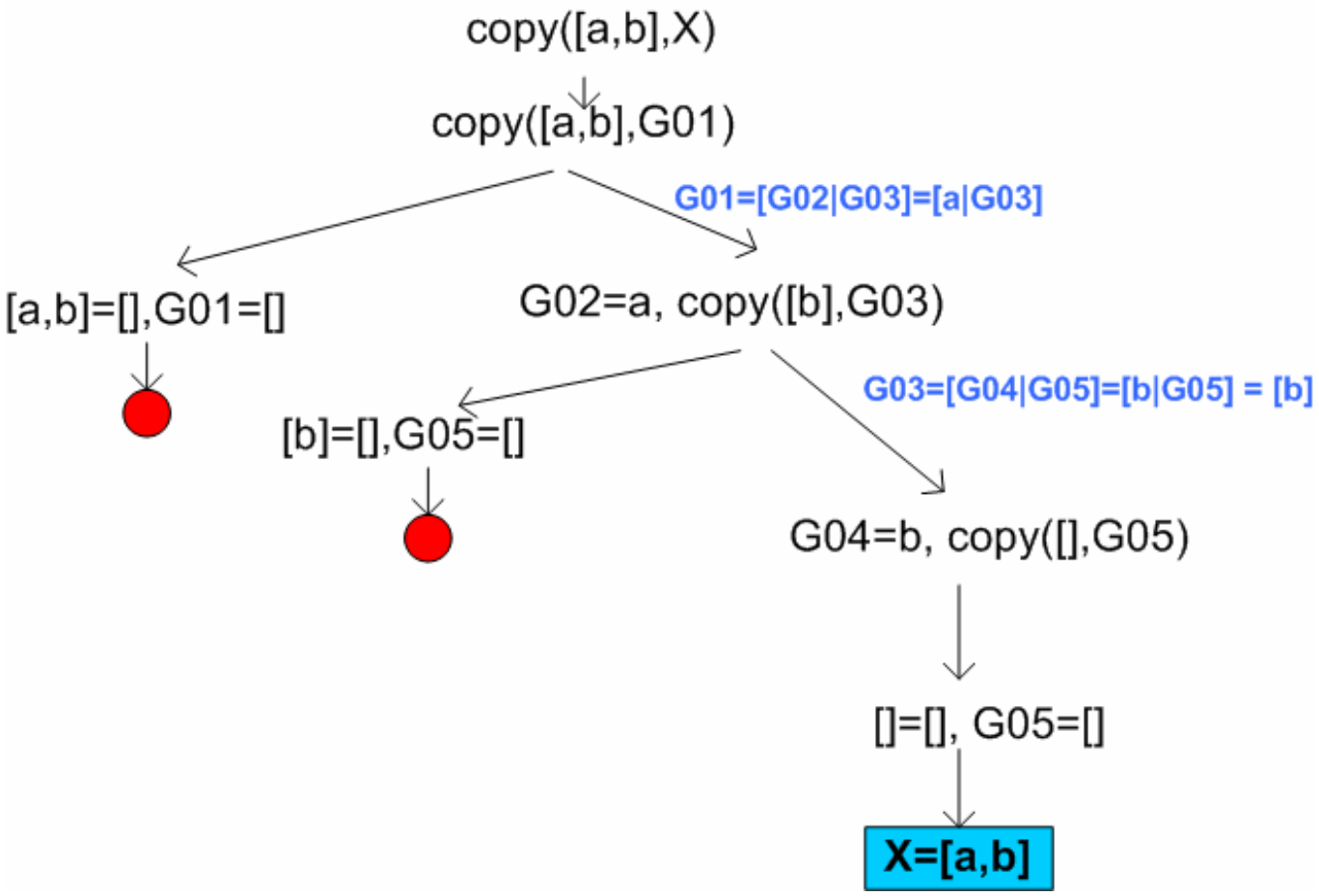
Now we can write a rule to copy a list in another one, element by element.

```
copy([],[]).
copy([H1|T1],[H2|T2]):-
        H1 = H2,
        copy(T1,T2).
```

OR

```
copy([],[]).
copy([H|T1],[H|T2]):-
        copy(T1,T2).
```

Example 2: Make a copy of a list



Example 3: Append

Write a rule to concatenate two lists. The rule must create a new list by appending the second list to the first one: **append2(list1,list2,X).**

a) Copy the elements from the first list to destination, until the first list gets empty.

b) Append the second list to destination

```
append2([],L2,Ldest):-
        Ldest = L2.
append2([H1|T1],X,[H2|T2]):-
        H2 = H1,
        append2(T1,X,T2).
```

Boundary condition: **list1** empty
We want to copy **L2** into **Ldest**

Element by element it shifts **list1** and copies the head of **list1** into the head of destination list. This step is repeated for each element in the list

N.B. At each step **append2** works on list tails, that are lists itselves. **Ldest** is the tail of the list in the second rule, hence unificating **L2** and **Ldest** means copying **L2** to **Ldest**.

Example 3: Append

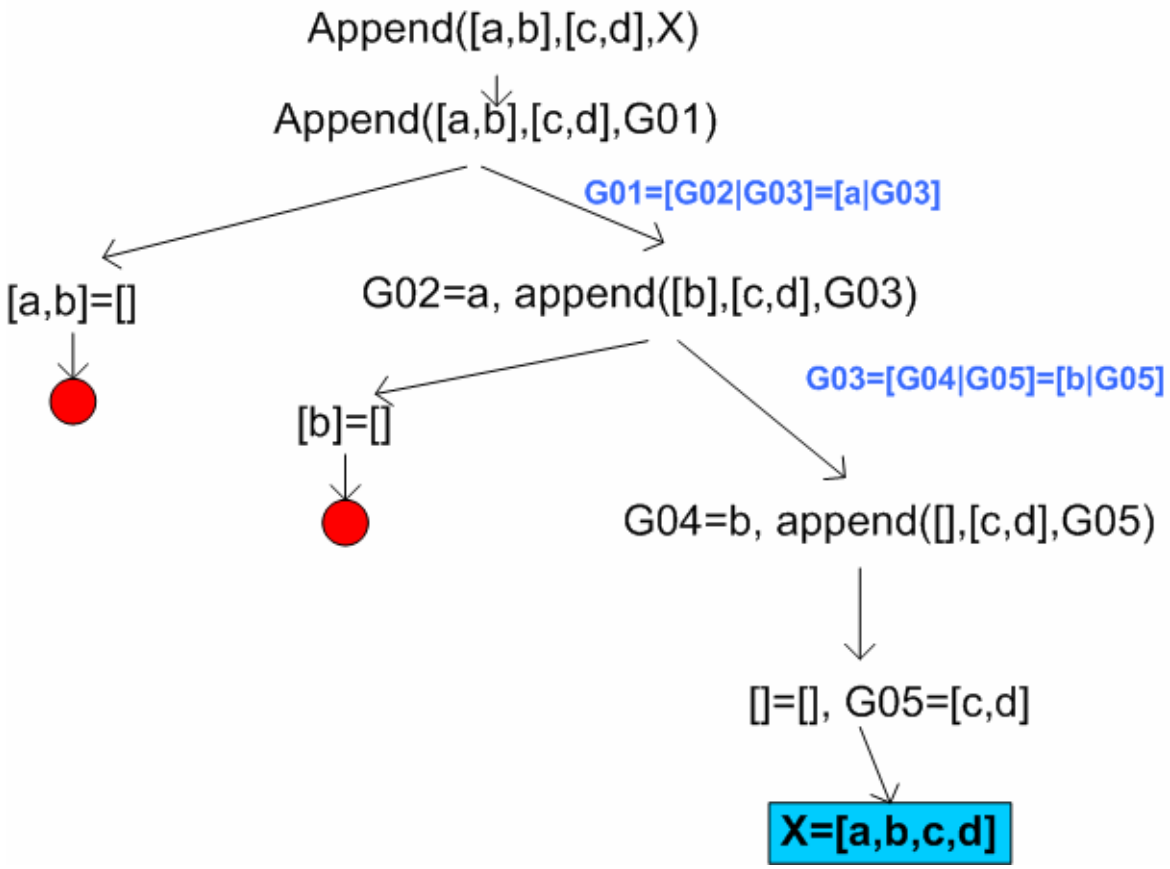
Let us write the append rule in a more compact form:

```
append2([],X,X).
append2([H|T1],X,[H|T2]):-
        append2(T1,X,T2).
```

The copy of the second list and the copy of the first element is performed implicitly

And for splitting a list?

Example 3: Append



X = G01 = [G02 | G03] =
= [a, G04 | G05] =
= [a, b | G05] =
= [a, b, c, d]

Example 4: Break out

Let us write a rule that breaks a list into its building elements and add the elements to the knowledge base through the stuff relation:

```
break_out([]).
break_out([Head | Tail]):-
        assertz(stuff(Head)),
        break_out(Tail).
```

It stops when the list is empty but Be careful, **the list is not actually empty**, only the tail passed as parameter will be empty

The built-in predicate **assertz** will add to the KB a **stuff()** relation instantion for each element contained in the list. If the query **break_out([a,b,c]).** is run, the KB will contains:

```
stuff(a).
stuff(b).
stuff(c).
```

Example 5: Add an element to a list

We want to write a rule to add an element to a list.
The simplest way to do this is:

```
puta(List,E,NewList):-
        NewList = [E|List].
```

This rule inserts the new element as the head of the list.

What if we want to add the element as the last element of the list?
We should write a recursive rule that will shift until the end of the list and then will add **E**.

Let us write the rule.

Example 5: Add an element to a list

```
putz([],E,[E]).
putz([H|T1],E,[H|T2]):-
        putz(T1,E,T2).
```

As usual the rule copy the list element by element. When the end of the list is reached, the boundary condition add the element **E** to the list.

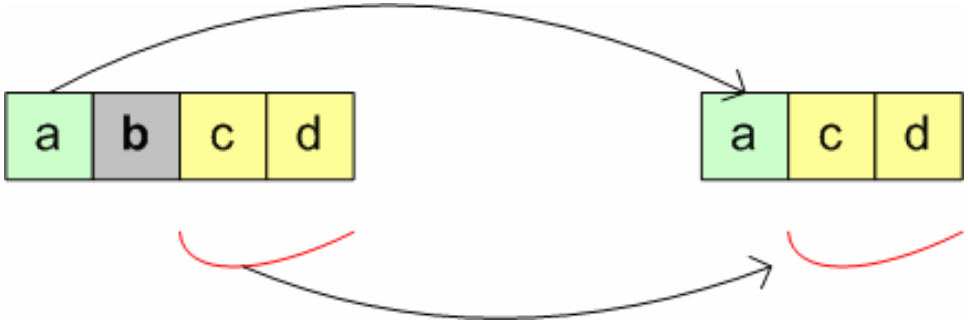
N.B. in the boundary condition the third parameters is the **tail** of the new list, **that is a list itself**, hence we want that an element (**E**) will be turned in a closed list (**[E]**) to let Prolog to do a right unification.

Example 6: Remove an element from a list

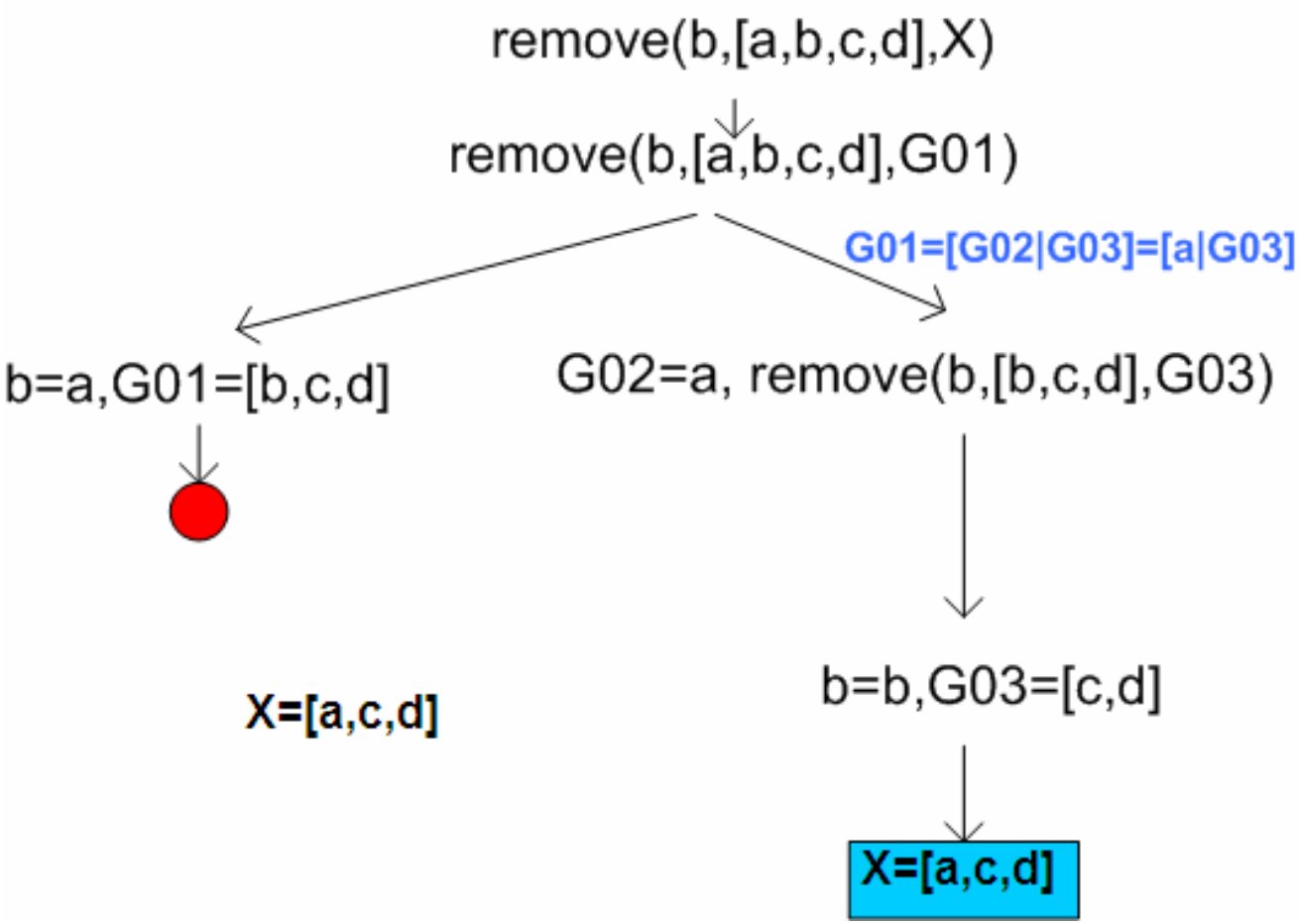
We want a rule that removes the first occurrence of a stated element from a list (More occurrences imply more solutions):
remove(Elem,SourceList, DestList).

```
remove(E,[E|T1],T1).
remove(E,[A|B],[A|D]):-
        remove(E,B,D).
```

The list is copied lement by element. When the Element is found, the tail is copied as tail of the destination list, resulting in a list without the first occurrence of the element **E**



Example 6: Remove an element from a list



Example 6: Remove an element from a list

We want a rule that removes ALL occurrences of a stated element from a list:

remove_all(Elem,SourceList, DestList).

```
remove_all(_,[],[]).
remove_all(E,[E|[H|T]],[H|T2]):-
        remove_all(E,T,T2).
remove_all(E,[A|B],[A|D]):-
        remove_all(E,B,D).
```

If the element E is found, the rule copies only the next element and the rule is repeated until the list gets empty

Example 7: Successor

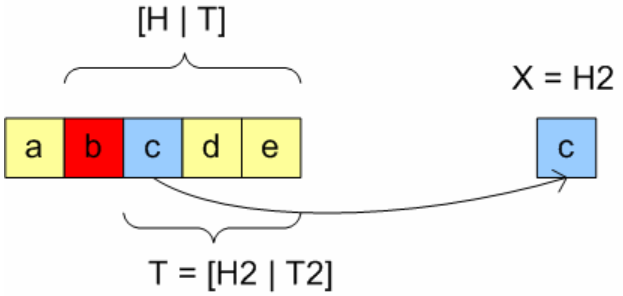
We want a rule that finds an element within a list and returns the next element.

```
succ(E, [E|[T|B]], T).
succ(E, [H|T], S):-
succ(E, T, S).
```

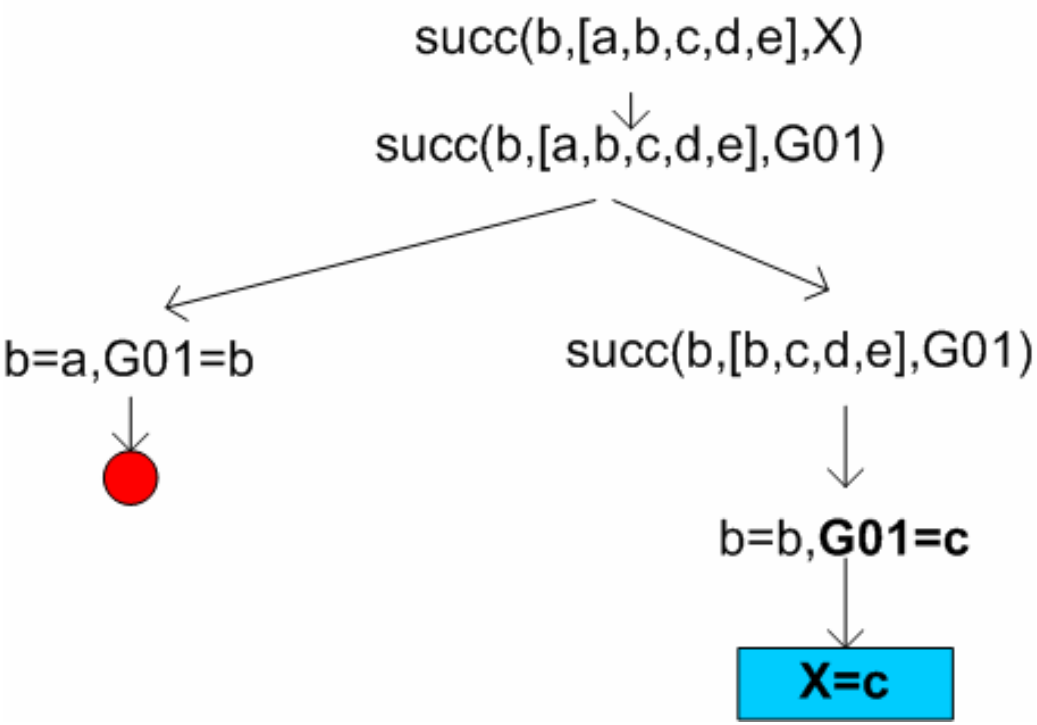
OR

```
succ(E, [H1|[H2|T]], X):-
        E=H1,
        X=H2.
succ(E, [H|T], S):-
        succ(E, T, S).
```

N.B. To select the first element of the tail is useful to represent the **tail as a list** composed by its head and tail.



Example 7: Successor



Example 8: Split a list

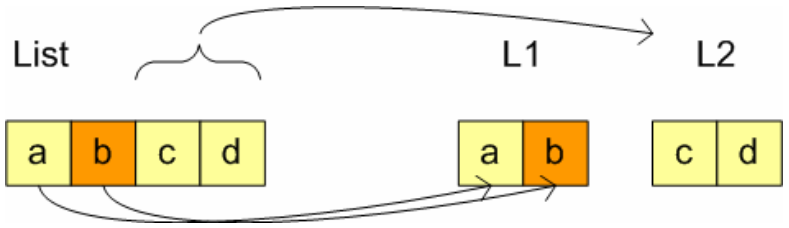
We want a rule that finds an element **E**, then it will create two lists: in the first one the last element will be **E**, in the second one the first element will be the following element in the source list:
divide(List,Element,List1,List2).

```
divide([E|T],E,[E],T).
divide([Hs|Ts],E,[Hs|Td],L2):-
        divide(Ts,E,Td,L2).
```

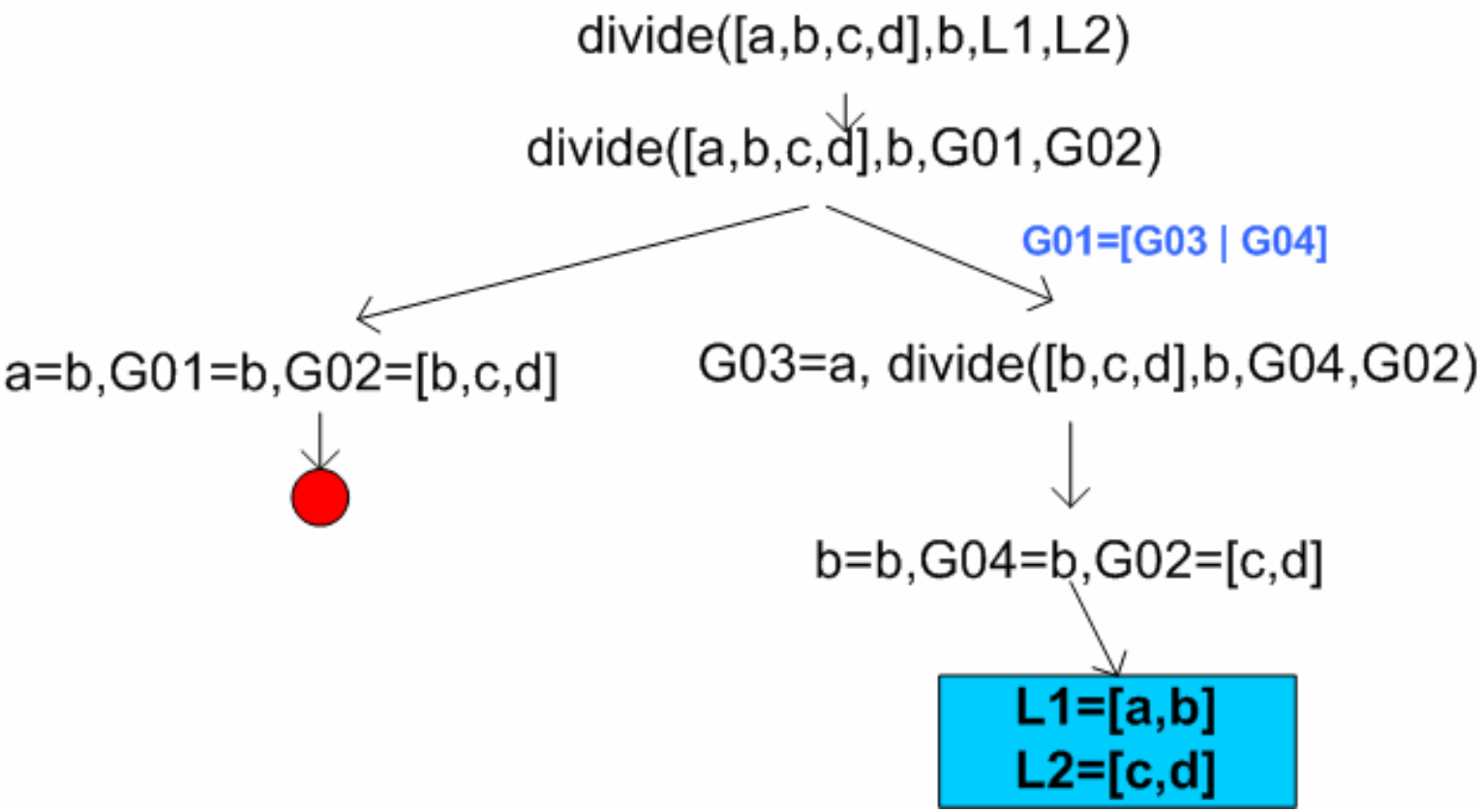
OR

```
divide([H,T],E,T1,T2):-
        H=E,
        T1=[E],
        T2=T.
divide([Hs|Ts],E,[Hd|Td],L2):-
        Hd=Hs,
        divide(Ts,E,Td,L2).
```

It copies the sorce list element by Element, until **E** is found. Then it unifies **E** as **List1 tail** and the **tail** as **List2**



Example 8: Split a list

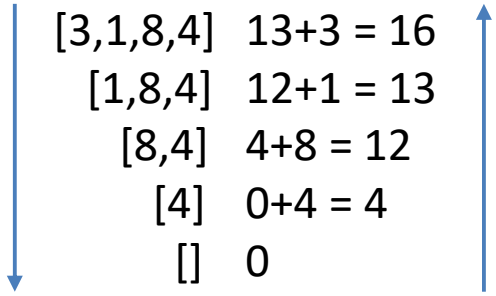


Example 9: Sum and count the elements

We want a rule that sums up all the elements of a list:
sum(List, S).

```
sum([],0).
sum([H|T],Sum):-
        sum(T,S),
        Sum is S+H.
```

As shown in the factorial function example we must avoid mistakes in initialization of the Sum variable. We instantiate it in the boundary condition and we sum the elements when we come back.



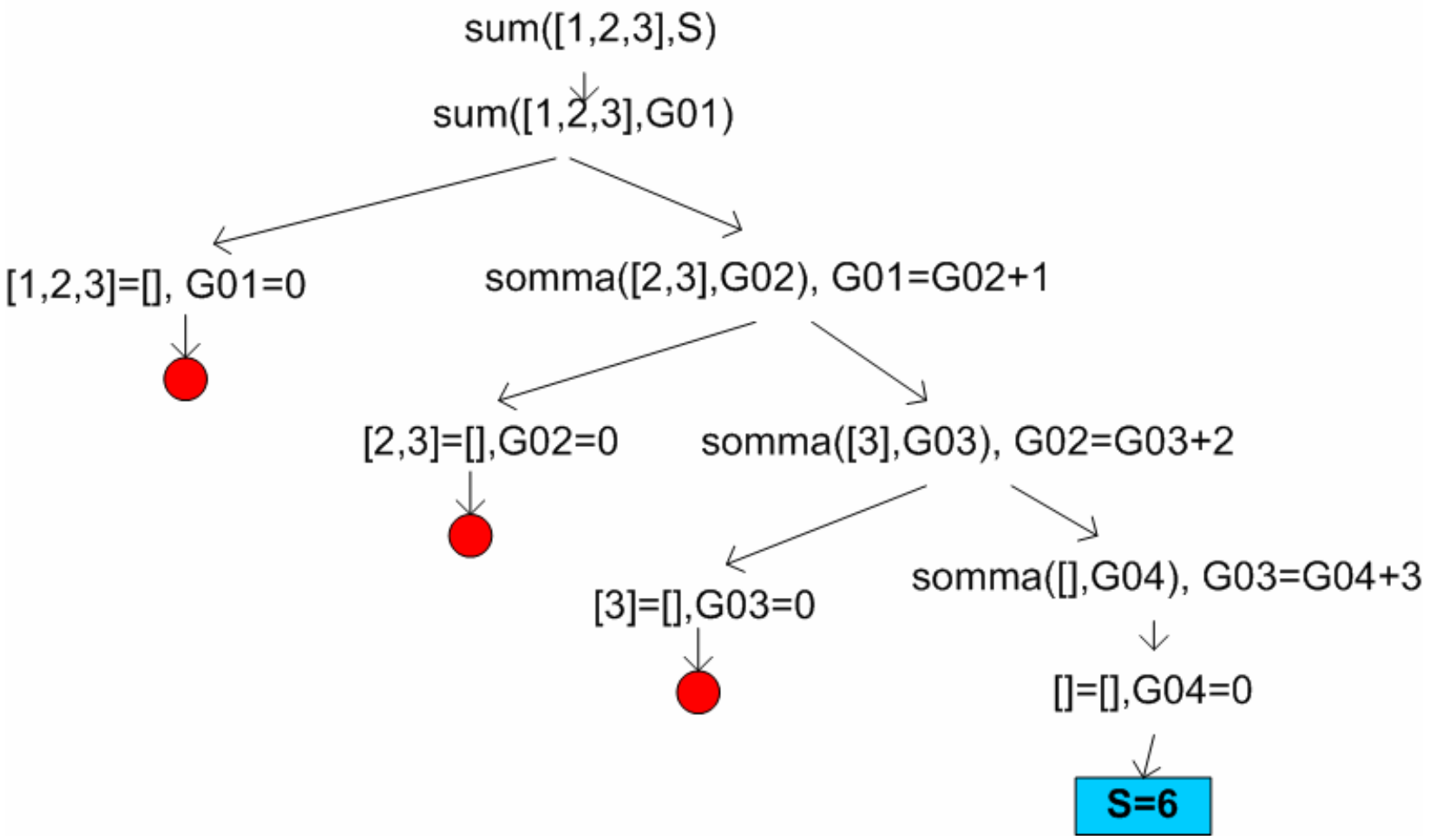
Example 9: Sum and count the elements

We want a rule that count the elements contained in a list:
count(List,C).

```
count([],0).
count([H|T],X):-
        count(T,X2),
        X is X2+1.
```

As in the former example we initialize the **counter=0** at the end of the list and then we increment by 1 for each element

Example 9: Sum and count the elements



Example 10: All equals

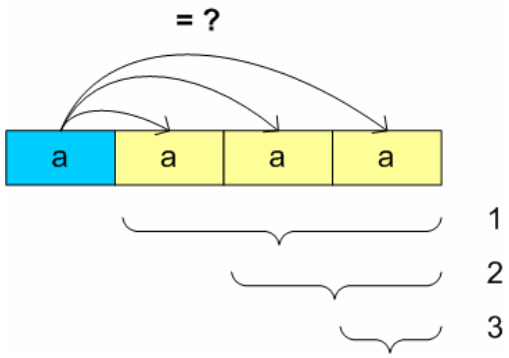
We want a rule that checks if all elements are equals.
We will compare all the elements in the list with the first one. If a check fails, the query fails.

```
allequals([H|T]):-
        equals(H,T).
equals(_,[]).
equals(E,[E|T]):-
        equals(E,T).
```

OR

```
allequals([H|T]):-
        equals(H,T).
equals(_,[]).
equals(E,[H|T]):-
        E=H,
        equals(E,T).
```

We use two rules. **allequals** selects the first element, equals compares it with all the elements in the tail.
When **E=T** fails, **equals** fails and **allequals** returns **FAIL**.



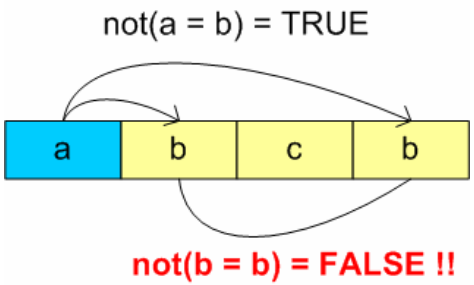
Example 11: All different

We want a rule that checks if all elements are different.
We can use the same former approach but we must repeat the check for all the elements of the list

```
alldifferent([]).
alldifferent([H|T]):-
        different(H,T),
        alldifferent(T).
different(_,[]).
different(E,[H|T]):-
        not(E = H),
        different(E,T).
```

The function different is dual with respect of equals. It is called on all the elements of the list. It works also if it stops at the second-last element. In this case the boundary condition would be **alldifferent([Last]).**

N.B. Modifying the former rule with **E=H→ not(E=H)** does not always work



Some useful rules

Let us write a couple of rules useful for more complex programs.

a) Write a Prolog rule that finds the last element of a list.

```
last([Last],Last).
last([_|T],L):-
        last(T,L).
```

OR

```
last([Last],L):-
        L = Last.
last([_|T],L):-
        last(T,L).
```

It shifts the list until the end, then the list contains only a closed list with one element, hence we set **L** equals to that element.

Es.: last([a,b,c],X).
X = c

Some useful rules

b) Check if the stated element is the smaller of all the elements contained in the list.

```
min(_,[]).
min(E,[H|T]):-
        E<H,
        min(E,T).
```

It shifts the list checking the relation. If the list gets empty, true will be returned. In other cases I would have found an element **<=E**

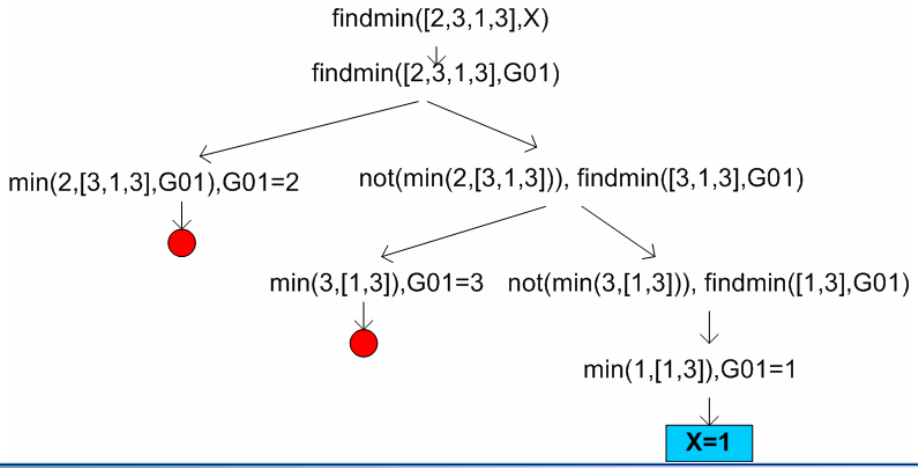
Example 12: Minimum of a list

Write a rule that finds the minimum of a list.

```
findmin([H|T],H):-
        min(H,T).
findmin([H|T],Min):-
        not(min(H,T)),
        findmin(T,Min).
```

If the head element is the minimum, it will be returned. Otherwise it shifts the list until an element that makes **min(H,T).** true is found.

Search tree without **min** branches:



Example 13: Sorting a list

Write a rule that sorts the elements in a list in ascending order.
sort(List,SortedList).

a) Find the minimum in the source list
b) Add the minimum as head of the destination list
c) Remove the minimum from the source list
d) Repeat from **a)** on **NL** and on the **tail** of the **sorted list L2**

```
sort([],[]).
sort(List,[Min|T2]):-
        findmin(List,Min),
        remove(Min,List,NL),
        sort(NL,T2).
```

The minimum is added as head of destination list with unification.
NL is the new source list without the minimum

Example 14: Reverse a list

Write a rule that reverse the order of the elements in a list.
mirror(L1, L2).

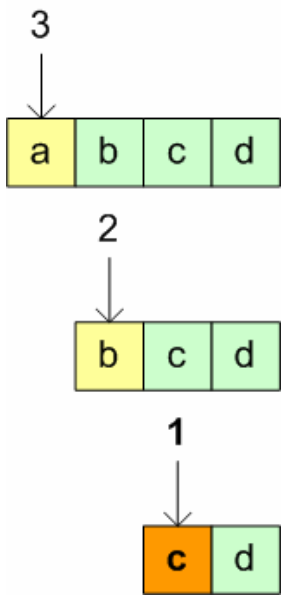
a) Find the last element in the source list
b) Add the minimum as head of the destination list
c) Remove the element from the source list
d) Repeat from **a)** on **NL** and on the **tail** of the **list L2**

```
mirror([],[]).
mirror(L1,[X|T2]):-
        last(L1,X),
        remove(X,L1,NL),
        mirror(NL,T2).
```

Example 15: Nth element of a list

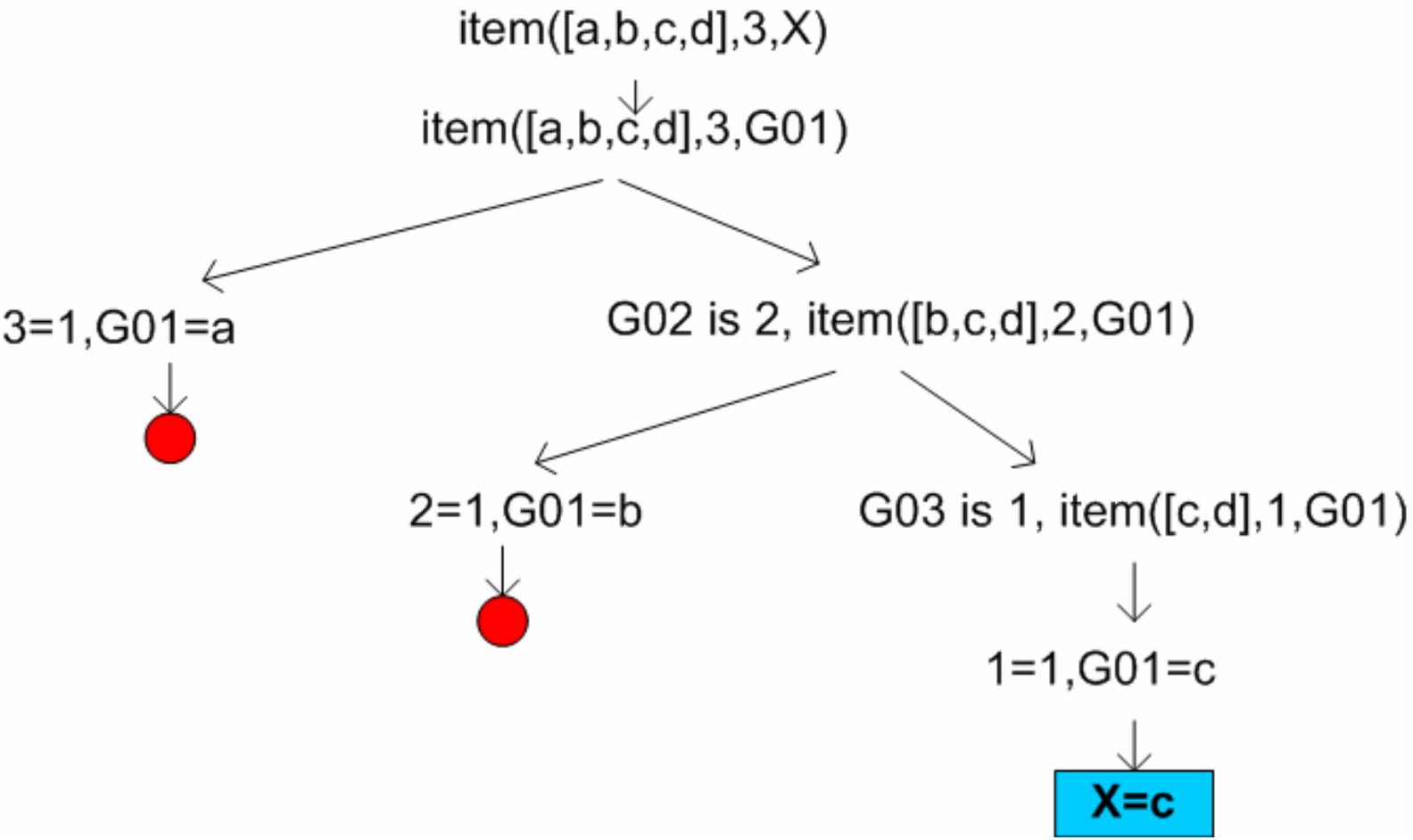
Write a rule that finds the element with index N in a list.

```
item([E|_],1,E).
item([H|T],I,E):-
        Ind is I-1,
        item(T,Ind,E).
OR
item([H|_],1,E):-
        E = H.
item([H|T],I,E):-
        Ind is I-1,
        item(T,Ind,E).
```



It shifts the list and reduces the value of the index until it gets to 1. When it happens the nth element is found.
If the list is 0-indexed the changes to be made at the program are trivial.

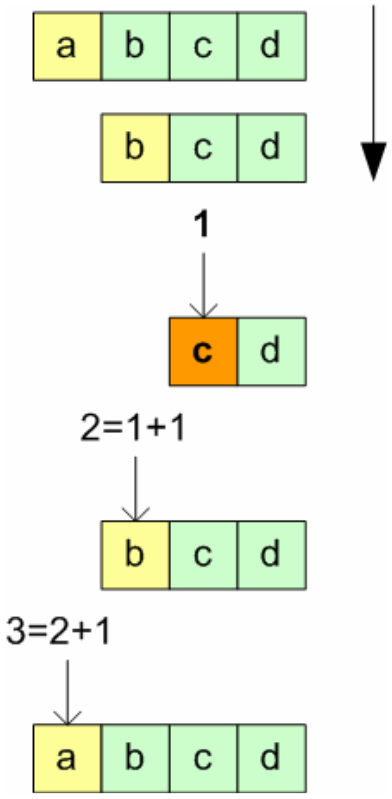
Example 15: Nth element of a list



Example 16: Index of element in a list

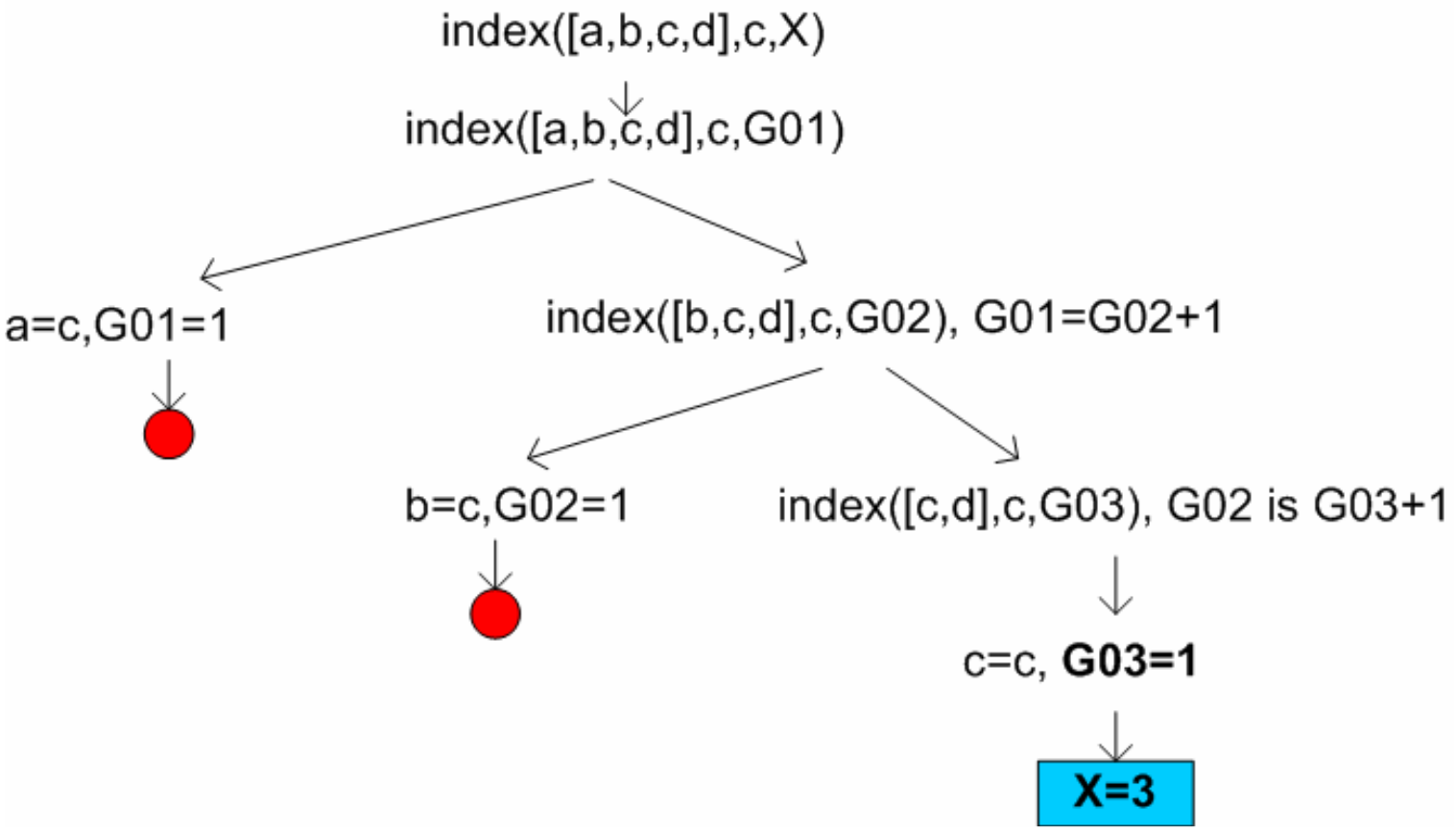
Write a rule that finds the index (or the indices) of an element in a list. The list is 1-indexed.

```
index([E|_],E,1).
index([H|T],E,N):-
        index(T,E,Ind),
        N is Ind+1.
```



The former rule (index → element) needed the index to be instantiated, hence it is not possible to use it for the dual purpouse. (element → index).
This one can be used in both cases (but is more efficient in the latter case). It shifts the list until the element is found (set Ind=1), hence the index increments coming back and returns the index of the element.

Example 16: Index of element in a list



Example 17: Merge sort

Let us write a rule that merge two lists, checking the head of the lists and selecting the lower between them. The rule will order correctly only if the input lists are ordered.

```
merge([],L2,L2).
merge(L1,[],L1).
merge([X|T1],[Y|T2],[X|T]):-
        X<Y, merge(T1,[Y|T2],T).
merge([X|T1],[Y|T2],[Y|T]):-
        X>Y, merge([X|T1],T2,T).
```

Examples:

```
?- merge([1,3],[2,4],X).
X = [1, 2, 3, 4]

?- merge([3,1],[2,4],X).
X = [2, 3, 1, 4]
```

Example 17: Merge sort

Now we can write down a rule that realizes the merge sort between two lists:

```
merge_sort(V1,V2,Dest):-
        sort(V1,V1ord),
        sort (V2,V2ord),
        merge(V1ord,V2ord,Dest).
```

Example 18: Merge sort 2

Another version of merge sort algorithm:

```
merge_sort2([],V2,V2ord):-
        sort(V2,V2ord).
merge_sort2(V1,[],V1ord):-
        sort (V1,V1ord).
merge_sort2(V1,V2,[M1|T]):-
        findmin(V1,M1),
        findmin(V2,M2),
        M1<M2,
        remove(M1,V1,New1),
        merge_sort2(New1,V2,T).
merge_sort2(V1,V2,[M2|T]):-
        findmin(V1,M1),
        findmin(V2,M2),
        M2<M1,
        remove(M2,V2,New2),
        merge_sort2(V1,New2,T).
```

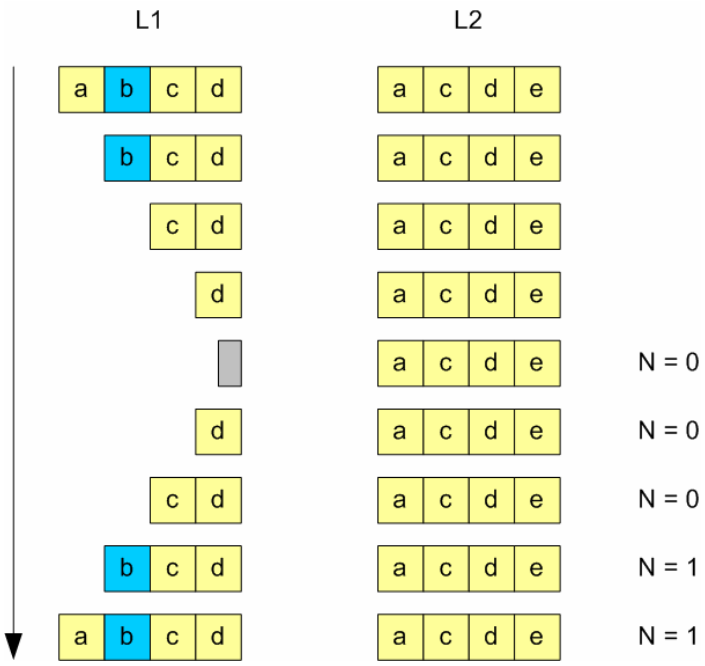
This version **finds minimum of both lists** and **it finds the lower between them**. That **element** is **removed** from the list and **added as head** of the destination list.

There are two boundary conditions depending on which list gets empty before.
When this condition is satisfied, it means that the elements of the other list are lower than the last element, hence we only need to sort the list and copy it into the destination list

Example 19: Count different elements

Write a rule that, two lists given, counts the number of the elements of the former list that are not in the latter.

```
notpresent([],_,0).
notpresent([H1|T1],L2,N):-
        notpresent(T1,L2,Num),
        different(H1,L2),
        N is Num+1.
notpresent([H1|T1],L2,N):-
        notpresent(T1,L2,N),
        not(different(H1,L2)).
```



We get the empty list and we instantiate N=0. For each list1 element different from all the elements from list2 the counter is increased by 1.

It is needed to define what to do whether **if H1 is not in L2 (increase and go on) that if H1 is in L2 (go on).**

Built-in Predicates in Prolog

Module of Logics and Artificial Intelligence course
a.a. 2016/2017

Arithmetic and Logical Operators

Classic notation	Prolog Notation
6+2=8	8 is 6+2.
6*2=12	12 is 6*2.
6-2=4	4 is 6-2.
6-8=-2	-2 is 6-8.
6 /2 =3	3 is 6/2.
7 mod 2 = 1	1 is mod(7,2).

Arithmetic and Logical Operators

?- 8 is 6+2.

?- 12 is 6*2.

? -2 is 6-8.

?- 3 is 6/2.

?- 1 is mod(7,2).

?- X is 6+2.

?- X is 6*2.

?- R is mod(7,2).

Arithmetic and Logical Operators

```
?- 8 is 6+2.
true
?- 12 is 6*2.
true
? -2 is 6-8.
true
?- 3 is 6/2.
true
?- 1 is mod(7,2).
true
?- X is 6+2.
X = 8
?- X is 6*2.
X = 12
?- R is mod(7,2).
R = 1
```

Examples

Sum 3 and double

sum3AndDouble(X,Y) :-

List length

Examples

Sum 3 and double

```
sum3AndDouble(X,Y) :- Y is (X+3)*2

?- somma_3_e_raddoppia(1,Y).
Y=8;
```

List length

```
len([],0).
len([_|T],N) :- len(T,X), N is X+1.

?- len([a,b,c,d,e,[a,b],g],X).
X = 7;
```

Arithmetic and Logical Operators

Notazione classica	Notazione Prolog
x<y	X<Y.
x<=y	X=<Y.
x>=y	X>=Y.
x=y	X=:=Y.
x!=y	X=\=Y.

```
4=:=4.
true
4=\=5.
true
4=\=4.
false
4 >= 4.
true
```

```
4 > 2.
true
2 < 4.
true
2 =< 4.
true
4 =< 4.
true
```

Example: Max of a List

Write a rule that finds the maximum of a list: **maxList(List,Max).**

Example: Max of a List

Write a rule that finds the maximum of a list: **maxList(List,Max).**

```
maxList([1,0,5,4,2], Max)
maxList([H|T],A,Max) :- H > A, maxList(T,H,Max).
maxList([H|T],A,Max) :- H =< A, maxList(T,A,Max).
maxList([],A,A).
```

```
?- maxList([1,0,5,4,2], Max).
Max = 5.
```

N.B. In this implementation two relations with different arities are defined:
maxList/2 and maxList/3

Built-in Predicates

Built-in predicates are already defined in Prolog implementation:

I/O predicates:

- **Write/1**: prints the argument value to the standard output
- **Nl/0**: breaks the line and returns a new line.
- **Tab/1**: inserts a number of tab spaces equal to the argument value.

When those predicate are called for the first time they success, if called more times they fail.

Not

It checks the **negation** of a predicate:

- **Not/1**: takes as argument a goal, Prolog tries to prove it and then negates the result.

Example:

?- not(is_cointained_in(key,office)).
false

Fail

- **fail/0**: It causes the predicate to fail.

It is useful to force backtracking and find more solutions.
It can be used to implement cycles, i.e. checking all the instantiations (by unification) that make a rule true.

Example:
Print all the elements within a list:

print_all(List):- is_contained_in(O,List),write(O),nl,fail.

Assert

- **assert/1**: adds the argument to the knowledge base.
- **asserta/1**: inserts the argument at the beginning of the knowledge base.
- **assertz/1**: appends the argument at the end of the knowledge base.

Assert and **assertz** actually behave the same.

Example:

```
insert_if_mortal(X):-person(X),assert(mortal(X)).
person(socrates).

?- insert_if_mortal(socrates).
true.
?- mortal(X).
X = socrates.
```

Retract

- **retract/1**: removes from the knowledge base the first fact or clause unifying with the argument.

- **retractall/1**: removes from the knowledge base any fact or clause unifying with the argument.

Findall

- **findall/3 (+Template, :Goal, -Bag)**: Create a list of the instantiations Template gets successively on backtracking over Goal and unify the result with Bag. Succeeds with an empty list if Goal has no solutions.

Example:

```
foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).

?- findall(C, foo(A, B, C), Cs).
Cs = [c, d, e, f, g]
```

Bagof

- **bagof/3 (+Template, :Goal, -Bag)**: Unify **Bag** with the alternatives of **Template**. If Goal has free variables besides the one sharing with Template, **bagof/3** will **backtrack over the alternatives** of these free variables, unifying Bag with the corresponding alternatives of Template. The construct **+Var^Goal** tells **bagof/3** not to bind Var in Goal. **bagof/3 fails** if Goal has no solutions.

Example:

foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).

?- bagof(C, foo(A, B, C), Cs).
A = a, B = b, C = G308, Cs = [c, d] ;
A = b, B = c, C = G308, Cs = [e, f] ;
A = c, B = c, C = G308, Cs = [g].

4 ?- bagof(C, A^foo(A, B, C), Cs).
A = G324, B = b, C = G326, Cs = [c, d] ;
A = G324, B = c, C = G326, Cs = [e, f, g].

Setof

- **setof/3 (+Template, :Goal, -Bag)**: it behaves as bagof but sorts the result to get a sorted list of alternatives without duplicates.

Example:

```
foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).
```

```
?- setof(C, foo(A, B, C), Cs).
A = a,
B = b,
Cs = [c, d] ;
A = b,
B = c,
Cs = [e, f] ;
A = B, B = c,
Cs = [g].
```

Cut

- **!/0**: Discards all alternatives since entering the predicate in which the cut appears. It **COMMITS** to the clause in which the cut appears and **FORCES** Prolog interpreter to not use backtracking to find alternatives for the goals to the left of cut in the current clause.

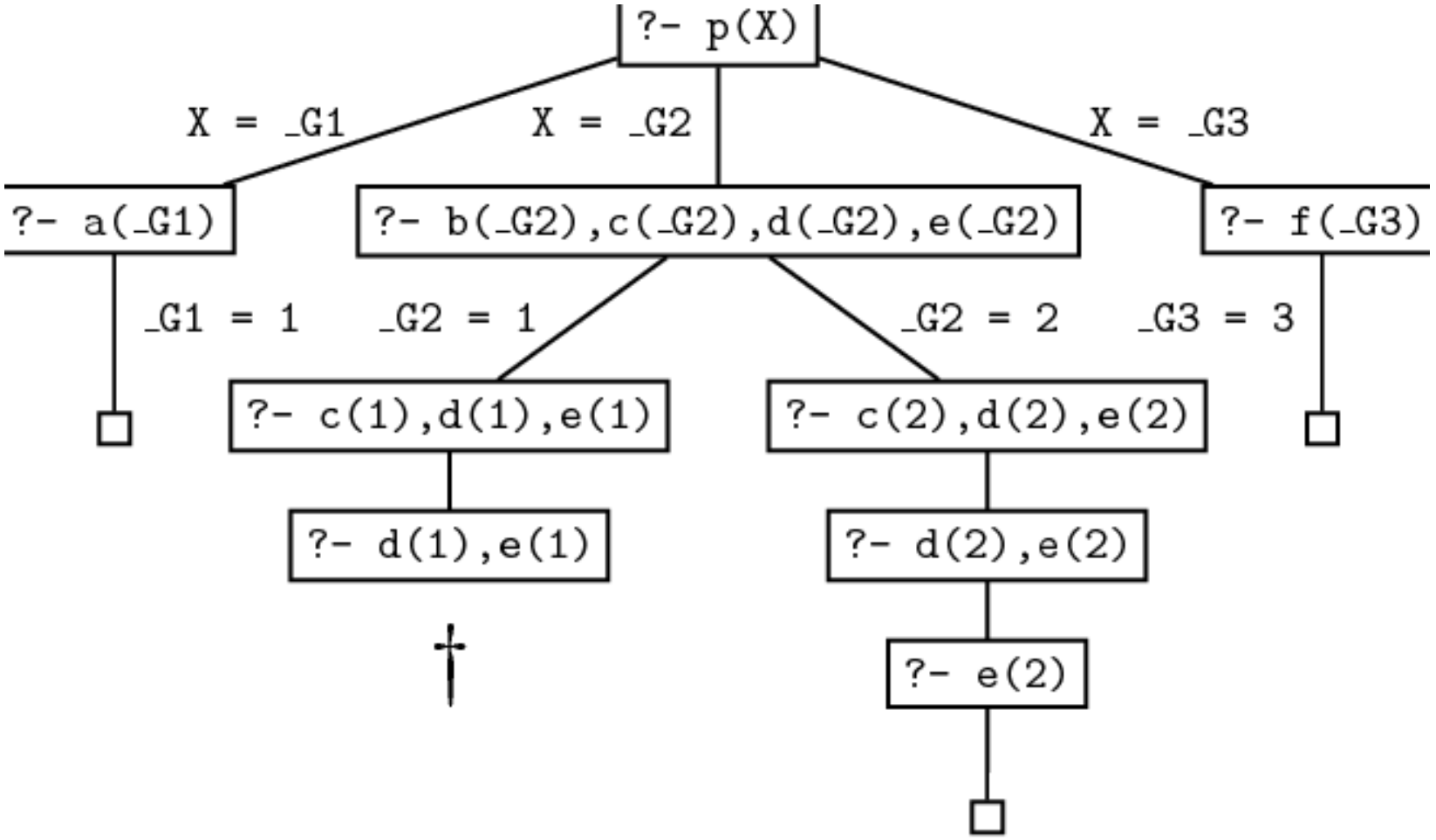
Example:

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).
c(1).
d(2).
e(2).
f(3). b(2). c(2).
```

```
?- p(X).
X = 1 ;
X = 2 ;
X = 3 .
```

it has to backtrack once, namely when it enters the second clause for p/1 and decides to unify the first goal withb(1) instead of b(2) .

Cut


?- p(X)
X = _G1
X = _G2
X = _G3
?- a(_G1)
?- b(_G2),c(_G2),d(_G2),e(_G2)
?- f(_G3)
_G1 = 1
_G2 = 1
_G2 = 2
_G3 = 3
?- c(1),d(1),e(1)
?- c(2),d(2),e(2)
?- d(1),e(1)
?- d(2),e(2)
?- e(2)

Cut

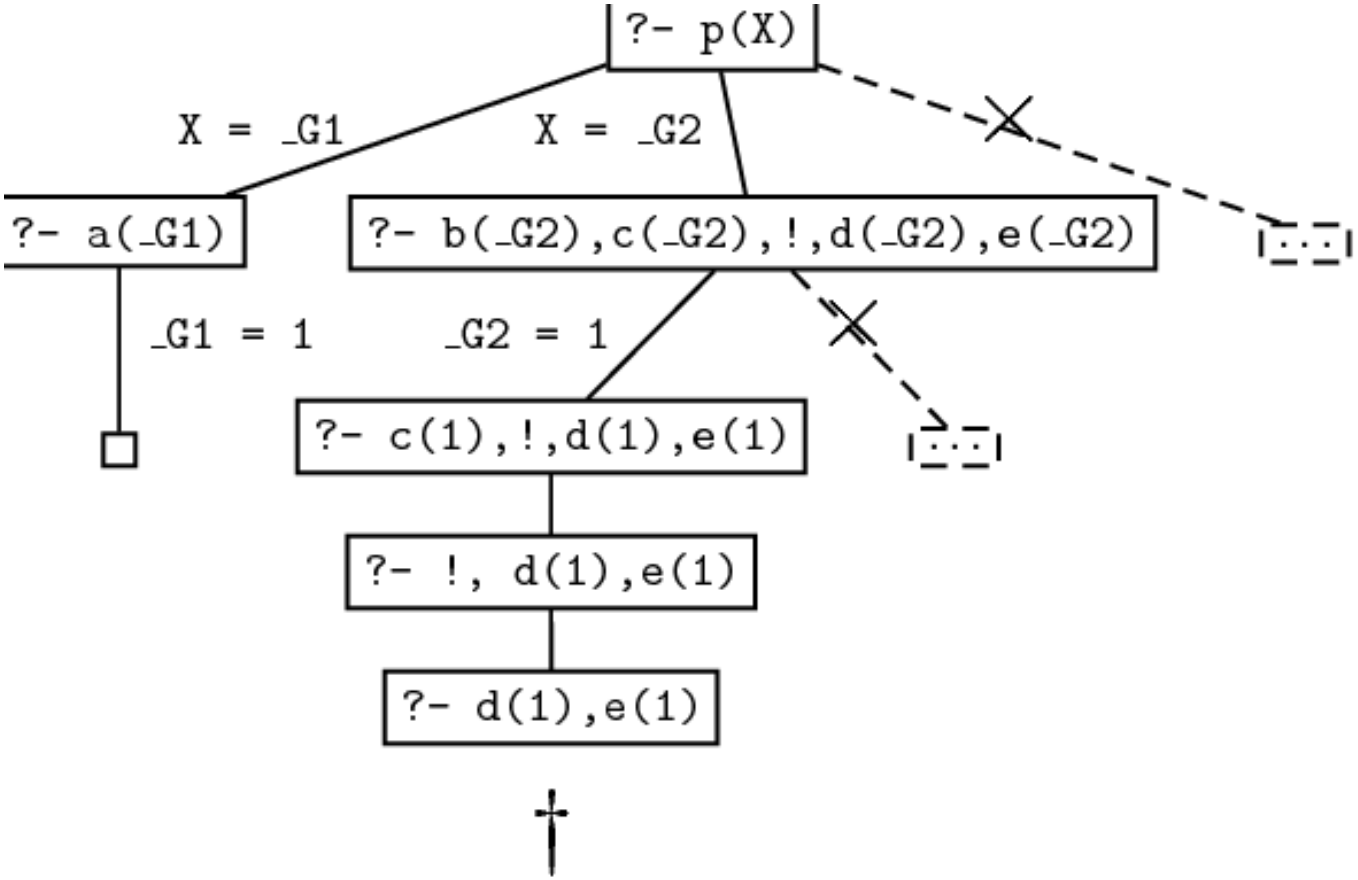
Now we modify the second clause inserting a cut:

Example:

```
p(X):- a(X).
p(X):- b(X), c(X), !, d(X), e(X).
p(X):- f(X).
a(1).
b(1).
c(1).
d(2).
e(2).
f(3). b(2). c(2).
```

```
?- p(X).
X = 1.
```

Cut



Cut

1. **p(X)** is first unified with the first rule, so we get a new goal **a(X)** . By instantiating X to 1 , Prolog **unifies a(X)** with the fact **a(1)** and we have found a solution. So far, this is exactly what happened in the first version of the program.
2. We then go on and look for a second solution. **p(X) is unified with the second rule**, so we get the new goals **b(X),c(X),!,d(X),e(X)** . By instantiating X to 1 , Prolog unifies **b(X) with the fact b(1)** , so we now have the goals c(1),!,d(1),e(1) . But **c(1) is in the database** so this simplifies to !,d(1),e(1) .
3. Now for the big change. The ! goal succeeds (as it always does) and **commits us to the choices made so far**. In particular, **we are committed to having X = 1 , and we are also committed to using the second rule.**
4. But **d(1) fails**. And there's **no way we can re-satisfy the goal p(X)** . Sure, if we were allowed to try the value X=2 we could use the second rule to generate a solution (that's what happened in the original version of the program). But we can't do this: **the cut has removed this possibility from the search tree**. And sure, if we were allowed to try the third rule, we could generate the solution X=3 . But we can't do this: once again, the cut has removed this possibility from the search tree.

Cut

N.B. The **cut only commits us to choices made since the parent goal was unified with the left hand side of the clause** containing the cut. For example, in a rule of the form

q:- p1,...,pn, !, r1,...,rm

when we reach the cut it commits us to using this particular clause for q and it commits us to the choices made when evaluating p1,...,pn .

However, **we are free to backtrack among the r1,...,rm** and **we are also free** to backtrack among **alternatives** for choices that were **made before reaching the goal q** .

Cut – More complete Example

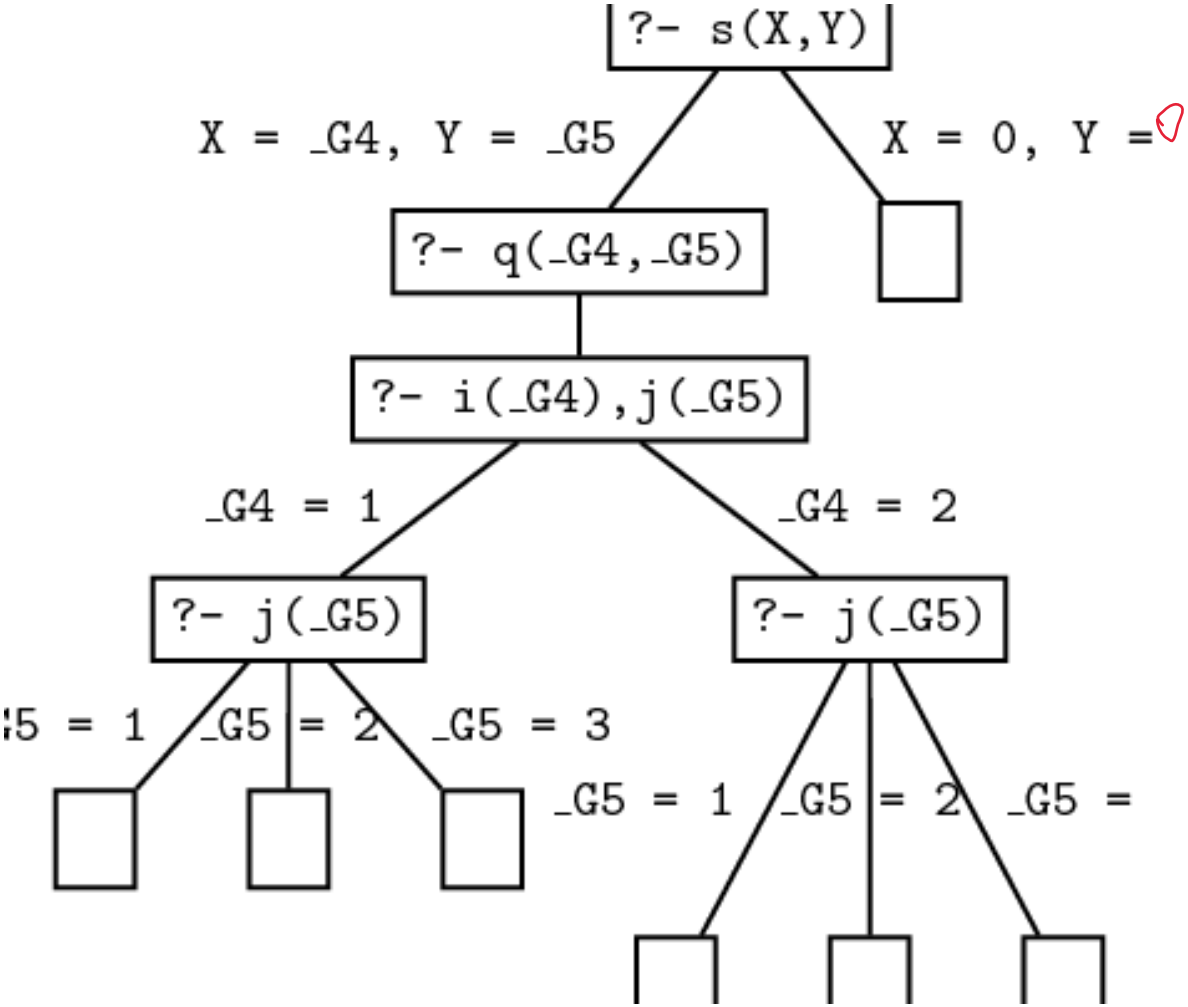
Consider the following cut-free program:

```
s(X,Y):- q(X,Y).
s(0,0).

q(X,Y):- i(X), j(Y).

i(1).
i(2).

j(1).
j(2).
j(3).
```

Cut – More complete Example

?- s(X,Y).

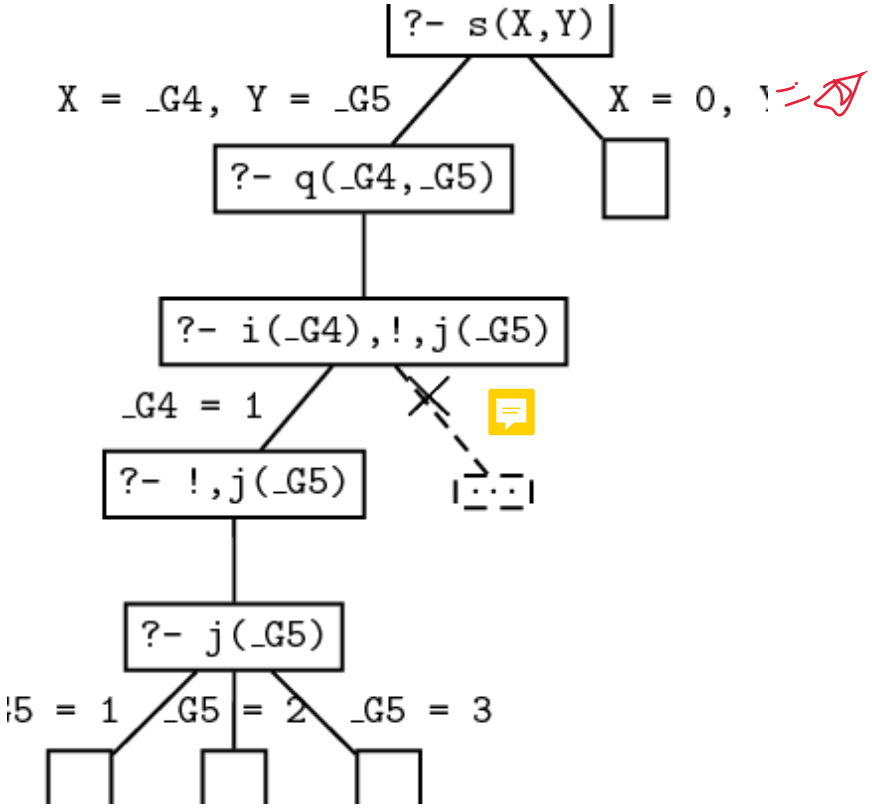
X = 1
Y = 1 ;
X = 1
Y = 2 ;
X = 1
Y = 3 ;
X = 2
Y = 1 ;
X = 2
Y = 2 ;
X = 2
Y = 3 ;
X = 0
Y = 0.



Cut – More complete Example

Consider the following program:

```
s(X,Y):- q(X,Y).
s(0,0).

q(X,Y):- i(X), !, j(Y).

i(1).
i(2).

j(1).
j(2).
j(3).
```




```
?- s(X,Y).

X = 1
Y = 1 ;

X = 1
Y = 2 ;

X = 1
Y = 3 ;

X = 0
Y = 0.
```

Cut – More complete Example

1. s(X,Y) is **first unified with the first rule**, which gives us a new goal q(X,Y) .
2. q(X,Y) is **then unified with the third rule**, so we get the new goals i(X),!,j(Y) . **By instantiating X to 1 , Prolog unifies i(X) with the fact i(1)** . This leaves us with the goal !,j(Y) . The cut succeeds, and **commits us to the choices made so far**.
3. **But what are these choices? These: that X = 1 , and that we are using this clause. But note: we have not yet chosen a value for Y .**
4. Prolog then goes on, and by instantiating Y to 1 , Prolog unifies j(Y) with the fact j(1) . So we have found a solution.
5. But we can find more. Prolog is free to try another value for Y . So it backtracks and sets Y to 2 , thus finding a second solution. And in fact it can find another solution: on backtracking again, it sets Y to 3 , thus finding a third solution.
6. **But those are all alternatives for j(X) . Backtracking to the left of the cut is not allowed**, so it can't reset X to 2 , so it won't find the next three solutions that the cut-free program found. **Backtracking over goals that were reached before q(X,Y) is allowed however, so that Prolog will find the second clause for s/2 .**

Cut - Example

Write a program, max/3 which takes integers as arguments and succeeds if the third argument is the maximum of the first two:

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

This program is correct BUT it is inefficient. In a very large program, if backtracking is forced, the program will try to satisfy the second clause (but there is no second solution to find!!!). The clauses are mutually exclusive so we rewrite the program with cut:

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

N.B. This CUT does not change the meaning of the program, but it is more efficient. **CUTS** like this are called **GREEN CUTS**

Cut - Example

Somebody could points that the second clause is redundant so she could propose a more compact (and **wrong**) version:

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

This program is wrong for simple queries like **max(2,3,2).**
This query will succeed!
The right version is:

```
max(X,Y,Z) :- X =< Y,!, Y = Z.
max(X,Y,X).
```

N.B. This cut changes the meaning of the program. Without the cut the program will behave differently. **Be very careful with those cuts**.
CUTS like this are called **RED CUTS**.