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.

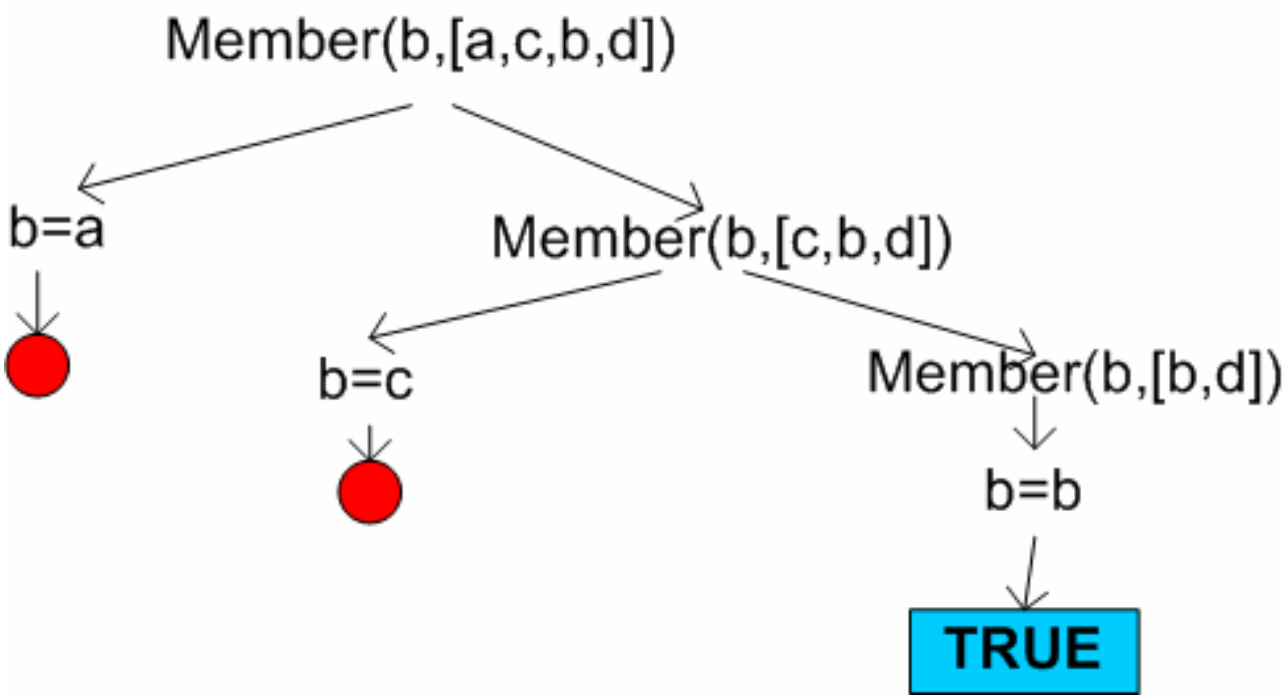
As expected after the check of the boundary condition the rule call recursively itself to check the remaining part of the list.
N.B. The variables names can be different in the rules.

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).
```

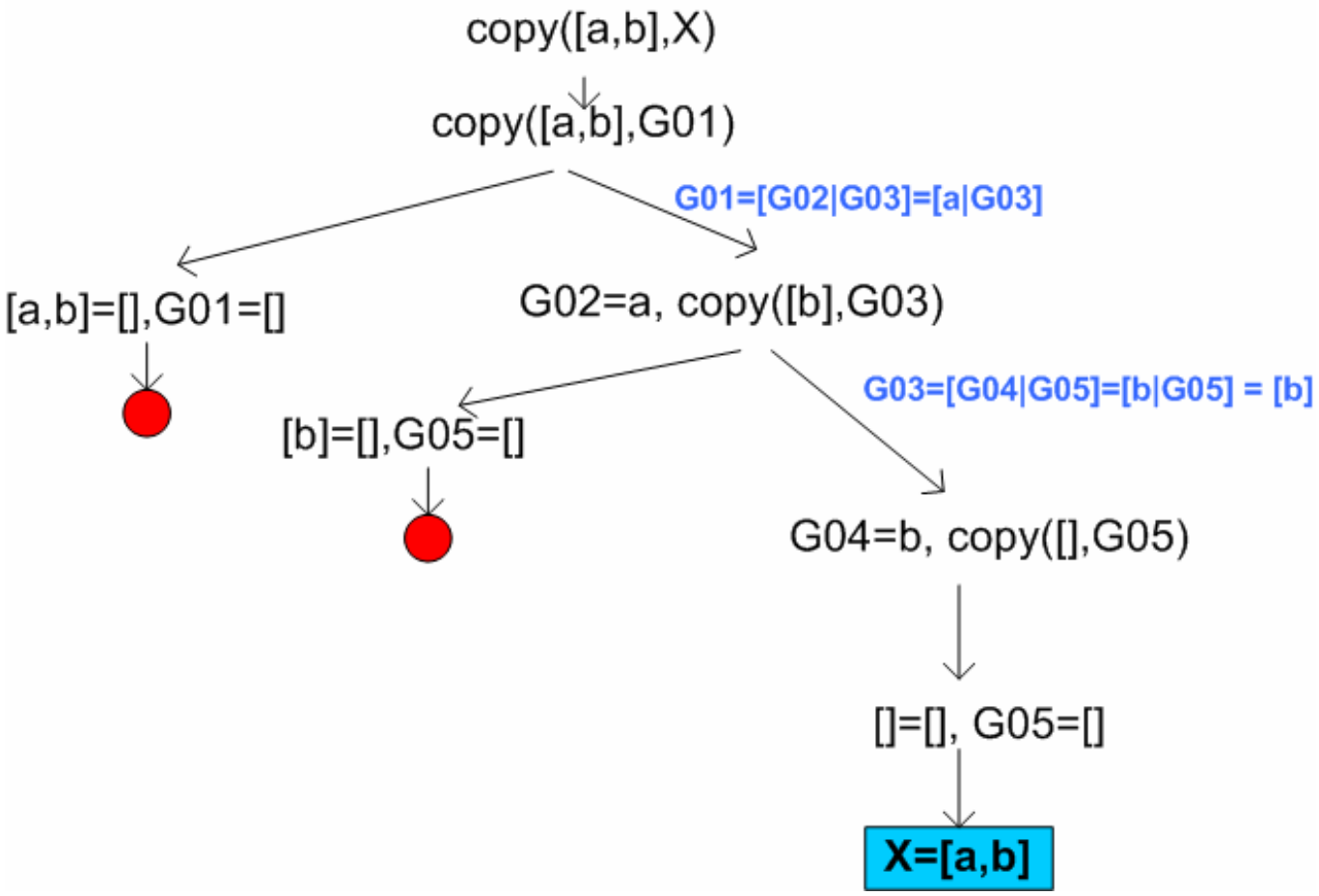
Now we can write a rule to copy a list in another one, element by element.

```
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

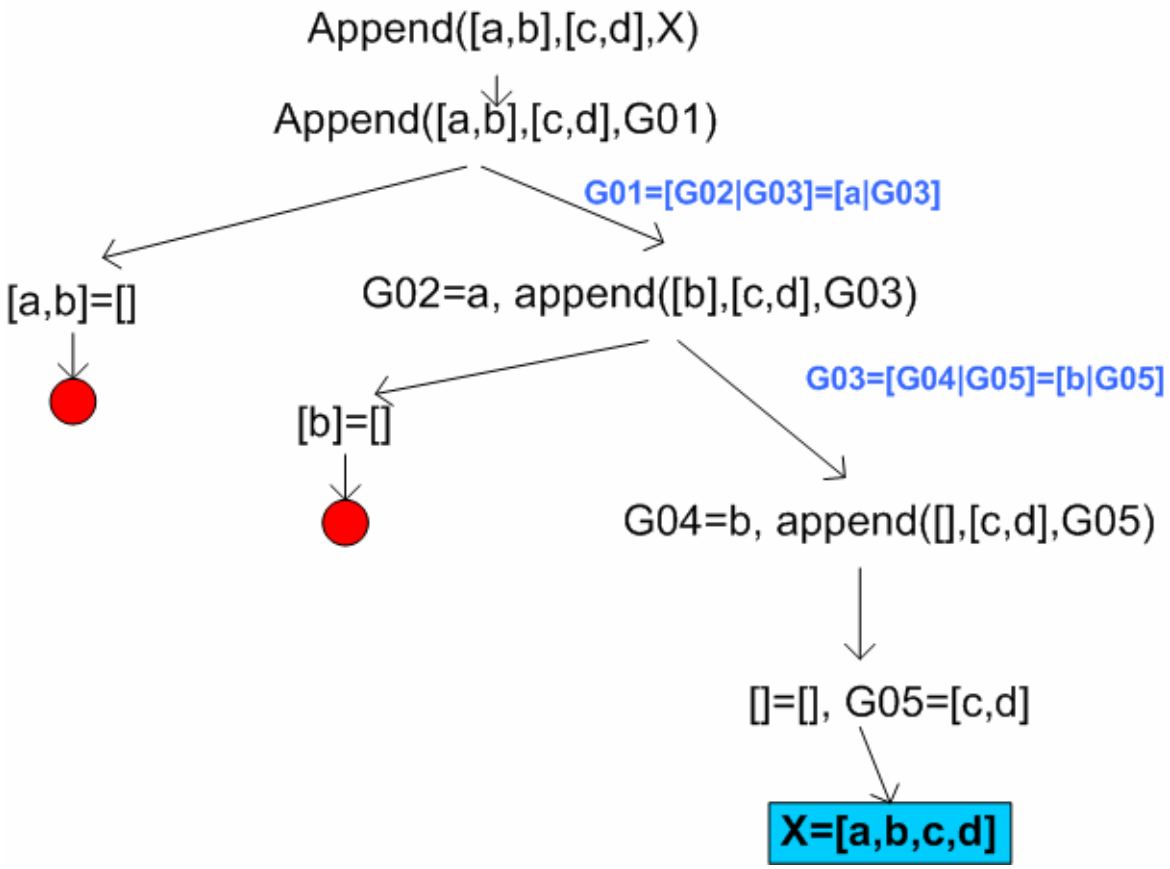
Let us write the append rule in a more compact form:

```
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.

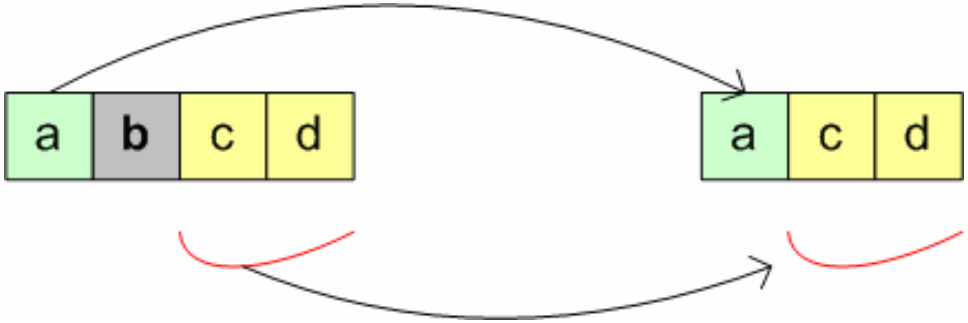
N.B. in the boundary condition the third parameters is the **tail** of the new list, **that is a list itself**, hence we want that an element (**E**) will be turned in a closed list (**[E]**) to let Prolog to do a right unification.

Example 6: Remove an element from a list

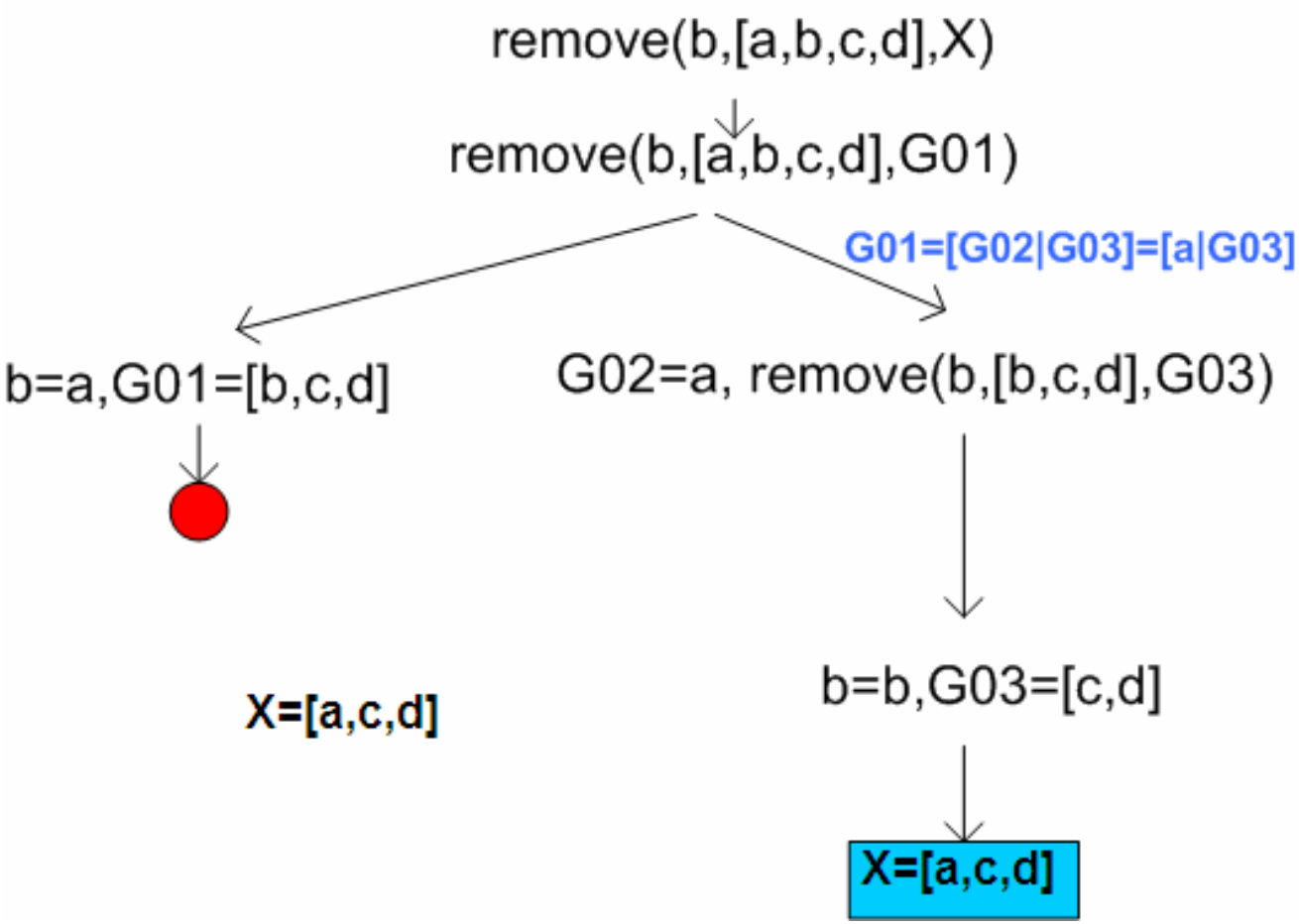
We want a rule that removes the first occurrence of a stated element from a list (More occurrences imply more solutions):
remove(Elem,SourceList, DestList).

```
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

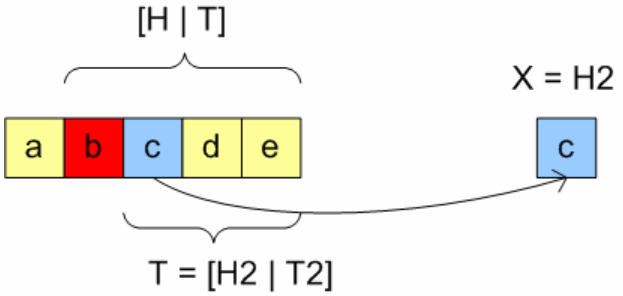
We want a rule that finds an element within a list and returns the next element.

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

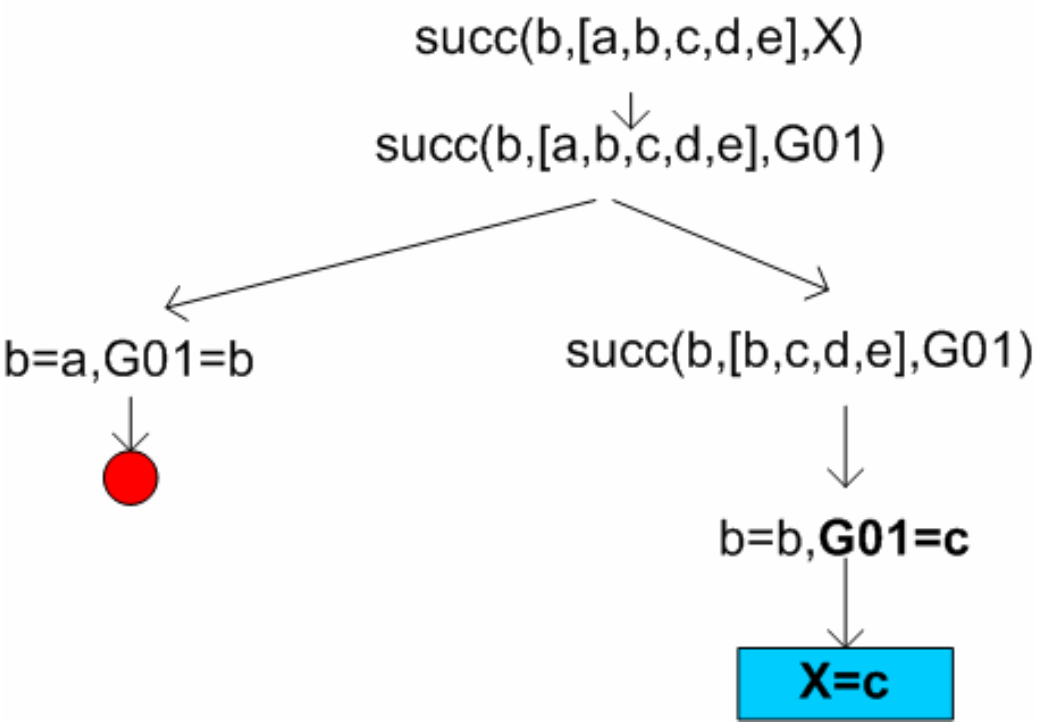
OR

```
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

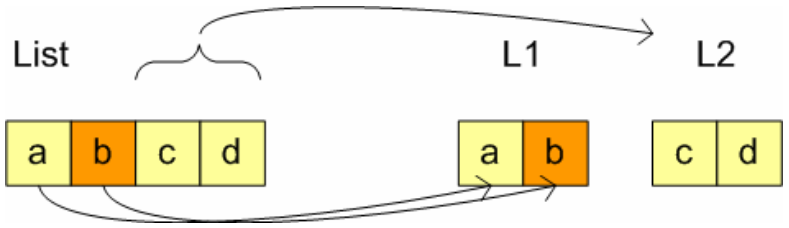
We want a rule that finds an element **E**, then it will create two lists: in the first one the last element will be **E**, in the second one the first element will be the following element in the source list:
divide(List,Element,List1,List2).

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

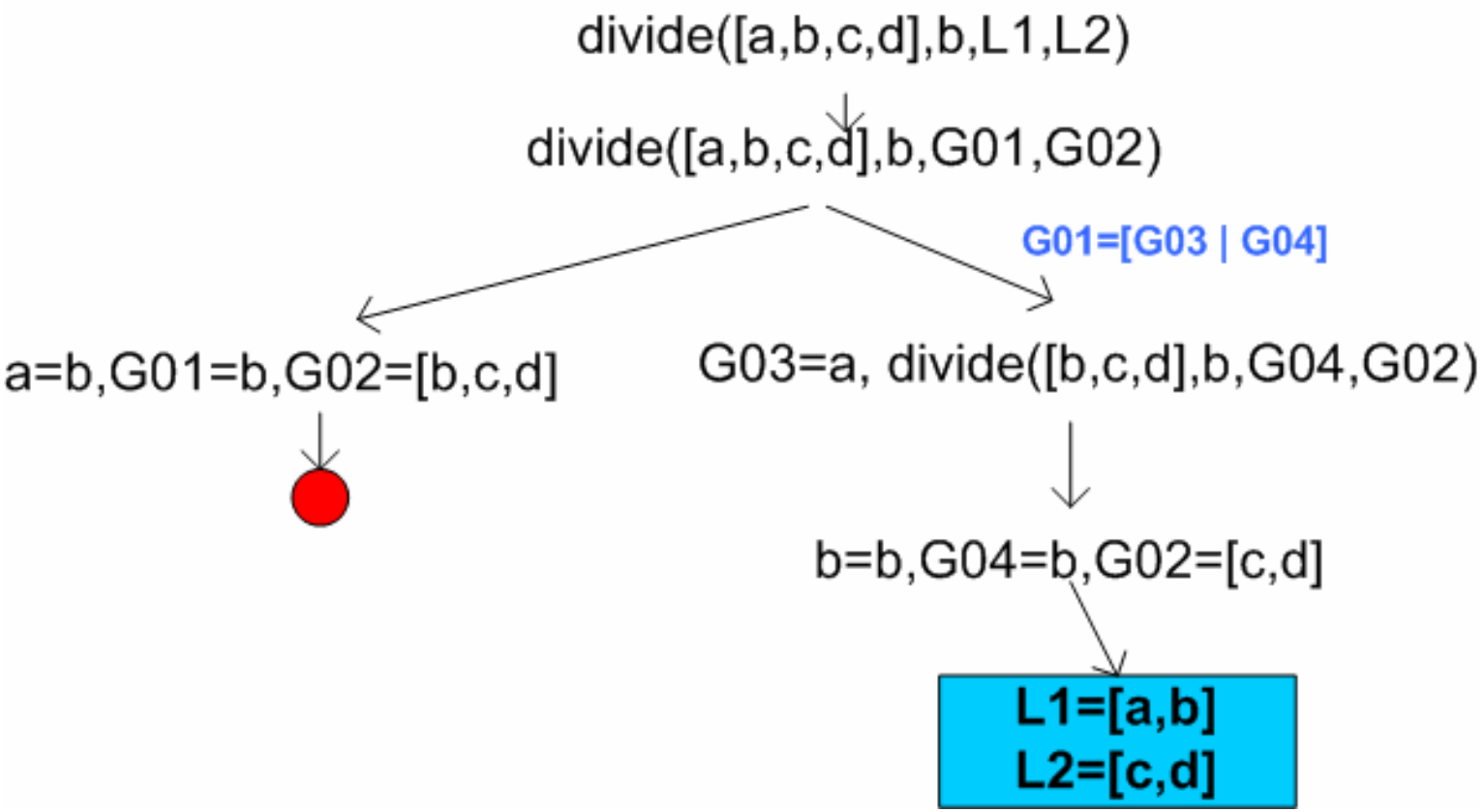
OR

```
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

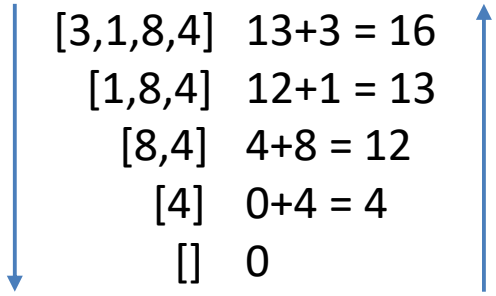


Example 9: Sum and count the elements

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.



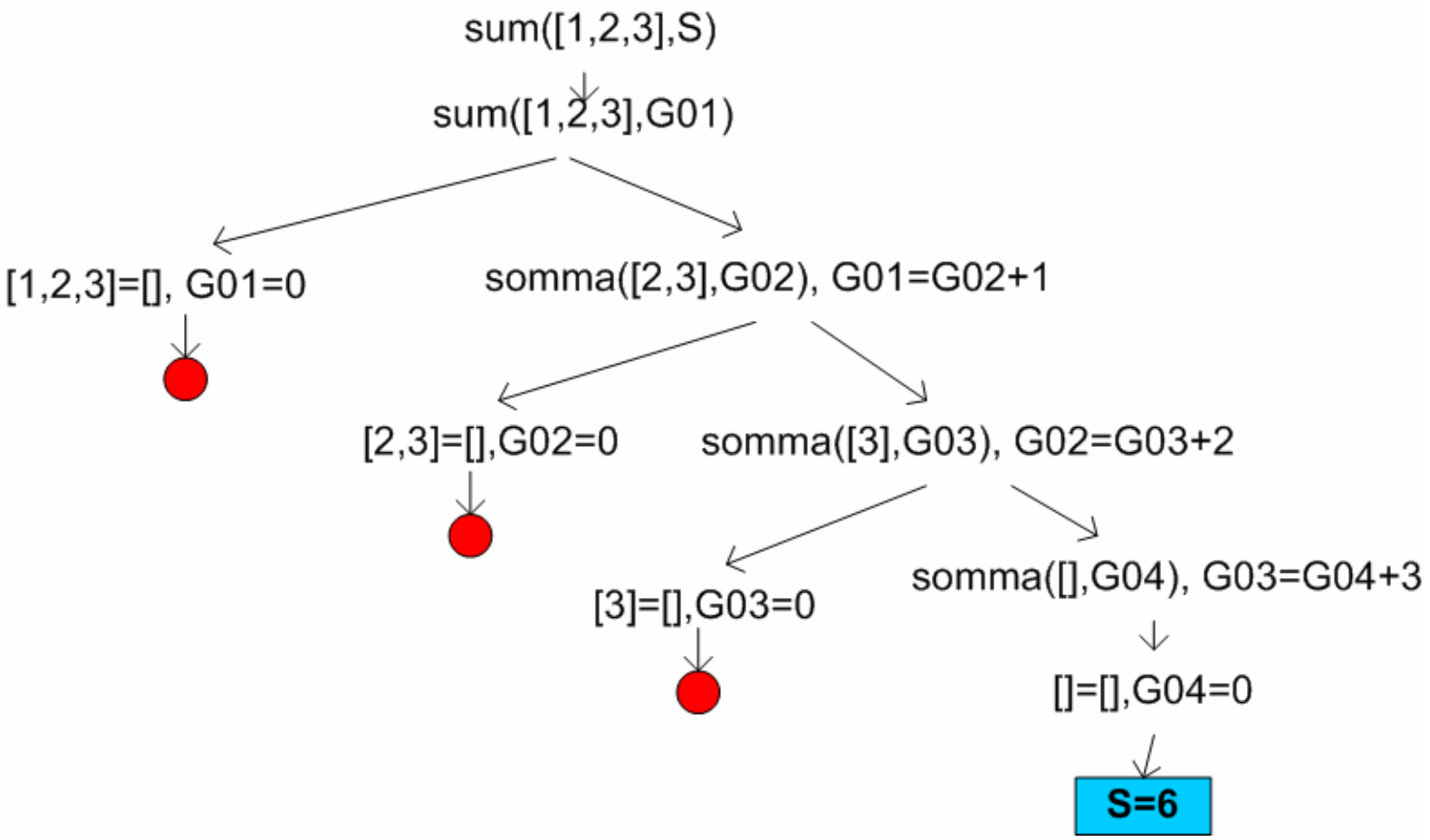
Example 9: Sum and count the elements

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

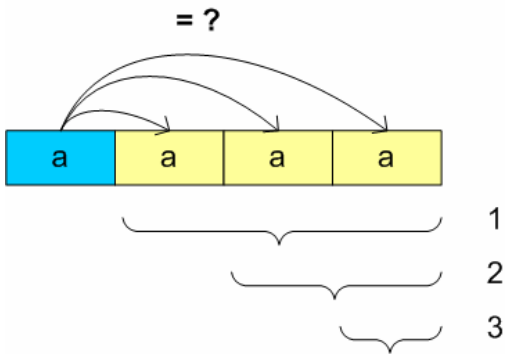
We want a rule that checks if all elements are equals.
We will compare all the elements in the list with the first one. If a check fails, the query fails.

```
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**.



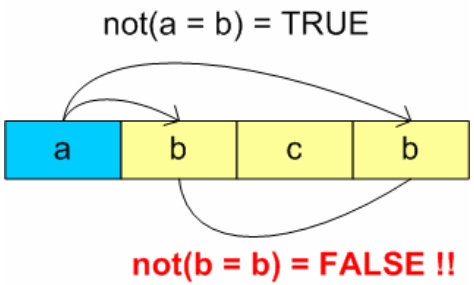
Example 11: All different

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**

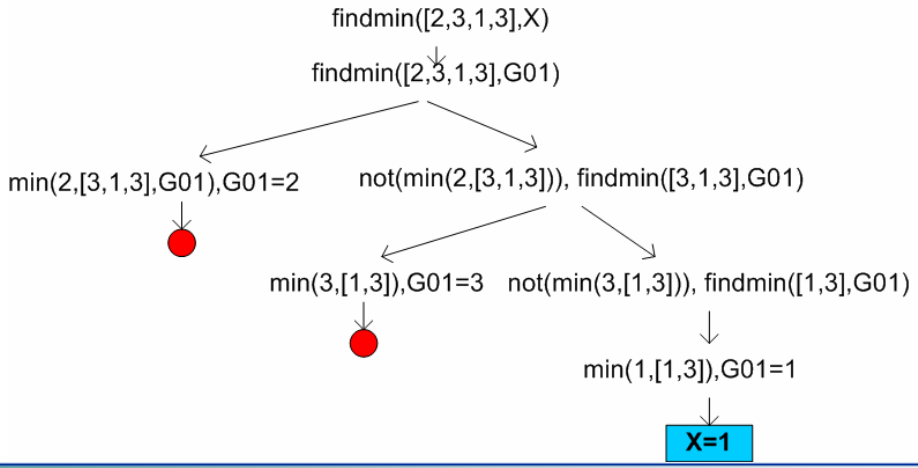
Example 12: Minimum of a list

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).

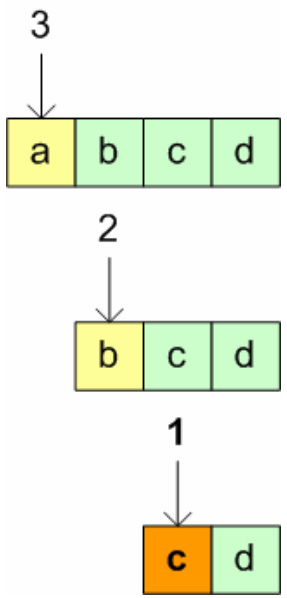
a) Find the last element in the source list
b) Add the minimum as head of the destination list
c) Remove the element from the source list
d) Repeat from **a)** on **NL** and on the **tail** of the **list L2**

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

Example 15: Nth element of a list

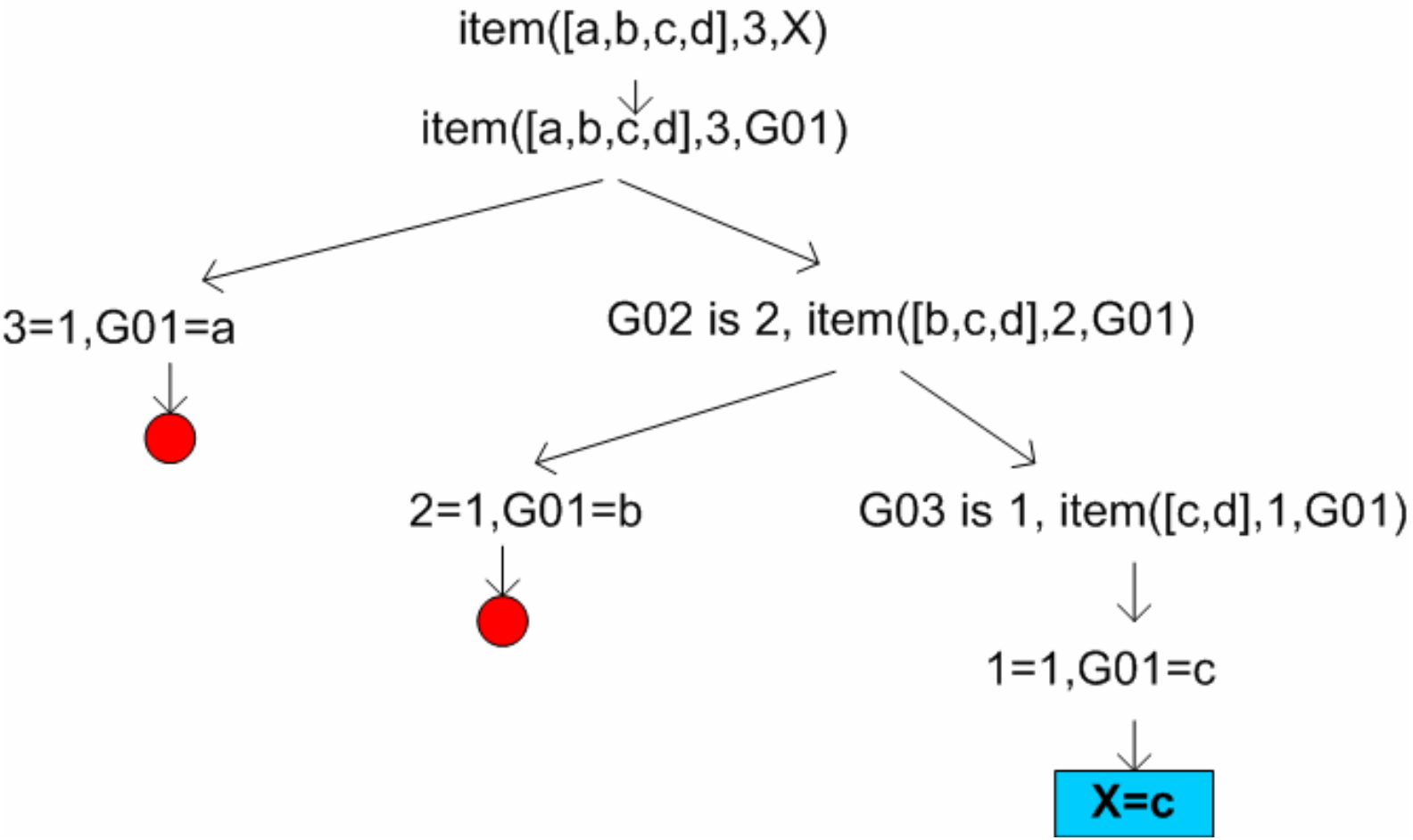
Write a rule that finds the element with index N in a list.

```
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.

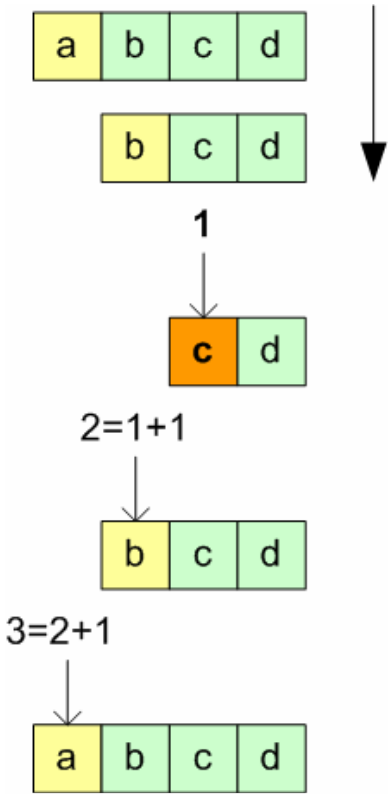
Example 15: Nth element of a list



Example 16: Index of element in a list

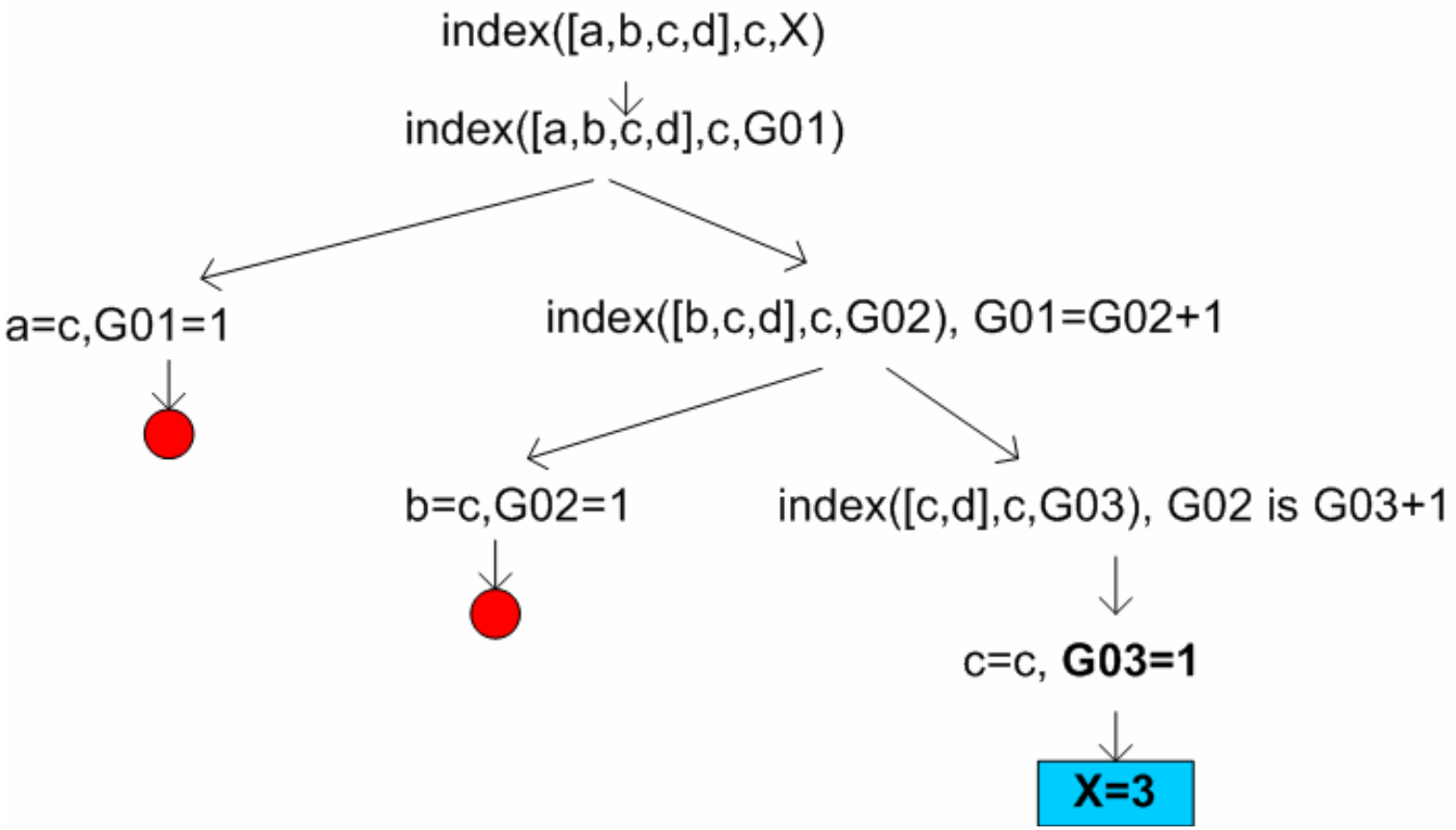
Write a rule that finds the index (or the indices) of an element in a list. The list is 1-indexed.

```
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).
```

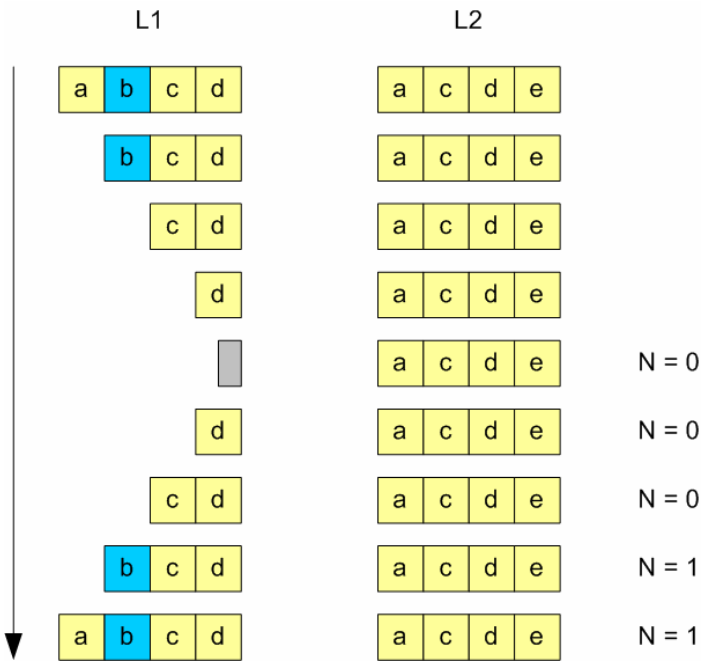
This version **finds minimum of both lists** and **it finds the lower between them**. That **element** is **removed** from the list and **added as head** of the destination list.

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).**