Logic and Constraint Programming

PROLOG

Prof. Fabrizio Fornari

May 27, 2022

Representation of lists

The *list* is a simple data structure widely used in non-numeric programming.

A sequence list is a sequence of any number of items, such as:

[ann, tennis, tom, skiing]

Remember: all structured objects in Prolog are trees. Lists are no exception to this.

Representation of lists

A list can be empty or non-empty.

Empty: **[ ]** .

Non-Empty:

1. the first items is called the *head* of the list;
2. the remaining part of the list is called the *tail*.

$$\textbf{[}\underbrace{\textbf{ann,}}_{\text{Head}}\ \underbrace{\textbf{tennis, tom, skiing}}_{\text{Tail}}\textbf{]}$$

Representation of lists

$$[\ \underbrace{\textbf{ann}}_{\text{Head}},\ \underbrace{\textbf{tennis, tom, skiing}}_{\text{Tail}}\ ]$$

The tail has to be a list. The head and the tail are combined into a structure by the functor “.”:

.(Head, Tail)

Since **Tails** is in turn a list, it is either empty or it has its own head and tail

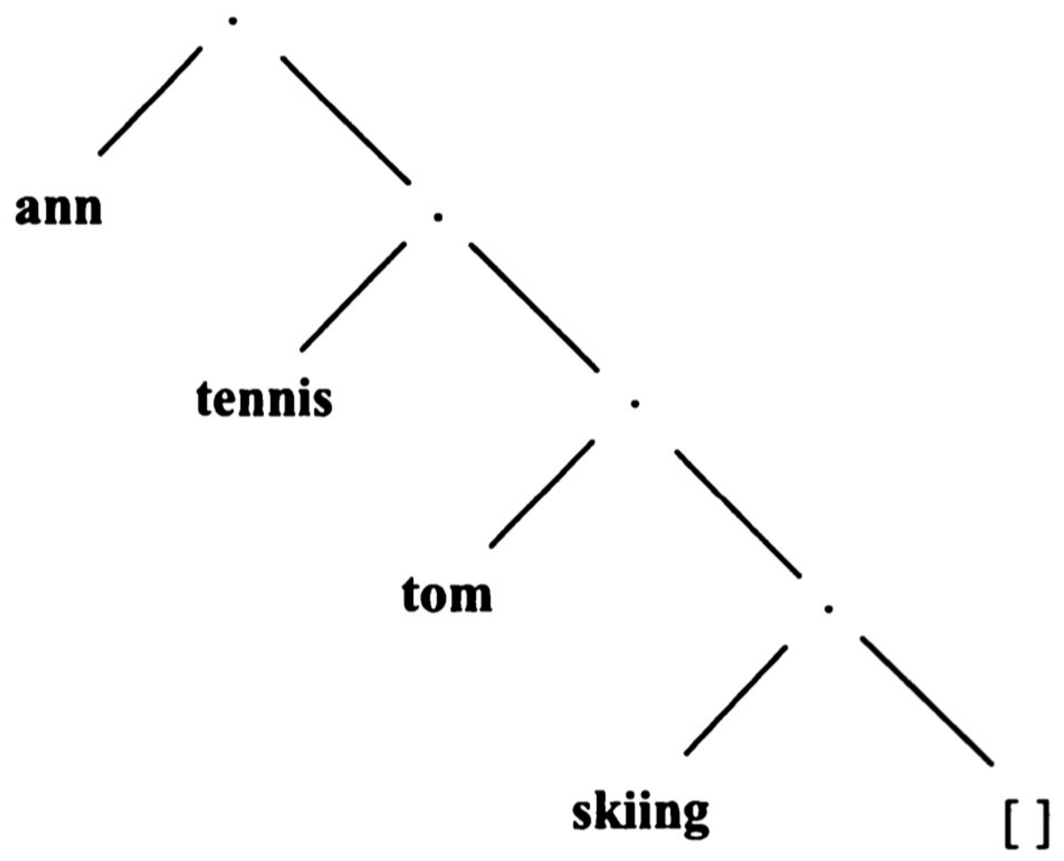
Representation of lists

$$[\ \underbrace{\textbf{ann,}}_{\text{Head}}\ \underbrace{\textbf{tennis, tom, skiing]}}_{\text{Tail}}$$

$$\textbf{.(ann, .(tennis, .(tom, .(skiing, } \underbrace{\textbf{[ ]}}_{\text{Empty list}} \textbf{))))}$$

How Prolog interprets a List

.(Head, Tail)



Operations on lists

Membership relation. Checking whether some object is an element of a list

Concatenation of two lists, obtaining a third one

Adding a new object to a list, or **deleting** an object from it.

Sublist

Let us define the sublist relation such that:

sublist([c,d,e], [a,b,c,d,e,f]) is true
sublist([c,e], [a,b,c,d,e,f]) is false

S is a sublist of L if:
1. L can be decomposed into two lists, L1 and L2, and
2. L2 can be decomposed into two lists, S and some L3.

```
sublist( S, L) :-
  conc( L1, L2, L),
  conc( S, L3, L2).
```

What if we ask **sublist(S, [a,b,c])**?

Permutation

We can generate permutations of a list through backtracking using the **permutation** procedure, as in the following:

```
?- permutation( [a,b,c], P).

P = [a, b, c] ;
P = [a, c, b] ;
P = [b, a, c] ;
P = [b, c, a] ;
P = [c, a, b] ;
P = [c, b, a] ;
false.
```

```
?- random_permutation( [a,b,c], P).
```

List length

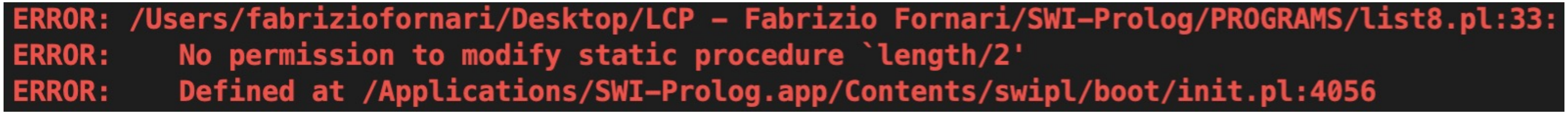
Let us count the elements in a list **List** and instantiate N to thein number.

1. If the list is empty then its length is 0

2. If the list is not empty then **List = [Head | Tail]**; then its length is equal to 1 plus the length of the tail **Tail**

```
length( [ ], 0).

length([_ | Tail], N) :-
   length( Tails, N1),
   N is 1 + N1.
```

```
?- length([ a,b, [ c,d], e], N).

N = 4
```



```
ERROR: /Users/fabriziofornari/Desktop/LCP - Fabrizio Fornari/SWI-Prolog/PROGRAMS/list8.pl:33:
ERROR:    No permission to modify static procedure `length/2'
ERROR:    Defined at /Applications/SWI-Prolog.app/Contents/swipl/boot/init.pl:4056
```

SWI-Prolog library

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

Let us define some relations

max(X, Y, Max) so that **Max** is the greater of two numbers X and Y.

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

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

Let us define some relations

maxlist(List, Max) so that **Max** is the greatest number in the list of numbers denoted by **List**.

```
maxlist([ X],X).                    % Maximum of single-element list

maxlist([X, Y | Rest], Max) :-      % At least two elements in list
   maxlist( [Y| Rest ], MaxRest),
   max( X, MaxRest, Max).           % Max is the greater of X and MaxRest
```

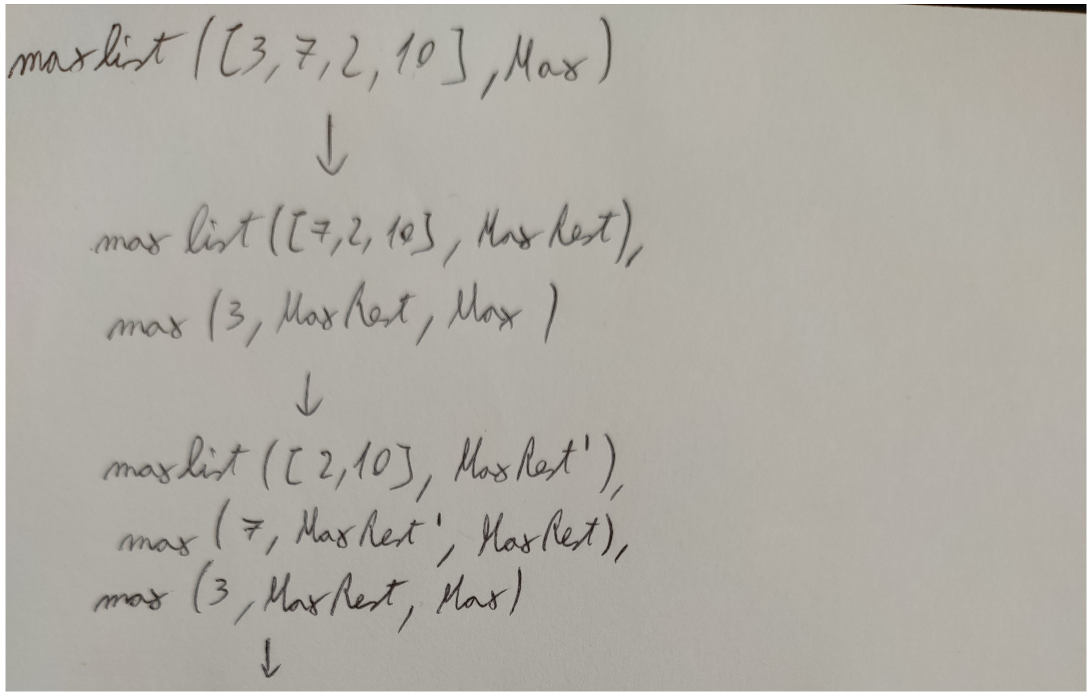
Let us define some relations

maxlist(List, Max) so that **Max** is the greatest number in the list of numbers denoted by **List**.

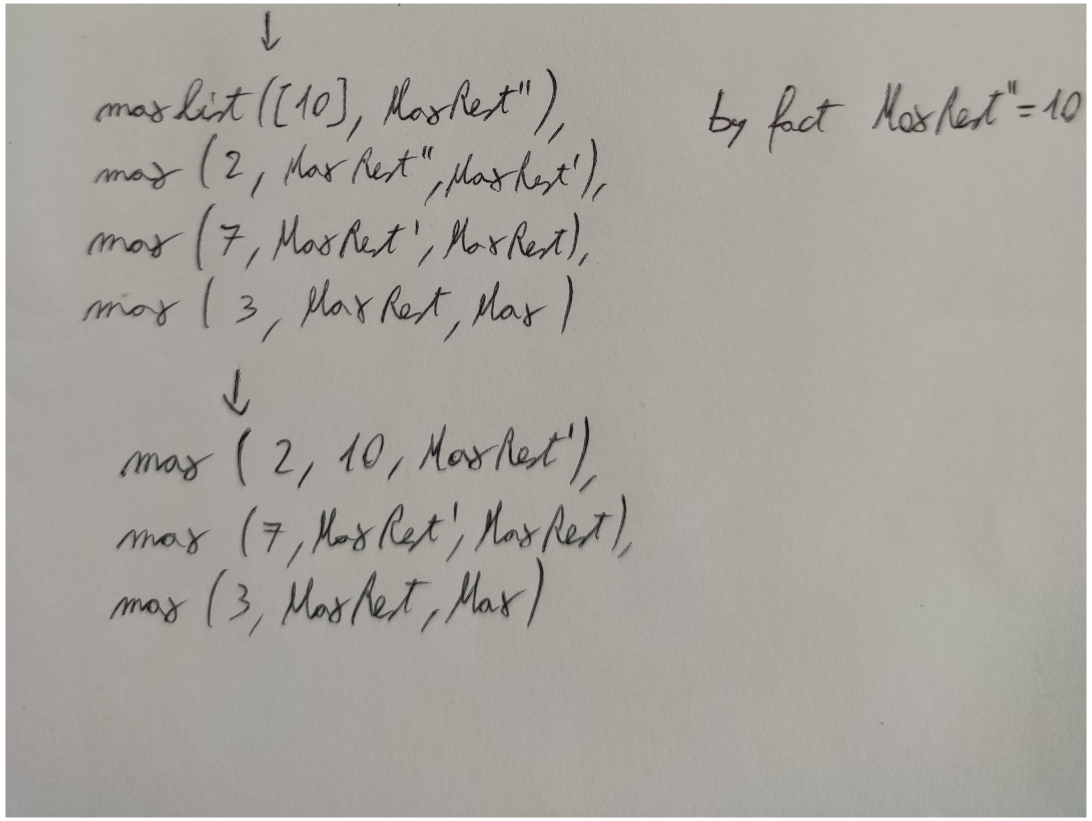
Which is the execution trace of **maxlist([3, 7, 2, 10], MaxRest)**

Let us define some relations

Which is the execution trace of **maxlist([3, 7, 2, 10], MaxRest)**



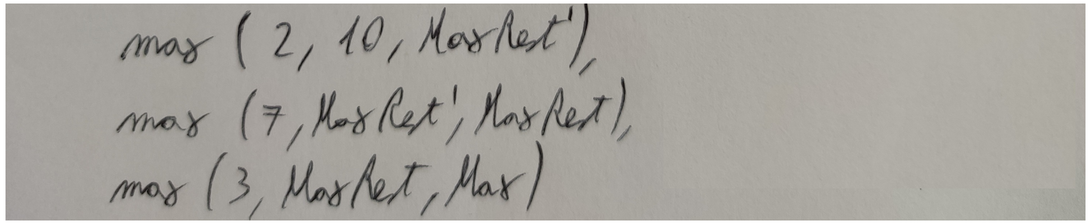
```
maxlist([3,7,2,10],Max)
        ↓
maxlist([7,2,10],MaxRest),
max(3,MaxRest,Max)
        ↓
maxlist([2,10],MaxRest'),
max(7,MaxRest',MaxRest),
max(3,MaxRest,Max)
        ↓
```



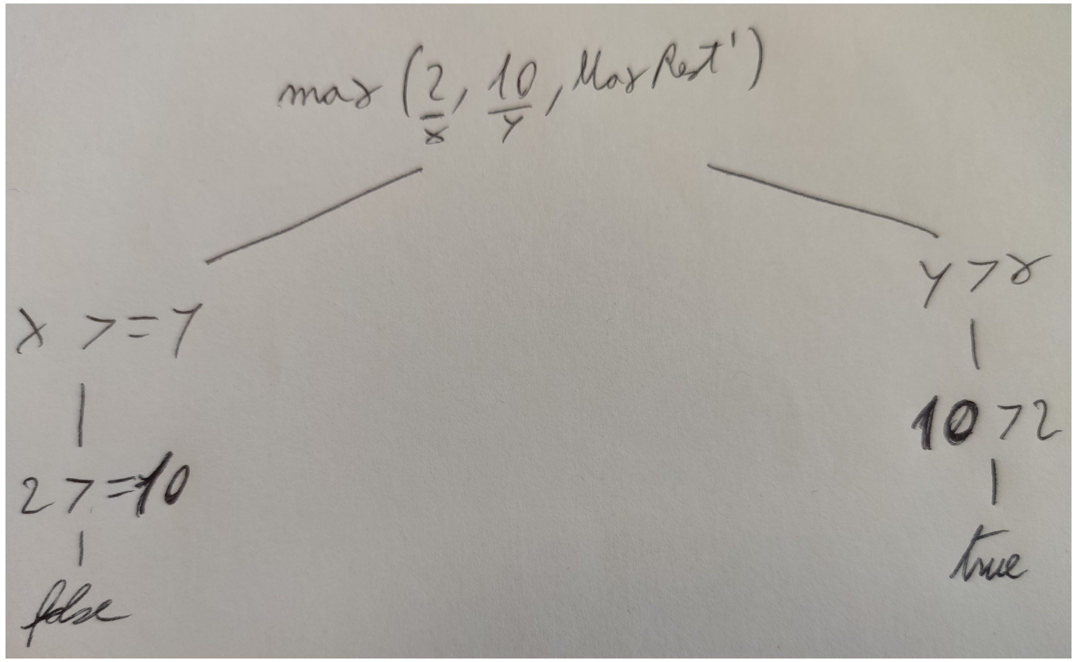
```
        ↓
maxlist([10],MaxRest''),              by fact MaxRest''=10
max(2,MaxRest'',MaxRest'),
max(7,MaxRest',MaxRest),
max(3,MaxRest,Max)
        ↓
max(2,10,MaxRest'),
max(7,MaxRest',MaxRest),
max(3,MaxRest,Max)
```

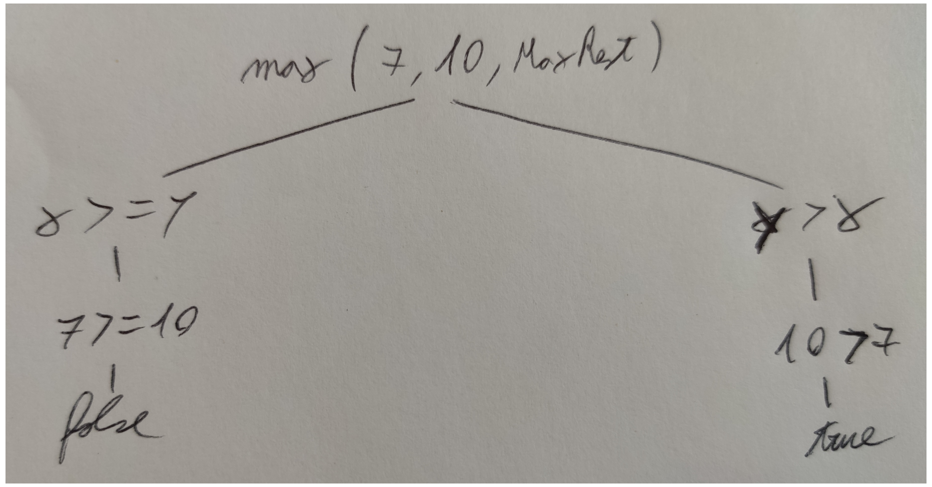
Let us define some relations

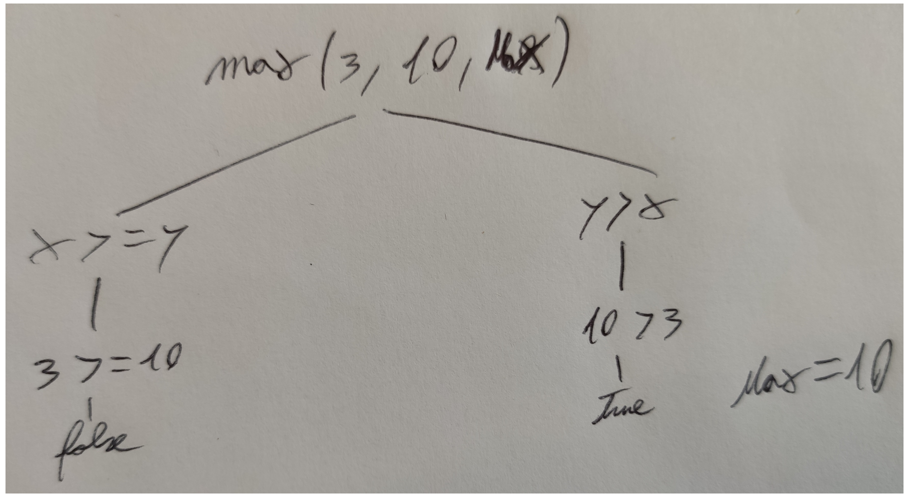
Which is the execution trace of **maxlist([3, 7, 2, 10], MaxRest)**



max(2, 10, MaxRest'),
max(7, MaxRest', MaxRest),
max(3, MaxRest, Max)





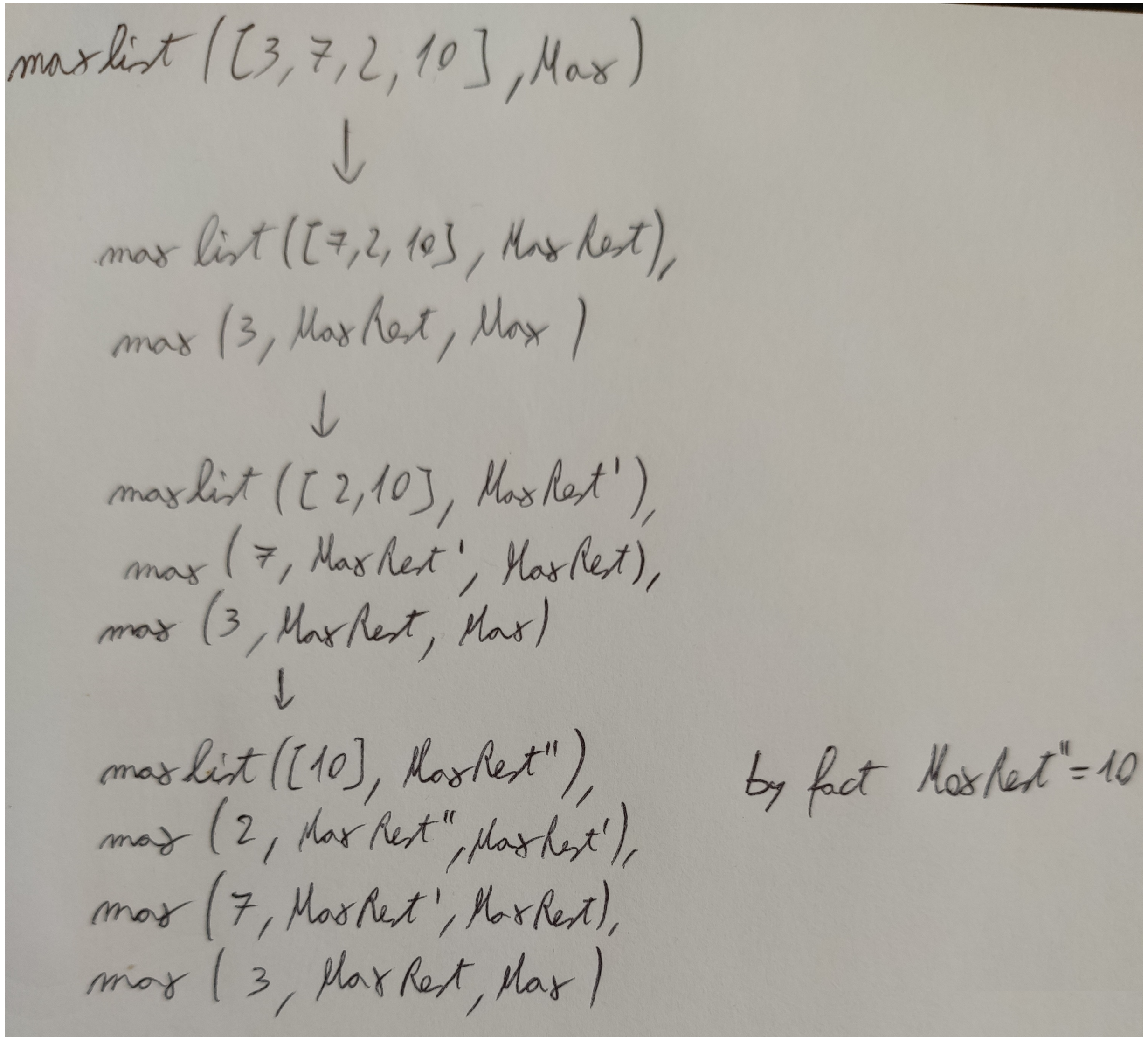


Let us define some relations

?- trace.

```
[trace]  ?- maxlist([3,7,2,10], Max).
```

```
Call: (10) maxlist([3, 7, 2, 10], _19236) ? creep
Call: (11) maxlist([7, 2, 10], _20488) ? creep
Call: (12) maxlist([2, 10], _21250) ? creep
Call: (13) maxlist([10], _22012) ? creep
Exit: (13) maxlist([10], 10) ? creep
```



maxlist([3,7,2,10], Max)

↓

max list([7,2,10], MaxRest),
max(3, MaxRest, Max)

↓

maxlist([2,10], MaxRest'),
max(7, MaxRest', MaxRest),
max(3, MaxRest, Max)

↓

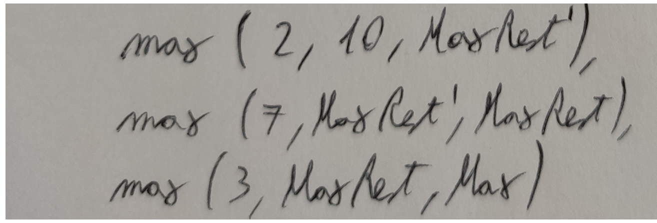
maxlist([10], MaxRest''),   by fact MaxRest''=10
max(2, MaxRest'', MaxRest'),
max(7, MaxRest', MaxRest),
max(3, MaxRest, Max)

Let us define some relations

?- trace.

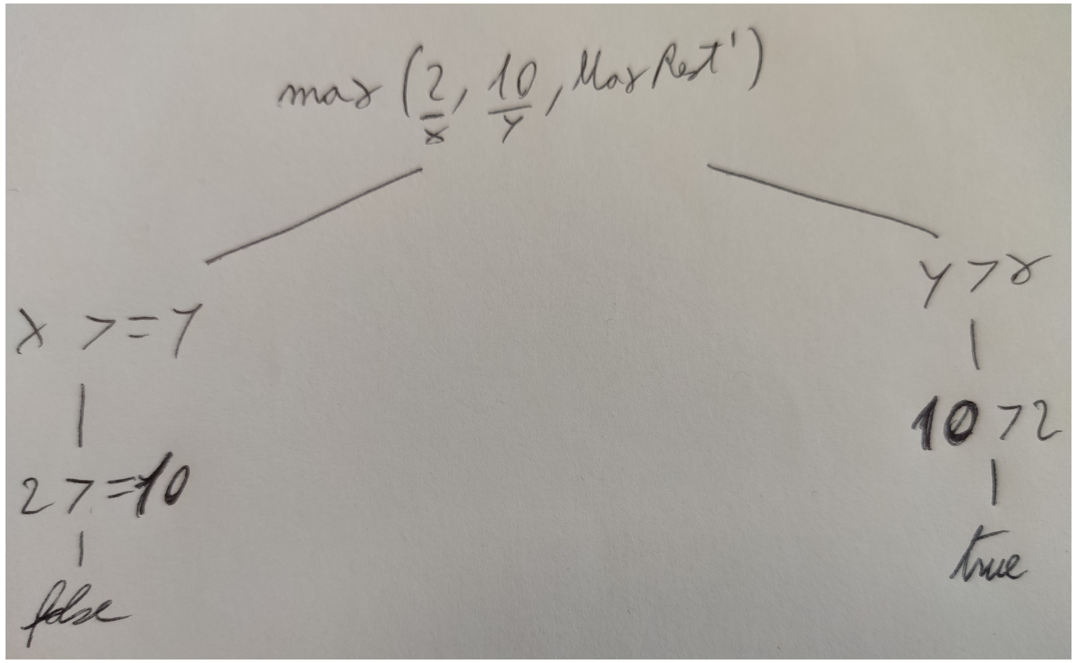
[trace] ?- maxlist([3,7,2,10], Max).

```
Call: (10) maxlist([3, 7, 2, 10], _19236) ? creep
Call: (11) maxlist([7, 2, 10], _20488) ? creep
Call: (12) maxlist([2, 10], _21250) ? creep
Call: (13) maxlist([10], _22012) ? creep
Exit: (13) maxlist([10], 10) ? creep

Call: (13) max(2, 10, _21250) ? creep
Call: (14) 2>=10 ? creep
Fail: (14) 2>=10 ? creep
Redo: (13) max(2, 10, _21250) ? creep
Call: (14) 10>2 ? creep
Exit: (14) 10>2 ? creep
Exit: (13) max(2, 10, 10) ? creep
Exit: (12) maxlist([2, 10], 10) ? creep
```



```
max(2, 10, MaxRest'),
max(7, MaxRest', MaxRest),
max(3, MaxRest, Max)
```

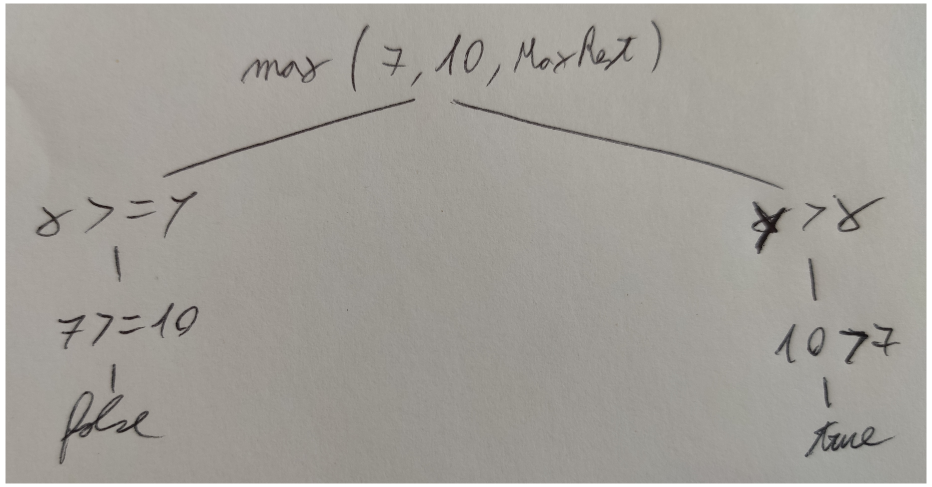
```
max( X, Y, X) :-
  X >= Y.
max( X, Y, Y) :-
  Y > X.
```



Let us define some relations

?- trace.

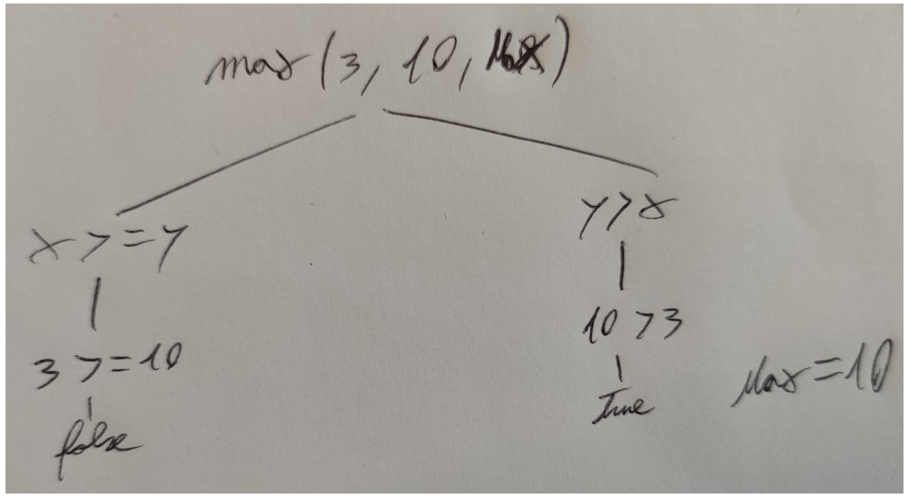
```
[trace]  ?- maxlist([3,7,2,10], Max).

Call: (10) maxlist([3, 7, 2, 10], _19236) ? creep
Call: (11) maxlist([7, 2, 10], _20488) ? creep
Call: (12) maxlist([2, 10], _21250) ? creep
Call: (13) maxlist([10], _22012) ? creep
Exit: (13) maxlist([10], 10) ? creep

Call: (13) max(2, 10, _21250) ? creep
Call: (14) 2>=10 ? creep
Fail: (14) 2>=10 ? creep
Redo: (13) max(2, 10, _21250) ? creep
Call: (14) 10>2 ? creep
Exit: (14) 10>2 ? creep
Exit: (13) max(2, 10, 10) ? creep
Exit: (12) maxlist([2, 10], 10) ? creep

Call: (12) max(7, 10, _20488) ? creep
Call: (13) 7>=10 ? creep
Fail: (13) 7>=10 ? creep
Redo: (12) max(7, 10, _20488) ? creep
Call: (13) 10>7 ? creep
Exit: (13) 10>7 ? creep
Exit: (12) max(7, 10, 10) ? creep
Exit: (11) maxlist([7, 2, 10], 10) ? creep
```



Let us define some relations

```
?- trace.
[trace]  ?- maxlist([3,7,2,10], Max).
```

```
Call: (10) maxlist([3, 7, 2, 10], _19236) ? creep
Call: (11) maxlist([7, 2, 10], _20488) ? creep
Call: (12) maxlist([2, 10], _21250) ? creep
Call: (13) maxlist([10], _22012) ? creep
Exit: (13) maxlist([10], 10) ? creep

Call: (13) max(2, 10, _21250) ? creep
Call: (14) 2>=10 ? creep
Fail: (14) 2>=10 ? creep
Redo: (13) max(2, 10, _21250) ? creep
Call: (14) 10>2 ? creep
Exit: (14) 10>2 ? creep
Exit: (13) max(2, 10, 10) ? creep
Exit: (12) maxlist([2, 10], 10) ? creep
```



```
Call: (11) max(3, 10, _19236) ? creep
Call: (12) 3>=10 ? creep
Fail: (12) 3>=10 ? creep
Redo: (11) max(3, 10, _19236) ? creep
Call: (12) 10>3 ? creep
Exit: (12) 10>3 ? creep
Exit: (11) max(3, 10, 10) ? creep
Exit: (10) maxlist([3, 7, 2, 10], 10) ? creep
Max = 10 .
```

Let us define some relations

```
?- trace.
[trace]  ?- maxlist([3,7,2,10], Max).

Call: (10) maxlist([3, 7, 2, 10], _19236) ? creep
Call: (11) maxlist([7, 2, 10], _20488) ? creep
Call: (12) maxlist([2, 10], _21250) ? creep
Call: (13) maxlist([10], _22012) ? creep
Exit: (13) maxlist([10], 10) ? creep

Call: (13) max(2, 10, _21250) ? creep
Call: (14) 2>=10 ? creep
Fail: (14) 2>=10 ? creep
Redo: (13) max(2, 10, _21250) ? creep
Call: (14) 10>2 ? creep
Exit: (14) 10>2 ? creep
Exit: (13) max(2, 10, 10) ? creep
Exit: (12) maxlist([2, 10], 10) ? creep

Call: (12) max(7, 10, _20488) ? creep
Call: (13) 7>=10 ? creep
Fail: (13) 7>=10 ? creep
Redo: (12) max(7, 10, _20488) ? creep
Call: (13) 10>7 ? creep
Exit: (13) 10>7 ? creep
Exit: (12) max(7, 10, 10) ? creep
Exit: (11) maxlist([7, 2, 10], 10) ? creep

Call: (11) max(3, 10, _19236) ? creep
Call: (12) 3>=10 ? creep
Fail: (12) 3>=10 ? creep
Redo: (11) max(3, 10, _19236) ? creep
Call: (12) 10>3 ? creep
Exit: (12) 10>3 ? creep
Exit: (11) max(3, 10, 10) ? creep
Exit: (10) maxlist([3, 7, 2, 10], 10) ? creep
Max = 10 .
```

Let us define some relations

Define the predicate **ordered(List)**

which is true if **List** is an ordered list of numbers such as

ordered([2, 3, 7, 10]).

Let us define some relations

Define the predicate **ordered(List)**

which is true if **List** is an ordered list of numbers such as

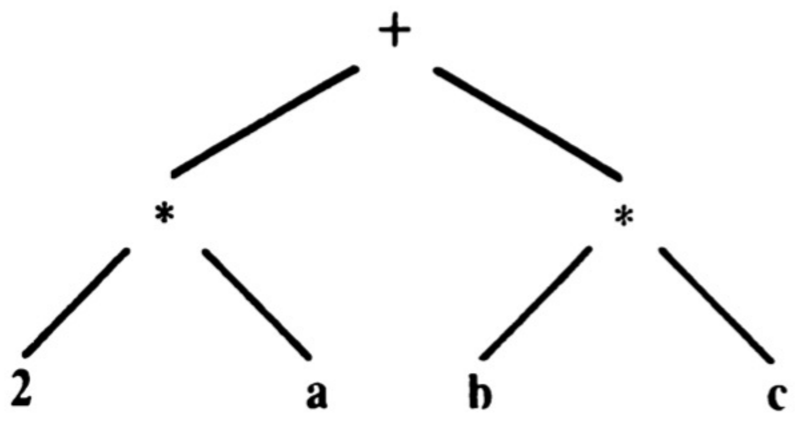
ordered([2, 3, 7, 10]).

```
ordered([X]).            % A single-element list is ordered

ordered([X,Y|Rest]) :-
   X =< Y,
   ordered([Y|Rest]).
```

Operators

2*a + b*c *infix*

+(*(2,a), *(b,c)) In Prolog



Operators

A programmer can define his or her own operators.

peter has information.
floor supports table.

These facts are exactly equivalent to:

has(peter, information).
supports(floor, table).

$$\textbf{:- op(}\ \underbrace{\textbf{600}}_{\text{precedence}}\textbf{,}\ \underbrace{\textbf{xfx}}_{\text{infix}}\textbf{,}\ \underbrace{\textbf{has}}_{\text{operator name}}\textbf{).}$$

Operators

A programmer can define his or her own operators.

peter has information.
floor supports table.

These facts are exactly equivalent to:

has(peter, information).
supports(floor, table).

$$\texttt{:- op(}\ \underbrace{\texttt{600}}_{\text{precedence}}\texttt{,}\ \underbrace{\texttt{xfx}}_{\text{infix}}\texttt{,}\ \underbrace{\texttt{has}}_{\text{operator name}}\texttt{).}$$

the operator denoted by “f” is between two arguments denoted by “x”

Operators

Operators are normally used, as functors, only to combine objects into structures and not to invoke actions on data.

Operator names are atoms. An operator’s precedence must be in some range which depends on the implementation. We can assume a range between 1 and 1200.

Precedence of Argument

If an argument is enclosed in parentheses or it in an unstructured object then its precedence is 0; if an argument is a sctucture then its precedence is equal to the precedence of its principal functor.

Three groups of operator types:

1. infix operators of three types: **xfx** **xfy** **yfx**
2. prefix operators of two types: **fx** **fy**
3. postfix operators of two types: **xf** **yf**

“x” represents an argument whose precedence <u>must be strictly lower</u> than that of the operator.
“y” represents an argument whose precedence <u>is lower or equal to</u> that of the operator.

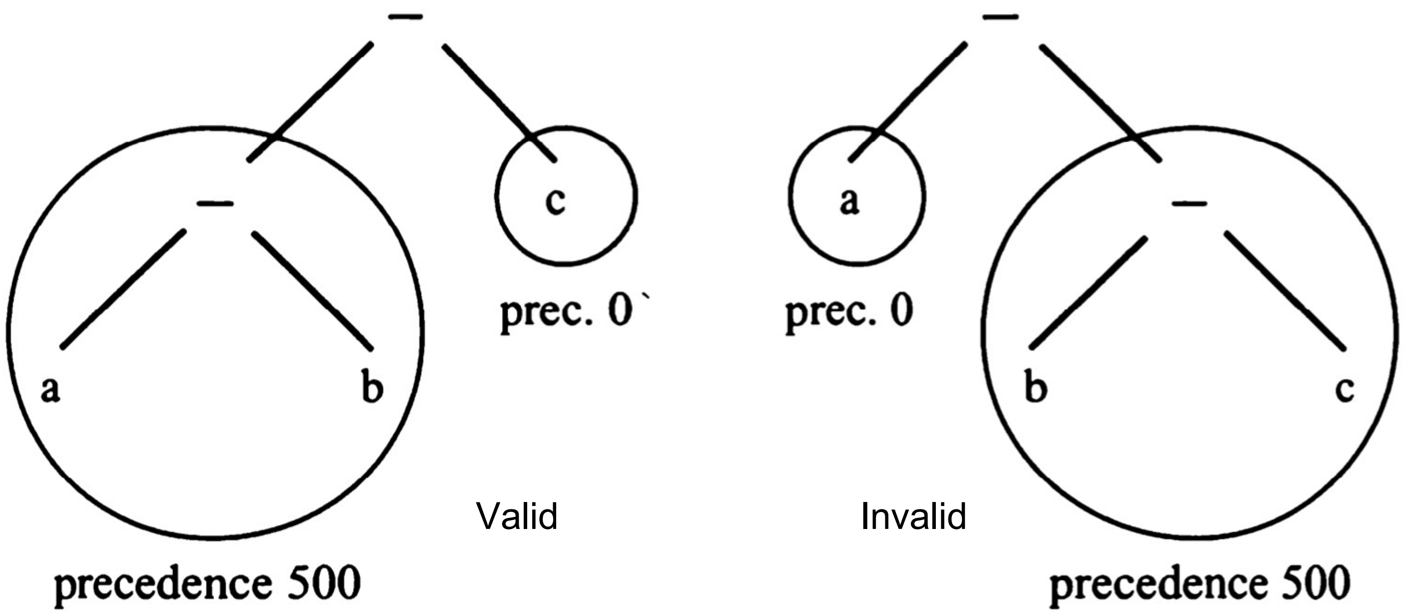
Precedence of Argument

It helps to disambiguate expressions with several operators of the same precedence.

a – b – c is normally understood as **(a – b) – c** and not as **a – (b – c).**

The operator “-” to achieve the normal interpretation has to be defined as **yfx**

assuming that “-” has precedence 500



Precedence of Argument

Another example with the prefix operator **not**:

not not p

is legal if **not** is defined as **fy**; it is illegal if defined as **fx** because the argument to the first **not** is **not p**, which has the same precedence as **not** itself.

In this case the expression has to be written with parentheses:

not(not p)

Some operators in the Prolog system are already defined. What they are and their precedence depends on the specific implementation of Prolog.

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

Precedence of Argument

The use of operators can greatly improve the readability of programs.

If we want to state one of de Morgan’s equivalence theorems written as:

$$\sim(A \;\&\; B) \;<===>\; \sim A \vee \sim B$$

One way is:

equivalence(not(and(A, B)), or(not(A), not(B))).

It is a good programming practice to try to retain as much resemblance as possible between the original problem notation and the notation used in the program

Precedence of Argument

equivalence(not(and(A, B)), or(not(A), not(B))).

We can define operators such as:

```
:- op( 800, xfx, <===>).    % Logical equivalence
:- op( 700, xfy, v).        % Disjunction
:- op( 600, xfy, &).        % Conjunction
:- op( 500, fy, ~).         % Negation
```

And write the de Morgan's theorem as:

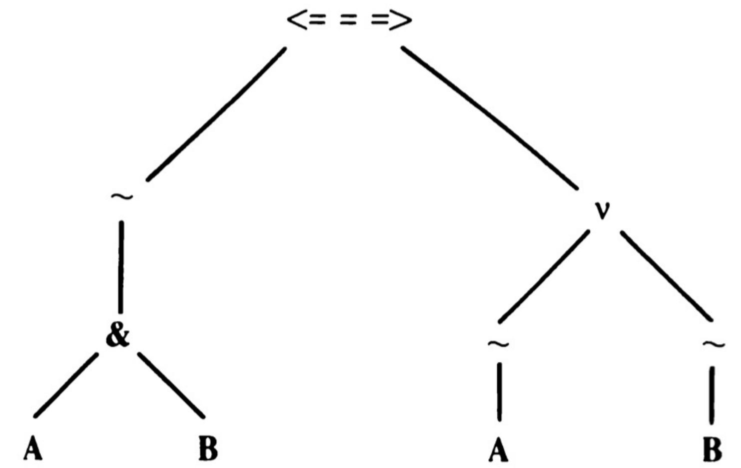
~(A & B) <===> ~A v ~B.

A lot similar to: $\sim(A \,\&\, B) \;<===>\; \sim A \vee \sim B$

Instead of:

equivalence(not(and(A, B)), or(not(A), not(B))).

How can we visualize the tree structure?



Arithmetic

Some of the predefined operators can be used for basic arithmetic operations.

+	addition
−	subtraction
*	multiplication
/	division
**	power
//	integer division
mod	modulo, the remainder of integer division

Try asking Prolog: `?- X = 1 + 2.`

Try asking Prolog: `?- X is 1 + 2.`

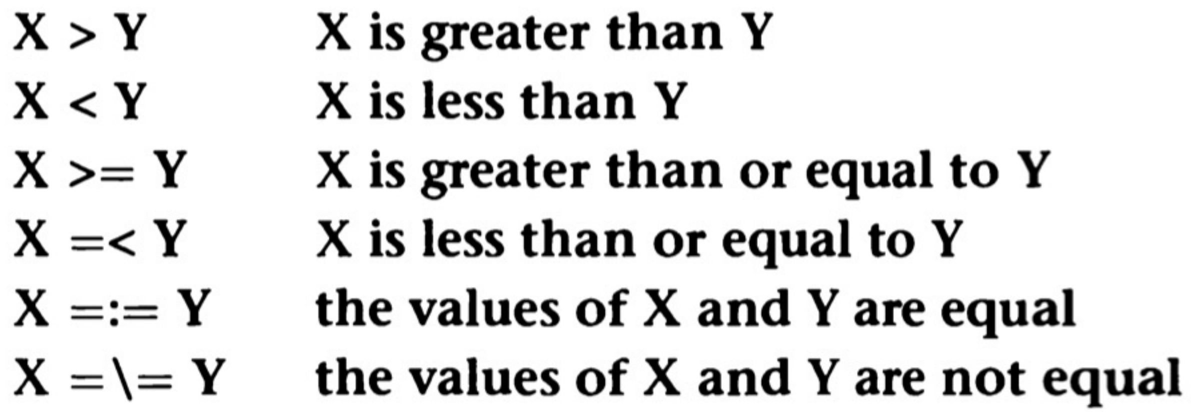
Try asking Prolog:

```
?- X is 5/2,
?- Y is 5//2,
?- Z is 5 mod 2.
```

Try asking Prolog: `?- 277 * 37 > 10000.`

Arithmetic

```
?- 1 + 2 = 2 + 1.
false

?- 1 + 2 =:= 2 + 1.
true

?- 1 + A = B + 2.
A = 2,
B = 1.
```



X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the values of X and Y are equal
X =\= Y	the values of X and Y are not equal

Arithmetic

Suppose we have a relation **born** that relates the names of people with their birth years. We could retrieve the names of people born between 1980 and 1990 inclusive with the followin question:

```
?- born( Name, Year),
   Year >= 1980,
   Year =< 1990.
```

Operators

Assume the operator definitions

```
:- op( 300, xfx, plays).
:- op( 200, xfy, and).
```

Then the following two terms are syntactically legal objects:

Term1 = jimmy plays football and squash
Term2 = susan plays tennis and basketball and volleyball

How are these terms understood by Prolog?
What are their principal functors and what is theirs structure?

Term1 = plays(jimmy, and(football, squash))
Term2 = plays (susan, and(tennis, and(basketball, volleyball)))

Summary

The list is a frequently used structure. It is either empty or consists of a *head* and a *tail* which is alist as well. Prolog provides a special notation for lists.

Common operations on lists, programmed in this chapter, are: list membership, concatenation, adding an item, deleting an item, sublist.

The *operator notation* allows the programmer to tailor the syntax of programs toward particular needs. Using operators the redability of programs can be greatly improved.

Summary

New operators are defined by the directive **op**, standing the name of an operator, its type and precedence.

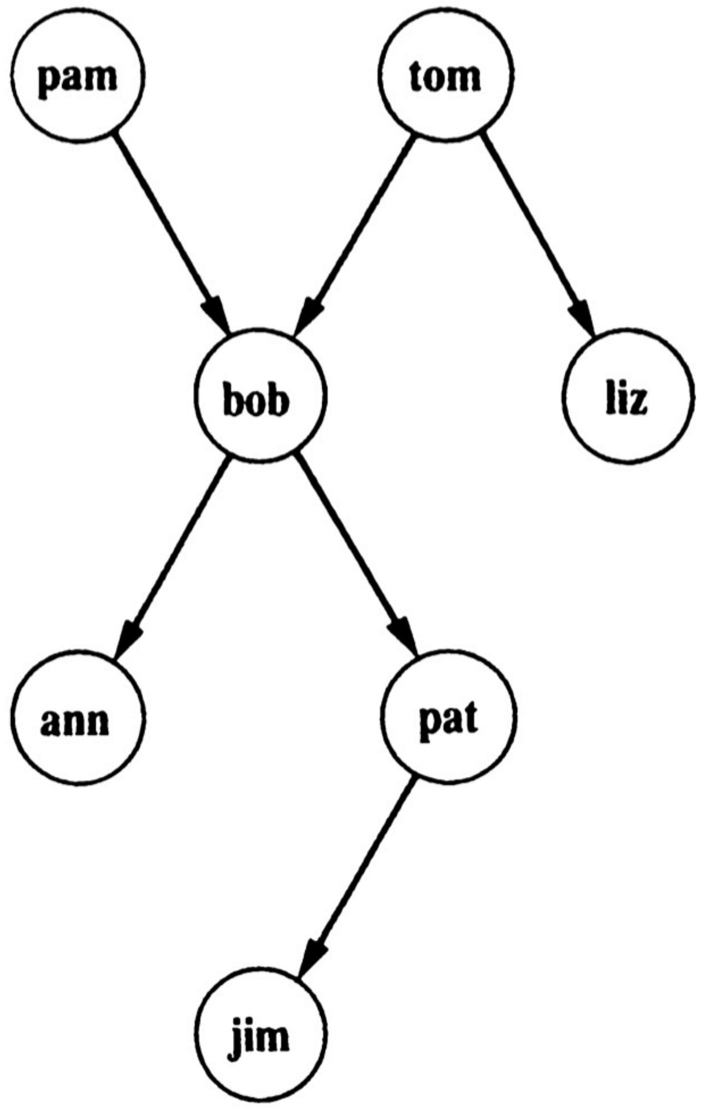
In principle, there is no operation associated with an operator; operators are merely a syntactic device providing an alternative syntax for terms.

Arithmetic is done by built-in procedures. Evaluation of an arithmetic expression is forced by the procedure **is** and by the copmarison predicates <, =<, etc.

Programming Examples

Graph structures are useful abstract representation for many problems.

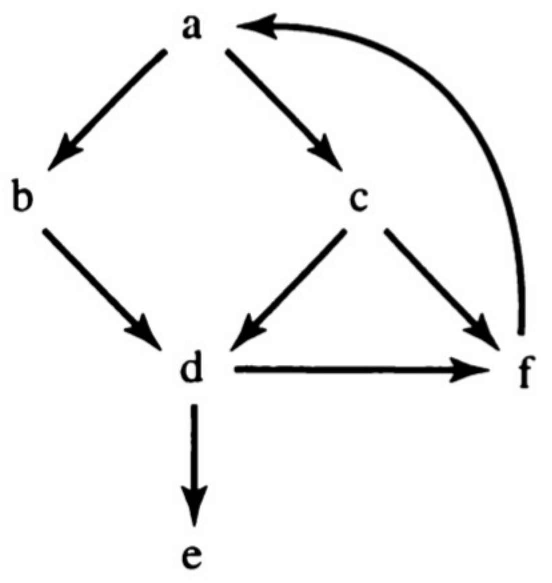
Directed graph



Programming Examples

Graph structures are useful abstract representation for many problems.

Directed graph



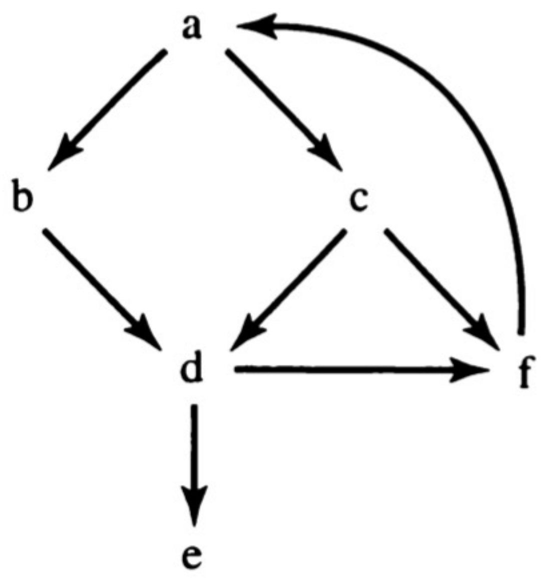
$$\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & & \\
b \arrow[dr] & & c \arrow[dl] \arrow[dr] & \\
 & d \arrow[r] \arrow[d] & & f \arrow[uull, bend right] \\
 & e & &
\end{tikzcd}$$

```
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

Graph structures are useful abstract representation for many problems.

Directed graph



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

Which is the most common type of question concerning a graph?

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

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

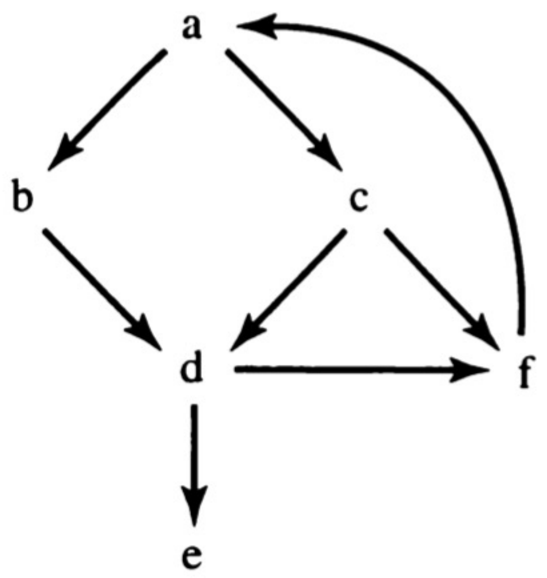
New Hampshire
Vermont
Massachusetts
Maine
North Dakota
Minnesota
Wisconsin
South Dakota
Michigan
New York
Rhode Islan
Connecticu
Pennsylvania
Nebraska
Iowa
Ohio
New Jerse
start
Illinois
Indiana
Delawar
Kansas
Missouri
Virginia
Marylan
Kentucky
West Virgini
North Carolina
Oklahoma
Arkansas
Tennessee
South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

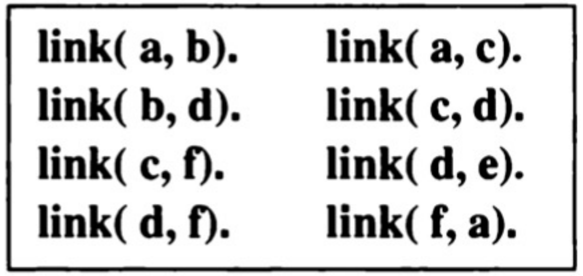
Milgram, S. (1967). The small world problem. *Psychology today*, *2*(1), 60-67.
The average path length for social networks of people in the United States.

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).
```


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



```
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).
```

What are we asking?

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
```

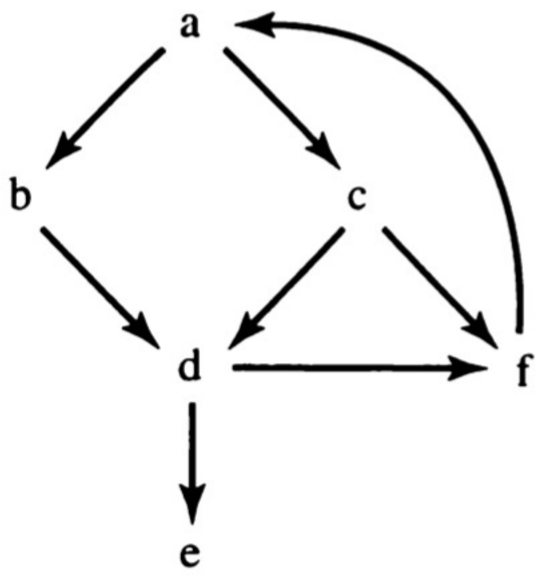
```
?- path(a,X).
```

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

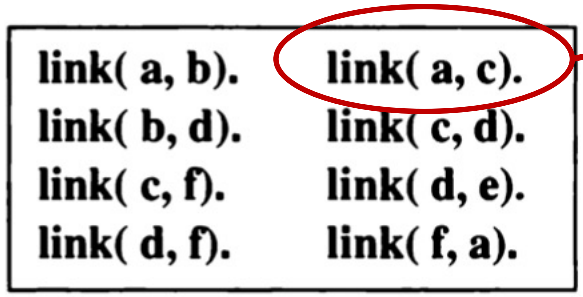
Programming Examples

?- path(a,X).

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



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



```
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 ;
…

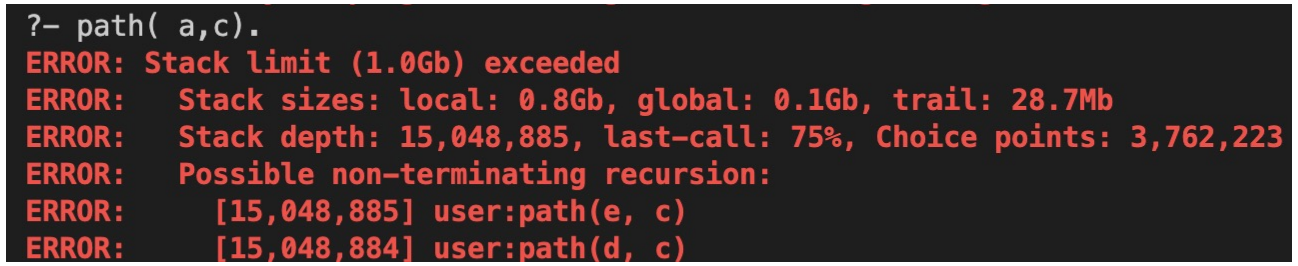
What happens?

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?



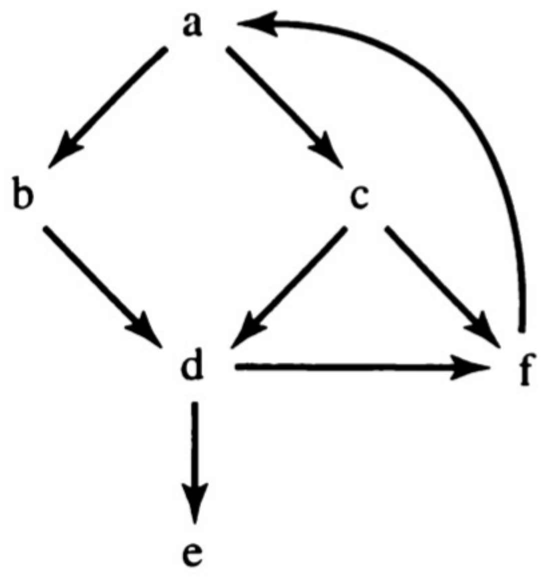
```
?- path( a,c).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.8Gb, global: 0.1Gb, trail: 28.7Mb
ERROR:   Stack depth: 15,048,885, last-call: 75%, Choice points: 3,762,223
ERROR:   Possible non-terminating recursion:
ERROR:     [15,048,885] user:path(e, c)
ERROR:     [15,048,884]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

Programming Examples

Which is the execution trace?

?- path(a,c).

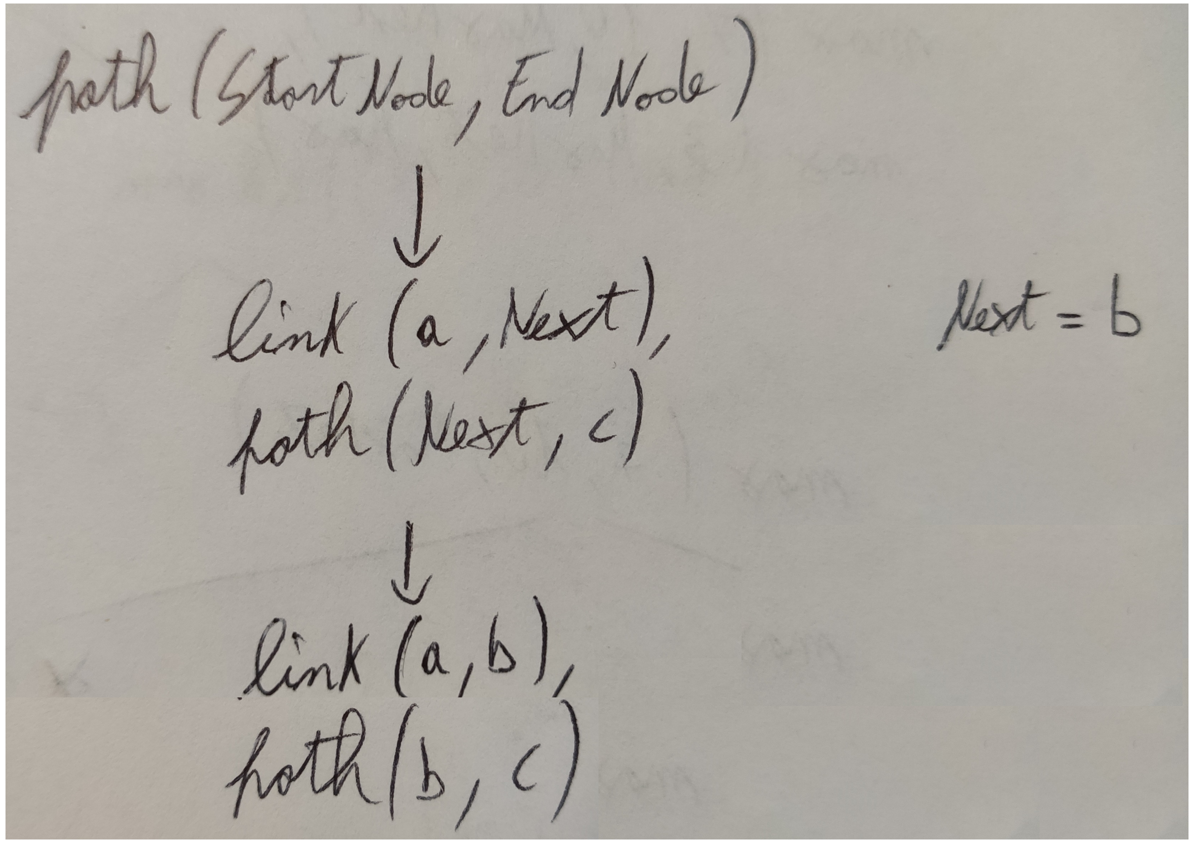
```
link( a,b).
link( a,c).
link( b,d).
link( c,d).
link( c,f).
link( d,e).
link( d,f).
link( f,a).
```



```
path( Node, Node).

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

Programming Examples

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



path(StartNode, EndNode)

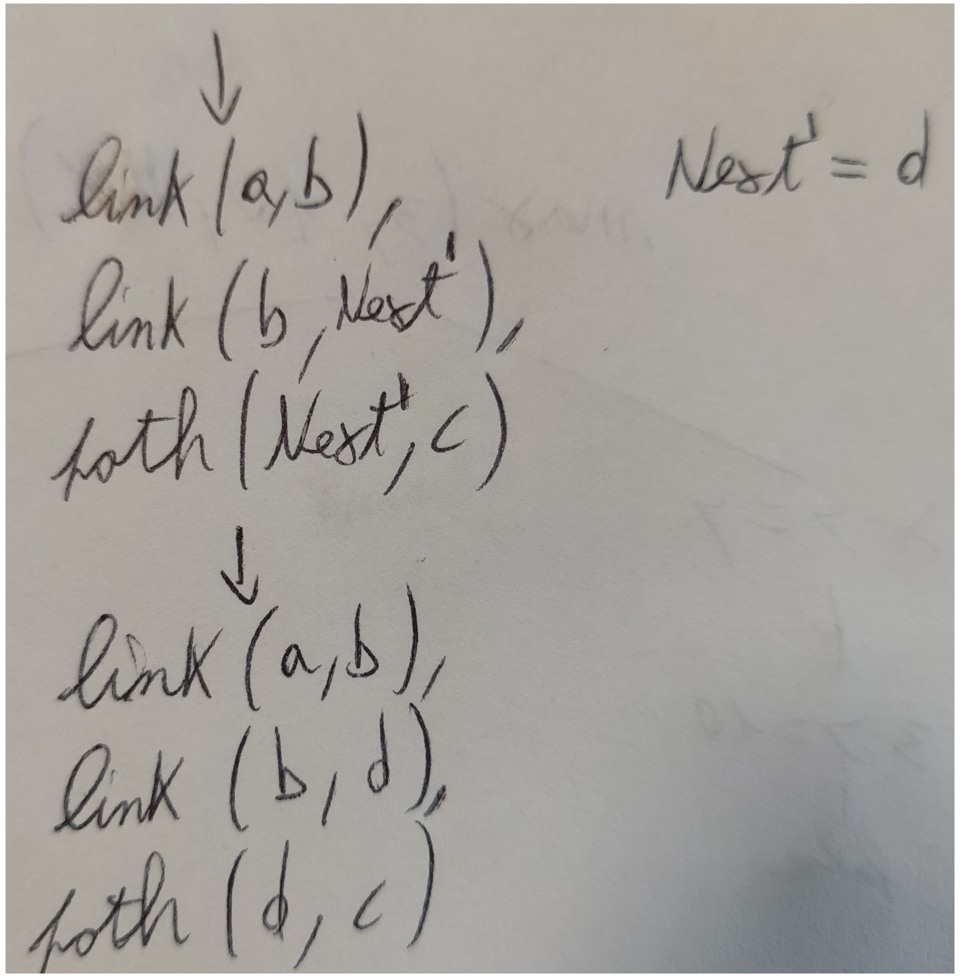
↓

link(a, Next),
path(Next, c)

Next = b

↓

link(a, b),
path(b, c)



↓

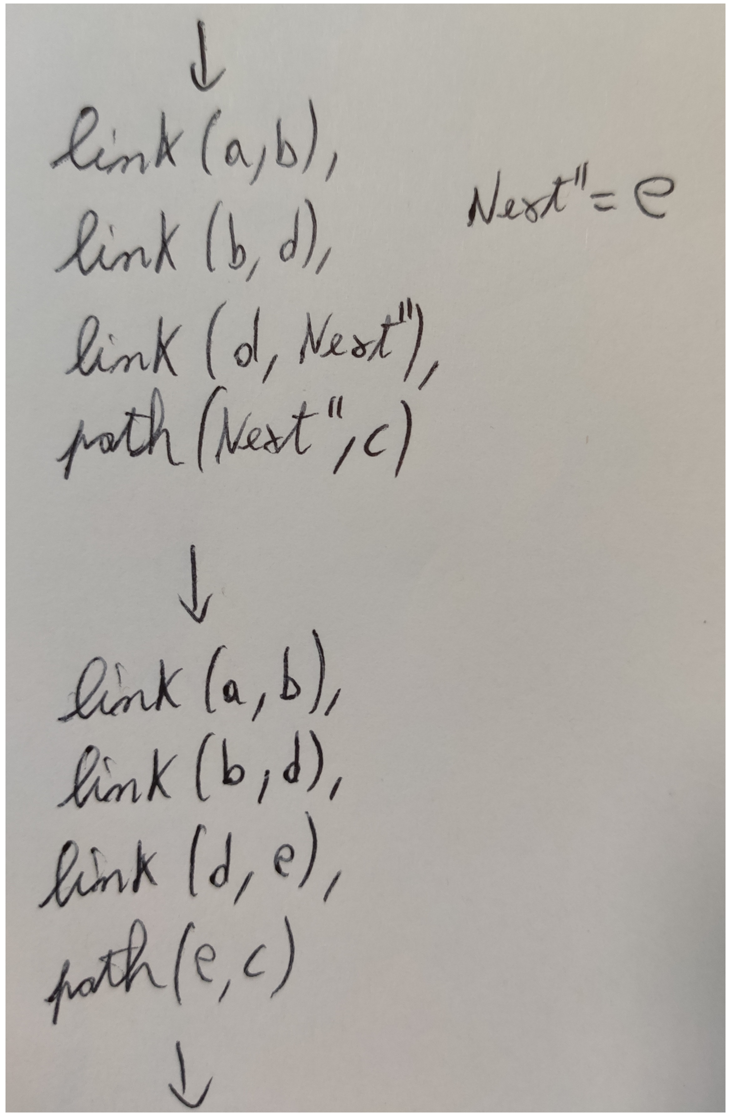
link(a,b),
link(b, Next'),
path(Next', c)

Next' = d

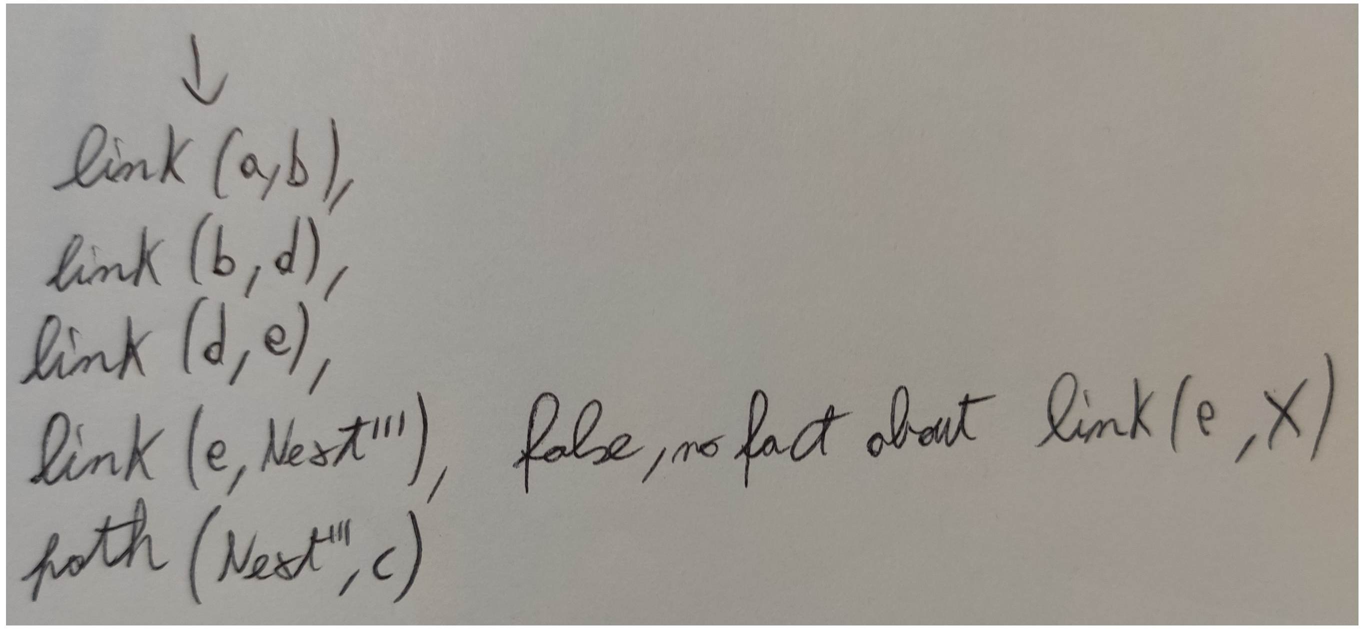
↓

link(a,b),
link(b, d),
path(d, c)

Programming Examples

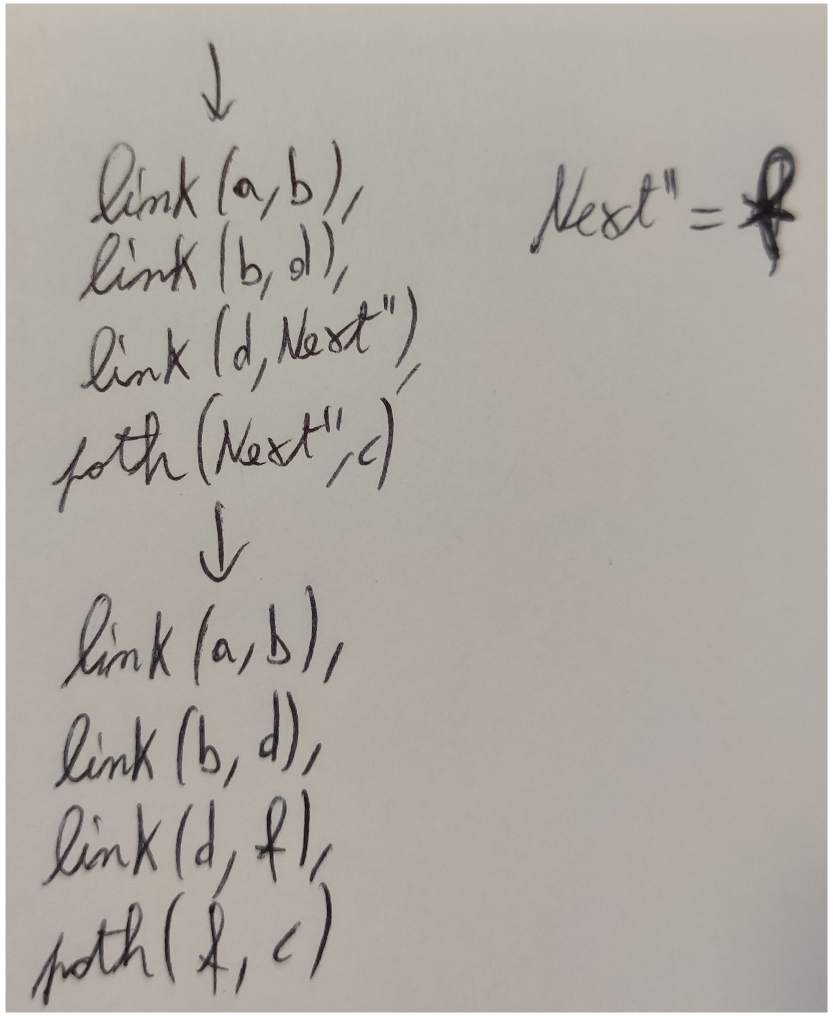


```
↓
link(a,b),
link(b,d),              Next''=e
link(d, Next''),
path(Next'',c)

↓

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



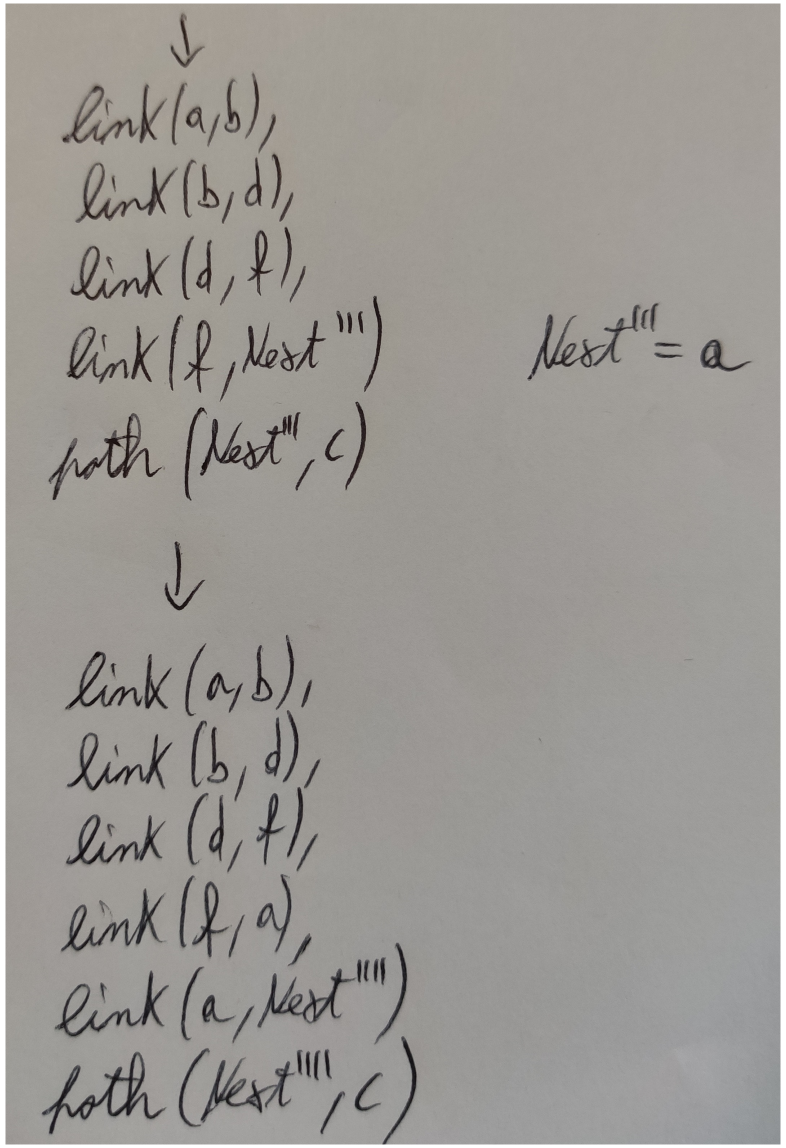
```
↓
link(a,b),
link(b,d),
link(d,e),
link(e, Next'''),   false, no fact about  link(e,X)
path(Next''',c)
```

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.