

2_IntroProlog

October 18, 2022

1 A more systematic introduction to Prolog

1.0.1 Propositions

Prolog programs consist of clauses. A clause is always terminated by a dot (.). Propositions start with a lower case letter or you can use quotes to use (almost) arbitrary strings as propositions. The simplest clauses are facts. Here we define three propositions to be true, for the last two we use quotes:

```
[47]: rains.  
      'I am not wearing a hat'.  
      'The sun is shining'.  
      beach :- fail.
```

```
Previously defined clauses of user:rains/0 were retracted:  
:- dynamic rains/0.
```

```
rains.
```

```
% Asserting clauses for user:rains/0
```

```
Previously defined clauses of user:I am not wearing a hat/0 were retracted:  
:- dynamic'I am not wearing a hat'/0.
```

```
'I am not wearing a hat'.
```

```
% Asserting clauses for user:I am not wearing a hat/0
```

```
Previously defined clauses of user:The sun is shining/0 were retracted:  
:- dynamic'The sun is shining'/0.
```

```
'The sun is shining'.
```

```
% Asserting clauses for user:The sun is shining/0
```

Previously defined clauses of user:beach/0 were retracted:
:- dynamic beach/0.

```
beach :-  
    fail.
```

% Asserting clauses for user:beach/0

We can now ask the Prolog system whether the sun is shining:

```
[ ]: ?- beach.
```

false

```
[49]: ?-'The sun is shining'.
```

true

More complicated clauses make use of the implication operator `:-`. They are also called rules. Logically they stipulate that the left-hand side of the clause must be true if the right-hand side is true. The right-hand side can contain multiple propositions separated by commas. The comma can be read as a logical conjunction (and).

```
[50]: carry_umbrella :- rains, 'I am not wearing a hat'.  
rainbow :- rains, 'The sun is shining'.
```

Previously defined clauses of user:carry_umbrella/0 were retracted:
:- dynamic carry_umbrella/0.

```
carry_umbrella :-  
    rains,  
    'I am not wearing a hat'.
```

% Asserting clauses for user:carry_umbrella/0

Previously defined clauses of user:rainbow/0 were retracted:
:- dynamic rainbow/0.

```
rainbow :-  
    rains,  
    'The sun is shining'.
```

% Asserting clauses for user:rainbow/0

```
[51]: ?- rainbow.
```

true

The corresponding logic formula to the rule for `rainbow` is `rainbow ← rains 'The sun is shining'`

1.0.2 Predicates

Instead of propositions we can also use predicates with arguments within our clauses. The arguments to predicates denote objects for which the predicate is true. Arguments which start with an upper-case letter are logical variables. Below `X` is such a variable and it can stand for any object.

Prolog provides a few built-in predicates like `>` or `=` or `is`.

```
[ ]: ?- 2>3.
```

false

```
[53]: ?- is(X,3+2).
```

X = 5

Let us now define our own predicates. In this case `mother/2` and `grandma/2`. Note: we often use the notation `p/n` to denote the fact that the predicate `p` takes `n` arguments. `n` is called the arity of `p`.

```
[54]: mother(a,b).  
      mother(b,c).  
      grandma(a,c) :- mother(a,b),mother(b,c).
```

Previously defined clauses of `user:mother/2` were retracted:

```
:- dynamic mother/2.
```

```
mother(a, b).
```

```
mother(b, c).
```

```
% Asserting clauses for user:mother/2
```

Previously defined clauses of `user:grandma/2` were retracted:

```
:- dynamic grandma/2.
```

```
grandma(A, B) :-  
    mother(A, C),  
    mother(C, B).
```

```
% Asserting clauses for user:grandma/2
```

You can now ask questions about logical consequences of your logic program. In simple queries you provide all arguments:

```
[55]: ?-grandma(a,c).
```

```
true
```

```
[56]: ?- grandma(a,c) ; mother(c,d).
```

```
true
```

1.1 Logical variables

Variables start with an upper-case letter or an underscore. Variables are called **logical variables** in Prolog: once assigned, their value is immutable and cannot be changed (except upon backtracking).

```
[57]: ?- X=1.
```

```
X = 1
```

Above we have set the logical variable **X** to 1. The scope of the name **X** is a Prolog clause (i.e., a fact or rule or a query). Thus, in the query below we talk about another **X**:

```
[58]: ?- X=2.
```

```
X = 2
```

However, in the same scope we cannot change the value of **X**, once assigned:

```
[ ]: ?- X=1, X=2.
```

```
false
```

```
[60]: ?- X=1, X2 is X+1.
```

```
X = 1,
```

```
X2 = 2
```

Within a clause variables are implicitly universally quantified. Let us now define the **grandma** predicate in a more general fashion:

```
[61]: grandma(X,Y) :- mother(X,Z), mother(Z,Y).
```

Previously defined clauses of `user:grandma/2` were retracted:
`:- dynamic grandma/2.`

```
grandma(a, c) :-  
    mother(a, b),
```

```
mother(b, c).
```

```
% Asserting clauses for user:grandma/2
```

The above clause is equivalent to this logical formula:

```
X,Y,Z . grandma(X,Y) ← mother(X,Z) mother(Z,Y)
```

Let us query the predicate:

```
[62]: ?- grandma(a,X).
```

```
X = c
```

When we have variables in a query, Prolog gives us solutions for variables such that the instantiated predicate calls are logical consequences of your program.

We can find all solutions using the `print_table` command of our Jupyter kernel:

```
[63]: jupyter:print_table(grandma(a,X))
```

```
—  
X  
—  
c  
—
```

```
true
```

Prolog also has a built-in predicate called `findall` which can be used to find all solutions in one go:

```
[64]: ?-findall(X,grandma(a,X),Results).
```

```
Results = [c]
```

1.1.1 Prolog terms and substitutions

Terms represent data values (aka objects). We have that - constants like `a` and `b` are terms - variables like `X` are terms - terms can also be constructed using function symbols

A predicate call takes terms as arguments. E.g. for `grandma(a,X)` we have the term `a` as first argument and the term `X` as second argument.

1.2 Exercise

Let us try exercise 2.1.1 (iii) from the Art of Prolog (<https://mitpress.mit.edu/9780262691635/the-art-of-prolog/>), describing the layout of Figure 2.3 using `left_of/2` and `above/2`.

```
[65]: left_of(bicycle,camera).  
      left_of(pencil,hourglass).
```

```

left_of(hourglass,butterfly).
left_of(butterfly,fish).

above(bicycle,pencil).
above(camera,butterfly).

```

Previously defined clauses of user:left_of/2 were retracted:
:- dynamic left_of/2.

```

left_of(bicycle, camera).
left_of(pencil, hourglass).
left_of(hourglass, butterfly).
left_of(butterfly, fish).

```

% Asserting clauses for user:left_of/2

Previously defined clauses of user:above/2 were retracted:
:- dynamic above/2.

```

above(bicycle, pencil).
above(camera, butterfly).

```

% Asserting clauses for user:above/2

We can use the Jupyter notebook to render the graph. The `print_transition_graph` predicate requires a ternary predicate, so that we can provide the edge labels:

```

[66]: edge(A,above,B) :- above(A,B).
      edge(A,left_of,B) :- left_of(A,B).

```

Previously defined clauses of user:edge/3 were retracted:
:- dynamic edge/3.

```

edge(A, above, B) :-
    above(A, B).
edge(A, left_of, B) :-
    left_of(A, B).

```

% Asserting clauses for user:edge/3

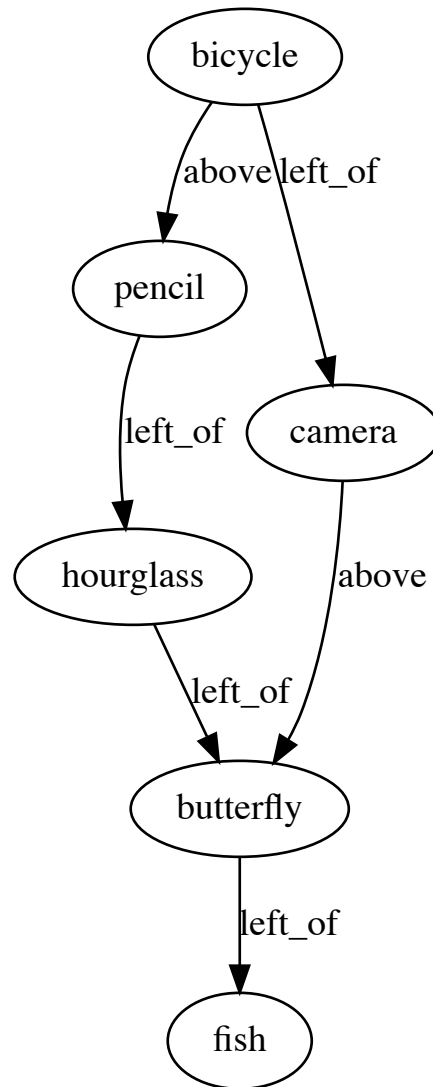
```

[67]: ?- edge(A,B,C).

```

```
A = bicycle,  
B = above,  
C = pencil
```

```
[68]: jupyter:print_transition_graph(edge/3, 1, 3, 2).
```



true

We now define the predicates `right_of` and `below` in terms of the existing predicates:

```
[69]: right_of(X,Y) :- left_of(Y,X).  
      below(X,Y) :- above(Y,X).
```

Previously defined clauses of `user:right_of/2` were retracted:

```
:- dynamic right_of/2.
```

```
right_of(A, B) :-  
    left_of(B, A).
```

```
% Asserting clauses for user:right_of/2
```

Previously defined clauses of user:below/2 were retracted:

```
:- dynamic below/2.
```

```
below(A, B) :-  
    above(B, A).
```

```
% Asserting clauses for user:below/2
```

```
[70]: jupyter:print_table(right_of(X,Y))
```

X	Y
camera	bicycle
hourglass	pencil
butterfly	hourglass
fish	butterfly

true

```
[71]: % next(A,B) :- above(A,B); below(A,B) ; left_of(A,B) ; right_of(A,B).  
next(A,B) :- edge(A,_,B).  
next(A,B) :- edge(B,_,A).
```

Previously defined clauses of user:next/2 were retracted:

```
:- dynamic next/2.
```

```
next(A, B) :-  
    edge(A, _, B).  
next(A, B) :-  
    edge(B, _, A).
```

```
% Asserting clauses for user:next/2
```

```
[72]: jupyter:print_table(next(X,Y))
```

X	Y
bicycle	pencil

X	Y
camera	butterfly
bicycle	camera
pencil	hourglass
hourglass	butterfly
butterfly	fish
pencil	bicycle
butterfly	camera
camera	bicycle
hourglass	pencil
butterfly	hourglass
fish	butterfly

true

1.3 Recursion

Recursion is also allowed in Prolog rules. We now define the simple graph of Figure 2.4 of the Art of Prolog as Prolog facts.

Note that Prolog allows the same predicate name to be used with multiple arities. Above we have defined `edge/3`, below we define `edge/2`. For Prolog these two predicates are different and there is no confusion within the Prolog system. However, for programmers it can be a bit tricky to read code which uses the same predicate name with multiple arities.

```
[73]: edge(a,b). edge(a,c).
      edge(b,d). edge(c,d).
      edge(d,e).
      edge(f,g).
```

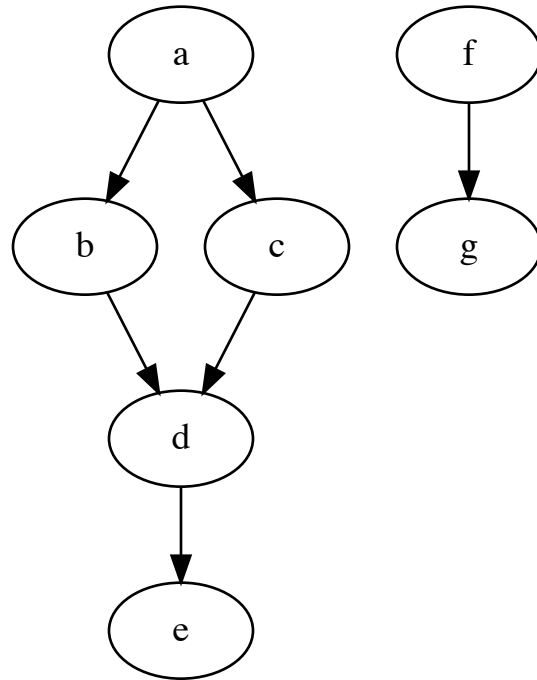
Previously defined clauses of `user:edge/2` were retracted:
`:- dynamic edge/2.`

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

`% Asserting clauses for user:edge/2`

With the underscore we indicate that we are not interested in an argument; it is an anonymous logical variable. Here we use this to find the last element of a list:

```
[74]: jupyter:print_transition_graph(edge/2, 1, 2,0).
```



true

```
[75]: conn(A,A) :- true.
      %conn(X,Y) :- edge(X,Y).
      conn(X,Y) :- edge(X,Z), conn(Z,Y).
```

Previously defined clauses of user:conn/2 were retracted:
:- dynamic conn/2.

```
conn(A, A).
conn(A, B) :-
    edge(A, C),
    conn(C, B).
```

% Asserting clauses for user:conn/2

```
[76]: ?- jupyter:print_table(conn(a,X)).
```

```

_
X
_
a
b
d
e

```

```

—
X
—
c
d
e
—

```

```
true
```

```
[77]: ?- findall(X, conn(a,X),Ls), length(Ls,Len).
```

```
Ls = [a,b,d,e,c,d,e],
```

```
Len = 7
```

Let us now try and define the transitive and reflexive closure of edge.

```
[78]: connected(N,N).
connected(N1,N2) :- edge(N1,Link), connected(Link,N2).
```

Previously defined clauses of user:connected/2 were retracted:
:- dynamic connected/2.

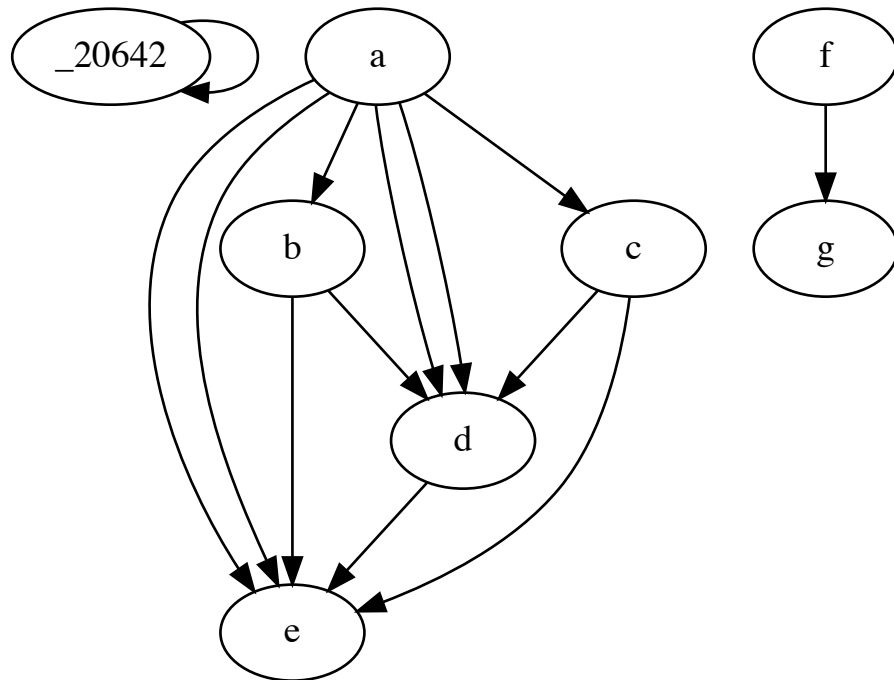
```
connected(A, A).
connected(A, B) :-
    edge(A, C),
    connected(C, B).
```

```
% Asserting clauses for user:connected/2
```

```
[79]: ?- connected(a,X).
```

```
X = a
```

```
[80]: jupyter:print_transition_graph(connected/2, 1, 2,0).
```



true

How should we adapt the definition to only provide the transitive (non-reflexive) closure?

```
[81]: conn1(X,Y) :- edge(X,Y).
      conn1(N1,N2) :- edge(N1,Link), conn1(Link,N2).
```

Previously defined clauses of user:conn1/2 were retracted:

```
:- dynamic conn1/2.
```

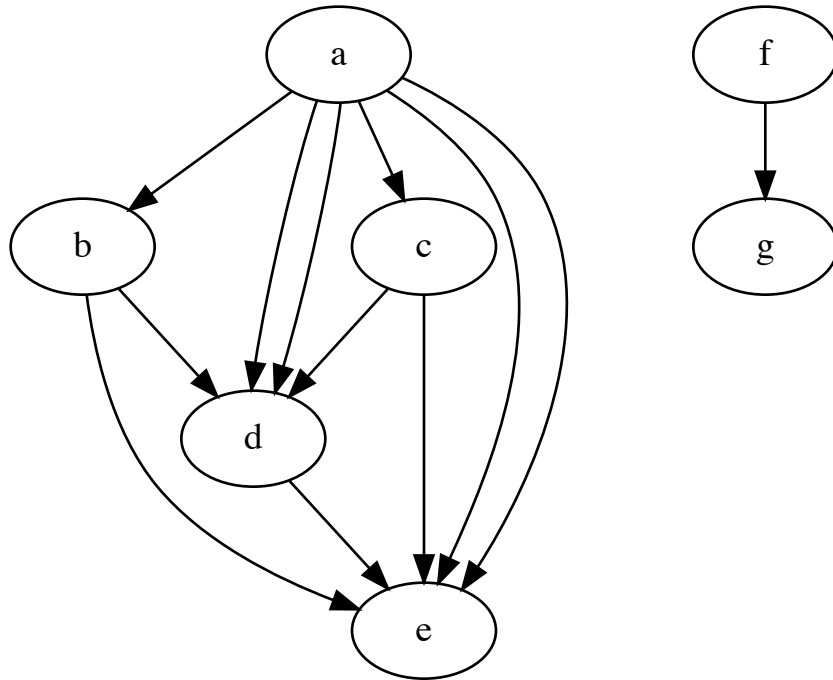
```
conn1(A, B) :-
    edge(A, B).
conn1(A, B) :-
    edge(A, C),
    conn1(C, B).
```

% Asserting clauses for user:conn1/2

```
[82]: ?- conn1(a,X).
```

X = b

```
[83]: jupyter:print_transition_graph(conn1/2, 1, 2,0).
```



true

1.4 Arithmetic

Prolog provides integers and floating point numbers as primitive data structures. With the `is` predicate we can for example compute with those numbers:

```
[84]: ?- X is 2^200.
```

```
X = 1606938044258990275541962092341162602522202993782792835301376
```

```
[85]: ?- X is 1.0+1.
```

```
X = 2.0
```

2 Compound data values

So far we have seen these primitive Prolog data values: - constants (called atoms in Prolog) like `a` and `b` - integers - floats

More complex data values can be wrapped in so-called functors (also called function symbols). Like predicates they have an arity and take terms as arguments. Unlike predicates, they denote a value and not a logical truth value.

This can be confusing to beginners: whether something is a predicate or functor depends on the position in the Prolog file: - top-level symbols in Prolog clauses are predicates - arguments to predicates and functors only contain functors

Functors have many uses in Prolog. They can be used for simple records up to recursive data structures like lists or trees.

Below we first use the functor `employee/2` as a simple record.

```
[86]: construct(Name,Department,employee(Name,Department)).  
  
get_name(employee(Name,_),Name).  
get_dept(employee(_,Dept),Dept).
```

Previously defined clauses of `user:construct/3` were retracted:

```
:- dynamic construct/3.
```

```
construct(A, B, employee(A, B)).
```

```
% Asserting clauses for user:construct/3
```

Previously defined clauses of `user:get_name/2` were retracted:

```
:- dynamic get_name/2.
```

```
get_name(employee(A, _), A).
```

```
% Asserting clauses for user:get_name/2
```

Previously defined clauses of `user:get_dept/2` were retracted:

```
:- dynamic get_dept/2.
```

```
get_dept(employee(_, A), A).
```

```
% Asserting clauses for user:get_dept/2
```

```
[87]: ?- construct(a,cs,E1), construct(b,cs,E2), get_name(E1,N1), get_dept(E2,D2).
```

```
E1 = employee(a,cs),
```

```
E2 = employee(b,cs),
```

```
N1 = a,
```

```
D2 = cs
```

The arguments to a functor can in turn also make use of a functor.

One could thus for example represent a list in Prolog by using a functor `cons/2` to denote a non-empty list and `nil/0` to denote an empty list. Note that a functor of arity 0 is simply a constant (aka atom in Prolog). So a list of length two with a and b as elements is represented as follows:

```
[88]: ?- Mylist = cons(a,cons(b,nil)).
```

```
Mylist = cons(a,cons(b,nil))
```

Let us now try and define some useful predicates for our data type: - `is_empty/1` to check if something is the empty list - `is_list/1` to check if something is a list - `head/1` to get the first element of a list - `element_of/2` to check if something is an element of a list - `last/1` to get the last element of a list

```
[89]: is_empty(nil) :- true.
```

Previously defined clauses of `user:is_empty/1` were retracted:

```
:- dynamic is_empty/1.
```

```
is_empty(nil).
```

```
% Asserting clauses for user:is_empty/1
```

This should succeed:

```
[90]: ?- is_empty(nil).
```

```
true
```

```
[ ]: ?- is_empty(cons(a,nil)).
```

```
false
```

Let us now define `is_list0` (`is_list` is predefined):

```
[92]: is_list0(nil).
is_list0(cons(_,B)) :- is_list0(B).

is_non_empty_list(cons(_,B)) :- is_list0(B).
```

```
% Asserting clauses for user:is_list0/1
```

```
% Asserting clauses for user:is_non_empty_list/1
```

```
[93]: ?-is_list0(cons(employee(a,cs),cons(b,nil))).
```

```
true
```

```
[94]: head(First,cons(First,_)) :- true.
```

```
% Asserting clauses for user:head/2
```

```
[95]: ?- head(X,cons(employe(a,b),cons(b,nil))).
```

```
X = employe(a,b)
```

```
[96]: element_of(First,cons(First,_)).  
      element_of(H,cons(_,T)) :- element_of(H,T).
```

```
% Asserting clauses for user:element_of/2
```

```
[97]: ?- element_of(c,cons(a,cons(b,Y))).
```

```
Y = cons(c,_18628)
```

```
[98]: jupyter:retry.
```

```
% Retrying goal: element_of(c,cons(a,cons(b,Y)))
```

```
Y = cons(_18626,cons(c,_18634))
```

```
[99]: ?- element_of(First,cons(a,nil))
```

```
First = a
```

```
[100]: jupyter:print_table(element_of(X,cons(a,cons(b,nil))))
```

```
—  
X  
—  
a  
b  
—
```

```
true
```

```
[101]: last0(X,cons(X,nil)).  
       last0(X,cons(_,Y)) :- last0(X,Y).
```

```
% Asserting clauses for user:last0/2
```

```
[102]: ?- last0(X,cons(a,cons(b,nil))).
```

```
X = b
```


2.1 Trees

As a quick example let us represent binary trees using compound Prolog terms. For this we use a ternary functor `tree/3`. It has three arguments: - the left sub-tree - the information at the root of the tree - the right sub-tree We also need the empty tree, which we represent by `nil`.

```
[103]: ?- Mytree = tree( tree(nil,a,nil), b, tree(nil,c,tree(nil,d,nil))).
```

```
Mytree = tree(tree(nil,a,nil),b,tree(nil,c,tree(nil,d,nil)))
```

```
[104]: revtree(nil,nil).
revtree(tree(L,Info,R),tree(RR,Info,RL)) :- revtree(L,RL), revtree(R,RR).
```

```
% Asserting clauses for user:revtree/2
```

```
[105]: ?- Mytree = tree( tree(nil,a,nil), b, tree(nil,c,tree(nil,d,nil))),
      revtree(Mytree,Result).
```

```
Mytree = tree(tree(nil,a,nil),b,tree(nil,c,tree(nil,d,nil))),
```

```
Result = tree(tree(tree(nil,d,nil),c,nil),b,tree(nil,a,nil))
```

2.2 Optional Appendix: Visualising data values as trees

Below we try to use the Jupyter graph visualisation to represent data values in a tree-like fashion.

```
[106]: :- use_module(library(lists)).
```

We define a subtree relation, using the `=..` built-in predicate, which deconstructs a term by generating a list consisting of the function symbol and all its arguments:

```
[107]: ?- tree(nil,a,nil) =.. List.
```

```
List = [tree,nil,a,nil]
```

We can now define a subtree relation:

```
[116]: subtree(Term,Nr,SubTerm) :- Term =.. [_|List], nth1(Nr,List,SubTerm).
```

```
% The Prolog server was restarted
```

```
% Asserting clauses for user:subtree/3
```

```
[117]: ?- Mytree = tree( tree(nil,a,nil), b, tree(nil,c,tree(nil,d,nil))),
      subtree(Mytree,Nr,SubTerm).
```

```
Mytree = tree(tree(nil,a,nil),b,tree(nil,c,tree(nil,d,nil))),
Nr = 1,
SubTerm = tree(nil,a,nil)
```

For the Jupyter graph visualisation we also need to restrict this relation and define a set of terms of interest. Indeed, otherwise there are infinitely many terms.

For this we define the transitive and reflexive closure of the subtree relation and only consider subtrees of a given starting term (here `tree(tree(nil,a,nil), b, tree(nil,c,tree(nil,d,nil)))`).

```
[118]: rec_subtree(Term,Sub) :- Term = Sub.
rec_subtree(Term,Sub) :- subtree(Term,_,X), rec_subtree(X,Sub).

of_interest(Term) :- rec_subtree(tree( tree(nil,a,nil), b,
    ↳tree(nil,c,tree(nil,d,nil))),Term).

subt(Term,Nr,SubTerm) :-
    of_interest(Term), % only consider subterms of the above term as nodes
    subtree(Term,Nr,SubTerm).
```

```
% Asserting clauses for user:rec_subtree/2
```

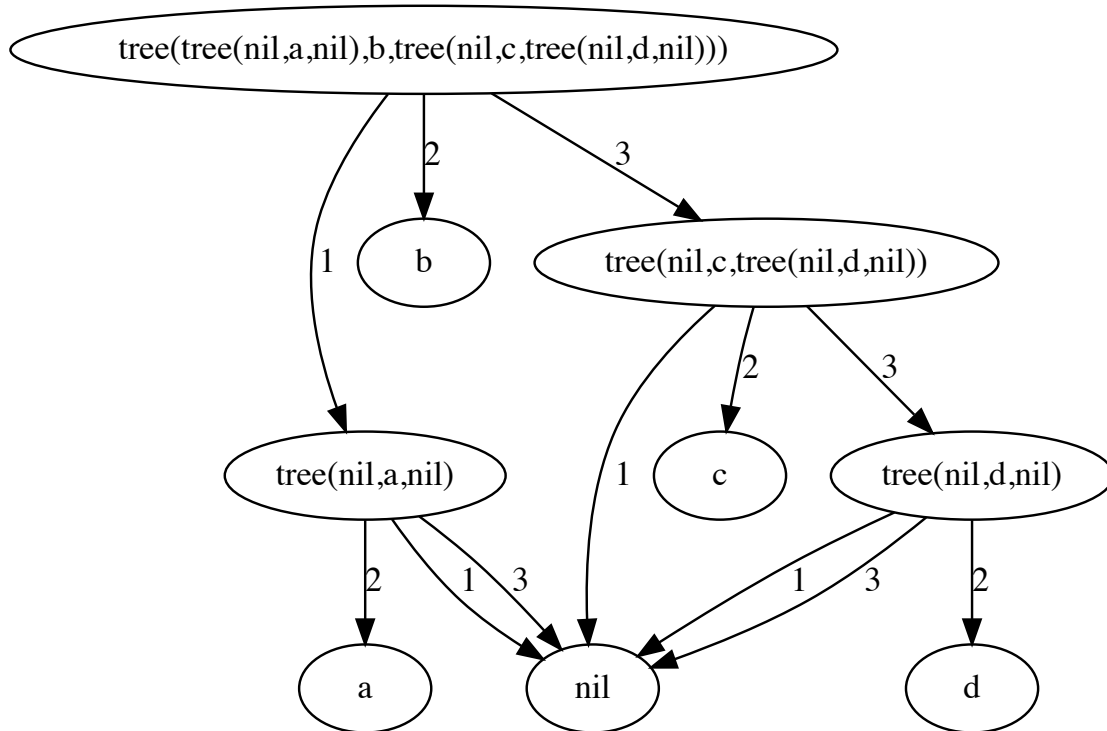
```
% Asserting clauses for user:of_interest/1
```

```
% Asserting clauses for user:subt/3
```

```
[119]: ?- Mytree = tree( tree(nil,a,nil), b, tree(nil,c,tree(nil,d,nil))),
    subtree(Mytree,Nr,SubTerm).
```

```
Mytree = tree(tree(nil,a,nil),b,tree(nil,c,tree(nil,d,nil))),
Nr = 1,
SubTerm = tree(nil,a,nil)
```

```
[120]: jupyter:print_transition_graph(subt/3, 1, 3,2).
```



true

[]: