Meta Programming (8A)

Young W. Lim
3/10/14

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice/OpenOffice.

Meta Programming (8A)

Clause in Prolog

A **clause** in Prolog is a unit of information in a Prolog program ending with a full stop (".").

a fact

```
likes(aa, bb).
food(cc).
```

a rule

```
eats(X, Y) :- likes(X, Y), food(Y).
```

a query

```
?- eats(ee, ff).
```

a procedure.
a group of clauses about the same relation

Clause/2 Predicate

clause(:Head, ?Body)

true if
Head can be unified with a clause head and
Body with the corresponding clause body.

Gives alternative clauses on backtracking.

for facts, Body is unified with the atom true.

Normally clause/2 is used to find clause definitions for a predicate, but it can also be used to find clause heads for some body template.

Meta-interpreter

```
/* true leaf */
clause_tree(true) :- !.

/* search each branch */
clause_tree((G, R)) :-
   !,
   clause_tree(G),
   clause_tree(R).

/* grow branches */
clause_tree(G) :-
   clause(G, Body),
   clause_tree(Body).
```

```
member(X, [X|_]).

member(X, [_|R]) :- member(X,R).
```

2 clauses : a procedure

consider this program as input data for other programs.

```
?- clause_tree(member(X, [a,b,c])).
X = a ;
X = b ;
X = c ;
no
```

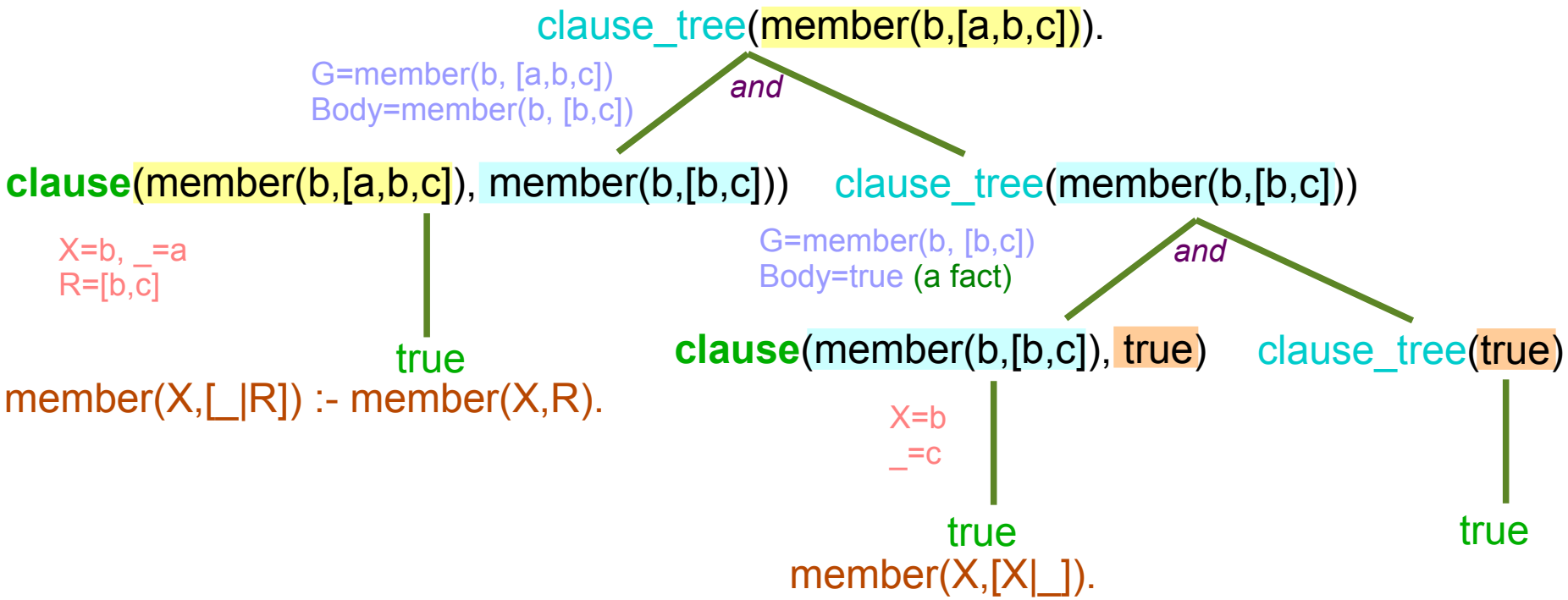
clause(G, Body),

member(X, [X|_])	true
member(X, [_|R])	member(X,R)

Meta-interpreter

```
clause_tree(true)       :-  !.
clause_tree((G, R))     :-  !,   clause_tree(G),   clause_tree(R).
clause_tree(G)          :-  clause(G, Body),       clause_tree(Body).
```

```
member(X,[X|_]).
member(X,[_|R]) :- member(X,R).
```



Evaluation

```
clause_tree(true) :- !.

clause_tree((G,R)) :-
   !,
   clause_tree(G),
   clause_tree(R).

clause_tree(G) :-                          or
   ( predicate_property(G,built_in) ;
     predicate_property(G,compiled) ),
   call(G).              %% let Prolog do it

clause_tree(G) :-
   clause(G,Body),
   clause_tree(Body).
```

predicate_property(G,built_in)

if the goal G is built_in (e.g., arithmetic)

; *or*

predicate_property(G,compiled)

if the goal G is compiled into memory

call(G)

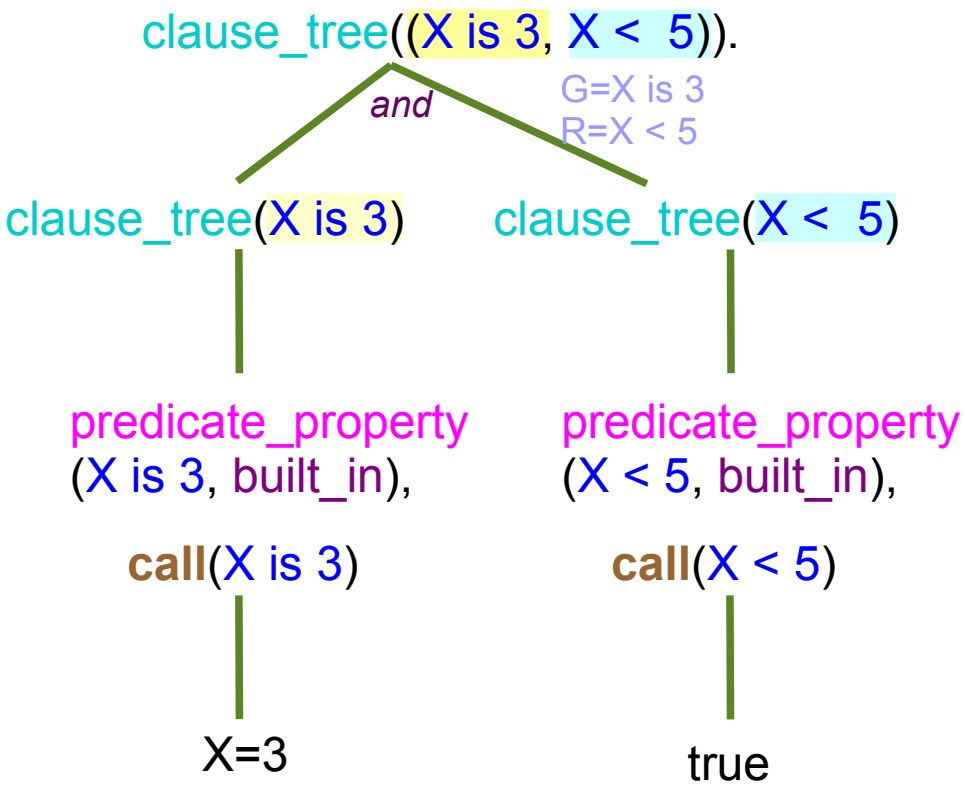
then let Prolog evaluate the goal G

Evaluation

```
clause_tree(true) :- !.

clause_tree((G,R)) :-
   !,
   clause_tree(G),
   clause_tree(R).

clause_tree(G) :-                          or
   ( predicate_property(G,built_in) ;
     predicate_property(G,compiled) ),
   call(G).                %% let Prolog do it

clause_tree(G) :-
   clause(G,Body),
   clause_tree(Body).
```

```
predicate_property(X is 3, built_in)

predicate_property(X < 5, built_in)
```

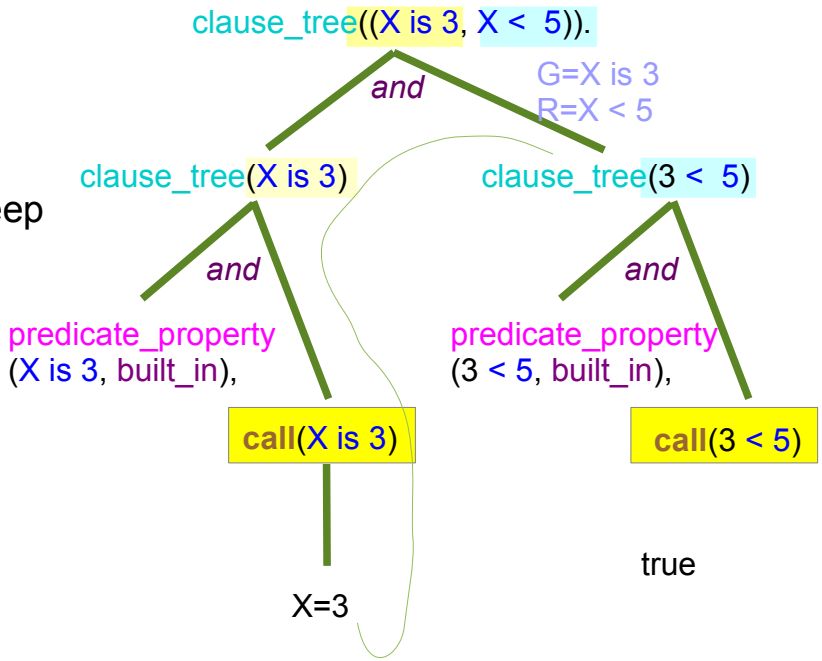
```
?- clause_tree((X is 3, X <  5)).
X = 3
```



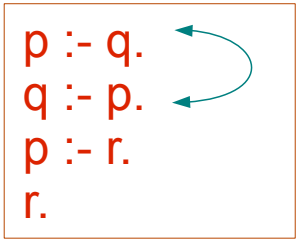
Evaluation

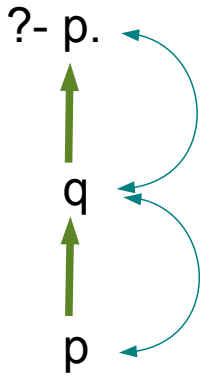
```
clause_tree(true) :- !.
clause_tree((G,R)) :-   !, clause_tree(G), clause_tree(R).
clause_tree(G) :- ( predicate_property(G,built_in) ;  predicate_property(G,compiled) ),
                  call(G).              %% let Prolog do it
clause_tree(G) :-    clause(G,Body), clause_tree(Body).
```

```
[trace]  ?- clause_tree((X is 3, X <  5)).

   Call: (6) clause_tree((_G336 is 3, _G336<5)) ? creep
   Call: (7)  clause_tree(_G336 is 3) ? creep
^  Call: (8) predicate_property(_G336 is 3, built_in) ? creep
^  Exit: (8) predicate_property(user: (_G336 is 3), built_in) ? creep
   Call: (8) _G336 is 3 ? creep
   Exit: (8) 3 is 3 ? creep
   Exit: (7)  clause_tree(3 is 3) ? creep
   Call: (7)  clause_tree(3<5) ? creep
^  Call: (8) predicate_property(3<5, built_in) ? creep
^  Exit: (8) predicate_property(user: (3<5), built_in) ? creep
   Call: (8) 3<5 ? creep
   Exit: (8) 3<5 ? creep
   Exit: (7) clause_tree(3<5) ? creep
   Exit: (6) clause_tree((3 is 3, 3<5)) ? creep
X = 3 .
```



Detecting Loops





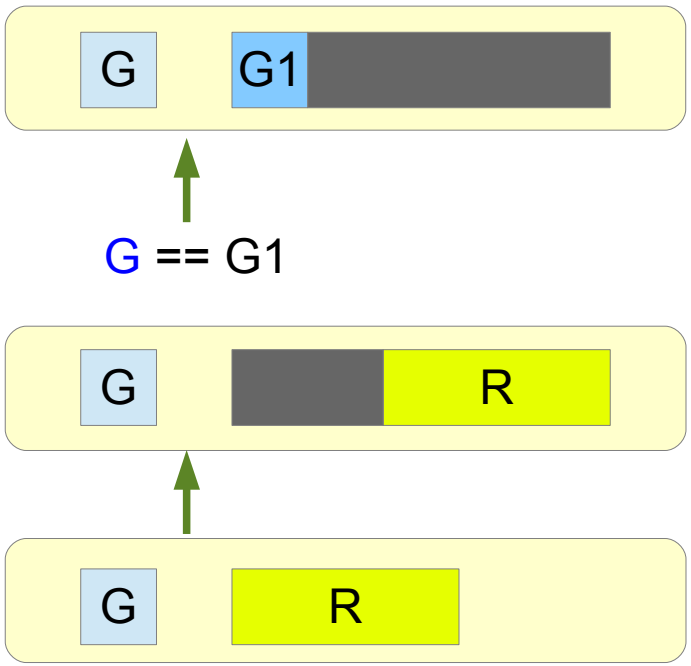
an example of the
incompleteness of Prolog

For the efficiency, Prolog does
not detect any loops

```
clause_tree(true, _) :- !.

clause_tree((G,R), Trail) :-
   !,
   clause_tree(G, Trail),
   clause_tree(R, Trail).

clause_tree(G, Trail) :-
   loop_detect(G,Trail),
   !,
   fail.

clause_tree(G, Trail) :-
   clause(G,Body),
   clause_tree(Body, [G|Trail] ).

loop_detect(G, [G1|_]) :- G == G1.
loop_detect(G, [_|R])  :- loop_detect(G,R).
```

loop_detect

```
loop_detect(G, [G1|_]) :-
      G == G1.

loop_detect(G, [_|R])  :-
      loop_detect(G,R).
```




```
clause_tree(true, _) :- !.

clause_tree((G,R), Trail) :-
   !,
   clause_tree(G, Trail),
   clause_tree(R, Trail).

clause_tree(G, Trail) :-
   loop_detect(G,Trail),
   !,
   fail.

clause_tree(G, Trail) :-
   clause(G,Body),
   clause_tree(Body, [G|Trail] ).

loop_detect(G, [G1|_]) :- G == G1.
loop_detect(G, [_|R])  :- loop_detect(G,R).
```

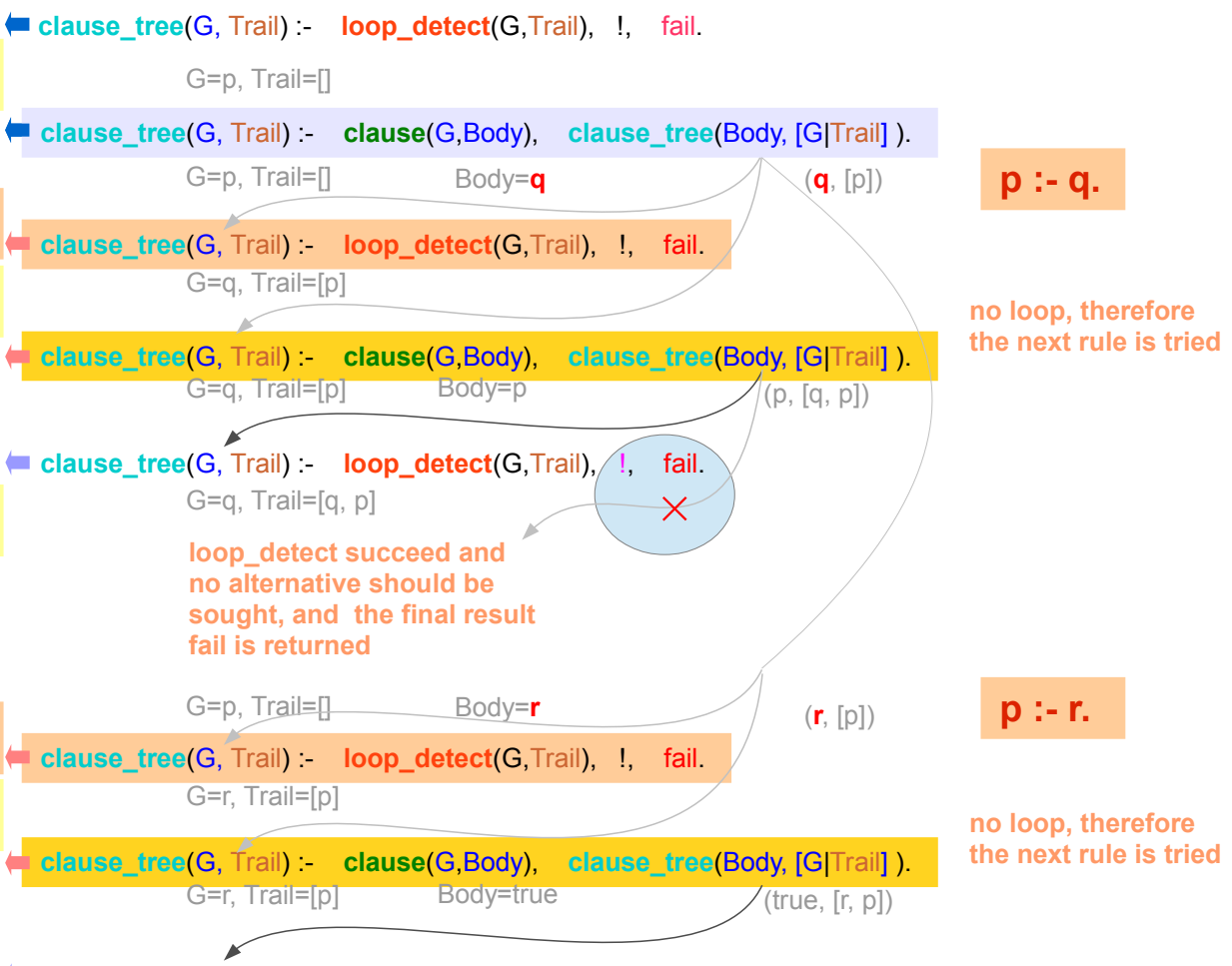
Detecting Loops

```
[trace]  ?- clause_tree(p, []).
   Call: (6) clause_tree(p, []) ? creep
   Call: (7) loop_detect(p, []) ? creep      G1=[]   (G, [G1|_])
   Fail: (7) loop_detect(p, []) ? creep
   Redo: (6) clause_tree(p, []) ? creep
^  Call: (7) clause(p, _G418) ? creep
^  Exit: (7) clause(p, q) ? creep
   Call: (7) clause_tree(q, [p]) ? creep
   Call: (8) loop_detect(q, [p]) ? creep     (G, [G1|_])
   Call: (9) q==p ? creep                    G1=p
   Fail: (9) q==p ? creep
   Redo: (8) loop_detect(q, [p]) ? creep     R=[]    (G, [_|R])
   Call: (9) loop_detect(q, []) ? creep      G1=[]   (G, [G1|_])
   Fail: (9) loop_detect(q, []) ? creep
   Fail: (8) loop_detect(q, [p]) ? creep
   Redo: (7) clause_tree(q, [p]) ? creep
^  Call: (8) clause(q, _G424) ? creep
^  Exit: (8) clause(q, p) ? creep
   Call: (8) clause_tree(p, [q, p]) ? creep
   Call: (9) loop_detect(p, [q, p]) ? creep  (G, [G1|_])
   Call: (10) p==q ? creep                   G1=q
   Fail: (10) p==q ? creep                   R=[p]
   Redo: (9) loop_detect(p, [q, p]) ? creep  (G, [_|R])
   Call: (10) loop_detect(p, [p]) ? creep    G1=p    (G, [G1|_])
   Call: (11) p==p ? creep
   Exit: (11) p==p ? creep
   Exit: (10) loop_detect(p, [p]) ? creep
   Exit: (9) loop_detect(p, [q, p]) ? creep
   Call: (9) fail ? creep
   Fail: (9) fail ? creep
   Fail: (8) clause_tree(p, [q, p]) ? creep
   Fail: (7) clause_tree(q, [p]) ? creep
^  Exit: (7) clause(p, r) ? creep
   Call: (7) clause_tree(r, [p]) ? creep
   Call: (8) loop_detect(r, [p]) ? creep     G1=p    (G, [G1|_])
   Call: (9) r==p ? creep
   Fail: (9) r==p ? creep
   Redo: (8) loop_detect(r, [p]) ? creep     R=[]    (G, [_|R])
   Call: (9) loop_detect(r, []) ? creep      G1=[]   (G, [G1|_])
   Fail: (9) loop_detect(r, []) ? creep
   Fail: (8) loop_detect(r, [p]) ? creep
   Redo: (7) clause_tree(r, [p]) ? creep
^  Call: (8) clause(r, _G424) ? creep
^  Exit: (8) clause(r, true) ? creep
   Call: (8) clause_tree(true, [r, p]) ? creep
   Exit: (8) clause_tree(true, [r, p]) ? creep
   Exit: (7) clause_tree(r, [p]) ? creep
   Exit: (6) clause_tree(p, []) ? creep
true.
```

```
loop_detect(G, [G1|_]) :-  G == G1.

loop_detect(G, [_|R])  :-  loop_detect(G,R).
```

Detecting Loops

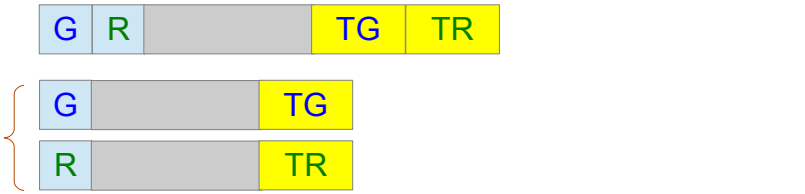
```
[trace]  ?- clause_tree(p, []).
   Call: (6) clause_tree(p, []) ? creep
   Call: (7) loop_detect(p, []) ? creep
   Fail: (7) loop_detect(p, []) ? creep
   Redo: (6) clause_tree(p, []) ? creep
^  Call: (7) clause(p, _G418) ? creep
^  Exit: (7) clause(p, q) ? creep
   Call: (7) clause_tree(q, [p]) ? creep
   Call: (8) loop_detect(q, [p]) ? creep  ...
   Fail: (8) loop_detect(q, [p]) ? creep
   Redo: (7) clause_tree(q, [p]) ? creep
^  Call: (8) clause(q, _G424) ? creep
^  Exit: (8) clause(q, p) ? creep
   Call: (8) clause_tree(p, [q, p]) ? creep
   Call: (9) loop_detect(p, [q, p]) ? creep  ...
   Exit: (9) loop_detect(p, [q, p]) ? creep
   Call: (9) fail ? creep
   Fail: (9) fail ? creep
   Fail: (8) clause_tree(p, [q, p]) ? creep
   Fail: (7) clause_tree(q, [p]) ? creep
^  Exit: (7) clause(p, r) ? creep
   Call: (7) clause_tree(r, [p]) ? creep
   Call: (8) loop_detect(r, [p]) ? creep  ...
   Fail: (8) loop_detect(r, [p]) ? creep
   Redo: (7) clause_tree(r, [p]) ? creep
^  Call: (8) clause(r, _G424) ? creep
^  Exit: (8) clause(r, true) ? creep
   Call: (8) clause_tree(true, [r, p]) ? creep
   Exit: (8) clause_tree(true, [r, p]) ? creep
   Exit: (7) clause_tree(r, [p]) ? creep
   Exit: (6) clause_tree(p, []) ? creep
true.
```



Making a clause tree list

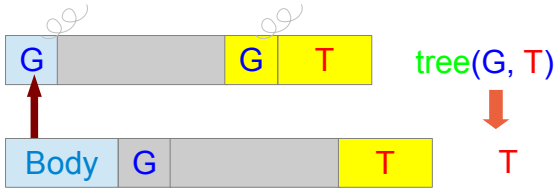
```
clause_tree(true, _, true) :- !.

clause_tree((G,R), Trail, (TG,TR)) :-
   !,
   clause_tree(G, Trail, TG),
   clause_tree(R, Trail, TR).

clause_tree(G, _, prolog(G)) :-
   ( predicate_property(G, built_in) ;
     predicate_property(G, compiled) ),
   !,
   call(G).              %% let Prolog do it

clause_tree(G, Trail, _) :-
   loop_detect(G, Trail),
   !,
   fail.

clause_tree(G, Trail, tree(G,T)) :-
   clause(G, Body),
   clause_tree(Body, [G|Trail], T).
```

TG: tree list representation of G
TR: tree list representation of R



G is a prolog expression

expand G in the tree list representation
with the body of a matching clause



Examples of making a clause tree list

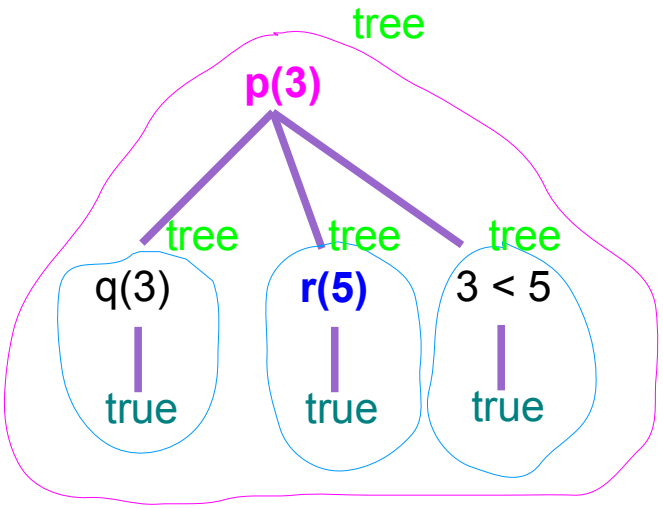
```
clause_tree(true, _, true) :- !.

clause_tree((G,R), Trail, (TG,TR)) :-
      !,   clause_tree(G, Trail, TG),   clause_tree(R, Trail, TR).

clause_tree(G, _, prolog(G)) :-
      ( predicate_property(G, built_in) ;  predicate_property(G, compiled) ),   !,  call(G).

clause_tree(G, Trail, _) :-
      loop_detect(G, Trail),   !,    fail.

clause_tree(G, Trail, tree(G,T)) :-
      clause(G, Body),    clause_tree(Body, [G|Trail], T).
```

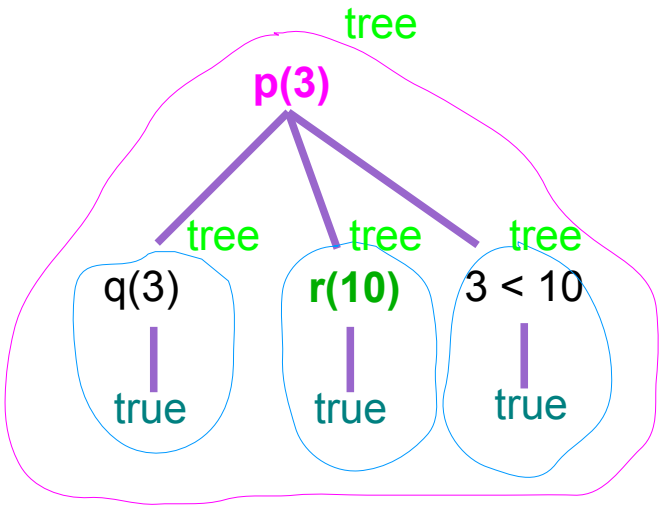
```
?- clause_tree(p(X),[],Tree)

Tree = tree( p(3), (tree(q(3),true), tree(r(5),true), prolog(3 < 5)) )
X = 3 ;

Tree = tree( p(3), (tree(q(3),true), tree(r(10),true), prolog(3 < 10)) )
X = 3 ;

No
```

clauses

```
p(X) :- q(X), r(Y), X < Y.
q(3).
r(2).
r(5).
r(10).
```

Lists and corresponding trees






```
?- clause_tree(p(X), [], Tree)

Tree = tree( p(3), (tree(q(3),true), tree(r(5),true), prolog(3 < 5)) )
X = 3 ;

Tree = tree( p(3), (tree(q(3),true), tree(r(10),true), prolog(3 < 10)) )
X = 3 ;

No
```

```
p(X) :- q(X), r(Y), X < Y.
q(3).
r(2).
r(5).
r(10).
```

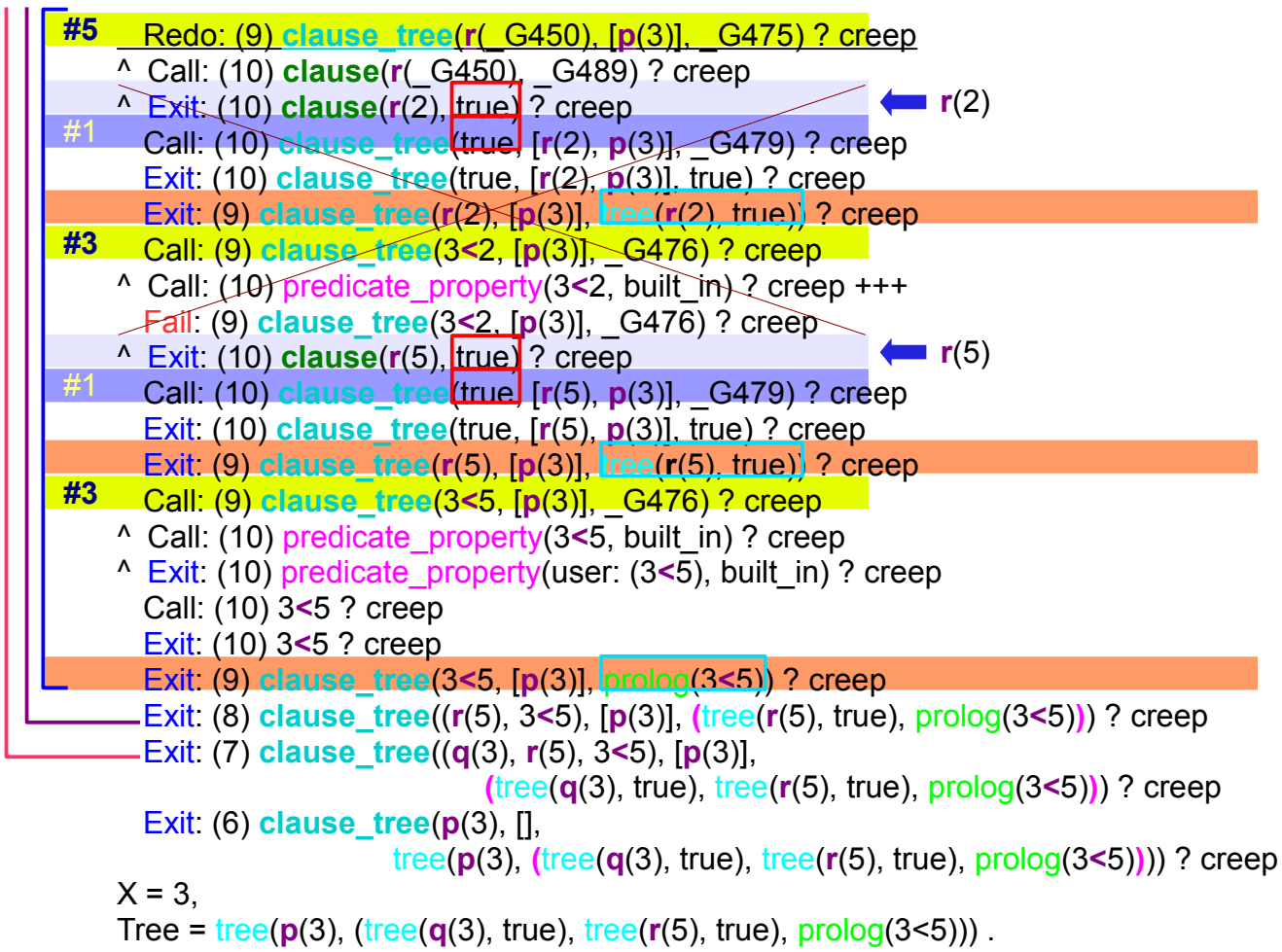
The trace result – overview (1)

```
[trace]  ?- clause_tree(p(X),[],Tree).
#3    Call: (6) clause_tree(p(_G356), [], _G360) ? creep
      ^ Call: (7) predicate_property(p(_G356), built_in) ? creep +++
      ^ Fail: (7) predicate_property(user:p(_G356), compiled) ? creep
#4    Redo: (6) clause_tree(p(_G356), [], _G360) ? creep
        Call: (7) loop_detect(p(_G356), []) ? creep +++
        Fail: (7) loop_detect(p(_G356), []) ? creep
#5    Redo: (6) clause_tree(p(_G356), [], _G360) ? creep
      ^ Call: (7) clause(p(_G356), _G451) ? creep +++
      ^ Exit: (7) clause(p(_G356), (q(_G356), r(_G450), _G356<_G450)) ? creep
#2    Call: (7) clause_tree((q(_G356), r(_G450), _G356<_G450), [p(_G356)], _G441) ? cr
#3    Call: (8) clause_tree(q(_G356), [p(_G356)], _G464) ? creep
      ^ Call: (9) predicate_property(q(_G356), built_in) ? creep +++
      ^ Fail: (9) predicate_property(user:q(_G356), compiled) ? creep
#4    Redo: (8) clause_tree(q(_G356), [p(_G356)], _G464) ? creep
        Call: (9) loop_detect(q(_G356), [p(_G356)]) ? creep +++
        Fail: (9) loop_detect(q(_G356), [p(_G356)]) ? creep
#5    Redo: (8) clause_tree(q(_G356), [p(_G356)], _G464) ? creep
      ^ Call: (9) clause(q(_G356), _G478) ? creep
      ^ Exit: (9) clause(q(3), true) ? creep
#1    Call: (9) clause_tree(true, [q(3), p(3)], _G468) ? creep
        Exit: (9) clause_tree(true, [q(3), p(3)], true) ? creep
        Exit: (8) clause_tree(q(3), [p(3)], (q(3), true)) ? creep
#2    Call: (8) clause_tree((r(_G450), 3<_G450), [p(3)], _G465) ? creep
#3    Call: (9) clause_tree(r(_G450), [p(3)], _G475) ? creep
      ^ Call: (10) predicate_property(r(_G450), built_in) ? creep +++
      ^ Fail: (10) predicate_property(user:r(_G450), compiled) ? creep
#4    Redo: (9) clause_tree(r(_G450), [p(3)], _G475) ? creep
        Call: (10) loop_detect(r(_G450), [p(3)]) ? creep +++
        Fail: (10) loop_detect(r(_G450), [p(3)]) ? creep
#5    Redo: (9) clause_tree(r(_G450), [p(3)], _G475) ? creep
```

```
#1 clause_tree(true, _, true) :- !.

#2 clause_tree((G,R), Trail, (TG,TR)) :-
   !,
   clause_tree(G, Trail, TG),
   clause_tree(R, Trail, TR).

#3 clause_tree(G, _, prolog(G)) :-
   ( predicate_property(G, built_in) ;
     predicate_property(G, compiled) ),
   !,
   call(G).              %% let Prolog do it

#4 clause_tree(G, Trail, _) :-
   loop_detect(G, Trail),
   !,
   fail.

#5 clause_tree(G, Trail, tree(G,T)) :-
   clause(G, Body),
   clause_tree(Body, [G|Trail], T).
```

```
p(X) :- q(X), r(Y), X < Y.
q(3).
r(2).
r(5).
r(10).
```

The trace result – overview (2)




```
#1 clause_tree(true, _, true) :- !.

#2 clause_tree((G,R), Trail, (TG,TR)) :-
   !,
   clause_tree(G, Trail, TG),
   clause_tree(R, Trail, TR).

#3 clause_tree(G, _, prolog(G)) :-
   ( predicate_property(G, built_in) ;
     predicate_property(G, compiled) ),
   !,
   call(G).              %% let Prolog do it

#4 clause_tree(G, Trail, _) :-
   loop_detect(G, Trail),
   !,
   fail.

#5 clause_tree(G, Trail, tree(G,T)) :-
   clause(G, Body),
   clause_tree(Body, [G|Trail], T).
```

```
p(X) :- q(X), r(Y), X < Y.
q(3).
r(2).
r(5).
r(10).
```

The trace result of making a tree list (1)

```
[trace]  ?- clause_tree(p(X),[],Tree).
#3    Call: (6) clause_tree(p(_G356), [], _G360) ? creep
      ^  Call: (7) predicate_property(p(_G356), built_in) ? creep
      ^  Fail: (7) predicate_property(user:p(_G356), built_in) ? creep
         Redo: (6) clause_tree(p(_G356), [], prolog(p(_G356))) ? creep
      ^  Call: (7) predicate_property(p(_G356), compiled) ? creep
      ^  Fail: (7) predicate_property(user:p(_G356), compiled) ? creep
#4    Redo: (6) clause_tree(p(_G356), [], _G360) ? creep
         Call: (7) loop_detect(p(_G356), []) ? creep
         Fail: (7) loop_detect(p(_G356), []) ? creep
#5    Redo: (6) clause_tree(p(_G356), [], _G360) ? creep
      ^  Call: (7) clause(p(_G356), _G451) ? creep
      ^  Exit: (7) clause(p(_G356), (q(_G356), r(_G450), _G356< _G450)) ? creep
#2    Call: (7) clause_tree((q(_G356), r(_G450), _G356< _G450), [p(_G356)], _G441) ? creep
#3    Call: (8) clause_tree(q(_G356), [p(_G356)], _G464) ? creep
      ^  Call: (9) predicate_property(q(_G356), built_in) ? creep
      ^  Fail: (9) predicate_property(user:q(_G356), built_in) ? creep
      Redo: (8) clause_tree(q(_G356), [p(_G356)], prolog(q(_G356))) ? creep
      ^  Call: (9) predicate_property(q(_G356), compiled) ? creep
      ^  Fail: (9) predicate_property(user:q(_G356), compiled) ? creep
#4    Redo: (8) clause_tree(q(_G356), [p(_G356)], _G464) ? creep
         Call: (9) loop_detect(q(_G356), [p(_G356)]) ? creep
         Call: (10) q(_G356)==p(_G356) ? creep
         Fail: (10) q(_G356)==p(_G356) ? creep
      Redo: (9) loop_detect(q(_G356), [p(_G356)]) ? creep
         Call: (10) loop_detect(q(_G356), []) ? creep
         Fail: (10) loop_detect(q(_G356), []) ? creep
         Fail: (9) loop_detect(q(_G356), [p(_G356)]) ? creep
#5    Redo: (8) clause_tree(q(_G356), [p(_G356)], _G464) ? creep
```

```
#1 clause_tree(true, _, true) :- !.

#2 clause_tree((G,R), Trail, (TG,TR)) :-
   !,
   clause_tree(G, Trail, TG),
   clause_tree(R, Trail, TR).

#3 clause_tree(G, _, prolog(G)) :-
   ( predicate_property(G, built_in) ;
     predicate_property(G, compiled) ),
   !,
   call(G).              %% let Prolog do it

#4 clause_tree(G, Trail, _) :-
   loop_detect(G, Trail),
   !,
   fail.

#5 clause_tree(G, Trail, tree(G,T)) :-
   clause(G, Body),
   clause_tree(Body, [G|Trail], T).
```

```
p(X) :- q(X), r(Y), X < Y.
q(3).
r(2).
r(5).
r(10).
```

The trace result of making a tree list (2)

```
#5   Redo: (8) clause_tree(q(_G356), [p(_G356)], _G464) ? creep
     ^ Call: (9) clause(q(_G356), _G478) ? creep
     ^ Exit: (9) clause(q(3), true) ? creep
#1   Call: (9) clause_tree(true, [q(3), p(3)], _G468) ? creep
     Exit: (9) clause_tree(true, [q(3), p(3)], true) ? creep
     Exit: (8) clause_tree(q(3), [p(3)], tree(q(3), true)) ? creep
#2   Call: (8) clause_tree((r(_G450), 3<_G450), [p(3)], _G465) ? creep
#3   Call: (9) clause_tree(r(_G450), [p(3)], _G475) ? creep
     ^ Call: (10) predicate_property(r(_G450), built_in) ? creep
     ^ Fail: (10) predicate_property(user:r(_G450), built_in) ? creep
     Redo: (9) clause_tree(r(_G450), [p(3)], prolog(r(_G450))) ? creep
     ^ Call: (10) predicate_property(r(_G450), compiled) ? creep
     ^ Fail: (10) predicate_property(user:r(_G450), compiled) ? creep
#4   Redo: (9) clause_tree(r(_G450), [p(3)], _G475) ? creep
       Call: (10) loop_detect(r(_G450), [p(3)]) ? creep
       Call: (11) r(_G450)==p(3) ? creep
       Fail: (11) r(_G450)==p(3) ? creep
     Redo: (10) loop_detect(r(_G450), [p(3)]) ? creep
       Call: (11) loop_detect(r(_G450), []) ? creep
       Fail: (11) loop_detect(r(_G450), []) ? creep
       Fail: (10) loop_detect(r(_G450), [p(3)]) ? creep
#5   Redo: (9) clause_tree(r(_G450), [p(3)], _G475) ? creep
     ^ Call: (10) clause(r(_G450), _G489) ? creep
     ^ Exit: (10) clause(r(2), true) ? creep
#1   Call: (10) clause_tree(true, [r(2), p(3)], _G479) ? creep
       Exit: (10) clause_tree(true, [r(2), p(3)], true) ? creep
       Exit: (9) clause_tree(r(2), [p(3)], tree(r(2), true)) ? creep
```

```
#1 clause_tree(true, _, true) :- !.

#2 clause_tree((G,R), Trail, (TG,TR)) :-
     !,
     clause_tree(G, Trail, TG),
     clause_tree(R, Trail, TR).

#3 clause_tree(G, _, prolog(G)) :-
     ( predicate_property(G, built_in) ;
       predicate_property(G, compiled) ),
     !,
     call(G).              %% let Prolog do it

#4 clause_tree(G, Trail, _) :-
     loop_detect(G, Trail),
     !,
     fail.

#5 clause_tree(G, Trail, tree(G,T)) :-
     clause(G, Body),
     clause_tree(Body, [G|Trail], T).
```

```
p(X) :- q(X), r(Y), X < Y.
q(3).
r(2).
r(5).
r(10).
```

The trace result of making a tree list (3)

```
#3    Call: (9) clause_tree(3<2, [p(3)], _G476) ? creep
      ^ Call: (10) predicate_property(3<2, built_in) ? creep
      ^ Exit: (10) predicate_property(user: (3<2), built_in) ? creep
        Call: (10) 3<2 ? creep
        Fail: (10) 3<2 ? creep
        Fail: (9) clause_tree(3<2, [p(3)], _G476) ? creep
      ^ Exit: (10) clause(r(5), true) ? creep
#1    Call: (10) clause_tree(true, [r(5), p(3)], _G479) ? creep
        Exit: (10) clause_tree(true, [r(5), p(3)], true) ? creep
        Exit: (9) clause_tree(r(5), [p(3)], tree(r(5), true)) ? creep
#3    Call: (9) clause_tree(3<5, [p(3)], _G476) ? creep
      ^ Call: (10) predicate_property(3<5, built_in) ? creep
      ^ Exit: (10) predicate_property(user: (3<5), built_in) ? creep
        Call: (10) 3<5 ? creep
        Exit: (10) 3<5 ? creep
        Exit: (9) clause_tree(3<5, [p(3)], prolog(3<5)) ? creep
        Exit: (8) clause_tree((r(5), 3<5), [p(3)], (tree(r(5), true), prolog(3<5))) ? creep
        Exit: (7) clause_tree((q(3), r(5), 3<5), [p(3)], (tree(q(3), true), tree(r(5), true), prolog(3<5))
        Exit: (6) clause_tree(p(3), [], tree(p(3), (tree(q(3), true), tree(r(5), true), prolog(3<5)))) ? cr
      X = 3,
      Tree = tree(p(3), (tree(q(3), true), tree(r(5), true), prolog(3<5))) .
```

```
#1 clause_tree(true, _, true) :- !.

#2 clause_tree((G,R), Trail, (TG,TR)) :-
     !,
     clause_tree(G, Trail, TG),
     clause_tree(R, Trail, TR).

#3 clause_tree(G, _, prolog(G)) :-
     ( predicate_property(G, built_in) ;
       predicate_property(G, compiled) ),
     !,
     call(G).              %% let Prolog do it

#4 clause_tree(G, Trail, _) :-
     loop_detect(G, Trail),
     !,
     fail.

#5 clause_tree(G, Trail, tree(G,T)) :-
     clause(G, Body),
     clause_tree(Body, [G|Trail], T).
```

```
p(X) :- q(X), r(Y), X < Y.
q(3).
r(2).
r(5).
r(10).
```

Drawing clause trees

```
why(G) :- clause_tree(G, [], T),
   nl,
   draw_tree(T, 5).

draw_tree(tree(Root, Branches), Tab) :- !,
   tab(Tab),
   write('|-- '),
   write(Root),
   nl,
   Tab5 is Tab + 5,
   draw_tree(Branches,Tab5).

draw_tree((B, Bs), Tab) :- !,
   draw_tree(B, Tab),
   draw_tree(Bs, Tab).

draw_tree(Node,Tab) :-
   tab(Tab),
   write('|-- '),
   write(Node),
   nl.
```

```
?- why(p(X)).

   |-- p(3)
       |-- q(3)
           |-- true
       |-- r(5)
           |-- true
       |-- prolog(3 <  5)

X = 3 ;

   |-- p(3)
       |-- q(3)
           |-- true
       |-- r(10)
           |-- true
       |-- prolog(3 <  10)

X = 3 ;

No
```

Drawing the 1st clause tree

```
why(G) :- clause_tree(G, [], T),   nl,   draw_tree(T, 5).

draw_tree(tree(Root, Branches), Tab) :- !,
   tab(Tab),   write('|-- '),   write(Root),   nl,   Tab5 is Tab + 5, draw_tree(Branches,Tab5).

draw_tree((B, Bs), Tab) :- !,   draw_tree(B, Tab),   draw_tree(Bs, Tab).

draw_tree(Node,Tab) :-   tab(Tab),   write('|-- '),   write(Node),   nl.
```

```
tree( p(3), (tree(q(3),true), tree(r(5),true), prolog(3 < 5)) )
(tree(q(3),true), tree(r(5),true), prolog(3 < 5))
tree(q(3),true)
q(3)
true
tree(r(5),true), prolog(3 < 5)
tree(r(5),true)
r(5)
true
prolog(3 < 5)
```

```
?- why(p(X)).

   |-- p(3)
        |-- q(3)
             |-- true
        |-- r(5)
             |-- true
        |-- prolog(3 <  5)

X = 3 ;
```

Drawing the 2nd clause tree

```
why(G) :- clause_tree(G, [], T),   nl,   draw_tree(T, 5).

draw_tree(tree(Root, Branches), Tab) :- !,
   tab(Tab),   write('|-- '),   write(Root),   nl,   Tab5 is Tab + 5, draw_tree(Branches,Tab5).

draw_tree((B, Bs), Tab) :- !,   draw_tree(B, Tab),   draw_tree(Bs, Tab).

draw_tree(Node,Tab) :-   tab(Tab),   write('|-- '),   write(Node),   nl.
```

```
tree( p(3), (tree(q(3),true), tree(r(10),true), prolog(3 < 10)) )
(tree(q(3),true), tree(r(10),true), prolog(3 < 10))
tree(q(3),true)
q(3)
true
tree(r(10),true), prolog(3 < 10)
tree(r(10),true)
r(10)
true
prolog(3 < 10)
```

```
|-- p(3)
     |-- q(3)
          |-- true
     |-- r(10)
          |-- true
     |-- prolog(3 <  10)

X = 3 ;

No
```

Iterative Deepening

```
clause_tree(true, _, _) :- !.

clause_tree(_, D, Limit) :-
      D > Limit,
      !,
      fail.  %% reached depth limit

clause_tree((A,B), D, Limit) :- !,
      clause_tree(A, D, Limit),
      clause_tree(B, D, Limit).

clause_tree(A, _, _) :-
      predicate_property(A, built_in),
      !,
      call(A).

clause_tree(A, D, Limit) :-
      clause(A, B),
      D1 is D+1,
      clause_tree(B, D1, Limit).
```

```
iterative_deepening(G, D) :-
      clause_tree(G, 0, D).

iterative_deepening(G, D) :-
      write('limit='),
      write(D),
      write('(Hit Enter to Continue.)'),
      get0(C),
      ( C == 10 ->
        D1 is D + 5,
        iterative_deepening(G, D1)  ).
```

Prolog Input (1)

get0(-Char)

Edinburgh version of the ISO **get_code**/1 predicate.
Note that Edinburgh Prolog didn't support wide characters and therefore technically speaking **get0**/1 should have been mapped to **get_byte**/1.
The intention of **get0**/1, however, is to read character codes.

get_code(-Code)

Read the current input stream and unify Code with the character code of the next character. Code is unified with -1 on end of file. See also get_char/1.

get_char(-Char)

Read the current input stream and unify Char with the next character as a one-character atom. See also atom_chars/2. On end-of-file, Char is unified to the atom end_of_file.

get_byte(-Byte)

Read the current input stream and unify the next byte with Byte (an integer between 0 and 255). Byte is unified with -1 on end of file.

Prolog Input (2)

input in Prolog, read, end_of_file, get, get_byte, getc, flush_output

read(X) reads the next term in the current input stream

end_of_file read(X) causes X to be bound to this special symbol

get_byte(C) reads a single character from the current input stream

get(C) reads the first non-blank character

flush_output is actually an output goal

get0(C) reads character codes.

eg. 10 (0A) LF (line feed)

Prolog if-then-else statement

The built-in infix predicate X -> Y ; Z functions
as an if X then Y else Z facility.

```
min(A, B, Min) :- A < B -> Min = A ; Min = B.

min(A, B, A) :- A <= B.
min(A, B, B) :- B < A.
```

```
If -> Then   ; _Else :- If, !, Then.
If -> _Then ; Else   :-    !, Else.
If -> Then           :- If, !, Then.
```

Please note that (If -> Then) acts as (If -> Then ; fail), making the construct fail if the condition fails. This unusual semantics is part of the ISO and all de-facto Prolog standards.

Predicate Description Notations (1)

predicate(?Variable1, +Variable2, –Variable3)

?Var1: Var1 can be either instantiated or not. Both ways are possible.

+Var2: Var2 is an input to the predicate. As such it must be instantiated.

–Var3: Var3 is an output to the predicate. It is usually non-instantiated,
but may be instantiated if you want to check for a specific "return value".

Predicate Description Notations (2)

+ Argument must be fully instantiated to a term
that satisfies the required argument type.
Think of the argument as input.

- Argument must be unbound. Think of the argument as output.

? Argument must be bound to a partial term of the indicated type.
Note that a variable is a partial term for any type.
Think of the argument as either input or output or
both input and output.
For example, in stream_property(S, reposition(Bool)),
the reposition part of the term is input and
the uninstantiated Bool is output.

: Argument is a meta-argument. Implies +.

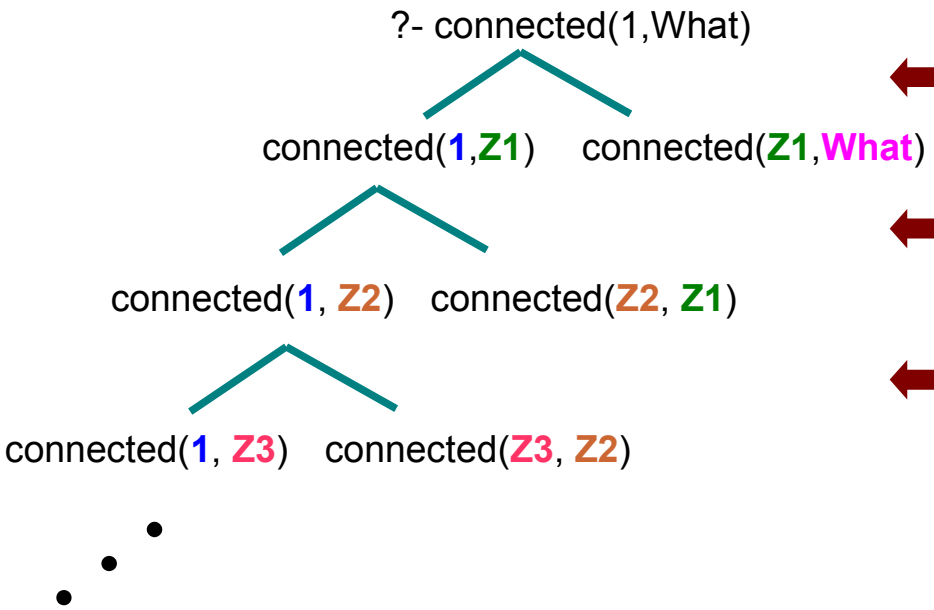
@ Argument is not further instantiated.
Typically used for type tests.

! Argument contains a mutable structure
that may be modified using setarg/3 or nb_setarg/3.

Infinite Descent

```
connected(X,Y) :- connected(X,Z), connected(Z,Y).
connected(1,2).
connected(2,3).
connected(3,4).
connected(4,5).
```

?- connected(1,2).
...



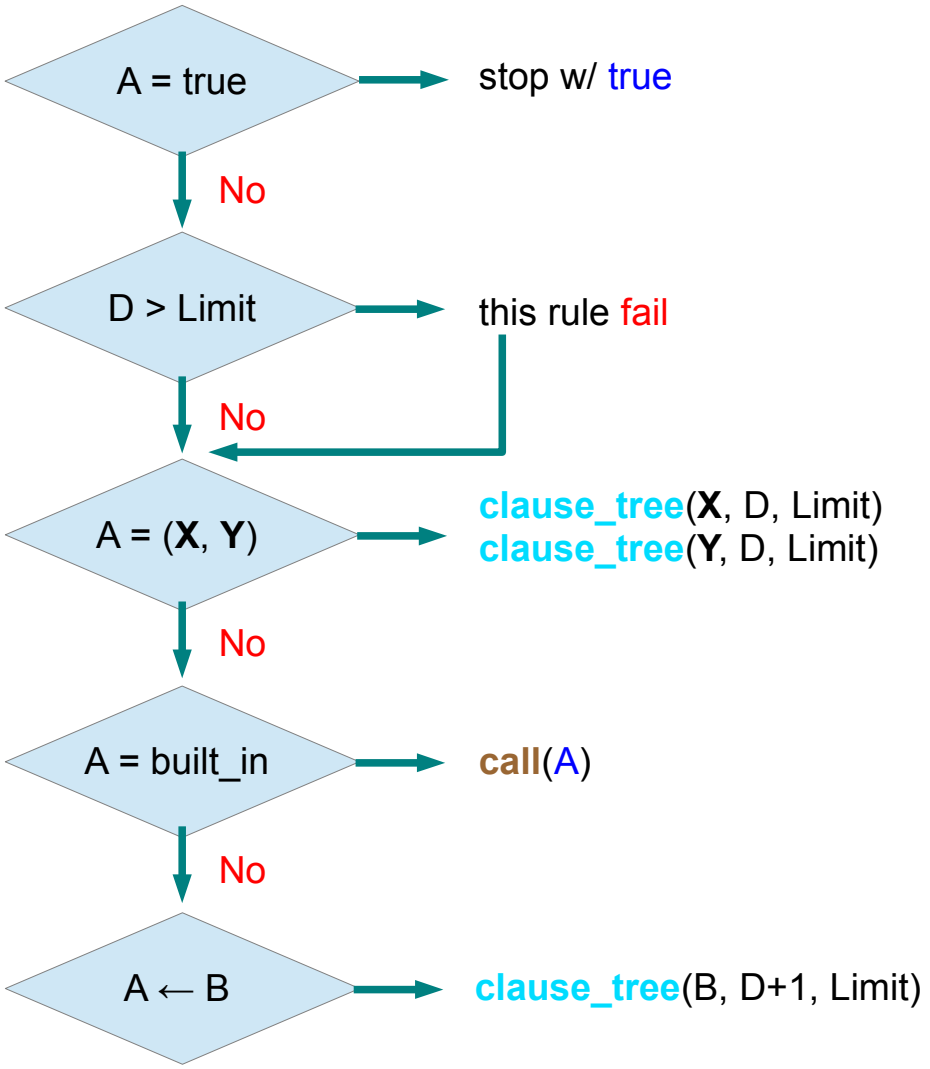
← connected(X,Y) :- connected(X,Z), connected(Z,Y).

← connected(X,Y) :- connected(X,Z), connected(Z,Y).

← connected(X,Y) :- connected(X,Z), connected(Z,Y).

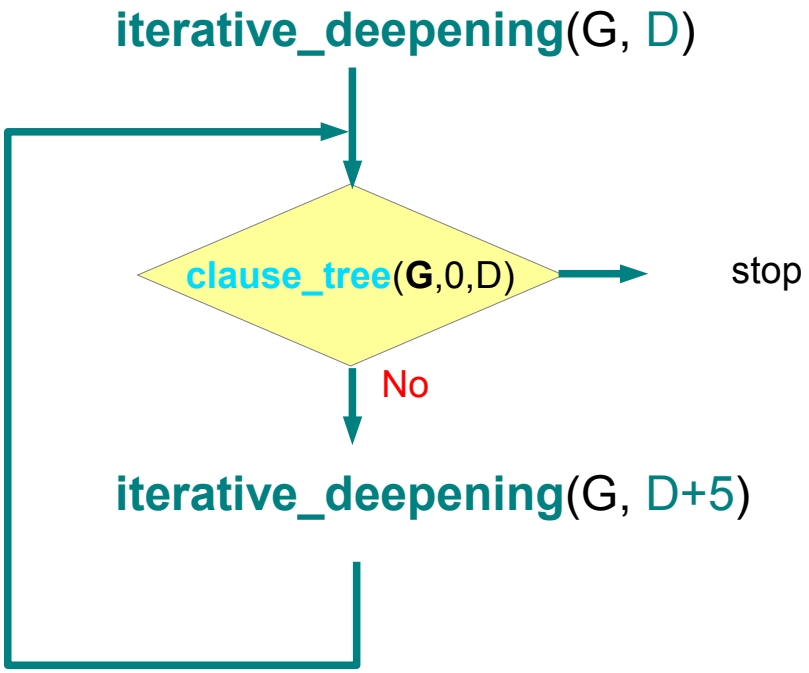
Iterative Deepening

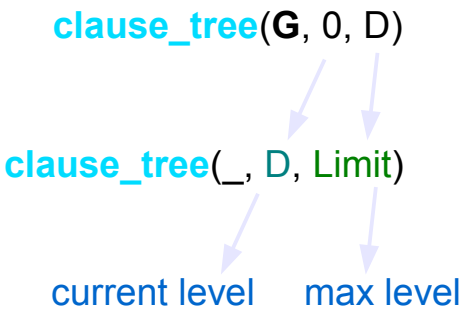
```
clause_tree(true, _, _) :- !.

clause_tree(_, D, Limit) :-
      D > Limit,
      !,
      fail.  %% reached depth limit

clause_tree((A,B), D, Limit) :- !,
      clause_tree(A, D, Limit),
      clause_tree(B, D, Limit).

clause_tree(A, _, _) :-
      predicate_property(A, built_in),
      !,
      call(A).

clause_tree(A, D, Limit) :-
      clause(A, B),
      D1 is D+1,
      clause_tree(B, D1, Limit).
```



Iterative Deepening

```
iterative_deepening(G, D) :-
      clause_tree(G, 0, D).

iterative_deepening(G, D) :-
      write('limit='),
      write(D),
      write('(Hit Enter to Continue.)'),
      get0(C),
      ( C == 10 ->
        D1 is D + 5,
        iterative_deepening(G, D1)  ).
```

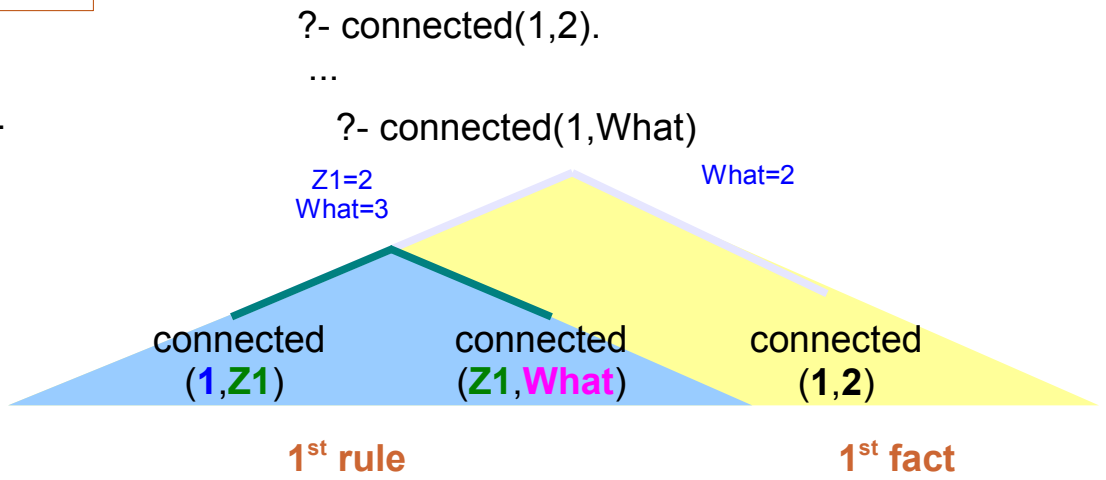




Infinite Descent

```
connected(X,Y) :- connected(X,Z), connected(Z,Y).
connected(1,2).
connected(2,3).
connected(3,4).
connected(4,5).
```

```
?- iterative_deepening(connected(1,What), 1).
What=3 ;
What=2 ;
Limit=1(Hit Enter to Continue)
What=5 ;
What=5 ;
What=5 ;
What=4 ;
What=5 ;
What=5 ;
What=4 ;
What=3 ;
What=2 ;
Limit=6(Hit Enter to Continue.)
What=5                          %% stop

Yes
```



Infinite Descent

```
?- [ideep].
% ideep compiled 0.00 sec, 9 clauses
true.

?- [ideep_rule].
% ideep_rule compiled 0.00 sec, 7 clauses
true.

?- trace(clause_tree).
%         clause_tree/3: [call,redo,exit,fail]
true.
```

Trace result with D+1 (1)

```
[debug]  ?- iterative_deepening(connected(1,What), 1).
 T Call: (7) clause_tree(connected(1, _G357), 0, 1)
 T Redo: (7) clause_tree(connected(1, _G357), 0, 1)
 T Redo: (7) clause_tree(connected(1, _G357), 0, 1)
 T       Call: (8) clause_tree((connected(1, _G435), connected(_G435, _G357)), 1, 1)
 T       Redo: (8) clause_tree((connected(1, _G435), connected(_G435, _G357)), 1, 1)
 T                 Call: (9) clause_tree(connected(1, _G435), 1, 1)
 T                 Redo: (9) clause_tree(connected(1, _G435), 1, 1)
 T                 Redo: (9) clause_tree(connected(1, _G435), 1, 1)
 T                           Call: (10) clause_tree((connected(1, _G453), connected(_G453, _G435)), 2, 1)
 T                           Fail: (10) clause_tree((connected(1, _G453), connected(_G453, _G435)), 2, 1)
 T                           Call: (10) clause_tree(true, 2, 1)
 T                           Exit: (10) clause_tree(true, 2, 1)
 T                 Exit: (9) clause_tree(connected(1, 2), 1, 1)
 T                 Call: (9) clause_tree(connected(2, _G357), 1, 1)
 T                 Redo: (9) clause_tree(connected(2, _G357), 1, 1)
 T                 Redo: (9) clause_tree(connected(2, _G357), 1, 1)
 T                           Call: (10) clause_tree((connected(2, _G459), connected(_G459, _G357)), 2, 1)
 T                           Fail: (10) clause_tree((connected(2, _G459), connected(_G459, _G357)), 2, 1)
 T                           Call: (10) clause_tree(true, 2, 1)
 T                           Exit: (10) clause_tree(true, 2, 1)
 T                 Exit: (9) clause_tree(connected(2, 3), 1, 1)
 T       Exit: (8) clause_tree((connected(1, 2), connected(2, 3)), 1, 1)
 T Exit: (7) clause_tree(connected(1, 3), 0, 1)
What = 3 ;
 T       Call: (8) clause_tree(true, 1, 1)
 T       Exit: (8) clause_tree(true, 1, 1)
 T Exit: (7) clause_tree(connected(1, 2), 0, 1)
What = 2 ;
```

```
clause_tree(true, _, _) :- !.

clause_tree(_, D, Limit) :-
        D > Limit,
        !,
      fail.  %% reached depth limit

clause_tree((A,B), D, Limit) :- !,
        clause_tree(A, D, Limit),
      clause_tree(B, D, Limit).

clause_tree(A, _, _) :-
        predicate_property(A, built_in),
      !,
      call(A).

clause_tree(A, D, Limit) :-
        clause(A, B),
        D1 is D+1,
      clause_tree(B, D1, Limit).

iterative_deepening(G, D) :-
        clause_tree(G, 0, D).

iterative_deepening(G, D) :-
        write('limit='),
      write(D),
      write('(Hit Enter to Continue.)'),
      get0(C),
      ( C == 10 ->
        D1 is D + 1,
        iterative_deepening(G, D1)  ).
```

Trace result with D+1 (2)

```
limit=1(Hit Enter to Continue.)
T    Call: (8) clause_tree(connected(1, _G357), 0, 2)
T    Redo: (8) clause_tree(connected(1, _G357), 0, 2)
T    Redo: (8) clause_tree(connected(1, _G357), 0, 2)
T        Call: (9) clause_tree((connected(1, _G438), connected(_G438, _G357)), 1, 2)
T        Redo: (9) clause_tree((connected(1, _G438), connected(_G438, _G357)), 1, 2)
T            Call: (10) clause_tree(connected(1, _G438), 1, 2)
T            Redo: (10) clause_tree(connected(1, _G438), 1, 2)
T            Redo: (10) clause_tree(connected(1, _G438), 1, 2)
T                Call: (11) clause_tree((connected(1, _G456), connected(_G456, _G438)), 2, 2)
T                Redo: (11) clause_tree((connected(1, _G456), connected(_G456, _G438)), 2, 2)
T                    Call: (12) clause_tree(connected(1, _G456), 2, 2)
T                    Redo: (12) clause_tree(connected(1, _G456), 2, 2)
T                    Redo: (12) clause_tree(connected(1, _G456), 2, 2)
T                        Call: (13) clause_tree((connected(1, _G474), connected(_G474, _G456)), 3, 2)
T     c(1, 2)            Fail: (13) clause_tree((connected(1, _G474), connected(_G474, _G456)), 3, 2)
T                        Call: (13) clause_tree(true, 3, 2)
T                        Exit: (13) clause_tree(true, 3, 2)
T                    Exit: (12) clause_tree(connected(1, 2), 2, 2)
T  c(1, 3)           Call: (12) clause_tree(connected(2, _G438), 2, 2)
T                    Redo: (12) clause_tree(connected(2, _G438), 2, 2)
T                    Redo: (12) clause_tree(connected(2, _G438), 2, 2)
T                        Call: (13) clause_tree((connected(2, _G480), connected(_G480, _G438)), 3, 2)
T     c(2, 3)            Fail: (13) clause_tree((connected(2, _G480), connected(_G480, _G438)), 3, 2)
T                        Call: (13) clause_tree(true, 3, 2)
T                        Exit: (13) clause_tree(true, 3, 2)
T                    Exit: (12) clause_tree(connected(2, 3), 2, 2)
T                Exit: (11) clause_tree((connected(1, 2), connected(2, 3)), 2, 2)
T            Exit: (10) clause_tree(connected(1, 3), 1, 2)
T            Call: (10) clause_tree(connected(3, _G357), 1, 2)
T            Redo: (10) clause_tree(connected(3, _G357), 1, 2)
T  c(1, 5)   Redo: (10) clause_tree(connected(3, _G357), 1, 2)
```

Trace result with D+1 (3)

```
   c(1, 5)
T                    Call: (11) clause_tree((connected(3, _G486), connected(_G486, _G357)), 2, 2)
T                    Redo: (11) clause_tree((connected(3, _G486), connected(_G486, _G357)), 2, 2)
T                          Call: (12) clause_tree(connected(3, _G486), 2, 2)
T                          Redo: (12) clause_tree(connected(3, _G486), 2, 2)
T                          Redo: (12) clause_tree(connected(3, _G486), 2, 2)
T                                Call: (13) clause_tree((connected(3, _G504), connected(_G504, _G486)), 3, 2)
T                     c(3, 4)    Fail: (13) clause_tree((connected(3, _G504), connected(_G504, _G486)), 3, 2)
T                                Call: (13) clause_tree(true, 3, 2)
T                                Exit: (13) clause_tree(true, 3, 2)
T              c(3, 5)           Exit: (12) clause_tree(connected(3, 4), 2, 2)
T                          Call: (12) clause_tree(connected(4, _G357), 2, 2)
T                          Redo: (12) clause_tree(connected(4, _G357), 2, 2)
T                          Redo: (12) clause_tree(connected(4, _G357), 2, 2)
T                                Call: (13) clause_tree((connected(4, _G510), connected(_G510, _G357)), 3, 2)
T                     c(4, 5)    Fail: (13) clause_tree((connected(4, _G510), connected(_G510, _G357)), 3, 2)
T                                Call: (13) clause_tree(true, 3, 2)
T                                Exit: (13) clause_tree(true, 3, 2)
T                          Exit: (12) clause_tree(connected(4, 5), 2, 2)
T                    Exit: (11) clause_tree((connected(3, 4), connected(4, 5)), 2, 2)
T               Exit: (10) clause_tree(connected(3, 5), 1, 2)
T          Exit: (9) clause_tree((connected(1, 3), connected(3, 5)), 1, 2)
T     Exit: (8) clause_tree(connected(1, 5), 0, 2)
What = 5 ;
T                    Call: (11) clause_tree(true, 2, 2)
T                    Exit: (11) clause_tree(true, 2, 2)
T               Exit: (10) clause_tree(connected(3, 4), 1, 2)
T          Exit: (9) clause_tree((connected(1, 3), connected(3, 4)), 1, 2)
T     Exit: (8) clause_tree(connected(1, 4), 0, 2)
What = 4 ;
```

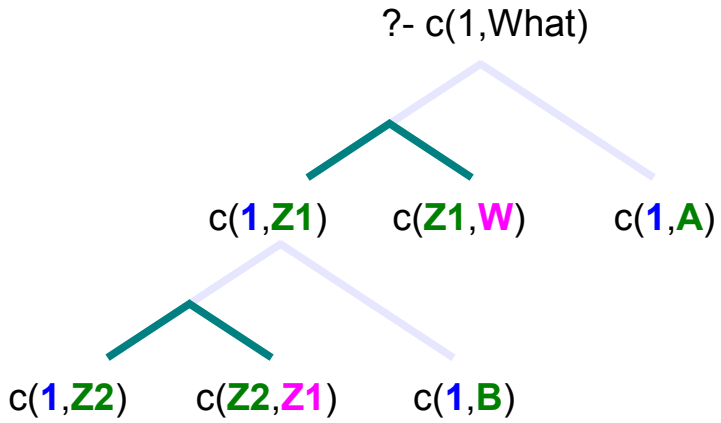
Infinite Descent (4)

```
T                                  Call: (11) clause_tree(true, 2, 2)
T                    c(1, 2)       Exit: (11) clause_tree(true, 2, 2)
T                  Exit: (10) clause_tree(connected(1, 2), 1, 2)
T                  Call: (10) clause_tree(connected(2, _G357), 1, 2)
T                  Redo: (10) clause_tree(connected(2, _G357), 1, 2)
T                  Redo: (10) clause_tree(connected(2, _G357), 1, 2)
T                              Call: (11) clause_tree((connected(2, _G462), connected(_G462, _G357)), 2, 2)
T                              Redo: (11) clause_tree((connected(2, _G462), connected(_G462, _G357)), 2, 2)
T                                          Call: (12) clause_tree(connected(2, _G462), 2, 2)
T                                          Redo: (12) clause_tree(connected(2, _G462), 2, 2)
T                                          Redo: (12) clause_tree(connected(2, _G462), 2, 2)
T                                                      Call: (13) clause_tree((connected(2, _G480), connected(_G480, _G462)), 3, 2)
T c(1, 4)                      c(2, 3)                 Fail: (13) clause_tree((connected(2, _G480), connected(_G480, _G462)), 3, 2)
T                                                      Call: (13) clause_tree(true, 3, 2)
T                                                      Exit: (13) clause_tree(true, 3, 2)
T                                          Exit: (12) clause_tree(connected(2, 3), 2, 2)
T                                          Call: (12) clause_tree(connected(3, _G357), 2, 2)
T                                          Redo: (12) clause_tree(connected(3, _G357), 2, 2)
T                                          Redo: (12) clause_tree(connected(3, _G357), 2, 2)
T                                                      Call: (13) clause_tree((connected(3, _G486), connected(_G486, _G357)), 3, 2)
T                                      c(3, 4)         Fail: (13) clause_tree((connected(3, _G486), connected(_G486, _G357)), 3, 2)
T                                                      Call: (13) clause_tree(true, 3, 2)
T                                                      Exit: (13) clause_tree(true, 3, 2)
T                                          Exit: (12) clause_tree(connected(3, 4), 2, 2)
T                              Exit: (11) clause_tree((connected(2, 3), connected(3, 4)), 2, 2)
T                  Exit: (10) clause_tree(connected(2, 4), 1, 2)
T            Exit: (9) clause_tree((connected(1, 2), connected(2, 4)), 1, 2)
T     Exit: (8) clause_tree(connected(1, 4), 0, 2)
What = 4 ;
```

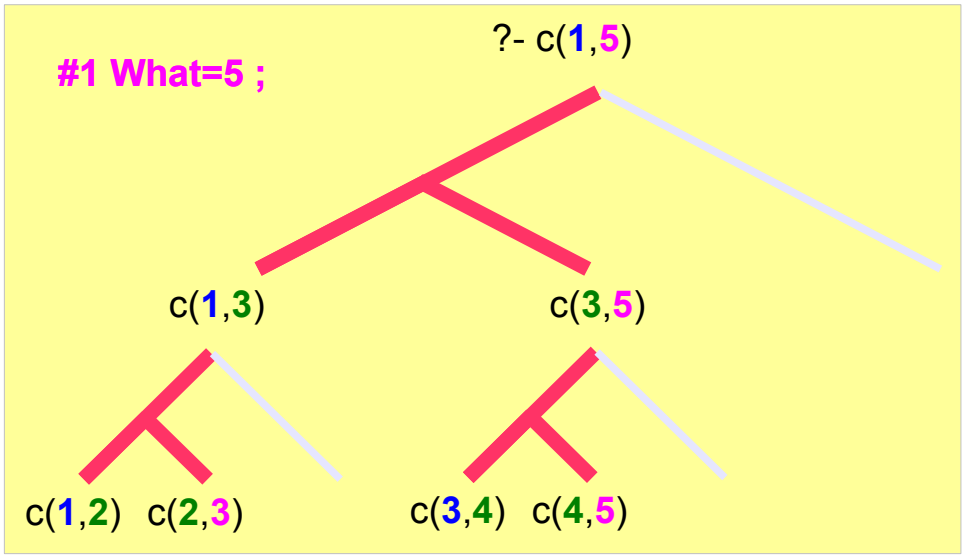
Trace result with D+1 (5)

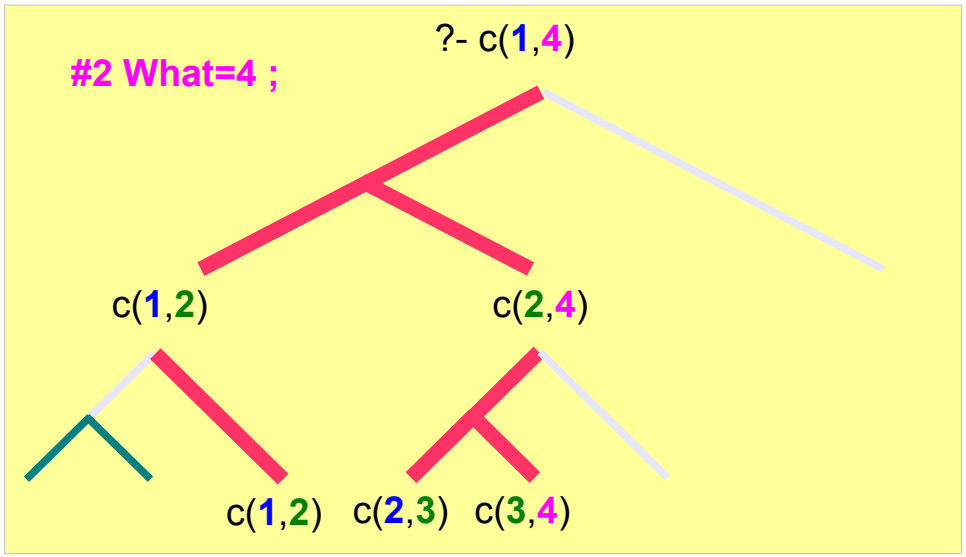
```
T  c(1, 3)      c(2, 3)  ➡ Call: (11) clause_tree(true, 2, 2)
T                          Exit: (11) clause_tree(true, 2, 2)
T                       Exit: (10) clause_tree(connected(2, 3), 1, 2)
T               Exit: (9) clause_tree((connected(1, 2), connected(2, 3)), 1, 2)
T       Exit: (8) clause_tree(connected(1, 3), 0, 2)
What = 3 ;
T  c(1, 2)      Call: (9) clause_tree(true, 1, 2)
T               Exit: (9) clause_tree(true, 1, 2)
T       Exit: (8) clause_tree(connected(1, 2), 0, 2)
What = 2 ;
limit=2(Hit Enter to Continue.)
```

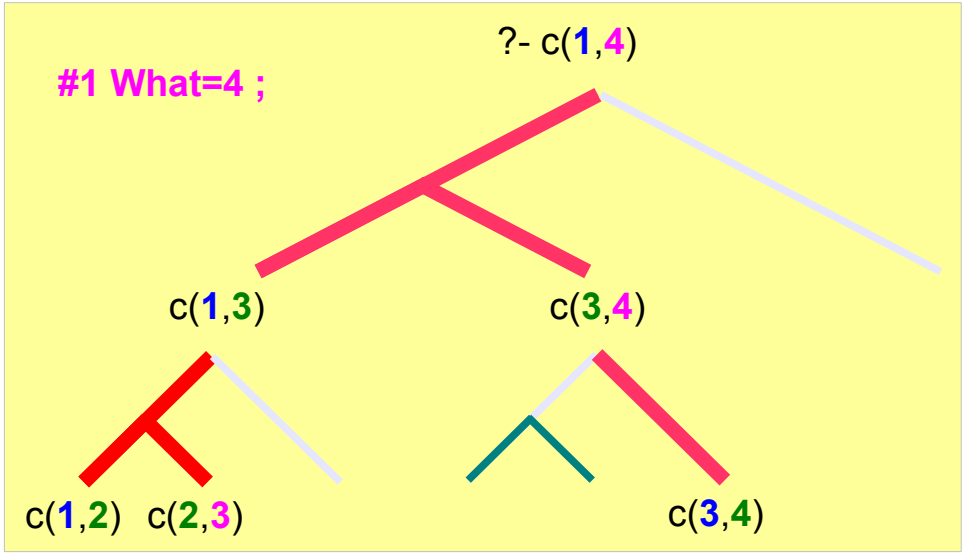
What=5 ;
What=4 ;
What=3 ;
What=2 ;

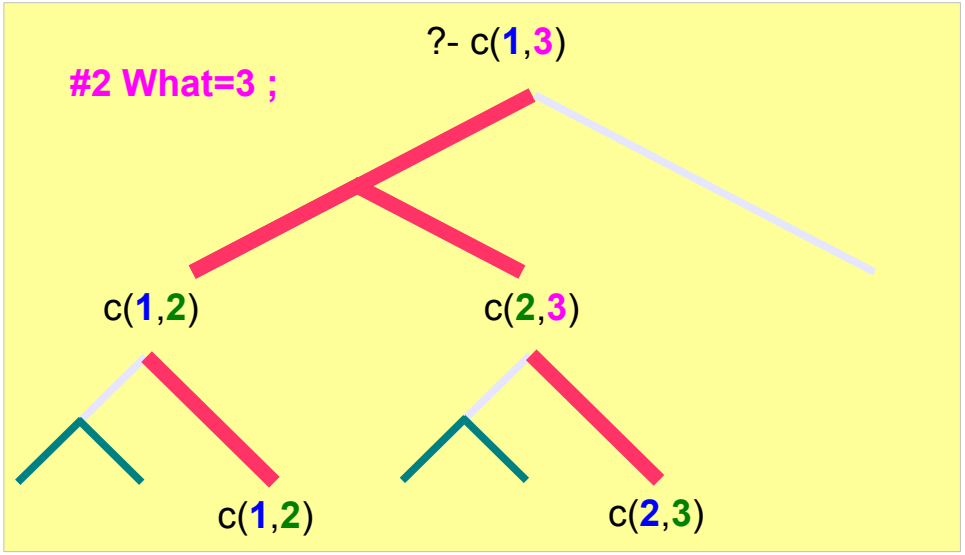


Tree result with D+1 (1)

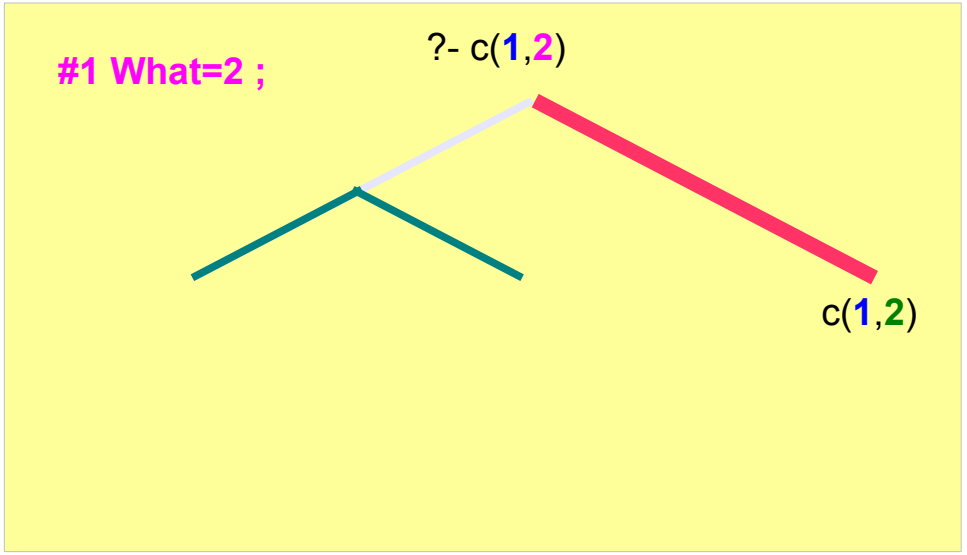








Tree result with D+1 (2)



Meta Programming (8A)

References

[1] en.wikipedia.org
[2] en.wiktionary.org
[3] U. Endriss, “Lecture Notes : Introduction to Prolog Programming”
[4] http://www.learnprolognow.org/ Learn Prolog Now!
[5] http://www.csupomona.edu/~jrfisher/www/prolog_tutorial
[6] www.cse.unsw.edu.au/~billw/cs9414/notes/prolog/intro.html
[7] www.cse.unsw.edu.au/~billw/dictionaries/prolog/negation.html