Programming Languages Workshop

CS 67420
Lecture 9

Huji, Spring 2016

Dr. Arie Schlesinger

Backtracking, cuts & negation

backtracking

During proof search, Prolog keeps track of *choicepoints*:

-situations where there is *more than one possible match*.

If the chosen path turns out to be a *failure* (or if the user asks for alternative solutions):

-the system can jump back to the *last choicepoint* and try the next alternative.

This process is known as backtracking.

-It is a central feature of Prolog,

-facilitates the concise implementation of many problem solutions.

Backtracking to Choice-points:

Choice-Points:

In `member(X, [a, b, c])`, var `X` can have (match to) 3 choices: `a`,`b`,`c`

Choicepoints are "provided" by any subgoals that can be satisfied with *more than one* match.

Backtracking: during goal execution Prolog *keeps track* of choice-points.

Why: if a certain path fails, it returns to the *last choice-point* and tries the *next* match.

Backtracking is basically a form of searching.

Suppose Prolog is trying to satisfy a sequence of goals: `goal_1, goal_2`.

When the Prolog interpreter finds a *set of var bindings* which allow *goal_1* to be satisfied,

-it *commits* itself to those bindings, and

-then seeks to satisfy *goal_2*.

Eventually one of two things happens:

(a) *goal_2* is satisfied, and finished with; or

(b) *goal_2* cannot be satisfied.

In case, (b) Prolog backtracks.

-It "**un-commits**" itself from the var bindings it made in satisfying *goal_1,* and goes looking for a *different* set of var bindings that allow *goal_1* to be satisfied.

-In case (a) (**;**), if *goal_2* is a *choicepoint*, it uncommits itself from var bindings it made in satisfying *it,* and it looks for a diff set of var bindings for *goal_2,* else it backtracks to *goal_1*, and it uncommits there.

Uses of Backtracking

If it finds a 2nd set of such bindings for *goal_1*, it commits to them, and proceeds to try to satisfy *goal_2* again, with the new bindings from *goal_1*.

- In case (a), the Prolog interpreter is looking for *extra* solutions,
- in case (b) it is *still* looking for the *first* solution.

So backtracking may serve:

- to find extra solutions to a problem, or
- to *continue the search* for a 1st solution, when a 1st set of assumptions (i.e. variable bindings) turns out *not to lead* to a solution.

Inefficiency ?

- (Automatic) backtracking is an important feature of Prolog, but still
- Prolog may waste time/memory exploring possibilities that lead nowhere (inefficiency).

– some backtracking *control* is needed

control backtracking – The Cut

- 2 simple ways to try to control this:

-changing rule order,

-changing goal order.

- The 3rd way : using the built-in Prolog predicate cut "!" (a special atom-written as: **!/0**).
- It can be added as a *goal* to a body of rules. Example :

```
p(X):- b(X), c(X), !, d(X), e(X).
```

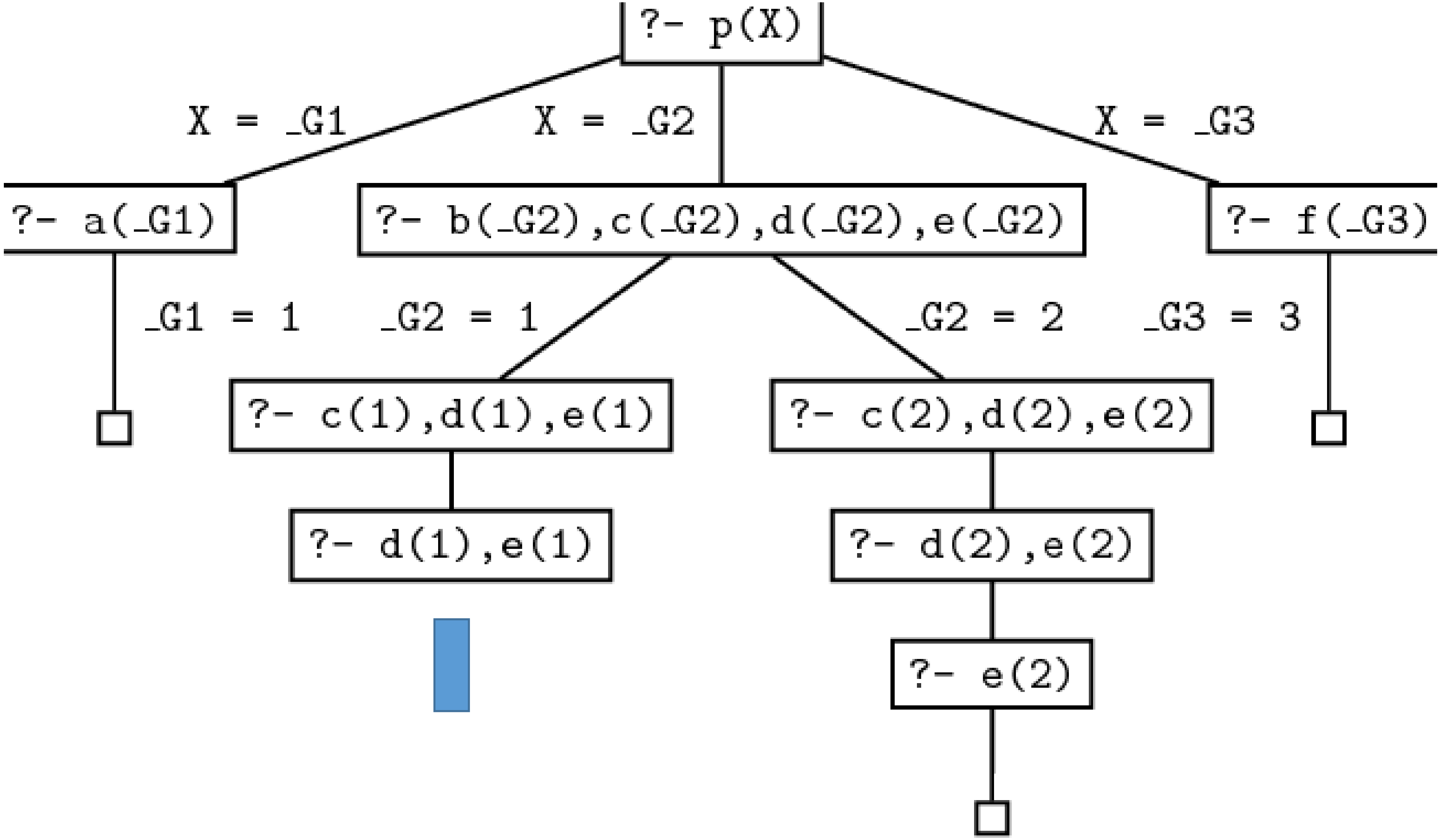
- Cut has no args, and it is a goal that *always succeeds*.
- It commits Prolog to the choices made since the parent goal was called
- It "tells" the system which previous choices need *not to be considered again* when it backtracks.
- It offers a more direct way of controlling the way Prolog *looks for solutions*.

Example without cut

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

a(1).
b(1). b(2).
c(1). c(2).
d(2).
e(2).
f(3).
```

```
?- p(X). % query
X = 1  ; % from a(..)
X = 2  ; % from b,c,d,e(..)
X = 3  ; % from f(..)
false
```

Backtrack tree



adding a cut

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

a(1).
b(1). b(2).
c(1). c(2).
d(2).
e(2).
f(3).
```

```
?- p(X).
X  =  1 ; % from a(..)
false

Reason:there is no d(1)
only d(2)
```

Explanation

- p(X) unifies with the 1st rule, we get a(X) , a(X) unifies with a(1) and we find a solution: **X=1**.
- for a 2nd solution (**;**), p(X) unifies with the 2nd rule, we get the new goals:

  `b(X), c(X), !, d(X), e(X).`

- By instantiating X to 1 , Prolog unifies **b(X)** with the fact **b(1)** , so now we have: **`c(1),!,d(1),e(1).`**
- **`c(1)`** is in the kb, so this simplifies to **!, d(1), e(1)** .
- The **!** goal succeeds (always), and *commits us* to the choices made so far, here: **`X = 1`** , and we are also *committed to using the 2nd rule*.
- But **d(1)** fails. And there's no way we can re-satisfy the goal **p(X)** .

Comment

If we were allowed to try the value X=2, we could use the second rule to generate a solution .

We can't try the value X=2 : the cut has *removed this possibility from the search tree.*

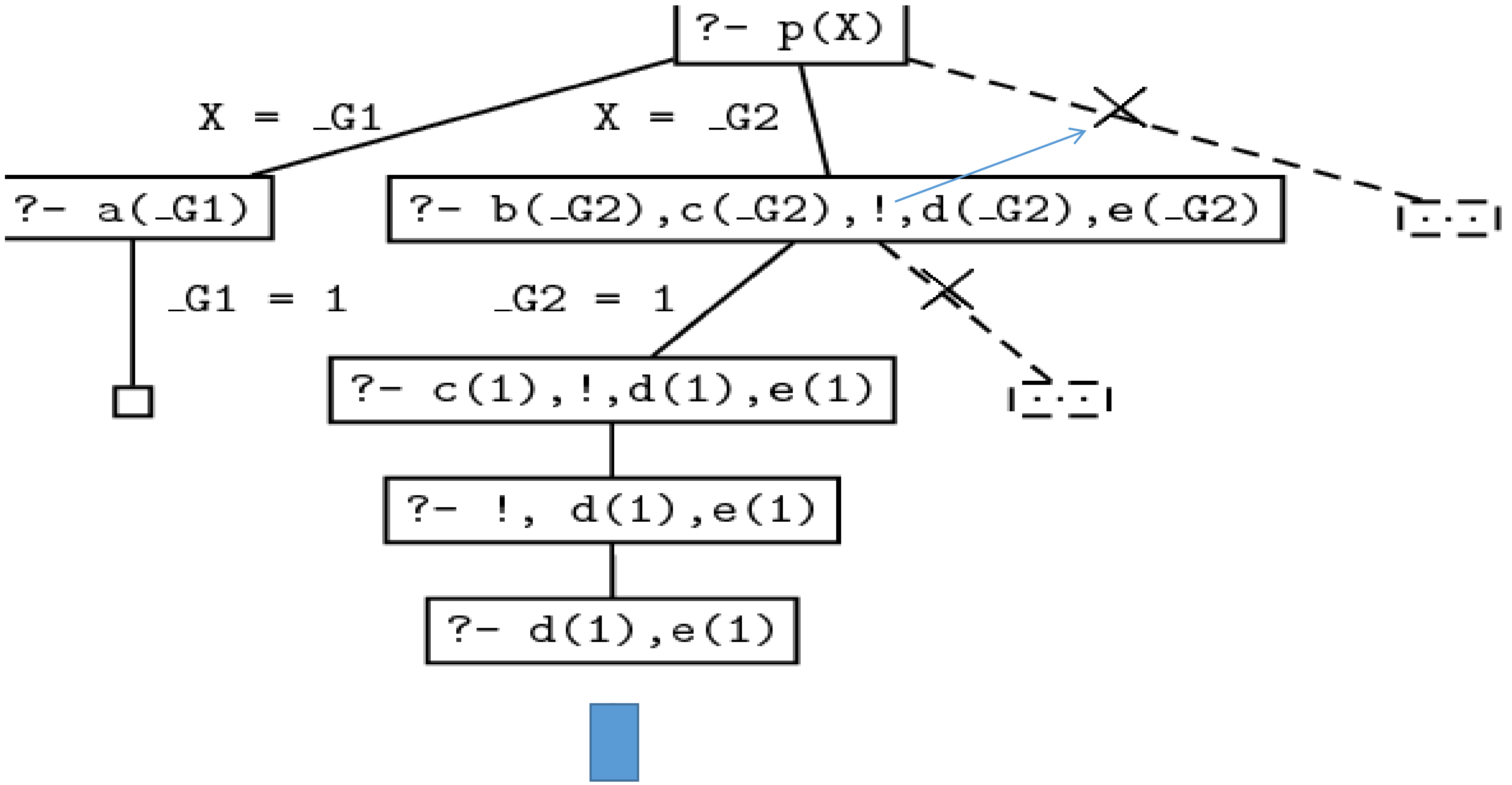
If we were allowed to try the 3rd rule, we could generate the solution X=3 .

But we can't do this either, the cut has removed this possibility too from the search tree.

We are commited (stucked) with X=1 from b(1)..

Search tree with cut

Search stops when the goal **d(1)** doesn't lead to any node where an alternative choice is available. The crosses indicate the branches that the cut *trimmed* away.



What can be done

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

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

- when we reach the cut, it commits us :

-to using *<u>this particular clause of q</u>*, and

-to the choices made when evaluating `p1,...,pn` .

-but NOT to choices made by `r1, … , rn`

However, we are free :

-to backtrack among the `r1,...,rm`, and

-to backtrack among alternatives for choices that were made <u>before reaching the goal q .</u>

How the cut stops searching "down"

```
b(X,1):-X=1.
b(X,2):-X=2.
b(X,Y):-X=5,Y=5.
```

```
?-b(A,B).
A=B,B=1;
A=B,B=2;
A=B,B=5
```

```
b(X,1):-X=1.
b(X,2):-X=2,!.
b(X,Y):-X=5,Y=5.
```

```
?-b(A,B).
A=B,B=1;
A=B,B=2;
No 5's here
```

piece-wise function example

Backtracking, cuts

Backtracking

- Consider a piece-wise function (*exclusive intervals*):
  - `if x < 3, then y = 0`
  - `if x >=3 and x < 6, then y = 2`
  - `if x >= 6, then y = 4`

- In Prolog:

```
f(X,0) :- X < 3.
f(X,2) :- 3 =< X, X < 6.
f(X,4) :- 6 =< X.
```

Backtracking

```
f(X,0) :- X < 3.
f(X,2) :- 3 =< X, X < 6.
f(X,4) :- 6 =< X.
```

- Query:

```
?- f(1,Y), 2<Y.
```

- This matches the `f(X,0)` predicate, which succeeds
  - Y is then instantiated to 0
  - The second part `(2<Y)` causes this query to fail
- Prolog then backtracks and tries the other predicates
  - But the others fail because `X=1,` and they are exclusive

Backtracking

- We want to tell Prolog that if the first one succeeds, there is no need to try the others

  (because they are exclusive)

- We do this with cuts:

```
f(X,0) :- X<3, !.
f(X,2) :- 3 =< X, X<6, !.
f(X,4) :- 6 =< X.
```

- The cut (‘!’) here prevents Prolog from backtracking backwards through the cut

Backtracking

- New Prolog code:

```
f(X,0) :- X<3, !.
f(X,2) :- 3 =< X, X<6, !.
f(X,4) :- 6 =< X.
```

- Note that if the first predicate fails, we know that x >= 3
  - Thus, we don’t have to check it in the second one.
  - Similarly with `x>=6` for the second and third predicates

- Revised Prolog code:

```
f(X,0) :- X<3, !.
f(X,2) :- X<6, !.
f(X,4).
```

Backtracking

- What if we remove the cuts ?

```
f(X,0) :- X<3.
f(X,2) :- X<6.
f(X,4).
```

- Then the following query:

```
?- f(1,X).
```

- Will produce three answers (0, 2, 4) when prompted with **;**

We'll talk later about cuts that when removed, change the meaning of the predicate

Another example

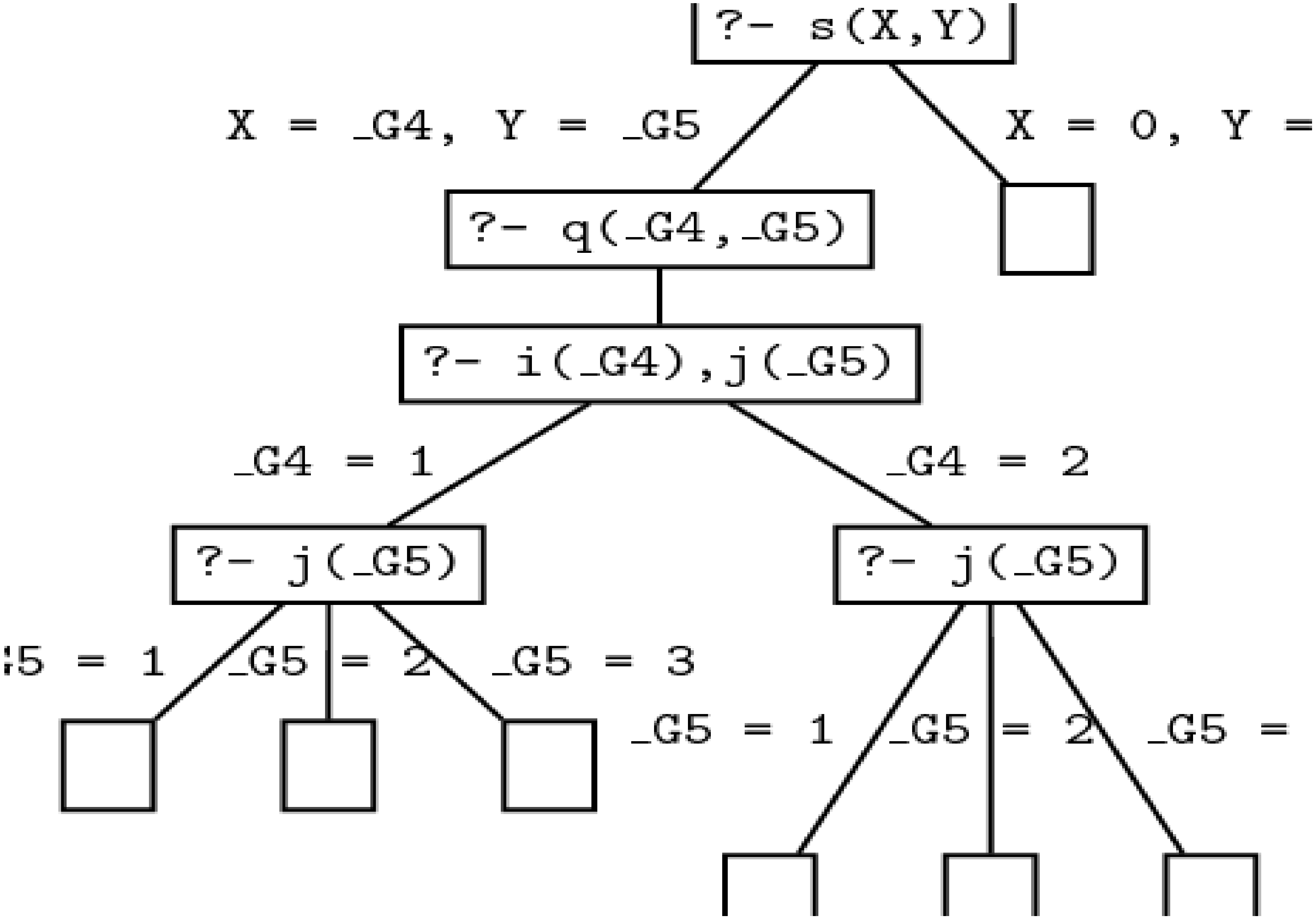
with/without cut

Without cut example

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

i(1). i(2).
j(1). j(2). j(3).
?-  s(X,Y).
```

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

false
```

Search tree



?- s(X,Y)
X = _G4, Y = _G5
X = 0, Y =
?- q(_G4,_G5)
?- i(_G4),j(_G5)
_G4 = 1
_G4 = 2
?- j(_G5)
?- j(_G5)
5 = 1
_G5 = 2
_G5 = 3
_G5 = 1
_G5 = 2
_G5 =

Adding the cut

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

i(1). i(2).
j(1). j(2). j(3).
?-  s(X,Y).
```

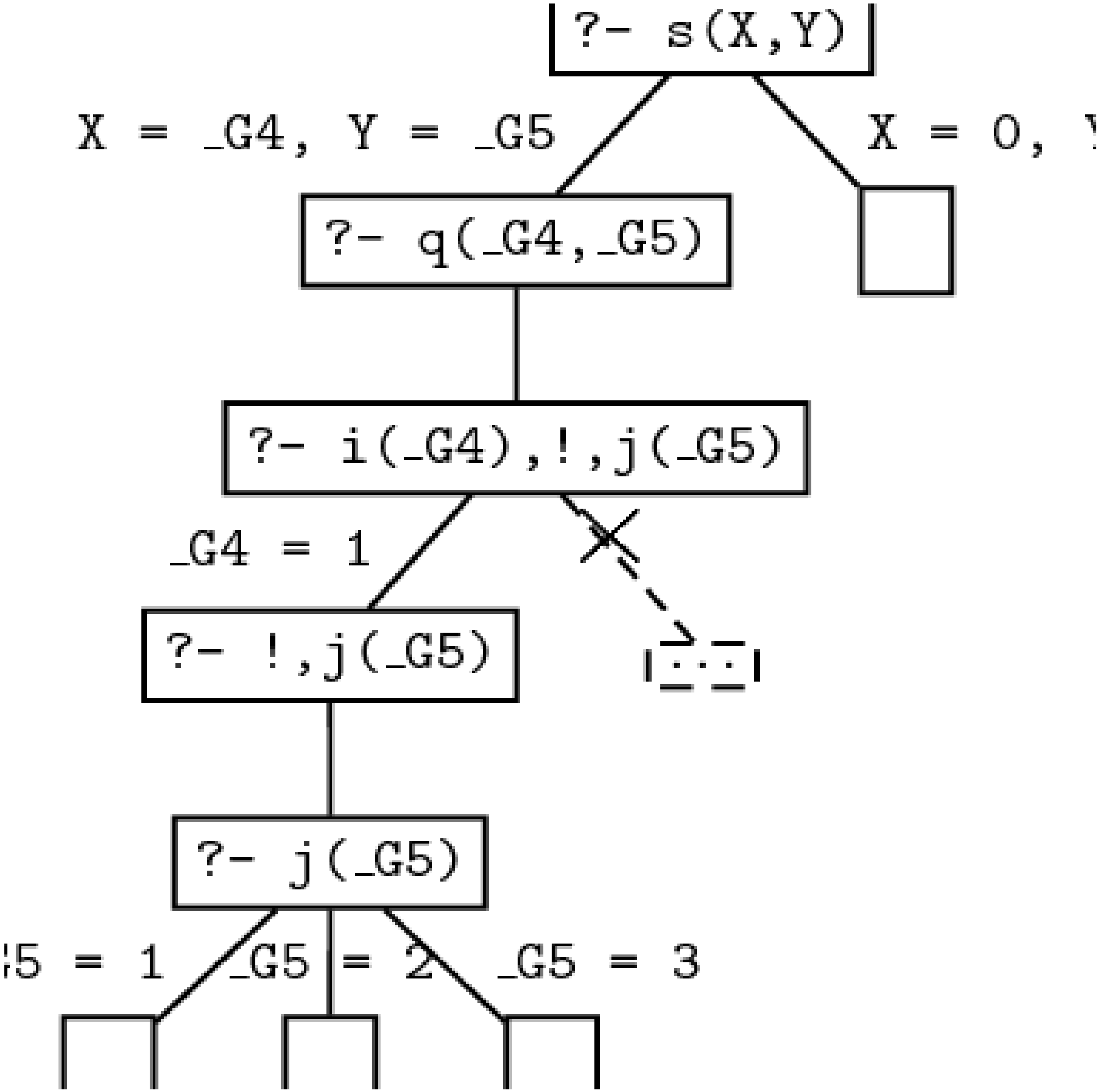
```
X  =  1
Y  =  1  ;

X  =  1
Y  =  2  ;

X  =  1
Y  =  3  ;

X  =  0
Y  =  0;
false
```

notes

- s(X,Y) unifies with the 1st rule; gives us a new goal q(X,Y) .
- q(X,Y) unifies with the 3rd rule, we get the new goals `i(X),!,j(Y)` .
- By instantiating X to 1 , Prolog unifies `i(X)` with the fact `i(1)` .
- This leaves us with the goal `!,j(Y)` . The cut, succeeds, and *commits us* to the choices made so far.
- Choices are: `X = 1` , and that we are using this clause. we have not yet chosen a value for Y .
- Prolog then instantiates Y to 1 , (unifies j(Y) with the fact j(1)) - we found a sol.
- we can find more. try another value for Y . So it backtracks and sets Y to 2 ,

  : a 2nd sol. it can find a 3rd solution: on backtracking again, it sets Y to 3 .
- those are all alternatives for j(X) . Backtracking to the left of the cut is not allowed, so it can't reset X to 2 , so it won't find the next three solutions that the cut-free program found. Backtracking over goals that were reached before q(X,Y) *is allowed* however, so that Prolog will find the second clause for s/2 .

Search tree



max/3 with/without the ~~cat~~ cut

Max/3

Example, a (cut-free) predicate max/3 which takes integers as arguments and succeeds if the 3rd arg is the max of the first two. The queries

```
?-   max(2,3,3).
?-   max(3,2,3).
?-   max(3,3,3).
```

should succeed, and the queries

```
?-   max(2,3,2).
?-   max(2,3,5).
```

should fail.

the program should work when the 3rd arg is a var:

```
 ?-   max(2,3,Max).
Max   =   3
true
?-   max(2,1,Max).
Max = 2
true
```

max/3 without cut

A first attempt:

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

- Problem: a potential inefficiency. the two clauses in the above program are *mutually exclusive*: if the first succeeds, the second must fail and vice versa.
- So attempting to re-satisfy this clause is a waste of time:

```
?- max(3,4,Y).
```

– It will correctly unify Y `with 4`

– But when asked for more solutions, it will try to satisfy the second clause.

which is pointless!

max/3 with cut

- With the cut: Prolog should never try both clauses:

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

- If the X =< Y succeeds, the cut commits us to this choice, and the 2nd clause of max/3 is *not considered*

– If the X =< Y fails, Prolog goes on to the second clause

Green Cuts

- Cuts that *do not change the meaning of a predicate* are called **green cuts**

- The cut in max/3 is an example of a green cut:

– the new code gives exactly the same answers as the old version, but it is *more efficient.*

- *Remove the cut – it still works correctly but maybe not efficiently*

Another max/3 with cut

- Why not remove the *body* of the second clause? After all, it is redundant.
- How good is it?

```
max(X,Y,Y):- X =< Y, !.
max(X,Y,X).
?- max(10,30,X).
X=30
true
?- max(50,20,X).
X=50
true
?- max(20,30,20).
true        % P R O B L E M
```

Revised max/3 with cut

- Unification *after* crossing the *cut* :

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

?-max(2,3,2).
false
```

Red Cuts

- Cuts that change the meaning of a predicate are called **red cuts**
- The cut in the revised max/3 is an example of a red cut:

– If we *take out the cut*, we don't get an equivalent/correct program

- The cut is indispensable to the correct functioning of the program
- the **green** cut merely improved efficiency.

- Programs containing red cuts

– Are not fully declarative *(not Horn clauses)*

– Can be hard to read

– Can lead to subtle programming mistakes

Advice

Try to get a good *cut-free* program working, and then try to improve its efficiency by using cuts.

Use **green** cuts whenever possible.

fail/0

Another build-in predicate: fail/0

- In Prolog it's simple to state *generalizations*.

- To say that Vincent enjoys burgers we write:

```
enjoys(vincent,X) :- burger(X).
```

- But rules may have exceptions. Perhaps Vincent doesn't like Big Kahuna burgers. Perhaps the correct rule is :

- *Vincent enjoys burgers, **except** Big Kahuna burgers*.

fail/0

- `fail/0` is a special symbol that *fails* immediately when Prolog encounters it as a goal.

- when Prolog fails, it tries to backtrack .

- `fail/0` can be viewed as an instruction to *force backtracking*.

- When used *in combination with cut, which blocks backtracking, fail/0 enables us to define <u>exceptions to general rules</u>.*

The cut– fail combination

```
enjoys(vincent,X):-
      bigKahunaBurger(X), !, fail.

enjoys(vincent,X):- burger(X).
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigMac(c).

bigKahunaBurger(b).
whopper(d).
```

```
?-enjoys(vincent,a).
true

?-enjoys(vincent,b).
false

?-enjoys(vincent,c).
true

?-enjoys(vincent,d).
true
```

why it's not ideal

the ordering of the rules is crucial:

-if we reverse the first two lines, we don't get the behavior we want.

the cut is crucial:

-if we remove it, the program doesn't behave in the same way (is a red cut).

Important observation - the rule:

```
enjoys(vincent,X):-
      bigKahunaBurger(X), !, fail.
```

is a way of saying that Vincent *does not enjoy X if X is a Big Kahuna burger.*

Negation-as-Failure

- The *cut-fail* combination lets us define a form of negation
- It is called ***negation-as-failure*, and** defined as follows:

```
neg(Goal):- Goal, !, fail.
neg(Goal).
```

For any Prolog goal, **neg(Goal)** will succeed *if Goal* ***does not*** *succeed.*

Using *neg/1* predicate, we can describe Vincent's preferences in a *clearer* way:

```
enjoys(vincent,X)  :-  burger(X),
                       neg(big_kahuna_burger(X)).
```

:Vincent enjoys X if ***X is a burger,*** and ***X is not a Big Kahuna burger***

Negation-as-failure is not logical negation

- *Negation-as-failure* it comes built-in as part of standard Prolog, so we don't have to define it at all.
- In standard Prolog the operator **\+** means negation-as-failure, so we could define Vincent's preferences as follows:

```
enjoys(vincent,X) :- burger(X), \+ big_kahuna_burger(X).
?- enjoys(vincent,X).
X=a; X=c; X=d;
```

Negation-as-failure is ***not logical negation***

Changing the order of the prev goals gives *a different behaviour*:

```
enjoys(vincent,X):- \+ bigKahunaBurger(X), burger(X).
?- enjoys(vincent,X). % false
```

Negation-as-failure

in the 1st version we use **\+** only after we have instantiated the variable X :

```
enjoys(vincent,X) :- burger(X), \+ big_kahuna_burger(X).
```

(it works right)

In the 2nd version we use **\+** before the instantiation of X:

```
enjoys(vincent,X):- \+ bigKahunaBurger(X), burger(X).
```

(it goes wrong)

The difference is crucial.

It is generally a better idea to try use *negation-as-failure* than to write code containing heavy use of red cuts.

Example

- write code for the following condition:

p holds if a and b hold, or, if a does not hold and c holds too .

This can be captured directly with the help of **negation-as-failure** :

```
p :- a,b.
p :- \+ a, c.
```

- But if a is a very complicated goal, that takes a lot of time to compute, we do not want to compute **a** twice, so this prog looks better:

```
p :- a,!,b.    % a computed once
P :- c.
```

- Note that this is a red cut: removing it changes the meaning of the program.

Negation in Prolog

```
?- \+ member(X, [a,b,c]).
false
```

- It might look like this query is asking "*Does there exist an X for which `member(X, [a,b,c])` does not succeed?*".
  - We know there are lots of values of X for which `member(X, [a,b,c])` does not succeed.
  - But that's not what negation means in Prolog.
  - There exists X for which `member(X, [a,b,c])` succeeds.
  - So then `\+ member(X, [a,b,c])` fails.

Isn't anyone sad?

```
sad(X) :- \+ happy(X).
happy(X):-beautiful(X), rich(X).
rich(beni).
beautiful(eti).
rich(eti).
beautiful(silvi).
```

Isn't anyone sad?

No, that just means *that it's not true we can not find anyone happy.*

– In other words, there exists someone who is happy. (eti..)

```
?- sad(beni).
true
?-sad(silvi).
true
?- sad(eti).
false
?- sad(iosi).
true
?- sad(Mishu).
false
```

Naf 1

```
likes(iosi, cheese).
likes(ety, cheese).
likes(eli, fish).
friend(X, Y) :- \+(X = Y),
    likes(X, Z), likes(Y, Z).
```

“iosi and ety like cheese; eli likes fish”

and

“Two different things are friends if they both like at least one of the same thing”.

?- likes(iosi, cheese). %true

?- friend(iosi, ety). %true

?- friend(iosi, iosi). %false

?- likes(iosi, **Q**).

Q = cheese

true

?- likes(**Q**, cheese).

Q = iosi ;

Q = ety

false

?- friend(iosi, **Q**).

false

Naf 2 improvement

```
likes(iosi, cheese).
likes(ety, cheese).
likes(eli, fish).
friend(X, Y) :-likes(X, Z),
       likes(Y, Z), \+(X = Y).
```

Changing places

```
?- friend(iosi, Q).
Q=ety
false
```

\+ is Not Logical Negation

The predicate *friends* produces the correct output after reordering its goals

The predicate \+(M) is implemented as "negation as failure" (NAF).

1. this predicate attempts to prove the goal M.
2. If this *can be satisfied*, then the predicate makes a cut.

Cuts freeze all variable assignments made in a rule, from the beginning up to the point at which the cut appears.

3. Finally, the predicate fails explicitly.

All of this machinery effectively causes \+(M)to be false if M is true.

- The incorrect behavior in the first program is due to the cut.

The M goal in \+(M) contains *free vars*, and it is easy to find an assignment for those vars such that M is true. The cut prevents any other assignments of these vars to atoms, and therefore *\+(M) is always false* and the rule can never succeed.

Trace 1

- The 1st time, Prolog does that:
- `Goal 1: friend(iosi, Y)`
- `Goal 2: \+(iosi = Y)`
- `Goal 3: iosi = Y`
- `Succeed 3: Y <-- iosi`
- `Cut and prevent reassignment of Y`
- Fail 2 with no possibility of success because \+(iosi = (**Y** = iosi)) is always false
- `Fail 1`

Trace 2

By moving the negation predicate to the end of the rule, Prolog will search for variable assignments for the variables in the goal M, **before** the negation predicate appears:

- `Goal 1: friend(iosi, `**`Y`**`)`
- `Goal 2: likes(iosi, `**`Z`**`)`
- `Succeed 2: `**`Z`**` <-- cheese`
- `Goal 3: likes(`**`Y`**`, cheese)`
- `Succeed 3: `**`Y`**` <-- iosi`
- `Goal 4: \+(iosi = iosi)`
- `Fail 4, with a similar reasoning as above`
- `Backtrack to 3 (`*cuts don't freeze variables set in other goals*`)`
- `Goal 3: likes(`**`Y`**`, `**`cheese`**`)`
- `Succeed 3: `**`Y`**` <-- ety`
- `Goal 4: \+(iosi = ety)`
- `Succeed 4, because iosi = ety is false`
- `Succeed 1, since we've succeeded in all sub-goals`

neg(Goal):- Goal, !, fail.
neg(Goal).

Common uses of cut

Three main cases:

1. To tell the system that *it found the right rule* for a particular goal. Confirming the choice of a rule.
2. **Cut-fail** combination: to tell the system to *fail a particular goal without trying for alternative* solutions.
3. To tell the system to terminate the generation of alternative solutions by backtracking. Terminate a "generate-and-test“ style of work.

Advantages

The program will run faster.

No time wasting on attempts to re-satisfy certain goals.

The program will occupy less memory.

Less backtracking points to be remembered.

Exercises

Ex 10.1

Exercise 10.1 Suppose we have the following database:

```
p(1).
p(2) :- !.
p(3).
```

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

X = 1;
X = 2; % false

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

X = 1
Y = 1;

X = 1
Y = 2;

X = 2
Y = 1 ;

X = 2
X = 2; % false

```
?- p(X),!,p(Y).
```

X=1
Y=1;

X=1
Y=2; % false

Exercise 10.2

Exercise 10.2

1. explain what the following

program does:

```
class(Number,positive) :- Number > 0.
class(0,zero).
class(Number, negative) :- Number < 0.
```

Call:

```
?-class(5,X).
X=positive
…
```

The program determines the "sign-class" of a Number:

positive if greater than 0,

zero if equal to 0, or

negative if less than 0.

2. Second, improve it by adding green cuts.

```
class(Number,positive) :- Number > 0.
class(0,zero).
class(Number, negative) :- Number < 0.
```

```
class(Number,positive) :- Number > 0, !.
class(0,zero) :- !.
class(Number,negative) :- Number < 0, !.
```

! forbids backtrackings, like these done with ;

10.3

Without using cut, write a predicate `split/3` that splits a list of integers into two lists: one containing the positive ones (and zero), the other containing the negative ones. example:

```
split([3,4,-5,-1,0,4,-9],P,N)
```

should return:

```
P = [3,4,0,4]
N = [-5,-1,-9].
```

Then improve this program, without changing its meaning, with the help of cut.

```
split([],[],[]).
split([X|Xs],[X|P],N):- X >= 0,
                        split(Xs,P,N).
split([X|Xs],P,[X|N]):- X < 0,
                        split(Xs,P,N).
```

```
split([],[],[]) :- !.
split([X|Xs],[X|P],N):- X >=0, !,
                            split(Xs,P,N).
split([X|Xs],P,[X|N]):- X < 0, !,
                            split(Xs,P,N).
```

Practice 10.1

Define a predicate nu/2 ("not unifiable") which takes two terms as args and succeeds if the two terms *do not unify*. For example:

```
nu(foo,foo).    % false
nu(foo,blob).   % true
nu(foo,X).      % false
```

You should define this predicate in three different ways:

1st write it with the help of = and \+.

2nd write it with the help of =, but without \+.

3rd write it using a cut-fail combination.

don't use = and don't use \+.

```
nu(X,Y) :- \+ (X=Y).
```

```
Remember neg:
neg(Goal) :- Goal,!,fail.
neg(Goal).
```

```
nu(X,Y) :- neg(X = Y).
```

```
nu(X,Y):- X \= Y.
```

```
Or:
nu(X,X) :- !,fail.
nu(_,_) :- !.
```

practice 10.2 unifiable predicate

Define a predicate **unifiable(List1,Term,List2)** where List2 is the list of all members of List1 that match Term, but are not instantiated by the matching. For example:

`unifiable([X,b,t(Y)],t(a),List]).`

should yield `List = [X,t(Y)].`

Note that X and Y are *still not instantiated*. So the tricky part is: how do we check that they match with `t(a)` without instantiating them? Hint: consider using the test

`\+ (term1 = term2).`

Also think about the Test

`\+(\+ (term1 = term2)).`

```
unifiable([],_,[]).

unifiable([X|Xs],Term,[X|Result]) :-
            \+(\+ X=Term),
            unifiable(Xs,Term,Result).

unifiable([X|Xs],Term,Result) :-
            \+ X=Term,
            unifiable(Xs,Term,Result).
```

```
?- \+ \+ X=5, Y=X.
X = Y. % where is 5

?- \+ X=5,Y=X.
false.
```

6.1

- **Exercise 6.1**

Let's call a list *doubled* if it is made of two consecutive blocks of elements that are exactly the same.

For example, [a,b,c,a,b,c] is doubled (it's made up of [a,b,c] followed by [a,b,c]) and so is [foo,gubble,foo,gubble].

On the other hand, [foo,gubble,foo] is not doubled.

Write a predicate `doubled(List)` which succeeds when List is a doubled list.

```
doubled(List) :- append(X,X,List).
```

```
?-doubled(L).
L = []
L = [_G3446, _G3446]
L = [_G3446, _G3452, _G3446, _G3452]
```

```
?- doubled([1,2|X]).
X = [1, 2]
X = [_G3689, 1, 2, _G3689]
X = [_G3689, _G3695, 1, 2, _G3689, _G3695] .
```

6.2 – palindrome

Write a predicate palindrome(List), which checks whether List is a palindrome. For example, to the queries

?- palindrome([r,o,t,a,t,o,r]).

and

?- palindrome([n,u,r,s,e,s,r,u,n]).

Prolog should respond 'true', but to the query

?- palindrome([n,o,t,h,i,s]).

Prolog should respond 'false'.

```
palindrome(List) :-
        reverse(List,List).
?- palindrome(X).
X = []
X = [_G5146]
X = [_G5146, _G5146]
X = [_G5146, _G5152, _G5146] .

?- palindrome([1,2|X]).
X = [1]
X = [2, 1]
X = [_G5200, 2, 1]
X = [_G5200, _G5200, 2, 1] .
```

6.3.1 - second

Write a predicate `second(X,List)` which checks whether X is the second element of `List`.

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

?- second(X,Y).
Y = [_G3759, X|_G3763].

?- second(X,[1]).
false.

?- second(X,[1|Y]).
Y = [X|_G3793].

?- second(a,Y).
Y = [_G3747, a|_G3751].
```

6.3.2 - swap12

Write a predicate `swap12(List1,List2)` which checks whether `List1` is identical to `List2`, except that the first two elements are exchanged.

```
swap12([X,Y|T],[Y,X|T]).
```

```
?- swap12(X,Y).
X = [_G3762, _G3765|_G3766],
Y = [_G3765, _G3762|_G3766].

?- swap12([a,b,c|X],Y).
Y = [b, a, c|X].

?- swap12(X,[1,A|B]).
X = [A, 1|B].
```

6.3.3 - final

Write a predicate `final(X,List)` which checks whether X is the last element of List.

```
final(X,List) :-
        reverse(List,[X|_]).
```

```
?- final(X,Y).
Y = [X]
Y = [_G5148, X]
Y = [_G5148, _G5154, X] .
```

6.3.4-toptail

Write a predicate toptail(InList,Outlist) which is ‘false’ if Inlist is a list containing fewer than 2 elements, and which deletes the first and the last elements of Inlist and returns the result as Outlist, when Inlist is a list containing at least 2 elements, example:

```
toptail([a],T).      % false
toptail([a,b],T).    % T=[]
toptail([a,b,c],T). % T=[b]
```

```
toptail([_|Xs],Outlist) :-
      append(Outlist,[_],Xs).
```

6.3.5 – swap first last

```
swapfl([X|Xs],List2) :-
        append(T,[H],Xs),
        append([H|T],[X],List2).
```

Write a predicate swapfl(List1,List2) which checks whether List1 is identical to List2, except that the *first* and *last* elements are exchanged. Hint: here's where append comes in useful again.

Database Manipulation

Database Manipulation

Prolog has five basic knowledge-base manipulation commands:

Adding info:

- `assert/1`
- `asserta/1`
- `assertz/1`

`Removing info`

- `retract/1`
- `retractall/1`

assert

Start with an empty kb:

```
?- listing.
true

?- assert(happy(mike)).
true
?- listing.
happy(mike).

?- assert(happy(iosi)),assert(happy(ety)),
              assert(happy(moshe)), assert(happy(joe)).
true
```

Changing meaning of predicates

The database manipulations have changed the meaning of the predicate `happy/1`

kb manipulation commands give us the ability to *change the meaning of predicates during runtime.*

Predicates which meaning change during runtime are called dynamic predicates

– `happy/1` is a dynamic predicate

Ordinary predicates are sometimes referred to as static predicates

Asserting in specific places, asserting rules

To place the asserted material at the *beginning* of the kb, use:

```
- asserta/1
```

Place at the *end* of the kb:

```
- assertz/1
```

Assert a rule:

```
?- assert( (naive(X):- happy(X)) ).
true
```

Removing information

– using the `retract/1` predicate, will remove *one* clause

– We can remove several clauses simultaneously with the `retractall/1` predicate :

```
?- retract(happy(joe)).
true
?- retract(happy(iosi)).
true

Retracting all happy/1
?- retract(happy(X)).
X=mike;
X=iosi;
X=ety;…
false
```

Memoisation/caching

a useful technique for storing the results to computations, in case we need to recalculate the same query.

Example of memoisation

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
         lookup(X,Y,Res),!.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

lookup(3,7,100).
lookup(3,4,49).
```

```
?- addAndSquare(3,7,X).
X=100
true

?- addAndSquare(3,4,X).
X=49
true

?-retractall(lookup(_, _, _)).
true
```

Collecting Solutions

Collecting Solutions

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- descend(martha,X).
X=charlotte;
X=caroline;
X=laura;
X=rose;
false
```

Collecting Solutions to a query

Prolog generates solutions one by one.

Prolog has three built-in predicates to get all the solutions in one go:

`findall/3`,
`bagof/3`
`setof/3`

All these predicates can collect all the solutions and put them into a single list.

(there are some differences between them)

findall/3

The query

```
?- findall(O,G,L).
```

produces a list L of all the objects O that satisfy the goal G

– Always succeeds

– if G *cannot* be satisfied, L is unified with the empty list

findall

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- findall(X,descend(martha,X),L).
L=[charlotte,caroline,laura,rose]
true
?- findall(X,descend(rose,X),L).
L=[ ]
true
?- findall(d,descend(martha,X),L).
L=[d,d,d,d]
true
?- findall(X,descend(Y,X),L).
L=[charlotte,caroline,laura, rose,
caroline,laura,rose,laura,rose,rose]
true
```