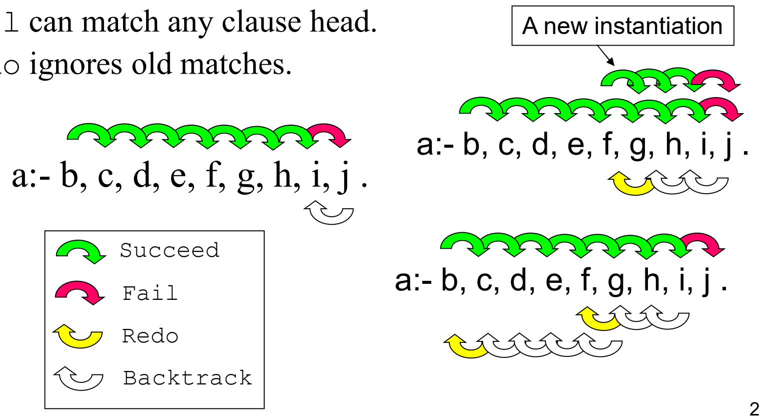
Cut Operator in Prolog

Logic in Computer Science

Prolog’s Persistence

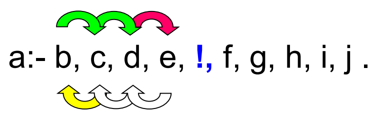
- When a subgoal fails, Prolog will backtrack to the most recent successful goal and try to find another solution.
- Once there are no more solutionss for this subgoal it will backtrack again; retrying every subgoal before failing the parent goal.
- A `call` can match any clause head.
- A `redo` ignores old matches.

```
b :- c.
b :- d.
c.
d.
e.
f :- g.
f :- h.
g.
h.
i.
```

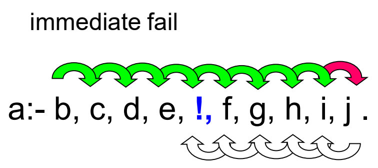

A new instantiation
a:- b, c, d, e, f, g, h, i, j .
a:- b, c, d, e, f, g, h, i, j .
a:- b, c, d, e, f, g, h, i, j .
Succeed
Fail
Redo
Backtrack

Cut !

- If we want to restrict backtracking we can control which sub-goals can be redone using the cut **!**.
- We use it as a predicate within the body of clause.
- It succeeds when `called`, but `fails` the parent goal (the goal that matched the head of the clause containing the cut) when an attempt is made to `redo` it on backtracking.
- It commits to the choices made so far in the predicate.
  - unlimited backtracking can occur before and after the cut but no backtracking can go through it.

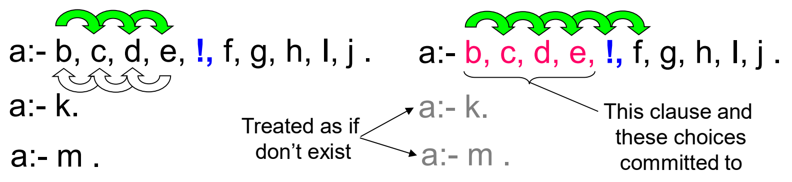


```
a:- b, c, d, e, !, f, g, h, i, j .
```



```
a:- b, c, d, e, !, f, g, h, i, j .
```

Failing the parent goal



```
a:- b, c, d, e, !, f, g, h, I, j .     a:- b, c, d, e, !, f, g, h, I, j .
a:- k.                                 a:- k.
a:- m .                                a:- m .
```

- The cut succeeds when it is `called` and commits the system to all choices made between the time the parent goal was invoked and the cut.
- This includes committing to the clause containing the cut.
  - = the goal can only succeed if this clause succeeds.
- When an attempt is made to backtrack through the cut
  - the clause is immediately failed, and
  - no alternative clauses are tried.

Mutually Exclusive Clauses

- We should only use a cut if the clauses are mutually exclusive (if one succeeds the others won't).
- If the clauses are mutually exclusive then we don't want Prolog to try the other clauses when the first fails
  = redundant processing.
- By including a cut in the body of a clause we are committing to that clause.
  - Placing a cut at the start of the body commits to the clause as soon as head unification succeeds.
    ```
    a(1,X):- !, b(X), c(X).
    ```
  - Placing a cut somewhere within the body (even at the end) states that we cannot commit to the clause until certain sub-goals have been satisfied.
    ```
    a(_,X):- b(X), c(X), !.
    ```

5

5

Mutually Exclusive Clauses (2)

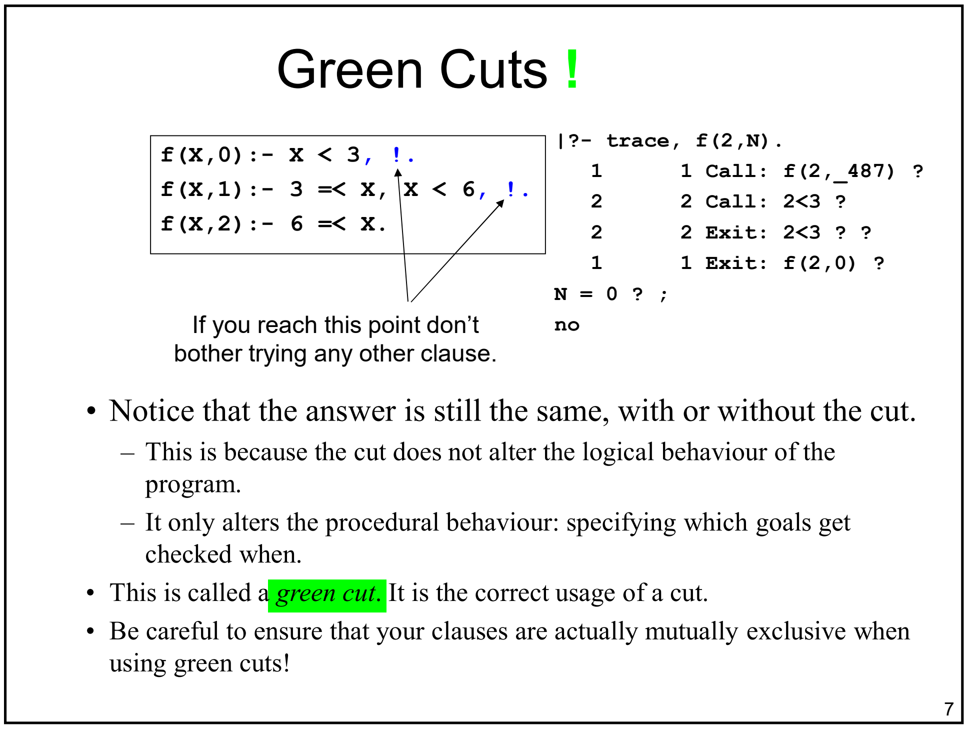
Example: Classify numbers into three classes:

```
f(X,0):- X < 3.
f(X,1):- 3 =< X, X < 6.
f(X,2):- 6 =< X.
```

```
|?- trace, f(2,N).
  1      1 Call: f(2,_487) ?
  2      2 Call: 2<3 ?
  2      2 Exit: 2<3 ? ?
  1      1 Exit: f(2,0) ?
N = 0 ? ;
  1      1 Redo: f(2,0) ?
  3      2 Call: 3=<2 ?
  3      2 Fail: 3=<2 ?
  4      2 Call: 6=<2 ?
  4      2 Fail: 6=<2 ?
  1      1 Fail: f(2,_487) ?
no
```

6

6



Green Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- 3 =< X, X < 6, !.
f(X,2):- 6 =< X.
```

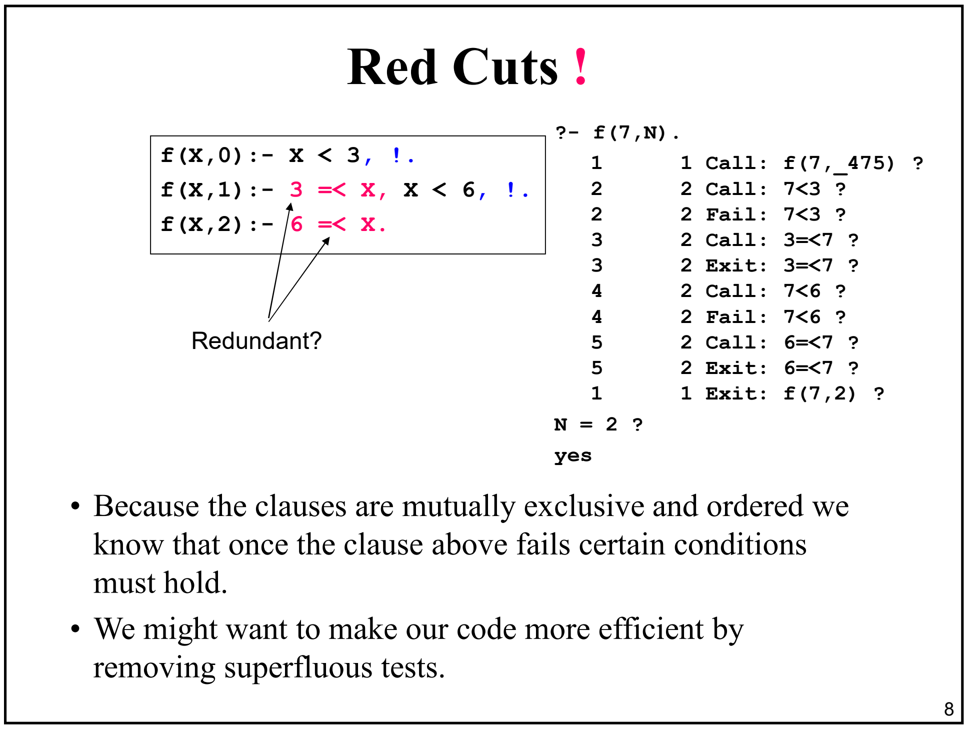
If you reach this point don't bother trying any other clause.

```
|?- trace, f(2,N).
   1      1 Call: f(2,_487) ?
   2      2 Call: 2<3 ?
   2      2 Exit: 2<3 ? ?
   1      1 Exit: f(2,0) ?
N = 0 ? ;
no
```

- Notice that the answer is still the same, with or without the cut.
  - This is because the cut does not alter the logical behaviour of the program.
  - It only alters the procedural behaviour: specifying which goals get checked when.
- This is called a *green cut.* It is the correct usage of a cut.
- Be careful to ensure that your clauses are actually mutually exclusive when using green cuts!

7

7



Red Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- 3 =< X, X < 6, !.
f(X,2):- 6 =< X.
```

Redundant?

```
?- f(7,N).
   1      1 Call: f(7,_475) ?
   2      2 Call: 7<3 ?
   2      2 Fail: 7<3 ?
   3      2 Call: 3=<7 ?
   3      2 Exit: 3=<7 ?
   4      2 Call: 7<6 ?
   4      2 Fail: 7<6 ?
   5      2 Call: 6=<7 ?
   5      2 Exit: 6=<7 ?
   1      1 Exit: f(7,2) ?
N = 2 ?
yes
```

- Because the clauses are mutually exclusive and ordered we know that once the clause above fails certain conditions must hold.
- We might want to make our code more efficient by removing superfluous tests.

8

8

Red Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- X < 6, !.
f(X,2).
```

```
f(X,0):- X < 3.
f(X,1):- X < 6.
f(X,2).
```

```
?- f(7,N).
   1      1 Call: f(7,_475) ?
   2      2 Call: 7<3 ?
   2      2 Fail: 7<3 ?
   3      2 Call: 7<6 ?
   3      2 Fail: 7<6 ?
   1      1 Exit: f(7,2) ?
N = 2 ?
yes
```

```
?- f(1,Y).
   1      1 Call: f(1,_475) ?
   2      2 Call: 1<3 ?
   2      2 Exit: 1<3 ? ?
   1      1 Exit: f(1,0) ?
   Y = 0 ? ;
   1      1 Redo: f(1,0) ?
   3      2 Call: 1<6 ?
   3      2 Exit: 1<6 ? ?
   1      1 Exit: f(1,1) ?
   Y = 1 ? ;
   1      1 Redo: f(1,1) ?
   1      1 Exit: f(1,2) ?
   Y = 2 ?
   yes
```

9

9

Using the cut

- *Red cuts* change the logical behaviour of a predicate.
- TRY NOT TO USE RED CUTS!
- Red cuts make your code hard to read and are dependent on the specific ordering of clauses (which may change once you start writing to the database).
- If you want to improve the efficiency of a program use *green cuts* to control backtracking.
- Do not use cuts in place of tests.

To ensure a logic friendly cut either:

```
p(X):- test1(X), !, call1(X).
p(X):- test2(X), !, call2(X).
p(X):- testN(X), !, callN(X).
```

`testI` predicates are mutually exclusive.

```
p(1,X):- !, call1(X).
p(2,X):- !, call2(X).
p(3,X):- !, callN(X).
```

The mutually exclusive tests are in the head of the clause.

10

10

A larger example

We'll define several versions of the disjoint partial map split.

split(*list of integers*, *non-negatives*, *negatives*).

1. A version not using cut. Good code (each can be read on its own as a fact about the program). Not efficient because choice points are retained. The first solution is desired, but we must ignore backtracked solutions.

```
split([], [], []).
split([H|T], [H|Z], R) :- H >= 0, split(T, Z, R).
split([H|T], R, [H|Z]) :- H < 0, split(T, R, Z).
```

11

11

A larger example

2. A version using cut. Most efficient, but not the best 'defensively written' code. The third clause does not stand on its own as a fact about the problem. As in normal programming languages, it needs to be read in context. This style is often seen in practice, but is deprecated.

```
split([], [], []).
split([H|T], [H|Z], R) :- H >= 0, !, split(T, Z, R).
split([H|T], R, [H|Z]) :- split(T, R, Z).
```

Is this read cut or green cut?

Minor modifications (adding more clauses) may have unintended effects. Backtracked solutions invalid.

12

12

A larger example

3. A version using cut which is also ‘safe’. The only inefficiency is that the goal H < 0 will be executed unnecessarily whenever H < 0.

```
split([], [], []).
split([H|T], [H|Z], R) :- H >= 0, !, split(T, Z, R).
split([H|T], R, [H|Z]) :- H < 0, split(T, R, Z).
```

Recommended for practical use. Hidden problem: the third clause does not capture the idea that H < 0 is a committal. Here committal is the default because H < 0 is in the last clause. Some new compilers detect that H < 0 is redundant.

13

13

A larger example

4. A version with unnecessary cuts

```
split([], [], []) :- !.
split([H|T], [H|Z], R) :- H >= 0, !, split(T, Z, R).
split([H|T], R, [H|Z]) :- H < 0, !, split(T, R, Z).
```

First cut unnecessary because anything matching first clause will not match anything else anyway. Most Prolog compilers can detect this.

Why is the third cut unnecessary? Because H<0 is in the last clause. Whether or not H<0 fails, there are no choices left for the caller of split. However, the above will work for any order of clauses.

14

14

The Cut (!)

- The one and only '!'
  - There are GOOD, BAD and Ugly ones (usages).
  - GREEN and RED ones (usages).
- Goals before a cut produce first set and only the first set of bindings for named variables
  - Commits a choice
  - No alternative matches considered upon backtracking.
- **Green Cuts**
  - Exclude clauses (solution attempts), but NOT solutions.
  - Removal of Cut does NOT change the meaning of the program. The cut's positioning just effects efficiency.
- **Red Cuts**
  - Alter the actual meaning of the program.

15

15

A Good Red Cut

```
if_then_else(If,Then,Else) :-
   call(If), !, call(Then).

if_then_else(If, Then, Else) :-
   call(Else).
```

```
?- if_then_else(true, write(equal),
   write(not_equal))
equal
yes.
?- if_then_else(false, write(equal),
   write(not_equal)) not_equal
yes.
```

If we take out the cut we change the meaning -- so the cut is RED.
But it is used to produce the meaning we want -- so the cut is GOOD.

```
if_then_else(If,Then,Else) :-
    call(If), call(Then).

if_then_else(If,Then,Else) :-
    call(Else).
```

```
?- if_then_else(true, write(equal),
     write(not_equal))

equal
not_equal
yes.
```

16

16

A Bad Red cut

- min(N1, N2, N1) :- N1<N2, !.
- min(_, N2, N2).

17

17

A BAD Red Cut

```
R1. pension(X,disabled) :- disabled(X), !.
R2. pension(X,senior) :- over65(X), paid_up(X), !.
R3. pension(X,supplemental) :- over65(X), !.
R4. pension(X,nothing).   %"The Default" If everything else fails.
```

The cut is used to implement the *default* case -- Yike!

```
F1.  disabled(joe).      F4. over65(lou).
F2.  over65(joe).        F5. paid_up(lou).
F3.  paid_up(joe).
```

Q1. ?- pension(joe, nothing) -> yes.

OOPS! "I'm sorry Mr. Joe...yes Mr. Joe you are entitled, it was a small computer error...really Mr. Joe computers DO make mistakes...I'm sorry what was that about intended meaning?".

Q2. ?- pension(joe, P) -> P = disabled

Does Joe get more than one pension payment?

Q3. ?- pension(X, senior) -> X = joe.

What happened to Lou's pension? Isn't he a senior?

18

18

Joe's Revenge

```
R1. pension(X,disabled_pension) :- disabled(X).
R2. pension(X,senior_pension) :- over65(X), paid_up(X).
R3. pension(X,supplemental_pension) :- over65(X).
R4. entitled(X,Pension) :- pension(X,Pension).
R5. entitled(X,nothing) :- \+(pension(X,Pension)).
```

```
F1. disabled(joe).   F4. over65(lou).
F2. over65(joe).     F5. paid_up(lou).
F3. paid_up(joe).
```

```
Q1. ?- entitled(joe,nothing)  -> no.

Q2. ?- entitled(joe, P)  -> 1. P = disabled, 2. P=senior, 3. P=supplemental

Q3. ?- entitled(X,senior_pension) -> 1. X = joe  2. X = lou

Q4. ?- entitled(X,disabled_pension) -> 1. X = joe.
```

19

19

Exercise Question

Given the following Prolog program for reverse:

```
rev([], []).
rev([A|B], C) :- rev(B, D), append(D, [A], C).
```

and the query

```
?- rev(X, [a, b]).
```

Please provide the corresponding resolution proof for the first output of this query. Can you get the second output? Why?

20

20

Negation as Failure

Using cut together with the built-in predicate fail, we may define a kind of negation, so that properties that cannot be specified by Horn clauses can be specified.

Examples: Mary likes any animals except reptiles:

```
likes(mary, X) :- reptile(X), !, fail.
likes(mary, X) :- animal(X).
```

A utility predicate meaning something like "not equals":

```
different(X, X) :- !, fail.
different(_,_).
```

Negation as Failure

We can use the same idea of "cut/fail" to define the predicate not, which takes a term as an argument. not will "call" the term, that is evaluate it as though it is a goal:

not(G) fails if G succeeds

not(G) succeeds if G does not succeed.

In Prolog,

```
not(G) :- call(G), !, fail.
not(_).
```

call is a built-in predicate which acts like: call(P) :- P.

Negation as Failure

Most Prolog systems have a built-in predicate like not. SICStus Prolog calls it \+. Remember, "not" does not correspond to logical negation, because it is based on the success/failure of goals. It can, however, be useful:

```
likes(mary, X) :- not(reptile(X)).

different(X, Y) :- not(X = Y).
```

23

23

Negation as Failure can be Misleading

Once upon a time, a student who missed some of these lectures was commissioned to write a Police database system in Prolog. The database held the names of members of the public, marked by whether they are innocent or guilty of some offence.

Suppose the database contains the following:

```
innocent(peter_pan).
innocent(X) :- occupation(X, nun).
innocent(winnie_the_pooh).
innocent(julie_andrews)
guilty(X) :- occupation(X, thief).
guilty(joe_bloggs).
guilty(rolf_harris).
```

Consider the following dialogue:

```
?- innocent(st_francis).
no.
```

24

24

Negation as Failure is not Logical

This cannot be right, beause everyone knows that St Francis is innocent. But in Prolog the above happens because st_francis is not in the program as a fact. Because the program is hidden from the user, the user will believe it because the computer says so.

How to solve this?

25

25

not makes things worse

Using not will not help you. Do not try to remedy this by defining:

```
guilty(X) :- not(innocent(X)).
```

This is useless, and makes matters even worse:

```
?- guilty(st_francis).
yes
```

It is one thing to show that st_francis cannot be demonstrated to be innocent. But it is quite another thing to incorrectly show that he is guilty.

26

26

Negation-by-failure is not logical

Some disturbing behaviour even more subtle than the innocent/guilty problem, and can lead to some extremely obscure programming errors. Here is a restaurant database:

```
good_standard(goedels).
good_standard(hilberts).
expensive(goedels).
reasonable(R) :- not(expensive(R)).
```

Consider the following query:

?- good_standard(X), reasonable(X).

X = hilberts

yes

But if we ask the logically equivalent question:

?- reasonable(X), good_standard(X).

no.

27

27

Question

Why do we get different answers for what seem to be logically equivalent queries?

The difference between the questions is as follows.

- In the first query, the variable X is always instantiated when reasonable(X) is executed.
- In the second query, X is not instantiated when reasonable(X) is executed.
- The semantics of reasonable(X) differ depending on whether its argument is instantiated.

28

28

Not a Good Idea!

It is bad practice to write programs that destroy the correspondence between the logical and procedural meaning of a program without any good reason for doing so.

Negation-by-failure does not correspond to logical negation, and so requires special care.

29

29

How to fix it?

One way is to specify that negation is undefined whenever an attempt is made to negate a non-ground formula.
A formula is 'ground' if is has no unbound variables.

Some Prolog systems issue a run-time exception if you try to negate a non-ground goal.

30

30

Close World Assumption

- In Logic, some propositions may be true, may be false, may be unknown.
- Close World Assumption (CWA) assumes that any proposition is either true or false.
- Using CWA in Prolog, then it provides a good base for Negation as Failure and some outputs will make sense.
- Example:

```
man(tom).
mortal(X) :- man(X).
?- mortal(X).
X = tom
?- mortal(jerry).
no.
```

31

31

Clauses and Databases

In a relational database, relations are regarded as tables, in which each element of an *n*-ary relation is stored as a row of the table having *n* columns.

supplier

jones	chair	red	10
smith	desk	black	50

Using clauses, a table can be represented by a set of unit clauses. An *n*-ary relation is named by an *n*-ary predicate symbol.

```
supplier(jones, chair, red, 10).
supplier(smith, desk, black, 50).
```

32

32

Clauses and Databases

Advantages of using clauses:

1. Rules as well as facts can coexist in the description of a relation.
2. Recursive definitions are allowed.
3. Multiple answers to the same query are allowed.
4. There is no role distinction between input and output.
5. Inference takes place automatically.

33

33

Negation and Representation

- Like databases, clauses cannot represent negative information. Only true instances are represented.
    - The battle of Waterloo occurred in 1815.
- How can we show that the battle of Waterloo did not take place in 1923? The database cannot tell us when something is not the case, unless we do one of the following:

1. 'Complete' the database by adding clauses to specify the battle didn't occur in 1814, 1813, 1812, ..., 1816, 1817, 1818,...
2. Add another clause saying the battle did not take place in another year (the battle occurred in *and only in* 1815).
3. Make the 'closed world assumption', implemented by 'negation by failure'.

34

34

Summary

- Controlling backtracking: the cut **!**
  - Efficiency: avoids needless REDO-ing which cannot succeed.
  - Simpler programs: conditions for choosing clauses can be simpler.
  - Robust predicates: definitions behave properly when forced to REDO.
- Green cut = cut doesn't change the logic = **good**
- Red cut = without the cut the logic is different = **bad**
- cut/fail: when it is easier to prove something is false than true.

35

35

Negation as Failure

We can use the same idea of "cut fail" to define the predicate not, which takes a term as an argument. not will "call" the term, that is evaluate it as though it is a goal:

not(G) fails if G succeeds

not(G) succeeds if G does not succeed.

In Prolog,

```
        not(G) :- call(G), !, false.
        not(_).
```

call is a built-in predicate.

36

36

Negation as Failure

Most Prolog systems have a built-in predicate like not. GProlog calls it \+. Remember, not does not correspond to logical negation, because it is based on the success/failure of goals. It can, however, be useful:

```
likes(mary, X) :- \+(reptile(X)).

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

37

37

Negation as Failure

```
single_student(X) :-
    (\+ married(X)),
    student(X).
student(bill).
student(joe).
married(joe).
```

```
?- single_student(bill).
    → yes.
?- single_student(joe).
    → no.
```

```
▯ ?- single_student(X)
        → no.
```

38

38

Negation as Failure

- The \+ prefix operator is the standard in modern Prolog.
- \+P means “P is unprovable”
- \+P succeeds if P fails (e.g., we can find no proof for P) and fails if we can find any single proof for P.

39

39

Negation as Failure

```
single_student(X) :-
    (\+ married(X)),
    student(X).
student(bill).
student(joe).
married(joe).
```

```
?- single_student(bill).
    → yes.
?- single_student(joe).
    → no.
```

```
?- single_student(X)
      → no.
```

40

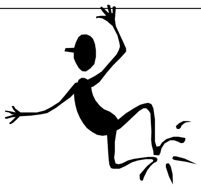
40

Negation as Failure 2nd Try

```
single_student(X) :-
  student(X),
  (\+ married(X)).
student(bill).
student(joe).
married(joe).
```

```
?- single_student(bill).
   → yes.
?- single_student(joe).
   → no.
```

□ ?- single_student(X)
→ X=bill.



41

41

Negation-by-failure can be non-logical

Some disturbing behaviour even more subtle than the innocent/guilty problem, and can lead to some extremely obscure programming errors. Here is a restaurant database:

```
good_standard(goedels).
good_standard(hilberts).
expensive(goedels).
reasonable(R) :- not(expensive(R)).
```

Consider the following dialogue:

?- good_standard(X), reasonable(X).

X = hilberts

But if we ask the logically equivalent question:

?- reasonable(X), good_standard(X).

no.

42

42

Question

Why do we get different answers for what seem to be logically equivalent queries?

The difference between the questions is as follows.

In the first question, the variable X is always instantiated when reasonable(X) is executed.

In the second question, X is not instantiated when reasonable(X) is executed.

The semantics of reasonable(X) differ depending on whether its argument is instantiated.

43

43

Not a Good Idea!

It is bad practice to write programs that destroy the correspondence between the logical and procedural meaning of a program without any good reason for doing so.

Negation-by-failure does not correspond to logical negation, and so requires special care.

44

44

How to fix it?

One way is to specify that negation is undefined whenever an attempt is made to negate a non-ground formula.
A formula is ‘ground’ if is has no unbound variables.

Some Prolog systems issue a run-time exception if you try to negate a non-ground goal.

45

45