

# Prolog

## negation and cut

Tim Finin

University of Maryland  
Baltimore County

9/19/01

1

## Negation and the Cut

- Negation as Failure
  - Negation succeeds if search fails.
  - Not Constructive - Unification does not produce any bindings.
  - Consistent Interpretation depends on Closed World Assumption
- The Cut '!'
  - A device for controlling the search
  - Used to increase efficiency
  - BUT can alter semantics of a program -- change its solution set.

UMBC

2

## 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.
```



UMBC

3

## 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.
- \+ is like a turnstile symbol with a line thru it

UMBC

4

## 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.



## Negation as Failure 2<sup>nd</sup> 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.



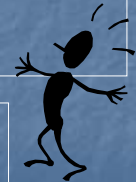
## Closed World Assumption

- Assumption that the world is defined in its entirety
  - The representation is "complete"/"closed"
- No true statement is missing from the representation
- In practice, assumed for conventional databases
  - "Sorry, sir you must NOT exist your social security number is NOT IN our database, bye, bye".
- From a logic program, P, allows us to conclude
  - the **negation of A**
  - IF **A is NOT IN** the meaning of P

## Negation as Failure & the CWA

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

```
:- single_student(bill).
   → yes.
:- single_student(joe).
   → no.
:- single_student(jim).
   → yes.
```



✓ But Jim IS married.  
✓ Maybe I should read up on the CWA.

## 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.
- Bad Cut**
  - A cut used in such a way as to make the actual meaning diverge from the intended meaning.
- Ugly Cut**
  - Obscures intended meaning but does not loose it

## A Green Cut

```
fact(N,1) :- N = 0, !.
fact(N,F) :-
    N > 0,
    M is N - 1,
    fact(M,F1)
    F is N * F1.
```

- If N = 0 in first clause we do not need to consider second clause. The second **will** fail, so we CUT to prune unnecessary consideration of the second clause.
- With or without the cut the program produces the same solutions. Its intended meaning is intact.

## A Good Red Cut

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

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

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

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

```
equal
not_equal
yes.
```

## 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.
```

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

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

```
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?
```

## 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)).  
%%%%%%%%%%%%%%%%R5. entitled(X,nothing).
```

```
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.
```

## Is it Good, Bad or Just Ugly?

```
not(P) :- P, !, fail.
```

```
not(P).
```