Prolog
Negation, Cut

Fariba Sadri

(Acknowledgement to Keith Clark for use of some slides)

Negation in Prolog \+

In Prolog negation is allowed only in queries and in the bodies of rules - not in the heads of rules.

E.g.

✔ **?- student(X), \+ gets_scholarship(X).**

✔ **happy(X):- owns_a_house(X),**

\+ has_mortgage(X).

- **\+ has_mortgage(X) :- child(X).**

- **\+ has_scholarship(john).**

Negation in Logic

In logic we can write both negative and positive statements.

E.g.

T: **student(john)**

student(mary)

gets_scholarship(john)

¬ gets_scholarship(mary)

we can show:

From T we can prove:

student(mary) ∧¬gets_scholarship(mary)

So how can Prolog show

student(mary) , \+ gets_scholarship(mary)?

It will use

the **Closed world assumption.**

Closed world assumption

In Prolog:

- Programs contain only positive statements (i.e. rules and facts with positive heads).
- Negative conditions are evaluated using:

  The closed world assumption: i.e.

  Any fact that cannot be inferred is false.

- Negative information is inferred by default, using the **Negation as failure (Naf) rule**.

E.g. In Prolog we would write:

```
student(john).
student(mary).
gets_scholarship(john).

?- student(X), \+ gets_scholarship(X).
X = mary
```


Because Mary is a student that Prolog *cannot* prove gets a scholarship.

The Negation as Failure (Naf) Rule

- **\+ Q** is proved if all evaluation paths for the query **Q** end in failure.
- Proof of **\+Q** will not generate any bindings for variables in **Q.**
- If **Q** contains variables $\mathbf{X_1,..,X_k}$, it effectively establishes:

$$\neg\exists X_1 \ldots \exists X_k \; Q$$

- \+ **Q** fails to be proved if there is some proof of **Q.**

Be careful with the order of subgoals

E.g. In Prolog we would write:

student(john).

student(mary).

gets_scholarship(john).

?- \+ gets_scholarship(X), student(X).

no

Justification of the NAF Rule

Naf rule is valid providing we assume clauses constitute *complete* definitions of the relations they describe, *and* that different terms denote *different* individuals (e.g. **paul≠peter**).

Justification of the NAF Rule cntd.

Program P:

student(john).

student(mary).

gets_scholarship(john).

Completion of P:

student(X) ↔ X=john ∨ X= mary

gets_scholarship(X) ↔ X=john

john ≠ mary

Example use of \+

```
dragon(puff).
dragon(macy).
dragon(timothy).
dragon(spyro).

magical(puff).
lives_by_the_sea(spyro).
vegetarian(macy).
fights(spyro).

lives_forever(X):- magical(X).
lives_forever(X):- vegetarian(X).
lives_forever(X):- lives_by_the_sea(X), \+ fights(X).
```

?- dragon(X), \+ lives_forever(X).

Construct the Prolog evaluation to see how it finds the answers.

Negated conditions with unbound variables

Be careful with variables in negative conditions:

?- \+ lives_forever(X), dragon(X).

will have no answers. Why?

Apply the Naf inference rule to first condition – what is the result?

More on Prolog Negation

\+ can be applied to conjunctions.

Example:

Assuming a set of male/1 and parent/2 facts:

we can define dragons with no sons:


```
no_sons(D):-
        dragon(D),
        \+(parent(D,C), male(C)).
```

\+ can be nested.

Example:

Dragons with no daughters:


```
no_daughters(D):-
        dragon(D),
        \+(parent(D, C),\+male(C)).
```

\+ can be applied to disjunctions.

Example:


```
damaged(D) :-
        dragon(D),
```



```
        \+ ( breathes_fire(D) ; has_wings(D)).
```

Prolog definition of \+

```
\+(P) :- P, !, fail.
\+(_).
```

Evaluation control : !

- Cut, denoted by "!", is a Prolog query evaluation control primitive.
- It is "extra-logical" and it is used to control the search for solutions and prune the search space.
- In logical reading it is ignored.
- The cut can only be understood procedurally, in contrast to the declarative style that logic programming encourages.
- But used wisely, it can significantly improve efficiency without compromising clarity too much.

Example program with needless search

```
send(Cust, Balance, Mess):-
        Balance =< 0,
        warning(Cust, Mess).
send(Cust, Balance, Mess):-
        Balance > 0,
        Balance =< 50000,
        credit_card_info(Cust, Mess).
send(Cust, Balance, Mess):-
        Balance > 50000
        investment_offer(Cust, Mess).
```

For a condition:

send(bill, -10, Message)

in a query for which all solutions are being sought, Prolog will try to use second and third clause after an answer has been found using the first clause.

Clearly this search is pointless.

Using !

```
send(Cust, Balance, Mess):-
        Balance =< 0, !,
        warning(Cust,Mess).
send(Cust,Balance, Mess):-
        Balance > 0,
        Balance =< 50000, !,
        credit_card_info(Cust,Mess).
send(Cust,Balance, Mess):-
        Balance > 50000
        investment_offer(Cust,Mess).
```

```
send(Cust, Balance, Mess):-
                Balance =< 0, !,
                warning(Cust,Mess).
send(Cust,Balance, Mess):-
delete ⟶        Balance > 0,
                Balance=< 50000, !,
                credit_card_info(Cust,Mess).
send(Cust,Balance, Mess):-
delete ⟶        Balance > 50000
                investment_offer(Cust,Mess).
```

The Effect of !

Program:

p(...):- T_1,. . . , T_k, !, B_1 . . . , B_n.

p(...):- …

p(...):- …

In trying to solve a call:

p(…)

if first clause is applicable, and

T_1,. . . , T_k

is provable, then on *backtracking*:

- ***do not try*** **to find an alternative solution for T_1,. . . , T_k and**
- ***do not try*** **to use a later applicable clause for the call p(…).**

Backtacking will happen as normal on B_1 . . . , B_n.

Cut Practice

Place a cut in different positions in the following program and test your understanding of its effects.


```
p(X,Y) :-  q(X), r(Y).
p(X,Y) :-  s(X,Y).

q(1).          q(2).
r(1).          r(2).          r(3).
s(10, 10).     s(20, 10).
```

Be careful with the cut!

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



```
?- max(1, 2, Z).
Z=2
?- max(1, 2, 1).
yes
```

An alternative definition of max

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



```
?- max(1,2,Z).
Z = 2
?- max(1,2,1).
no
```