PROLOG 2025

ELTE, LOGIC DEPARTMENT

Attila Molnár
Kőbányai Szent László Gimnázium

September 24, 2025

Cut

Cut

Prolog works with backtracing which can lead to inefficiency, i.e. wasting time and memory on possibilities that lead nowhere.

We can control backtracking by the cut predicate: !/0

Cut is a goal that always succeeds, so  will always get through it.

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

vs

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

CUT

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.

$\texttt{q:- p}_1\texttt{,..., p}_m\texttt{, !, r}_1\texttt{, ..., r}_n\texttt{.}$

when we reach the cut it commits us:

- to this particular clause of `q`
- to the choices made by $\texttt{p}_1\texttt{, ..., p}_m$
- NOT to choices made by $\texttt{r}_1\texttt{, ..., r}_n$

GREEN CUT

Let's consider the following `max/3` predicate:

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

Green cut

Let's consider the following `max/3` predicate:

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

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

Green cut

Let's consider the following `max/3` predicate:

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

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

?- max(7,3,7)
```

GREEN CUT

Let's consider the following `max/3` predicate:

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

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

?- max(7,3,7)
yes
```

Green cut

Let's consider the following `max/3` predicate:

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

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

?- max(7,3,7)
yes
```

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

GREEN CUT

Let's consider the following `max/3` predicate:

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

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

?- max(7,3,7)
yes
```

```
?- max(2,3,2)
no

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

GREEN CUT

Let's consider the following `max/3` predicate:

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

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

?- max(7,3,7)
yes
```

```
?- max(2,3,2)
no

?- max(2,3,5)
no
```

GREEN CUT

Let's consider the following `max/3` predicate:

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

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

?- max(7,3,7)
yes
```

```
?- max(2,3,2)
no

?- max(2,3,5)
no
```

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

GREEN CUT

Let's consider the following `max/3` predicate:

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

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

?- max(7,3,7)
yes
```

```
?- max(2,3,2)
no

?- max(2,3,5)
no
```

```
?- max(2,3,Y)
Y = 3

?- max(7,3,_)
```

GREEN CUT

Let's consider the following `max/3` predicate:

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

`?- max(2,3,3)`
`yes`
 	
`?- max(7,3,7)`
`yes`
	`?- max(2,3,2)`
	`no`
	
	`?- max(2,3,5)`
	`no`
		`?- max(2,3,Y)`
		`Y = 3`
		
		`?- max(7,3,_)`
		`true`

GREEN CUT

Let's consider the following `max/3` predicate:

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

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

?- max(7,3,7)
yes
```

```
?- max(2,3,2)
no

?- max(2,3,5)
no
```

```
?- max(2,3,Y)
Y = 3

?- max(7,3,_)
true
```

After `?- max(2,3,Y)` if asked for more solutions, polog will try to satisfy the second clause, which is completely useless – since we know the two clauses are exclusive (hence, it returns with `false`).

Green cut

We can fix this by adding a cut to the rule:

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

Green cut

We can fix this by adding a cut to the rule:

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

Now

- If the `X =< Y` succeeds, the cut commits us to this choice, and the second clause of `max/3` is not considered.
- If the `X =< Y` fails, Prolog goes on to the second clause.

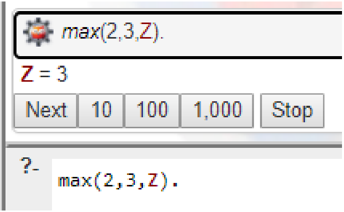
Without cut

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

With cut

WITHOUT CUT

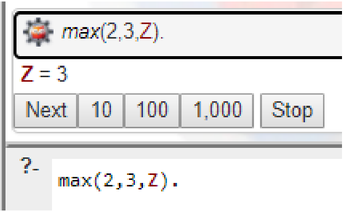
```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X > Y.
```

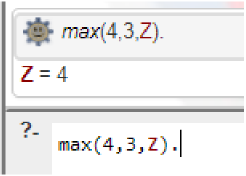


WITH CUT

Without cut

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

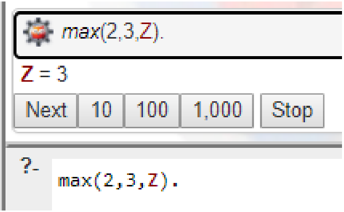


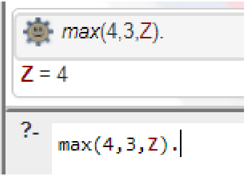


With cut

Without cut

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



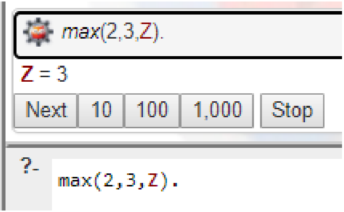


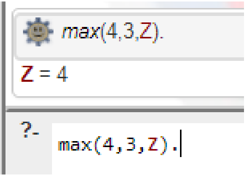
With cut

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

Without cut

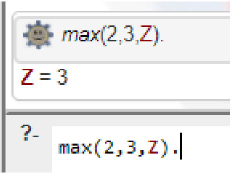
```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X > Y.
```





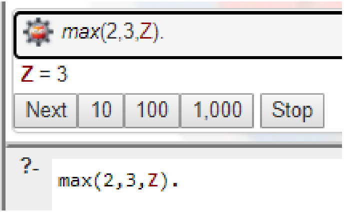
With cut

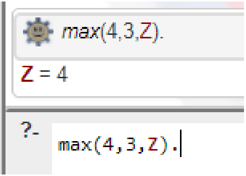
```
max(X,Y,Y):- X =< Y, !.
max(X,Y,X):- X > Y.
```



Without cut

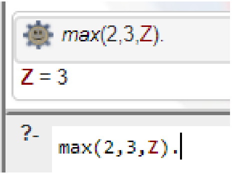
```
max(X,Y,Y) :- X =< Y.
max(X,Y,X) :- X > Y.
```

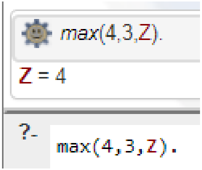




With cut

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





Green cut, red cut

Green cut: cuts that do not change the meaning of a predicate.
E.g. `max/3` – the new code gives exactly the same answers as the old version, but it is more efficient.

Green cut, red cut

GREEN CUT: cuts that do not change the meaning of a predicate.
E.g. `max/3` – the new code gives exactly the same answers as the old version, but it is more efficient.

To see how red cuts work, let's modify our `max/3`:

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

The second clause is basically redundant. Let's remove it!

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

Now what?

Red cut

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

Red cut

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

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

RED CUT

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

```
?- max(2,3,X)
X = 3
```

RED CUT

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

```
?- max(2,3,X)
X = 3
```

```
?- max(4,3,X)
```

RED CUT

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

```
?- max(2,3,X)
X = 3
```

```
?- max(4,3,X)
X = 4
```

Red cut

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

```
?- max(2,3,X)
X = 3
```

```
?- max(4,3,X)
X = 4
```

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

Red cut

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

```
?- max(2,3,X)
X = 3
```

```
?- max(4,3,X)
X = 4
```

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

Red cut

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

```
?- max(2,3,X)
X = 3
```

```
?- max(4,3,X)
X = 4
```

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

Now  will be able to answer alternative questions/queries, but not yes-no ones.

Because of the form of the query, it will go directly to the second clause, and only check `max(X,Y,X).`, which is obviously true – syntactically.

Red cut

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

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

...however, we can patch it with a unification after the cut:

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

Red cut

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

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

…however, we can patch it with a unification after the cut:

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

Mind the `Z` in the head of the rule! This `(X,Y,Z)` tempts  to try to substitute in the first clause, since `(X,Y,Z)` under appropriate circumstances can unify with `(X,Y,X)`!

Red cut

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

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

...however, we can patch it with a unification after the cut:

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

Mind the `Z` in the head of the rule! This `(X,Y,Z)` tempts  to try to substitute in the first clause, since `(X,Y,Z)` under appropriate circumstances can unify with `(X,Y,X)`!

After `X =< Y` (2 =< 3) succeeds, we ask if 3=2, which will be false.

RED CUT

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

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

…however, we can patch it with a unification after the cut:

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

Mind the `Z` in the head of the rule! This `(X,Y,Z)` tempts to try to substitute in the first clause, since `(X,Y,Z)` under appropriate circumstances can unify with `(X,Y,X)`!

After `X =< Y` (2 =< 3) succeeds, we ask if 3=2, which will be false.

Because of the `!` cannot go back and try another setup for substitution (`max(X,Y,X)`), it will be stuck with what was given before the `!`.

RED CUT

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

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

...however, we can patch it with a unification after the cut:

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

Mind the `Z` in the head of the rule! This `(X,Y,Z)` tempts  to try to substitute in the first clause, since `(X,Y,Z)` under appropriate circumstances can unify with `(X,Y,X)`!

After `X =< Y` (2 =< 3) succeeds, we ask if 3=2, which will be false.

Because of the `!`  cannot go back and try another setup for substitution (`max(X,Y,X)`), it will be stuck with what was given before the `!`.

However, if we ask `max(4,3,X)`, it will be able to go to the second clause after failing with the first, since if fails already before the `!`, at `X =< Y`.

RED CUT

RED CUT: a cut that changes the meaning of the predicate. If we remove it, we do not get an equivalent program.

RED CUT

RED CUT: a cut that changes the meaning of the predicate. If we remove it, we do not get an equivalent program. Using red cut the resulting program

- will not be fully declarative (we interfere with the control flow)
- can be harder to read
- might be prone to subtle programming mistakes

Negation as Failure

Combining `!, fail.` we can express negation:

NEGATION AS FAILURE

Combining `!, fail.` we can express negation:

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).
```

Negation as Failure

Combining `!, fail.` we can express negation:

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

Negation as Failure

Combining `!, fail.` we can express negation:

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).

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

Negation as Failure

Combining `!, fail.` we can express negation:

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).

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

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

Negation as Failure

Combining `!, fail.` we can express negation:

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).

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

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

Negation as Failure

Combining `!, fail.` we can express negation:

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).

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

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

`a` gets substituted in `bugKahunaBurger(X)` where it will fail and go for the second clause. Among `burger`s `bigMac` is the first, so it succeeds, the query will be true.

Negation as Failure

Combining `!, fail.` we can express negation:

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).

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

?- enjoys(vincent,b).
```

Negation as Failure

Combining `!, fail.` we can express negation:

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).

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

?- enjoys(vincent,b).
false
```

Negation as Failure

Combining `!,` `fail.` we can express negation:

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).

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

?- enjoys(vincent,b).
false
```

Prolog tries to substitute `b` in `bigKahunaBurger(X)` in the first clause. It will succeed, since `bigKahunaBurger(b).`, so it goes on, through `!` all the way to `fail`. Because of the `!`  cannot go back and look for other possible substitutions for `b`...

Negation as Failure

The cut-fail combination offers us some form of negation.

It is called negation as failure, and defined as follows:

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

If Goal succeeds, make the clause fail, otherwise let it go on.

Since negation as failure is frequently used in , there is a built-in predicate for it: `\+`

Negation as Failure

3 equivalent expressions of the fact that `vincent enjoys` all sorts of `burger`s except for `bigKahunaBurger`:

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

enjoys(vincent,X):- burger(X), neg(bigKahunaBurger(X)).

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

Negation as failure is not a logical negation.

After the `!` we cannot go back, the program will fail.

This is why we need to have `burger(X)` before `\+`. We need to allow 🦉 to freely select, substitute from our set of `burger`s first.

A trick with `fail`

A TRICK WITH `FAIL`

Let's say we want not only to give us a suggestion about what `burger`s `vincent` would like, but we want to be able to have a full list of these `burger`s!

A TRICK WITH `FAIL`

Let's say we want  not only to give us a suggestion about what `burger`s `vincent` would like, but we want to be able to have a full list of these `burger`s!

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).

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

A TRICK WITH `FAIL`

Let's say we want not only to give us a suggestion about what `burger`s `vincent` would like, but we want to be able to have a full list of these `burger`s!

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).

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

whattobuyto(Y):- enjoys(Y,X), write(X), nl, fail; true.
```

A TRICK WITH `FAIL`

Let's say we want not only to give us a suggestion about what `burger`s `vincent` would like, but we want to be able to have a full list of these `burger`s!

```
burger(X):- bigMac(X).
burger(X):- bigKahunaBurger(X).
burger(X):- whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).

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

whattobuyto(Y):- enjoys(Y,X), write(X), nl, fail; true.
```

Until there are `burger`s that have never been tested, `fail` will make going back to `burger`s – since backtracks locally. At the point where all the `burger`s inthe KB have been checked, will go back to the main rule (`whattobuyto`) to see if there are any other ways to save the day,