

**Consider:** the following pseudo-Pascal code snippet

```
if X < 3 then Y := 0
if 3 <= X and X < 6 then Y := 2
if 6 <= X then Y = 4
```

**In Prolog:** `f(X,0) :- X < 3.`

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

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

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

fails, backtracking two useless (mutually exclusive) alternatives.

**Stop backtracking:** using the *cut* pseudo-predicate !:

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

**“Green” cuts:** do not change the declarational meaning of the program, only the procedural. Generation of results and their order will be *exactly* identical to a program where green cuts are removed. However, the efficiency may be dramatically enhanced.

**Consider:** `?- f(7, Y)..` The failure/success structure looks as following for the successive clauses:

1. goal fails
2. first goal succeeds, second fails
3. goal succeeds

**Note:** because of the mutual exclusiveness semantics, the first goal of each the second and third clause is redundant, as it can be captured by the last predicate of the earlier clause. We must only ensure that the program will never continue to the other clauses, once a clause has been satisfied.

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

**“Red” cuts:** The cuts in this example *can not* be removed. This would change also the declarative meaning of the program. The program will do different things with and without those cuts.

**Cut: Definition**

**Cut:** consider a parent goal invocation, i.e. a goal that is unified with the head of a clause during a prolog program run. On crossing a cut during processing of the clause subgoals, the choices made between parent goal invocation and cut are frozen. No alternatives are considered on backtracking over the cut.

In other words, once crossing a cut, on backtracking the system will backskip from the cut to the parent goal without considering any alternatives, neither between the parent clause and cut, nor in the sibling clauses of the same predicate invocation. That is, once crossing a cut on backtracking, processing of the present predicate is considered complete.

**Note:** Cuts allows prolog to save resources and judicious use of cuts can greatly enhance space and time efficiency of prolog programs, first by having opportunity to cleanup backtrack memory, second by avoiding alternatives. However, this comes at a significant cost for legibility and maintainability of programs.

**Example:** `C:- P, Q, R, !, S, T, U.`

`C:- V.`  
`A:- B, C, D.`

On invoking A, C behaves as usual, as long as P, Q and R are being evaluated, and also if V is being evaluated. On crossing the cut, regular backtracking will be performing usual backtracking behaviour on S, T, U, but if these fail, C will fail as whole, leading to backtrack through B. If B succeeds again, producing another candidate solution, C's invocation will resume from scratch, starting from the new candidate solution produced by B.

**Negation**

**Example:** “Ron likes al animals but spiders”. We cannot use

```
likes(ron, X) :- animal(X).
```

**Use a trick:**

```
likes(ron, X) :- spider(X), !, fail.
likes(ron, X) :- animal(X).
```

or

```
likes(ron, X) :-
    spider(X), !, fail
    ;
    animal(X).
```

**Note:** built-in predicate `fail` always fails. If X is identified as a spider, the cut is crossed and the cut/fail combination causes `likes` to fail immediately. This allows us to formulate the concept of *negation* in prolog.

**“Not” predicate:** `not(P) :-`

```
P, !, fail
;
true.
```

**Example (continued):** with appropriate operator declaration, the example from above becomes more clearly readable:

```
likes(ron, X) :-
    animal(X),
    not snake(X).
```

## Problems with Negation

**Closed World Assumption:** prolog negation can not create facts from "nowhere" — only filtering existing test solutions.

**Example:**

```
good_music(beethoven).
good_music(mozart).

loud(beethoven).

quiet(Music) :- not loud(Music).
```

## Reading

**Ask:**

```
?- good_music(M), quiet(M).
M = mozart

?- quiet(M), good_music(M).
no
```

- Read Bratko, Ch. 6, I/O

**Why?** do not use negation to generate cases; due to closed world assumption, negation can only filter, not generate. Put (positive) generating predicate first, followed by negating predicates. In particular, without suitable arrangements, predicates containing negation do not in general allow programs with correct declarative semantics have a correct procedural semantics.