Logic Programming

Prolog

Joey W. Coleman, Stefan Hallerstede

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

11 April 2014

Prolog and logic programming

Prolog unification

Prolog search

Cuts and negation

Extra-logical predicates

Differences to logic programming

- Prolog unification omits the occur check
- Prolog uses a deterministic search heuristic
  - clauses are tried from top to bottom and
  - goals in clause bodies are evaluated from left to right
- In Prolog alternative clauses are evaluated by backtracking
- Prolog provides extra-logical predicates, such as I/O or cut
- Prolog programs may be non-terminating

Prolog and logic programming

Prolog unification

Prolog search

Cuts and negation

Extra-logical predicates

Unification in Prolog

Algorithm for computing the unifier of two terms u and v

Input: a pair of terms $u \approx v$
Output: a substitution θ such that $s\theta = t\theta$ (in solved form) or ***failure***
$\Theta := \{u \approx v\}$
repeat
  select an arbitrary $s \approx t$ from Θ;
  if $s = f(s_1, \ldots, s_n)$ ***then***
    if $t = X$ ***then*** replace $s \approx t$ by $t \approx s$
    elsif $t = f(t_1, \ldots, t_n)$ ***then*** replace $s \approx t$ by $s_1 \approx t_1, \ldots, s_n \approx t_n$
    else return failure
  elsif $s = X$ ***then***
    if $t = X$ ***then*** remove $s \approx t$
    else replace all other occurrences of X in Θ by t
until Θ remains unchanged for any $s \approx t$ from Θ;
$\theta := \{s = t \mid s \approx t \in \Theta\}$;
return θ

Efficiency considerations

- The occur-check is worst-case exponential
- That is, potentially it is very inefficient
- Unfortunately, unification is used in each computation step
- Without the occur check
  most unifiers can be computed in constant time
- This is very efficient
- This is the only reason for the difference
  between unification in logic programming and Prolog
- When needed, “proper” unification can be implemented in Prolog

Unification example

Unify $f(X, g(X))$ and $f(Z, Z)$

$$\{f(X, g(X)) \approx f(Z, Z)\}$$

$$\rightarrow \{X \approx Z, g(X) \approx Z\}$$

$$\rightarrow \{X \approx Z, g(Z) \approx Z\}$$

$$\rightarrow \{X \approx Z, Z \approx g(Z)\}$$

$$\rightarrow \{X \approx Z, Z \approx g(g(Z))\}$$

$$\rightarrow \{X \approx Z, Z \approx g(g(g(Z)))\}$$

$$\rightarrow \ldots$$

Some Prolog implementations yield non-terminating computations.

Others may yield infinite types, e.g., $Z = g(g^*)$

However, without occur-check the result will always be different from the logic programming unification.

Explicit unification in Prolog

- The goal `s = t` attempts unification of the terms `s` and `t`
- If `p(X,X).` is a fact,
  then `p(s,t)` attempts unification of the terms `s` and `t`
- If `p(X,Y) :- X = Y.` is a rule,
  then `p(s,t)` attempts unification of the terms `s` and `t`

Prolog and logic programming

Prolog unification

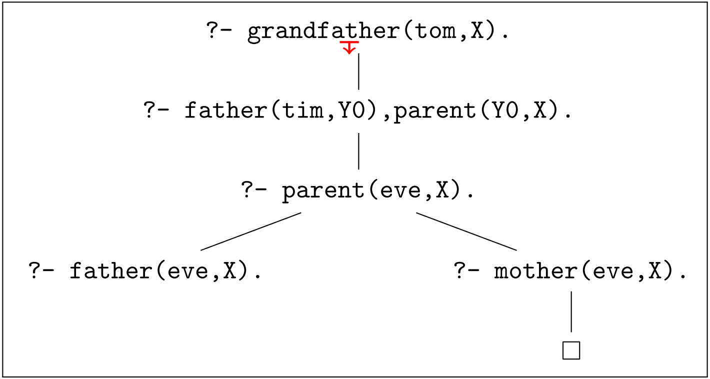
Prolog search

Cuts and negation

Extra-logical predicates

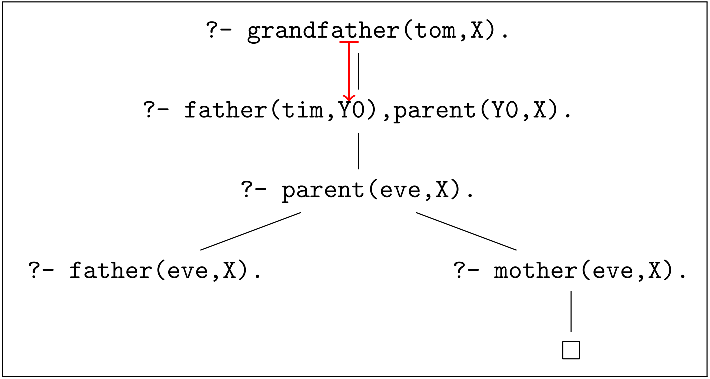
Example 1

```
grandfather(X,Z) :- father(X,Y),parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(tom, eve).
mother(eve, tim).
```



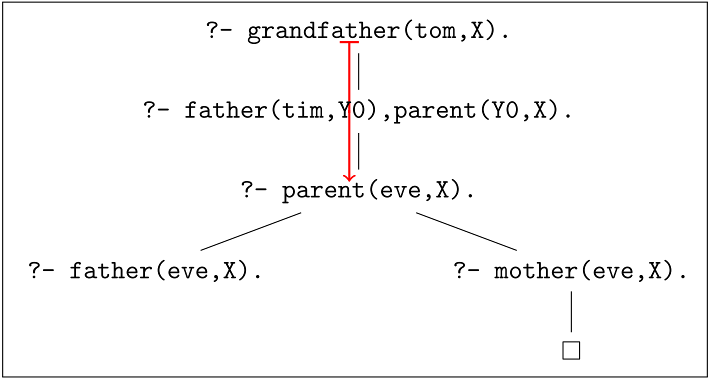
Example 1

```
grandfather(X,Z) :- father(X,Y),parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(tom, eve).
mother(eve, tim).
```



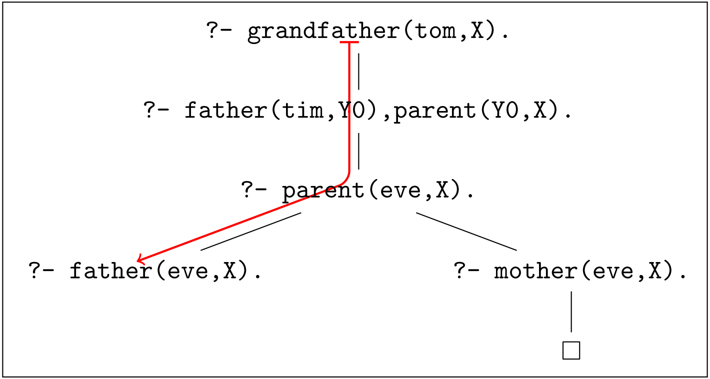
Example 1

```
grandfather(X,Z) :- father(X,Y),parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(tom, eve).
mother(eve, tim).
```



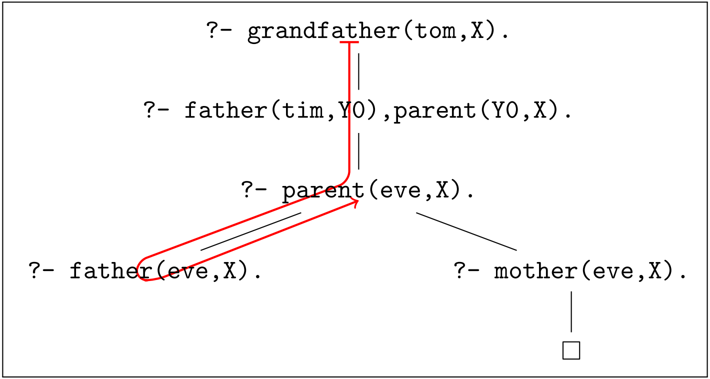
Example 1

```
grandfather(X,Z) :- father(X,Y),parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(tom, eve).
mother(eve, tim).
```



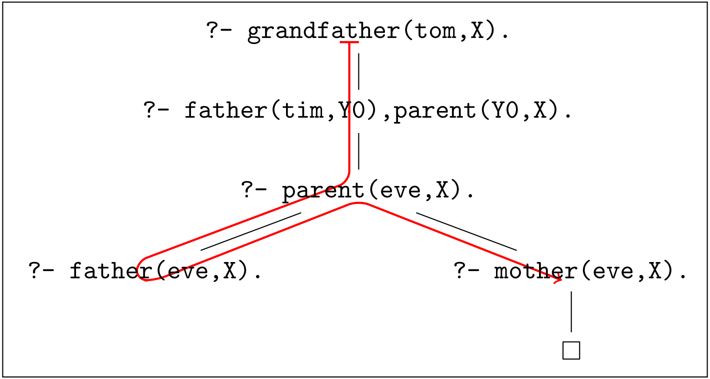
Example 1

```
grandfather(X,Z) :- father(X,Y),parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(tom, eve).
mother(eve, tim).
```



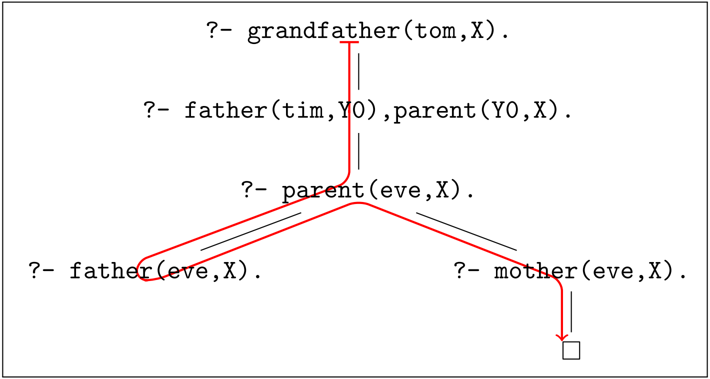
Example 1

```
grandfather(X,Z) :- father(X,Y),parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(tom, eve).
mother(eve, tim).
```



Example 1

```
grandfather(X,Z) :- father(X,Y),parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(tom, eve).
mother(eve, tim).
```



Backtracking

- Returning to an earlier node of the tree is called ***backtracking***
- Backtracking involves
  - remembering the current clause of each node and
  - unwinding unifications to nodes occurring earlier in the tree
- It is a major challenge to make backtracking efficient
- The ***Warren abstract machine*** provides an efficient approach

Generate and test

- A common technique in algorithm design and programming
- Useful for problem solving
- Generate-and-test principle:
  - One routine generates candidate solutions
  - Another routine tests the candidates with respect to the solution of the problem

Example: “send more money”

- Solve the equation

  `SEND + MORE = MONEY`

- ...where `S,E,N,D,M,O,R,Y` are different digits between `0` and `9`
- ...such that `M>0` and `S>0`

(Remark. More efficient technique: ***constraint solving***.)

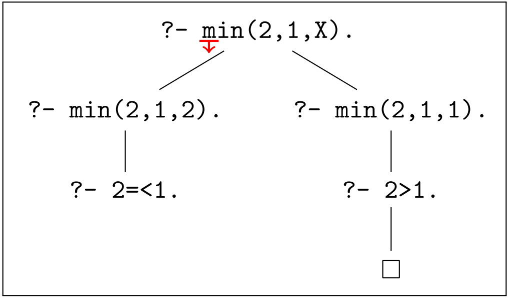
Example 2

```
send_more_money :-
  X = [S,E,N,D,M,O,R,Y],
  Digits = [0,1,2,3,4,5,6,7,8,9],
  assign_digits(X, Digits),
  M > 0,
  S > 0,
  1000*S + 100*E + 10*N + D +
  1000*M + 100*O + 10*R + E =:=
  10000*M + 1000*O + 100*N + 10*E + Y,
  write(X).

select(X, [X|R], R).
select(X, [Y|Xs], [Y|Ys]):- select(X, Xs, Ys).

assign_digits([], _List).
assign_digits([D|Ds], List):-
  select(D, List, NewList),
  assign_digits(Ds, NewList).
```

Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

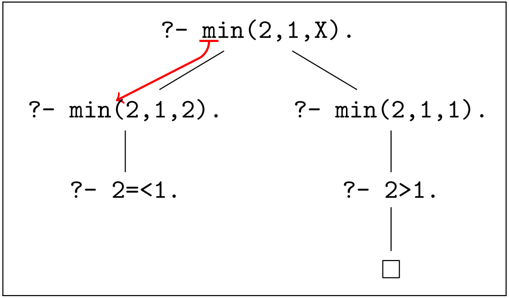
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "!" prunes the search tree of a Prolog program.
Example:

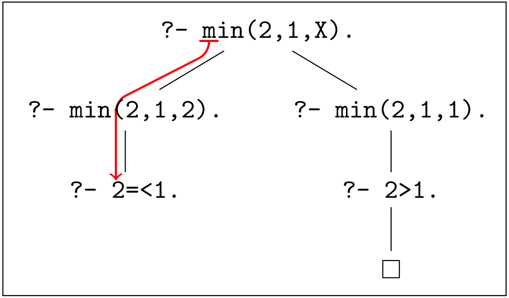
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

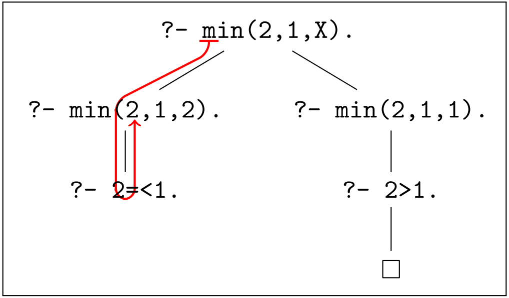
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

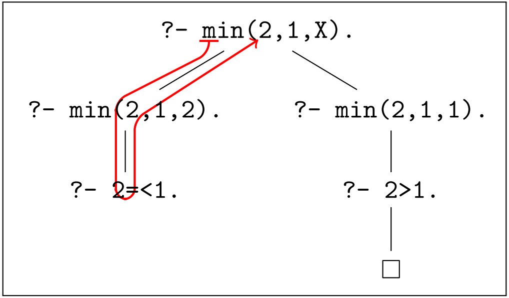
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

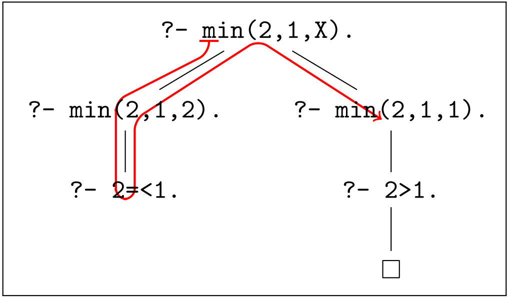
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

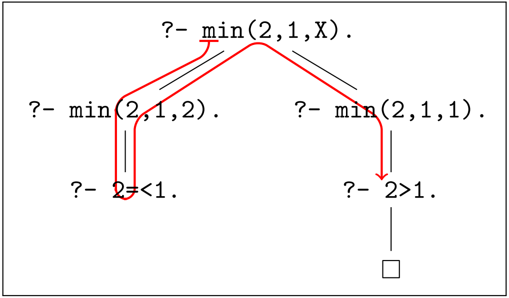
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** “`!`” prunes the search tree of a Prolog program.
Example:

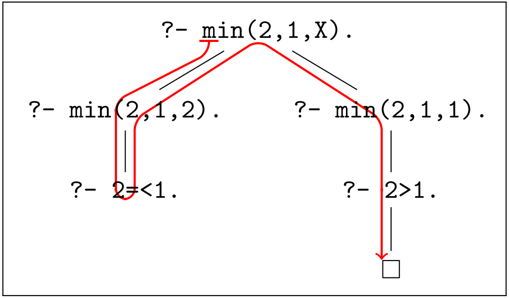
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

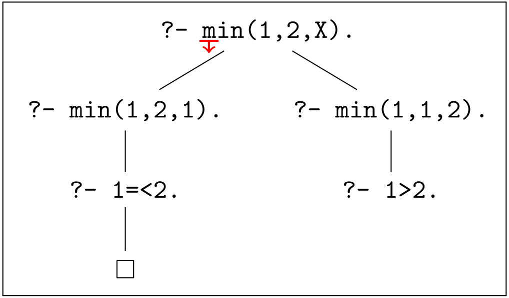
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

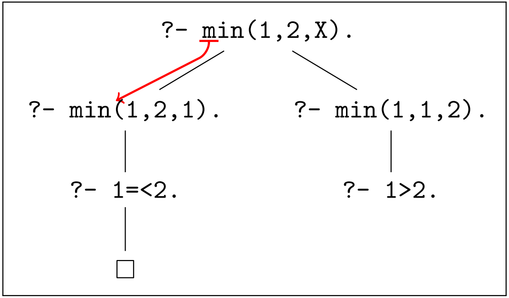
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** “`!`” prunes the search tree of a Prolog program.
Example:

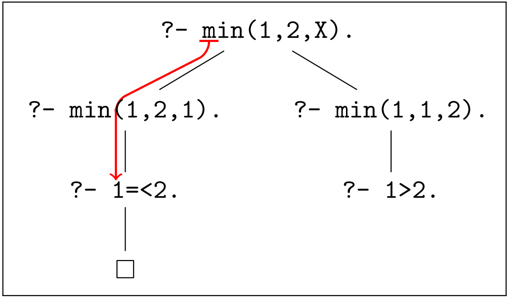
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

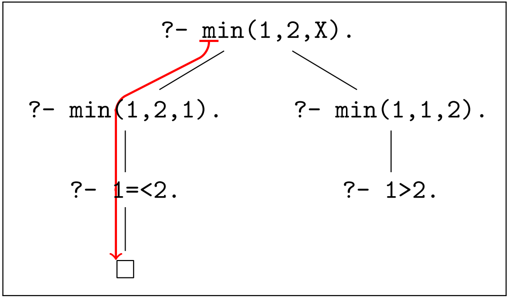
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** “`!`” prunes the search tree of a Prolog program.
Example:

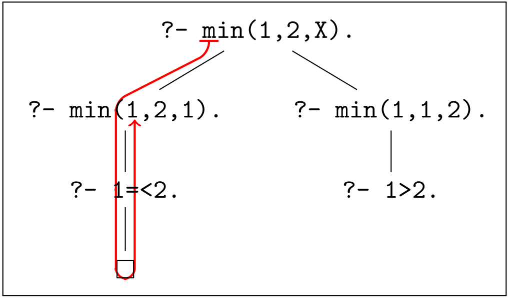
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

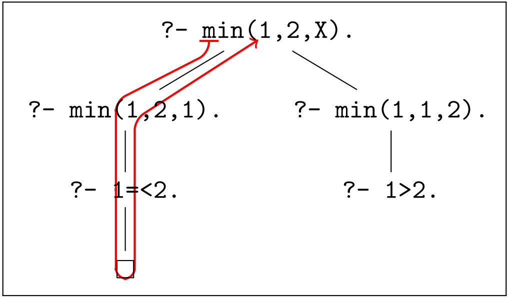
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

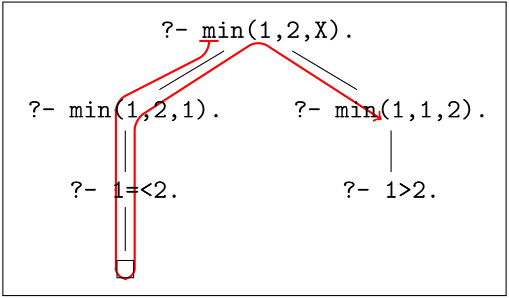
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

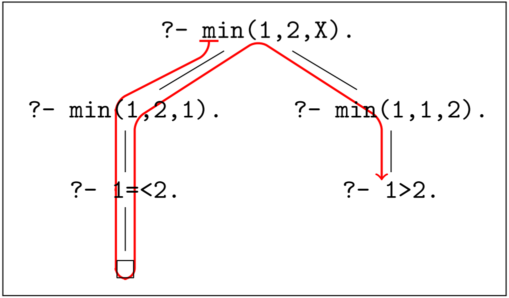
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

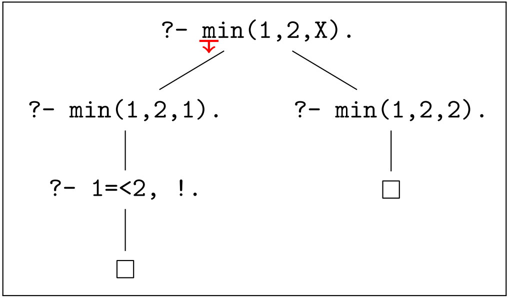
```
min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

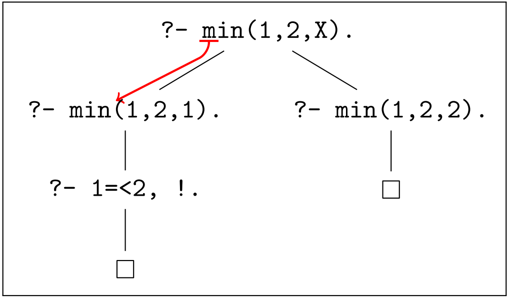
```
min(X,Y,X) :- X=<Y, !.
min(X,Y,Y).
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

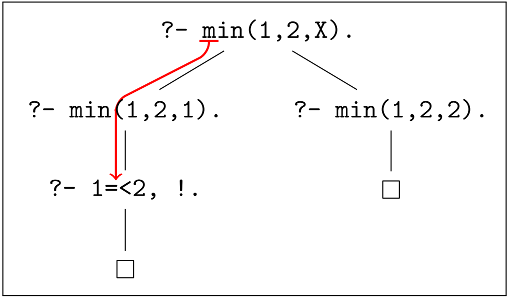
```
min(X,Y,X) :- X=<Y, !.
min(X,Y,Y).
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

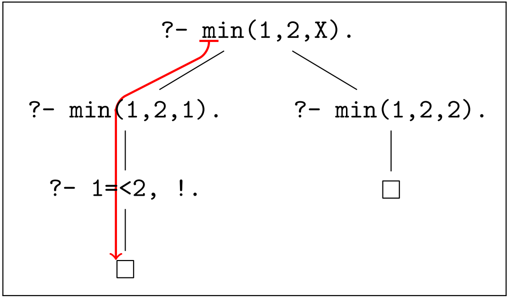
```
min(X,Y,X) :- X=<Y, !.
min(X,Y,Y).
```



Cuts

A ***cut*** “`!`” prunes the search tree of a Prolog program.
Example:

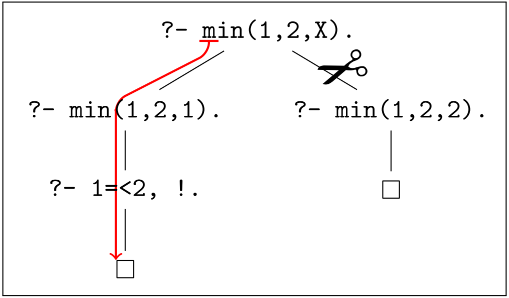
```
min(X,Y,X) :- X=<Y, !.
min(X,Y,Y).
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

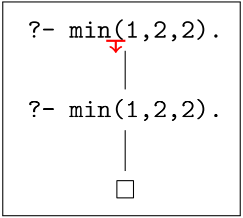
```
min(X,Y,X) :- X=<Y, !.
min(X,Y,Y).
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

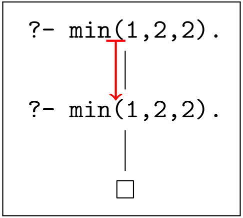
```
min(X,Y,X) :- X=<Y, !.
min(X,Y,Y).
```



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

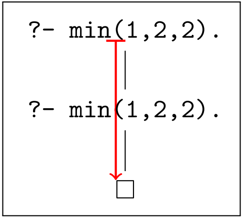
```
min(X,Y,X) :- X=<Y, !.
min(X,Y,Y).
```



Cuts

A ***cut*** "!" prunes the search tree of a Prolog program.
Example:

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



Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

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

```
?- min(1,2,2).
      |
?- min(1,2,2).
      |
   ?- 1>2.
```

Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

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

```
?- min(1,2,2).
      |
?- min(1,2,2).
      |
   ?- 1>2.
```

Cuts

A ***cut*** "`!`" prunes the search tree of a Prolog program.
Example:

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

```
?- min(1,2,2).
       |
?- min(1,2,2).
       |
   ?- 1>2.
```

If-statement

A better way to improve the original version of the "`min` program would have been to use Prolog's if statement:
Example:

```
min(X,Y,Z) :- X=<Y -> Z = X ; Z = Y.
```

The if-statement `P -> Q ; R` chooses deterministically to evaluate `Q` or `R`
depending on the "value" of `P` :

- If `P` succeeds, then `Q` is evaluated
- If `P` fails, then `R` is evaluated

Green and red cuts

- ***Green cuts*** are used to express determinism in Prolog programs
- They preserve the logical soundness of the program
- All other cuts are called ***red cuts***
- Example:

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

  does not preserve logical soundness
  because the second clause evaluated separately does not express a (logical) property of `min`

What "cut" does

- Consider the two clauses of predicate H of the form:

```
H1 :- B1,...,Bi,!,Bj,...,Bk.
H2 :- Bm,...,Bn.
```

- The goals in the sequence $B_1, \ldots, B_i$ may backtrack among themselves
- And if B_1 fails the second clause will be attempted
- If B_i succeeds, the cut is "crossed"
- And the system is committed to the current choice of clause
- All other choices are discarded
- The goals $B_j, \ldots, B_k$ may backtrack among themselves
- But if B_j fails, then the original goal H fails.
- The subsequent clause will not be attempted

Uses of "cut"

- Obtain and commit to the first solution:[1]

```
once(G) :- call(G), !.
```

- The goal `for(N,G)` executes `N` times goal `G`:

```
for(0,G) :- !.
for(N,G) :- N>0, call(G), M is N-1, for(M,G), !.
```

- `not(G)` fails if `G` succeeds
  `not(G)` succeeds if `G` fails

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

- The latter is called ***negation-as-failure***
- The Prolog operator for `not` is `\+`
- Example:

```
min(X,Y,Z) :- X=<Y -> Z = X ; Z = Y.
```

The query `\+min(1,2,X)` yields `no`

[1] The predicate `call(G)` calls the goal `G` for Prolog to solve

Pitfalls of negation-as-failure 1

- Consider the following program:

```
innocent(tom).
innocent(eve).
guilty(mike).
```

- and the query `:- innocent(peter).`
- The answer is `no`.
- An “improvement” would be to use negation-as-failure in the definition of guilty:

```
innocent(tom).
innocent(eve).
guilty(X) :- \+innocent(X).
```

- Consider the query `:- guilty(peter).`
- The answer is `yes`.
- The fact that `peter` is unknown to the system is unknown to the system

Pitfalls of negation-as-failure 2

- Consider the following program:

```
good_hotel(goedels).
good_hotel(freges).
good_hotel(tarskis).
good_hotel(quines).
expensive_hotel(goedels).
expensive_hotel(quines).
```

- And the additional clause

```
reasonable(H) :- \+expensive_hotel(H).
```

- We ask: `?- good_hotel(X), reasonable(X).`
- Answer: `X = freges`
- We ask: `?- reasonable(X), good_hotel(X).`
- Answer: `no`
- In the first query `X` is instantiated in `reasonable(X)`, in the second it isn't
- Negation-as-failure is not logical negation

Output

- Predicates:
  - `write(X).` writes (the value of) `X` to the output stream
  - `nl.` writes a “new line” to the output stream
- Using these, we can define the predicate `writeln/1` as follows:

```
writeln([X|Xs]) :- write(X), writeln(Xs).
writeln([]) :- nl.
```

- Output has no effect on the state of the logic program
- It could be replaced by `true` everywhere without changing the behaviour of the program

Input

- Predicate:
  - `read(X).` reads a value from the input stream and returns it in `X`
- Each time `read` is invoked it possibly yields a different value determined by the input stream
- It has no logical meaning and affects the behaviour of the program

Program access and manipulation

- Predicates:
  - `asserta(X)` add clause `X` to the beginning of the current program
  - `assertz(X)` add clause `X` to the end of the current program
  - `retract(X)` removes the first clause from the current program that unifies with `X`
  - `clause(H,B)` retrieve clauses of the current program whose head unifies with `H`
    - `H` must be instantiated, i.e, select a specific predicate
    - `B` is unified with the body of a clause matching `H :- B.`
    - Backtracking retrieves all matching clauses
- Example:

```
is1(X) :-
  asserta((test(A) :- A = 1,!)),
  clause(test(Y), B),
  call(B),
  X = Y.
```

Memo-functions

- `lemma(Goal)` attempts to prove `Goal` and if it succeeds stores it:
  `lemma(P) :- P, assert((P :- !)).`
-

Specification of the Fibonacci function in Prolog:

```
fib(0,0).
fib(1,1).
fib(X,Y) :-
  A is X-1,
  fib(A,P),
  B is X-2,
  fib(B,Q),
  Y is P + Q.
```

Specification of the Fibonacci function with memo-function:

```
:- dynamic fibm/2.

fibm(0,0).
fibm(1,1).
fibm(X,Y) :-
  B is X-2,
  lemma(fibm(B,Q)),
  A is X-1,
  fibm(A,P),
  Y is P + Q.
```

More on input and output

- Input predicates:
  - `seeing(F)` unify `F` with the current input stream
  - `see(F)` open file `F` as current input stream
  - `read(D)` read `D` from current input stream
  - `seen` close current input stream
- Output predicates:
  - `telling(F)` unify `F` with the current output stream
  - `tell(F)` open file `F` as current output stream
  - `write(D)` write `D` to current output stream
  - `nl` write “new line” to current output stream
  - `writeln(D)` write `D` followed by “new line” to current output stream
  - `told` close current output stream
- More output predicates:
  - `listing` write current program to output stream
  - `listing(A)` write clauses named `A` to output stream
  - `listing(A/N)` write clauses named `A` with arity `N` to output stream