5.4.1: Built-in Cut-Predicate

Goal: do not traverse certain parts of the SLD-tree when backtracking
(in order to increase efficiency or to avoid non-termination)

5.4.2: implement Meta-Predicates like negation

\[
\text{female}(X) :\neg \text{male}(X).
\]

\[
\text{scd a negation was not available yet}
\]

Reason: \{ \text{female}(X), \text{male}(X) \} is no Horn Clause

5.4.1. The Cut Predicate

Backtracking: if one reaches finite failure
or if user enters ";;" after reaching \(\Omega\).

Cut: avoid certain backtracking

Ex:

\[
f(x) = \begin{cases} 
0, & \text{if } x < 3 \\
1, & \text{if } 3 \leq x \leq 6 \\
2, & \text{if } 6 \leq x
\end{cases}
\]

\[
f(x \leq 3) \iff x < 3.
\]

\[
2 - f(A,Y) \quad 0 < Y
\]
\[ f(X, 0) : - X \leq 3. \]
\[ f(X, 1) : - 3 = \leq X, X \leq 6. \]
\[ f(X, 2) : - 6 = \leq X. \]

Observation: The conditions
\[ X \leq 3 \]
\[ 3 = \leq X, X \leq 6 \]
\[ 6 = \leq X \]

exclude each other.

If proving one of these conditions succeeds, one should not backtrack to try the other \( f \)-clauses.

Solution: Cut predicate !

- Predicate of arity 0
- Proof of ! always succeeds
- Side effect: as soon as ! has been proved, certain alternative paths of the SLD tree are not explored anymore.

The cuts in this program are "green cuts": only influence efficiency, but if one removes the cuts, one still gets the same results.
Efficiency of example program can be improved further: \( ? - f(7, Y) \).

If \( X < 3 \) succeeds in the first clause, then we will not read clause 2 + 3 (because of !).
If \( X < 3 \) fails in the first clause then there is no need to check \( 3 = \leq X \) in clause 2, because \( 7 \times 3 \) implies \( 3 = \leq X \).

\[ \implies \text{remove} \ 3 = \leq X \text{ from clause 2} \]
\[ \text{remove} \ 6 = \leq X \text{ from clause 3} \]

Now the cuts are "red cuts". Removing the cuts would yield different new answer substitutions: \(? - f(0,2)\). (if cuts are removed)

In general: What is the effect of a cut?

If a query \(? - A_1, \ldots, A_n\)

is resolved with a prog. clause \( B \vdash \neg C_1, \ldots, \neg C_k, 1, C_{k+1}, \ldots, C_m \)

using unq. \( J \) of \( A_i \) and \( B \)
then one obtains the SLD-Tree on the slide.
Cut means that no alternatives are considered anymore for those nodes between
\( A_1, \ldots, A_n \) and
\( c'(!, C_{n+1}, \ldots, C_n, A_2, \ldots, A_n) \).
But for the nodes above and below those two nodes, back-tracking works as usual.

Example to illustrate the full effect of cut:

Version without cuts.

?\(- a(X). \)

\( X=0; \ X=1; \ X=2; \ X=3; \ X=4; \ X=5 \)

Now replace the second \( b \)-clause by
\( b(X) :\!:- c(Y), \!, \!\ d(X, Y). \)

?\(- a(X). \)

\( X=0; \ X=1; \ X=5 \)

Examples for using the cut in natural programs:

* \( \text{gcd} \) (greatest common divisor)
?- gcd (12, 3, Z),
Z = 3

* remove
remove (X, Xs, Ys) if Ys results from Xs
by removing all occurrences of the
element X from the list Xs.

?- remove (1, [0, 1, 2, 1, 3], Ys).
Ys = [0, 2, 3]
The cut in clause 2 is needed to ensure that
clause 3 is only re-enabled if X ≠ Y (i.e., if
the element to be removed is not at the be-
ginning of the list).

If this cut were deleted, we would get:
?- remove (1, [0, 1, 2, 1, 3], Ys).
Ys = [0, 2, 3]; Ys = [0, 2, 1, 3]; Ys = [0, 1, 2];
Ys = [0, 1, 2, 1, 3]

5.4.2. Meta-Variables and Negation

Prolog allows the use of meta-variables:
Variables: can be instantiated by terms
meta-variables: $\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)
is pre-defined under the name "$j$" using the directive

\[ :- \text{op} (\text{iffy}, j) \]

Thus, one can ask query:

\[ ?- X = 4 ; X = 5. \]

\[ X = 4 ; X = 5 \]

One can also implement a meta-predicate for "if-then-else":

\[ \text{if} (A, B, C) \text{ should implement } \text{if } A \text{ then } B \text{ else } C \]

\[ \text{if} (A, B, C) :- A, !, B. \]

\[ \text{if} (A, B, C) :- C. \]

**C+ is needed to ensure that one does not reach clause 2 if A holds**

\[ \text{if} (A, B, C) \text{ is pre-defined in Prolog under the name } A \rightarrow B \uparrow C \]

Negation is implemented as "finite failure" ("Negation as failure"): 
"Negation as failure":

Goal: prove \( \neg A \)

\( \neg (A) :- A, !, fail. \)

\( \neg (A) \).

is pre-defined in Prolog, can also be used as prefix-operator \(+\)

\( \neg \_equal(X, Y) :- \neg (X = Y). \)

?- \neg \_equal(1, 2).

true

?- \neg \_equal(1, X).

false

Negation in Prolog uses two assumptions:

1. If a query doesn't hold, then this is determined in finite time.
But: \( \text{? - not even (1)} \),
does not return "true", because even(1) doesn't terminate. Although "even(1)" doesn't hold, we can't detect it in finite time.

2. Closed World Assumption: If something can't be proved with our program, then it must be false.

\( \text{? - not (even (-2))} \).

true

Since proof of even (-2) fails.