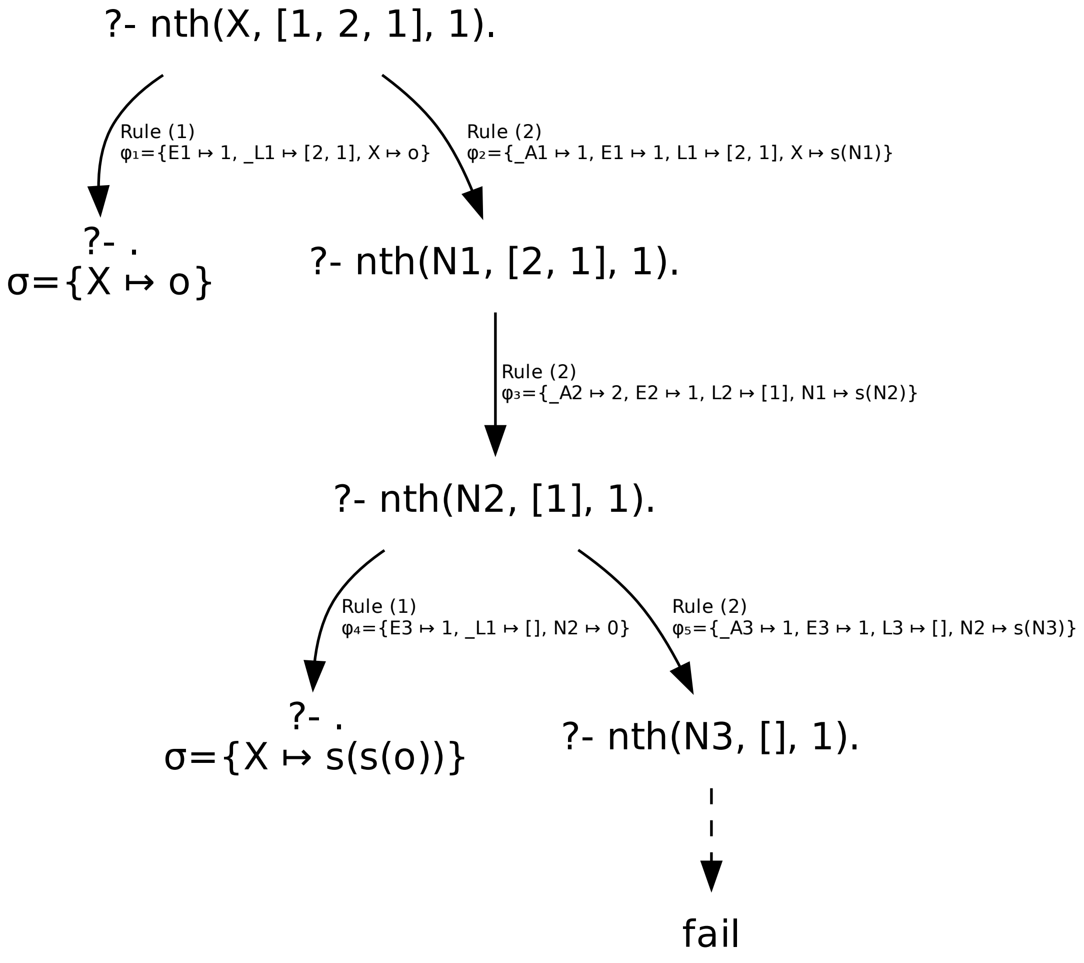
?- nth(X, [1, 2, 1], 1).
Rule (1)
φ₁={E1 ↦ 1, _L1 ↦ [2, 1], X ↦ o}
Rule (2)
φ₂={_A1 ↦ 1, E1 ↦ 1, L1 ↦ [2, 1], X ↦ s(N1)}
?- .
σ={X ↦ o}
?- nth(N1, [2, 1], 1).
Rule (2)
φ₃={_A2 ↦ 2, E2 ↦ 1, L2 ↦ [1], N1 ↦ s(N2)}
?- nth(N2, [1], 1).
Rule (1)
φ₄={E3 ↦ 1, _L1 ↦ [], N2 ↦ 0}
Rule (2)
φ₅={_A3 ↦ 1, E3 ↦ 1, L3 ↦ [], N2 ↦ s(N3)}
?- .
σ={X ↦ s(s(o))}
?- nth(N3, [], 1).
fail

Abbildung 1: SLD-Resolution-Tree for the goal '?- nth(X, [1, 2, 1], 1).'

1 - SLD Resolution 1

2 - Resolution with Cuts

We create the SLD tree first.

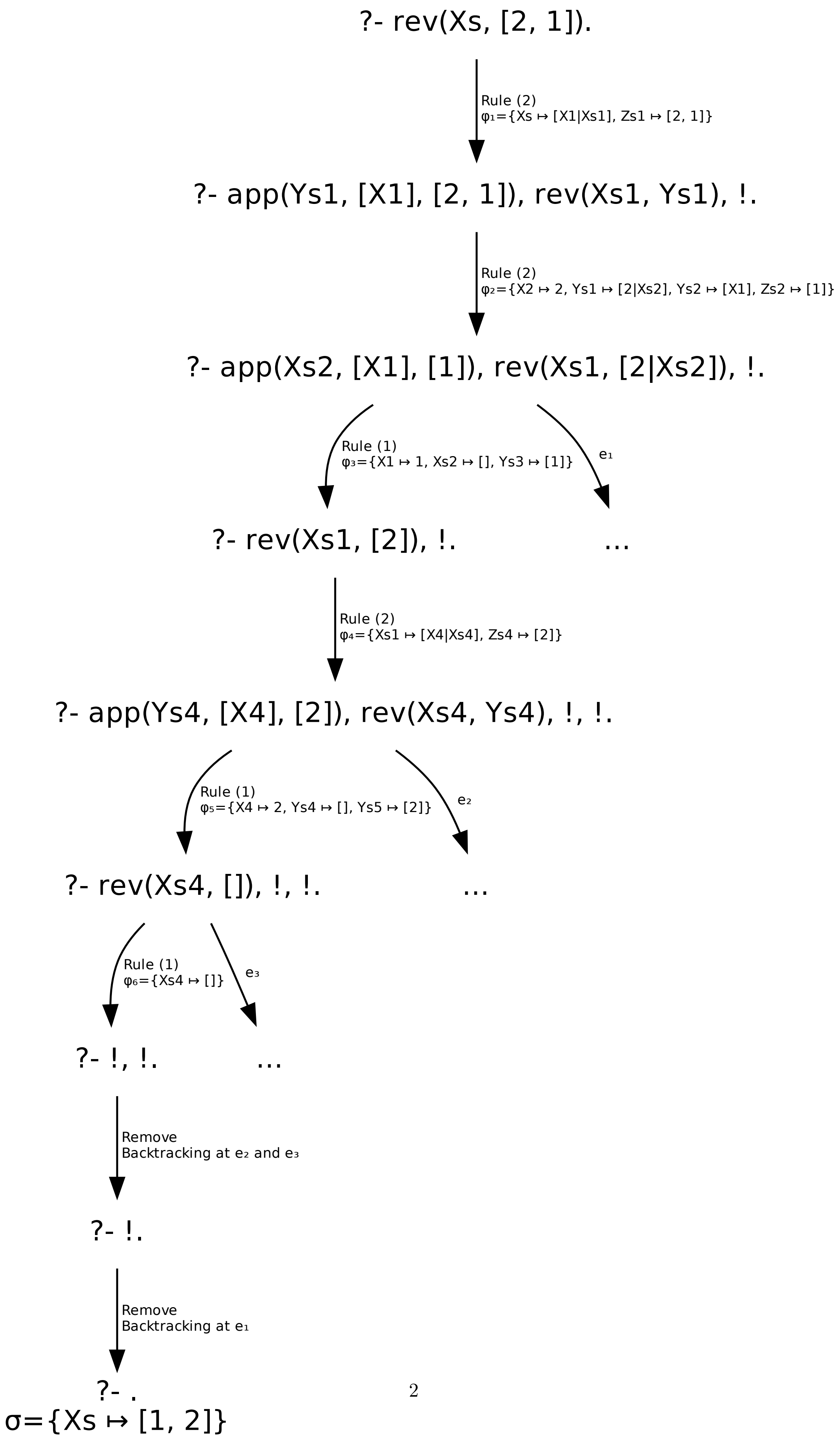
?- rev(Xs, [2, 1]).
Rule (2)
φ₁={Xs ↦ [X1|Xs1], Zs1 ↦ [2, 1]}
?- app(Ys1, [X1], [2, 1]), rev(Xs1, Ys1), !.
Rule (2)
φ₂={X2 ↦ 2, Ys1 ↦ [2|Xs2], Ys2 ↦ [X1], Zs2 ↦ [1]}
?- app(Xs2, [X1], [1]), rev(Xs1, [2|Xs2]), !.
Rule (1)
φ₃={X1 ↦ 1, Xs2 ↦ [], Ys3 ↦ [1]}
e₁
?- rev(Xs1, [2]), !.
...
Rule (2)
φ₄={Xs1 ↦ [X4|Xs4], Zs4 ↦ [2]}
?- app(Ys4, [X4], [2]), rev(Xs4, Ys4), !, !.
Rule (1)
φ₅={X4 ↦ 2, Ys4 ↦ [], Ys5 ↦ [2]}
e₂
?- rev(Xs4, []), !, !.
...
Rule (1)
φ₆={Xs4 ↦ []}
e₃
?- !, !.
...
Remove
Backtracking at e₂ and e₃
?- !.
Remove
Backtracking at e₁
?- .
σ={Xs ↦ [1, 2]}

Abbildung 2: SLD-Resolution-Tree for the goal '?- rev(Xs, [2, 1]).'

Prolog yields the same solutions.

```
% ?- rev(Xs, [2, 1]).
%    |- {Xs -> [X1|Xs1], Zs1 -> [2,1]} (2nd rule)
% ?- app(Ys1, [X1], [2, 1]), rev(Xs1, Ys1), !.
%    |- {X2 -> 2, Ys1 -> [2|Xs2], Ys2 -> [X1], Zs2 -> [1]} (2nd rule)
% ?- app(Xs2, [X1], [1]), rev(Xs1, [2|Xs2]), !. (*)
%    |- {X1 -> 1, Xs2 -> [], Ys3 -> [1]} (1st rule)
% ?- rev(Xs1, [2]), !.
%    |- {Xs1 -> [X4|Xs4], Zs4 -> [2]} (2nd rule)
% ?- app(Ys4, [X4], [2]), rev(Xs4, Ys4), !, !. (**)
%    |- {X4 -> 2, Ys4 -> [], Ys5 -> [2]} (1st rule)
% ?- rev(Xs4, []), !, !. (***)
%    |- {Xs4 -> []} (1st rule)
% ?- !, !.
%    Remove Backtracking at (***) and (**).
% ?- !.
%    Remove Backtracking at (*).
% ?- .
%    Output: Xs = [1, 2]
```

Therefore, the solution is `Xs = [1, 2]`.

If the Cut-operator is omitted, `rev` can still be used in both directions if the second parameter is a free variable, but does not terminate after finding a solution.

If the order is also changed, the *forward direction* would terminate again, but the search will no longer terminate if the first parameter is a free variable.

So we can see that the Cut-operator makes a lot of sense here.

3 - SLD Resolution 2

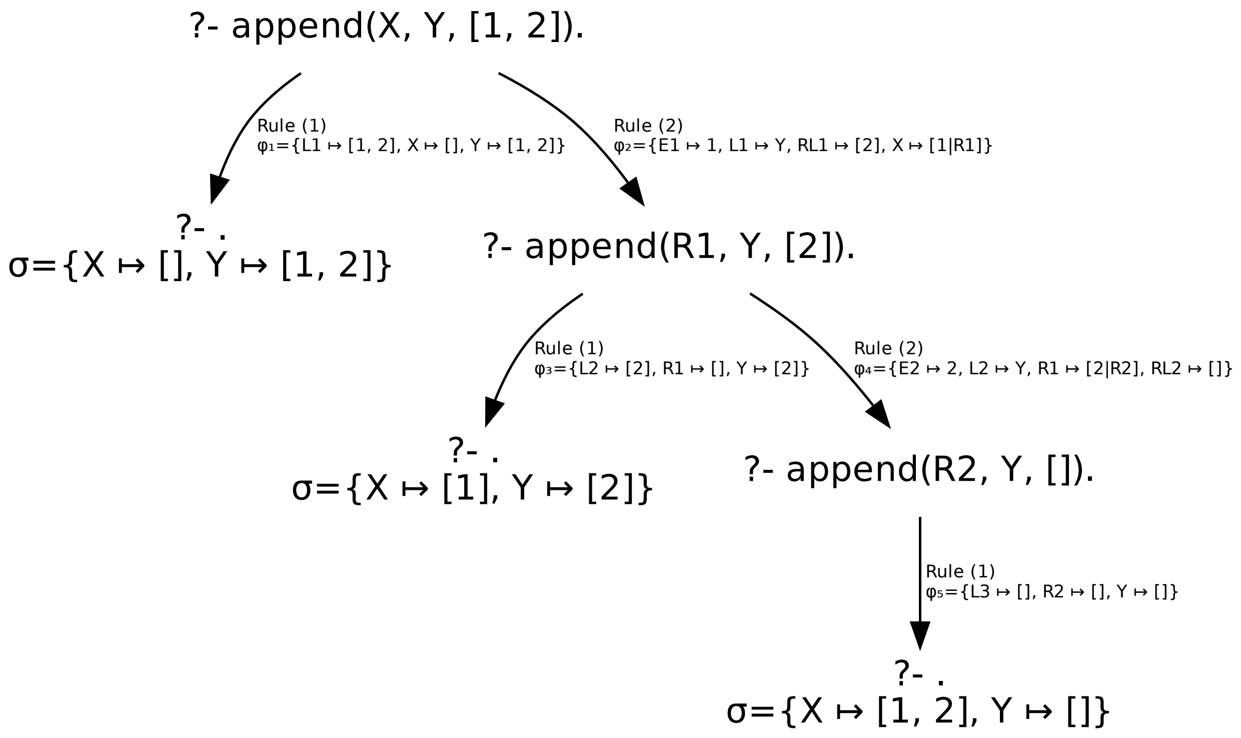

Abbildung 3: SLD-Resolution-Tree for the goal ‘?- append(X, Y, [1, 2]).‘

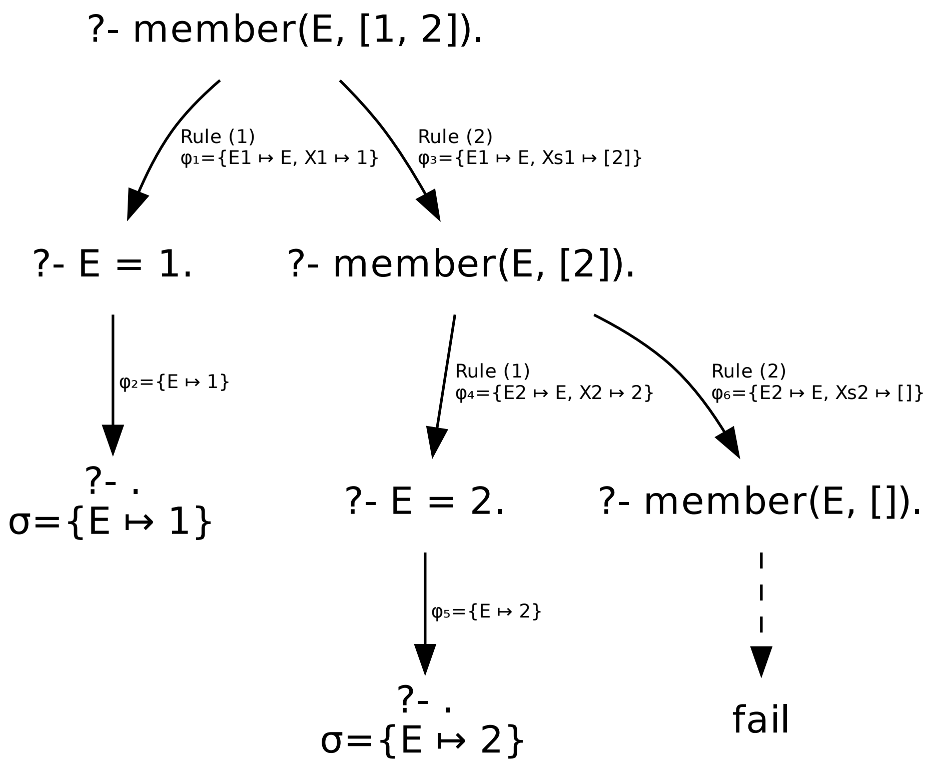

Abbildung 4: SLD-Resolution-Tree for the goal '?- member(E, [1, 2]).'

4 - Resolution

We create the SLD tree first.

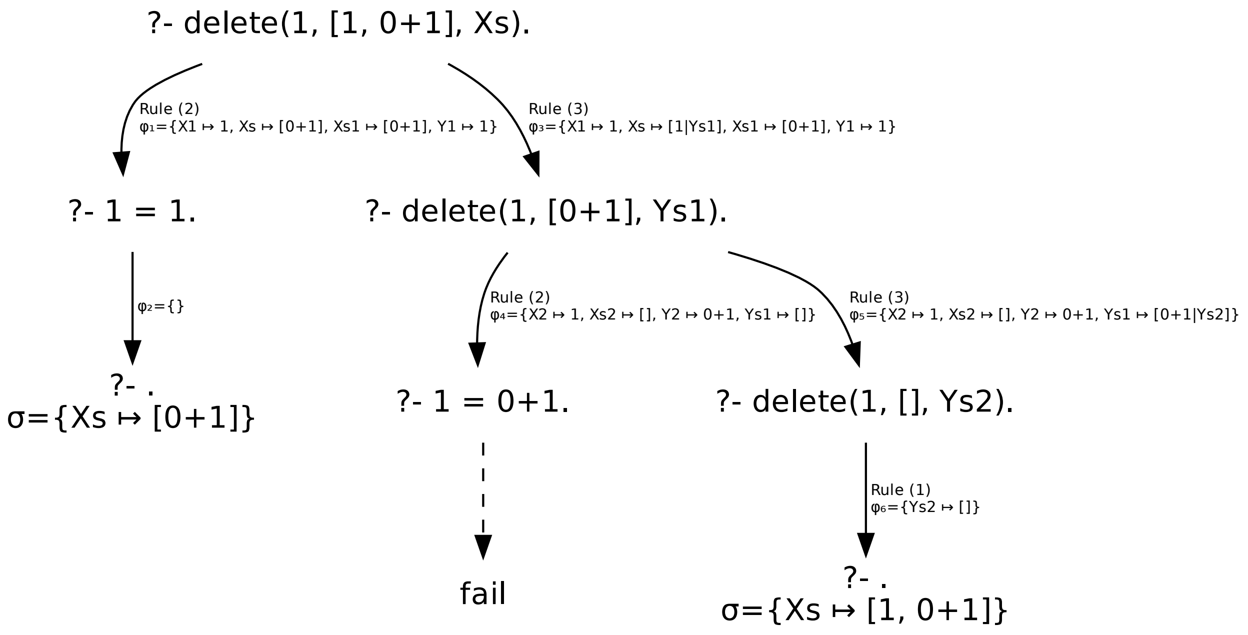
?- delete(1, [1, 0+1], Xs).
Rule (2)
φ1={X1 ↦ 1, Xs ↦ [0+1], Xs1 ↦ [0+1], Y1 ↦ 1}
Rule (3)
φ3={X1 ↦ 1, Xs ↦ [1|Ys1], Xs1 ↦ [0+1], Y1 ↦ 1}
?- 1 = 1.
?- delete(1, [0+1], Ys1).
φ2={}
Rule (2)
φ4={X2 ↦ 1, Xs2 ↦ [], Y2 ↦ 0+1, Ys1 ↦ []}
Rule (3)
φ5={X2 ↦ 1, Xs2 ↦ [], Y2 ↦ 0+1, Ys1 ↦ [0+1|Ys2]}
?- .
σ={Xs ↦ [0+1]}
?- 1 = 0+1.
?- delete(1, [], Ys2).
Rule (1)
φ6={Ys2 ↦ []}
fail
?- .
σ={Xs ↦ [1, 0+1]}

Abbildung 5: SLD-Resolution-Tree for the goal '?- delete(1, [1, 0+1], Xs).'

Computing solutions according to the Prolog search procedure works as follows.

```
% ?- delete(1, [1, 0+1], Xs). (*)
%    |- {X1 -> 1, Xs -> [0+1], Xs1 -> [0+1], Y1 -> 1} (2nd rule)
% ?- 1 = 1.
%    |- {}
% ?- .
%    Output: Xs = [0+1]
%    Reset with 3rd rule at (*).
%    |- {X1 -> 1, Xs -> [1|Ys1], Xs1 -> [0+1], Y1 -> 1}
% ?- delete(1, [0+1], Ys1). (**)
%    |- {X2 -> 1, Xs2 -> [], Y2 -> 0+1, Ys1 -> []} (2nd rule)
% ?- 1 = 0+1.
%    Failure
%    Reset with 3rd rule at (**).
%    |- {X2 -> 1, Xs2 -> [], Y2 -> 0+1, Ys1 -> [0+1|Ys2]}
% ?- delete(1, [], Ys2).
%    |- {Ys2 -> []} (1st rule)
%    Output: Xs = [1, 0+1]
```

To remove only the first occurrence of the 1st parameter from the list, we add a Cut to the 2nd rule.

```
deleteFirst(_, [],     []).
deleteFirst(X, [Y|Xs], Xs)     :- X = Y, !.
deleteFirst(X, [Y|Xs], [Y|Ys]) :- deleteFirst(X, Xs, Ys).
```

To remove all occurrences of the 1st parameter from the list, we must adjust the 2nd rule so that after deleting the 1st occurrence, the remaining list is searched for further occurrences. Furthermore, in the 3rd rule it must be additionally ensured that the 1st parameter of `delete` and the 1st list element are unequal.

```
deleteAll(_, [],     []).
deleteAll(X, [Y|Xs], Ys)     :- X = Y, deleteAll(X, Xs, Ys).
deleteAll(X, [Y|Xs], [Y|Ys]) :- X \= Y, deleteAll(X, Xs, Ys).
```

5 - Negation as Failure

`nodup` checks whether a list contains no duplicates.

```
nodup([]).
nodup([X|Xs]) :- nodup(Xs), \+ member(X, Xs).

member(X, [X|_ ]).
member(X, [_|Xs]) :- member(X, Xs).
```

`neq` corresponds to inequality in Prolog.

```
neq(X, Y) :- \+ X = Y.
```

It should be noted that the *Occurs Check* in SWI-Prolog is disabled by default (for efficiency reasons). Occurs checking can be enabled using the flag `occurs_check`-Flag. Alternatively, the predefined predicate `unifies_with_occurs_check` can be used.

`remove(X, Xs, Ys)` removes all occurrences of `X` from the list `Xs`.

```
remove(_, [],      []).
remove(X, [X|Ys], Zs)      :- remove(X, Ys, Zs).
remove(X, [Y|Ys], [Y|Zs]) :- neq(X, Y), remove(X, Ys, Zs).
```

`nub(Xs, Ys)` holds if `Ys` is the list `Xs` without duplicates.

```
nub([],      []).
nub([X|Xs], [X|Ys]) :- remove(X, Xs, Zs), nub(Zs, Ys).
```

6 - Arithmetic

Both definitions are straight forward.

```
fromPeano(o,     0).
fromPeano(s(N), X) :- fromPeano(N, Y), X is Y + 1.

toPeano(0, o).
toPeano(X, s(N)) :- X > 0, Y is X - 1, toPeano(Y, N).
```

However, in Prolog one would expect these predicates be identical, with different argument order. Hence let's check how the two implementations behave if the are called in a way they are not supposed to be used.

```
?- fromPeano(X, N).
X = o,
N = 0;
X = s(o),
N = 1;
X = s(s(o)),
N = 2;
X = s(s(s(o))),
N = 3.

?- fromPeano(X, 3).
X = s(s(s(o)));
< infinite loop >
```

In contrast, using `toPeano` in the reverse order is not possible at all.

```
?- toPeano(X, s(s(o))).
ERROR: Arguments are not sufficiently instantiated
```

If you want to integrate both definitions into one, you can use the predicate `var/1`, which checks whether a parameter is unbound.

```
peanoInt(N,    X) :- var(N), var(X), !, fromPeano(N, X).
peanoInt(N,    X) :- var(N), \+ var(X), !, toPeano(X, N).
peanoInt(o,    0).
peanoInt(s(N), X) :- peanoInt(N, Y), X is Y + 1.
```

This is a bit more complicated, but can now also be used to enumerate all integers larger than a given Peano number.

```
?- peanoInt(s(s(X)), Y).
X = o,
Y = 2;
X = s(o),
Y = 3;
X = s(s(o)),
Y = 4.
```

```
intToBin(0, o).
intToBin(1, i).
intToBin(M, B) :- M > 1,
                  N is M div 2,
                  intToBin(N, X),
                  (0 is M mod 2 -> B = o(X); B = i(X)).

binToInt(o,    0).
binToInt(i,    1).
binToInt(o(B), X) :- binToInt(B, Y), X is Y * 2.
binToInt(i(B), X) :- binToInt(B, Y), X is Y * 2 + 1.
```

Starting from the given rule, we can add a rule for the special case of division.

```
eval(E1/E2, R) :- !, eval(E2, R2), R2 \= 0, eval(E1, R1), R is R1 / R2.
eval(E,     R) :- R is E.
```

With the cut, it is guaranteed that we fist check whether `E2` evaluates to zero before computing a division. Hence, we get the following.

```
?- eval(4/2, R).
R = 2.
?- eval(4/0, R).
false.
?- eval(4/(2-2), R).
false.
```

If the division occurs deeper in a computation, this is not detected.

```
?- eval(2+4/0, R).
ERROR: Arithmetic: evaluation error: `zero_divisor'
```

We have to check every evaluation and define `eval` for all possible cases of the expressions.

```
eval(E1/E2, R) :- !, eval(E2, R2), R2 \= 0, eval(E1, R1), R is R1 / R2.
eval(E1+E2, R) :- !, eval(E1, R1), eval(E2, R2), R is R1 + R2.
eval(E1*E2, R) :- !, eval(E1, R1), eval(E2, R2), R is R1 * R2.
eval(E1-E2, R) :- !, eval(E1, R1), eval(E2, R2), R is R1 - R2.
eval(R,     R).
```

The last rule is necessary to evaluate numbers. Since the last rule overlaps with all the other rules, we have to add a cut to every rule, such that the last rule cannot be chosen if any of the other rules match.