A Prolog program is a data base of clauses. It can read its own code during runtime using the pre-defined predicate `clause/2`. Here, `clause(t_1, t_2)` is true iff there is a prog. clause \[ B : - C_1, \ldots, C_n \]

such that `clause(t_1, t_2)` unifies with \[ \text{clause}(B, (C_1, \ldots, C_n)) \]

**Ex:** Prog contains 2 clauses:

`times(_, 0, 0).` is regarded as `times(_, 0, 0) :- true.`

\[ \text{times}(X, Y, Z) : - \quad Y > 0, \quad Y_1 \text{ is } Y-1, \quad \text{times}(X, Y_1, Z_1), \quad Z \text{ is } Z_1 + X. \]

?- `clause(t_1, t_2), Body).`

\[ \begin{align*}
Y = 0, \quad Z = 0, \quad \text{Body} = \text{true} \quad ; \\
\text{Body} = (\ Y > 0, \quad Y_1 \text{ is } Y-1, \quad \text{times}(X, Y_1, Z_1), \quad Z \text{ is } Z_1 + X) \\
\end{align*} \]

stands for

\[ \begin{align*}
Y > 0, \quad (Y_1 \text{ is } \ldots, \quad (\text{times}(X \ldots), \quad Z \text{ is } \ldots)) \\
\end{align*} \]

Up to now, we considered the data base of prog. clauses to be static. But it is possible to
modify a Prolog-program during its execution (⇒ dynamic).

assert/1: assert(t) always succeeds and as a side-effect, the clause t is added at the end of the program.

Ex: times-prog.

?- assert(p(0)). ← fact p(0).
true

?- p(X).

X = 0.

?- assert(square(X,Y) :- times(X,X,Y)).
true

?- square(3,Y).

Y = 9.

There are also predicates asserta/1 and assertz/1.

... at the end of the prog (like assert).

⇒ adds clause at the beginning of the prog.
One can only modify program clauses for dynamic predicates: All predicates introduced by assert are dynamic. But the pred. in the program are static unless they are explicitly declared as dynamic.

To this end, one needs a corresponding directive in the program:

```
:- dynamic times/3.
times(_, 0, 0).
times(X, Y, Z) :- Y > 0, Y1 is Y - 1, times(X, Y1, Z1), Z is Z1 + X.
```

?- asserta(times(X, Y, X)).
true
?- clause(times(X, Y, Z), B).
X = X, Y = Y, B = true
```
retract/1 is used to remove clauses (of dynamic predicates).

retract(t) succeeds if a prog. clause unifies with t. As a side effect, the first prog. clause that unifies with t
```
Ex: times
?- retract(times(X,Y,Z) :- B).

A first clause for times is deleted.

By pressing ; several times, one can delete all clauses for times.

Instead:
?- retract(times(X,Y,Z)).

would only delete facts of times

Prog. clauses can also contain assert and retract \( \rightarrow \) prog. can modify itself while it is running. \( \rightarrow \) Can lead to non-understandable programs \( \text{\emph{use assert + retract with care}} \).

A useful application of assert + retract is to save intermediate results for later use.

Ex: We want to save intermediate results of multiplication.
\( \Rightarrow \) Create table which stores \( X \times Y \) for all numbers \( X \) and \( Y \) between 0 and 9.

\[ L = [0,1,2,3,4,5,6,7,8,9] \]

\[ \text{member} (X,L), \quad \text{member} (Y,L), \quad \text{member}(X,L \times Y), \]
member(Y,L),
\[ t \text{ is } X \cdot Y, \]
\[ \text{assert(times(X,Y,t))}, \]
\[ \text{fail}. \]

\[ \text{members(X,L,Ys)}:=-\]
\[ \text{member(X,Ys)}. \]

\[ \text{creates 100 times-facts due to backtracking} \]

?- maketable.
false
?- times(X,Y,8).
\[ X=1, \ Y=8 \ ; \ X=2, \ Y=4 \ ; \ X=4, \ Y=2 \ ; \ X=8, \ Y=1. \]

**Ex:** Predicate *findall* should not only find the first solution to a query (and wait for the user to press `,`) but it should find all solutions to a query. This predicate is predefined in Prolog, but we could define it ourselves with *assert* and *retract*.

\[ \text{findall}(t,g,l) \text{ is true iff the following holds:} \]
\[ \text{One builds up the complete SLD-tree for the query } g \]
\[ \text{and computes all answer substitutions } \sigma_1, \ldots, \sigma_n. \]
\[ \text{Then } \text{findall}(t,g,l) \text{ is true iff } l \text{ is the list } \]
\[ [\sigma_1(t), \ldots, \sigma_n(t)]. \]

**E.g.: Consider the family-example.**

?- findall(Y,fatherOf(gerd,Y),L).
\[ L = [\text{susanne, peter}]. \]

?- findall(fatherOf(gerd,Y),fatherOf(gerd,Y),L).
\[ L = \text{[fatherOf(gerd, susanne), fatherOf(gerd, peter)]} \]

Now we implement \text{findall} ourselves:

\begin{verbatim}
findall (X, Query, Xlist) :- Query,
    \text{assert (answer(X))},
    \text{collect_answers(Xlist)}.
\end{verbatim}

\begin{verbatim}
collect_answers ([X|Rest]) :- \text{retract (answer(X))},
    \text{collect_answers(Rest)}.
\end{verbatim}

Another application of meta-programming is the use of Prolog to write interpreters for programming languages. In particular, one can implement interpreters for Prolog (or variants of Prolog) in Prolog easily.

\begin{itemize}
  \item Simple Meta-Interpreter for Prolog:
    \begin{verbatim}
    prove (goal) :- goal.
    \end{verbatim}
  \end{itemize}

This will serve as the basis for other interpreters for
variants of logic programming.
prove(true) :- !.
prove((Goal1, Goal2)) :- !, prove(Goal2), prove(Goal1).
prove(Goal) :- clause(Goal, Body), prove(Body).

Now one can modify this interpreter to experiment with alternative semantics of logic programming.
We now implement an interpreter which handles sequences of literals from right to left:
prove(true) :- !.
prove((Goal1, Goal2)) :- !, prove(Goal2), prove(Goal1).
prove(Goal) :- clause(Goal, Body), prove(Body).

Another possibility is a meta-interpreter which also computes the length of proofs (i.e., the number of needed resolution steps):
prove(true, 0) :- !.
prove((Goal1, Goal2), N) :- !, prove(Goal1, N1), prove(Goal2, N2),
N is N1 + N2.
prove(Goal, N) :- clause(Goal, Body), prove(Body, N1),
N is N1 + 1.