Prolog can manipulate terms (Sect 5.6.1) and programs (Sect 5.6.2). In particular, a program can manipulate itself while it is running.

5.6.1. Manipulation of terms and formulas

pre-defined predicates to manipulate/access/recognize certain forms of terms:

- **number/1**: checks whether arg. is a number
- **var/1**: \( \forall t \) \( \text{var}(t) \) is true iff \( t \) is an (uninstantiated) variable

\[
? - \text{var}(X).
\]

true

\[
? - X = 2, \text{var}(X).
\]

false

- **nonvar/1**: \( \forall t \) \( \text{nonvar}(t) \) is true iff \( t \) is no variable

\[
? - \text{nonvar}(a).
\]

true

\[
? - X = 2, \text{nonvar}(X).
\]

\( X = 2 \)

\[
? - \text{nonvar}(X). \quad \text{although there is an instantiation}
\]
* atomic/1: atomic(t) is true iff t is a function symbol of arity 0 or a number

?- atomic(a).  true
?- atomic(-2).  true
?- atomic(a(a)).  false
?- atomic(X).  false

* Compound/1: Compound(t) is true iff t is a term/formula which does not just consist of a symbol of arity 0 or a number or a variable

?- Compound(a).  false
?- Compound(X).  false
?- Compound(a+2).  true
?- Compound(a(a)).  true

These predicates can be used to recognize certain forms of terms. But we also want to extract certain parts of terms (decomposition) and to con-
struct new terms.
Solution: transform terms to lists or vice versa

\[ f(a, b) \text{ can be transformed to } \left[ f, a, b \right] \]

Pre-defined predicate: = \( \div \) \( \times \) / 2

(infix notation)

\[ t = \ldots \text{l is true iff } \]
\[ \text{l is the representation of the term } t \]

?- \( f(a, b) = \ldots \text{l} \).
\[ \text{l} = \left[ f, a, b \right] \]

?- \( 1 + 2 = \ldots \text{l} \).
\[ \text{l} = \left[ +, 1, 2 \right] \]

?- \( f(g(a), b) = \ldots \text{l} \).
\[ \text{l} = \left[ f, g(a), b \right] \]
?- T = .. [f, a, b].
T = f(a, b)
?- T = .. [f].
T = f.
?- X = .. Y.
error
?- X = .. [Y, a, b].
error
?- X = .. [f(X)].
error

Example for using =..: Represent and enlarge different geometrical figures

square (Side) ← term to represent \( \square \) \( \text{Side} \)
rectangle (Side1, Side2) ← \( \square \) \( \frac{\text{Side1}}{\text{Side2}} \)

Triangle (Side1, Side2, Side3)
Circle (Radius)

Reg: enlarge (square (Side), Factor, square (N Side)) :- NSide is Factor \( \times \) Side.
enlarge (rect (S1, S2), F, rect (NS1, NS2)) :- NS1 is F \( \times \) S1, NS2 is F \( \times \) S2.
enlarge (triangle \( S_1, S_2, S_3 \)), \ldots \) :- \ldots
enlarge (circle \( R \)), \ldots \) :- \ldots

Disadvantage: new enlarge-clause for each geometrical figure, although all these clauses essentially do the same.

Better solution:

\[
\text{enlarge}(\text{Fig}, \text{Factor}, \text{NFig}) :- \text{Fig} = \ldots \left[ \text{Type} \mid \text{Param} \right], \\
\text{multlist}(\text{Param}, \text{Factor}, \text{NParam}), \\
\text{NFig} = \ldots \left[ \text{Type} \mid \text{NParam} \right].
\]

\[
\text{multlist}(\left[ \text{I} \right], -, \left[ \text{I} \right]).
\]

\[
\text{multlist}(\left[ \text{EX} \mid \text{L} \right], \text{Factor}, \left[ \text{NX} \mid \text{NL} \right]) :- \text{NX} \text{ is Factor} \& X, \\
\text{multlist}(\left[ \text{L} \right], \text{Factor}, \text{NL}).
\]

There are additional predicates to access/manipulate parts of terms:

- \text{functor/3}:
  
  \[
  \text{functor}(t, f, n) \text{ is true iff} \]

  \[
  \text{f/n is the leading fun/ped symbol} \& t
  \]

  \[
  \text{?- functor(g(f(x)), X, g), F, N).} \\
  
  F = g, N = 3
  \]

  \[
  \text{?- functor(T, g, 3).}
  \]
\[ T = g(X, Y, Z). \]

- \texttt{avg/3}: \texttt{arg} \((n, t, a)\) is true if \(a\) is the \(n\)-th argument of \(t\)

\[ ?- \texttt{arg} \((3, g(f(x), x, g), A). \]

\[ A = g \]

\[ ?- \texttt{functor} \((D, \text{date}, 3), \]
\[ \texttt{arg} \((1, D, 19), \]
\[ \texttt{arg} \((2, D, 6), \]
\[ \texttt{arg} \((3, D, 2015). \]

\[ D = \texttt{date}(13, 6, 2015). \]

**Ex:** Predicate to check whether a term is variable-free:

\[ \texttt{ground}(T) :- \texttt{nonvar}(T). \]

\[ T = .. [ \texttt{Functor} \mid \texttt{Argumentlist}], \]

\[ \texttt{groundlist} \((\texttt{Argumentlist}). \]

\[ \texttt{groundlist} \(([])). \]

\[ \texttt{groundlist} \((\llbracket \texttt{T} \mid \texttt{T}\rrbracket) :- \texttt{ground}(T), \texttt{groundlist}(\texttt{Ts}). \]
5.6.2 Manipulation of Programs

Prolog - prog ≤ data base of clauses which can be read and modified.

?- clause (t₁, t₂).

is true iff there is a program clause

\[ B := C_1, \ldots, C_n \quad \text{such that} \]

\[ \text{clause } (t₁, t₂) \text{ unifies with} \]

\[ \text{clause } (B, (C_1, \ldots, C_n)). \]

Ex: \[ \text{times } (-, 0, 0). \]

\[ \text{times } (X, Y, Z) := Y > 0, \forall \text{ is } Y - 1, \text{times } (X, Y, Z), \]

\[ Z \text{ is } Z + X. \]

?- clause (times (X, Y, Z), Body).

\[ Y = 0, \ Z = 0, \ Body = \text{true}; \]

\[ Body = (Y > 0, \forall \text{ is } Y - 1, \ldots, Z \text{ is } Z + X). \]
While "clause" can be used to read the code of the running program, there also exist predicates that can modify the text of the running program:

```prolog
assert(+1) and retract(+1)
```

?- assert(t).
Proof always succeeds, but as a side-effect, the clause t is added at the end of the program. (The predicate asserta(t) adds the clause t at the beginning of the program. The pred. assertz is like assert.)

**Ex:**

```prolog
?- assert(p(0)).
```

```
tme
?- p(X).
X = 0
?- clause(p(X), B).
X = 0, B = tme
?- assert(square(X,Y) :- times(X,X,Y)).
tme
```

Clauses built with "clause" cannot be asserted. "{clause" is static

Predicates can be static or dynamic. By default, all predicates in the prog. are static.
Clauses for static predicates cannot be added or removed by \texttt{assert} + \texttt{retract}.

Predicates introduced by "assert" are dynamic. Moreover, predicates in the program can be declared to be dynamic by a corresponding directive:

\textbf{Ex:} \begin{align*}
\text{:- dynamic } & \text{times/3.} \\
\text{times } & (-, 0, 0).
\end{align*}
\begin{align*}
\text{times } & (X, Y, Z) \leftarrow Y>0. \\
\end{align*}

\begin{itemize}
\item ?- \text{times}(2, 3, Z).
\item \text{Z=6}
\item ?- \text{asserta( times(X,Y,X) )}.
\item \text{true}
\item ?- \text{clause( times(X,Y,Z), B )}.
\item \text{X=Z, Y=X, B=\text{true}}
\end{itemize}

?- \text{retract( t) }

proof succeeds iff \texttt{true} is a prog. clause that unifies with \texttt{t}. As a side effect, this prog. clause is removed.

\textbf{Ex:} ?- \text{retract( times(X,Y,Z) :- Body) }.
\begin{itemize}
\item Y=1, Body=\text{true} ; \text{ - removes times(X,Y,Z)}
\end{itemize}
X = 0, Y = 0, Body = true; \leftarrow \text{removes times}(0,0)

\begin{align*}
\text{Body} &= \ldots \leftarrow \text{removes times}(X, Y) := Y > 0, \ldots
\end{align*}

assert + retract can lead to completely non-understandable programs \Rightarrow use them only for certain purposes.

Sensible use of assert + retract: compute results and store them for later use.

\textbf{Ex:} Store results of computations in a table to re-use these results later on and avoid their repeated re-computation.

\begin{tabular}{|c|c|c|c|}
\hline
X & Y & \ldots & g \\
\hline
0 & 0 & 0 & 0 \\
1 & 0 & \ldots & g \\
\vdots & \vdots & \vdots & \vdots \\
9 & 0 & g & 89 \\
\hline
\end{tabular}

\begin{align*}
\text{member}(X, [X | Y | Z]), \quad & \text{member}(X, [X | Y | Z]) := \\
& \text{member}(X, L). \\
\text{table} := & [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], \\
& \text{member}(X, L), \\
& \text{member}(Y, L), \\
& Z \text{ is } X \times Y, \\
& \text{assert} (\text{mult}(X, Y, Z)), \\
& \text{fail.} \quad \leftarrow \text{enforces backtracking}
\end{align*}
\[ X \text{ and } Y \text{ will range over all numbers from } 0, \ldots, 9 \]
\[ 100 \text{ new facts are added to the program.} \]

?- make_table.
false
?- mclt(X, Y, 8).
X = 1, Y = 8;
X = 2, Y = 4;
X = 4, Y = 2;
X = 8, Y = 1.

There exists a pre-defined predicate `findall/3` which finds all solutions to a query (i.e., without needing the user to press `.`).

`findall(T, G, L)` is true iff the following holds:

- Prolog tries to prove the query `G` and builds up the full SLD tree.
- Then it collects all answer substitutions
\( \sigma_1, \ldots, \sigma_n \) (in left-to-right depth-first search).

*Then \( \text{findall}(t, g, l) \) is true iff \( l \) is the list \( [\sigma_1(t), \sigma_2(t), \ldots, \sigma_n(t)] \).*

Ex: family program including the rule

\[
\text{fatherOf}(F, C) : \neg \text{married}(F, W), \text{motherOf}(W, C).
\]

? - \( \text{findall}(C, \text{fatherOf}(\text{gerd}, C), L) \).

\[
L = [\text{susanne}, \text{peter}]
\]

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

\[
L = [\text{fatherOf}(\text{gerd}, \text{susanne}), \text{fatherOf}(\text{gerd}, \text{peter})]
\]

\( \text{findall} \) could be programmed ourselves using \texttt{assert} and \texttt{retract}:

\[
\text{findall}(X, \text{Query}, \text{XList}) : \neg \text{Query}, \begin{array}{l}
\text{assert(\text{answer}(X))},
\end{array}
\]

\[
\text{retract(XList)}.
\]

LP 2015 Seite 12
\begin{verbatim}
fail ;
!
CollectAnswers(Xlist).

collectAnswers([X|Rest]) :- retract(answer(X)),
    CollectAnswers(Rest).

collectAnswers([]).
\end{verbatim}

Since Prolog-programs can also be regarded as terms, one can use Prolog to write

\underline{meta-programs} (programs that operate on programs,
  e.g., compilers and interpreters)

and

\underline{meta-interpreters} (interpreter for a prog. language
  that is written in this prog. language).

In particular, one can also easily write interpreters
  for variants of Prolog.

\underline{Simplest meta-interpreter (Meta-Interpreter 0)}

\texttt{prove (Goal) :- Goal.}
If prog contains $p(0)$

? - prove ($p(X)$).

$X = 0$

**Meta-Interpreter 1** (for pure logic programs)

prove (true) :- !.

prove ( (Goal₁, Goal₂) ) :- !, prove(Goal₁), prove(Goal₂).

prove (Goal) :- clause(Goal, Body), prove(Body).

Variant of this meta-interpreter where composed goals are handled from right to left:

**Meta-Interpreter 2**

Variant of meta-interpreter 1 which also returns the length of the proof:

**Meta-Interpreter 3**

? - prove (fatherOf (garth, C), N).

C = Susanne, N = 3