5. The programming language Prolog

Prolog: most popular logic prog. language, developed in the 1970s by Kowalski & Colmerauer.
Essentially, Prolog uses the same syntax as the logic programs in Sect 4.

- function and pred. symbols: strings starting with lower-case letter, strings consisting of special symbols (e.g., `<-->, +, <=, ...), ...

Variables: strings starting with upper-case letter or with _ (e.g., _G192). Special anonymous variable _: its instantiation is not included in answer substitutions and several occurrences of _ can be instantiated differently.

Ex: \text{Prolog} \quad p(a,b,c).

\text{?- } p(_, _, X).

Answer: \quad X = c

Prolog allows overloading of fun. and pred. symbols.
One may have different symbols with the same name, but different arity:

\[ p(a,b,c), \quad 2 \text{ different } \]
\[ p(a,c). \quad \in \ p\text{-symbols} \]
\[ p(a,c). \quad \text{that have no connection} \]

To distinguish such symbols, we often write

\[ p/3 \quad \text{and} \quad p/2. \]

To gain efficiency, Prolog does not implement proper unification, but it does not perform the occur check.

To unify \( X \) with a term \( t \), one has to check whether \( X \) occurs in \( t \). In that case, \( X \) and \( t \) are not unifiable (unless \( X=t \)).

Prolog omits this check.

So far both, \( X \) and \( f(X) \) are unifiable.

The unifier instantiates \( X \) with \( f(f(\ldots))) \). infinitely many.

More precisely, \( X \) is replaced by a pointer to \( f(X) \).

\[ \Rightarrow \text{we obtain} \]
\[ \begin{array}{ccc}
  f & \downarrow & f \\
  \uparrow & \nearrow & \uparrow \\
  X & f & f \\
\end{array} \]
Sed terms are called rational terms (can be represented by finite graphs).

Good prog. style: avoid this problem, do not write programs where the occur check would fail, do not write programs that construct such infinite terms.

Prolog has many pre-defined predicates, including a predicate for proper unification:

```
 unify_with_occurs_check

?- unify_with_occurs_check (X, f(X)).
false

?- unify_with_occurs_check (X, f(Y)).
X = f(Y)
```

5.1. Arithmetic
5.2. Lists
5.3. Operators
5.4. Cut + Negation
5.5. Input + Output
5.6. Meta-Programming

FRI: lectures
now in AH 1
(same room as exercise course)
5.7. Parsing with Prolog

5.1. Arithmetic

All data objects have to be represented as terms suitable for data structures like lists, trees, graphs...

Unsuitable for numbers:

One can represent \( \mathbb{N} \) by the fact symbols

\[ 0 \in \Sigma_0 \quad \text{and} \quad s \in \Sigma_1. \]

Then

\[ 0 \equiv 0 \]
\[ s(0) \equiv 1 \]
\[ s(s(0)) \equiv 2 \]
\[ \vdots \]
\[ s^{1000}(0) \equiv 1000 \]

Drawbacks:

- one can’t use efficient arithmetic operations of the operating system processor
- unsuitable for large numbers

⇒ Prolog has built-in numbers.

Addition algorithm on user-defined numbers:

To implement a function of arity \( n \), one needs a predicate of arity \( n+1 \).

\( \text{add/3} : \ add(t_1, t_2, t_3) \iff t_1 + t_2 = t_3 \)
add(X, 0, X).
add(X, s(Y), s(Z)) :- add(X, Y, Z).

?- add(s(0), s(s(0)), X).  \( \leftarrow \) computes 1+2
X = s(s(s(0))))

?- add(X, s(s(0)), s(s(s(0))) ). \( \leftarrow \) computes 3-2
X = s(0)

The same alg. can be used for addition and subtraction
\( \Rightarrow \) Bidirectionality

?- add(s(0), Y, Z).
\( \leftarrow \) infinitely many answer substitutions

\( \Rightarrow \) Even simple (and reasonable) programs may have an infinite SLD-tree if one uses an unfortunate query
\( \Rightarrow \) termination depends on query

Built-in numbers in Prolog:

arithmetic expression : term built from
- numbers (0, 1, 2, ...)
- Variables (X, Y, ...)

* Binary infix function symbols for arithmetic:
  
  $\ast$, $\frac{..}{..}$, $\uparrow$, $\ast \uparrow$, ...

  Integer division

  Exponentiation

  Unary negation symbol

In principle, arithmetic expressions are ordinary terms. Most Prolog predicate symbols treat them as ordinary terms.

\[
\text{equal}(X,Y).
\]

?- \text{equal}(X,Y).

\[
X = Y.
\]

?- \text{equal}(3, 1+2).

false.  \leftarrow \text{Reason: 3 and } 1+2 \text{ cannot be unified.}

There are some pre-defined Prolog predicates that evaluate arithmetic expressions. This is in contrast to ordinary logic programming where fact symbols are never evaluated.

Pre-defined predicates for comparison of arithmetic expressions:

\[
<, >, \leq, \geq, \equiv, \neq, \wedge, \vee, \Rightarrow, \Leftarrow
\]
For such a predicate \( op \):
\[
?- t_1 \ op \ t_2.
\]

Succeeds iff at the point of evaluation, \( t_1 \) and \( t_2 \) are fully instantiated arithmetic expressions and the result of evaluating \( t_1 \) is in relation \( 'op' \) to the result of evaluating \( t_2 \).

- \(?- 1 < 2.\)
  \text{true}

- \(?- 1 + 1 < 1 + 1.\)
  \text{true}

- \(?- 6 \div 3 < 5-4.\)
  \text{false}

- \( p(1). \quad q(2). \quad \leftarrow \)
  \( ?- p(X), q(Y), X < Y. \quad \text{true} \)

- \(?- X < 1. \quad \)
Program stops with error (X is not fully instantiated.)
* ?- a < 1.
Program stops with error (a is no arithmetic expr.)

Problem: these predicates can't be used to instantiate variables

?- X ::= 1.

Will not result in answer subst X = 1, but in a prog. error.

Thus: another pre-defined predicate "is".

?- t₀ is t₂.

succeeds iff t₂ is a fully instantiated arithmetic expression, t₂ evaluates to some number \( t \), and t₀ unifies with \( t \).

?- X is 2.
X=2
?- X is 1+1.
X=2
?- 2 is 1+1.
true
?- 1+1 is 2.
false
?- X is 3+4, Y is X+1.
X=7, Y=8
?- Y is X+1, X is 3+4.
prog. error

Equality predicates:

\(=\) : arithmetic equality, both left- and right-hand side are evaluated

is : arithmetic equality, only right-hand side is evaluated (without occur check)

\(=\) : syntactic unification (i.e., is defined by the only fact \(X=X\).)

\(=\) : syntactic identity (no unification)

?- a=a.
true
?- 2 = 1+1.
false

? - \( \text{false} \).
\( \forall X = \text{T} \).
\( \forall Y = \text{T} \).
\( \forall \text{add}(X, 0, X) \).
\( \forall \text{add}(X, s(Y), s(Z)) : \text{add}(X, Y, Z) \).
\( \forall \text{add}(X, Y, Z) : \text{Y} > 0, \text{Y} \text{ is } Y - 1, \text{add}(X, Y', Z') \).
\( \exists Z \text{ is } Z' + 1 \).
\( \forall \text{add}(1, 2, X) \).

Addition with built-in numbers:

- \( \text{add}(X, 0, X) \).
- \( \text{add}(X, s(Y), s(Z)) : \text{add}(X, Y, Z) \).
- \( \text{add}(X, Y, Z) : \text{Y} > 0, \text{Y} \text{ is } Y - 1, \text{add}(X, Y', Z') \).
- \( \exists Z \text{ is } Z' + 1 \).

true

? - \( \text{T} \).

? - \( X = X \).

true

? - \( X = Y \).

false
X = 3
?
- add (X, 2, 3).

prop. error, because one reaches an is-literal where the right-hand side is not fully instantiated.

⇒ bidirectional lity is lost, because the built-in arithmetic predicates are not bidirectional.

Why don’t we use the following alg:

add (X, 0, X).
add (X, Y+1, Z+1) :- add (X, Y, Z).

?- add (1, 2, X).
false

?- add (1, 0+1, X).
X = X+1

More arithmetic algorithms:

\( \text{fact}(t_1, t_2) \) holds iff \( t_2 = t_1! \).

\( \text{fact}(0, 1) \).
0

\( \text{fact}(4, 24) \), since \( 4 \cdot 3 \cdot 2 \cdot 1 = 24 \)
\text{fact}(X, Y) : - X > 0, X' is X - 1, \text{fact}(X', Y')
\quad Y is X \times Y'.

? - \text{fact}(4, Y),
\quad Y = 24

\text{gcd (greatest common divisor)}:
\quad \text{gcd}(t_1, t_2, t_3) \text{ iff } t_3 \text{ is the gcd of } t_1 \text{ and } t_2 \text{ (for natural numbers)}

? - \text{gcd}(28, 36, Z).
\quad Z = 4

\text{gcd}(X, 0, X).
\text{gcd}(0, X, X).

\text{gcd}(X, Y, Z) : - X = < Y, X > 0, Y' is Y - X,
\quad \text{gcd}(X, Y', Z).

\text{gcd}(X, Y, Z) : - X > Y, Y > 0, X' is X - Y,
\quad \text{gcd}(X', Y, Z).

There is a pre-defined predicate number/1 to check whether the argument is a number.
?- number (2).
true
?- number (X+1).
false
?- X is X+1, number (X).
X=2

For numbers, the built-in numbers lead to more readable and more efficient algorithms. Arithmetic functions are evaluated using efficient arithmetic operations of the operating system.

For lists, there is also a pre-defined data structure to increase readability.