# 5. The Programming Language Prolog

developed by Kowalski + Colmerauer in the 1970s

Fixes certain syntax rules:

- :- , ?- , ...
- function + predicate symbols: strings starting with lower-case symbol, strings of special symbols (<->), strings in quotes 'X'
- variables: strings starting with upper-case symbols or with _ ( _G192 )
- special anonymous variable _ ("don't care")
  - each occurrence of _ can be instantiated differently
  - instantiations of _ are not contained in the answer subst.

Ex: Prog contains the fact p(a,b,c).
?- p(_,_,X).
X = c

- Prolog allows overloading of pred./fct. symbols.

p(a,b,c). p/3 2 unrelated
p(d,e). p/2 pred. symbols

one often writes pred./fct. symbols

in this way to distinguish pred./fct. symbols of different arities.

- Prolog uses a variant of unification without occur check to improve efficiency.

equal(X,X).

?- equal(f(Y), g(Y)).
false

clash failure

?- equal(f(0), f(Y)).
Y=0

?- equal(Y, f(Y)).
Y = f(Y)

succeeds in Prolog although one should have occur failure

Alternatives:
Y = f(~~**~~)
Y = f(f(...))

Solution: Y/f(f(...))

infinite term

In general, one should avoid Prolog programs where possible occur failures could happen and where infinite terms are constructed.

Prolog has a pre-defined predicate for proper unification:

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

Now we will introduce several features of Prolog

that go beyond pure logic programming.
Sect 5.1 + 5.2: typical pre-defined data types

## 5.1. Arithmetic

Prolog has no data types, but it only operates on terms.
$\Rightarrow$ Data objects have to be represented by terms.

Ex: $\mathbb{N}$ can be represented by terms over $0 \in \Sigma_0$ and $s \in \Sigma_1$:

```
add(X, 0, X).
add(X, s(Y), s(Z)) :- add(X, Y, Z).
```

Can be used for addition, subtraction, ...:
?- add(X, s(0), s(s(0)))
Bidirectionality

Disadvantage: inefficient and hard to read

$1000 \ \hat{=} \ \underbrace{s(s(\ \dots\ s}_{1000 \text{ times}}(0))\dots)$

Prolog has built-in arithmetic that allows us to write numbers as usual and to use efficient arithmetic operations provided by the operating system.

Arithmetic expression: term built from numbers, variables,
binary infix symbols +, -, *, //, **, ...,
// : integer division
** : exponentiation
unary negation -

In principle, these are terms as usual.

```
equal(X,X).
?- equal(3, 1+2).
false
```

+, -, ... are syntactic

false

?- equal(Y, 1+2).

Y = 1+2.

are syntactic fct. symbols that are not evaluated in syntactic unification

There exist special pre-defined predicates which <u>evaluate</u> arithmetic expressions: <, >, =<, >=,

=:= , =\=

equality non-equality

For an operation op like this:

?- $t_1$ op $t_2$.

When evaluating this query, $t_1$ and $t_2$ must be fully instantiated arithmetic expressions. Then the pre-defined fcts +, -, ... are evaluated and the boolean result of the comparison determines whether the query succeeds.

?- 1 < 2.

true

?- 1 * 1 < 1 + 1.

true

?- -2 > 1.

false

?- X > 1.

error

?- monika > 1

error

These predicates cannot be used to instantiate variables.

?- X =:= 2.

error

Therefore, there is another predicate is/2.

?- $t_1$ is $t_2$.

When evaluating the query, $t_2$ must be a fully instantiated arith. expr. Afterwards, $t_1$ is unified with the result of evaluating $t_2$.

?- 2 is 1+1.

true

?- 1+1 is 2.

false

?- X is 1+1.

X = 2

?- X+1 is 1+1.

false

?- X is 3+4, Y is X+1.

X = 7, Y = 8

?- Y is X+1, X is 3+4.

error

Prolog has several predicates for equality:

- =:= arithmetic equality where both arguments are evaluated

- is equality, where the right argument is evaluated and afterwards one performs unification
- = unification (corresponds to equal(X,X). ) without occur check

?- monika = monika
true

?- f(X)=f(a).
X=a

?- X = 1+1.
X=1+1

?- 1+1 = 2.
false

?- f(X) = X.
X = f(X)

- unify_with_occurs_check
- == syntactic equality

?- monika == monika.
true

?- f(X) == f(Y).

false

Computing with arithmetic:

add(X,0,X).

without built-in

```
add(X,0,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).
```

without built-in arithmetic

Instead:

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

Disadvantage:

```
?- add'(1,2,X).
false
?- add'(1,0+1,X).
X = 1+1
```

Better:

```
add''(X,0,X).
add''(X,Y,Z) :- Y>0, Y1 is Y-1,
                add''(X,Y1,Z1), Z is Z1+1.
```

```
?- add''(1,2,X).
X = 3
```

```
?- add''(X,2,3).
error
```

because at some point one has to evaluate an is-literal where the right argument is not fully instantiated

⇒ We lose bidirectionality.

Easier and much more efficient: add(X, Y, Z) :- Z is X+Y.

Ex: gcd on natural numbers

```
gcd(X, 0, X).
gcd(0, X, X).
gcd(X,Y,Z) :- X =< Y, X > 0, Y1 is Y-X,
              gcd(X, Y1, Z).
gcd(X,Y,Z) :- X > Y, Y > 0, X1 is X-Y,
              gcd(X1, Y, Z).
```

```
?- gcd(28, 36, X).
X = 4
```

Again, this is not bidirectional. To implement bidirectional arithmetic programs: Constraint Logic Programming (CLP)

Pre-defined pred. number/1 is true if

the argument is a number when the pred. is evaluated:

```
?- number(2).              ?- number(1+1).
true                       false

?- X is 1+1, number(X).
X = 2

?- number(X).
false
```