U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEPARTMENT OF INFORMATICS ENGINEERING

U.PORTO
FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

DEPARTMENT OF COMPUTER SCIENCE

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2021/2022 – 1st Semester

Prolog

Meta-Programming and Operators

Daniel Castro Silva
dcs@fe.up.pt

Agenda

- Meta-Programming
- Operators

Meta-Programming

- Prolog has some meta-logical predicates for type checking
  - ***integer(A)*** – A is an integer
  - ***float(A)*** – A is a floating point number
  - ***number(A)*** – A is a number (integer or float)
  - ***atom(A)*** – A is an atom
  - ***atomic(A)*** – A is an atom or a number
  - ***compound(A)*** – A is a compound term
  - ***nonvar(A)*** – A is an atom, a number or a compound term
  - ***var(A)*** – A is a variable (it is not instantiated)
  - ***ground(A)*** – A is *nonvar*, and all substructures are *nonvar*

Meta-logical Predicates

- These predicates can be very useful to implement different versions of predicates depending on variable instantiation

```
grandparent(X, Y):- nonvar(Y), !, parent(Z, Y), parent(X, Z).
grandparent(X, Y):- parent(X, Z), parent(Z, Y).
```

  - We can think of an implementation of the *sum/3* predicate that tests for instantiation, using a more appropriate definition in each case

```
sum(A, B, S):- number(A), number(B), !, S is A + B.
sum(A, B, S):- number(A), number(S), !, B is S - A.
sum(A, B, S):- number(B), number(S), !, A is S - B.
```

See section 4.8.1 of the SICStus Manual for more information

Meta-Programming

- Other predicates allow access to terms and their arguments
  - *functor(+Term, ?Name, ?Arity)* or *functor(?Term, +Name, +Arity)*
    - If *Term* is instantiated, returns the name and arity of the term
    - If *Term* is not instantiated, creates a new term with given name and arity
  - *arg(+Index, +Term, ?Arg)*
    - Given an index and a term, instantiates *Arg* with the argument in the Nth position (index starts in 1)
  - *+Term =.. ?[Name | Args]* or *?Term =.. +[Name | Args]*
    - Given a term, returns a list with the name and arguments of the term
    - Given a proper list, creates a new term with name and arguments as specified by the contents of the list

Meta-Programming

- The functionality of *univ* (=..) can be attained using *functor* and *arg* (and vice-versa)

```
| ?- functor(parent(homer, bart), Name, Arity).
Name = parent,
Arity = 2 ?
yes
| ?- functor(Term, parent, 2).
Term = parent(_A,_B) ?
yes
| ?- arg(2, parent(homer, bart), Arg).
Arg = bart ?
yes
| ?- parent(homer, bart) =.. List.
List = [parent,homer,bart] ?
yes
| ?- Term =.. [parent, homer, bart].
Term = parent(homer,bart) ?
yes
```

Meta-Programming

- The *call/1* predicate calls (executes) a given goal
  - However, it's use is not really necessary

```
| ?- C = write('Hello World!'), call(C).
Hello World!
C = write('Hello World!') ?
yes
| ?- C = write('Hello World!'), C.
Hello World!
C = write('Hello World!') ?
yes
| ?- G =.. [write, 'Hi there!'], G.
Hi there!
G = write('Hi there!') ?
yes
```

  - In the example, *C* is a meta-variable - it represents a callable goal
  - ***callable/1*** verifies if a term is callable

Meta-Programming

- Can be used to implement *higher-order predicates*

```
map([], _).
map([H|T], P):-
        G =.. [P, H],
        G,
        map(T, P).
```

```
| ?- map([1,a,2,b], write).
1a2b
yes
| ?- map([1,a,2,b], number).
no
| ?- map([1,a,2,b], atomic).
yes
```

Operators

- Prolog allows for the definition of new operators
  - We can easily change the way we write programs

```
homer likes marge.
marge likes homer.
homer and marge parented bart.
homer and marge parented lisa.
```

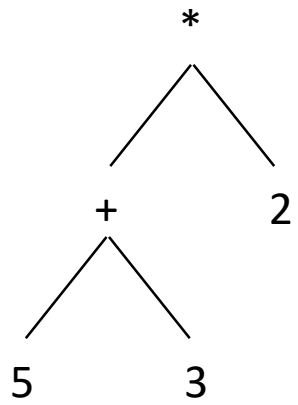
- Operators are characterized by precedence and associativity

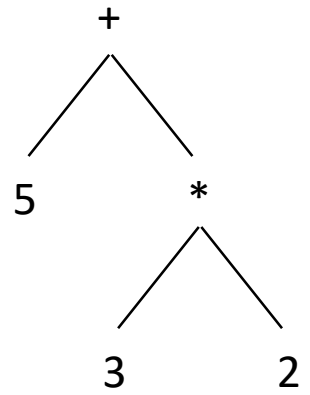
Operators

- Precedence determines which operation is executed first

```
X is 5 + 3 * 2.
```

```
| ?- X is (5 + 3) * 2.
X = 16 ?
yes
| ?- X is 5 + (3 * 2).
X = 11 ?
yes
```





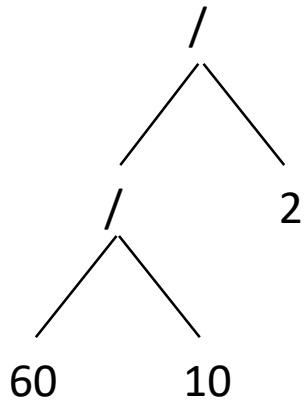
- Precedence in Prolog is given by a number between 1 and 1200
  - Multiplication has precedence level 400
  - Addition has precedence level 500

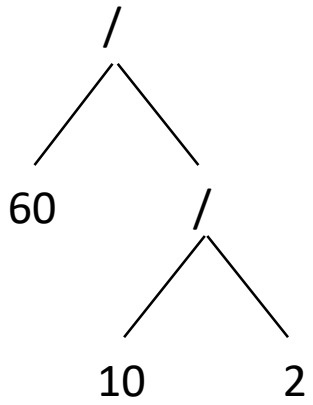
Operators

- Associativity determines how to associate operations

```
X is 60 / 10 / 2.
```

```
| ?- X is (60 / 10) / 2.
X = 3.0 ?
yes
| ?- X is 60 / (10 / 2).
X = 12.0 ?
yes
```





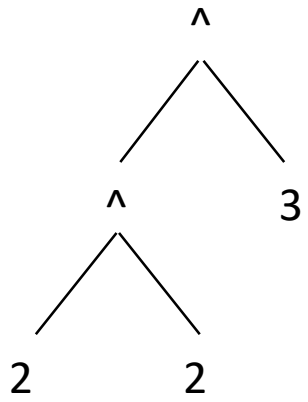
- Division is left-associative

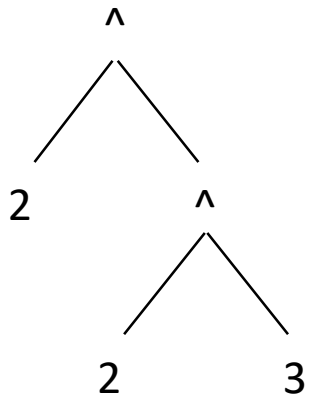
Operators

- Associativity determines how to associate operations

```
X is 2 ^ 2 ^ 3.
```

```
| ?- X is (2^2)^3.
X = 64 ?
yes
| ?- X is 2^(2^3).
X = 256 ?
yes
| ?- X is 2^2^3.
X = 256 ?
yes
```





- The ^ operator is right-associative

Operators

- The *op/3* predicate can be used to specify new operators

```
op(+Precedence, +Type, +Name).
```

  - Precedence is a number between 1 and 1200
  - Type defines the type and associativity of the operator
    - Prefix – fx or fy
    - Postfix – xf or yf
    - Infix – xfx, xfy or yfx
    - f defines the position of the operator
    - x and y represent the operands
    - x means non-associative
    - y means side-associative

Operators

- Built-in operators

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1150,  fx, [ mode, public, dynamic, volatile, discontiguous,
                    multifile, block, meta_predicate,
                    initialization ]).
:- op( 1100, xfy, [ ;, do ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).
:- op(  900,  fy, [ \+, spy, nospy ]).
:- op(  700, xfx, [ =, \=, is, =.., ==, \==, @<, @>, @=<, @>=,
                                    =:=, =\=, <, >, =<, >= ]).
:- op(  550, xfy, [ : ]).
:- op(  500, yfx, [ +, -, \, /\, \/ ]).
:- op(  400, yfx, [ *, /, //, div, mod, rem, <<, >> ]).
:- op(  200, xfx, [ ** ]).
:- op(  200, xfy, [ ^ ]).
:- op(  200,  fy, [ +, -, \ ]).
```

Operators

- Defining operators allows for a new syntax

```
:-op(380, xfy, and).
:-op(400, xfx, likes).
:-op(400, xfx, practices).

tom likes wine and cheese.
richard likes cheese.
harry practices tennis and golf.
```

```
| ?- harry practices X and Y.
X = tennis,
Y = golf ?
yes
| ?- richard likes X.
X = cheese ?
yes
| ?- tom likes X.
X = wine and cheese ?
yes
| ?- X likes wine and cheese.
X = tom ?
yes
```

Q & A