UCS1524 – Logic Programming

Operators and Arithmetic in Prolog

Session Meta Data

Author	**Dr. D. Thenmozhi**
Reviewer	
Version Number	1.2
Release Date	10 September 2022

Session Objectives

- Understanding operators and arithmetic in Prolog

Session Outcomes

- At the end of this session, participants will be able to
  - Develop program in Prolog using operators and arithmetic.

Agenda

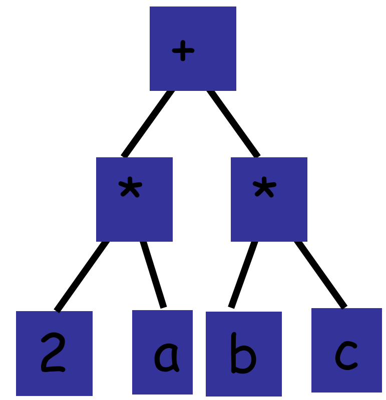
- Operator notation
- Arithmetic

Operator notation

- Consider an expression **2*a+b*c**
- In this, + and * are said to be infix operators because they appear between the two arguments.

- Such expressions can be represented as trees, and can be written as Prolog terms with + and * as functors:

  +(*(2,a),*(b,c))



- The general rule is that the operator with the highest precedence is the principal functor of the term.
  - If '+' has a higher precedence than '*', then the expression **a+ b*c** means the same as

    a + (b*c). (+(a, *(b,c)))
  - If '*' has a higher precedence than '+', then the expression **a+ b*c** means the same as

    (a + b)*c. (*(+(a,b),c))

Operator notation

- A programmer can define his or her own operators.
- For example:
  - We can define the atoms **has** and **supports** as infix operators and then write in the program facts like:

    peter has information.

    floor supports table.
  - The facts are exactly equivalent to:

    has(peter, information).

    supports(floor, table).

Operator notation

- Define new operators by inserting into the program special kinds of clauses, called directives:

  :- op(600, xfx, has).

  - The precedence of ‘has’ is 600.
  - Its type ‘xfx’ is a kind of infix operator. The operator denoted by ‘f’ is between the two arguments denoted by ‘x’.
- The operator definitions do not specify any operation or action.
- Operators are normally used, as functors, only to combine objects into structure.
- Operator names are atoms.
- We assume that the range of operator’s precedence is between 1 and 1200.

Operator notation

- There are three groups of operator types:

  (1) **Infix** operators of three types:

  xfx xfy yfx

  (2) **Prefix** operators of two types:

  fx fy

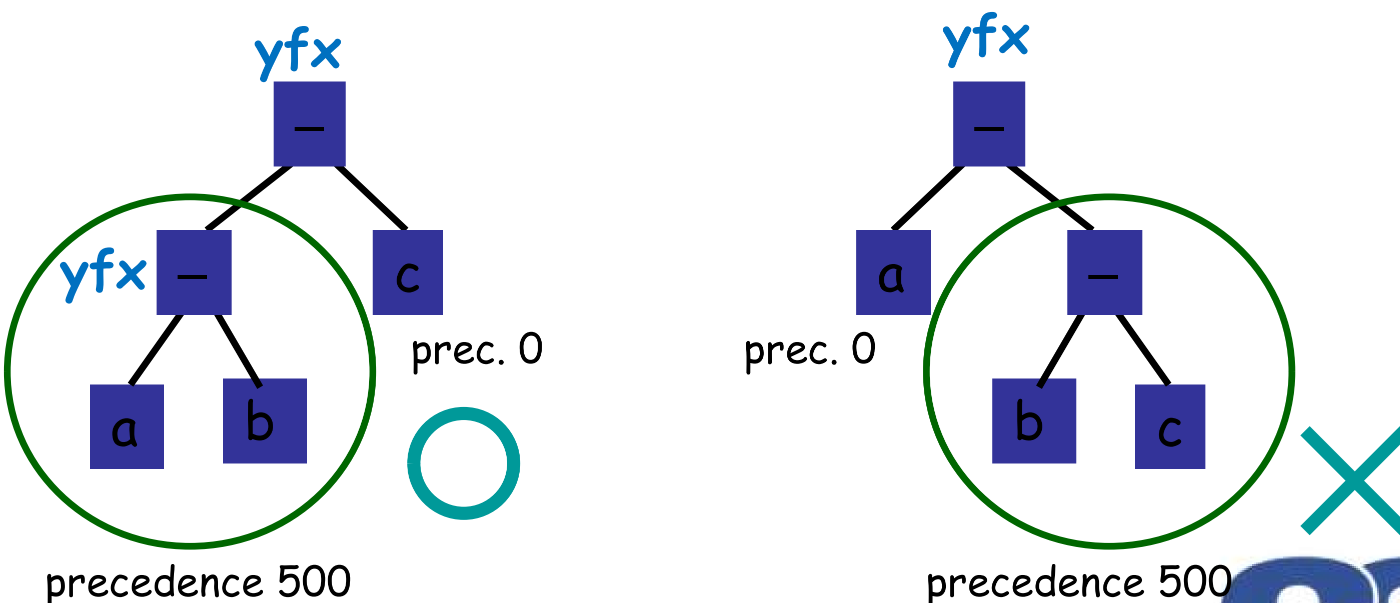
  (3) **postfix** operators of two types:

  xf yf

- Precedence of argument:
  - If an argument is enclosed in parentheses or it is an unstructured object then its precedence is 0.
  - If an argument is a structure then its precedence is equal to the precedence of its principal functor.
  - ‘x’ represents an argument whose precedence must be strictly lower than that of the operator.
  - ‘y’ represents an argument whose precedence is lower or equal to that of the operator.

Operator notation

- Precedence of argument:
  - This rules help to disambiguate expressions with several operators of the same precedence.
  - For example:
    - Assume that '–' has precedence 500. If '–' is of type **yfx**, then the right interpretation is invalid because the precedence of **b-c** is not less than the precedence of '–'.
    - So, **a – b – c** is **(a – b) – c** not **a –(b – c)**



Operator notation

- Another example: operator **not**
  - If **not** is defined as **fy** then the expression

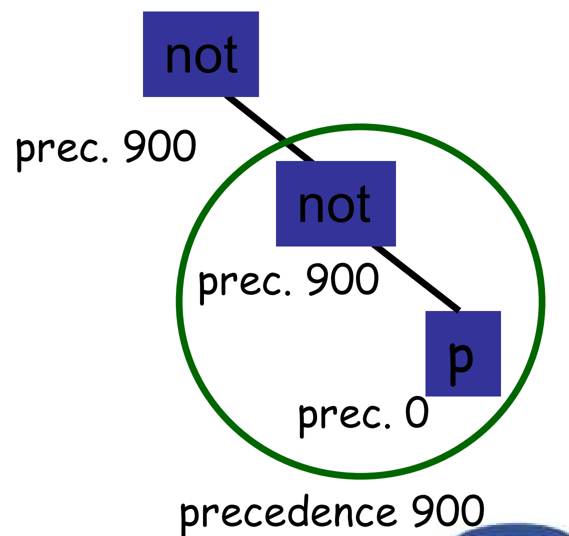
    not not p

    is legal.
  - If **not** is defined as **fx** then the expression

    not not p

    is illegal, because the argument to the first **not** is **not p**. ➔ here **not (not p)** is legal.

```
:- op( 900, fy, not).
| ?- X = not(not(P)).
X = (not not P)
Yes
:- op( 900, fx, not).
| ?- X = not(not(P)).
X = (not (not P))
Yes
```



Operator notation

- A set of predefined operators in the Prolog standard.

```
:- op( 1200, xfx, [:-, -->]).
:- op( 1200, fx, [:-, ?-]).
:- op( 1050, xfy, ->).
:- op( 900, fy, not).
:- op( 700, xfx, [=, \=, ==, \==, =..]).
:- op( 700, xfx, [is, =:=, =\=, <, =<, >, >=, @<, @=<,
                  @>, @>=]).
:- op( 500, yfx, [+, -]).
:- op( 400, yfx, [*, /, //, mod]).
:- op( 200, xfx, **).
:- op( 200, xfy, ^).
:- op( 200, fy, -).
```

Operator notation

- Boolean expressions

  de Morgan's theorem:

  ~(A & B) <===> ~A v ~B

  – One way to state this in Prolog is

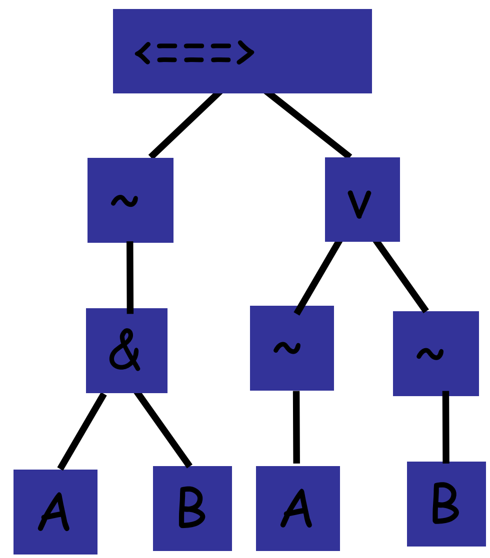
  equivalence(not(and(A, B)), or(not(A), not(B))).

  – If we define a suitable set of operators:

  ```
  :- op( 800, xfx, <===>).
  :- op( 700, xfy, v).
  :- op( 600, xfy, &).
  :- op( 500, fy, ~).
  ```

  – Then the de Morgan's theorem can be written as the fact.

  ~(A & B) <===> ~A v ~B



(please define all the Boolean relations by yourselves)

Boolean expressions

~(A & B) <===> ~A v ~B

```
| ?- #\ (A #/\ B) #<=> #\ A #\/ #\B.
A = _#18(0..1)
B = _#36(0..1)
yes
```

FD Expression	Result
Prolog variable	domain 0..1
FD variable X	domain of X, X is constrained to be in 0..1
0 (integer)	0 (false)
1 (integer)	1 (true)
#\ E	not E
E1 #<=> E2	E1 equivalent to E2
E1 #\<=> E2	E1 not equivalent to E2 (i.e. E1 different from E2)
E1 ## E2	E1 exclusive OR E2 (i.e. E1 not equivalent to E2)
E1 #==> E2	E1 implies E2
E1 #\==> E2	E1 does not imply E2
E1 #/\ E2	E1 AND E2
E1 #\/\ E2	E1 NAND E2
E1 #\/ E2	E1 OR E2
E1 #\\/ E2	E1 NOR E2

Operator notation

- Summarize:
  - Operators can be infix, prefix, or postfix.
  - Operator definitions do not define any action, they only introduce new notation.
  - A programmer can define his or her own operators. Each operator is defined by its name, precedence, and type.
  - The precedence is an integer within some range, usually between 1 and 1200.
  - The operator with the highest precedence in the expression is the principal functor of the expression.
  - Operators with lowest precedence bind strongest.
  - The type of an operator depends on two things:
    - The position of the operator with respect to the arguments
    - The precedence of the arguments compared to the precedence of the operator itself.
    - For example: **xfy**

Arithmetic

- Predefined basic arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
**	power
//	integer division
mod	modulo, the remainder of integer division

```
| ?- X = 1+2.
X = 1+2
yes
| ?- X is 1+2.
X = 3
yes
```

Operator '**is**' is a built-in procedure.

Arithmetic

- Another example:

```
| ?- X is 5/2,
     Y is 5//2,
     Z is 5 mod 2.
 X = 2.5
 Y = 2
 Z = 1
```

- Since X is 5-2-1→ X is (5-2)-1, parentheses can be used to indicate different associations. For example, X is 5-(2-1).
- Prolog implementations usually also provide standard functions such as sin(X), cos(X), atan(X), log(X), exp(X), etc.

```
| ?- X is sin(3).
X = 0.14112000805986721
```

- Example:

```
| ?- 277*37 > 10000.
 yes
```

Arithmetic

- Predefined comparison operators:

  X > Y X is greater than Y

  X < Y X is less than Y

  X >= Y X is greater than or equal to Y

  X =< Y X is less than or equal to Y

  X =:= Y the values of X and Y are equal

  X =\= Y the values of X and Y are not equal

```
| ?- 1+2 =:= 2+1.
yes
| ?- 1+2 = 2+1.
no
| ?- 1+A = B+2.
A = 2
B = 1
yes
```

Arithmetic

- GCD (greatest common divisor) problem:
  - Given two positive integers, X and Y, their greatest common divisor, D, can be found according to three cases:
    - (1) If X and Y are equal then D is equal to X.
    - (2) If X < Y then D is equal to the greatest common divisor of X and the difference Y-X.
    - (3) If Y < X then do the same as in case (2) with X and Y interchanged.
  - The three rules are then expressed as three clauses:

```
gcd( X, X, X).
gcd( X, Y, D) :- X<Y, Y1 is Y-X, gcd( X, Y1, D).
gcd( X, Y, D) :- Y<X, gcd( Y, X, D).
?- gcd( 20, 25, D).
D=5
```

GCD

```
(1) gcd( X, X, X).
(2) gcd( X, Y, D) :- X<Y, Y1 is Y-X, gcd( X, Y1, D).
(3) gcd( X, Y, D) :- Y<X, gcd( Y, X, D).
```

```
| ?- gcd( 10, 25, D).
     1    1  Call: gcd(10,25,_23) ?
     2    2  Call: 10<25 ?
     2    2  Exit: 10<25 ?
     3    2  Call: _121 is 25-10 ?
     3    2  Exit: 15 is 25-10 ?
     4    2  Call: gcd(10,15,_23) ?
     5    3  Call: 10<15 ?
     5    3  Exit: 10<15 ?
     6    3  Call: _199 is 15-10 ?
     6    3  Exit: 5 is 15-10 ?
     7    3  Call: gcd(10,5,_23) ?
     8    4  Call: 10<5 ?
     8    4  Fail: 10<5 ?
     8    4  Call: 5<10 ?
     8    4  Exit: 5<10 ?
```

```
 9    4  Call: gcd(5,10,_23) ?
10    5  Call: 5<10 ?
10    5  Exit: 5<10 ?
11    5  Call: _327 is 10-5 ?
11    5  Exit: 5 is 10-5 ?
12    5  Call: gcd(5,5,_23) ?
12    5  Exit: gcd(5,5,5) ?
 9    4  Exit: gcd(5,10,5) ?
 7    3  Exit: gcd(10,5,5) ?
 4    2  Exit: gcd(10,15,5) ?
 1    1  Exit: gcd(10,25,5) ?

D = 5 ?

(15 ms) yes
```

Arithmetic

- Length counting problem:
  (Note: **length** is a build-in procedure)
  - Define procedure **length(List, N)** which will count the elements in a list **List** and instantiate **N** to their number.
    (1) If the list is empty then its length is 0.
    (2) If the list is not empty then **List = [Head|Tail];** then its length is equal to 1 plus the length of the tail **Tail**.
  - These two cases correspond to the following program:

```
length1( [], 0).
length1( [ _| Tail], N) :- length1( Tail, N1),
                           N is 1 + N1.
```

```
?- length1( [a, b, [c, d], e], N).
N = 4
```

Arithmetic

- Another programs:

```
length2( [], 0).
length2( [_ | Tail], N) :- length2( Tail, N1),
                           N = 1 + N1.

| ?- length2( [a, b, [c, d], e], N).
N = 1+(1+(1+(1+0)))

length3( [], 0).
length3( [_ | Tail], N) :- N = 1 + N1,
                           length3( Tail, N1).

length3( [_ | Tail], 1 + N) :- length3( Tail, N).

| ?- length3([a,b,c],N), Length is N.
Length = 3
N = 1+(1+(1+0))
```

Arithmetic

- Summarize:
  - Build-in procedures can be used for doing arithmetic.
  - Arithmetic operations have to be explicitly requested by the built-in procedure **is**.
  - There are build-in procedures associated with the predefined operators +, -, *, /, **div** and **mod**.
  - At the time that evaluation is carried out, all arguments must be already instantiated to numbers.
  - The values of arithmetic expressions can be compared by operators such as <, =<, etc. These operators force the evaluation of their arguments.

Summary

- Operator
  - Operator notation
  - Operator types (infix, prefix and postfix)
  - Precedence
  - Predefined operators
  - User defined operators
- Arithmetic
  - Build in procedures
  - Evaluation operator “is”
  - Operations with arithmetic and comparison operators

Check your understanding

- If the ‘**+**’ and ‘**–**’ operators are defined as

  :- op(500, xfy, [+,–]).

  Please draw their corresponding binary trees.

  (A) a+b–c–d

  (B) a+b–(c–d)

  (C) (a+b)–c–d

  (D) a+(b–c)–d

Check your understanding

- Assuming the operator definitions

:-op(200, xfx, plays).   Plays(jummy, and(football, squash))

:-op(300, xfy, and).   Plays(susan, and(tennis, and(basketball, volleyball))

then the following two terms are syntactically legal objects:

- Terml - jimmy plays football and squash
- Term2 : susan plays tennis and basketball and volleyball
- How are these terms understood by Prolog? What are their principal functors and what is their structure?

Check your understanding

- Consider the program:

  t(0+1, 1+0).

  t(X+0+1, X+1+0).

  t(X+1+1, Z) :- t(X+1, X1), t(X1+1, Z).

  How will this program answer the following questions if '+' is an infix operator of type **yfx** (as usual):

  (a) ?- t(0+1, A).

  (b) ?- t(0+1+1, B).

  (c) ?- t(1+0+1+1+1, C).

  (d) ?- t(D, 1+1+1+0).

  (e) ?- t(1+1+1, E).

Check your understanding

- Find maximum of two elements

```
max( X, Y, X) :- X >= Y.
max( X, Y, Y) :- X < Y.
```

- Find maximum of a list.

```
maxlist( [X], X).          % single element list
maxlist( [X, Y | Rest], Max):- maxlist( [Y | Rest], MaxRest),
    max( X, MaxRest, Max).
```

Check your understanding

- Define the predicate
  sumlist(List, Sum)
  so that **Sum** is the sum of a given list of numbers **List**.

- Define the predicate
  ordered(List)
  which is true if **List** is an ordered list of numbers.
  For example: **ordered([1,5,6,6,9,12])**