Operators

Operators

- The notation used for predicates is the standard one of a functor followed by a number of arguments in parentheses, e.g. **likes(john,mary)**.
- As an alternative, any user-defined predicate with two arguments (a *binary predicate*) can be converted to an *infix operator*
- Infix operator enables the functor (predicate name) to be written between the two arguments with no parentheses, e.g. **john likes mary**

Operators…

- Any user-defined predicate with one argument (a *unary predicate*) can be converted to a *prefix operator*
- This enables the functor to be written before the argument with no parentheses, e.g. **isa_dog fred** instead of **isa_dog(fred)**
- Alternatively, a unary predicate can be converted to a *postfix operator*. This enables the functor to be written after the argument, e.g. **fred isa_dog**

Operators…

- Operator notation can also be used with rules to aid readability
- A rule such as:

  likes(john,X):-is_female(X),owns(X,Y),isa_cat(Y).

- may be written as :

  john likes X:- X is_female, X owns Y, Y isa_cat.

Operators...

- The standard bracketed 'functor and arguments' notation, e.g. **likes(john,X)** can still be used with operators if preferred

- 'Mixed' notation is also permitted, e.g. if **likes/2**, **is_female/1**, **owns/2** and **isa_cat/1** are all operators then

  likes(john,X):-is_female(X),X ownsY,isa_cat(Y).

  is a valid form of the previous rule

Operators...

- Any user-defined predicate with one or two arguments can be converted to an operator by entering a goal using the **op** predicate at the system prompt

- This predicate takes three arguments, for example

  ?-op(150,xfy,likes).

Operators...

- The first argument is the 'operator precedence', which is an integer from 0 upwards. The range of numbers used depends on the particular implementation.
- The lower the number, the higher the precedence
- Operator precedence values are used to determine the order in which operators will be applied when more than one is used in a term
- The most important practical use of this is for operators used for arithmetic. In most other cases it will suffice to use an arbitrary value such as 150.

Operators...

- The second argument should normally be one of the following three atoms:
  - **xfy** meaning that the predicate is binary and is to be converted to an infix operator
  - **fy** meaning that the predicate is unary and is to be converted to a prefix operator
  - **xf** meaning that the predicate is unary and is to be converted to a postfix operator

Operators...

- The third argument specifies the name of the predicate that is to be converted to an operator
- A predicate can also be converted to an operator by placing a line such as

  ?-op(150,xfy,likes).

  in a Prolog program file to be loaded using **consult** or **reconsult**
- When a goal is used in this way, the entire line is known as a *directive*
- In this case, the directive must be placed in the file before the first clause that uses the operator *likes*.

9 Operators 01/03/2020

Operators...

- Several built-in predicates have been pre-defined as operators. These include *relational operators* for comparing numerical values, including < denoting 'less than' and > denoting 'greater than'.
- Thus the following are valid terms, which may be included in the body of a rule:
  - **X>4**
  - **Y<Z**
  - **A=B**
- Bracketed notation may also be used with built-in predicates that are defined as operators, e.g. **>(X,4)** instead of **X>4**.

10 Operators 01/03/2020

Built-in Operators

Name of Operator **,** [comma]

Type of Operator infix

Syntax **Goal1,Goal2**

Description

Succeeds if and only if Goal1 and Goal2 are both true.

11 Operators 01/03/2020

Built-in Operators...

Name of Operator **;** [semicolon]

Type of Operator infix

Syntax **Goal1;Goal2**

Description

Succeeds if either Goal1 or Goal2 is true (or both).

12 Operators 01/03/2020

Built-in Operators…

Name of Operator =

Type of Operator infix

Syntax **Term1 = Term2**

Description

Succeeds if terms Term1 and Term2 unify

Built-in Operators…

Name of Operator \=

Type of Operator infix

Syntax **Term\=Term2**

Description

Succeeds if Term1 does not unify with Term2

Built-in Operators…

Name of Operator ==

Type of Operator infix

Syntax **Term1 == Term2**

Description

Succeeds if Term1 is identical to Term2

Built-in Operators…

Name of Operator \==

Type of Operator infix

Syntax **Term1 \== Term2**

Description

Succeeds if Term1 is not identical to Term2

Built-in Operators...

Name of Operator **=:=**

Type of Operator infix

Syntax **Exp1 =:= Exp2**

Description

Succeeds if the arithmetic expressions Exp1 and Exp2 evaluate to the same numerical value

Built-in Operators...

Name of Operator **=\=**

Type of Operator infix

Syntax **Exp1 =\= Exp2**

Description

Succeeds if the arithmetic expressions Exp1 and Exp2 do not evaluate to the same numerical value

Built-in Operators...

Name of Operator **=..** [pronounced 'univ']

Type of Operator infix

Syntax **Term=..List**

Description

Converts from a list to a term or vice versa

Built-in Operators...

Name of Operator **<**

Type of Operator infix

Syntax **Exp1<Exp2**

Description

Succeeds if the value of arithmetic expression Exp1 is less than the value of arithmetic expression Exp2.

Built-in Operators…

Name of Operator **=<**

Type of Operator infix

Syntax **Exp1=<Exp2**

Description

Succeeds if the value of arithmetic expression Exp1 is less than or equal to the value of arithmetic expression Exp2.

Built-in Operators…

Name of Operator **>**

Type of Operator infix

Syntax **Exp1>Exp2**

Description

Succeeds if the value of arithmetic expression Exp1 is greater than the value of arithmetic expression Exp2.

Built-in Operators…

Name of Operator **>=**

Type of Operator infix

Syntax **Exp1>=Exp2**

Description

Succeeds if the value of arithmetic expression Exp1 is greater than or equal to the value of arithmetic expression Exp2.

Built-in Operators…

Name of Operator **>=**

Type of Operator infix

Syntax **Exp1>=Exp2**

Description

Succeeds if the value of arithmetic expression Exp1 is greater than or equal to the value of arithmetic expression Exp2.

Built-in Operators…

Name of Operator **is/2**

Type of Operator infix

Syntax **Result is Expression**

Description

Expression must be a valid arithmetic expression which is evaluated to give a number. If Result is an unbound variable (the usual case) the variable is bound to the value of the expression. If Result is a bound variable with a numerical value or a number, the goal succeeds if the values of both sides of the **is** operator are the same and fails otherwise.

Built-in Operators…

Name of Operator **not/1**

Type of Operator prefix

Syntax **not Goal**

Description

Succeeds if Goal fails, fails if Goal succeeds.

Arithmetic Operators

- Prolog also provides facilities for doing arithmetic using a notation similar to that which will already be familiar to many users from basic algebra
- This is achieved using the built-in predicate **is/2**, which is predefined as an infix operator and thus is written between its two arguments
- The most common way of using **is/2** is where the first argument is an unbound variable. Evaluating the goal **X is –6.5** will cause X to be bound to the number –6.5 and the goal to succeed.

Arithmetic Operators…

- The second argument can be either a number or an arithmetic expression e.g. **X is 6*Y+Z-3.2+P-Q/4** (* denotes multiplication).
- Any variables appearing in an arithmetic expression must already be bound (as a result of evaluating a previous goal) and their values must be numerical they are, the goal will always succeed and the variable that forms the first argument will be bound to the value of the arithmetic expression. If not, an error message will result.

Arithmetic Operators…

?- X is 10.5+4.7*2.

X = 19.9

?-Y is 10,Z is Y+1.

Y = 10 ,

Z = 11

Arithmetic Operators…

- Symbols such as + - * / in arithmetic expressions are a special type of infix operator known as *arithmetic operators*
- Unlike operators used elsewhere in Prolog they are not predicates but *functions*, which return a numerical value.
- As well as numbers, variables and operators, arithmetic expressions can include *arithmetic functions*, written with their arguments in parentheses (i.e. not as operators)

Arithmetic Operators…

- Like arithmetic operators these return numerical values, e.g. to find the square root of 36:

  ?- X is sqrt(36).

  X = 6

Arithmetic Operators…

- The arithmetic operator **-** can be used not only as a binary infix operator to denote the difference of two numerical values, e.g. **X-6**, but also as a unary prefix operator to denote the negative of a numerical value, e.g.

  ?- X is 10,Y is -X-2.

  X = 10 ,

  Y = -12

Arithmetic Operators...

- Arithmetic operators and arithmetic functions available in Prolog:

X+Y	the sum of X and Y
X-Y	the difference of X and Y
X*Y	the product of X and Y
X/Y	the quotient of X and Y
X//Y	the 'integer quotient' of X and Y (the result is truncated to the nearest integer between it and zero)
X^Y	X to the power of Y
-X	the negative of X
abs(X)	the absolute value of X
sin(X)	the sine of X (for X measured in degrees)
cos(X)	the cosine of X (for X measured in degrees)
max(X,Y)	the larger of X and Y
sqrt(X)	the square root of X

33 Operators 01/03/2020

Arithmetic Operators...

- Example

?- X is 30,Y is 5,Z is X+Y+X*Y+sin(X).

X = 30 ,

Y = 5 ,

Z = 185.5

- Although the **is** predicate is normally used in the way described here, the first argument can also be a number or a bound variable with a numerical value. In this case, the numerical values of the two arguments are calculated. The goal succeeds if these are equal. If not, it fails.

34 Operators 01/03/2020

Arithmetic Operators...

?- X is 7,X is 6+1.

X = 7

?- 10 is 7+13-11+9.

no

?- 18 is 7+13-11+9.

yes

35 Operators 01/03/2020

Arithmetic Operators...

Unification

The previous description can be simplified by saying that the second argument of the **is/2** operator is evaluated and this value is then *unified* with the first argument. This illustrates the flexibility of the concept of unification.

(a) If the first argument is an unbound variable, it is bound to the value of the second argument (as a side effect) and the **is** goal succeeds.

(b) If the first argument is a number, or a bound variable with a numerical value, it is compared with the value of the second argument. If they are the same, the **is** goal succeeds, otherwise it fails.

If the first argument is an atom, a compound term, a list, or a variable bound to one of these (none of which should happen), the outcome is implementation-dependent. It is likely that an error will occur.

36 Operators 01/03/2020

Arithmetic Operators…

- Note that a goal such as **X is X+1** will always fail, whether or not X is bound.

  ?- X is 10,X is X+1.

  no
- To increase a value by one requires a different approach.

Arithmetic Operators…

- /* Incorrect version */

  increase(N):-N is N+1.

  ?- increase(4).

  no

- /*Correct version */

  increase(N,M):-M is N+1.

  ?- increase(4,X).

  X = 5

Operator Precedence in Arithmetic Expressions

- When there is more than one operator in an arithmetic expression, e.g. **A+B*C-D**, Prolog needs a means of deciding the order in which the operators will be applied.
- For the basic operators such as + - * and / it is highly desirable that this is the customary 'mathematical' order, i.e. the expression **A+B*C-D** should be interpreted as 'calculate the product of B and C, add it to A and then subtract D'

Operator Precedence in Arithmetic Expressions…

- Prolog achieves this by giving each operator a numerical *precedence value*.
- Operators with relatively high precedence such as * and / are applied before those with lower precedence such as + and -.
- Operators with the same precedence (e.g. + and -, * and /) are applied from left to right.
- If a different order of evaluation is required this can be achieved by the use of brackets, e.g. **X is (A+B)*(C-D)**. Bracketed expressions are always evaluated first.

Relational Operators

- The infix operators =:= =\= > >= < =< are a special type known as *relational operators*. They are used to compare the value of two arithmetic expressions

- The goal succeeds if the value of the first expression is equal to, not equal to, greater than, greater than or equal to, less than or less than or equal to the value of the second expression, respectively

Relational Operators…

- Both arguments must be numbers, bound variables or arithmetic expressions (in which any variables are bound to numerical values).

```
?- 88+15-3=:=110-5*2.
yes
?- 100=\=99.
yes
```


Equality Operators

- There are three types of relational operator for testing equality and inequality available in Prolog

- The first type is used to compare the values of arithmetic expressions. The other two types are used to compare terms.

Equality Operators…

Arithmetic Expression Equality =:=

- **E1=:=E2** succeeds if the arithmetic expressions E1 and E2 evaluate to the same value.

```
?- 6+4=:=6*3-8.
yes
?- sqrt(36)+4=:=5*11-45.
yes
```

Equality Operators…

- To check whether an integer is odd or even we can use the **checkeven/1** predicate defined below.

```
checkeven(N):-M is N//2,N=:=2*M.
?- checkeven(12).
yes
?- checkeven(23).
no
?- checkeven(-11).
no
?- checkeven(-30).
yes
```


Equality Operators…

- The integer quotient operator // divides its first argument by its second and truncates the result to the nearest integer between it and zero
- So **12//2** is 6, **23//2** is 11, **-11//2** is -5 and **-30//2** is -15
- Dividing an integer by 2 using // and multiplying it by 2 again will give the original integer if it is even, but not otherwise.

Equality Operators…

Arithmetic Expression Inequality =\=

- **E1=\=E2** succeeds if the arithmetic expressions E1 and E2 do not evaluate to the same value

```
?- 10=\=8+3.
yes
```


Equality Operators…

Terms Identical ==

- Both arguments of the infix operator == must be terms. The goal *Term1==Term2* succeeds if and only if *Term1* is identical to *Term2*
- Any variables used in the terms may or may not already be bound, but no variables are bound as a result of evaluating the goal.

```
?- likes(X,prolog)==likes(X,prolog).
X = _
?- likes(X,prolog)==likes(Y,prolog).
no
```

- (X and Y are different variables)

Equality Operators…

?- X is 10,pred1(X)==pred1(10).

X = 10

?- X==0.

no

?- 6+4==3+7.

no

Equality Operators…

- The value of an arithmetic expression is only evaluated when used with the **is/2** operator
- Here **6+4** is simply a term with functor **+** and arguments **6** and **4**. This is entirely different from the term **3+7**.

Equality Operators…

Terms Not Identical \==

- *Term1* \==*Term2* tests whether *Term1* is not identical to *Term2*. The goal succeeds if *Term1*==*Term2* fails. Otherwise it fails.

  ?- pred1(X)\==pred1(Y).

  X = _ ,

  Y = _

- (The output signifies that both *X* and *Y* are unbound and are different variables.)

Equality Operators…

Terms Identical With Unification =

- The term equality operator = is similar to == with one vital (and often very useful) difference
- The goal *Term1*=*Term2* succeeds if terms *Term1* and *Term2* *unify*, i.e. there is some way of binding variables to values which would make the terms identical
- If the goal succeeds, such binding actually takes place

Equality Operators…

?- pred1(X)=pred1(10).

X = 10

(Variable *X* is bound to 10, which makes the two terms identical.)

?- likes(X,prolog)=likes(john,Y).

X = john ,

Y = prolog

- (Binding *X* to the atom *john* and *Y* to the atom *prolog* makes the two terms identical.)

Equality Operators…

?- X=0,X=:=0.

X = 0

- (*X*=0 causes *X* to be bound to 0. The goal **X=:=0** succeeds, which confirms that *X* now has the value zero.)

?- 6+4=3+7.

no

- (For the reason explained under ==.)

Equality Operators…

?- 6+X=6+3.

X = 3

- (Binding *X* to 3 makes the two terms identical. They are both 6+3, not the number 9.)

?- likes(X,prolog)=likes(Y,prolog).

X = Y = _

- (Binding *X* and *Y* makes the terms identical.)

?- likes(X,prolog)=likes(Y,ada).

no

- (No unification can make the atoms *prolog* and *ada* identical.)

Equality Operators…

Non-Unification Between Two Terms \=

- The goal *Term1*\=*Term2* succeeds if *Term1*=*Term2* fails, i.e. the two terms cannot be unified. Otherwise it fails.

?- 6+4\=3+7.

yes

Equality Operators...

?- likes(X,prolog)\=likes(john,Y).

no

- (Because binding *X* to *john* and *Y* to *prolog* will make the terms identical.)

?- likes(X,prolog)\=likes(X,ada).

X = _

Logical Operators

The *not* Operator

- The prefix operator **not/1** can be placed before any goal to give its negation
- The negated goal succeeds if the original goal fails and fails if the original goal succeeds.
- The following examples illustrate the use of **not/1**. It is assumed that the database contains the single clause

Logical Operators...

dog(fido).

?- not dog(fido).

no

?- dog(fred).

no

?- not dog(fred).

yes

?- X=0,X is 0.

X = 0

?- X=0,not X is 0.

no

Logical Operators...

The Disjunction Operator

- The disjunction operator **;/2** (written as a semicolon character) is used to represent 'or'. It is an infix operator that takes two arguments, both of which are goals.
- ***Goal1;Goal2*** succeeds if either ***Goal1*** or ***Goal2*** succeeds.

?- 6<3;7 is 5+2.

yes

?- 6*6=:=36;10=8+3.

yes

END