

## Formulas

We can represent propositional formulas as terms by treating the logical operators as function symbols.

$X \wedge Y$	<code>and(X, Y)</code>
$X \vee Y$	<code>or(X, Y)</code>
$X \longrightarrow Y$	<code>implies(X, Y)</code>
$\neg X$	<code>not(X)</code>

Thus the formula  $(p \wedge q) \vee (r \longrightarrow \neg s)$  can be represented by the term

`or(and(p, q), implies(r, not(s)))`

where  $p, q, r, s$  are constant symbols.

Suppose we are given facts for all the propositional letters:

```
atom(p).  
atom(q).  
atom(r).    (etc)
```

Computing the list of subformulas of a formula:

```
subformulas(X, [X]) :- atom(X).  
  
subformulas(not(X), [not(X)|R]): - subformulas(X, R).  
  
subformulas(and(X, Y), [and(X, Y)|R]): -  
    subformulas(X, R1), subformulas(Y, R2), concat(R1, R2, R).  
  
(etc)
```

### Arithmetic Expressions

Prolog reserves a set of symbols including “+”, “-”, “\*” for arithmetic operators. It treats these as function symbols, and allows arithmetic expressions to be written in infix notation. The usual priorities apply.

$+(2, *(X, Y))$  can also be written  $2 + X * Y$

Note that a ground arithmetic expression, like  $(3+4)*(8 +2)$  is *not* automatically evaluated. It is just a term like any other.

```
: X = 2 + 2 ?
```

```
X = 2 + 2
```

```
: 2 + 2 = 4 ?
```

```
** no
```

For evaluating arithmetic expressions. Prolog provides the special binary predicate symbol `is`, which can also be written in infix notation. The semantics of this is as follows:

Once an atom `T1 is T2` becomes the leftmost atom during a resolution proof,

1. It is checked that `T2` is a well formed *ground* arithmetic expression. If there are any variables, an error condition is raised.
2. If `T2` is a valid arithmetic expression, the value of `T2` is computed and *unified* with `T1`.

```
: X = +(2,*(4,7)), Y is X?
```

```
X = 2 + 4 * 7
```

```
Y = 30
```

```
: 4 is (2+2)?
```

```
** yes
```

```
: X is 2+ p?
```

```
ERROR: 2nd argument must be a number.
```

```
: X is Y?
```

```
ERROR: Unbound variable in expression evaluation.
```

### Summing a list - two ways

```
sum1([], 0).
```

```
sum1([X|Y],X+Z) :- sum1(Y,Z).
```

```
sum2([],0).
```

```
sum2([X|Y],Z) :- sum1(Y,T), Z is X+T.
```

```
: sum1([1,2,3],X), Y is X?
```

```
X = 1 + (2 + (3 +0)),
```

```
Y = 6
```

```
: sum2([1,2,3],X)?
```

```
X = 6
```

### Arithmetic Operators

$X + Y$     addition  
 $X - Y$     subtraction  
 $X * Y$     multiplication  
 $X/Y$       floating point division  
 $X//Y$      integer division  
 $X \text{ mod } Y$    modular arithmetic

### Comparisons

$X \neq Y$     inequality  
 $X < Y$     less than  
 $X \leq Y$    less than or equals  
 $X \geq Y$    greater than or equals

These assume that the arguments are bound to arithmetic expressions

```
: 2+2 <5?
```

```
** yes
```

```
: X <3?
```

```
ERROR: Unbound variable in expression evaluation.
```

```
In:        _R0 < 3
```

== is used for comparison of terms (without attempting to unify!)

```
: X = a(Y), Z = a(U), X = Z?
```

```
X = a(_R9)
```

```
Y = _R9
```

```
Z = a(_R9)
```

```
U = _R9
```

```
: X = a(Y), Z = a(U), X == Z?
```

```
** no
```

```
: X = a(Y), Z = a(Y), X == Z?
```

```
X = a(_R9)
```

```
Y = _R9
```

```
Z = a(_R9)
```

### Underscore

Underscore prefixed expressions such as “\_R9” in answers are “internal” variables used by Prolog during resolution.

An underscore “\_” in a program represents an unnamed variable. Each occurrence represents a *different* variable.

```
parent(X,Y) :- parents(X,Y,_).
```

```
parent(X,Y) :- parents(X,_,Y).
```

```
hasparents(X) :- parents(X,_,_).
```

## Trees

labelled binary trees can be represented as terms using the function symbols

- `node(Label,Left_Subtree,Right_subtree)`
- `leaf(Label)`

Example:

```
node(a, node(b,leaf(c),leaf(d)), leaf(e))
```

Trees with an arbitrary amount of branching can be represented using nested lists:

```
[Label,Subtree_1, . . . . , Subtree_n]
```

Example:

```
[a, [b, [c], [d], [e]], [f, [g], [h]]]
```

Computing the height of a tree in the nested list representation:

```
height([X],1).
```

```
height([X|R],H) :-  
    heights(R,L), max(L,H1), H is 1 + H1.
```

```
heights([],[]).
```

```
heights([X|R],[H|R1]) :- height(X,H), heights(R,R1).
```

```
max([X],X).
```

```
max([X|R],X) :- max(R,Z), X >= Z.
```

```
max([X|R],Z) :- max(R,Z), X < Z.
```

### The Generate and Test Paradigm

Constraint satisfaction problems can be solved using logic programs with the following pattern:

```
solve(Problem,Soln) :- possible_solution(Problem,Soln),  
                        solves(Problem,Soln).
```

where

- `possible_solution(P,S)` is a program that has been designed to *generate*, through backtracking, a set of answers for `S` that have the right structure for a solution of the problem `P`.
- `solves(P,S)` *tests* whether a given candidate solution `S` actually solves the problem `P`.

### Example: Graph Three Colouring

Suppose we represent a graph as a ground term of the following form:

```
graph( vertices([u,v,w,...]),  
       edges([edge(u,v),edge(v,w),....]) )
```

and a colouring as a list of the form

```
[col(u,red), col(v,blue), ....]
```

Suppose we have the following facts about colours

```
colour(red).  
colour(blue).  
colour(green).
```

We can generate all possible colourings of a list of vertices by means of the following:

```
add_colours([], []).  
add_colours([U|R], [col(U,C)|R1]) :-  
    colour(C), add_colours(R,R1).
```

We can test that no two adjacent nodes have the same colour by the following program:

```
colours_ok([],Col).
colours_ok([edge(U,V)|R],Col) :- colour_of(U,Col,C1),
                                colour_of(V,Col,C2),
                                C1 \= C2,
                                colours_ok(R,Col).
```

```
colour_of(Vert,[col(Vert,C)|_],C).
colour_of(Vert,[_|R],C):- colour_of(Vert,R,C).
```

The following program now generates all possible colourings and tests if they are OK:

```
three_colouring(graph(vertices(V),edges(E)),Col) :-
    add_colours(V,Col),
    colours_ok(E,Col).
```