Formulas

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

\[ X \land Y \quad \text{and}(X,Y) \]
\[ X \lor Y \quad \text{or}(X,Y) \]
\[ X \rightarrow Y \quad \text{implies}(X,Y) \]
\[ \neg X \quad \text{not}(X) \]

Thus the formula \((p \land q) \lor (r \rightarrow \neg s)\) can be represented by the term

\[ \text{or}(\text{and}(p,q), \text{implies}(r, \text{not}(s))) \]

where \(p,q,r,s\) are constant symbols.

Suppose we are given facts for all the propositional letters:

\[
\text{atom}(p).
\text{atom}(q).
\text{atom}(r). \quad \text{(etc)}
\]

Computing the list of subformulas of a formula:

\[
\text{subformulas}(X,[X]) :- \text{atom}(X).
\]
\[
\text{subformulas}(\neg(X), [\neg(X)|R]) :- \text{subformulas}(X,R).
\]
\[
\text{subformulas}(\text{and}(X,Y), [\text{and}(X,Y)|R]) :-
\text{subformulas}(X,R1), \text{subformulas}(Y,R2), \text{concat}(R1,R2,R).
\]
\[
\text{(etc)}
\]
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, \ast (X,Y)) can also be written \( 2 + X \ast Y \)

Note that a ground arithmetic expression, like \((3+4)\ast (8 +2)\) 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 \( \texttt{is} \), which can also be written in infix notation. The semantics of this is as follows:

Once an atom \( \texttt{T1 is T2} \) becomes the leftmost atom during a resolution proof,

1. It is checked that \( \texttt{T2} \) is a well formed ground arithmetic expression. If there are any variables, an error condition is raised.

2. If \( \texttt{T2} \) is a valid arithmetic expression, the value of \( \texttt{T2} \) is computed and unified with \( \texttt{T1} \).
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 mod Y  modular arithmetic

Comparisons

X \= Y  inequality
X < Y  less than
X =< Y  less than or equals
X >= 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,_,_).

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:

\[
\text{height}([\text{X}], 1).
\]

\[
\text{height}([\text{X}|\text{R}], H) :- \\
\quad \text{heights}(\text{R}, L), \text{max}(L, H1), H \text{ is } 1 + H1.
\]

\[
\text{heights}([], []).
\]

\[
\text{heights}([\text{X}|\text{R}], [H|R1]) :- \text{height}([\text{X}], H), \text{heights}([\text{R}], R1).
\]

\[
\text{max}([\text{X}], X).
\]

\[
\text{max}([\text{X}|\text{R}], X) :- \text{max}(\text{R}, Z), X >= Z.
\]

\[
\text{max}([\text{X}|\text{R}], Z) :- \text{max}(\text{R}, Z), X < Z.
\]

---

The Generate and Test Paradigm

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

\[
\text{solve(Problem, Soln)} :- \text{possible\_solution(Problem, Soln),}
\]

\[
\text{solves(Problem, Soln)}.
\]

where

- \text{possible\_solution(P, S)} is a program that has been designed to \textit{generate}, through backtracking, a set of answers for \textit{S} that have the right structure for a solution of the problem \textit{P}.

- \text{solves(P, S)} \textit{tests} whether a given candidate solution \textit{S} actually solves the problem \textit{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).
```