Arithmetic Terms are Symbolic

Evaluation of an arithmetic term into a numeric value must be *forced*.

That is, **1+2** is an infix representation of the relation **+(1,2)**. This term is *not* an integer!

Therefore

```
| ?- 1+2=3.
no
```

To force arithmetic evaluation, we use the infix predicate **is**.

The right-hand side of **is** must be all *ground terms* (literals or variables that are already bound). No *free* (unbound) variables are allowed.

Hence

```
|?- 2 is 1+1.
yes
| ?- X is 3*4.
X = 12
| ?- Y is Z+1.
! Instantiation error in argument
2 of is/2
! goal:  _10712 is _10715+1
```

The requirement that the right-hand side of an **is** relation be ground is essentially procedural. It exists to avoid having to invert complex equations. Consider,

```
(0 is (I**N)+(J**N)-K**N)), N>2.
```


Counting in Prolog

Rules that involve counting often use the **is** predicate to evaluate a numeric value.

Consider the relation **len(L,N)** that is true if the length of list **L** is **N**.

```
len([],0).
len([_|T],N) :-
   len(T,M), N is M+1.
| ?- len([1,2,3],X).
X = 3
| ?- len(Y,2).
Y = [_10903,_10905]
```

The symbols **_10903** and **_10905** are “internal variables” created as needed when a particular value is not forced in a solution.

Debugging Prolog

Care is required in developing and testing Prolog programs because the language is untyped; undeclared predicates or relations are simply treated as false.

Thus in a definition like

```
adj([A,B|_]) :- A=B.
adj([_,B|T]) :- adk([B|T]).
| ?- adj([1,2,2]).
no
```

(Some Prolog systems warn when an undefined relation is referenced, but many others don’t).

Similarly, given

```
member(A,[A|_]).
member(A,[_|T]) :-
  member(A,[T]).
| ?- member(2,[1,2]).
```

`Infinite recursion!` (Why?)

If you're not sure what is going on, Prolog's trace feature is very handy.

The command

`trace.`

turns on tracing. (**notrace** turns tracing off).

Hence

```
| ?- trace.
yes
[trace]
| ?-  member(2,[1,2]).
```



```
   (1) 0 Call: member(2,[1,2]) ?
   (1) 1 Head [1->2]:
member(2,[1,2]) ?
   (1) 1 Head [2]:
member(2,[1,2]) ?
   (2) 1 Call: member(2,[[2]]) ?
   (2) 2 Head [1->2]:
member(2,[[2]]) ?
   (2) 2 Head [2]:
member(2,[[2]]) ?
   (3) 2 Call: member(2,[[]]) ?
   (3) 3 Head [1->2]:
member(2,[[]]) ?
   (3) 3 Head [2]: member(2,[[]])
?
   (4) 3 Call: member(2,[[]]) ?
   (4) 4 Head [1->2]:
member(2,[[]]) ?
   (4) 4 Head [2]: member(2,[[]])
?
   (5) 4 Call: member(2,[[]]) ?
```


Termination Issues in Prolog

Searching infinite domains (like integers) can lead to non-termination, with Prolog trying *every* value.

Consider

```
odd(1).
odd(N) :- odd(M), N is M+2.
| ?- odd(X).
X = 1 ;
X = 3 ;
X = 5 ;
X = 7
```


A query

```
| ?- odd(X), X=2.
```

going into an *infinite* search, generating each and every odd integer and finding none is equal to 2!

The obvious alternative,

odd(2) (which is equivalent to **X=2, odd(X)**) also does an infinite, but fruitless search.

We'll soon learn that Prolog does have a mechanism to "cut off" fruitless searches.

Definition Order can Matter

Ideally, the order of definition of facts and rules should not matter.

But,

in practice definition order can matter. A good general guideline is to define facts before rules. To see why, consider a very complete database of **motherOf** relations that goes back as far as

```
motherOf(cain,eve).
```

Now we define

```
isMortal(X) :-
  isMortal(Y), motherOf(X,Y).
isMortal(eve).
```


These definitions state that the first woman was mortal, and all individuals descended from her are also mortal.

But when we try as trivial a query as

```
| ?- isMortal(eve).
```

we go into an infinite search!

Why?

Let's trace what Prolog does when it sees

```
| ?- isMortal(eve).
```

It matches with the first definition involving **isMortal**, which is

```
isMortal(X) :-
  isMortal(Y), motherOf(X,Y).
```

It sets **X=eve** and tries to solve

```
isMortal(Y), motherOf(eve,Y).
```

It will then expand **isMortal(Y)** into


```
isMortal(Z), motherOf(Y,Z).
```

An infinite expansion ensues.

The solution is simple—place the "base case" fact that terminates recursion *first*.

If we use

```
isMortal(eve).
isMortal(X) :-
  isMortal(Y), motherOf(X,Y).
yes
| ?-  isMortal(eve).
yes
```

But now another problem appears!

If we ask

```
| ?-  isMortal(clarkKent).
```

we go into another infinite search!

Why?

The problem is that Clark Kent is from the planet Krypton, and

hence won't appear in our **motherOf** database.

Let's trace the query.

It doesn't match

```
isMortal(eve).
```

We next try

```
isMortal(clarkKent) :-
  isMortal(Y),
  motherOf(clarkKent,Y).
```

We try **Y=eve**, but **eve** isn't Clark's mother. So we recurse, getting:

```
isMortal(Z), motherOf(Y,Z),
motherOf(clarkKent,Y).
```

But **eve** isn't Clark's grandmother either! So we keep going further back, trying to find a chain of descendents that leads from **eve** to **clarkKent**. No such chain exists, and there is no limit to how long a chain Prolog will try.

There is a solution though!

We simply rewrite our recursive definition to be

```
isMortal(X) :-
   motherOf(X,Y),isMortal(Y).
```

This is logically the same, but now we work from the individual **X** back toward **eve**, rather than from **eve** toward **X**. Since we have no **motherOf** relation involving **clarkKent**, we immediately stop our search and answer **no**!

Extra-logical Aspects of Prolog

To make a Prolog program more efficient, or to represent negative information, Prolog needs features that have a procedural flavor. These constructs are called "extra-logical" because they go beyond Prolog's core of logic-based inference.

The Cut

The most commonly used extra-logical feature of Prolog is the "cut symbol," "!"

A **!** in a goal, fact or rule "cuts off" backtracking.

In particular, once a **!** is reached (and automatically matched), we may *not backtrack* across it. The rule we've selected and the bindings we've already selected are "locked in" or "frozen."

For example, given

```
x(A) :- y(A,B), z(B), ! , v(B,C).
```

once the **!** is hit we can't backtrack to resatisfy **y(A,B)** or **z(B)** in some other way. We are locked into this rule, with the bindings of **A** and **B** already in place.

We *can* backtrack to try various solutions to **v(B,C)**.

It is sometimes useful to have several **!**'s in a rule. This allows us to find a partial solution, lock it in, find a further solution, then lock it in, etc.

For example, in a rule

```
a(X) - b(X), !, c(X,Y), ! , d(Y).
```

we first try to satisfy **b(X)**, perhaps trying several facts or rules that define the **b** relation. Once we have a solution to **b(X)**, we lock it in, along with the binding for **X**.

Then we try to satisfy **c(X,Y)**, using the fixed binding for **X**, but perhaps trying several bindings for **Y** until **c(X,Y)** is satisfied.

We then lock in this match using another **!**.

Finally we check if **d(Y)** can be satisfied with the binding of **Y** already selected and locked in.

When are Cuts Needed?

A cut can be useful in improving efficiency, by forcing Prolog to avoid useless or redundant searches.

Consider a query like

```
member(X,list1),
 member(X,list2), isPrime(X).
```

This asks Prolog to find an **X** that is in **list1** and also in **list2** and also is prime.

X will be bound, in sequence, to each value in **list1**. We then check if **X** is also in **list2**, and then check if **X** is prime.

Assume we find **X=8** is in **list1** and **list2**. **isPrime(8)** fails (of course). We backtrack to **member(X,list2)** and resatisfy it with the same value of **X**.

But clearly there is *never* any point in trying to resatisfy **member(X,list2)**. Once we know a value of **X** is in **list2**, we test it using **isPrime(X)**. If it fails, we want to go right back to **member(X,list1)** and get a different **X**.

To create a version of member that never backtracks once it has been satisfied we can use **!**.

We define

```
member1(X,[X|_]) :- !.
member1(X,[_|Y]) :-
  member1(X,Y).
```

Our query is now

```
member(X,list1),
 member1(X,list2), isPrime(X).
```

(Why isn't **member1** used in both terms?)

Expressing Negative Information

Sometimes it is useful to state rules about what *can't* be true. This allows us to avoid long and fruitless searches.

fail is a goal that always fails. It can be used to represent goals or results that can never be true.

Assume we want to optimize our **grandMotherOf** rules by stating that a male can never be anyone's grandmother (and hence a complete search of all **motherOf** and **fatherOf** relations is useless).

A rule to do this is

```
grandMotherOf(X,GM) :-
  male(GM), fail.
```

This rule doesn't do quite what we hope it will!

Why?

The standard approach in Prolog is to try other rules if the current rule fails.

Hence we need some way to "cut off" any further backtracking once this negative rule is found to be applicable.

This can be done using

```
grandMotherOf(X,GM) :-
  male(GM),!, fail.
```


Other Extra-Logical Operators

• **assert** and **retract**

These operators allow a Prolog program to add new rules during execution and (perhaps) later remove them. This allows programs to learn as they execute.

•**findall**

Called as **findall(X,goal,List)** where **X** is a variable in **goal**. All possible solutions for **X** that satisfy **goal** are found and placed in **List**.

For example,

```
findall(X,
(append(_,[X|_],[-1,2,-3,4]),(X<0)), L).
L = [-1,-3]
```


• **var** and **nonvar**

var(X) tests whether **X** is *unbound* (free).

nonvar(Y) tests whether **Y** is *bound* (no longer free).

These two operators are useful in tailoring rules to particular combinations of bound and unbound variables. For example,

```
grandMotherOf(X,GM) :-
  male(GM),!, fail.
```

might backfire if **GM** is not yet bound. We could set **GM** to a person for whom **male(GM)** is true, then fail because we don't want grandmothers who are male!

To remedy this problem. we use the rule only when **GM** is bound. Our rule becomes

```
grandMotherOf(X,GM) :-
 nonvar(GM), male(GM),!, fail.
```


An Example of Extra-Logical Programming

Factorial is a very common example program. It's well known, and easy to code in most languages.

In Prolog the "obvious" solution is:

```
fact(N,1) :- N =< 1.
fact(N,F) :- N > 1, M is N-1,
  fact(M,G), F is N*G.
```

This definition is certainly correct. It mimics the usual recursive solution.

But,

in Prolog "inputs" and "outputs" are less distinct than in most languages.

In fact, we can envision 4 different combinations of inputs and outputs, based on what is fixed (and thus an input) and what is free (and hence is to be computed):

1. **N** and **F** are both ground (fixed). We simply must decide if F=N!

2. **N** is ground and **F** is free. This is how **fact** is usually used. We must compute an **F** such that F=N!

3. **F** is fixed and **N** is free. This is an uncommon usage. We must find an **N** such that F=N!, or determine that no such **N** is possible.

4. Both **N** and **F** are free. We generate, in sequence, pairs of **N** and **F** values such that F=N!

Our solution works for combinations 1 and 2 (where **N** is fixed), but not combinations 3 and 4. (The problem is that **N =< 1** and **N > 1** can't be satisfied when **N** is free).

We'll need to use **nonvar** and **!** to form a solution that works for all 4 combinations of inputs.

We first handle the case where **N** is ground:

```
fact(1,1).
fact(N,1) :- nonvar(N), N =< 1, ! .
fact(N,F) :- nonvar(N), N > 1, !,
 M is N-1, fact(M,G), F is N*G, ! .
```

The first rule handles the base case of **N=1**.

The second rule handles the case of **N<1**.

The third rule handles the case of **N >1**. The value of **F** is computed recursively. The first **!** in each of these rules forces that rule to be the *only one* used for the values of **N** that match. Moreover, the second **!** in the third rule states that after **F** is computed, further backtracking is useless; there is only one **F** value for any given **N** value.

To handle the case where **F** is bound and **N** is free, we use

```
fact(N,F) :- nonvar(F), !,
 fact(M,G), N is M+1, F2 is N*G,
 F =< F2, !, F=F2.
```

In this rule we generate **N**, **F2** pairs until **F2 >= F**. Then we check if **F=F2**. If this is so, we have the **N** we want. Otherwise, no such **N** can exist and we fail (and answer no).

For the case where both **N** and **F** are free we use:

```
fact(N,F) :- fact(M,G), N is M+1,
 F is N*G.
```

This systematically generates **N**, **F** pairs, starting with **N=2**, **F=2** and then recursively building successor values (**N=3**, **F=6**, then **N=4**, **F=24**, etc.)

Parallelism in Prolog

One reason that Prolog is of interest to computer scientists is that its search mechanism lends itself to *parallel evaluation*.

In fact, it supports two different kinds of parallelism:

- AND Parallelism
- OR Parallelism

And Parallelism

When we have a goal that contains subgoals connected by the "," (And) operator, we may be able to utilize "and parallelism."

Rather than solve subgoals in sequence, we may be able to solve them in parallel *if* bindings can be properly propagated.

Thus in

```
a(X), b(X,Y), c(X,Z), d(Y,Z).
```

we may be able to first solve **`a(X)`**, binding **`X`**, then solve **`b(X,Y)`** and **`c(X,Z)`** *in parallel*, binding **`Y`** and **`Z`**, then finally solve **`d(Y,Z)`**.

An example of this sort of and parallelism is

```
member(X,list1),
 member1(X,list2), isPrime(X).
```

Here we can let **`member(X,list1)`** select an **`X`** value, then test **`member1(X,list2)`** and **`isPrime(X)`** in parallel. If one or the other fails, we just select another **`X`** from **`list1`** and retest **`member1(X,list2)`** and **`isPrime(X)`** in parallel.

OR Parallelism

When we match a goal we almost always have a choice of several rules or facts that may be applicable. Rather than try them in sequence, we can try several matches of different facts or rules in parallel. This is "or parallelism."

Thus given

```
a(X) :- b(X).
a(Y) :- c(Y).
```

when we try to solve

```
a(10).
```

we can simultaneously check both **`b(10)`** and **`c(10)`**.

Recall our definition of

```
member(X,L) :-
 append(P,[X|S],L).
```

where **append** is defined as

```
append([],L,L).
append([X|L1],L2,[X|L3]) :-
 append(L1,L2,L3).
```

Assume we have the query

`| ? member(2,[1,2,3]).`

This immediately simplifies to

`append(P,[2|S],[1,2,3]).`

Now there are two **append** definitions we can try in parallel:

(1) match `append(P,[2|S],[1,2,3])` with `append([],L,L)`. This requires that `[2|S] = [1,2,3]`, which must fail.

(2) match `append(P,[2|S],[1,2,3])` with `append([X|L1],L2,[X,L3])`.

This requires that `P=[X|L1]`, `[2|S]=L2`, `[1,2,3]=[X,L3]`. Simplifying, we require that `X=1`, `P=[1|L1]`, `L3=[2,3]`.

Moreover we must solve `append(L1,L2,L3)` which simplifies to `append(L1,[2|S],[2,3])`.

We can match this call to **append** in two different ways, so or parallelism can be used again.

When we try matching `append(L1,[2|S],[2,3])` against `append([],L,L)` we get `[2|S]=[2,3]`, which is satisfiable if `S` is bound to `[3]`. We therefore signal back that the query is true.

Speculative Parallelism

Prolog also lends itself nicely to *speculative* parallelism. In this form of parallelism, we "guess" or speculate that some computation *may* be needed in the future and start it early. This speculative computation can often be done in parallel with the main (non-speculative) computation.

Recall our example of

```
member(X,list1),
 member1(X,list2), isPrime(X).
```

After `member(X,list1)` has generated a preliminary solution for `X`, it is tested (perhaps in parallel) by `member1(X,list2)` and `isPrime(X)`.

But this value of `X` may be rejected by one or both of these

tests. If it is, we'll ask `member(X,list1)` to find a new binding for `X`. If we wish, this next binding can be generated *speculatively*, while the current value of `X` is being tested. In this way if the current value of `X` is rejected, we'll have a new value ready to try (or know that no other binding of `X` is possible).

If the current value of `X` is accepted, the extra speculative work we did is ignored. It wasn't needed, but was useful insurance in case further `X` bindings were needed.