**Arithmetic in Prolog**

Prolog provides a number of basic arithmetic tools for manipulating integers (that is, numbers of the form ...-3, -2, -1, 0, 1, 2, 3, 4...). Most Prolog implementation also provide tools for handling real numbers (or floating point numbers) such as 1.53 or 6 . 35 × 10 5, but we’re not going to discuss these, for they are not particularly useful for the symbolic processing tasks discussed in this book. Integers, on the other hand, are useful in connection with symbolic tasks (we use them to state the length of lists, for example) so it is important to understand how to work with them. We’ll start by looking at how Prolog handles the four basic operations of addition, multiplication, subtraction, and division.

| Arithmetic examples | Prolog Notation |
| --- | --- |
| 6 + 2 = 8 | 8  is  6+2. |
| 6 ∗ 2 = 12 | 12  is  6\*2. |
| 6 − 2 = 4 | 4  is  6-2. |
| 6 − 8 = − 2 | -2  is  6-8. |
| 6 ÷ 2 = 3 | 3  is  6/2. |
| 7 ÷ 2 = 3 | 3  is  7/2. |
| 1 is the remainder when 7 is divided by 2 | 1  is  mod(7,2). |

Note that as we are working with integers, division gives us back an integer answer. Thus 7 ÷ 2 gives 3 as an answer, leaving remainder 1.

Posing the following queries yields the following responses :

   ?-  8  is  6+2.  
   yes  
     
   ?-  12  is  6\*2.  
   yes  
     
   ?-  -2  is  6-8.  
   yes  
     
   ?-  3  is  6/2.  
   yes  
     
   ?-  1  is  mod(7,2).  
   yes

More importantly, we can work out the answers to arithmetic questions by using variables. For example:

   ?-  X  is  6+2.  
     
   X  =  8  
     
   ?-  X  is  6\*2.  
     
   X  =  12  
     
   ?-  R  is  mod(7,2).  
     
   R  =  1

Moreover, we can use arithmetic operations when we define predicates. Here’s a simple example. Let’s define a predicate add\_3\_and\_double/2 whose arguments are both integers. This predicate takes its first argument, adds three to it, doubles the result, and returns the number obtained as the second argument. We define this predicate as follows:

   add\_3\_and\_double(X,Y)  :-  Y  is  (X+3)\*2.

And indeed, this works:

   ?-  add\_3\_and\_double(1,X).  
     
   X  =  8  
     
   ?-  add\_3\_and\_double(2,X).  
     
   X  =  10

One other thing. Prolog understands the usual conventions we use for disambiguating arithmetical expressions. For example, when we write 3 + 2 × 4 we mean 3 + (2 × 4) and not (3 + 2) × 4, and Prolog knows this convention:

   ?-  X  is  3+2\*4.  
     
   X  =  11

### Arithmetic and Lists

Probably the most important use of arithmetic in this book is to tell us useful facts about data-structures, such as lists. For example, it can be useful to know how long a list is. We’ll give some examples of using lists together with arithmetic capabilities.

How long is a list? Here’s a recursive definition.

1. The empty list has length zero.
2. A non-empty list has length 1 + len (T), where len (T) is the length of its tail.

This definition is practically a Prolog program already. Here’s the code we need:

   len([],0).  
   len([\_|T],N)  :-  len(T,X),  N  is  X+1.

This predicate works in the expected way. For example:

   ?-  len([a,b,c,d,e,[a,b],g],X).  
     
   X  =  7

Now, this is quite a good program: it’s easy to understand and efficient. But there is another method of finding the length of a list. We’ll now look at this alternative, because it introduces the idea of accumulators. If you’re used to other programming languages, you’re probably used to the idea of using variables to hold intermediate results. An accumulator is the Prolog analog of this idea.

Here’s how to use an accumulator to calculate the length of a list. We shall define a predicate accLen /3 which takes the following arguments.

   accLen(List,Acc,Length)

Here List is the list whose length we want to find, and Length is its length (an integer). What about Acc ? This is the accumulator we will use to keep track of intermediate values for length (so it will also be an integer). Here’s what we do. When we call this predicate, we are going to give Acc an initial value of 0 . We then recursively work our way down the list, adding 1 to Acc each time we find a head element, until we reach the empty list. When we reach the empty list, Acc will contain the length of the list. Here’s the code:

   accLen([\_|T],A,L)  :-    Anew  is  A+1,  accLen(T,Anew,L).  
   accLen([],A,A).

The base case of the definition, unifies the second and third arguments. Why? Because this trivial unification is a nice way of making sure that the result, that is, the length of the list, is returned. When we reach the end of the list, the accumulator (the second variable) contains the length of the list. So we give this value (via unification) to the length variable (the third variable). Here’s an example trace. You can clearly see how the length variable gets its value at the bottom of the recursion and passes it upwards as Prolog is coming out of the recursion.

   ?-  accLen([a,b,c],0,L).  
         Call:  (6)  accLen([a,  b,  c],  0,  \_G449)  ?  
         Call:  (7)  \_G518  is  0+1  ?  
         Exit:  (7)  1  is  0+1  ?  
         Call:  (7)  accLen([b,  c],  1,  \_G449)  ?  
         Call:  (8)  \_G521  is  1+1  ?  
         Exit:  (8)  2  is  1+1  ?  
         Call:  (8)  accLen([c],  2,  \_G449)  ?  
         Call:  (9)  \_G524  is  2+1  ?  
         Exit:  (9)  3  is  2+1  ?  
         Call:  (9)  accLen([],  3,  \_G449)  ?  
         Exit:  (9)  accLen([],  3,  3)  ?  
         Exit:  (8)  accLen([c],  2,  3)  ?  
         Exit:  (7)  accLen([b,  c],  1,  3)  ?  
         Exit:  (6)  accLen([a,  b,  c],  0,  3)  ?

As a final step, we’ll define a predicate which calls accLen for us, and gives it the initial value of 0:

   leng(List,Length)  :-  accLen(List,0,Length).

So now we can pose queries like this:

   ?-  leng([a,b,c,d,e,[a,b],g],X).

Accumulators are extremely common in Prolog programs. (We’ll see another accumulator based program in this chapter, and some more in later chapters.) But why is this? In what way is accLen better than len ? After all, it looks more difficult. The answer is that accLen is tail recursive while len is not. In tail recursive programs, the result is fully calculated once we reached the bottom of the recursion and just has to be passed up. In recursive programs which are not tail recursive, there are goals at other levels of recursion which have to wait for the answer from a lower level of recursion before they can be evaluated. To understand this, compare the traces for the queries accLen([a,b,c],0,L) (see above) and len([a,b,c],0,L) (given below). In the first case the result is built while going into the recursion — once the bottom is reached at accLen([],3,\_G449) , the result is there and only has to be passed up. In the second case the result is built while coming out of the recursion; the result of len([b,c],  \_G481) , for instance, is only computed after the recursive call of len has been completed and the result of len([c],\_G489) is known. In short, tail recursive programs have less bookkeeping overhead, and this makes them more efficient.

   ?-  len([a,b,c],L).  
         Call:  (6)  len([a,  b,  c],  \_G418)  ?  
         Call:  (7)  len([b,  c],  \_G481)  ?  
         Call:  (8)  len([c],  \_G486)  ?  
         Call:  (9)  len([],  \_G489)  ?  
         Exit:  (9)  len([],  0)  ?  
         Call:  (9)  \_G486  is  0+1  ?  
         Exit:  (9)  1  is  0+1  ?  
         Exit:  (8)  len([c],  1)  ?  
         Call:  (8)  \_G481  is  1+1  ?  
         Exit:  (8)  2  is  1+1  ?  
         Exit:  (7)  len([b,  c],  2)  ?  
         Call:  (7)  \_G418  is  2+1  ?  
         Exit:  (7)  3  is  2+1  ?  
         Exit:  (6)  len([a,  b,  c],  3)  ?

### Comparing Integers

Some Prolog arithmetic predicates actually do carry out arithmetic all by themselves (that is, without the assistance of is ). These are the operators that compare integers.

| Arithmetic examples | Prolog Notation |
| --- | --- |
| x < y | X  <  Y. |
| x ≤ y | X  =<  Y. |
| x = y | X  =:=  Y. |
| x ⁄ = y | X  =\=  Y. |
| x ≥ y | X  >=  Y |
| x > y | X  >  Y |

These operators have the obvious meaning:

   ?-  2  <  4.  
   yes  
     
   ?-  2  =<  4.  
   yes  
     
   ?-  4  =<  4.  
   yes  
     
   ?-  4=:=4.  
   yes  
     
   ?-  4=\=5.  
   yes  
     
   ?-  4=\=4.  
   no  
     
   ?-  4  >=  4.  
   yes  
     
   ?-  4  >  2.  
   yes

Moreover, they force both their right hand and left hand arguments to be evaluated:

   ?-  2  <  4+1.  
   yes  
     
   ?-  2+1  <  4.  
   yes  
     
   ?-  2+1  <  3+2.  
   yes

Note that =:= is different from = , as the following examples show:

   ?-  4=4.  
   yes  
     
   ?-  2+2  =4.  
   no  
     
   ?-  2+2  =:=  4.  
   yes

That is, = tries to unify its arguments; it does not force arithmetic evaluation. That’s =:= ’s job.

Whenever we use these operators, we have to take care that any variables are instantiated. For example, all the following queries lead to instantiation errors.

   ?-  X  <  3.  
     
   ?-  3  <  Y.  
     
   ?-  X  =:=  X.

Moreover, variables have to be instantiated to integers . The query

   ?-  X  =  3,  X  <  4.

succeeds. But the query

   ?-  X  =  b,  X  <  4.

fails.

Ok, let’s now look at an example which puts Prolog’s abilities to compare numbers to work. We’re going to define a predicate which takes a non-empty list of non-negative integers as its first argument, and returns the maximum integer in the list as its last argument. Again, we’ll use an accumulator. As we work our way down the list, the accumulator will keep track of the highest integer found so far. If we find a higher value, the accumulator will be updated to this new value. When we call the program, we set the accumulator to an initial value of 0.

Here’s the code. Note that there are two recursive clauses:

   accMax([H|T],A,Max)  :-  
         H  >  A,  
         accMax(T,H,Max).  
     
   accMax([H|T],A,Max)  :-  
         H  =<  A,  
         accMax(T,A,Max).  
     
   accMax([],A,A).

The first clause tests if the head of the list is larger than the largest value found so far. If it is, we set the accumulator to this new value, and then recursively work through the tail of the list. The second clause applies when the head is less than or equal to the accumulator; in this case we recursively work through the tail of the list using the old accumulator value. Finally, the base clause unifies the second and third arguments; it gives the highest value we found while going through the list to the last argument.

Here’s an example query:

   ?-  accMax([1,0,5,4],0,Max).

Here the first clause of accMax applies, resulting in the following goal:

   ?-  accMax([0,5,4],1,Max).

**Exercise**

1. Define a 2-place predicate increment that holds only when its second argument is an integer one larger than its first argument. For example, increment(4,5) should hold, but increment(4,6) should not.

increment(X,Y):-X<Y.

1. Define a 3-place predicate sum that holds only when its third argument is the sum of the first two arguments. For example, sum(4,5,9) should hold, but sum(4,6,12) should not.

sum(A,B,C):-C is +(A,B).

**Exercise**  Write a predicate addone/2 whose first argument is a list of integers, and whose second argument is the list of integers obtained by adding 1 to each integer in the first list. For example, the query

   ?-  addone([1,2,7,2],X).

should give

   X  =  [2,3,8,3].

addone([],[]).

addone([H1|T1],[H2|T2]):-is(H2,H1+1),addone(T1,T2).

addone([1,2,7,2],X).

H1=1,T1=[2,7,2] H2=[] T2=[]

Check base =[][]---->fail

after is(H2,H1+1)

H1=1,T1=[2,7,2] H2=[2] T2=[]

Check base =[][]---->fail

addone([2,7,2],T2).

H1=2,T1=[7,2] H2=[2,3] T2=[]

Example: Another fundamental operation on vectors is the dot product . This operation combines two vectors of the same dimension and yields a number as a result. The operation is carried out as follows: the corresponding elements of the two vectors are multiplied, and the results added. For example, the dot product of [2,5,6] and [3,4,1] is 6+20+6 , that is, 32 . Write a 3-place predicate dot whose first argument is a list of integers, whose second argument is a list of integers of the same length as the first, and whose third argument is the dot product of the first argument with the second. For example, the query

   ?-  dot([2,5,6],[3,4,1],Result).

should yield

   Result  =  32

accDot([],[],A,A).

accDot([H1|T1], [H2|T2], A, Result) :-

is(Anew,+(A,\*(H1,H2))),

accDot(T1, T2, Anew, Result).

dot(Vector1, Vector2, Result) :-

accDot(Vector1, Vector2, 0, Result).