Tutorial SHEET-6

Arithmetic

1. Arithmetic in Prolog

Following are the Prolog handles of basic arithmetic operations:

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 a reminder of 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. Let's define a predicate `add_3_and_double2/2` 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
```

Prolog also 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
```

It is important to grasp that `+`, `-`, `*`, `/` and `mod` do not carry out any arithmetic. In fact, expressions such as `3+2`, `3-2` and `3*2` are simply terms. The functors of these terms are `+`, `-` and `*` respectively, and the arguments are `3` and `2`. Apart from the fact that the functors can go between their arguments these are ordinary Prolog terms, and unless we do something special, Prolog will not actually do any arithmetic. In particular, if we pose the query

```
?- X = 3+2.
```

we don't get back the answer X=5. Instead we get back

```
X = 3+2
yes
```
That is, Prolog has simply bound the variable `X` to the complex term `3+2`. It has not carried out any arithmetic. It has simply done what it usually does: performed unification Similarly, if we pose the query
```
?- 3+2*5 = X.
```
we get the response
```
X = 3+2*5
yes
```
To force Prolog to actually evaluate arithmetic expressions we have to use `is` just as we did in earlier examples. In fact, `is` does something very special: it sends a signal to Prolog that call up its inbuilt arithmetic capabilities and carries out the calculations.

Prolog's job is unifying variables to structures. Arithmetic is something extra that has been bolted on to the basic Prolog engine because it is useful. Thus, there are some restrictions on this extra ability, and we need to know what they are. For a start, the arithmetic expressions to be evaluated must be on the right hand side of `is`. In our earlier examples we carefully posed the query
```
?- X is 6+2.
X = 8
```
which is the right way to do it. If instead we had asked
```
6+2 is X.
```
we would have got an error message saying `instantiation_error`, or something similar. Moreover, although we are free to use variables on the right hand side of `is`, when we actually carry out evaluation, the variable must already have been instantiated to an integer. If the variable is uninstantiated, or if it is instantiated to something other than an integer, we will get some sort of `instantiation_error` message.

For example, `add_3_and_double/2` returns does not work the other way around. For example, we might hope that posing the query
```
add_3_and_double(X,12).
```
would return the answer X=3. But it doesn't! Instead we get the `instantiation_error` message.

For Prolog, `3 + 2` is just a term and really means the term `+(3,2)`. The expression `3 + 2` is just a user-friendly notation. This means that if we can give Prolog queries like:

```
X is +(3,2).
```

and Prolog will correctly reply

```
X = 5
```

Actually, you can even given Prolog the query

```
is(X,+(3,2)).
```

and Prolog will respond

```
X = 5
```

2. Arithmetic and Lists

Let's define the length of a list recursively:
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.

The corresponding Prolog program would be:

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

This program is good: efficient and correct. However, 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**, a standard Prolog technique which can be quite useful in other programs.

An accumulator is an analog to a variable to keep track of intermediate values.

Let's define a predicate `accLen3/` 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). `Acc` is a variable we will use to keep track of intermediate values for length. 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 do reach the empty set, `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. There are two reasons. The first is because 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). The second is that this trivial unification gives a nice way of stopping the recursion when we reach the empty list. Here's an example trace:

```
?- 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).
```

`accLen` is better than `len` because it is **tail recursive**. In tail recursive programs the result is all calculated once we reach the bottom of the recursion and just has to be passed up. In recursive programs which are not tail recursive

there are goals in one level of recursion which have to wait for the answer of 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)` and `len([a,b,c],0,L)`. In the first case the result is *built while going into the recursion* – once the bottom is reached at `accLen([],3,_G449)`. 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.

Trace of `len([a,b,c],L)`

```
?- 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)
```

3. Comparing Integers

Some built-in Prolog predicates that carry out arithmetic by themselves (without the assistance of `is`) are:

Arithmetic examples	Prolog Notation
$x < y$	`X < Y.`
$x \leq y$	`X =< Y.`
$x = y$	`X =:= Y.`
$x \neq y$	`X =\= Y.`
$x \geq y$	`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 `=:=` really 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.

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 list of non-negative integers as its first argument, and returns the maximum integer in the list as its last argument. 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).
```

However, initialising to 0 may be problematic if all the numbers in the list are negative. Hence we can simply initialize the accumulator `A` to `H`, the head of the list. Here is the predicate that uses `accMax`

```
max(List,Max) :- List = [H|_], accMax(List,H,Max).
```

Pose queries for the predicate and trace them.

4. Exercises

1. Define a 3-place predicate `accMin` which returns the minimum of a list of integers.

2. How does Prolog respond to the following queries?

a. X = 3*4.
b. X is 3*4.
c. 4 is X.
d. X = Y.
e. 3 is 1+2.
f. 3 is +(1,2).
g. 3 is X+2.
h. X is 1+2.
i. 1+2 is 1+2.
j. is(X,+(1,2)).
k. 3+2 = +(3,2).
l. *(7,5) = 7*5.
m. *(7,+(3,2)) = 7*(3+2).
n. *(7,(3+2)) = 7*(3+2).
o. *(7,(3+2)) = 7*(+(3,2)).

3. 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.

4. 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.

5. Write a predicate `addone2/` 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].`

6. In mathematics, an *n-dimensional vector* is a list of numbers of length n. For example, [2,5,12] is a 3-dimensional vector, and [45,27,3,-4,6] is a 5-dimensional vector. One of the basic operations on vectors is *scalar multiplication*. In this operation, every element of a vector is multiplied by some number. For example, if we scalar multiply the 3-dimensional vector [2,7,4] by 3 the result is the 3-dimensional vector [6,21,12].

Write a 3-place predicate `scalarMult` whose first argument is an integer, whose second argument is a list of integers, and whose third argument is the result of scalar multiplying the second argument by the first. For example, the query

```
scalarMult(3,[2,7,4],Result).
```

should yield

```
Result = [6,21,12]
```

7. 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. 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
```