3.13 Suggest an appropriate definition of operators ('was', 'of', 'the') to be able to write clauses like

diana was the secretary of the department.

and then ask Prolog:

?- Who was the secretary of the department.
Who = diana

?- diana was What.
What = the secretary of the department

3.14 Consider the program:

```
t( 0+1, 1+0).
t( X+0+1, X+1+0).
t( X+1+1, Z) :-
  t( X+1, X1),
  t( X1+1, Z).
```

How will this program answer the following questions if '+' is an infix operator of type **yfx** (as usual):

(a) **?- t(0+1, A).**

(b) **?- t(0+1+1, B).**

(c) **?- t(1+0+1+1+1, C).**

(d) **?- t(D, 1+1+1+0).**

3.15 In the previous section, relations involving lists were written as:

member(Element, List),
conc(List1, List2, List3),
del(Element, List, NewList), ...

Suppose that we would prefer to write these relations as:

Element in List,
concatenating List1 and List2 gives List3,
deleting Element from List gives NewList, ...

Define 'in', 'concatenating', 'and', etc. as operators to make this possible. Also, redefine the corresponding procedures.

3.4 Arithmetic

Prolog is mainly a language for symbolic computation where the need for numerical calculation is comparatively modest. Accordingly, the means for

doing arithmetic in Prolog are also rather simple. Some of the predefined operators can be used for basic arithmetic operations. These are:

+	addition
–	subtraction
*	multiplication
/	division
mod	modulo, the remainder of integer division

Notice that this is an exceptional case in which an operator may in fact invoke an operation. But even in such cases an additional indication to perform the action will be necessary. Prolog knows how to carry out the calculation denoted by these operators, but this is not entirely sufficient for direct use. The following question is a naive attempt to request arithmetic computation:

```
?- X = 1 + 2.
```

Prolog will 'quietly' answer

```
X = 1 + 2
```

and not X = 3 as we might possibly expect. The reason is simple: the expression 1 + 2 merely denotes a Prolog term where + is the functor and 1 and 2 are its arguments. There is nothing in the above goal to force Prolog to actually activate the addition operation. A special predefined operator, **is**, is provided to circumvent this problem. The **is** operator will force evaluation. So the right way to invoke arithmetic is:

```
?- X is 1 + 2.
```

Now the answer will be:

```
X = 3
```

The addition here was carried out by a special procedure that is associated with the operator +. We call such procedures *built-in procedures*.

There is no generally agreed notational convention for arithmetic in Prolog, so different implementations of Prolog may use somewhat different notations. For example, the '/' operator may denote integer division or real division, depending on the implementation. In this book, we will assume that '/' denotes real division, and that the **div** operator denotes integer division. Accordingly, the question

```
?- X is 3/2,
   Y is 3 div 2.
```

is answered by

```
X = 1.5
Y = 1
```

The left argument of the **is** operator is a simple object. The right argument is an arithmetic expression composed of arithmetic operators, numbers and variables. Since the **is** operator will force the evaluation, all the variables in the expression must already be instantiated to numbers at the time of execution of this goal. The precedence of the predefined arithmetic operators (see Figure 3.8) is such that the associativity of arguments with operators is the same as normally in mathematics. Parentheses can be used to indicate different associations. Note that +, –, *, / and **div** are defined as **yfx**, which means that evaluation is carried out from left to right. For example,

```
X is 5 - 2 - 1
```

is interpreted as

```
X is (5 - 2) - 1
```

Arithmetic is also involved when *comparing* numerical values. We can, for example, test whether the product of 277 and 37 is greater than 10000 by the goal:

```
?- 277 * 37 > 10000.
yes
```

Note that, similarly to **is**, the '>' operator also forces the evaluation.

Suppose that we have in the program a relation **born** that relates the names of people with their birth years. Then we can retrieve the names of people born between 1950 and 1960 inclusive with the following question:

```
?- born( Name, Year),
   Year >= 1950,
   Year =< 1960.
```

The comparison operators are as follows:

X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the values of X and Y are equal
X =\= Y	the values of X and Y are not equal

Notice the difference between the matching operators '=' and '=:='; for example, in the goals **X = Y** and **X =:= Y**. The first goal will cause the matching of the objects X and Y, and will, if X and Y match, possibly instantiate some variables in X and Y. There will be no evaluation. On the other hand, **X =:= Y** causes the arithmetic evaluation and cannot cause any instantiation of variables. These differences are illustrated by the following examples:

```
?- 1 + 2 =:= 2 + 1.

yes

?- 1 + 2 = 2 + 1.

no

?- 1 + A = B + 2.

A = 2
B = 1
```

Let us further illustrate the use of arithmetic operations by two simple examples. The first involves computing the greatest common divisor; the second, counting the items in a list.

Given two positive integers, X and Y, their greatest common divisor, D, can be found according to three cases:

(1) If X and Y are equal then D is equal to X.

(2) If X < Y then D is equal to the greatest common divisor of X and the difference Y – X.

(3) If Y < X then do the same as in case (2) with X and Y interchanged.

It can be easily shown by an example that these three rules actually work. Choosing, for example, X = 20 and Y = 25, the above rules would give D = 5 after a sequence of subtractions.

These rules can be formulated into a Prolog program by defining a three-argument relation, say

```
gcd( X, Y, D)
```

The three rules are then expressed as three clauses, as follows:

```
gcd( X, X, X).

gcd( X, Y, D) :-
  X < Y,
  Y1 is Y - X,
  gcd( X, Y1, D).
```

```
gcd( X, Y, D) :-
  Y < X,
  gcd( Y, X, D).
```

Of course, the last goal in the third clause could be equivalently replaced by the two goals:

```
X1 is X - Y,
gcd( X1, Y, D)
```

Our next example involves counting, which usually requires some arithmetic. An example of such a task is to establish the length of a list; that is, we have to count the items in the list. Let us define the procedure

```
length( List, N)
```

which will count the elements in a list **List** and instantiate N to their number. As was the case with our previous relations involving lists, it is useful to consider two cases:

(1) If the list is empty then its length is 0.

(2) If the list is not empty then **List = [Head | Tail]**; then its length is equal to 1 plus the length of the tail **Tail**.

These two cases correspond to the following program:

```
length( [], 0).

length( [_ | Tail], N) :-
  length( Tail, N1),
  N is 1 + N1.
```

An application of **length** can be:

```
?- length( [a,b,[c,d],e], N).

N = 4
```

Note that in the second clause of **length**, the two goals of the body cannot be swapped. The reason for this is that N1 has to be instantiated before the goal

```
N is 1 + N1
```

can be processed. With the built-in procedure **is**, a relation has been introduced that is sensitive to the order of processing and therefore the procedural considerations have become vital.

It is interesting to see what happens if we try to program the **length** relation without the use of **is**. Such an attempt can be:

```
length1( [], 0).

length1( [_ | Tail], N) :-
  length1( Tail, N1),
  N = 1 + N1.
```

Now the goal

```
?- length1( [a,b,[c,d],e], N).
```

will produce the answer:

```
N = 1+(1+(1+(1+0))).
```

The addition was never explicitly forced and was therefore not carried out at all. But in **length1** we can, unlike in **length**, swap the goals in the second clause:

```
length1( [_ | Tail], N) :-
  N = 1 + N1,
  length1( Tail, N1).
```

This version of **length1** will produce the same result as the original version. It can also be written shorter, as follows,

```
length1( [_ | Tail], 1 + N) :-
  length1( Tail, N).
```

still producing the same result. We can, however, use **length1** to find the number of elements in a list as follows:

```
?- length1( [a,b,c], N), Length is N.

N = 1+(1+(1+0))
Length = 3
```

To summarize:

- Built-in procedures can be used for doing arithmetic.
- Arithmetic operations have to be explicitly requested by the built-in procedure **is**. There are built-in procedures associated with the pre-defined operators +, −, *, /, **div** and **mod**.
- At the time that evaluation is carried out, all arguments must be already instantiated to numbers.

- The values of arithmetic expressions can be compared by operators such as <, =<, etc. These operators force the evaluation of their arguments.

Exercises

3.16 Define the relation

max(X, Y, Max)

so that **Max** is the greater of two numbers X and Y.

3.17 Define the predicate

maxlist(List, Max)

so that **Max** is the greatest number in the list of numbers **List**.

3.18 Define the predicate

sumlist(List, Sum)

so that **Sum** is the sum of a given list of numbers **List**.

3.19 Define the predicate

ordered(List)

which is true if **List** is an ordered list of numbers. For example, **ordered([1,5,6,6,9,12])**.

3.20 Define the predicate

subsum(Set, Sum, SubSet)

so that **Set** is a list of numbers, **SubSet** is a subset of these numbers, and the sum of the numbers in **SubSet** is **Sum**. For example:

```
?- subsum( [1,2,5,3,2], 5, Sub).

Sub = [1,2,2];
Sub = [2,3];
Sub = [5];
...
```

3.21 Define the procedure

between(N1, N2, X)

which, for two given integers N1 and N2, generates through backtracking all the integers X that satisfy the constraint $N1 \leqq X \leqq N2$.

3.22 Define the operators 'if', 'then', 'else' and ':=' so that the following becomes a legal term:

```
if X > Y then Z := X else Z := Y
```

Choose the precedences so that 'if' will be the principal functor. Then define the relation 'if' as a small interpreter for a kind of 'if-then-else' statement of the form

```
if Val1 > Val2 then Var := Val3 else Var := Val4
```

where **Val1**, **Val2**, **Val3** and **Val4** are numbers (or variables instantiated to numbers) and **Var** is a variable. The meaning of the 'if' relation should be: if the value of **Val1** is greater than the value of **Val2** then **Var** is instantiated to **Val3**, otherwise to **Val4**. Here is an example of the use of this interpreter:

```
?- X = 2, Y = 3,
   Val2 is 2*X,
   Val4 is 4*X,
   if Y > Val2 then Z := Y else Z := Val4,
   if Z > 5 then W := 1 else W := 0.

X = 2
Y = 3
Z = 8
W = 1
```

Summary

- The list is a frequently used structure. It is either empty or consists of a *head* and a *tail* which is a list as well. Prolog provides a special notation for lists.
- Common operations on lists, programmed in this chapter, are: list membership, concatenation, adding an item, deleting an item, sublist.
- The *operator notation* allows the programmer to tailor the syntax of programs toward particular needs. Using operators the readability of programs can be greatly improved.
- New operators are defined by the directive **op**, stating the name of an operator, its type and precedence.
- In principle, there is no operation associated with an operator; operators are merely a syntactic device providing an alternative syntax for terms.
- Arithmetic is done by built-in procedures. Evaluation of an arithmetic expression is forced by the procedure **is** and by the comparison predicates <, =<, etc.