2. Write a function `remainder` which defines the remainder after integer division using only subtraction. Ignore the possibility of division by zero.
3. Write a function `divide` which defines integer division using only addition and subtraction. Ignore division by zero.
4. Here are some exercises with divisibility: show for all natural numbers x, y and z that

(a)	$1 \mid y$	(b)	$x \mid y \wedge x \mid z \wedge y \geq z \rightarrow x \mid (y - z)$
(c)	$x \mid 0$	(d)	$x \mid y \wedge y \mid z \rightarrow x \mid z$
(e)	$x \mid x$	(f)	$x \mid y \wedge x \mid z \rightarrow x \mid (y + z)$
(g)	$0 \mid y \leftrightarrow y = 0$	(h)	$x \mid y \wedge y \mid x \rightarrow x = y$

5. (a) Use the method of 'multiplication without multiplying' to compute exponentiation, `power` x n= x^n, making use of the facts that

$$x^n = x^{n \texttt{ div } 2} \times x^{n \texttt{ div } 2} \text{ if } n \text{ is even}$$

and

$$x^n = x^{n \texttt{ div } 2} \times x^{n \texttt{ div } 2} \times x \text{ if } n \text{ is odd}$$

   (b) Write a Miranda function, `multiplications`, that computes the number of multiplications performed by `power`(x, n) given the value of n. How would this compare with the corresponding count of multiplications for a more simple-minded recursive calculation of x^n, using $x^{n+1} = x^n * x$?
6. (Tricky) Specify and define a function `middle` to find the middle one of three numbers. Prove that the definition satisfies its specification.
7. Prove that the principles of simple induction and course of values induction are equivalent. In other words, though course of values induction looks stronger (can prove more things), it is not.
   First, show that any simple induction proof can easily be converted into a course of values induction proof.
   Second, show that if you have a course of values induction proof of $\forall n : nat.\ P(n)$ then its ingredients can be used to make a simple induction proof of $\forall n : nat.\ (\forall m : nat.\ (m < n \rightarrow P(m)))$, and that this implies $\forall n : nat.\ P(n)$.
8. Newton's method for calculating a square root $\sqrt{x}$ works by producing a sequence $y_0, y_1, \ldots$ of better and better approximations to the answer, where

$$y_{n+1} = \frac{1}{2}(y_n + \frac{x}{y_n})$$

   The starting approximation y_0 can be very crude — we shall use $x + 1$. We shall deem y_n accurate enough when $\mid y_n^2 - x \mid <$ `epsilon`, `epsilon` being some small number defined elsewhere in the program (for instance, `epsilon = 0.01`). Here is a Miranda definition:

```
newtonsqrt::num -> num
||pre: x >= 0 & epsilon > 0
||post: abs(r*r - x) < epsilon & r >= 0
||      where r = newtonsqrt x
newtonsqrt x = ns1 x (x+1)
ns1::num -> num -> num
||pre: x >= 0 & epsilon > 0
||     & a >= 0 & a*a >= x & (a = 0 -> x = 0)
||post: abs(r*r - a) < epsilon & r >= 0
||      where r = ns1 x a
ns1 x a = a,        if a*a - x < epsilon
   = ns1 x ((a + x/a)/2),  otherwise
```

(The last three pre-conditions of `ns1` need some thought. $a \geq 0$ looks reasonable enough, $a = 0 \to x = 0$ avoids the risk of dividing by zero, and $a^2 \geq x$ is not strictly necessary but, as we shall see, it makes it easier to find a recursion variant.)

(a) Show that `newtonsqrt` and `ns1` satisfy their specification, assuming that the recursive call in `ns1` works correctly. This is easy, and the proof is finished once we have found a recursion variant; that is the difficult part!

(b) If $x \geq 0$, $a^2 \geq x$ and $b = \frac{1}{2}(a + \frac{x}{a})$ (for instance, if $a = y_n$ and $b = y_{n+1}$), show that

$$0 \leq b^2 - x = \frac{1}{4}(1 - \frac{x}{a^2})(a^2 - x) \leq \frac{1}{4}(a^2 - x)$$

(c) The basis for a recursion variant is $a^2 - x$. As this gets smaller, the approximation gets better and we are making progress towards the answer. However, as it stands it cannot be a recursion variant because it is not a natural number. (Unlike the case with natural numbers, a positive real number can decrease strictly infinitely many times, by smaller and smaller amounts.) Use (b) to show that a suitable variant is

$$max(0, 1 + entier(\log_4 \frac{a^2 - x}{epsilon}))$$

(This gives a number that — by (b) — decreases by at least 1 each time, *entier* turns it into an integer, and dividing $a^2 - x$ by *epsilon* ensures that this integer is a natural number except for the last time round, which is coped with by $max(0, 1 + \ldots)$.)

Chapter 6

Lists

6.1 Introduction

The various data types encountered so far, such as `num` and `bool`, are capable of holding only one data value at a time. However, it is often necessary to represent a number of related items of data in some way and then be able to have a single name which refers to these related items. What is required is an *aggregate type*, which is a data type that allows more than one item of data to be referenced by a single name. Aggregate types are also called *data structures* since they represent a collection of data in a structured and orderly manner.

In this chapter we introduce the list aggregate type, together with the various predefined operators and functions in Miranda that manipulate lists. We shall also see how to use lists of characters to represent strings.

6.2 The list aggregate type

Lists are used to list values (the *elements* of the list) of the same type, and they can be written in Miranda using square brackets and commas. The following are examples of lists of numbers, Booleans, other lists, and functions — notice how we also use square brackets for describing the list *types*. (In mathematics square brackets are also used for bracketing expressions, but the two uses are distinguishable by context.)

`[1,2,3]`	is of type	`[num]`
`[False,False,True]`	"	`[bool]`
`[[1,2],[],[3]]`	"	`[[num]]`
`[(+),(*)]`	"	`[num -> num -> num]`

The third example is a valid list since the elements of the list have the same type; they are all lists of numbers. The *empty* list `[]`, which has no elements, is rather special because it could be of type `[*]`, where the symbol `*` represents *any* type. (In fact, if you enter `[]::` in Miranda, which asks for the type of `[]`, the system will respond `[*]`.) Similarly, the fourth example illustrates a valid list since all its elements have the same type, namely functions that map two numbers to a number.

A list `[x]` with just one element is known as a *singleton* list. Two lists are equal if and only if they have the same values with the same number of occurrences in the same order. Otherwise they are different, so the lists

```
[1,2]  [2,1]  [1,1,2]  [1,2,1]  [2,1,1]
```

are all different even though they have the same elements 1 and 2.

Concatenation

The most important operator for lists is `++` (called *concatenate* or *append*), which joins together two lists of the same type to form a single composite list. For example,

```
[1,2,3]++[1,5] = [1,2,3,1,5]
```

We shall see shortly that there is another method for building up lists, called *cons*; none the less `++` is usually conceptually more natural, and it is often useful in specifications. We can formalize the condition that a value x is an element of a list xs as

$$\exists us, vs.\ (xs = us\texttt{++[}x\texttt{]++}vs)$$

Note that, like `+` and `*`, `++` is *associative*: the equation

$$xs\texttt{++}(ys\texttt{++}zs) = (xs\texttt{++}ys)\texttt{++}zs$$

always holds, and so you might as well write `xs++ys++zs`. In fact, there is no need for brackets for any number of lists appended together. Concatenating any list xs with the empty list `[]` returns the given list. This is called the *unit law* and `[]` is the *unit* (just like 0 for `+` or 1 for `*`) with respect to `++`:

$$xs\texttt{++[]} = \texttt{[]++}xs = xs$$

List deconstruction

The function `hd` (pronounced *head*) selects the first element of a list, and `tl` (pronounced *tail*) selects the remaining portion:

```
hd [1,2,3] = 1
tl [1,2,3] = [2,3]
```

Notice the type difference — the result of `hd` is an *element*, that of `tl` is *another list*. It is an error to apply either of these functions to an empty list, and so appropriate tests must be carried out (using guards or pattern matching) to avoid such errors.

Indexing and finding lengths of lists

A list can be indexed by a natural number n in order to find the value appearing at a given position using the `!` infix operator:

```
[11,22,33] ! 1 = 22
[10,200,3000] ! 0 = 10
```

Note that the first element of the list has index 0: $xs!0$ = `hd` xs. Thus, one would use the index $n-1$ for the nth element of a list.

The prefix operator `#` returns the length of a list (that is, the number of elements that it contains):

```
#[] = 0
#[x] = 1
#[1,1,2,2,3,3] = 6
#(xs++ys) = (#xs) + (#ys)
```

Cons

The *cons* (for *construct*) operator `:` is an inverse of `hd` and `tl`. It takes a value and a list (of matching types) and puts the value in front to form a new list, for example,

```
1:[2,3,4] = [1,2,3,4] = 1:2:3:4:[]
x:xs = [x]++xs
hd (x:xs) = x
tl (x:xs) = xs
xs = (hd xs):(tl xs),       if xs ~= []
```

Some convenient notations for lists

The special form `[a..b]`, where `a` and `b` are numbers, denotes the list of numbers `[a,a+1,a+2, ...b]` in increasing order from `a` to `b` inclusive. This will be `[]` if `a > b`.

Lists of characters (also called *strings*) can alternatively be denoted by using double quotation marks. For example, `"hello"`.

```
Miranda  "cow" ++ "boy"
cowboy
```

An important feature of strings is how they are printed.

```
Miranda  "cowboy"
cowboy
Miranda  ['c','o','w','b','o','y']
cowboy
Miranda  "this line has \none newline"
This line has
one newline
```

The double quotation marks do not appear in the output and special characters are printed as the character they represent. This printing convention gives programmer control over the layout of results.

Cons as constructor

From the human point of view, there is often nothing to indicate that one end of a list should be given any preference over the other. However, functional programming interpreters store the elements in a manner such that those elements from one end are much more accessible than those from the other.

Imagine a list as having its elements all parcelled up together, but in a nested way. If you unwrap the parcel you find just one element, the head, and another parcel containing the tail. (The empty list is special, of course.) The further down the sequence a value is, the more difficult it is to get out, because you have to unwrap more parcels.

From this point of view, the most accessible element in a list is the *first*, that is, the leftmost in the `[...]` notation.

Storing a list `[x0,x1,x2,...,xn]` in this way corresponds notationally to writing it, using *cons*, as `x0:x1:x2:...:xn:[]`; and the way the function *cons* is applied in the computer, for example to evaluate `x:xs`, does not perform any real calculations, but, rather, just puts `x` and `xs` together wrapped up in a wrapper that is clearly marked ':'. (The empty list is just a wrapper, marked '*empty*'.) A function implemented in this way is called a 'constructor' function, and there are some more examples in Chapter 7. Obviously, a crucial aspect is that you can unwrap to regain the original arguments, so it is important that : is 'one-to-one' — different arguments give different results — or, more formally,

$$\forall x, y, xs, ys.\ (x : xs = y : ys \rightarrow x = y \wedge xs = ys)$$

`++` is not one-to-one and so could never be implemented as a constructor function, but `snoc`, defined by

```
snoc [] = []
snoc xs x = xs++[x]
```

is one-to-one and could have been implemented as a constructor function for lists instead of :, but it is not.

Special facilities for pattern matching on lists

Because every list can be expressed in terms of `[]` and `:` in *exactly one way*, we can pattern match on lists using `[]` and `:`. For example, any of the following will match a two-element list.

```
 a:b:[]   a:[b]   [a,b]
```

Figure 6.1 shows the function `isempty` which uses pattern matching to determine if a given list is empty or not. Of course, an easier definition is

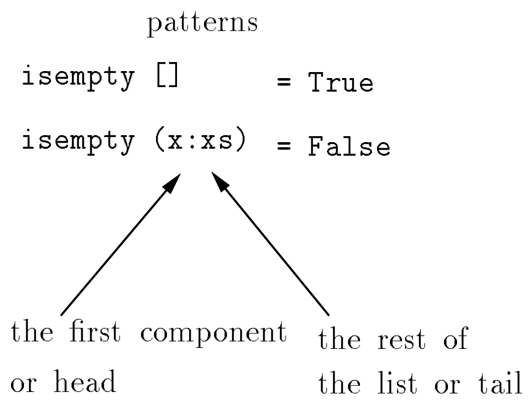

Figure 6.1

just `isempty x = x = []`. Similarly, we can formally define `hd` and `tl` (not that one would need to) by:

```
hd  (x:xs)   =   x
tl  (x:xs)   =   xs
```

Notice how pattern matching does not just express implicit tests on the actual arguments (Are they empty or non-empty? Is the wrapper marked *empty* or *cons*?) as we saw in Section 4.4; it also provides the right-hand side of the equation with names for the unwrapped contents of the arguments.

6.3 Recursive functions over lists

Because of the way in which lists are stored, recursion (and also induction) on lists is usually based on two cases: the empty list `[]`, and lists of the form (*x*:*xs*). As an example, consider the function which finds the length of a list (that is, the operator `#`):

```
length :: [num] -> num
||pre: none
||post: length xs = #xs
length [] = 0
length (x:xs) = 1 + (length xs)
```

which can be evaluated as follows:

$\underline{\texttt{length [10,20,30]}}$	
$= 1+\underline{(\texttt{length [20,30]})}$	by the second equation
$= 1+(1+\underline{(\texttt{length [30]})})$	’
$= 1+(1+(1+\underline{(\texttt{length []})}))$	’
$= 1+(1+(1+(0)))$	by the first equation
$= 3$	by built-in rules for `+`

Of course, we should ask what the recursion variant of `length` *xs* is; it is just `#`*xs* — in the recursive call, the length of the argument has gone down by 1. In fact, it is almost always the case for recursively defined list functions that the recursion variant is the length of some list.

That is pretty silly in this example. Either we are assuming that the length function `#` already exists, in which case there is no point in redefining it as `length`, or we are not, in which case we cannot use it for a recursion variant. However, there is an important lesson to be drawn regarding *infinite lists*.

Infinite lists

Some lists in Miranda can be *infinite*, such as the following examples:

```
zeros = 0:zeros             || = [0,0,0,...]
nandup n = n:(nandup (n+1)) || = [n,n+1,n+2,...]
cards = nandup 0            || = [0,1,2,3,...]
```

Some calculations using these will be potentially infinite, and you will need to press *control-C* when you have had enough. For instance, evaluating `zeros` or `cards` will start to produce an infinite quantity of output, and evaluating `#zeros` or `#cards` will enter an infinite loop.

However, the lazy evaluation of Miranda means that it will not go into infinite computations unnecessarily. For instance, `hd (tl cards)` gives 1 as its result and stops.

Now the problem is that we thought we had proved that `length` *xs* always terminates, because it has a recursion variant `#`*xs*; `length zeros` does not terminate, and this is because the variant `#zeros` is undefined (or infinite, which is just as bad). The moral is:

> Our reasoning principles using recursion variants only work for *finite* lists.

This is a shame because infinite lists can be useful and well-behaved; in fact research into finding the most convenient ways of reasoning about infinite lists is ongoing. However, we shall only deal with finite lists and shall make the implicit assumption — usually amounting to an implicit pre-condition — that our lists are finite. Then we can use their lengths as recursion variants, and the 'circular reasoning' technique for recursion works exactly as before.

Another example

The following is a less trivial example. It tests whether a given number occurs as an element of a given list of numbers. Note how this condition can be expressed precisely using `++` in the specification. If x is an element of xs, then xs can be split up as us`++[`x`]++`vs, where us and vs are the sublists of xs coming before and after some occurrence of x:

```
isin ::  num -> [num] -> bool
||pre:  none
||post: isin x xs <-> (E)us,vs:[num].  xs = us++[x]++vs
||recursion variant = #xs
isin x [] = False
isin x (y:ys) = True,      if x = y
              = isin x ys, otherwise
```

The recursion variant in `isin` x xs is `#`xs, and we can reason that `isin` x xs works correctly as follows.

Proposition 6.1 `isin` x meets its specification. If $xs =$ `[]`, then we cannot possibly have $xs = us$`++[`x`]++`vs, for that would have length at least 1. Hence the result `False` is correct.

If xs has the form $(y\colon ys)$ then note that, from the definition, `isin` x $(y\colon ys) \equiv (x = y) \vee$ `isin` x ys. Hence we must prove

$$(x = y) \vee \texttt{isin}\ x\ ys \leftrightarrow \exists us, vs.\ ((y\colon ys) = us\texttt{++[}x\texttt{]++}vs)$$

assuming that the recursive call works correctly. For the $\rightarrow$ direction, we have the following two cases:

1. If $x = y$ then $(y\colon ys) =$ `[]++[`x`]++`ys.
2. If `isin` x ys then by induction $ys = U$`++[`x`]++`V for some U and V and so $y\colon ys = (y\colon U)$`++[`x`]++`V.

For the $\leftarrow$ direction, we have $(y{:}\,ys) = U$`++[`x`]++`V for some U and V (not necessarily the same as before). If $U =$ `[]` then $y = x$, while if $U \neq$ `[]` then $ys =$ (`tl` U)`++[`x`]++`V and so `isin` x ys by induction. □

Although this may look a little too much like hard work, something of value has been achieved. The post-condition is very much a *global* property of the function — a property of what has been calculated rather than how the calculation was done. It is tempting to think of the function definition itself as a formal description of what the intuition 'x is an element of the list xs' means, but actually the specification comes closer to the intuitive idea. You can see this if you think how you might prove such intuitively obvious facts as 'if x is in xs then it is also in xs++ys and ys++xs for any ys' — this is immediate from the specification, but less straightforward from the definition.

Let us note one point that will be dealt with properly in Chapter 7, but is useful already. You could replace `num` in `isin` by `char` or `bool` or `[num]` or any other type at all to give other versions of `isin`, but the actual definition would not suffer *any changes whatsoever*: it is 'polymorphic' (many formed), and it is useful to give its type 'polymorphically' as `* -> [*] -> bool`, leaving `*` to be replaced by whatever type you actually want. Indeed, Miranda itself understands these polymorphic types.

6.4 Trapping errors

The evaluator will generate a run-time error message for cases where no matching equation has been found for a particular function application. However, it is always a good idea not to rely on this. Either convince yourself that your program cannot cause a run-time error, or — for a defensive specification — traps errors at the program level. In this way it is possible to generate more meaningful error messages and to bring the execution to a graceful halt. Such program generated information may then be more useful for debugging purposes. The predefined function

```
error :: [char] -> *
```

can be used for this purpose. (The `*` means that the result of `error` — actually not a result at all because the program has aborted — can be considered formally to be of any type: it will not cause type checking errors.)

As examples, the following are defensive specifications for `hd` and `divide`. Again, the `*`s represent any type:

```
hd  :: [*] -> *
||pre:  none
||post: (E)ys:[*]. xs = [hd xs]++ys
||      \/ xs = [] & error message generated
hd (x:ys) = x
hd  [] = error "hd of []"
```

```
divide :: num -> num -> num
||pre:  none
||post: y ~= 0 & x = (divide x y)*y
||      \/ y = 0 & error message generated
divide x 0 = error "Sorry! divide by 0"
divide x y = x/y
```

It is good programming practice to ensure that a given function performs just one activity. So it is better if a defensive function performs the validations (the checks) and error responses itself, but calls on a separate non-defensive function to perform the actual calculations.

6.5 An example — insertion sort

Here we will consider a slightly larger problem and use a top-down design technique to arrive at a solution. We shall look at the problem of sorting data items into ascending order. There are many algorithms for doing this, and one of simplest methods — though not a very efficient one — is the *insertion* sort, which sorts a list by first sorting the tail and then inserting the head in the correct place. We shall look at a more efficient algorithm, 'quick sort', in Chapter 12.

Sortedness

Let us start by specifying when a list is sorted (in ascending order) — if $xs = [x_0, x_1, x_2, \ldots, x_n]$ then we write *Sorted(xs)* to mean that informally

$$x_0 \leq x_1 \leq x_2 \leq \ldots \leq x_n$$

This can be formalized quite straightforwardly using the subscripting operator ! but another way, using ++, is as follows:

$$Sorted(xs) \overset{\text{def}}{=} \forall us, vs : [\,*\,].\ \forall a, b : *.\ \ xs = us\texttt{++}[a, b]\texttt{++}vs \rightarrow a \leq b$$

In other words, whenever we have two adjacent elements a and b in xs (with a first), then $a \leq b$.

Note that we used a polymorphic type — we wrote * for the type of the elements, [*] for that of the lists. Of course, it only makes sense to call a list sorted if we know what $\leq$ means for its elements. It is obvious how to do this when their type is num, but Miranda understands $\leq$ for many other types. For instance, values of type char have a natural ordering (by ASCII code), and this is extended to strings (values of type [char]) by lexicographic ordering and to values of other list types by the same method. The sorting algorithm works 'polymorphically' — it does not depend on the type. We shall therefore express its type using $*$, but remember (as implicit pre-conditions) that $*$ must represent a type for which $\leq$ is understood.

Let us prove some useful properties about sortedness.

Proposition 6.2

1. The empty list `[]` and singleton $[x]$ are sorted.
2. $[x, y]$ is sorted iff $x \leq y$.
3. If xs is sorted, then so is any sublist ys (that is, such that we can write $xs = xs_1$++ys++xs_2 for some lists xs_1 and xs_2).
4. Suppose xs++ys and ys++zs are both sorted, and ys is *non-empty*. Then xs++ys++zs is sorted.

Proof

1. This is obvious, because the decomposition $xs = us$++$[a, b]$++vs can only be done if #$xs \geq 2$.
2. This is obvious, too.
3. If $ys = us$++$[a, b]$++vs, then $xs = (xs_1$++$us)$++$[a, b]$++$(vs$++$xs_2)$, and so $a \leq b$ because xs is sorted.
4. Suppose xs++ys++$zs = us$++$[a, b]$++vs. It is clear that a and b are either both in xs ++ys or both in ys ++zs, and so $a \leq b$.

□

The third case, set out in full using box notation (Chapters 16 and 17), can be seen in Figure 6.2.

	xs is sorted	
	$\forall a, b, us, vs.\ (xs = us$++$[a, b]$++$vs \rightarrow a \leq b)$	def of sublist
	$xs = xs1$++ys++$xs2$	assumption
$\forall\mathcal{I}\ \ A, B, US, VS$		
	$ys = US$++$[A, B]$++VS	
	$xs = xs_1$++US++$[A, B]$++VS++xs_2	def sublist
	$A \leq B$	assoc of ++ and $\forall\rightarrow\mathcal{E}$
	$ys = US$++$[A, B]$++$VS \rightarrow A \leq B$	$\rightarrow\mathcal{I}$
	$\forall a, b, us, vs.\ (ys = us$++$[a, b]$++$vs) \rightarrow a \leq b$	$\forall\mathcal{I}$
	ys is sorted	def

Figure 6.2

When we sort a list, we obviously want the result to be sorted, and this will be specified in the post-condition. The other property that we need is that the result has the same elements as the argument, but possibly rearranged — the result is a *permutation* of the argument.

Let us write *Perm(xs,ys)* for '*ys* is a permutation of *xs*'. We shall not define this explicitly in formal terms, but use the following facts:

- $Perm(xs,xs)$
- $Perm(xs,ys) \to Perm(ys,xs)$
- $Perm(xs,ys) \wedge Perm(ys,zs) \to Perm(xs,zs)$
- $Perm(us{+}{+}vs{+}{+}ws{+}{+}xs{+}{+}ys,\ us{+}{+}xs{+}{+}ws{+}{+}vs{+}{+}ys)$, that is, *vs* and *xs* are swapped

In fact, any permutation can be produced by a sequence of swaps of adjacent elements. We are now ready to specify the function `sort`:

```
sort :: [*] -> [*]
||pre:  none (but, implicitly, there is an ordering over *)
||post: Sorted(ys) & Perm(xs,ys)
||        where ys = sort xs
```

Recall that the method of insertion sort was to `sort x:xs` by first sorting `xs` and then inserting `x` in the correct place. We therefore define

```
sort []       =   []
sort (x:xs)   =   insert x (sort xs)
```

The following is an example of how we intend `sort` to evaluate:

$\underline{\texttt{sort [4, 1, 9, 3]}}$

$= \texttt{insert 4 (}\underline{\texttt{sort [1, 9, 3]}}\texttt{)}$

$= \texttt{insert 4 (insert 1 (}\underline{\texttt{sort [9, 3]}}\texttt{))}$

$= \texttt{insert 4 (insert 1 (insert 9 (}\underline{\texttt{sort [3]}}\texttt{)))}$

$= \texttt{insert 4 (insert 1 (insert 9 (insert 3 (}\underline{\texttt{sort []}}\texttt{))))}$

$= \texttt{insert 4 (insert 1 (insert 9 (}\underline{\texttt{insert 3 []}}\texttt{)))}$

$= \texttt{insert 4 (insert 1 (}\underline{\texttt{insert 9 [3]}}\texttt{))}$

$= \texttt{insert 4 (}\underline{\texttt{insert 1 [3, 9]}}\texttt{)}$

$= \underline{\texttt{insert 4 [1, 3, 9]}}$

$= \texttt{[1, 3, 4, 9]}$

Specifying `insert`

`insert` will be defined later — this is 'top-down programming'. However, we must *specify* `insert` immediately.

We want to say three things about `insert` a xs. First, it contains the elements of xs, in the same order, with a inserted somewhere in the middle. Imagine that xs is prised apart as $xs = xs_1$`++`xs_2, and then a is inserted in the gap to give the result xs_1`++[`a`]++`xs_2. Next, we want to say that an a is inserted in the *correct* place in the middle — in other words, the result is sorted. Finally, when we use `insert` in `sort`, its second argument is always sorted and we expect this fact to make it easier to implement `insert`. This gives us a pre-condition:

```
insert :: * -> [*] -> [*]
||pre:  Sorted(xs)
||post: Sorted(ys) &
||      (E)x1s,x2s:[*]. (xs = x1s++x2s & ys = x1s++[a]++x2s)
||             where ys = (insert a xs)
```

`sort` is correctly implemented

That is to say, `sort` will work correctly provided that `insert` satisfies its specification. Of course, when we do get round to implementing `insert` it may have any number of errors in it and they will lead `sort` astray also, but that is not the point. We can regard `sort` now as correct and finished because our reasoning about it uses the specification of `insert`, not the implementation. The only thing that could thwart us is if we discover that the specification of `insert` as it stands cannot be implemented.

Let us now prove that `sort` is correct. First, and crucially, we have a recursion variant `#`xs for `sort` xs. As usual, this proves termination, at least when xs is finite (we could not expect that sorting an infinite list would terminate), and allows us to assume that the recursive calls all work correctly. The two alternatives in the definition cover all possible cases, so we must just check that they give correct answers.

Proposition 6.3 `sort` meets its specification.

Proof First we must check that `[]` is sorted and a permutation of `[]`. This is obvious.

Next we must check `sort` x:xs. Let ys = `insert` x (`sort` xs). We can assume that `sort` xs is sorted and a permutation of xs; we deduce in particular that the pre-condition of `insert` is satisfied. The post-condition of `insert` tells us that ys is sorted, as required, and it remains to show that ys is a permutation of x:xs. By the post-condition of `insert`, there are lists ys_1 and ys_2 such that

$$\begin{aligned} \texttt{sort}\ xs &= ys_1\texttt{++}ys_2 \\ ys &= ys_1\texttt{++[}x\texttt{]++}ys_2 \end{aligned}$$

Hence ys is a permutation of $x{:}ys_1$`++`$ys_2 = x{:}$(`sort` xs), which is a permutation of $x{:}xs$ because the recursive call worked correctly. □

Implementing `insert`

The idea in `insert` a xs is that we must move past all the elements of xs that are smaller than a (they will all come together at the start of xs) and put a in front of the rest. Hence there are two cases for `insert` a $(x:xs)$: the head is either a or x, according to which is bigger, and if a is bigger then it must be inserted into xs:

```
||insert was specified above
insert a [] = [a]
insert a (x:xs) = a:x:xs,            if a <= x
                = x:(insert a xs),   otherwise
```

for example,

```
insert 3 [1,4,9] = 1:(insert 3 [4,9]) = 1:3:4:[9] = [1,3,4,9]
```

`insert` is correctly implemented

The recursion variant for `insert` a xs is `#`xs. The three alternatives in the definition cover all possible cases, so we must just check that each one gives a satisfactory answer.

Proposition 6.4 `insert` meets its specification.

Proof For `insert` a `[]`: we must check that `[`a`]` is sorted (this is obvious), and that we can find lists xs_1 and xs_2 such that `[]` $=$ xs_1`++`xs_2 and `[`a`]` $=$ xs_1`++[`a`]++`xs_2. This is easy — take $xs_1 = xs_2 =$`[]`.

For `insert` a $(x:xs)$ when $x:xs$ is sorted and $a \leq x$, the result $a:x:xs$ is sorted by Proposition 6.2 — for `[`a`]++[`x`]`and `[`x`]++`xs are both sorted. To find xs_1 and xs_2 such that $x{:}xs = xs_1$`++`xs_2 and $a{:}x{:}xs = xs_1$`++[`a`]++`xs_2, we take $xs_1 =$`[]`and $xs_2 = x:xs$.

The final case is for `insert` $a(x{:}xs)$ when $x{:}xs$ is sorted (so xs is sorted and the pre-condition for `insert` is satisfied) and $a > x$; let $ys =$ `insert` a xs. By induction, ys is sorted and there are lists xs_1 and xs_2 such that $xs = xs_1$`++`xs_2 and $ys = xs_1$`++[`a`]++`xs_2. It follows immediately that $x{:}xs = (x{:}xs_1)$`++`xs_2, and the result, $x{:}ys$, is $(x{:}xs_1)$`++[`a`]++`xs_2.

Proposition 6.2 tells us that $x{:}ys$ is sorted. For either $xs_1 =$ `[]`, in which case $x{:}ys =$ `[`x`]++[`a`]++`xs_2 with both `[`x`]++[`a`]` and `[`a`]++`xs_2 (that is, ys) sorted, or $xs_1 \neq$ `[]`, in which case $x{:}ys =$ `[`x`]++`xs_1`++`$(a{:}xs_2)$ with both `[`x`]++`xs_1 (a sublist of $x{:}xs$) and xs_1`++`$(a{:}xs_2)$ (that is, ys) sorted. □

This completes the development of `sort` and `insert`.

6.6 Another example — sorted merge

In the preceding example, insertion sort, we introduced the predicates *Sorted* and *Perm*. These are very useful in their own right, and because (at least for *Perm*) a direct formalization into logic is difficult, we used an *axiomatic* approach starting from useful properties. The example in this section uses a similar method with another useful predicate, *Merge*.

$Merge(xs, ys, zs)$ means that the list zs is made up of xs and ys merged together. That is to say, the elements of xs and the elements of ys have been kept in the same order but interleaved to give zs. For instance,

$Merge(\text{‘}abcd\text{’}, \text{‘}123\text{’}, \text{‘}1ab2c3d\text{’})$	
$\neg Merge(\text{‘}abcd\text{’}, \text{‘}123\text{’}, \text{‘}1ba2c3d\text{’})$	a and b used in wrong order
$\neg Merge(\text{‘}abcd\text{’}, \text{‘}1234\text{’}, \text{‘}a1ab2c3d\text{’})$	a used twice, 4 not used
$Merge(\text{‘}abcd\text{’}, \text{‘}123\text{’}, \text{‘}ab12cd3\text{’})$	
$Merge(\text{‘}1abd\text{’}, \text{‘}2c3\text{’}, \text{‘}1ab2c3d\text{’})$	

We shall use the following properties:

1. $Merge(xs, ys, \texttt{[]})$ iff $xs = ys = \texttt{[]}$
2. $Merge(xs, ys, \texttt{[}z\texttt{]})$ iff $(xs = \texttt{[}z\texttt{]} \wedge ys = \texttt{[]}) \vee (xs = \texttt{[]} \wedge ys = \texttt{[}z\texttt{]})$
3. $Merge(xs, ys, zs_1 \texttt{++} zs_2)$ iff $\exists xs_1, xs_2, ys_1, ys_2.$
   $(xs = xs_1 \texttt{++} xs_2 \wedge ys = ys_1 \texttt{++} ys_2 \wedge Merge(xs_1, ys_1, zs_1) \wedge Merge(xs_2, ys_2, zs_2))$

Note that the right-to-left parts can be written more simply, as

1. $Merge(\texttt{[]}, \texttt{[]}, \texttt{[]})$
2. $Merge(\texttt{[}z\texttt{]}, \texttt{[]}, \texttt{[}z\texttt{]})$
   $Merge(\texttt{[]}, \texttt{[}z\texttt{]}, \texttt{[}z\texttt{]})$
3. $Merge(xs_1, ys_1, zs_1) \wedge Merge(xs_2, ys_2, zs_2) \rightarrow$
   $Merge(xs_1 \texttt{++} xs_2, ys_1 \texttt{++} ys_2, zs_1 \texttt{++} zs_2)$

If the left-to-right direction of (3) seems difficult to understand, think of xs_1 and ys_1 as the parts of xs and ys that go into zs_1, and xs_2 and ys_2 as the rest.

Let us now look at sorted merge. The idea is that if you have two sorted lists, then it is quite easy to merge them into a sorted result. Imagine merging two files by reading from the inputs and writing to the output. At each stage, the item to write is the smaller of the two front input items. The following is a Miranda version:

```
smerge :: [*] -> [*] -> [*]
||pre:  Sorted(xs) & Sorted(ys)
||post: Sorted(zs) & Merge(xs,ys,zs)
||        where zs = smerge xs ys
||recursion variant = #xs + #ys
smerge [] ys = ys
smerge (x:xs) [] = x:xs
smerge (x:xs) (y:ys) = x:(smerge xs (y:ys)),   if x <= y
                     = y:(smerge (x:xs) ys),   otherwise
```

It is easy enough to see that this works correctly in the first two cases. The fourth is just like the third, so we shall concentrate on that. We must show the following.

Suppose $x{:}xs$ and $y{:}ys$ are both sorted, and that $x \leq y$. Let $ws = ($`smerge` $xs\ (y:ys))$. The pre-conditions for this are satisfied (xs and $y{:}ys$ are both sorted), so we know that ws is sorted and that $Merge(xs,\ y{:}ys,\ ws)$. We must show that $Merge(x{:}xs,\ y{:}ys,\ x{:}ws)$ (this is almost immediate), and that $x{:}ws$ is sorted. The intuitive reason why $x{:}ws$ is sorted is easy enough to see; ws is sorted, and x is less than all the elements of ws — these are either from xs and are $\geq x$ because $x{:}xs$ is sorted, or they are from $y{:}ys$ and are bigger than x because y is the smallest and $x \leq y$. We could quite reasonably be satisfied with this argument, but let us also show it slightly more formally by going back to the definition of sortedness.

Suppose $x{:}ws = us$`++[`a,b`]++`vs. If $us =$ `[]` , then $x = a$ and $ws = b{:}vs$. Two possibilities arise because $Merge(xs,\ y{:}ys,\ b{:}vs)$, namely that b is either `hd` xs or y. If $b =$ `hd` xs, then $x{:}xs$, which is sorted, is `[]++[`x,b`]++` (`tl` xs) and so $x \leq b$ giving $a \leq b$. If $b = y$, then $x \leq b$ by assumption giving $a \leq b$. If us is non-empty, then $ws =$ (`tl` us)`++[`a,b`]++`vs, and so $a \leq b$ because ws is sorted.

The formal version, written in box notation, appears in Figure 6.3

6.7 List induction

The reasoning techniques using recursion variants are usually all we need for proving that functions satisfy their specifications, but for more general properties they may break down. This is particularly the case when we want to compare the results of different calls of the same function. The following is an example with a function to reverse a list.

reverse

The `reverse` function is defined as follows:

	1	$x \leq y$			
	2	$Merge(xs, y{:}ys,\ ws)$			
	3	ws is sorted $x{:}xs$ is sorted			assumptions
$\forall\mathcal{I}\ \ US, VS, A, B$	4				
	5	$x{:}ws = US$`++[`A,B`]++`VS			
	6	$US =$ `[]` $\vee\ US \neq$ `[]`			
	7	case 1 of $\vee\mathcal{E}$			
	8	$US =$ `[]`			
	9	$x = A$			
	10	$ws = B : VS$			
	11	$B =$ `hd` $xs \vee B = y$			def *Merge*
	12	$B =$ `hd` xs		$B = y$	
	13	$x{:}xs =$ `[]++[`x, B`]++tl` xs		$x \leq y$	assumed
	14	$x \leq B$	($x{:}xs$ sorted)	$A \leq B$	eqsub
	15	$A \leq B$	eqsub		
	16	$A \leq B$			$\vee\mathcal{E}(11)$
	17	case 2 of $\vee\mathcal{E}$			
	18	$US \neq$ `[]`			
	19	$ws =$ `tl` US`++[`A, B`]++`VS			
	20	$A \leq B$			(ws sorted)
	21	$A \leq B$			$\vee\mathcal{E}(6)$
	22	$x : ws = US$`++[`A, B`]++`$VS \rightarrow A \leq B$			$\rightarrow\mathcal{I}$
	23	$x : ws$ is sorted			$\forall\mathcal{I}$

Figure 6.3

```
reverse :: [*] -> [*]
||pre:  none
||post: reverse xs is the reverse of xs
||recursion variant for reverse xs is #xs
reverse []     = []
reverse (x:xs) = (reverse xs)++[x]
```

It is not clear how this function ought to be specified. But bearing in mind that the specification is supposed to say how we can make use of the function, and bearing also in mind our idea that `++` is more useful than *cons* in specifications because it does not prefer one end of the list to the other, let us try to elaborate the specification by giving some useful properties of the function:

- (`reverse []`) = `[]`
- (`reverse [`x`]`) = `[`x`]`
- (`reverse` (xs`++`ys)) = (`reverse` ys)`++`(`reverse` xs)

These are enough to force the given definition, for we must have

```
reverse (x:xs) = reverse ([x]++xs)
          = (reverse xs)++(reverse [x])
          = (reverse xs)++[x]
```

There still remains the question of whether the definition does indeed satisfy these stronger properties. The first two are straightforward from the definition, but the third is trickier. It is certainly not obvious whether the recursion variant method gives a proof.

The principle of list induction

What we shall use is a new principle, the *Principle of List Induction.* It is the exact analogue of simple mathematical induction, but applied to lists instead of natural numbers.

Recall that each natural number is either 0 or $N+1$ for some N, and so simple induction requires us to prove a property P in the *base case*, $P(0)$, and also in the other cases, $P(N+1)$. But that was not all. In the other cases the principle gave us a valuable free gift, the *induction hypothesis*, by allowing us to assume $P(N)$. Proving $P(N+1)$ from $P(N)$ was the *induction step.* Using boxes, an induction proof is shown in Figure 6.4 List induction is

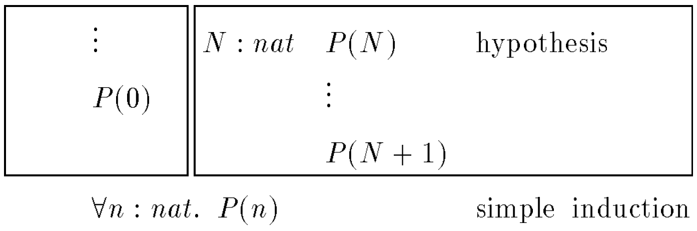

Figure 6.4

similar, but uses the fact that every list is either `[]` or x:xs for some x and xs. It says:

Let $P(xs)$ be a property of lists xs. To prove $\forall xs : \texttt{[}*\texttt{]}.\ P(xs)$, it is enough to prove:

base case: $P(\texttt{[]})$.
induction step: $P(x{:}xs)$ on the assumption of the *induction hypothesis*, $P(xs)$.

The box proof version of list induction appears in Figure 6.5.

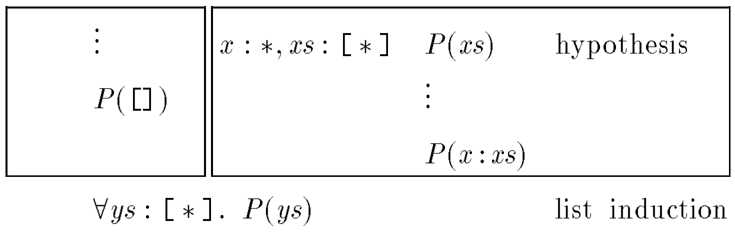
⋮
$P(\texttt{[]})$
$x : *, xs : \texttt{[}*\texttt{]}$
$P(xs)$
hypothesis
⋮
$P(x : xs)$
$\forall ys : \texttt{[}*\texttt{]}.\ P(ys)$
list induction

Figure 6.5

REMEMBER! All lists here are assumed to be finite. The induction principle will not tell you anything about infinite lists.

The principle can be justified in the same way as the principle of simple mathematical induction — if P does *not* hold for all lists xs, then what is a shortest possible list for which it fails? Surely not `[]`, if we have proved the base case; and if it is $x{:}xs$ then xs is shorter, so $P(xs)$ holds, and the induction step tells us that P also holds for $x{:}xs$ — a contradiction.

Alternatively, it can be justified using simple induction — see Exercise 17. However, more important than the justification is knowing how to use the principle.

Application to `reverse`

Proposition 6.5 Let xs and ys be lists. Then
$(\texttt{reverse}\ (xs\texttt{++}ys)) = (\texttt{reverse}\ ys)\texttt{++}(\texttt{reverse}\ xs)$

Proof We use list induction on xs to prove $\forall xs : \texttt{[}*\texttt{]}.\ P(xs)$, where

$$P(xs) \overset{\text{def}}{=} \forall ys : \texttt{[}*\texttt{]}.\ (\texttt{reverse}\ (xs\texttt{++}ys)) = (\texttt{reverse}\ ys)\texttt{++}(\texttt{reverse}\ xs)$$

base case: $xs = \texttt{[]}$

$$\begin{aligned}\text{LHS} &= (\texttt{reverse}\ (\texttt{[]++}ys)) = (\texttt{reverse}\ ys)\\ &= (\texttt{reverse}\ ys)\texttt{++[]} && \text{unit law}\\ &= (\texttt{reverse}\ ys)\texttt{++}(\texttt{reverse []}) = \text{RHS}\end{aligned}$$

induction step: Assume $P(xs)$; then in the equation for $P(x{:}xs)$:

$$\text{LHS} = \texttt{reverse}\ (x : xs\texttt{++}ys)$$