$$
\begin{array}{ll}
= (\texttt{reverse}\ (xs \texttt{++} ys)) \texttt{++} [x] & \text{definition} \\
= ((\texttt{reverse}\ ys) \texttt{++} (\texttt{reverse}\ xs)) \texttt{++} [x] & \text{induction} \\
= (\texttt{reverse}\ ys) \texttt{++} (\texttt{reverse}\ (x : xs)) & \text{definition} \\
= \text{RHS} &
\end{array}
$$

□

Note how although we have two lists to deal with, *xs* and *ys*, in this example we only need to use induction on one of them: *xs*. If you try to prove the result by induction on *ys*, you will find that the proof just does not come out.

To illustrate the advantage of using our stronger properties (Proposition 6.5) instead of just the definition, let us prove the intuitively obvious property that if you reverse a list twice you get the original one back. If you try to prove this directly from the definition, you will find that it is not so easy.

Proposition 6.6 Let *xs* be a list. Then (`reverse (reverse` *xs*)) = *xs*

Proof We use list induction on *xs*.

base case: *xs* = `[]` `reverse (reverse []) = (reverse []) = []`
induction step: When the list is not empty,

$$
\begin{array}{ll}
(\texttt{reverse}\ (\texttt{reverse}(x : xs))) & \\
\quad = (\texttt{reverse}\ ((\texttt{reverse}\ xs)\ \texttt{++}[x])) & \\
\quad = (\texttt{reverse}\ [x]) \texttt{++} (\texttt{reverse}\ (\texttt{reverse}\ xs)) & \\
\quad = [x] \texttt{++} xs & \text{by induction} \\
\quad = x : xs &
\end{array}
$$

□

6.8 Summary

- A list is a sequence of values, its *elements*, all of the same type. Lists are widely used in functional languages and are provided as a built-in type in Miranda in order to provide some convenient syntax for their use, for example, `[]` (the *empty* list), `[1,3,5,7]`.
- If *xs* is a list whose elements are of type `*`, then *xs* is of type `[*]`.
- The *append* operator `++` on lists puts two lists together. For example, `[1,2,3,4]++[5,6,7,8] = [1,2,3,4,5,6,7,8]`. It satisfies the laws

  $$
  \begin{array}{ll}
  xs \texttt{++} [] = [] \texttt{++} xs = xs & \text{unit laws} \\
  xs \texttt{++} (ys \texttt{++} zs) = (xs \texttt{++} ys) \texttt{++} zs & \text{associativity}
  \end{array}
  $$

  As a consequence of associativity, if you append together several lists, you do not need any parentheses to show in which order the appends are done.
- As long as a list *xs* is not empty, then its first element is called its *head*, `hd xs`, and its other elements form its *tail*, `tl` *xs* (another list). If

x is a value (of the right type) and xs a list, then `x:xs = [x]++xs` is a new list, 'cons of x and xs', whose head is x and whose tail is xs.

- Some other operators on lists are `#` (length) and `!` (for indexing).
- Every list can be expressed in terms of `[]` and `:` in *exactly one way*. Thus pattern matching can be performed on lists using `[]` and `:`. This makes `:` particularly useful in implementations, though `++` is usually more useful in specifications.
- The special form `[a..b]` denotes the list of numbers in increasing order from `a` to `b` inclusive.
- A list of characters (also called a *string*) can alternatively be denoted by using double quotation marks.
- For a recursively defined list function, the recursion variant is usually the length of some list.
- The principle of list induction says that to prove $\forall xs : [*].\ P(xs)$, it suffices to prove

  base case: $P([])$

  induction step: $\forall x : *.\ \forall xs : [*].\ (P(xs) \to P(x : xs))$

  This only works for finite lists.

6.9 Exercises

1. How would the evaluator respond to the expressions `[1]:[]` and `[]:[]`?
2. How would you use `#` and `!` to find the *last* element of a list?
3. Explain whether or not the expression `[8,'8']` is well-formed and if not why not.
4. Describe the difference between `'k'` and `"k"`.
5. Define a function `singleton` which given any list returns a Boolean indicating if the list has just one element or not. Write a function `has2items` to test if a list has exactly two items or not. Do not use guards or the built-in operator `#`.
6. Consider the following specification of the indexing function `!`:

   ```
   ||pre:  0 <= n < #xs
   ||post: (E)us,vs:[*]. (#us = n & xs = us++[x]++vs)
   ||          where x = xs!n
   ```

   (This is not quite right — the built-in `!` has a defensive specification.) Write a recursive definition of this function, and prove that it satisfies the specification.

   A straightforward way of writing specifications for list functions is often to use the indexing function and discuss the elements of the list. For instance, you could specify `++` by

```
||pre:  none
||post: #zs = #xs+#ys
||      & (A)n:nat. ((0 <= n < #xs -> zs!n = xs!n)
||            & (#xs <= n < #xs+#ys -> zs!n = ys!(n-#xs)))
||      where zs = xs++ys
```

Although this is straightforward, it has one disadvantage: when we appended the lists, we had to re-index their elements and it is not so terribly obvious that we did the calculations correctly.
For this reason, the specifications in this book avoid the 'indexing' approach for lists wherever possible, and this exercise shows that even indexing can be specified using `++` and `#`.

7. Write a definition of the function `count`:

```
count :: * -> [*] -> num
||pre:  none
||post: (count x ys) = number of occurrences of x in ys
```

For example (using strings), `count` *'o'* "*quick brown fox*" = 2.
The specification is only informal, but try to show informally that your definition satisfies it.

8. Consider the function `locate` of type `* -> [*] -> num`, `locate` x ys being the subscript in ys of the first occurrence of the element x, or `#`ys if x does not occur in ys. (In other words, it is the length of the largest initial sublist of ys that does not contain an x.) For instance,

   `locate` *'w'* "*the quick brown*" = 13

   Specify `locate` with pre- and post-conditions, write a Miranda definition for it, and prove that it satisfies its specification.
   If a character c is in a string s, then you should have

   $s\,!(\texttt{locate}\ c\ s) = c$

   Check this for some values of s and c.
9. Use box notation to write the proof of Proposition 6.1.
10. Specify and write the following functions for strings.

    (a) Use `count` to write a function `table` which produces a list of the the numbers of times each of the letters in the lowercase alphabet and space appear in a string:

    `table`"*a bad dog*" = $[2, 1, 0, 2, 0, 0, 1, 0, 0, 0, \ldots, 0, 2]$

    You may find it useful to define a constant containing the characters that you are counting:

```
alphabetsp = "abcdefghijklmnopqrstuvwxyz "
```

In writing this function you may find it helpful to define an auxiliary function which takes as an additional argument *as*. With the auxiliary function you can then step through the letters of the alphabet counting the number of times each letter appears in the string passed as an argument to `table`.

(b) Write a simple enciphering function, `cipher` that uses `locate`, `!` and `alphabetsp` to convert a character to a number, add a number to it, and convert it back to a character by indexing into `alphabetsp`. The type of `cipher` is then `num -> [char] -> [char]`. It should carry out this function on every character *separately* in the string it is given, to produce the encrypted string as its output.

`cipher` 2 "*quick brown fox*" = "*swkembdtqypbhqz*"
`cipher` (−2) "*swkembdtqypbhqz*" = "*quick brown fox*"

Use the function `table` on a string and the same string in enciphered form. What is the the relation between the two tables?

If you have a table generated from a large sample of typical English text how might you use this information to decipher an enciphered string. Can you think of a better enciphering method?

11. Consider the following Miranda definition:

```
scrub ::  * -> [*] -> [*]
scrub x [] = []
scrub x(y:ys) = scrub x ys,     if x=y
              = y:(scrub x ys), otherwise
```

(a) Write informal pre- and post-conditions for `scrub`.

(b) Use list induction on *ys* to prove that for all *x* and *ys*,

```
scrub x(scrub x ys) = scrub x ys
```

(c) Prove that for all *x*, *ys* and *zs*,

`scrub` *x* (*ys*`++`*zs*) = (`scrub` *x* *ys*)`++`(`scrub` *x* *zs*)

Now consider the following more formal specification for `scrub`:

```
||pre:  none
||post: ~isin(x,s)
||          /\ (E)xs:[*] ((A)y:* (isin(y,xs) -> y=x)
||                        /\ Merge(xs,s,ys))
||           where s=scrub x ys
```

(d) Show that the definition of `scrub` satisfies this.

(e) Show by induction on s that the specification specifies the result uniquely. (In fact, it specifies both ys and xs uniquely.)

(f) Use (e) to show (b) and (c) without induction.

12. Use the ideas of the preceding exercise to specify `count` more formally and prove that your definition satisfies the new specification.
13. Suppose `f :: [*] -> num` satisfies the following property:

$$\forall xs, ys : [*].\ f\ (xs \texttt{++} ys) = (f\ xs) + (f\ ys)$$

Prove that

$$\forall xs : [*].\ f\ (\texttt{reverse}\ xs) = f\ xs$$

14. Rewrite the proof for Proposition 6.5 using box notation.
15. Use induction on ws to show that if xs is sorted and can be written as $us\texttt{++}[a]\texttt{++}ws\texttt{++}[b]\texttt{++}vs$ then $a \leq b$. (The definition of sortedness is the special case when $ws = []$.)
16. In Proposition 6.2 it is proven that if $xs\texttt{++}ys$ and $ys\texttt{++}zs$ are both sorted, and ys is *non-empty*, then $xs\texttt{++}ys\texttt{++}zs$ is sorted. Rewrite this proof in box notation.
17. Suppose that you believe simple induction on natural numbers, but not list induction. Use the box notation to show how, if you have the ingredients of a proof by list induction of $\forall xs : [*].\ P(xs)$, you can adapt them to create a proof by simple induction of $\forall n : nat.\ Q(n)$ where

$$Q(n) \quad \stackrel{\text{def}}{=} \quad \forall xs : [*].\ (\#xs = n \rightarrow P(xs))$$

Show that (assuming, as usual, that all lists are finite) $\forall xs : [*].\ P(xs)$ and $\forall n : nat.\ Q(n)$ are equivalent.

18. Give specifications (pre-conditions and post-conditions) in logic for the following programs.

(a) `ascending :: [num] -> bool`; returns true if the list is ascending, false otherwise.

(b) `primes :: num -> [num]`; `primes n` returns a list of the primes up to `n`.

(c) `unique :: [num] -> bool`; returns true if the list has no duplicates, false otherwise.

Chapter 7

Types

7.1 Tuples

Recall three properties of lists of type `[*]` (for some type `*`):

1. They can be as long as you like.
2. All their elements must be of the same type, `*`.
3. They can be written using square brackets, `[-,-,...,-]`.

There is another way of treating sequences that relaxes (2) (you can include elements of different types) at the cost of restricting (1) (the length becomes a fixed part of the type). They are written using parentheses and are called *tuples*.

The simplest are the *2-tuples* (length 2), or *pairs*. For instance, `(1,9)`, `(9,1)` and `(6,6)` are three pairs of numbers. Their type is `(num, num)`, and their elements are called *components*. A *triple* (*3-tuple*) of numbers, such as `(1,2,3)`, has a different type, namely `(num, num, num)`.

Note that each of the types `(num,(num, num))`, `((num, num), num)` and `(num, num, num)` is distinct. The first is a pair whose second component is also a pair, the second is a pair whose first component is a pair, and the third is a triple. There is *no* concept of a one-tuple, so the use of parentheses for grouping does not conflict with their use in tuple formation. One advantage of the use of tuples is that if, for example, one accidentally writes a pair instead of a triple, then the strong typing discipline can pinpoint the error.

We can define functions over tuples by using pattern matching. For example, selector functions on pairs can be defined by:

```
fst :: (*, **) -> *
snd :: (*, **) -> **
fst (x,y) = x
snd (x,y) = y
```

Both `fst` and `snd` are polymorphic functions; they select the first and second components of any *pair* of values. Neither function works on any other tuple-type. Selector functions for other kinds of tuples have to be defined separately for each case.

The following is a function which takes and returns a tuple (the quotient and remainder of one number by another):

```
quotrem :: (num, num) -> (num, num)
quotrem (x,y) = (x div y, x mod y)
```

`quotrem` is defined to be a function of just one argument (a pair of numbers) and its definition is read as: `quotrem` takes a pair and returns a pair. Thus using tuples we can construct multiple arguments or results which are packaged up in the form of a *single* value. You can also mix the types of components, for instance the pair `(10, [True])` has type `(num, [bool])`.

The following is an example using lists. `zip` takes two lists — which should be of the same length — and 'zips' them together, making a single list of pairs. For instance,

```
zip [1,3,5] [2,4,6] = [(1,2),(3,4),(5,6)]
```

(It does not matter if `*` and `**` are two different types.)

```
zip :: [*] -> [**] -> [(*,**)]
||pre:  #xs = #ys  (for zip xs ys)
||post: difficult to make logical specification much
||      different from definition, but see Exercise 2
||recursion variant = #xs
zip [] [] = []      ||3 different types for [] here
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

(Note that the pre-condition ensures that there is no need to consider cases where one argument is empty and the other is not.)

To unzip a list, you want in effect *two* results — the two unzipped parts. So the actual (single) result can be these two paired together, for example,

```
unzip [(1,2),(3,4),(5,6)] = ([1,3,5],[2,4,6])
```

```
unzip :: [(*,**)] -> ([*],[**])
||pre:  none
||post: zip xs ys = ps
||         where (xs, ys) = unzip ps
||recursion variant = #ps.
unzip [] = ([],[])
unzip (x,y):ps = (x:xs,y:ys)
                    where (xs,ys) = unzip ps
```

This illustrates in two places how pattern matching can be used to give names to the components of a pair: first in `(x,y):ps`, to name the components of the head pair in the argument, and second in the `where` part for the components of the result of the recursive call.

7.2 More on pattern matching

Patterns in general are built from variables and constants, using constructors. For example,

```
x   5   (x,4,y)
```

are a variable, a constant and a triple built from two variables and a constant using the `(,,)` constructor for triples. The components of a structured pattern can themselves be arbitrary patterns, thus allowing nested structures of any depth. The constructors which can be used in patterns include those of tuple formation `(...,...)`, list formation `[...,...]`, and those of user-defined types (which we will see later in this chapter). In addition we have also seen the special facilities for pattern matching on lists and natural numbers. Patterns are very useful in the left-hand side of function definitions for two reasons:

1. They provide the right-hand side with names for subcomponents of the arguments.
2. They can serve as guards.

Pattern matching can also be combined with the use of guards:

```
last (x:xs) = x,         if xs = []
            = last xs,   otherwise
last []     = error "last of empty"
```

Patterns in the above definition are disjoint. In Miranda, patterns may also contain repeated variables. In such cases identical variables implicitly express the condition that their corresponding matched expressions must also be identical. For example,

```
equal :: * -> * -> bool
equal a a = True
equal a b = False
```

Such patterns match a value *only* when the parts of the value corresponding to the occurrences of the same repeated variable are equal.

Finally, patterns can be used *in conjunction with local definitions* — `where` parts, as in `unzip` to decompose compound structures or user-defined data types. In the following example if the value of the right-hand side matches the structure of the given pattern, the variables in the pattern are bound to the corresponding components of the value. This is useful since it enables the programmer to decompose structures and name its components:

```
...where

        [3,4,x,y] = [3,4,8,9]
        (a,b,c,a) = fred
        (quot,rem) = quotrem (14,3)
```

For the second definition to make sense the type of `fred` must be a 4-tuple. If the match fails anywhere, all the variables on the left will be undefined and an error message will result if you try to access those values in any way.

7.3 Currying

Now that you have seen pairs, it might occur to you that there are different ways of supplying the arguments to a multi-argument function. One is the way that you have seen repeatedly already, as in

```
cylinderV :: num -> num -> num
cylinderV h r = volume h (areaofcircle r)
```

Another is to pair up the arguments, into a single tuple argument, as in

```
cylinderV' :: (num, num) -> num
cylinderV' (h,r) = volume h (areaofcircle r)
```

You might think that the difference is trivial, but for Miranda they are quite different functions, with different types and different notation (the second must have its parentheses and comma).

To understand the difference properly, you must realize that the first type, `num -> num -> num`, is actually shorthand for `num -> (num -> num)`; `cylinderV` is really a function of one argument (`h`), and the result of applying it, `cylinderV h`, is *another function*, of type `num -> num`. `cylinderV h r` is another shorthand, this time for `(cylinderV h) r`, that is, the result of applying the *function* `cylinderV h` to an argument `r`.

This simple device for enabling multi-argument functions to be defined without the use of tuples is called *currying* (named in honour of the mathematician Haskell Curry). Therefore, multi-argument functions such as `cylinderV` are said to be *curried* functions. `cylinderV` is the curried version of `cylinderV'`.

Partial application

One advantage of currying is that it allows a simpler syntax by reducing the number of parentheses (and commas) needed when defining multi-argument functions. But the most important advantage of currying is that a curried function does not have to be applied to all of its arguments at once. Curried

functions can be *partially applied* yielding a function which requires fewer arguments.

For example, the expression `(cylinderV 7)` is a perfectly well-formed expression which is a partial application of the function `cylinderV`. This expression is an anonymous function (that is, a function without a name) which maps a number to another number. Once this expression is applied to some argument, say `r`, then a number is returned which is the volume of a cylinder of height 7 and base radius of `r`.

Partial application is extremely convenient since it enables the creation of new functions which are specializations of existing functions. For example, if we now require a function, `volume_cylinder100`, which computes the volume of a cylinder of height 100 when given the radius of the base, this function can be defined in the usual way:

```
volume_cylinder100 :: num -> num
volume_cylinder100 radius  =  cylinderV 100 radius
```

However, the same function can be written more concisely as

```
volume_cylinder100 =  cylinderV 100
```

or indeed we may not even define it as a separate function but just use the expression `(cylinderV 100)` in its place whenever needed.

Even more importantly, a partial application can also be used as an actual parameter to another function. This will become clear when we discuss higher-order functions in Chapter 8.

Order of association

For currying to work properly we require function application to 'associate to the left': for example, `smaller x y` means `(smaller x) y` not `smaller (x y)`.

Also, in order to reduce the number of parentheses required in type declarations the function type operator `->` associates to the *right*. Thus `num -> num -> num` means `num -> (num -> num)` and not `(num -> num) -> num`. You should by now be well used to omitting these parentheses, but as always, you should put them in any cases where you are in doubt.

Partial application of predefined operators

Any curried function can be partially applied, be it a user-defined function or a predefined operator or function. Similarly, primitive infix operators can also be partially applied. We have seen how parenthesized operators can be used just like ordinary prefix functions in expressions. This notational device is extended in Miranda to partial application by allowing an argument to be also enclosed along with the operator (see Figure 7.1). For example,

`(1/)`	is the	'reciprocal'	function
`(/2)`	'	'halving'	'
`(^3)`	'	'cubing'	'
`(+1)`	'	'successor'	'
`(!0)`	'	'head'	'

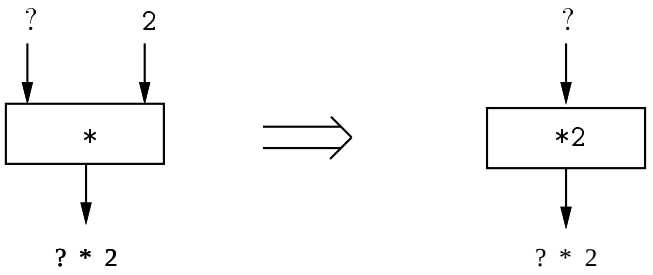


Figure 7.1

These forms can be regarded as the analogue of currying for infix operators. They are a minor syntactic convenience, since all the above functions can be explicitly defined. Note that there is one exception which applies to the use of the minus operator. `(-x)` is always interpreted by the evaluator as being an application of unary minus operator. Should the programmer want a function which subtracts x from numbers then a function must be defined explicitly.

More examples of such partial applications are given in Chapter 8, where simple higher-order functions are discussed.

7.4 Types

As we have seen from Chapter 4, expressions and their subexpressions all have types associated with them.

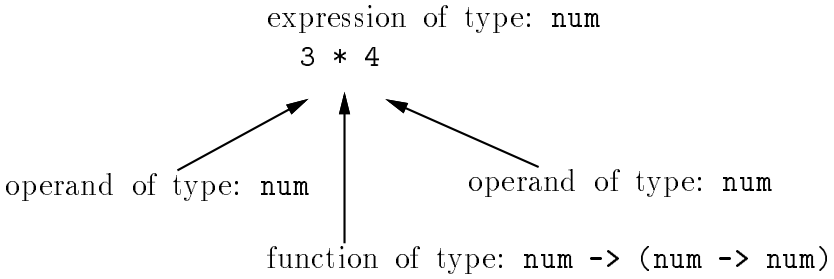


Figure 7.2

There are *basic* or primitive types (`num`, `bool` and `char`) whose values are built-into the evaluator. There are also *compound* types whose values are

constructed from those of other types. For example,

- tuples of types,
- function types (that is, from one given type to another),
- lists of a given type.

Each type has associated with it certain operations which are not meaningful for other types. For example, one cannot sensibly add a number to a list or concatenate two functions.

Strong typing

Functional languages are *strongly typed*, that is, every well-formed expression can be assigned a type that can be deduced from its subexpressions alone. Thus any expression which cannot be assigned a sensible type (that is, is not well-formed) has no value and is regarded as illegal and is rejected by Miranda before evaluation. Strong typing does not require the explicit type declaration of functions. The types can be inferred automatically by the evaluator.

There are two stages of analysis when a program is submitted for evaluation: first the *syntax analysis* picks up 'grammatical' errors such as `[1,)2(()];` and if there are no syntax errors then the *type analysis* checks that the expressions have sensible types, picking up errors such as `9 ++ True`. Before evaluation, the program or expression must pass both stages. A large number of programming errors are due to functions being applied to arguments of the wrong type. Thus one advantage of strong typing is that type errors can be trapped by the type checker prior to program execution. Strong typing also helps in the design of clear and well-structured programs. There are also advantages with respect to the efficiency of the implementation of the language. For example, because all expressions are strongly typed, the operator `+` knows at run-time that both its arguments are numeric it need not perform any run-time checks.

Type polymorphism

As we have already seen with a number of list functions, some functions have very general argument or result types. For example,

```
id  x = x
```

The function `id` maps every member of the source type to itself. Its type is therefore `* -> *` for some suitable type `*`. But `*` suits every type since the definition does not require any particular properties from the elements of `*`. Such general types are said to be *generic* or *polymorphic* (many-formed) types

and can be represented by *type variables*. In Miranda there is an alphabet of type variables, written `*`, `**`, `***`, etc., each of which stands for an arbitrary type. Therefore, `id` can be declared as follows:

```
id :: * -> *
```

Like other kinds of variables, a type variable can be instantiated to different types in different circumstances. The expression `(id 8)` is well-formed and has type `num` because `num` can be substituted for `*` in the type of `id`. Similarly, `(id double)` is well-formed and has type `num -> num`. Similarly, `(id id)` is well-formed and has type `* -> *` because the type `(* -> *)` can be substituted for `*`. Thus, again like other kinds of variables, type-variables are instantiated consistently throughout a single function application. The following are some more examples:

```
sillysix :: * -> num
sillysix x = 6
second :: * -> ** -> **
second x y = y
```

Notice that in a type expression all occurrences of the same type variable (for example, `**`) refer to *the same* unknown type at every occurrence.

Example — comparison operators

The comparison operators `=`, `<`, `<=`, and so on, are all polymorphic: the two values being compared must be of the same type, but it does not matter what that type is. Each operator has type `* -> * -> bool`.

Having said that, not all choices of `*` are equally sensible. `id :: * -> *` is polymorphic because it genuinely does not care what type its argument is — the algorithm is always the same. The comparisons, on the other hand, have to use different algorithms for different types (such polymorphism is often called *ad hoc*). The following are the *ad hoc* methods used.

- On `num`, the comparisons are numeric in the standard way.
- On `bool`, `False` $<$ `True`.
- On `char`, the comparisons are determined by the ASCII codes for characters. For instance, `'a'` $<$ `'p'` because 'a' comes before 'p' in the ASCII table.
- On list types `[*]`, comparisons use the lexicographic, or 'alphabetical' ordering. It does not work only with lists of type `[char]`. For instance, with lists of numbers the same idea tells you that

  $$[1] < [1,0] < [1,5] < [3] < [3,0]$$

- On tuple types, comparisons are similar. For instance, for pairs,

  $$(a,b) < (c,d) \text{ iff } (a < c) \vee ((a = c) \wedge (b < d))$$

- On function types, no comparisons are possible. (Consider, for example, the problems of computing `f` = `g`, that is, $\forall x$. `f` x = `g` x.)

Example — the empty list

As we have seen before, the empty list `[]` has type `[*]`. Being used in a particular expression may force `[]` to have more refined (specific) type. For instance, in `[[],[1]], []` must have type `[num]` to match that of `[1]`.

Type synonyms

Although it is a good idea to declare the type of all functions that we define, it is sometimes inconvenient, or at least uninformative, to spell out the types in terms of basic types. For such cases type synonyms can be used to give more meaningful names. For example,

```
name      ==  [char]
parents   ==  (name, name)
age       ==  num
weight    ==  num
date      ==  (num, [char], num)
```

A type synonym declaration does not introduce a new type, it simply attaches a name to a type expression. You can then use the synonym in place of the type expression wherever you want to. The special symbol `==` is used in the declaration of type synonyms; this avoids confusion with a value definition. Type synonyms can make type declaration of functions shorter and can help in understanding what the function does. For example,

```
databaseLookup :: name -> database -> parents
```

Type synonyms can *not* be recursive. Every synonym must be expressible in terms of existing types. In fact should the program contain type errors the type error messages will be expressed in terms of the names of the existing types and not the type synonyms.

Type synonyms can also be *generic* in that they can be parameterized by type variables. For example, consider the following type synonym declaration:

```
binop * == * -> * -> *
```

Thus `binop num` can be used as shorthand for `num -> num -> num`, for example,

```
smaller, cylinderV :: binop num
```

7.5 Enumerated types

We can define some simple types by explicit enumeration of their values (that is, explicitly *naming* every value). For example,

```
day       ::= Mon | Tue | Wed | Thu | Fri | Sat | Sun
direction ::= North | South | East | West
switch    ::= On | Off
bool      ::= False | True       ||predefined
```

Note that the names of these values all begin with upper case letters. This is a rule of Miranda. Values of enumerated type are ordered by the position in which they appear in the enumeration, for instance

```
Mon < Tue < ...
```

These are easily used with pattern matching. For instance, suppose a point on the plane is given by two Cartesian coordinates

```
point == (num, num)
```

A function to move a point in some direction can be defined by

```
move :: direction -> num -> point -> point
move North d (x,y) = (x,y+d)
move South d (x,y) = (x,y-d)
move East  d (x,y) = (x+d,y)
move West  d (x,y) = (x-d,y)
```

It is possible to code these values as numbers, and indeed in some programming languages that is the only option. However, this is prone to error, as the coding is completely artificial — there is no natural way of associating numerical values with (for example) days of the week, so you are at risk of forgetting whether day 1 was supposed to be Sunday or Monday. A single lapse will introduce errors into your program. With enumerated types you do not have to remember such coding details, and, also, the strong typing guards against meaningless errors such as trying to add together two days of the week.

7.6 User-defined constructors

Recall the idea of constructors — 'packaging together several values in a distinctive wrapper'. The main examples that you have seen so far have been *cons* and tupling, but there are ways of defining your own.

You have just seen the simplest examples! Each value (for example, `Mon, Tue, Wed`, etc.) in an enumerated type is a trivial, 'nullary' constructor that is nothing but the distinctive wrapper — no values packaged inside. (You may remember that the empty list could be considered like this.) It is also easy to define non-trivial constructors.

For example, we can define a new datatype `distance` to express the fact that distances may be measured by different units. The subsequent definition of `addDistances` is designed to eliminate the possibility of a programmer attempting to mix operations on distances of different kinds. Note again that constructor names in Miranda must start with upper case letters:

```
distance ::=  Mile num  |  Km num | NautMile num

addDistances :: distance -> distance -> distance
addDistances (Mile x) (Mile y) = Mile (x+y)
addDistances (Km x) (Km y) = Km (x+y)
addDistances (NautMile x) (NautMile y) = NautMile (x+y)
addDistances x y = error "different units of measurement!"
```

In this way it is guaranteed that adding distances of different measurement units (or attempting to multiply, divide or subtract two distances) is not performed accidentally. This is because the predefined arithmetic operators will not operate on any datatype other than `num`. Therefore, programmers are forced to think carefully about their intentions and are helped to avoid mistakes by the type checker. This style of programming is clearly much better than simply using `num`s to represent all three kinds of distance. The constructor functions (`Mile`, `Km` and `NautMile`) are essential in the datatype definitions, for otherwise there will be no way of, say, determining whether `6` has type `num` or `distance`.

Notice that the type `bool` need not be considered as primitive. It can be defined by two nullary constructors `True` and `False` (both of type `bool`). Similarly, one may argue that type `char` can also be defined using nullary constructors `Ascii0...Ascii127`. But characters, like numbers and lists, are more of a special case as they require a different, non-standard naming and printing convention.

Another example is that of *union types*. Suppose, for instance, you have mixed data, some numeric and some textual. You can use constructors to say what sort each item of data is, by

```
data ::= Numeric num | Text [char]
```

The following is an example with 2-argument constructors, representing a complex number by either Cartesian or polar coordinates:

```
complex ::= Cart num num | Polar num num

multiply :: complex -> complex -> complex
multiply (Cart u v) (Cart x y) = Cart (u*x - v*y) (u*y + v*x)
multiply (Polar r theta) (Polar s psi) = Polar (r*s) (theta+psi)
|| and two more cases for mixed coordinates
```

Finally, it is also possible to have polymorphic constructors. A standard example is pairing. Of course, this is already built into Miranda with its own special notation `(-,-)`, but just to illustrate the technique we can define it in a do-it-yourself way:

```
diypair * ** ::= Pair * **
```

For instance, `Pair 10 [True]` (our do-it-yourself version of `(10,[True])`) has type `diypair num [bool]`. A new type can have one or more constructors. Each constructor may have zero or more fields/arguments of any type at all (including the type of the object returned by the constructor). The constructor itself also has a type, usually a function type. So *Pair* has type `* -> ** -> (diypair * **)`.

The number of fields taken by a constructor is called its *arity*, hence a constructor of arity zero is called a *nullary* constructor. Constructors (like other values) can appear in lists, tuples and definitions. Just as with ordinary functions, constructor names must also be unique. Unlike ordinary functions, constructor names must begin with a capital letter. Constructors are notionally 'applied' just like ordinary functions. However, two key properties distinguish constructors from other functions:

1. They have no rules (that is, definitions) and their application cannot be further reduced.
2. Unlike ordinary functions they can appear as patterns on the left-hand side of definitions.

It is always possible to define 'selector' functions for picking the components of such data types, but in practice, like `fst` and `snd` for pairs, this is not necessary. Pattern matching can be used instead.

7.7 Recursively defined types

The greatest power comes from the ability to use recursion in a type definition. To illustrate the principle let us define do-it-yourself lists. These really are lists implemented in the same way as Miranda itself uses, but without the notational convenience of : and the square brackets. Instead, there are explicit constructors *Emptylist* and *Cons*, and for our do-it-yourself version of `[*]` we write

```
diylist * ::= Emptylist | Cons * (diylist *)
```

The 'recursive call' here (of `diylist *`) is really no more of a problem than

it would be in a function definition, as you should understand from your experience with lists.

The following is another do-it-yourself type, this time without polymorphism. It is for natural numbers:

```
diynat ::= Zero | Suc diynat
```

The idea is that every natural number is either (and in a unique way) `Zero` or 'the successor of' (one plus) another natural number, and can be represented uniquely as `Zero` with some number of `Suc`s applied to it. For instance, 5 is represented as

```
Suc (Suc (Suc (Suc (Suc Zero))))
```

It is no accident that the two examples given here are exactly the types for which you have seen induction principles: the induction is closely bound up with the recursion in the definition, and generalizes to other datatypes. We will explore this more carefully after looking at a datatype that does not just replicate standard Miranda.

Trees

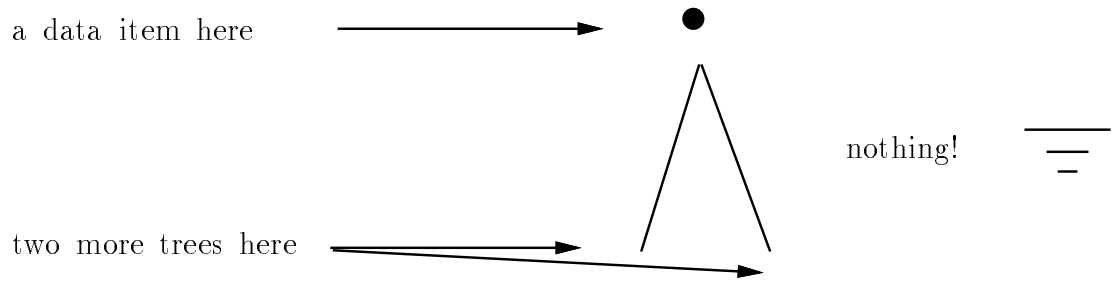


Figure 7.3

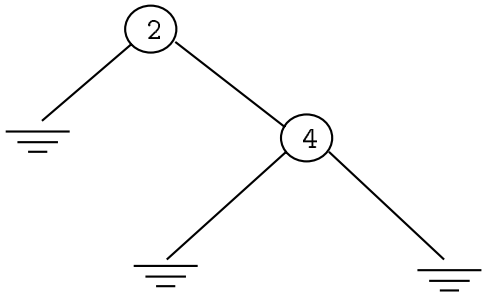
By 'tree' here, we mean some branching framework within which data can be stored. In its greatest generality, each *node* (branching point) can hold some data and have branches hanging off it (computer trees grow down!); and each branch will lead down to another node. Also, branches do not rejoin lower down — you never get a node that is at the *bottom* of two different branches. To refer to the tree as a whole you just refer to its top node, because all the rest can be accessed by following the branches down.

We are going to look at a particularly simple kind in which there are only two kinds of nodes:

- a 'tree' node has an item of data and two branches.
- a 'leaf' node has no data and no branches.

These will correspond to two constructors: the first, `Node`, packages together data and two trees and the second, `Emptytree`, packages together nothing:

`tree * ::= Emptytree | Node (tree *) * (tree *)`
where `*` is the type of the data items.




```
Node Emptytree 2 (Node Emptytree 4 Emptytree)
```

Figure 7.4

As an example (see Figure 7.4), let us look at ordered trees. Orderedness is defined as follows. First, `Emptytree` is ordered. Second, `Node` t_1 x t_2 is ordered iff

- t_1 and t_2 are both ordered;
- the node values in t_1 are all $\leq x$ (let us say 'x is an upper bound for t_1');
- the node values in t_2 are all $\geq x$ ('x is a lower bound for t_2').

Ordered trees are very useful as storage structures, storing data items (of type `*`) as the 'x' components of `Nodes`. This is because to check whether y is stored in `Node` t_1 x t_2, you do not have to search the whole tree. If $y = x$ then you have already found it; if $y < x$ you only need to check t_1; and if $y > x$ you check t_2.

Hence lookup is very quick, but there is a price: when you insert a new value, you must ensure that the updated tree is still ordered. The following is a function to do this. Notice that we have fallen far short of a formal logical account; there is a lot of English. But we have at least given a reasoned account of what we are trying to do and how we are doing it, so it can be considered fairly rigorous:

```
insertT :: * -> (tree *) -> (tree *)
||pre:  t is ordered
||post: insertT n  t is ordered, and its node
||      values are those of t together with n.

insertT n  Emptytree = Node Emptytree n Emptytree
insertT n (Node t1  x t2)
    = Node (insertT n t1) x t2,    if n <= x
    = Node t1  x (insertT n t2),   otherwise
```

Proposition 7.1 The definition of `insertT` satisfies its specification.

Proof If t is ordered, we must show that `insertT` n t then terminates giving a result that satisfies the post-condition. We shall use the usual 'circular reasoning' technique, but note that it remains to be justified because we have not given a recursion variant. We shall discuss this afterwards.

If t = `Emptytree` (which is ordered), then `insertT` n `Emptytree` terminates immediately, giving result `Node Emptytree` n `Emptytree`. This is ordered, and its node values are those of `Emptytree` (none) together with n, as required.

Now suppose that t = `Node` t_1 x t_2, and assume that the recursive calls work correctly. Since t is ordered, so, too, are t_1 and t_2, so the pre-conditions for the recursive calls hold. There are two cases, as follows:

Case 1 , $n \leq x$: `insertT` n t terminates, giving result r (say) = `Node` (`insertT` n t_1) x t_2. From the recursive post-condition, `insertT` n t_1 is ordered, and its node values are those of t_1 together with n. Hence the node values of r are those of t_1, n, x and those of t_2: that is, those of t together with n, as required.

Also, r is ordered, for the following reasons. `insertT` n t_1 and t_2 are both ordered, and x is a lower bound for t_2 because t is ordered.

x is also an upper bound for `insertT` n t_1 because the node values are those of t_1 (for which x is an upper bound because t is ordered) together with n (and $x \geq n$ because we are looking at that case).

Case 2 , $n > x$, is similar. □

As promised, we must justify the circular reasoning, and the obvious way is to find a recursion variant. We will show how to do this, but let us stress right away that the technique that we are actually going to recommend is slightly different, and that the calculation of a recursion variant is just to give you a feel for how it works.

The recursion variant technique is really a partial substitute for induction. It is not always applicable, but when it is applicable it is very convenient and smooth and the idea is to make it as streamlined as possible. What we shall see in a while is that you should try to think of *the tree itself* as a kind of recursion variant, 'decreasing' in the recursive calls from `Node` t_1 x t_2 to either