Chapter 2
Advanced concepts: Miranda

In this chapter the more sophisticated language features as found in modern lazy functional languages are discussed. Again Miranda (Turner, 1985) is taken as example language. The intention of this chapter is to show the programming power and notational elegance provided by such a state-of-the-art-language.

Many of the advanced concepts treated in this chapter can also be found in other languages. Those who are already familiar with similar languages like ML (Harper *et al.*, 1986), LML (Augustsson, 1984), HOPE (Burstall *et al.*, 1980), Haskell (Hudak *et al.*, 1992) or others will recognize most concepts.

Examples of simple function definitions using patterns and guarded equations have been shown in the previous chapter. In addition Miranda allows function definitions with a local scope to support a more modular programming style (see Section 2.1). Besides lists, tuples are also available as predefined data structures. Furthermore, a powerful and elegant notation based on mathematics is provided to generate all kinds of lists (Section 2.2). Miranda's type system has many features useful for programming. Functions can be of polymorphic type (Section 2.3). User-defined data structures (Section 2.4) can be declared using algebraic type declarations or abstract type declarations. The use of (higher order) functions as basic building blocks (Section 2.5) is a useful and reliable method for producing more complex functions. Due to lazy evaluation it is possible to write fairly general interactive programs in a functional language (Section 2.6). Finally, in Section 2.7 some guidelines are given on how to perform a top-down design of a functional program.

2.1 Defining functions

The basic ideas of Miranda are taken from earlier functional languages such as SASL (Turner, 1979b). The word ‘Miranda’ is Latin and it means ‘she who is to be admired’.

Miranda is a *lazy* functional programming language featuring the basic concepts discussed in the previous chapter. Miranda is equipped with many additional features such as a rich type system that helps the user to develop meaningful, elegant and compact programs.

A Miranda program consists of two parts: a collection of declarations, called the **script**, and an **initial expression** to be evaluated. The declarations in the script can be function definitions or type declarations. In general, a function definition has the following form:

f_name pattern$_1$ pattern$_2$ … pattern$_n$ = expression

Some examples of valid Miranda expressions:

x	|| an identifier
127	|| a value of type num(eral)
True	|| a value of type bool(ean)
'a'	|| a value of type char(acter)
"hello world"	|| a value of type [char]
y * x – 17	|| an expression of type num using predefined operators
[1,5,7,9]	|| a list of nums
[+, –, *, /]	|| a list of operators of type num -> num -> num
fac 3	|| a function application
(any_expression)	|| an expression

There are a variety of predefined infix and prefix operators, of various binding powers (see below).

List of prefix and infix operators, in order of increasing binding power. The table reveals that Miranda allows functions of polymorphic type (see Section 2.3). Comparison operators can be continued, e.g. 0 <= x < 5.

Left associative operators:	+ , – , * , / , div , mod , !
Right associative operators:	++ , – – , : , ^
Associative operators:	., \/ , &
Prefix operators:	~ , # , –
Continuable operators:	> , >= , = , ~= , <= , <

Operator	**Type**	**Meaning**
:	* -> [*] -> [*]	prefix an element to a list
++, – –	[*] -> [*] -> [*]	list concatenation, list subtraction
\/, &	bool -> bool -> bool	logical OR, AND

~	bool -> bool	logical negation
>, >=, =, ~=, <=, <	* -> * -> bool	greater, greater than, equal, not equal, less than, less
+, –	num -> num -> num	plus, minus
–	num -> num	unary minus
*, /, div, mod	num -> num -> num	times, divide, integer divide, remainder
^	num -> num -> num	to the power of
.	(** -> ***) -> (* -> **) -> * -> ***	function composition
#	[*] -> num	length of list
!	[*] -> num -> *	list subscription

The evaluation strategy in Miranda is lazy. Function alternatives are tried in textual order and the patterns in an alternative are tried from left to right. This strategy is intuitively easy to understand. However, it is not *truly lazy*, i.e. it does not always find a normal form if it exists.

Miranda's evaluation strategy is not truly lazy. Take the following definitions:

```
f 1 [ ]       =  0
f x (a : b)   =  a

inf    =  inf
```

With the initial expression f inf [1..2] the evaluation in Miranda will not terminate since the first argument will be evaluated first. So the strategy will not find the normal form 1. However, one would expect that a truly lazy strategy would evaluate the second argument of f first, but the strategy of Miranda first tries to match the patterns of the topmost rule from left to right and therefore it first evaluates the first argument of f.

Furthermore some specific aspects of Miranda are:

- Named constants in patterns are not possible: you can define a function linewidth = 80 for usage as a named constant throughout your program, but it is not possible to use this constant as a pattern on the left-hand side of a function definition: syntactically it would be a pattern variable.
- A string actually is a list of characters; hence "It is fun" is shorthand for 'I' : 't' : ' ' : 'i' : 's' : ' ' : 'f' : 'u' : 'n' : [ ]. This implies that one can use the pattern match mechanism for lists to decompose strings.

2.1.1 Local function definitions

Ordinary functions defined in Miranda have a **global** scope. This means that such a function can be applied in any expression anywhere in the program script. It is also possible to define functions with a **local** scope using a *where clause*. Such a **local** function is only defined in the smallest surrounding *block*. Outside a block such a function is not visible. In most programming languages the beginning and the end of a block structure are indicated by special keywords such as ‘begin’ or ‘end’, or in other languages a block is surrounded by parentheses. In Miranda a block is indicated by the *layout* of a program: a **block** consists of all expressions and definitions within one function definition or where clause that is either directly below or to the right of the first token after the equal sign of the function definition. A token that is outside the block of a definition is said to be **offside** (Landin, 1966).

The boxes indicate the blocks of, respectively, the function definition and the where clause:

```
glob_func a1 … an =  expr1
glob_func b1 … bn =  expr2
                     where
                     local_func1 c1 c2 … cm  =  expr3
                     local_func2 d1 d2 … ds  =  expr4
```

A **where clause** can contain any number of definitions. The where clause must be indented, to show that it is part of the right-hand side.

```
primes:: [num]
primes =  sieve (from 2)
          where
          sieve (p : x)     =  p : sieve (filter p x)
          from n            =  n : from (n + 1)
          filter p (a : x)  =  filter p x,     if a mod p = 0
                            =  a : filter p x, otherwise
```

A disadvantage of the offside rule is that it can lead to errors when names are changed globally in the script.

A not-offside program:

```
f a    =  g
          where g = 3
```

An offside program:

```
a_meaningful_long_name_for_this_function a = g
          where g = 3
```

Actually (for purists), it is a consequence of the offside rule that a Miranda program is no longer referentially transparent: a uniform substitution can introduce an error and hence change the meaning of a program. One can overcome this problem by using a layout that is *safe* with respect to global name changes.

An example of such a safe layout is given below:

```
f a
=  g
   where g = 3
```

2.2 Predefined data structures

In Miranda two kinds of data structure are predefined: *lists* and *tuples*. In the first part of this section tuples are discussed and compared with lists which were already introduced in the previous chapter.

Modern languages like Miranda and Haskell furthermore offer compact alternative notations to define lists: *list comprehensions*. These list comprehensions offer the possibility of defining lists in a style commonly used in mathematics. In the second part of this section these notations are discussed.

2.2.1 Tuples

The most important difference between lists and tuples is that the elements of a tuple can have *different* types while all the elements of a list must be of the *same* type. Furthermore a list is a linked list, possibly of *infinite* length. A tuple is just a record with some *finite* number of fields (Figure 2.1). There must be at least two fields in the tuple. Each field contains an object of a certain type.In Miranda, the notation for lists and tuples is quite similar: brackets are used to denote lists, parentheses are used to denote tuples. Parentheses are also used for disambiguating the structure of expressions. There is no concept of a 1-tuple, so the use of parentheses for disambiguating never conflicts with their use for tupling.

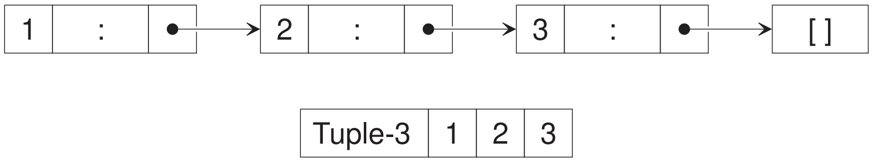

Figure 2.1 General difference between a list and a tuple.

The correspondence between the Miranda notation for lists and tuples:

```
[1, 2, 3]         || the list 1 : 2 : 3 : [ ]
(1, 2, 3)         || a tuple containing the elements 1, 2 and 3
```

In a tuple the elements can have *different* types. If the elements of a tuple are of type T_1, T_2 … T_n, the type of that tuple is indicated by (T_1, T_2, … , T_n) (see also Section 2.3).

Example of tuples and their types:

```
(3, 0, 'a')       || 3-tuple of type (num, num, char)
([0,1,2], +)      || 2-tuple of type ([num], num -> num -> num)
```

As with any other data structure, pattern matching can be used to access elements. In this way projection functions can be constructed that select a certain element from a tuple.

Using pattern matching to select an element of a tuple with three elements:

```
first (x, y, z)    =  x
second (x, y, z)   =  y
last (x, y, z)     =  z
```

Here follows a summary of the differences between the two predefined data structures lists and tuples:

- the length of lists can vary and can even be infinite, the length of tuples will always be fixed and finite;
- all the elements in a list must be of the same type, the elements in a tuple need not;
- lists are enclosed in square brackets, tuples are enclosed in round parentheses;
- in a pattern match and in a denotation all the elements in a tuple must be enumerated, whereas for lists this is not necessary;
- subscription as in lists is not defined in tuples; however there is the possibility of extracting elements of a tuple by using pattern matching.

As will be shown in Section 2.4, both lists and tuples could have been defined as ordinary user-defined algebraic data types. The reason that they have been predefined is that they are used so frequently that both data structures deserve a compact notation as well as special attention in an implementation.

2.2.2 List comprehensions

Miranda offers three alternative notations to define lists in a style commonly used in mathematics: by means of *dotdot expressions*, *ZF-expressions* and *recurrent generators*. These notations do not offer any additional expressive power, but are just syntactic sugar. The advantage of the notations is that they may be used to express certain algorithms concisely and elegantly (although the concept of recurrent generators is a bit overdone in our opinion). A disadvantage of the notations is that it may be less clear how the list will be computed. As a result programmers may sometimes find their programs to be unexpectedly slow or even non-terminating.

Dotdot expressions

The following abbreviations are provided for defining lists whose members form a finite or infinite arithmetic series.

Dotdot notations (let a, b, c stand for arbitrary numeric expressions):

[a..b]	finite list of numbers from a with interval 1, last member not exceeding b
[a..]	infinite list from a with interval 1
[a,b..c]	finite list from a with interval (b–a), last member not exceeding c
[a,b..]	infinite list from a with interval (b–a)

The following examples show the use of these abbreviations together with an equivalent definition showing that it is all just syntactical sugar.

Two auxiliary functions that are used in the expansion of the abbreviations:

```
inbetween:: num -> num -> num -> [num]
inbetween start limit interval
= start : inbetween (start + interval) limit interval,
              if (interval >= 0 & start <= limit) \/ (interval <= 0 & start >= limit)
= [ ],        otherwise

from:: num -> num -> [num]
from start interval   =  start : from (start + interval) interval
```

Examples of the use of dotdot expressions with (on the right) an equivalent ordinary definition.

The ciphers from 0 up to 9:

ciphers:: [num]	ciphers:: [num]
ciphers = [0..9]	ciphers = inbetween 0 9 1

The natural numbers:

nats:: [num] nats = [1..]	nats:: [num] nats = from 1 1

Even ciphers:

evenciphers:: [num] evenciphers = [0, 2..9]	evenciphers:: [num] evenciphers = inbetween 0 9 (2–0)

Even natural numbers:

evens:: [num] evens = [2, 4..]	evens:: [num] evens = from 2 (4–2)

ZF-expressions

ZF-expressions give a concise syntax for a general class of iterations over lists. The notation is adapted from the Zermelo–Fraenkel set theory (Fraenkel, 1922; Zermelo, 1908). Note that Miranda is dealing with lists and not with sets. A *ZF*-expression has two possible forms:

```
[ exp | qualifiers ]    or    [ exp // qualifiers ]
```

The above must be read as 'list of all exp such that qualifiers'. The expression yields a list with elements of the form dictated by exp. The elements are produced by qualifiers. When multiple qualifiers are specified, they are separated by semicolons. There are two kinds of **qualifiers**: *generators* and *filters*. A qualifier can be a **generator**, of the form:

pattern$_1$, pattern$_2$, … , pattern$_n$ <- exp

which is a shorthand for the following multiple generators:

pattern$_1$ <- exp; pattern$_2$ <- exp; … ; pattern$_n$ <- exp

The '<-' sign denotes that the patterns, of which the variables are locally introduced on its left, range over all the elements of the list on its right.

The function squares yields an infinite list containing (in order) the squares of all natural numbers. The *ZF*-expression is read aloud as 'list of all n*n such that n is drawn from the list [1..]'. The pattern n is just a local variable.

```
squares:: [num]
squares     =  [ n*n | n <- [1..] ]
```

The function sqlist employs a more complex pattern to yield a list of tuples. With the pattern (x, y) these tuples are generated from that list and used to yield the list of cubes, i.e. the list of all numbers to the power 3.

```
cubes:: [num]
cubes    =  [ x*y | (x, y) <- sqlist ]

sqlist:: [(num,num)]
sqlist     =  [ (n, n*n) | n <- [1..] ]
```

If there is more than one generator specified as qualifier, the rightmost will vary first, as if the generators were specified as nested for loops.

Example of the use of multiple generators:

```
list:: [(num, num)]
list     =  [ (x, y) | x, y <- [1..2] ]
```

is an abbreviation for

```
list     =  [ (x, y) | x <- [1..2]; y <- [1..2] ]
```

The rightmost generator varies first, so list yields the following list of tuples: [(1,1), (1,2), (2,1), (2,2)].

A qualifier can also be a **filter**, i.e. a Boolean expression used to impose restrictions on the objects generated by the generators.

The function factors gives the list of all factors of a given number n using a filter qualifier.

```
factors:: num -> [num]
factors n     =  [ r | r <- [1..n div 2]; n mod r = 0 ]
```

A nice demonstration of the elegance of *ZF*-expressions is given by the following definition of the quicksort algorithm:

```
quick_sort:: [num] -> [num]
quick_sort [ ]          =  [ ]
quick_sort (a : x)      =  quick_sort [b | b <- x ; b <= a] ++
                           [a] ++
                           quick_sort [b | b <- x ; b > a]
```

Since in the case of multiple generators the rightmost generator will vary first, other generators cannot produce new solutions if the rightmost generator yields an infinite sequence. Hence, one has to be aware of the possibility that a specification looks fine at first sight, but is not producing any result due to the way the computation is performed.

The Pythagorean numbers with multiple generators varying over infinite lists:

```
pyths:: [(num, num, num)]
pyths     =  [ (a, b, c) | a, b, c <- [1..]; a^2 + b^2 = c^2]
```

This 'solution' will not yield anything because of the order in which the generators vary: [(1, 1, 1), (1, 1, 2), (1, 1, 3), …]

The problem can be solved by **diagonalizing** generators such that all generators have equal priority, i.e. these generators are evaluated breadth-first from left to right instead of depth-first. This different type of generator is specified by using '//' instead of the '|'.

Example: the Pythagorean numbers with diagonalized generators:

```
pyths:: [(num, num, num)]
pyths     =  [ (a, b, c) // a, b, c <- [1..]; a^2 + b^2 = c^2]
```

Now the generators yield a list of tuples in the following way: [(1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 1, 2), (3, 1, 1), (1, 1, 3), (2, 2, 1), …].

Recurrent generators

Recurrent generators allow the construction of lists from arbitrary recurrence relations using the dotdot notation:

pattern <- exp_1, exp_2 ..

The meaning of a recurrent generator is expressed by expanding it to a normal generator in the following way:

pattern <-   iterate f exp_1
             where f pattern = exp_2

```
iterate:: (* -> *) -> * -> [*]
iterate f x   =  x : iterate f (f x)
```

The result of iterate is the infinite list [x, f x, f (f x), f (f (f x)), …]. The meaning of the type specification of iterate will be explained in the next section.

The recurrent generator combines the pattern and the second expression into a recurrent relation f. This f is iterated with start expression exp_1 yielding a list.

With a recurrent generator the Fibonacci series can be defined as:

```
fibs:: [num]
fibs   =  [ a | (a,b) <- (1,1), (b,a+b).. ]
```

This can be expanded to:

```
fibs   =  [ a | (a,b) <-    iterate f (1,1) ]
          where
          f (a,b)    =  (b, a+b)
```

The generator yields [(1,1), (1,2), (2,3), (3,5), (5,8),…]

Recurrent generators are special syntactical constructs for a special kind of recursive definition, generating a list.

2.3 The type system

Miranda is a **strongly typed** language, i.e. every expression and every subexpression has a type. The type system of Miranda is based upon the type assignment systems of Milner–Mycroft (Milner, 1978; Mycroft, 1984). Explicit type declarations are not required. If the types are not specified, they are inferred by the compiler by analysing the function definitions. Any inconsistency in the deduced types or any conflict with the specified types results in a compile-time error message. However, the explicit specification of types is highly recommended. It gives a lot of information about the specified functions. By choosing meaningful names for user-defined types (and for all other identifiers used in the program) the readability can be improved dramatically.

The advantages of adding a type system to a language are twofold. Firstly, it stimulates the programmer to write correct programs: only correctly typed programs are accepted by the compiler. Hence, for example, if the type of a function application is inconsistent with the type of the corresponding function definition, a compile-time error is generated. In this way the type system is of help in writing programs that make (more) sense.

Given the type of the infix + to be num -> num -> num, the expression

```
1 + 'a'
```

will be detected by the compiler as a type error.

A second advantage of having a typed language is that the type information can be used to generate efficient code. This is explained in Part 4.

There are also some disadvantages in having a typed language. To obtain the advantages claimed above, it is necessary to determine the consistency of types at compile-time. It can be proven that it is in general impossible to determine a proper type for *all* sensible expressions (see also Chapter 6). Fortunately, powerful type systems such as the Milner–Mycroft scheme used in Miranda can deduce and check the types of *most* expressions. As shown in this chapter later on, some expressions cannot be typed in this system and a compile-time error is generated even if the expression made sense. A peculiarity of the combination of the Milner type system (type *deduction* of a function that has not been typed) and the way the Mycroft system is used (*checking* a type specified by a programmer) is that in some cases expressions are accepted when they are explicitly typed by the programmer, but rejected otherwise (see also Section 2.3.6).

Hence, a type system always limits the number of legal programs. In most cases this limitation is of great help for the programmer, because an illegal program generally contains a program error. But in some more rare cases an illegal program does not contain a programming error at all. It simply could not be handled by the type system. In such cases the programmer is obliged to change the program to make it acceptable for the type system.

2.3.1 Basic types

In Miranda three basic types are predefined: num(bers), bool(eans) and char(acters). There is no type distinction between integer and floating-point numbers: they are both of type num. At run-time integer numbers are taken where possible. They are automatically converted into floating-point numbers when needed. It is important to note that due to the nature of floating-point numbers (rounding errors) this automatic conversion can create a conflict with referential transparency.

2.3.2 Type constructors

Complex types can be constructed by using *type constructors*. **Type constructors** make it possible to construct new types by combining existing types. The available type constructors are '[' and ']' (for constructing lists), '(' and ')' (for constructing tuples) and '->' (for constructing functions).

For these type constructors the following holds:

- If T is a type, then [T] denotes the type *list of* T.
- If T1,...,Tn are types, then (T1,...,Tn) denotes the type *tuple of* T1,...,Tn.
- If T1,T2 are types, then (T1 -> T2) denotes the type *function from* T1 *to* T2.

The best way to understand complex types built with the type constructors is to read them aloud, as in the following example.

Using type constructors to build complex types:

```
[ [num] ]                    || list of list of num
([char],[num] -> bool)       || tuple with list of char and
                             || function from list of num to bool
num -> num -> num            || function from num to
                             || function from num to num
```

2.3.3 Polymorphic types

A useful type system should not restrict the programmer too much. The example below illustrates this.

Function reverse:

```
reverse [ ]        =  [ ]
reverse (a : as)   =  reverse as ++ [a]
```

If the function reverse is applied to a list of characters, it will yield a list in which all characters are in reversed order. On the other hand, if applied to a list of numbers, it will yield this list in reversed order too. Hence it would be nice if the type system would allow the use of such a function for *any* sort of list. The Miranda system allows such a type specification. The type of reverse is denoted by

```
reverse:: [*] -> [*]
```

which indicates that reverse takes a list with elements of an *arbitrary* type as an argument and yields a list with elements of the *same* type.

To indicate arbitrary types there is an alphabet of **generic type variables** denoted by the symbols *, **, ***, and so on. A variable name can occur more than once in a type definition. In such a case each occurrence of the *same* name denotes the *same* type. Unequal names only indicate that the corresponding types can differ from each other (they could be the same). If a type variable occurs in the type definition of a function, the type of the function is said to be **polymorphic (or generic)** in the sense of Milner (1978).

Some examples of polymorphic functions:

```
ident:: * -> *
ident x  =  x
```

```
hd:: [*] -> *
hd (x : xs)  =  x

second:: (*, **, ***) -> **
second (x, y, z)  =  y
```

2.3.4 Type deduction

Generally a programmer will know the type of the function that he or she is defining. It is not always clear what the type of a concrete function is. For example, consider the function twice and try to deduce (infer) its type.

```
twice f x     =  f (f x)
```

In order to solve this problem the programmer actually has to perform the same kind of analysis that is performed by the type system incorporated in the Miranda compiler. This is only possible in all cases if the type system and the *type inference* mechanism are fully understood by the programmer. For the moment we shall only present a basic inference scheme that helps to derive the type of a function in most cases. Type systems and type inference mechanisms are explained in detail in Chapter 6.

The general idea of the inference scheme is the following: first the *left-hand side* of a function definition is analysed, making assumptions for the type of the function, its arguments and its result. The next step is to analyse this *right-hand side* using all assumptions already made. The type of the result must be compatible with the type of the right-hand side. Furthermore, function and arguments must be used consistently in the right-hand side. This analysis leads to *equations* that have to be solved to infer the type of the function.

Below the basic inference scheme is explained in more detail. For reasons of readability Greek characters are used for type variables instead of the *s that are used in Miranda.

1. Left-hand side analysis

For the general case f x_1 x_2 ... x_n = right-hand side for each argument and for the result a new type variable is introduced leading to the following assumptions:

x_1:: α_1, x_2:: α_2, ..., x_n:: α_n, result:: α_{n+1}
f:: α_1 -> α_2 ->... -> α_n -> α_{n+1}

If a pattern is specified, the corresponding type information is incorporated directly in the assumptions.

Incorporating pattern information in assumptions for left-hand side type analysis:

```
hd (x : y)    =  x
```

The constructor ‘:’ is a constructor of the type list, so the following types are assumed:

(x : y):: [α_1], x:: α_1, y:: [α_1], result:: α_2, hd:: [α_1] -> α_2.

2. Right-hand side analysis

The right-hand side is analysed using the assumptions made so far, leading to one or more equations. First of all, the assumed result type must be compatible with the type of the right-hand side as a whole. Furthermore, every application must be type-consistent. So for every occurrence of an application (function argument) on the right-hand side the required equation for the type of the function, the argument and the result of the application itself is:

function type = argument type -> result type.

Remember that all functions are considered to be curried. So for a function with multiple arguments several applications must be considered.

Already known types of functions are used in such a way that for each application of a polymorphic function in the assumed type fresh variables are taken. When no assumption is available new assumptions are made introducing fresh variables. Equations are constructed that specify the conditions that have to be fulfilled to make every application type consistent.

Right-hand side type analysis of previous example: hd (x : y) = x. Left-hand side assumptions were: (x : y):: [α_1], x:: α_1, y:: [α_1], result:: α_2, hd:: [α_1] -> α_2. Inspecting the right-hand side shows that the result type must equal the type of x (the right-hand side):

$\alpha_2 = \alpha_1$

This equation is solved in the next step.

3. Solving equations

The equations are solved by substituting for the type variables other type variables or actual types. In general there are many solutions for the equations. The solution we want is of course the most general solution. The process of finding such a solution is called **unification**, and the requested solution is called the **most general unifier.** In Chapter 6 a method will be given that performs the unification by solving the equations during the analysis.

> Solving equations for the running example: hd (x : y) = x. Assumptions: (x : y):: [α_1], x:: α_1, y:: [α_1], result:: α_2, hd:: [α_1] -> α_2. Equations: $\alpha_2 = \alpha_1$.
>
> Solution: substitute α_2 by α_1, leading to the following derived types:
>
> (x : y):: [α_1], x:: α_1, y:: [α_1], result:: α_1, hd:: [α_1] -> α_1
>
> We could also substitute α_1 by α_2 yielding the same solution, since the names of the type variables can be chosen freely. The compiler would have deduced the type hd:: [*] -> *. A different solution of the equation would have been found by substituting both α_2 and α_1 by num, leading to hd:: [num] -> num. Obviously, this is not the most general solution.

These three steps complete the basic type inference scheme. In this scheme not all syntactic constructs of Miranda, such as local definitions, type synonyms, algebraic data types, *ZF*-expressions etc., are taken into account. It is left to the reader to extend the scheme where necessary. To illustrate the basic type inference scheme it is applied to the following examples:

```
doublehd x     =  hd (hd x)
```

> *Left-hand side analysis*: x:: α_1, result:: α_2, doublehd:: α_1 -> α_2
>
> *Right-hand side analysis*: generally, it is easier to do the innermost application first.
> Analysis of application: hd x leads to: type of hd = (type of x)->(type of (hd x))
> Take fresh variables for the type of hd:: [α_4] -> α_4. The type of x is already assumed to be α_1. For hd x no assumption is available, so assume hd x:: α_3.
> The resulting equation: [α_4] -> α_4 = α_1 -> α_3
> The application hd (hd x) leads in the same way to the following equation:
> [α_5] -> α_5 = α_3 -> α_2
>
> *Solving the equations*: both sides of -> have to be equal:
> [α_4] = α_1, $\alpha_4 = \alpha_3$, [α_5] = α_3, $\alpha_5 = \alpha_2$.
> Leading to the following substitutions: α_1 = [[α_5]], $\alpha_2 = \alpha_5$.
> So doublehd:: [[α_5]] -> α_5.
> The compiler would have deduced the type of doublehd as [[*]] -> *.
>
> Another example: twice f x = f (f x)
>
> *Left-hand side analysis*: f:: α_1, x:: α_2, result:: α_3, twice:: α_1 -> α_2 -> α_3.

Right-hand side analysis:
Analysis of application: f x leads to: type of f = (type of x) -> (type of (f x))
For f x no assumption is available, so assume f x:: α_4.
The resulting equation: $\alpha_1 = \alpha_2 \text{ -> } \alpha_4$
Analysis of application: f (f x) leads to: (type of f) = (type of (f x)) -> (type of (f (f x)))
f (f x) is the right-hand side, so its type is assumed to be the result type: α_3.
The resulting equation: $\alpha_1 = \alpha_4 \text{ -> } \alpha_3$

Solving the equations leads to $\alpha_2 = \alpha_4 = \alpha_3$ and $\alpha_1 = \alpha_2 \text{ -> } \alpha_2$.
So the derived type is twice:: (α_2 -> α_2) -> α_2 -> α_2.
The compiler would have deduced the type of twice as (* -> *) -> * -> *.

2.3.5 Untypable function definitions

Not all functions can be typed in Milner's type system (see also Chapter 6). Two different kinds of untypable functions are given below:

A Milner untypable function:

```
selfapplication f = f f
```

This function cannot be typed in Milner's system because f cannot have both * and * -> * as type since the equation $\alpha_1 = \alpha_1 \text{ -> } \alpha_1$ has no solution.

The function definition in the example above cannot be typed. In this case one can argue that such functions should not be written. However, although the Milner type system is very powerful compared with the type systems generally available in more traditional languages, some very reasonable functions also cannot be expressed.

A Milner untypable function:

```
length:: [*] -> num
length [ ]        =  0
length (a : b)    =  1 + length b

fun:: ([*] -> num) -> [*] -> [*] -> num
fun f list1 list2     =  f list1 + f list2

missionimpossible  =  fun length [1,2,3] "abc"
```

The actual argument f is not allowed to have two different types within the body of the function fun. In this case f has to have type [num] -> num when applied to list1 but type [char] -> num when applied to list2.

Milner's type inference scheme inherently does not allow **internal polymorphism**, where the same object is used with different types within one function definition. Although the deduction scheme used can be improved for some cases, a satisfactory algorithm for general cases is not possible.

The problem mentioned above can be solved by making the polymorphism external:

```
fun2:: ([*] -> num) -> ([**] -> num) -> [*] -> [**] -> num
fun2 f1 f2 list1 list2  =  f1 list1 + f2 list2

missionpossible:: num
missionpossible  =  fun2 length length [1,2,3] "abc"
```

2.3.6 Differences between type deduction and checking

Explicit declaration of types makes it possible to declare the type of twice more restrictively as (num -> num) -> num -> num. In this way a type declaration can also be used to impose additional restrictions on a function (the declared type is more restrictive than the general derived type). Another difference between type deduction and type checking is due to the fact that the *type checking* algorithm uses an algorithm of Mycroft that differs from the *type deduction* algorithm of Milner (this is discussed in more detail in Chapter 6).

Difference between type inferencing and type checking:

```
g x = 1 : g (g 'c')
```

The type *deduction* algorithm of Milner cannot deduce the type of g. Hence an error message will be produced. In this case g is used polymorphically in the right-hand side of its own definition, with type instances: char -> [num] and [num] -> [num]. However, when the user *explicitly* specifies the type of g as g:: * -> [num] the type *checker* has no problems handling this case: the specified type is accepted and no error message is produced.

Maybe it would be better to insist always on explicit specification of types and to have a type inference tool assisting with this specification.

Another example with a difference between type checking and type deduction:

```
g  x  =  g  1
```

Type deduction yields g:: num -> * while type checking will accept g:: ** -> *.

2.3.7 Type synonym declaration

In Miranda it is possible to define a new name for a given type by means of a type synonym declaration.

Examples of type synonym declarations:

```
string    ==   [char]
matrix    ==   [ [num] ]
```

For obvious reasons (what should be substituted for what), recursive type synonyms are not permitted. Type synonyms are entirely transparent to the type system, and it is best to think of them as macros.

Using generic type variables, it is also possible to introduce a new name for a family of types.

Type synonym for a family of types:

```
array *   ==   [*]
```

In the examples above the predefined type [[num]] is renamed. It can be called matrix, but also array (array num). In both cases, an object of the newly defined type will be checked by the type system against the type [[num]], illustrating that indeed the type synonyms are entirely transparent (which is the only way to handle synonym types in the case that the type system must be able to deduce the type of the objects on its own).

Example of a type synonym declaration:

```
invert * **   ==   (* -> **) -> (** -> *)
string        ==   [char]
```

invert num string is then shorthand for (num -> [char]) -> ([char] -> num).

2.4 User-defined data structures and their types

Data structures are defined by defining new types. For this purpose it is possible to define (polymorphic) types using *algebraic type definitions* and *abstract type definitions*. Such a type specifies all legal occurrences of the user-defined data structure. With an *algebraic data type* a new data structure (like a record) can recursively be defined that is composed of existing data structures. Each record contains a discriminating *data constructor* (see also the section on lists in the previous chapter) and zero or more other objects called the **arguments of the data constructor**. The data constructor is an identifier (tag) that uniquely identifies an object to be of a specific type. *Abstract data types* make it possible to create data structures of which the actual implementations are hidden from the user. On such a data structure only the functions specified in the *abstract signature* can be applied. The concrete definition of these functions can be defined elsewhere.

2.4.1 Algebraic data types

A (recursive) **algebraic data type** definition enumerates the new data constructors with their arguments. Every alternative of an algebraic type definition begins with a unique discriminating data constructor.

Examples of algebraic type definitions:

```
day        ::=  Mon | Tue | Wed | Thu | Fri | Sat | Sun
numlist    ::=  Nillist | Numlist num numlist
numtree    ::=  Niltree | Node num numtree numtree
```

In these definitions there are predefined types (num), user-defined algebraic types (day, numlist, numtree) and user-defined data constructors (Mon, Tue, … , Sun, Nillist, Numlist, Niltree, Node). In Miranda, data constructors have to start with a capital letter.

The data constructors defined in an algebraic type can be used in function definitions. They appear in expressions to construct new objects of the specified algebraic type. They appear in patterns, for instance to discriminate between objects of the same algebraic type or to make projection functions for accessing the arguments of the data constructor. The influence of pattern matching on soundness criteria for type systems is discussed in Chapter 6. There it is shown that the use of algebraic data constructors as proposed here is indeed sound with respect to the type inference mechanism.

Constructing objects of an algebraic type:

```
payment_day:: day
payment_day     =  Fri

tree1:: numtree
tree1    =  Node 1 (Node 3 Niltree Niltree) Niltree
```

Using data constructors as patterns, the following function yields the mirror image of a numtree.

```
mirror:: numtree -> numtree
mirror Niltree        =  Niltree
mirror (Node n l r)   =  Node n (mirror r) (mirror l)
```

In a definition of an algebraic data type a new discriminating constructor is needed for every new type and for every new alternative. Consequently, it is impossible to define a general Nil. Every new type must have its own special constructor to denote an empty object of that type.

2.4.2 Polymorphic algebraic types

Algebraic types can also be **polymorphic**. One or more generic type variables can be used as parameters in the type definition.

Polymorphic algebraic type definition:

```
tree *    :: =   Niltree | Node * (tree *) (tree *)
```

This definition can be used for instance as follows:

```
numtree    = =   tree num
```

In this way a family of tree types can be defined (including tree num, tree bool, tree char, tree (num -> num), and so on). It is important to know that, in principle, all predefined types of Miranda can be defined using polymorphic algebraic type definitions. So in theory no built-in predefined types are needed. In practice, however, representing all predefined types algebraically would be rather inconvenient and inefficient.

Algebraic type definition for predefined types:

```
bool      :: =   True | False
char      :: =   'A' | 'B' | 'C' | …
[*]       = =    list *        :: =   Nil | Cons * (list *)
(*, **)   = =    tuple2 * **   :: =   Tuple2 * **
```

2.4.3 Abstract data types

An **abstract data type** is a *hidden* (algebraic or synonym) type for which functions are defined. These functions are the only way to create and access objects of that specific abstract type. The concrete definition of the abstract type and the concrete definitions of the functions, the **concrete signature**, are not visible to the outside world. Only the abstract type itself and the type of its functions, called the **abstract signature**, are visible.

Example: the well-known abstract data type stack with its signature.

```
abstype stack *                  || the abstract data type
with   empty      ::  stack *    || and its abstract signature
```

```
isempty    ::  stack * -> bool
push       ::  * -> stack * -> stack *
pop        ::  stack * -> stack *
top        ::  stack * -> *
```

The concrete definition of the abstract type and the functions defined on it are specified separately. The outside world cannot make use of the particular implementation chosen.

For example: the abstract data type stack can be implemented as a list. The outside world cannot make use of this fact: [ ] and empty are incomparable objects of two different and unrelated types, [*] and stack * respectively.

```
stack *      ==   [*]           || the concrete data type
empty        =  [ ]             || its implementation equations
isempty [ ]  =  True
isempty s    =  False
push e s     =  e : s
pop (e : s)  =  s
top (e : s)  =  e
```

The implementation equations do not have to appear immediately after the corresponding abstype declaration. They can occur anywhere in the script, but it is probably best to keep them nearby.

2.5 The use of functions as basic building blocks

In this section some important functions are given that, due to their general nature, are often used as basic building blocks in functional programs. The availability of higher order functions in combination with polymorphic types makes it possible to write very powerful functions that cannot be defined in most imperative languages that often use a more rigid type system. These powerful higher order functions can be used as basic building blocks for the creation of more sophisticated functions. Bird and Wadler extensively show the use of higher order functions as basic building blocks in the construction of functional programs and their correctness proofs (Bird and Wadler, 1988).

Function composition

There is a special operator to compose two functions. This associative operator is denoted as a single dot '.' and can be used in infix notation like the other operators in Miranda. The composition of two functions, say f and g, is a *function* comp f g, such that

```
comp f g x  =  f (g x)
```

or, using the infix operator:

```
(f . g) x  =  f (g x)
```

The operator '.' takes two functions as arguments and yields a new function: the composition of the two argument functions. The type of the composition operator is therefore (** -> ***) -> (* -> **) -> (* -> ***). The rightmost parentheses are not necessary, but they make it more clear that the composition of two functions yields a function as result. This type can be deduced as follows. Look at the function body in the definition of the composition operator: f (g x). Assume that x is of type *. Then g must be a function of type (* -> **). Hence f must be of type (** -> ***). The composition of f and g therefore yields a function of type (* -> ***).

Example of function composition. The function code is a predefined function in Miranda that returns the ASCII code of a character. The function decode is also predefined and returns the corresponding ASCII character when it is applied to a number between 0 and 256. The function nextchar is constructed by using function composition and yields the next character in the ASCII table when it is applied to a character with ASCII code less than 255.

```
code:: char -> num
decode:: num -> char

nextchar:: char -> char
nextchar    =  decode . (+ 1) . code
```

The predefined operator + is used in a special curried way here. The following expressions are all equivalent in Miranda: a + b, (+) a b, (a +) b, (+ b) a. The same holds for other predefined infix operators.

Evaluation of the function nextchar:

```
nextchar 'b'
→     ((decode . (+ 1)) . code) 'b'
→     (decode . (+ 1))(code 'b')        || according to the definition of '.'
→     decode ((+ 1) (code 'b'))         || according to the definition of '.'
→     decode ((+ 1) 98)
→     decode 99
→     'c'
```

Three alternative definitions of nextchar that are equivalent to the one above:

```
nextchar1 x   =  (decode . (+ 1) . code) x
nextchar2 x   =  decode ((+ 1) (code x))
nextchar3 x   =  decode (code x + 1)
```

The example above also illustrates that a sequence of functions can be combined as one new function using function composition. One can regard a composition of the form (f_n . f_{n-1} . … . f_1) x as a pipeline of functions applied to x, as illustrated in Figure 2.2.

Figure 2.2 A 'pipeline' of functions created with function composition.

Functions to manipulate lists

There are several handy functions that are frequently used to manipulate lists. In this section an overview is given of these functions and their use is illustrated with some examples.

The polymorphic function map is very powerful: it applies any function of appropriate type to all the elements of a list.

```
map:: (* -> **) -> [*] -> [**]
map f [ ]        =  [ ]
map f (x : xs)   =  f x : map f xs
```

For example, if

```
max:: num -> num -> num
max a b     =  a,  if a > b
            =  b,  otherwise
```

then

```
map (max 0) [–2, 3, –5]   →   …   →   [0, 3, 0]
```

The function filter has as arguments a predicate and a list. The result list will contain only those elements from the argument list that satisfy the predicate.

```
filter:: (* -> bool) -> [*] -> [*]
filter pred [ ]          =  [ ]
filter pred (x : xs)     =  x : filter pred xs,  if pred x
                         =  filter pred xs,      otherwise

filter (> 0) [2, –3, 5, 1, –2, –100]   →   …   →   [2, 5, 1]
```

The function take has as arguments a number, say n, and a list and yields the first n elements of that list.

```
take:: num -> [*] -> [*]
take n [ ]         =  [ ]
take n (x : xs)    =  x : take (n – 1) xs,   if n > 0
                   =  [ ],                   otherwise

take 2 [1, 2, 3, 4]   →   …   →   [1, 2]
take 3 "abcd"         →   …   →   "abc"
```

The function drop when applied to a number, say n, and a list of elements returns that list without the first n elements.

```
drop:: num -> [*] -> [*]
drop n [ ]         =  [ ]
drop n (x : xs)    =  drop (n – 1) xs,   if n > 0
                   =  x : xs,            otherwise

drop 2 ['a','b','c']    →   …       ['c']
```

The function takewhile has a predicate and a list as arguments. It takes elements from the front of the input list as long as they satisfy the predicate, and returns these elements in a list.

```
takewhile:: (* -> bool) -> [*] -> [*]
takewhile pred [ ]        =  [ ]
takewhile pred (x : xs)   =  x : takewhile pred xs,   if pred x
                          =  [ ],                     otherwise

takewhile (~= "stop") ["hello","world","stop","nice"]
→    …    →    ["hello","world"]
```

The function dropwhile is analogous to takewhile, but now the elements that satisfy the predicate are dropped. The rest of the list is the result of this function.

```
dropwhile:: (*->bool)-> [*] -> [*]
dropwhile pred [ ]          =  [ ]
dropwhile pred (x : xs)     =  dropwhile pred xs,   if pred x
                            =  x : xs,              otherwise

dropwhile (~= "stop") ["hello","world","stop","nice"]
→    …    →    ["stop","nice"]
```

The function zip2 has two lists as arguments and yields a list of pairs. The *n*th pair of the result list consists of the *n*th element of the first list together with the *n*th element of the second list. Analogous functions that 'zip' three, four, five or six lists together are also useful. The resulting list of tuples is as long as the shortest of the argument lists.

```
zip2:: [*] -> [**] -> [(*,**)]
zip2 (x : xs)   (y : ys)   =  (x,y) : zip2 xs ys
zip2 xs         ys         =  [ ]

zip2 [1..] "index"  →   …   →   [(1,'i'),(2,'n'),(3,'d'),(4,'e'),(5,'x')]
```

The function last yields the last element of a given list. This is the dual function of the function hd, which yields the head of a list.

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

last [1,2,3]        …   →   3
```

The function init is the dual function of the tail-function tl. It takes a list and yields a list of all elements of the input list without the last one.

```
init:: [*] -> [*]
init (x : xs)  =  [ ],          if xs = [ ]
               =  x : init xs,  otherwise
init [ ]       =  error "init of [ ]"

init ['g','r','a','v','e', 'l']   →   …   →   ['g','r','a','v','e']
```

A frequently used function is the function foldr. foldr is used to apply a function, say f, recursively in an accumulating way on all elements, say a_1, a_2, … a_n, of a list as follows:

$$f\ a_1\ (f\ a_2\ (\ \ldots\ (f\ a_n\ i)\ \ldots\ ))$$

The initial value i indicates the value to be substituted for the empty list.

```
foldr:: (* -> ** -> **) -> ** -> [*] -> **
foldr f i [ ]        =  i
foldr f i (x : xs)   =  f x (foldr f i xs)
```

Using foldr it is easy to give examples defining functions that calculate the product of all elements of a list, or their sum etc.

```
product  =  foldr (*) 1
sum      =  foldr (+) 0
```

```
and    = foldr (&) True
or     = foldr (\/) False
```

In these examples predefined operators are passed as arguments to the function foldr. To disambiguate this use of a predefined operator with its ordinary use the additional parentheses are needed.

```
product [1, 2, 3, 4]
→   foldr (*) 1 [1, 2, 3, 4]
→   (*) 1 (foldr (*) 1 [2, 3, 4])
→   (*) 1 ((*) 2 (foldr (*) 1 [3, 4]))
→   (*) 1 ((*) 2 ((*) 3 (foldr (*) 1 [4])))
→   (*) 1 ((*) 2 ((*) 3 ((*) 4 (foldr (*) 1 [ ]))))
→   (*) 1 ((*) 2 ((*) 3 ((*) 4 1)))
→   ...   →   24
```

On finite lists foldr is often used instead of recursion.

Example of the use of foldr

This example shows how to design an algorithm for computing the powerset of a given set of elements with the help of the function foldr.

Example of the function :

```
powerset [1,2,3]  =  [ [ ],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3] ]
```

The result list represents a set of elements, so it is not allowed to have more than one occurrence of an element. powerset [1,2,3] can be made out of powerset [2,3] by adding 1 to every element of the powerset [2,3] and concatenating it with powerset [2,3]:

```
powerset [2,3]    =  [ [ ],[2],[3],[2,3] ]
powerset [1,2,3]  =  [1 : [ ],1 : [2],1 : [3],1 : [2,3] ] ++ powerset [2,3]
```

Recursive definition of the function powerset:

```
powerset:: [*] -> [ [*] ]
powerset [ ]         =  [ [ ] ]
powerset (x : xs)    =  map (x :) (powerset xs) ++ powerset xs
```

The above example contains a halting criterion: powerset [ ] = [[ ]]. The definition also clearly reflects the above described method for making the powerset of a list (x : xs) out of the powerset of the list xs.

The definition of foldr has three formal arguments. These arguments are in the following way related to aspects of a recursive definition. The halting criterion, in the definition of foldr called i (derived from init), can be defined as [[ ]], like the halting criterion in the recursive definition. The relation between the powerset of list (x : xs) and the list xs can be represented by the function f from the parameter list of foldr. Define f as:

```
f:: * -> [ [*] ] -> [ [*] ]
f x powersofar     =  map (x :) powersofar ++ powersofar
```

powersofar stands for the result of the powerset of the list without the new element x.

The new definition of the function powerset using the function foldr:

```
powerset:: [*] -> [ [*] ]
powerset xs    =  foldr f [ [ ] ] xs
```

So combining the halting criterion with a constructive inductive recursion step leads to a short definition with foldr.

2.6 Interactive programs

Requiring that a functional program is interactive at first sight seems to be in contrast with referential transparency. An interactive system does not always react in the same way on the same input value. Take for example a program that repeatedly gets a value as input and outputs the sum of *all* the values so far. Such a program reacts differently when it gets the same value two times in a row. However, it always reacts in the same way on the whole input so far. This observation leads to the conclusion that interactive programs can be written in a functional language by considering the input as an argument that is a conceptually infinite list of values (called a **stream**). The output is also a stream: the result of the program. Owing to lazy evaluation such infinite lists can be handled. On the input side this means that not all the input has to be given at once. On the output side it means that those parts of the result that are already available can be printed. Together this allows interactive programs to be written.

Combining both input and output it is possible to write a program that gives a prompt (part of the result is already evaluated), reads a value (the head of an infinite input list is needed for the evaluation), responds and prompts again. One has to take care that the rest of the input list is passed as a parameter to functions that require input and furthermore one has to take care that no cyclic dependencies are required.

One way to create an interactive functional program in a lazy language:

```
function:: * -> [char] -> [char]
function state input  =  response ++ function newstate restinput
                         where
                         response   =  …
                         newstate   =  …
                         restinput  =  …
```

Generally some kind of state is adapted due to the input. Furthermore, one has to be able to indicate that a certain argument is actually the input of the program. In Miranda the input of the program can be indicated in the initial expression by specifying $– as an argument. So a typical call of the function above would be:

```
function initialstate $–
```

in which initialstate is a function that produces the initial state.

A program that repeatedly checks an input line for palindromicity (a word or sentence is a **palindrome** if it is the same as its reverse, e.g. 'able was I ere I saw elba' is a palindrome). The basic function is rather simple using the previously defined function reverse.

```
palindrome:: [char] -> bool
palindrome   chars   =    chars = reverse chars
```

Extending the program for interactive use requires some extra definitions in which the interaction with the user is defined. The program will prompt for an input line, respond and stop when an empty line is given as input:

```
pal_io:: [char] -> [char]
pal_io input   =  prompt ++ pal_react input

prompt:: [char]
prompt  =  "Type in a line, please?\n"

pal_react:: [char] -> [char]
pal_react input   =  "Have a good day.\n",           if thisline = ""
                  =  response ++ pal_io restlines,   otherwise
                     where
                     response   =  "Is a palindrome!\n", if palindrome thisline
                                =  "Is not a palindrome.\n", otherwise
                     (thisline, restlines)  =  splitline input
```

```
splitline:: [char] -> ([char], [char])
splitline chars     =  (line, restlines)
                       where
                       line        =  takewhile (~= '\n') chars
                       restlines   =  tl (dropwhile (~= '\n') chars)
```

The function splitline is defined using the basic functions tl, takewhile and dropwhile defined previously. Note that this is not a very efficient program: it would have been more efficient to define a new function that combined the definitions of takewhile and dropwhile producing both results together. The program is called with the following initial expression:

```
pal_io $–
```

In Koopman (1987) it is shown how using this method one can write strongly interactive programs like an editor. The basic structure of, for example, such an editor is given below:

```
commandinterpreter:: [char] -> [char] -> [char]
commandinterpreter text commands
   =  response ++ prompt ++ commandinterpreter newtext nextcommands
      where
      (commandline, rest)    =  splitline commands
      editoperation          =  parse commandline
      (response, newtext, nextcommands) =  editoperation text rest
      prompt        =  ...
      splitline x   =  ...
      parse x       =  ...
```

This command interpreter parses a command line and produces directly a function (not some encoding for a function) that performs the required operation.

A disadvantage of the given method for writing interactive functional programs is that every part of the output has to be part of the result of the whole program on the top level. As a consequence it is not a trivial task to change a program in such a way that for a specific function that is used somewhere in an expression the value of its arguments is printed every time the function is called. A change like that, which can be of interest for tracing or debugging, requires a change of the structure of the whole program.

Furthermore, it does not seem to be very practical to program file handling and menu and mouse-driven applications with the facilities that are provided by Miranda. In the language Concurrent Clean a different method for writing interactive programs is given that offers the required practical facilities for such applications (see Chapter 8).

2.7 Top-down functional programming

There is already much knowledge about programming in imperative programming languages. The two most important programming styles used are the *top-down* method and the *bottom-up* method.

Generally, when a large program is developed, several layers of abstraction are created. Each layer of abstraction is based on layers on a lower level of abstraction. Each layer has its own locally defined data structures with operations on them and performs its own task.

In a **bottom-up** style one starts with the lowest layer upwards to the next layer and finally one defines the top layer. In this style one designs data structures and the corresponding operations on them that are made available for the next layer. In a **top-down** style it is exactly the other way around. The problems left in each layer are solved in the next layer. In this way one refines the problems step-wise down to the basic level. Generally both methods are mixed when large programs are developed. One could say that with the bottom-up approach one designs the programming tools needed on a certain level of abstraction, whereas with the top-down approach one decides how these tools are being used.

These programming styles, commonly used to develop imperative programs, can also be used for the development of functional programs. For a bottom-up style of programming one can use, for instance, abstract data types. An example of their use was given in Section 2.4.3. The kind of data structures and operations will strongly depend on the kind of problem one has to solve. Top-down programming is based on **step-wise refinement**, which means that a problem is divided into subproblems if there is no simple way to write down the solution directly with the help of basic language constructs. In functional programming languages there is an excellent way to glue those partial solutions together, for solutions will consist of function definitions, and functions can be composed using function composition or function application. And of course, there is no danger of side-effects. Furthermore the strong typing provides an extra test of whether the functions are composed correctly.

2.7.1 A check-list for a top-down design

The check-list in Table 2.1 may be useful for designing functional programs. The check-list consists of seven parts that reflect the common phases of a top-down program design. Each phase consists of issues to think about when designing a program. It is not strictly necessary to work out all the issues nor to do them in the given order: experienced programmers can merge some of these steps or do them unconsciously or implicitly.

Table 2.1 A check-list for a top-down design.

1 Analysis

1.1 Work out some examples to get an understanding of the problem.
1.2 Write down the essential aspects of the given problem.
1.3 Give the relation between these aspects and the desired result.
1.4 Work out concrete examples.
1.5 Try to find general constraints and exceptions.
1.6 Find out whether there are special constraints for the solution.

2 Specification

2.1 Give the type of the arguments of the function(s).
2.2 Give the type of the result of the function(s).
2.3 Give the type of the function(s).
2.4 Look for inconsistencies and missing specifications in the problem.
2.5 Formulate the specification with the help of mathematics or a graphical representation.

3 Design

3.1 Make the data structure that will be used explicit.
3.2 Formulate a global algorithm that will solve the problem.
3.3 Decide whether the solution can be given directly with the help of language elements and if not
3.4 Divide the problem into subproblems.
3.5 Apply the method of this check-list to each of the subproblems.
3.6 Make alternative designs and choose the best one to implement.

4 Implementation

4.1 Work out the designed algorithm (including the data structures and the operations on them) into a Miranda script with comments, significant names and a structured layout.
4.2 Type the script into the computer.

5 Test and debug

5.1 Design critical test values to test all designed functions.
5.2 Test functions that solve subproblems before testing the total solution.
5.3 Debug functions if necessary.

6 Documentation

6.1 Check comments and names on consistency with (revised) functions.

7 Reflection

7.1 Note why decisions were made, and check whether this is documented.
7.2 Summarize what you have learned about designing an algorithm for the given problem.

In addition to this check-list it is useful to pay attention to the following points:

- use significant names everywhere;
- include meaningful comments where needed;
- always write down the type of a function explicitly;
- do not combine too many function compositions as this may make a program unreadable;
- use pattern matching where possible.

Example of a top-down design

Take again the roman number problem from the previous chapter. A closer look shows that the designed function romtonum is not very robust. It assumes that its argument will always be a correct roman number. With the help of the check-list below a function will be designed that decides whether a list of characters represents a correct roman number. Not all the points of the check-list are worked out because most of the points are trivial for experienced programmers (in an imperative language) and they do not differ much from the way one could handle them in the design of an imperative language. Names of subproblems that are not solved yet are printed in italic.

1 Analysis

1.1 Work out some examples to get an understanding of the problem.

```
MDCCCLXXXVIII  =  1888
MCDXCII        =  1492
```

1.2 Write down the essential aspects of the given problem.

It is important to know what exactly a roman number is. From the description of the problem one can find that:

- Roman ciphers consist of the following characters: M, D, C, L, X, V and I.
- These characters have values, respectively: 1000, 500, 100, 50, 10, 5 and 1.
- The characters in the roman numbers are always sorted from the highest value to the lowest from left to right, with the exception that some combinations of characters are abbreviated:

```
DCCCC  =  CM      LXXXX  =  XC      VIIII  =  IX
CCCC   =  CD      XXXX   =  XL      IIII   =  IV
```

1.3 Give the relation between these aspects and the desired result.

Given: Description of roman numbers.

Desired result: A function that decides whether a list of characters is a roman number.

1.4 Work out concrete examples.

```
MCDXCII      =  MCCCCLXXXXII
```

1.5 Try to find general constraints and exceptions.
It is not possible to denote negative numbers or zero.

2 Specification

2.3 Give the type of the function(s).

Let us call the desired function `isroman`; it will be of type `[char] -> bool`.

2.4 Look for inconsistencies and missing specifications in the problem.

According to the description of the problem, IVI is a correct roman number (5). However, this is not true. An extra constraint on Roman numbers has to be added, namely that roman numbers are always written in the shortest possible way.

2.5 Formulate the specification with the help of mathematics or a graphical representation.

isroman romnumber → true <=>

romnumber is a non-empty character sequence that obeys the syntax of roman numbers. The syntax of roman numbers is as follows (a BNF-like notation (Backus–Naur form) is used; literals are put between quotes):

```
Roman-number  =  Thousands;
Thousands     =  {'M'}  Fivehundreds;
Fivehundreds  =  'CM'  Fifties   |  'D'   Hundreds
              |  'CD'  Fifties   |  Hundreds;
Hundreds      =  ['C'] ['C'] ['C']  Fifties;
Fifties       =  'XC'  Fives     |  'L'   Tens
              |  'XL'  Fives     |  Tens;
Tens          =  ['X'] ['X'] ['X']  Fives;
Fives         =  'IX'            |  'V'   Ones
              |  'IV'            |  Ones;
Ones          =  ['I'] ['I'] ['I'];
```

3 Design

3.1 Make the data structure that will be used explicit.

```
rom_num  = =  [char]
```

3.2 Formulate a global algorithm that will solve the problem.

Apart from a global test for non-emptiness for each syntax rule there is a function that handles that rule.

3.4 Divide the problem into subproblems.

3.5 Apply the method of this check-list to each of the subproblems.

```
rom_num = = [char]

isroman:: rom_num -> bool
isroman    [ ]        =  False
isroman    roms       =  syntax_check roms

syntax_check:: rom_num -> bool
syntax_check roms     =  thousands roms

thousands:: rom_num -> bool
thousands  ('M' : rest)  =  thousands rest
thousands  roms          =  fivehundreds roms

fivehundreds:: rom_num -> bool
fivehundreds  ('C' : 'M' : rest)   =  fifties rest
fivehundreds  ('D' : rest)         =  hundreds rest
fivehundreds  ('C' : 'D' : rest)   =  fifties rest
fivehundreds  roms                 =  hundreds roms

hundreds:: rom_num -> bool
hundreds   ('C' : 'C' : 'C' : rest)  =  fifties rest
hundreds   ('C' : 'C' : rest)        =  fifties rest
hundreds   ('C' : rest)              =  fifties rest
hundreds   roms                      =  fifties roms

fifties:: rom_num -> bool
fifties    ('X' : 'C' : rest)    =  fives rest
fifties    ('L' : rest)          =  tens rest
fifties    ('X' : 'L' : rest)    =  fives rest
fifties    roms                  =  tens roms

tens:: rom_num -> bool
tens  ('X' : 'X' : 'X' : rest)  =  fives rest
tens  ('X' : 'X' : rest)        =  fives rest
tens  ('X' : rest)              =  fives rest
tens  roms                      =  fives roms
```

```
fives:: rom_num -> bool
fives  ['I', 'X']       =  True
fives  ('V' : rest)  =  ones rest
fives  ['I', 'V']       =  True
fives  roms          =  ones roms

ones:: rom_num -> bool
ones      ['I', 'I', 'I']    =  True
ones      ['I', 'I']         =  True
ones      ['I']              =  True
ones      [ ]                =  True         ||  all possibilities have been checked:
ones      roms               =  False        ||  if anything is left, it must be incorrect
```

3.6 Make alternative designs and choose the best one to implement.

It is much more efficient to combine the functions for checking and for calculating the value described in this chapter and in the previous chapter into one function that runs through the character sequence only once. So the design (and in fact even the task definition) is changed accordingly.

4 Implementation

4.1 Implement the designed algorithm (including the data structures and their operations) into a Miranda script with comments, significant names and a structured layout.

```
||  An efficient algorithm that computes the value of a proper roman number.
||  Error messages will be given for incorrect arguments.
||  Roman numbers are represented as character sequences
||  rom_num = = [char]
||  The main function is    rom_to_num:: rom_num -> num

rom_num   = =    [char]

rom_to_num:: rom_num -> num
rom_to_num   [ ]          =  error "an empty sequence is not a roman number"
rom_to_num   roms         =  compute_and_check roms

compute_and_check:: rom_num -> num
compute_and_check roms    =  thousands roms

thousands:: rom_num -> num
thousands  ('M' : rest)  =  value 'M' + thousands rest
thousands  roms          =  fivehundreds roms
```

```
fivehundreds:: rom_num -> num
fivehundreds  ('C' : 'M' : rest)    =  value 'M' – value 'C' + fifties rest
fivehundreds  ('D' : rest)          =  value 'D' + hundreds rest
fivehundreds  ('C' : 'D' : rest)    =  value 'D' – value 'C' + fifties rest
fivehundreds  roms                  =  hundreds roms

hundreds:: rom_num -> num
hundreds   ('C' : 'C' : 'C' : rest)  =  3 * value 'C' + fifties rest
hundreds   ('C' : 'C' : rest)        =  2 * value 'C' + fifties rest
hundreds   ('C' : rest)              =  value 'C' + fifties rest
hundreds   roms                      =  fifties roms

fifties:: rom_num -> num
fifties    ('X' : 'C' : rest)    =  value 'C' – value 'X' + fives rest
fifties    ('L' : rest)          =  value 'L' + tens rest
fifties    ('X' : 'L' : rest)    =  value 'L' – value 'X' + fives rest
fifties    roms                  =  tens roms

tens:: rom_num -> num
tens  ('X' : 'X' : 'X' : rest)  =  3 * value 'X' + fives rest
tens  ('X' : 'X' : rest)        =  2 * value 'X' + fives rest
tens  ('X' : rest)              =  value 'X' + fives rest
tens  roms                      =  fives roms

fives:: rom_num -> num
fives  ['I', 'X']      =  value 'X' – value 'I'
fives  ('V' : rest)    =  value 'V' + ones rest
fives  ['I', 'V']      =  value 'V' – value 'I'
fives  roms            =  ones roms

ones:: rom_num -> num
ones     ['I', 'I', 'I']    =  3 * value 'I'
ones     ['I', 'I']         =  2 * value 'I'
ones     ['I']              =  value 'I'
ones     [ ]                =  0
ones     roms               =  error  ( "not a proper roman number: " ++
                                               roms ++ " is the incorrect part" )

value:: char -> num
value 'M'    =     1000
value 'D'    =      500
value 'C'    =      100
value 'L'    =       50
value 'X'    =       10
value 'V'    =        5
value 'I'    =        1
```

5 Test and debug

5.1 Design critical test values to test all designed functions.

Check types, design critical test function. For example, test the function rom_to_num with arguments IVI, –I, IIIII, IVXLCDM, CMI, I.

6 Documentation

6.1 Check comments and names on consistency with (revised) functions.

Check the significance of names, check the comments to see if they are consistent with the latest version of the program.

7 Reflection

7.2 Summarize what you have learned about designing an algorithm for the given problem.

To avoid running through a list structure more times than necessary it is in many cases more efficient to combine function definitions that manipulate a list structure into new function definitions that produce multiple results, e.g. in a tuple. Accordingly, the algorithm was changed (for further reflection see Exercise 2.9).

Summary

- The functional programming language Miranda extends the basic concepts with *syntactical sugar* incorporating most of the advanced concepts found in modern functional languages.
- The language has a *nice*, purely functional *notation* fully based on *function application*.
- *Pattern matching* can be used in combination with *guarded equations*.
- *Local function definitions* can be defined in a nested way inside the scope of a global function definition.
- Data structuring facilities are given by the presence of *lists*, *tuples*, *list comprehensions*, *algebraic data types* and *abstract data types*.
- Milner's *type inference* algorithm provides *strong polymorphic typing*. However, for some cases Mycroft's *type checking* algorithm is used allowing more functions to be correctly typed.
- With these sophisticated features one can write *compact* and *elegant* (even *interactive*) functional programs.
- The elegant mathematical style of programming common in the functional programming community still requires, of course, a programmer to pay attention to a good *programming methodology*.

EXERCISES

2.1 Define a function takesum that takes two functions f:: num -> num, g:: num -> num and a number n to calculate

$$\sum_{i=0}^{n} f(g^i(0)) = f(0) + f(g(0)) + f(g(g(0))) + \ldots$$

For instance, if $f(x) = x^2$, $g(x) = x + 1$ and $n = 2$ than the result must be: $0^2 + g(0)^2 + g(g(0))^2 = 0^2 + 1^2 + 2^2 = 5$

Show how this function can be used to approximate $e = \sum_{i=0}^{\infty} \frac{1}{i!}$

2.2 Write functions that compute the following lists using *list comprehensions* where possible:

(a) All 5-tuples of integers such that their sum is less than 7 and greater than –7.

(b) All perfect natural numbers (natural numbers that are equal to the sum of their dividers; e.g. 6 = 1 + 2 + 3)

2.3 The function numchar yields the number of times a given character is appearing in a finite list of characters. Define numchar (a) with recursion, (b) with foldr and (c) with list comprehension.

2.4 Define the function map (a) with foldr and (b) with a *ZF*-expression.

2.5 Write a function eightqueens that yields all solutions of the eight queens problem: the queens should be put down on a 8 * 8 chessboard such that they cannot take each other.

2.6* A binary search tree *T* is an AVL-tree if for every node *v* in *T*: *–1 <= depth (right (v)) – depth (left (v)) <= 1* with *depth*, *right* and *left* defined as usual. The *balance factor* of a node *v* is: *depth (right (v)) – depth (left (v))*. So in an AVL-tree the balance factor of every node is –1, 0 or +1. Inserting or deleting a node cannot be done in the way it is done in ordinary binary search trees. The AVL-tree may not be balanced any more after a deletion or insertion. In such a case there is a node with balance factor +2 or –2. Define an abstract data type avl_tree * with, as operations:

empty	delivers an empty AVL-tree.
key	delivers the value of the root.
left, right	gives the left or right subtree of an AVL-tree.
is_empty	tests if an AVL-tree is empty.
depth	gives the depth of an AVL-tree.

retrieve	delivers the *i*th value in the AVL-tree.
find	decides if a certain value is contained in the tree.
insert	inserts a given value in the AVL-tree.
delete	deletes a value in the AVL-tree.

2.7* Write a small editor. An editor operates on a set of lines, called the buffer. Implement the buffer as an abstract data type with appropriate operations (get_line, remove_ line). An editor should be able to show lines, insert lines, delete lines and change lines (replace a substring).

2.8* Write an interactive program that can play the Mastermind game. The program has to guess a secret code of 4 numbers (0–9). For each guess given by the program a player returns how many numbers appear in the code and how many of these numbers are on a correct position.

2.9 In the final program in Section 2.7 an error is given if the argument is incorrect. So the function cannot be used in a context in which the programmer wants to specify what has to be done for incorrect arguments. Write a function that produces a tuple with a boolean indicating the correctness and a number indicating the value if the argument was correct.