Part 1
Functional programming

This part serves as an introduction to functional programming.

Since there are many different functional programming languages, the first chapter focuses on the basic concepts most functional languages have in common. By first focusing on these concepts it will be relatively easy to master the basics of any functional language with its own syntactical sugar. The concrete examples in Chapter 1 are given in a subset of Miranda.

In Chapter 2 the more sophisticated features of Miranda are shown to demonstrate the expressive power and notational elegance of a specific state-of-the-art functional programming language. More elaborate program examples are given with modest suggestions for a functional programming methodology.

For readers who want to study the art of functional programming in more detail good introductory books are available (e.g. Bird and Wadler, 1988).

Chapter 1
Basic concepts

This chapter first explains the advantages and drawbacks of a functional programming style compared with the commonly used classical imperative style of programming (Section 1.1). The difference in objectives between a mathematical specification of a function and a function specification in a functional programming language is explained in Section 1.2.

This is followed by an overview of the basic concepts of most functional programming languages (Sections 1.3–1.7). These concepts are explained using a concrete functional programming language: Miranda (Turner, 1985). Functions can be defined by a single equation (Section 1.3) as well as by more than one equation using guards and patterns to discriminate between the alternative equations (Section 1.5). Other topics that are covered are: higher order functions and currying (Section 1.7), lists as basic data structures (Section 1.6) and (lazy as well as eager) function evaluation (Section 1.4).

Traditional proof techniques like symbolic substitution and mathematical induction can be used to prove correctness of functional programs (Section 1.8). Finally, some small example programs are presented in Section 1.9.

1.1 Why functional programming?

Imagine the availability of perfect computer systems: software and hardware systems that are user-friendly, cheap, reliable and fast. Imagine that programs can be specified in such a way that they are not only very understandable but that their correctness can easily be proven as well. Moreover, the underlying hardware ideally supports these software systems and superconduction in highly parallel architectures makes it possible to get an answer to our problems at dazzling speed.

Well, in reality people always have problems with their computer systems. Actually, one does not often find a bug-free piece of software or hardware. We have learned to live with the *software crisis*, and have to accept that most software products are *unreliable*, *unmanageable* and *unprovable*. We spend money on a new release of a piece of software in which old bugs are removed and new ones are introduced. Hardware systems are generally much more reliable than software systems, but most hardware systems appear to be designed in a hurry and even well-established processors contain errors.

Software and hardware systems clearly have become very complex. A good, orthogonal design needs many person years of research and development, while at the same time pressure increases to put the product on the market. So it is understandable that these systems contain bugs. The good news is that hardware becomes cheaper and cheaper (thanks to very large scale integration) and speed can be bought for prices one never imagined. But the increase of processing power automatically leads to an increase of the use of that power with an increasing complexity of the software as a result. So computers are never fast enough, while the complexity of the systems is growing enormously.

The two key problems that the computer science community has to solve are:

- How can we, at low cost, make large software systems that remain reliable and user-friendly?
- How can we increase processing power at low cost?

Researchers are looking for solutions to these problems: by investigating *software engineering techniques*, to deal with problems related to the management of software projects and the construction and maintenance of software; by designing new *proof techniques* to tackle the problems in proving the correctness of systems; by developing *program transformation techniques*, to transform the specification of a problem into a program that solves it; and by designing new (parallel) *computer architectures* using many processors (thousands or more). In the mean time the quest for revolutionary new *technologies* (e.g. optical chips, superconduction) is always going on.

Another approach is based on the idea that the problems mentioned above are *fundamental* problems that cannot be solved unless a totally different approach is used and hardware and software are designed with a completely *different model of computation* in mind.

1.1.1 An imperative programming style

Most computer programs are written in an **imperative programming language** in which algorithms are expressed by a sequence of commands. These languages, such as FORTRAN, C, Algol, COBOL, PL/1 and Pascal, are originally deduced from (and form an abstraction of) the computer architectures they are running on (see Figure 1.1). These computer architectures, although different in detail, are all based on the same architecture: the **Von Neumann computer architecture** (Burks *et al.*, 1946). The Von Neumann computer architecture is based on a mathematical model of computation proposed by Turing in 1937: the **Turing machine**.

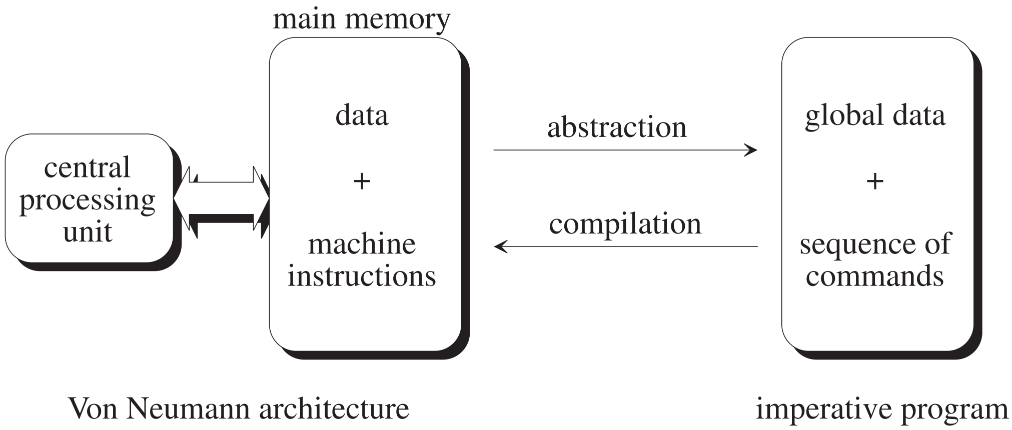

Figure 1.1 Imperative programming and the Von Neumann computer architecture.

The great advantage of this model and the corresponding architecture is that they are extremely simple. The Von Neumann architecture consists of a piece of memory that contains information that can be read and changed by a central processing unit (the **CPU**). Conceptually there are two kinds of information: program **instructions** in the form of machine code, information that is interpreted by the CPU and that controls the computation in the computer; and **data**, information that is manipulated by the program. This simple concept has made it possible to make efficient realizations in hardware at a relatively low cost.

In the beginning of the computer era, the 'high-level' imperative programming languages were designed to provide a notation to express computations in a machine-independent way. Later on, one recognized the importance of expressing computations such that programs are understandable for the human being and their correctness can be proven. It became clear that, in order to make it possible to reason about programs, not every machine instruction should have a direct equivalent in the high-level programming language. For instance, it is common knowledge that with the use of GOTO statements, which are the direct abstraction of the branch and jump instructions that are available on any computer, programs can be written in such a way that reasoning about them is almost impossible (Dijkstra, 1968). We strongly believe that a similar kind of problem is caused by the assignment statement.

Consider the following example written in an imperative programming style:

```
BOOLEAN even := TRUE;
…
PROCEDURE calculate (INTEGER value) : INTEGER;
BEGIN
   even := NOT even;
   IF even
   THEN value + 1
   ELSE value + 2
   ENDIF
END;
…
print(calculate (6));
…
print(calculate (6));
```

Both print statements in this program are syntactically the same. Still they may produce different results. Clearly either the value 7 or 8 is printed in both cases, but the exact value printed depends on the number of times the procedure calculate is called. The result returned by the procedure not only depends on the actual value of its argument, but also on the value the global boolean has at that particular moment. This value is 'secretly' changed in each procedure call. Such side-effects cause the result of a procedure call to be context dependent. Results become very hard to predict when, in a large program, global variables are changed all over the place.

One of the most important drawbacks of an imperative programming style is that an imperative program consists of a sequence of commands of which the dynamic behaviour must be known to understand how such a program works. The assignment causes problems because it changes the value (and often even the meaning) of a variable. Owing to side-effects it can happen that evaluating the same expression in succession produces different answers. Reasoning about the correctness of an imperative program is therefore very difficult.

Furthermore, because of the command sequence, algorithms are more sequential than is necessary. Therefore it is hard to detect which parts of the algorithm can or cannot be executed concurrently. This is a pity, since concurrent evaluation seems to be a natural way to increase execution speed.

A conjecture adopted by many researchers nowadays is that the software crisis and the speed problem are inherent to the nature of imperative programming languages and the underlying model of computation. Therefore, other styles of programming, such as *object oriented*, *logical* and *functional* styles of programming are investigated.

1.1.2 A functional programming style

John Backus (1978) pointed out that the solution for the software problems has to be found in using a new discipline of programming: a functional programming style instead of an imperative one.

In a **functional program** the result of a function call is uniquely determined by the actual values of the function arguments. No assignments are used, so no side-effects are possible. As a consequence, it makes no difference where and under what conditions a function is called. The result of a function will, under all conditions, be determined solely by the value of its arguments. For instance, in the program below the value printed will be 42 in both cases.

```
FUNCTION increment (INTEGER value) : INTEGER;
BEGIN
   value + 1
END;
…
print(increment (41));
…
print(increment (41));
```

It is much easier to reason about the result produced by a function that has no side-effects than about the result of an imperative procedure call with side-effects (see also Section 1.8).

Advantages of functional programming languages

So perhaps a functional programming style is important and side-effects, and hence the assignment statement, should be abandoned. But why should we use a functional programming language? Is it not possible to use the familiar languages? The common imperative languages also have functions, so why not restrict ourselves and only use the functional subset of for instance C, Algol or Modula2?

Well, one can use the functional subset of imperative languages (i.e. only using functions and leaving out the assignment), but then one is deprived of the expressive power of the new generation of functional languages that treat functions as 'first class citizens'. In most imperative languages functions can only be used restrictively. An *arbitrary* function cannot be passed as an argument to a function nor yielded as result.

For instance, the function twice takes a function and an argument, applies the function 'twice' to the argument and yields the result that again might be a function. The function twice can be used in various ways, e.g. by applying the result again as a function to respectively one or two arguments, etc.

```
FUNCTION
twice (f : FUNCTION from ANYTYPE to ANYTYPE, x: ANYTYPE ) :
   result ANYTYPE;
BEGIN
      f ( f (x) )
END;
…
print( twice (increment, 0) );
print( ( twice (twice, increment) ) (0) );
print( ( twice (twice, twice) ) (increment, 0) );
print( ( ( twice (twice, twice) ) (twice, increment) ) (0) );
…
```

Functions like twice are hard to express in a classical imperative language.

Functional programming languages have the advantage that they offer *a general use of functions* which is not available in classical imperative languages. This is a fact of life, not a fundamental problem. The restricted treatment of functions in imperative languages is due to the fact that when these languages were designed people simply did not know how to implement the general use of functions efficiently. It is also not easy to change the imperative languages afterwards. For instance, the type systems of these languages are not designed to handle these kinds of function. Also the compilers have to be changed dramatically.

In a traditional imperative language one would probably have severe problems expressing the type of twice. In a functional language such a function definition and its application can be expressed and typed in the following way:

```
twice:: (* -> *) -> * -> *          || type definition
twice f x     =  f (f x)            || function definition

?twice increment 0                  || function application, yields a number
```

```
?twice twice increment 0            || function application, yields a number
?twice twice twice increment 0      || function application, yields a number
…
```

Another advantage is that in most modern functional programming language(s) (FPLs) *the functional programming style is guaranteed*: the assignment statement is simply not available (like GOTOs are not available in decent modern imperative languages). FPLs in which there are no side-effects or imperative features of any kind are called **pure** functional languages. Examples of pure functional languages are Miranda, LML (Augustsson, 1984), HOPE (Burstall *et al.,* 1980), Haskell (Hudak *et al.,* 1992) and Concurrent Clean (Nöcker *et al.*, 1991b). LISP (McCarthy, 1960) and ML (Harper *et al.,* 1986) are examples of well-known functional languages which are impure. From now on only *pure* aspects of FPLs are considered.

In pure FPLs the programmer can only define functions that compute values uniquely determined by the values of their arguments. The assignment statement is not available, and nor is the heavily used programming notion of a variable as something that holds a value that is changed from time to time by an assignment. Rather, the **variables** that exist in purely functional languages are used in mathematics to name and refer to a yet *unknown constant value*. This value can never be altered. In a functional style a desired computation is expressed in a static fashion instead of a dynamic one. Due to the absence of side-effects, program correctness proofs are easier than for imperative languages (see Section 1.8). Functions can be evaluated in any order, which makes FPLs suitable for parallel evaluation (see Section 1.4). Furthermore, the guaranteed absence of side-effects enables certain kinds of analysis of a program, for example strictness analysis (see Chapter 7) and uniqueness analysis (see Chapter 8).

Besides the full availability of functions, the new generation functional languages also offer an elegant, user-friendly notation. Patterns and guards provide the user with simple access to complex data structures; basically one does not have to worry about memory management any more. Incorrect access of data structures is impossible. Functional programs are in general much shorter than their conventional counterparts and thus in principle easier to enhance and maintain.

The question arises, is it possible to express any possible computation by using pure functions only? Fortunately, this question has already been answered years ago. The concept of a function is one of the fundamental notions in mathematics. One of the greatest advantages of functional languages is that they are based on a sound and well-understood mathematical model, the **λ-calculus** (Church, 1932; 1933). One could say that functional languages are sugared versions of this calculus. The λ-calculus was introduced at approximately the same time as the Turing model (Turing, 1937). *Church's thesis* (Church, 1936) states that the class of **effectively computable functions**, i.e. the functions that intuitively can be computed, is the same as the class of functions that can be defined in the λ-calculus. Turing formalized machine computability and showed that the resulting notion of *Turing computability* is equivalent to *λ-definability*. So the power of both models is the same. Hence *any* computation can be expressed using a functional style only. For more information on λ-calculus we refer to Barendregt (1984).

Disadvantages of functional programming languages

The advantages mentioned above are very important. But although functional languages are being used more frequently, in particular as a language for rapid prototyping and as a language in which students learn how to program, functional languages are not yet commonly used for general purpose programming. The two main drawbacks lie in the fields of

- *efficiency* and
- *suitability* for applications with a strongly *imperative nature*.

Firstly, until recently programs written in a functional language ran very, very slowly in comparison with their imperative counterparts. The main problem is that the traditional machine architectures on which these programs have to be executed are not designed to support functional languages. On these architectures function calls are relatively expensive, in particular when the lazy evaluation scheme (see Section 1.4) is used. Our computers are ideally suited for destructive updates as present in imperative languages (assignments), but these are conceptually absent in functional languages. It is therefore not easy to find an efficient compilation scheme. Another big problem is caused by the fact that for some algorithms, due to the absence of destructive updates, the time and space complexity can be much worse for a functional program than for its imperative equivalent. In such a case, to retain the efficiency, program transformations are necessary.

As will be shown in this textbook, by using several new (and old) compilation techniques, the efficiency of (lazy) functional programs can nowadays be made acceptable in many (but certainly not all) cases. New compilation techniques have been developed at several research institutes. These techniques are quite complex. Commercial compilers are therefore not widely available yet. This will soon change. Anyhow, in our opinion, the advantages of functional languages are so important that some loss of efficiency is quite acceptable. One has to keep in mind that decades ago we accepted a loss of efficiency when we started to use high-level imperative languages instead of machine assembly languages.

Secondly, a very important drawback of functional languages was that some algorithms could not be expressed elegantly in a functional programming style. In particular, this seemed to hold for applications that strongly interact with the environment (interactive programs, databases, operating systems, process control). But, the problem is largely caused by the fact that the art of functional programming is still in development. We had and still have to learn how to express the different kinds of applications elegantly in a functional style. We now know that strongly interactive applications can be expressed very elegantly in a functional programming style. One example is the way interactive programs that use windows, dialogs, menus and the like can be specified in Clean (see Chapter 8). Another example is the definition of the abstract imperative machine given in Chapter 10.

The advantages of a functional programming style are very important for the development of reliable software. The disadvantages can be reduced to an acceptable level. Therefore we strongly believe that one day functional languages will be used worldwide as general purpose programming languages.

1.2 Functions in mathematics

Before discussing the basic concepts of most functional languages, we want to recall the mathematical concept of a function. In mathematics a **function** is a mapping from objects of a set called the **domain** to objects of a set called **co-domain** or **range** (see Figure 1.2).

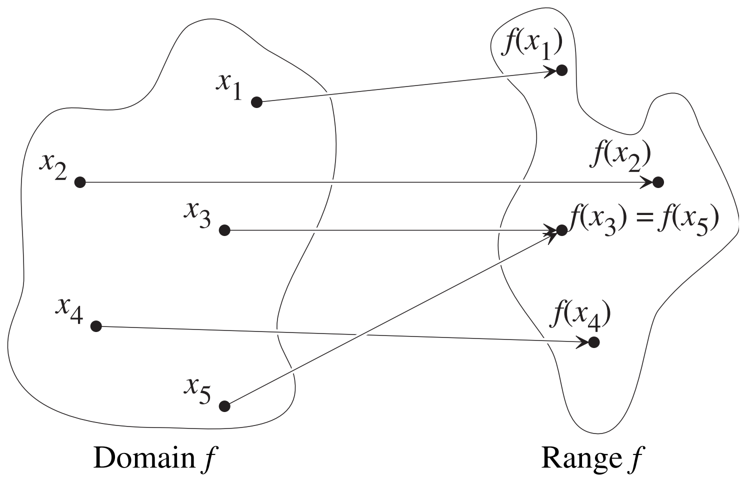

Figure 1.2 A function f maps objects x from the domain set to objects $f(x)$ in the range set.

This mapping need not be defined for all objects in the domain. If the mapping is defined for an object, this object is mapped to exactly one object in the range. This object in the range is called the **image** of the corresponding object in the domain. If all objects in the domain have an image, the function is called a **total function**, otherwise it is called a **partial function**. If x is an object in domain A and f is a function defined on domain A, the image of x is called $f(x)$.

The **type of a function** is defined as follows. If x is an object in the domain A, x is said to be of *type* A. If y is an object in range B, y is said to be of *type* B. If a function f maps objects from the domain A to the range B, f is said to be of *type* $A \to B$, which is pronounced as 'from A to B'. The type of f is generally specified as:

$$f\colon\ A \to B$$

In mathematics, there are several ways to define a function. The type of a function can be specified separately from the function definition.

One way to define a function is by explicit *enumeration* of all objects in the domain on which the function is defined with their corresponding images. An example of this is the following partial function (domain names Z and N are used for the domains of integers and natural numbers).

$$\begin{aligned} &abs\colon Z \to N \\ &abs(-1) = 1 \\ &abs(\ 0) = 0 \\ &abs(\ 1) = 1 \end{aligned}$$

Another way to define functions is by using definitions that consist of one or more (recursive) *equations*. For example, with this method the *abs*-function above can easily be defined as a total function, applicable for all objects in the domain. Of course, the functions and operators used on the right-hand side must be defined on the appropriate domains.

$$\begin{aligned} &abs\colon Z \to N \\ &abs(n) = n, \quad n > 0 \\ & = 0, \quad n = 0 \\ & = -n, \quad n < 0 \end{aligned}$$

A function like factorial can be defined as follows:

$$\begin{aligned} &fac\colon N \to N \\ &fac(n) = 1, \quad n = 0 \\ & = n * fac\ (n-1), \quad n > 0 \end{aligned}$$

or an alternative definition method is:

$$\begin{aligned} &fac\colon N \to N \\ &fac(0) = 1 \\ &fac(n) = n * fac\ (n-1), \quad n > 0 \end{aligned}$$

A mathematician would consider the definitions above as very common, ordinary function definitions. But these examples are also perfect examples of function definitions in a functional programming language. Notationally a function definition in a functional language has many similarities with a function definition in mathematics. However, there is an important difference in objective. The objective in a functional language is not only to define a function, but also to define a *computation* that automatically computes the image (result) of a function when it is applied to a specific object in its domain (the actual argument of the function).

Some function definitions, well-defined from a mathematical point of view, cannot be defined similarly in a functional language, because the images of some functions are very difficult to compute or even cannot be computed at all.

Consider, for example, the following function definition:

$$\begin{aligned}
&\textit{halting: All_Programs} \rightarrow N \\
&\textit{halting}(p) \;=\; 1, \quad \textit{if the execution of p will stop} \\
&\qquad\qquad\quad\;\; 0, \quad \textit{otherwise}
\end{aligned}$$

The *halting* function as indicated above is a problem that is not computable, and therefore an attempt to express it will not produce the desired computation. Suppose that a function in an FPL would try to calculate the image of the *halting* function for an arbitrary program. The only way of doing this is more or less running the program. But then the function would simply not terminate if the argument program does not terminate, in which case the result 0 would never be produced. For another example, consider:

$$\begin{aligned}
&f: R \rightarrow R,\; g: R \rightarrow R \\
&f'' - 6g' \;=\; 6 \sin x \\
&6g'' + a^2 f' \;=\; 6 \cos x \\
&f(0) = 0,\, f'(0) = 0,\, g(0) = 1,\, g'(0) = 1
\end{aligned}$$

The equations for f, g and their derivatives f', f'', g' and g'' are solvable, but it is not easy to compute such functions.

Some special purpose programming languages are able to calculate functions by applying special purpose calculation techniques (symbolic computation using computer algebra techniques or formula transformations). But a general purpose functional programming language uses a very simple model of computation, based on substitution. So when functions are defined in an FPL a computation through substitutions is defined implicitly (see Section 1.4).

1.3 A functional program

A program written in a functional language consists of a collection of *function definitions* written in the form of *recursive equations* and an *initial expression* that has to be evaluated. From now on a Miranda-based syntax (see Appendix A) will be used.

1.3.1 Function definitions

A **function definition** consists of one or more equations. An **equation** consists of a left-hand side, an equals symbol (=) and a right-hand side.

The left-hand side defines the **function name** and its **formal arguments** (also called **formal parameters**). The right-hand side specifies the **function result**. It is also called the **function body**. This function body consists of an expression. Such an **expression** can be a denotation of some *value*, or it can be a *formal argument*, or a *function application*.

In a **function application** a function is applied to an expression, the **actual argument**. The application of a function *f* to an expression *a* is denoted as *f a*. So function application is denoted by simple juxtaposition of the function and its argument. An important syntactical convention is that in every expression function application has always the *highest* priority (on both sides of the equations). A function definition can be preceded by its *type definition* (indicated by post-fixing the function name with a '::' followed by its type).

Below are some examples of function definitions (now in Miranda-based notation). In Section 1.6 more complex definitions can be found with more than one alternative per function and guards or patterns to indicate which alternative has to be chosen. The || indicates that the rest of the line is a comment.

```
ident:: num -> num                  || ident is a function from num to num,
ident x  =  x                       || the identity function on numbers

alwaysseven:: num -> num            || a function from num to num,
alwaysseven x  =  7                 || that yields 7, independent of arg. x

inc:: num -> num                    || a function from num to num,
inc x  =  x + 1                     || that returns the value of its arg. + 1

square:: num -> num                 || square function
square x  =  x * x

squareinc:: num -> num              || square increment of argument
squareinc x  =  square (inc x)

fac:: num -> num                    || the factorial function
fac x  =  cond (x = 0) 1 (x * fac (x – 1))
```

In the last example cond is a predefined operator with three arguments: a boolean, a then part and an else part. Its semantics corresponds to a conditional choice in imperative languages.

A formal argument, such as x in the example above, is also called a **variable**. The word variable is used here in the mathematical sense of the word that is not to be confused with the use of the word variable in an imperative language. This variable does not vary. Its scope is limited to the equation in which it occurs (whereas the defined function names have the whole program as scope).

Functions defined as above are called **user-defined functions**. One can also denote and manipulate objects of certain *predefined types* with given *predefined operators.* These predefined basic operators can be regarded as **predefined functions**. For mathematical–historical reasons, and therefore also for user convenience, such primitive functions are often defined as *infix* functions or operators. This is in contrast to the user-defined functions that are generally defined as *prefix* functions.

Examples of predefined types (numbers, booleans, characters), corresponding to predefined operators (functions), denotation of values (concrete objects) of these types, and the ‘real-life’ domain with which they can be compared.

Types	**Operators**	**Denotation of values**	**Comparable with**
num	+, –, *, …	0, 1, 34.7, –1.2E15, …	real numbers
bool	and, or, …	True, False	truth values
char	=, <, …	'a', 'c', …	characters

1.3.2 The initial expression

The **initial expression** is the expression whose value has to be calculated.

For example, in the case that the value of 2 + 3 has to be calculated, the initial expression 2 + 3 is written. But one can also calculate any application of user-defined functions: squareinc 7.

1.4 The evaluation of a functional program

The execution of a functional program consists of the evaluation of the initial expression in the context of the function definitions in the program called the **environment**.

A functional program: a set of function definitions and an initial expression.

```
ident:: num -> num
ident x  =  x
```

```
inc:: num -> num
inc x  =  x + 1

square:: num -> num
square x    =  x * x

squareinc:: num -> num
squareinc x    =  square (inc x)

squareinc 7
```

The evaluation consists of repeatedly performing *reduction* or *rewriting steps*. In each **reduction step** (indicated by a '→') a function application in the expression is replaced (*reduced, rewritten*) by the corresponding function body (the right-hand side of the equation), substituting the formal arguments by the corresponding actual arguments. A (sub)expression that can be rewritten according to some function definition is called a **redex** (**red**ucible **ex**pression). The basic idea is that the reduction process stops when none of the function definitions can be applied any more (there are no redexes left) and the initial expression is in its most simple form, the **normal form.** This is the result of the functional program that is then printed out.

For instance, given the function definitions above (the environment), the initial expressions below can be evaluated (reduced) as follows. In the examples the redex that will be reduced has been underlined.

$\underline{\text{ident 42}} \quad \rightarrow \quad 42$

$\underline{\text{squareinc 7}} \quad \rightarrow \quad \text{square } (\underline{\text{inc 7}}) \quad \rightarrow \quad \text{square } (\underline{7 + 1})$
$\rightarrow \quad \underline{\text{square 8}} \quad \rightarrow \quad \underline{8 * 8} \quad \rightarrow \quad 64$

$\underline{\text{square } (1 + 2)} \quad \rightarrow \quad (\underline{1 + 2}) * (1 + 2) \quad \rightarrow \quad 3 * (\underline{1 + 2}) \quad \rightarrow \quad \underline{3 * 3} \quad \rightarrow \quad 9$

However, the initial expression may *not* have a normal form at all. As a consequence, the evaluation of such an initial expression will not terminate. Infinite computations may produce partial results that will be printed as soon as they are known (see Section 1.6.3).

Example of a non-terminating reduction. Take the following definition:

```
inf    =  inf
```

then the evaluation of the following initial expression will not terminate:

$\underline{\text{inf}} \quad \rightarrow \quad \underline{\text{inf}} \quad \rightarrow \quad \underline{\text{inf}} \quad \rightarrow \quad \dots$

1.4.1 The order of evaluation

Because there are in general many redexes in the expression, one can perform rewrite steps in several orders. The actual order of evaluation is determined by the **reduction strategy** which is dependent on the kind of language being used. There are a couple of important things to know about the ordering of reduction steps.

Due to the absence of side-effects, the result of a computation does *not* depend on the chosen order of reduction (see also Chapter 3). If all redexes are vanished and the initial expression is in normal form, the result of the computation (if it terminates) will always be the same: the normal form is *unique*.

For instance, one can compute one of the previous expressions in a different order, but the result is identical:

```
square (1 + 2)   →   square 3        →   3 * 3   →   9
```

It is sometimes even possible to rewrite several redexes at the same time. This forms the basis for *parallel* evaluation.

Reducing several redexes at the same time:

```
square (1 + 2)   →   (1 + 2) * (1 + 2)   →   3 * 3   →   9
```

However, the order is not completely irrelevant. Some reduction orders may not lead to the normal form at all. So a computation may not terminate in one particular order while it would terminate when the right order was chosen (see again Chapter 3).

Example of a non-terminating reduction order. Assume that the following (recursive) functions are defined:

```
inf = inf

alwaysseven x = 7
```

Now it is possible to repeatedly choose the 'wrong' redex which causes an infinite calculation:

```
alwaysseven inf   →   alwaysseven inf   →   alwaysseven inf   →   …
```

In this case another choice would lead to termination and to the unique normal form:

```
alwaysseven inf   →   7
```

The reduction strategy followed depends on the kind of FPL. In some languages, e.g. LISP, ML and HOPE, the arguments of a function are always reduced before the function application itself is considered as a redex. These languages are called **eager** or **strict** languages. In most recent FPLs, e.g. Miranda and Haskell, the rewriting is done lazily. In **lazy functional languages** the value of a subexpression (redex) is calculated if and only if this value must be known to find the normal form. The lazy evaluation order will find the normal form if it exists.

Illustrating lazy rewriting:

```
alwaysseven inf     →   7
```

A more complex example to illustrate lazy evaluation is shown below. The predefined conditional function demands the evaluation of its first argument to make a choice between the then and else parts possible. The equality function forces evaluation in order to yield the appropriate Boolean value. Multiplication and subtraction are only possible on numbers; again evaluation is forced.

```
fac 2
→   cond (2 = 0) 1 (2 * fac (2 – 1))
→   cond FALSE 1 (2 * fac (2 – 1))
→   2 * fac (2 – 1)
→   2 * cond (2 – 1 = 0)1 ((2 – 1) * fac ((2 – 1) – 1))
→   2 * cond (1 = 0) 1 ((2 – 1) * fac ((2 – 1) – 1))
→   2 * cond FALSE 1 ((2 – 1) * fac ((2 – 1) – 1))
→   2 * (2 – 1) * fac ((2 – 1) – 1)
→   2 * 1 * fac ((2 – 1) – 1)
→   2 * fac ((2 – 1) – 1 )
→   2 * cond ((2 – 1) – 1 = 0) 1 (((2 – 1) – 1) * fac (((2 – 1) – 1) – 1))
→   …   →   2 * 1 * 1   →   2 * 1 →   2
```

1.5 Functions with guarded equations and patterns

In the previous section ordinary recursive equations were used to define a function. But often one needs a function description that depends on the actual values of the objects in the domain, or one wants to make a separate description for each subclass of the domain that has to be distinguished. Of course, a (predefined) function can be used, like `cond` in the factorial example in Section 1.3. However, it is much more convenient to use *guarded equations* or *patterns* for such a case analysis.

1.5.1 Guarded equations

The right-hand side of a definition can be a guarded equation. A function definition using **guarded equations** consists of a sequence of *al-*

ternative equations, each having a **guard**: an expression yielding a Boolean result, the textual first alternative for which the corresponding guard is True being selected. Guards can be preceded by the keyword if. The guard of the last alternative can be just the keyword otherwise and the corresponding alternative is chosen if and only if all other guards evaluate to False. Although it is allowed, it is good programming style not to use overlapping guards.

Function definitions with guarded equations:

```
fac:: num -> num
fac n   =  1,                  if  n = 0
        =  n * fac (n – 1),    if  n > 0

abs:: num -> num
abs n   =  n,     if  n >= 0
        =  –n,    otherwise
```

It is quite possible to define partial functions using guards. When a function is called with actual parameters that do not satisfy any of the guards, it is considered to be a fatal programming error.

For example, the Fibonacci function below will cause a fatal error when it is called with an argument less than one (the ∨-operator denotes the logical OR).

```
fib:: num -> num
fib n  =  1,                          if  (n = 1) ∨ (n = 2)
       =  fib (n – 1) + fib (n – 2),  if  n > 2
```

The programmer should of course avoid this situation. One way to avoid it is to specify the domain (type) of a function accurately, in such a way that a total function is defined on that domain. The type system of the language should therefore allow the specification of new types or of subtypes (see Chapter 2). However, an arbitrary accurate specification of the domain is generally not possible because it leads to undecidable type systems. Total functions can be accomplished in another way by adjusting the definition of the partial function. The guards should then always cover the whole domain the function is defined on. To make this possible in an easy way most languages are equipped with an error routine that fits in any type.

The Fibonacci function above is now changed into a total function. It will still yield a run-time error when it is called with an argument less than one. But now this situation is handled by the programmer and an appropriate error message can be given.

```
fib:: num -> num
fib n  =  1,                                              if  (n = 1) \/ (n = 2)
       =  fib (n – 1) + fib (n – 2),                      if  n > 2
       =  error "Fibonacci called with argument less than one", otherwise
```

1.5.2 Patterns

It is also possible to discriminate between alternative equations by using *patterns* on the left-hand side. These **patterns** are *values* (e.g. 0) including *data constructors* (see the next section) or *variables*. The meaning of a pattern is that the equation in question is only applicable if the actual arguments of the function match the pattern. An actual argument **matches** a corresponding *pattern value* if it has the *same* value. A *pattern variable* is matched by any actual argument. An equation is only applicable if all the actual arguments match the corresponding patterns. Patterns are tried from left to right, equations are tried in textual order: from top to bottom. When an actual argument is matched against a non-variable pattern, the argument is evaluated first after which the resulting value is compared with the specified pattern.

A function definition with patterns:

```
fac:: num -> num
fac 0    =  1
fac n    =  n * fac (n – 1)
```

0 and n are the patterns of the first two rules (a variable as formal parameter indicates that it does not matter what the value is). Calling fac (7 – 1) will result in a call to the pattern-matching facility, which decides that (after the evaluation of the actual argument) only the second rule is applicable (6 ~= 0).

Patterns can also be used in combination with guarded equations.

```
fac:: num -> num
fac 0    =  1
fac n    =  n * fac (n – 1),                                 if  n > 0
         =  error "factorial called with argument less than zero",  otherwise
```

1.5.3 The difference between patterns and guards

Patterns have a limited power: they can only be used to test whether actual arguments are of a certain value or form. Using guards any function yielding a Boolean result can be applied to the actual arguments. So guards are more powerful than patterns, but, on the other hand, patterns are easier to read and sufficient for most definitions. Using patterns in combination with guards often leads to clearer and more concise definitions and is highly recommended.

1.6 Data structures

In imperative languages one can define global data structures that are globally accessible for reading and writing during execution. Since this is not possible in functional languages it will be clear that the use of data structures in these languages is quite different. After creation of a data structure, the only possible access is read access. Data structures cannot be overwritten. Furthermore, data structures are not globally available but they must always be passed as arguments to the functions that need them.

Modern functional languages allow the definition of structured data objects like the records in Pascal. How user-defined data structures can be defined is explained in the next chapter. In this chapter only lists are treated.

Lists are the most important basic data structure in any FPL. **Lists** in FPLs are actually linked lists, each element in the list has the *same*[1] type *T* (see Figure 1.3). The *last* element in the list is indicated by a special element, generally called Nil.

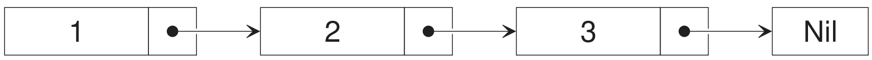

Figure 1.3 An example of a list.

Because lists are so important, they are generally predefined. A list is conceptually not different from any other user-defined data structure (see Chapter 2).

Lists, like any data structure in an FPL, are built using *data constructors*. A **data constructor** is a special constant value that is used as a tag that *uniquely* identifies (and is part of) an object of a certain type. So one can recognize the type of a data structure just by looking at the data constructor it contains. Several different data constructors can be used to identify the different objects of the same type.

Lists are constructed using two data constructors (see Figures 1.4 and 1.5). A list element is tagged with a constructor that is usually named 'Cons' (prefix notation) or ':' (infix notation, as used in Miranda). The end of a list is an empty list that just contains a tag, the constructor 'Nil' or '[ ]'. A non-empty list element contains, besides the constructor 'Cons' (or ':'), a value of a certain type *T* and (a reference

---

[1] Note that in most languages all elements of a list have to be of the same type. In some other languages (e.g. LISP) the types of the list elements may differ from each other. In Miranda tuples are used for this purpose (see Chapter 2).

to) the rest of the list of that same type *T*. A *list of elements of type* T is denoted as [T].

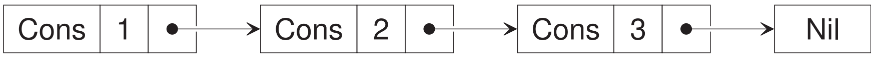

Figure 1.4 A list prefix tagged with the constructors Cons and Nil.

Hence, in Miranda, a non-empty list element contains the infix constructor ':', while the end of the list only contains a '[ ]'.

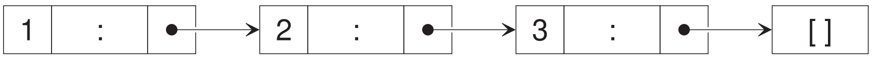

Figure 1.5 Miranda lists are infix tagged with the constructors : and [ ].

1.6.1 Denotation of lists

In Miranda, lists are denoted as follows.

Denotation of lists:

```
1 : (2 : (3 : (4 : (5 : [ ]))))      || list of numbers from 1 up to 5
True : (False : (False : [ ]))       || list of booleans
[ ]                                  || denotes the empty list
1 : 2 : 3 : 4 : 5 : [ ]              || list of numbers from 1 up to 5
```

Note that the list constructor ':' is right associative. For the convenience of the programmer Miranda allows a special notation for lists, with square brackets:

Lists, shorthand notation:

```
[1, 2, 3, 4, 5]                      || same as 1 : 2 : 3 : 4 : 5 : [ ]
[True, False, False]                 || same as True : False : False : [ ]
0 : [1, 2, 3]                        || same as [0, 1, 2, 3]
```

1.6.2 Predefined functions on lists

The data constructors ':' and '[ ]' are special constant values. The elements of a list can therefore easily be selected by using the pattern match mechanism in which these data constructors appear.

The (usually predefined) projection functions on lists are head (hd) and tail (tl). Projection functions like these are of general use and can be defined on lists of any type (see Chapter 2). The functions given below are restrictively defined on list of numbers.

```
hd:: [num] -> num                      || hd is a function from list-of-num to num
hd (first : rest)     =  first         || yields the first element of a non-empty list

tl:: [num] -> [num]                    || a function from list-of-num to list-of-num
tl (first : rest)     =  rest          || returns the tail of the list
```

Note that the parentheses on the left-hand side are not part of the list denotation. They are added to disambiguate the patterns concerning the application.

This way of handling lists is very user-friendly. One does not need to worry about 'memory management' nor about 'updating pointers'.

Below some other predefined operations on lists are given. These are: *length of a list* (denoted by #), *subscription,* followed by a number indicating the subscript (counting from zero), and *concatenation* (++).

List operations:

```
# [2, 3, 4, 5]                         || length of list, yields 4
[2, 3, 4, 5] ! 2                       || subscription, yields 4
[0, 1] ++ [2, 3]                       || concatenation, yields [0, 1, 2, 3]
```

1.6.3 Infinite lists

One of the powerful features of lazy FPLs is the possibility of declaring infinite data structures by recursive definitions.

Definition of a function yielding an infinite list of numbers n, equal to [n, n, n, n, …]:

```
infnums:: num -> [num]
infnums n   =   n : infnums n
```

The following program, the sieve of Eratosthenes, yields an infinite list of prime numbers. First an infinite list of all numbers [n, n+1, n+2, …] is defined using the function gen.

```
gen:: num -> [num]
gen n     =   n : gen (inc n)
```

filter has two arguments: a number and a (possibly infinite) list. The filter removes all multiples of the given number pr from the list. Note the type of this function. It is explained in the next section.

```
filter:: num -> [num] -> [num]
filter pr (x : xs)    =  x : filter pr xs,  if  x mod pr ~= 0
                      =  filter pr xs,      otherwise
```

sieve yields an infinite list of primes. It is assumed that the first element in the infinite list is a prime. This value is delivered as result in the head of a list. The function recursively calls itself, filtering all multiples of the found prime from the input list.

```
sieve:: [num] -> [num]
sieve (pr : rest)   =  pr : sieve (filter pr rest)
```

The initial expression that will yield the infinite list of all prime numbers [2, 3, 5, 7, 11, …] is the following:

```
sieve (gen 2)
```

Programs having infinite data structures as results do not terminate, of course. When an infinite list is yielded, the lazy evaluation scheme will force the evaluation of the list elements from 'left to right'. It would not be wise to postpone printing this list until the last element has been calculated. Therefore, one after another the values of the elements of the list are printed as soon as they have been evaluated.

Infinite data structures are of more practical importance than one may conclude from the examples above. One has to remember that, if functions are evaluated with the lazy evaluation scheme, the computation of a value is only started when its value is needed to produce the result. For instance, if one needs a certain element of an infinite list, the computation will not force the evaluation of the whole list, but only of the part that is needed for the evaluation of the required element. Therefore, such a computation can be performed in finite time, even if an infinite list is being used. Of course, if an attempt to find the last element of an infinite list is made, the computation will not terminate.

So in *lazy* languages *infinite* data structures can be defined; moreover, they make elegant programs possible. For example, if a function has to be defined that yields the first thousand prime numbers, one can simply take the first thousand elements of the list of all primes.

The following initial expression will yield the second prime number of an infinite list of all primes. The evaluation will terminate even if an infinite list is being used.

```
hd (tl (sieve (gen 2)))
→     hd (tl (sieve (2 : gen (inc 2))))
→     hd (tl (2 : sieve (filter 2 (gen (inc 2)))))
→     hd (sieve (filter 2 (gen (inc 2))))
→     hd (sieve (filter 2 (inc 2 : gen (inc (inc 2)))))
→     hd (sieve (filter 2 (3 : gen (inc (inc 2)))))
→     hd (sieve (3 : filter 2 (gen (inc (inc 2)))))
```

```
→    hd (3 : sieve (filter 3 (filter 2 (gen (inc (inc 2))))))
→    3
```

Infinite data structures cannot be handled in *eager* languages, because they evaluate arguments regardless of whether they are needed or not, which leads to infinite computations.

The evaluation of the initial expression in the previous example in eager FPLs will not terminate.

```
hd (tl (sieve (gen 2)))
→    hd (tl (sieve (2 : gen (inc 2))))
→    hd (tl (sieve (2 : gen 3)))
→    hd (tl (sieve (2 : 3 : gen (inc 3))))
→    hd (tl (sieve (2 : 3 : …)
```

In a lazy functional programming language most functions that are defined on lists can also be used to handle infinite lists. For instance, the hd function (defined above) can take the head of all lists, regardless of whether the list is finite or infinite.

1.7 Higher order functions and currying

Compared with the traditional imperative languages, which normally allow also the declaration of functions, functional programming languages such as Miranda have a more general view of the concept of functions: they are treated as 'first-class citizens', i.e. functions are treated just like other objects in the language. As a consequence, functions can have *functions* as arguments and as results.

1.7.1 Higher order functions

Functions that have functions as actual arguments or yield a function as result are called **higher order functions**, in contrast to **first-order functions,** which have only non-function values as argument or as result.

Example of a higher order function that takes a function as argument (note how this is reflected in the type):

```
atzero:: (num -> num) -> num
atzero f =  f 0
```

Now consider the following initial expressions (inc, square and ident are taken to be defined as in Section 1.3):

```
atzero inc      →  inc 0     →  0 + 1  →  1
atzero square   →  square 0  →  0 * 0  →  0
atzero ident    →  ident 0   →  0
```

Example of a higher order function that yields a function as result (note the type):

```
funcresult:: num -> (num -> num)
funcresult 0   =  inc
funcresult 1   =  square
funcresult n   =  ident
```

And consider the following initial expressions:

```
funcresult 0 6  →  inc 6     →  6 + 1  →  7
funcresult 1 6  →  square 6  →  6 * 6  →  36
funcresult 2 6  →  ident 6   →  6
funcresult 3 6  →  ident 6   →  6
```

At first sight the substitutions in the example above may seem a bit strange. But, remember that the application of a function f to an expression a is denoted as $f\ a$. So if a function is applied to more than one argument, say n, this can be denoted as:

$$(\dots ((f\ a_1)\ a_2) \dots a_n)$$

Function application is left associative. So this can be denoted in an equivalent, more readable way:

$$f\ a_1\ a_2 \dots a_n$$

Hence, the expression funcresult 0 6 is equivalent to (funcresult 0) 6, which immediately explains why the substitution above is allowed. Higher order functions are very natural and provide a powerful programming tool, as is shown below.

1.7.2 Currying

The possibility of yielding a function as result makes it unnecessary to have functions with more than one argument. A function with n arguments can be simulated by a higher order function with *one* argument that returns a new function. Now this new function can be applied to the *next* argument, and so on, until finally *all* n arguments are handled.

The idea of handling functions with n arguments by using a sequence of applications of higher order functions with one argument is called **currying** (Schönfinkel, 1924; Curry and Feys, 1958), see also

Chapter 3. The final result of such a sequence of applications of higher order functions with one argument is the same as the result yielded by the equivalent function with *n* arguments.

Consider a function definition, in Miranda, of a function with more than one argument, say n. Assume that these arguments are, respectively, of type A_1, A_2, …, A_n and that the result of the function is of type B. In general such a function definition has the following form:

```
function-name arg1 arg2 ... argn  =  expression
```

In Miranda, all functions are used in a curried manner. Functions with more than one argument are simulated by a sequence of applications using currying. Therefore the definition above should be read as:

```
( ... ((function-name arg1) arg2) ... argn)  =  expression
```

The type of this function is not

```
function-name:: A1 x A2 x ... x An -> B
```

which is the mathematical type with as domain the Cartesian product of the n argument types, but

```
function-name:: A1 -> (A2 -> ... -> (An -> B) ... )
```

The arrow -> is defined to be right associative, so the type can also be specified as

```
function-name:: A1 -> A2 -> ... -> An -> B
```

1.7.3 The use of currying

The question arises: why should one use this currying scheme and make things more complicated? Well, notationally it is almost equivalent to having functions with more than one argument: function application is left associative, so the parentheses can be left out. Currying enables the use of a familiar notation; only the types of the functions are different. But a great advantage of currying is that it increases the expressive power of a functional language.

Parametrizing functions

The currying scheme makes it possible to apply a function defined on *n* arguments to any number of arguments from 1 up to *n*, resulting in a *parametrized* function. A function defined with *n* arguments can, without currying, only be applied when all these *n* arguments are available.

In the curried variant some of the arguments can be 'fed' to the function; the result (a **parametrized function**) can be passed to another function that can fill in the remaining arguments.

Consider the following definition:

```
plus:: num -> num -> num
plus x y  =  x + y
```

This definition is equivalent to:

```
plus:: num -> (num -> num)
(plus x) y   =  x + y
```

Now take the following initial expression:

```
plus 1
```

Clearly, we need another argument to be able to actually perform the addition specified in the function body. The result of this initial expression is a function of type num -> num. It is a function of one argument; the function has no name.

Assuming plus to be defined as above, the function incr can be defined in two ways: as a function with an explicit formal argument:

```
incr:: num -> num
incr x     =  plus 1 x
```

or as a parametrized version of plus:

```
incr:: num -> num
incr   =  plus 1
```

The only difference between the two incr functions is that the second definition is more concise.

It is very useful to be able to create new functions by adding parameters to more general functions. So currying enables a new way of programming by elegantly specifying general purpose functions.

Another example of parametrized functions. The function map takes a function of type num -> num, a (possibly infinite) list of numbers and applies the given function to all numbers in the list. incr and plus are assumed to be defined as above.

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

A function that increments the elements of a list can now simply be constructed as follows:

```
mapincr:: [num] -> [num]
mapincr     =  map incr
```

or as

```
mapincr:: [num] -> [num]
mapincr     =  map (plus 1)
```

In Miranda it is, in general, not the intention to yield a function as the final result of a program. Functions cannot be printed. The intention is to yield some non-function value, probably a string of characters, as the final result. Therefore, in ordinary programs curried functions will in the end receive all arguments needed to do the computation.

The following example of an expression with mapincr as defined above (the second definition) shows how evaluation might actually proceed when currying is used.

```
mapincr [1,2,3]
→     map (plus 1) [1,2,3]
=     map (plus 1) (1 : 2 : 3 : [ ])
→     plus 1 1 : map (plus 1) (2 : 3 : [ ])
→     2 : map (plus 1) (2 : 3 : [ ])
→     2 : (plus 1) 2 : map (plus 1) (3 : [ ])
→     …     →     2 : 3 : 4 : [ ] = [2,3,4]
```

1.8 Correctness proof of functional programs

A functional programming style has advantages that are common in any mathematical notation:

- There is consistency in the use of names: variables do not vary, they stand for a, perhaps not yet known, constant value throughout their scope.
- Thanks to the absence of side-effects, in FPLs the same expression *always* denotes the same value. This property is called **referential transparency.**

A definition like x = x + 1 would mean in an imperative language that x is incremented. In a functional language however this definition means that *any* occurrence of x can always be substituted by x + 1. Clearly the initial expression x does not have a normal form: substitution will continue forever (some systems recognize such situations in certain simple cases and report an error message).

$\underline{x} \quad \rightarrow \quad \underline{x} + 1 \quad \rightarrow \quad (\underline{x} + 1) + 1 \quad \rightarrow \quad \ldots$

Due to the referential transparency[1] one is allowed to replace in *any* expression *any* occurrence of *any* left-hand side of a definition by its corresponding right-hand side and vice versa.

Referential transparency makes it possible to use common mathematical techniques such as *symbolic substitution* and *induction*. With the help of these techniques a program can be transformed into a more efficient one or certain properties of a program, such as its correctness, can be proven. Take again the factorial example:

```
fac:: num -> num
fac 0    =  1                                              || (1)
fac n    =  n * fac (n – 1),                    if  n > 0  || (2)
         =  error "factorial called with argument less than zero", otherwise
```

The Miranda definition of this function has a great similarity to the mathematical definition of factorial. In order to prove that this function indeed calculates the mathematical factorial written as n! for n >= 0, mathematical induction is used: first it has to be proven that the function calculates factorial for a start value n = 0 (step 1). Then, under the assumption that the function calculates factorial for a certain value n (the induction hypothesis), it has to be proven that the function also calculates factorial for the value n + 1 (step 2). The proof is trivial:

```
step 1:  fac 0 = 1                          , by applying rule (1)
         1  =  0!                           , by the definition of factorial

step 2:  Assume that fac n = n!, n >= 0  (induction hypothesis)
         fac (n + 1) = (n + 1) * fac n      , by applying rule (2): n+1>0
         (n + 1) * n!                       , by the induction hypothesis
         (n + 1)!                           , by the definition of factorial
```

[1] Sometimes the notion ‘functional programming language’ is used for languages which support higher order functions and the notion ‘applicative programming language’ for languages which support referential transparency. Outside the functional programming community the notion ‘functional’ is widely used as a synonym for ‘useful’.

Note that the proof assumes, as is common in mathematics, that the function is only applied to arguments on which it is defined: the mathematical definition does not concern itself with 'incorrect input' (n < 0). In general, a proof must cover the complete domain of the function.

There is a strong correspondence between a recursive definition and an induction proof. Recursive functions generally have the form of a sequence of definitions: first the special cases (corresponding to the start values in an induction proof), textually followed by the general cases that are recursively expressed (corresponding to the induction step).

1.9 Program examples

This section illustrates the expressive power of the functional programming languages in two small examples.

1.9.1 Sorting a list

The function (quick) sort needs a list of numbers as argument and delivers the sorted list as result.

```
sort:: [num] -> [num]
sort [ ]   =  [ ]
sort (x : xs)     =  sort (smalleq x xs) ++ [x] ++ sort (greater x xs)
```

The functions smalleq and greater take two arguments, namely an element and a list. This element must be of the same type as the type of the elements of the list. It is assumed that the elements are of type num; the operators <= and > are therefore well defined.

```
smalleq:: num -> [num] -> [num]
smalleq a [ ]          =  [ ]
smalleq a (x : xs)     =  x : smalleq a xs,  if  x <= a
                       =  smalleq a xs,      otherwise

greater:: num -> [num] -> [num]
greater a [ ]          =  [ ]
greater a (x : xs)     =  x : greater a xs,  if  x > a
                       =  greater a xs,      otherwise
```

1.9.2 Roman numbers

Outline of the problem

Roman numbers consist of the characters (roman ciphers) M, D, C, L, X, V and I. Each of these characters has its own value. The values of roman ciphers are: M := 1000, D := 500, C := 100, L := 50, X := 10, V := 5, I := 1.

These characters always occur in sorted order, characters with a higher value before characters with a lower value. Exceptions to this rule are a number of ‘abbreviations’, given below. The value of a roman number can be found by adding the values of the characters that occur in the roman number (MCCLVI = 1000 + 100 + 100 + 50 + 5 + 1 = 1256). The following abbreviations are commonly used: DCCCC := CM, CCCC := CD, LXXXX := XC, XXXX := XL, VIIII := IX, IIII := IV. These abbreviations make it less simple to calculate the value of a roman number because now the value of the character depends on its position in the string. Negative numbers and the number zero cannot be expressed in roman numbers.

Task

- Develop an algorithm that calculates the integer value of a roman number, represented as a string, assuming that the string is a proper roman number.
- Develop an algorithm that converts an integer value into a roman number without abbreviations, assuming that the integer value is positive.

Solution

First the value of a roman cipher is defined:

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

The function romtonum converts a roman number to a decimal number. It assumes that the supplied argument is a proper roman number.

```
romtonum:: [char] -> num
romtonum ('C' : 'M' : rs)   =  value 'M' – value 'C' + romtonum rs
romtonum ('C' : 'D' : rs)   =  value 'D' – value 'C' + romtonum rs
romtonum ('X' : 'C' : rs)   =  value 'C' – value 'X' + romtonum rs
romtonum ('X' : 'L' : rs)   =  value 'L' – value 'X' + romtonum rs
romtonum ('I' : 'X' : rs)   =  value 'X' – value 'I' + romtonum rs
romtonum ('I' : 'V' : rs)   =  value 'V' – value 'I' + romtonum rs
romtonum ( r : rs)          =  value r + romtonum rs
romtonum [ ]                =  0
```

The function numtorom converts a decimal number to a roman number by repeated subtraction of 1000, 500, 100, 50, 10, 5 and 1, in this order. It assumes that the argument is a number greater than zero.

```
numtorom:: num -> [char]
numtorom n   =  countdown n ['M','D','C','L','X','V','I']

countdown:: num -> [char] -> [char]
countdown 0 rs         =  [ ]
countdown n (r : rs)   =  [r] ++ countdown (n – value r) (r : rs), if  n >= value r
                       =  countdown n rs,                         otherwise
```

The style in which these algorithms were presented (bottom-up) is not the way in which they were deduced. In Chapter 2 styles of functional programming are discussed.

Summary

- A *functional program* consists of a collection of (predefined) *function definitions* and an *initial expression* that has to be evaluated according to these definitions.
- A *function definition* consists of a sequence of one or more alternative *equations*. The choice between these alternatives is determined by *patterns* and/or *guards*.
- The evaluation of a functional program is called *reduction* or *rewriting*.
- A function application that can be rewritten according to a function definition is called a *redex*.
- In each *reduction step* a redex is replaced by the corresponding function body, *substituting* formal arguments by actual arguments.
- The evaluation of a functional program stops if there are no more redexes left in the resulting expression. Then the expression is in *normal form.*
- Redexes can be chosen in *arbitrary* order; in principle it is even possible that they are reduced in parallel.
- A *reduction strategy* determines the order in which redexes are reduced.
- In a *lazy* FPL redexes are only chosen if their result is needed to achieve the normal form.
- In an *eager* FPL the arguments of a function are reduced to normal form before the function application itself is reduced.

- *Data structures* in FPLs are not globally available but constructed in expressions, passed as arguments, decomposed and used as components for the construction of new structures.
- Data structures in FPLs are composed using *data constructors*. With pattern matching data structures can be decomposed.
- The most important basic data structure in FPLs is a *list*.
- Lazy FPLs can handle *infinite* data structures, eager FPLs cannot.
- A function with *n* arguments can be simulated using higher order functions with at most *one* argument (*currying*).
- FPLs have a high expressive power due to the availability of higher order functions, pattern matching and guarded equations.
- Due to *referential transparency* traditional mathematical proof techniques, such as *induction* and *substitution*, can be used for the correctness proof of functional programs.

EXERCISES

1.1 Write in your favourite imperative language a program that finds the maximum element in a list of numbers. Rewrite the program so that only the functional subset of the imperative language is used.

1.2 Consider the following function definition:

```
maxi:: num -> num -> num
maxi x y    = x,   if  x >= y
            = y,   if  x < y
```

Check whether the following initial expressions are legal, and if so, give the result yielded:

- maxi 5 6
- maxi (5)
- maxi 5
- maxi (5,6)
- maxi maxi 5 6 4
- maxi [5,6]
- maxi (maxi 5 6) 4
- maxi 'a' 4

1.3 Write a function maxilist that uses the function maxi of Exercise 1.2 to calculate the *greatest* element of a list of numbers. How does maxilist react on the empty list? Are there lists for which your program does not terminate? Does it make any difference if you change the order of the guards or patterns in your program?

1.4 Suppose you now want to find the *smallest* element of a list with minimal rewriting of the programs written in Exercise 1.3. To solve this problem first write a higher order function find in Miranda that gets a list of numbers and a function f of type num -> num

-> num as arguments and produces a num as its result such that find $[a_1, a_2, \ldots, a_n]$ f = f a_1 (f a_2 (… (f a_{n-1} a_n))).

Then, write down the stepwise lazy evaluation of the function application find [1,2,3] s with s:: num -> num -> num and s a b = a + b. What does find list s calculate?

Finally, write a function to find the smallest element of a list.

1.5 Define functions that print a list of characters on the screen in different ways: vertically below each other; each character in the same place; with two spaces between the characters; diagonally.

1.6 Define a function which yields the last two elements in a list.

1.7 Define a function which yields the average of a list of numbers.

1.8 Define a function search that takes two arguments of type [char] yielding a bool indicating whether the first argument is part of the second.

1.9 Examine the definition of the function map in this chapter. Define a function mapfun that applies a list of functions to a number and returns a list of the results of the function applications. So the following must hold: mapfun [f,g,h] x = [f x, g x, h x].

1.10 Define a function that yields all factorial numbers.