1

FUNCTIONAL PROGRAMMING AND COMPUTATIONAL MODELS

In this introductory chapter an overview is given of the field of models of computation with respect to functional programming languages. The importance of defining a model of computation is explained. The correspondence between paradigms in programming languages and computational models is discussed. The importance of functional languages is addressed. Furthermore some basic models of computation which are used traditionally for describing the behaviour of implementations of functional languages, are introduced very briefly.

More information on paradigms, functional languages, computational models, graph rewriting, implementation methods, intermediate languages, compilation schemes and abstract and concrete machine architectures can be found in Plasmeijer & van Eekelen (198-).

1.1 PARADIGMS OF PROGRAMMING: LANGUAGES AND MODELS

LANGUAGES

A language is a programming language if its syntax and semantics are formally defined and implementable. Such a formal definition can be based on any formal description method e.g. denotational, algebraic, categorical or operational. In particular, a compiler and an interpreter are themselves formal definitions of the semantics, where another programming language is used as the formal operational description method. In the case of a compiler generally even two other programming languages are used. The basis of all implemented formal descriptions of semantics is the operational semantics of the machine language which by itself is realized in hardware via a particular machine architecture.

Every program which accepts input induces its own special purpose programming language. Some examples of special purpose languages are the set of commands for an editor, the Unix shell and a data base query language. Usually no model of computation is given for a special purpose language. The meaning of such languages is mostly defined via user manuals.

General purpose languages do not focus on a specific (class of) algorithm(s). Being general purpose they have a general view on the world. Such a view can be formalized yielding a model of computation of the language.

MODELS

A general purpose programming language is usually composed out of many language constructs. Examining the semantics of a language carefully it is possible to classify these constructs: some of them can be regarded as the basic concepts of the language, while others are purely syntactic sugar added to the language for programming convenience or for software engineering reasons. In order to understand the facilities offered by a language it is important to know what the essential language constructs are and what they mean. It is sufficient to examine the semantics of these essential language constructs since the semantics of other language constructs can be expressed in terms of the basic constructs by removing syntactical sugar. A computational model tries to capture the essential aspects of a programming language in a formal model in order to be able to reason about them. However, in principle for a specific language there are many computational models which can be used. For instance, any deterministic computation can be expressed on a Turing Machine (Turing (1936)). But this will not greatly simplify the reasoning about the correctness of a particular program written in a specific language because the Turing machine generally is too far away from the semantics of a specific language.

Furthermore, a computational model tries to capture the essential aspects of (a class of) implementations of the programming language.

A model of computation tries to capture the essential aspects of a language by making some abstractions. A *model of computation* (or a *computational model*) of a programming language is a formal model as close as possible to both semantics and implementation still modelling only the essential aspects of them. If such a model of computation is known it is much easier to reason about the correctness of specific programs, the essential properties of the language, the expressive power, the orthogonality of the design, the implementation methods for a given computer architecture and the design constraints for new architectures to support the language.

It is of course very difficult to find such an ideal model which models every essential aspect of a programming language. In fact the classical (sequential and imperative) programming languages all have the same model of computation: the Turing machine. Clearly this model can only describe the very basic concepts of those languages. Lately, newly defined programming languages all come with a model of computation. The new language and the model of computation are closely related: either the language is built on top of the model or the other way around or sometimes they are even developed together. When defining a model of computation for a language one often has to choose between what does the language mean (the semantical aspects) and how is it done (the behavioural aspects). Many times this leads to a language having both a denotational and an operational semantics. These two semantics may be based on different models of computation which together fully specify what the language is about.

PARADIGMS

A *paradigm* in programming is an intuitive view on what the essence of programming is. In the computer science community the traditional imperative paradigm has prevailed for many years. Until recently, proposals for new programming languages based on other paradigms did not really catch on because the implementation techniques to translate them efficiently to the imperative machine architectures were not developed enough. The most important proposed new paradigms for programming languages are object-oriented, dataflow and declarative. In the following paragraphs each of the paradigms will be discussed.

Imperative

Von Neumann machine architectures (Burks *et al.* (1946)) all use the imperative paradigm. In the imperative paradigm there is a memory containing data and this memory also contains an ordered sequence of commands (*instructions*) and a locus of control which indicates which instruction has to be executed next. The instructions may or may not change any part of the memory. For many years, using this paradigm implementation and programming techniques were developed, refined and optimized. Many imperative programming languages were introduced generally leading to higher levels of abstraction in programming. This trend towards higher levels of abstraction is enabled by improved implementation techniques and by increased computational power of the hardware.

An imperative program (written in Modula-2 (Wirth (1982)) calculating an array with the first `N` Fibonacci numbers:

```
VAR
   fib: ARRAY [0..N-1] OF INTEGER;
   i:   CARDINAL;
BEGIN
   fib[0] := 1; fib[1] := 1; i := 1;
   WHILE i < (N-1)
   DO
      i := i + 1;
      fib[i] := fib[i-1] + fib[i-2]
   END
END
```

The basic computational model for Von Neumann machine architectures is introduced by Turing in 1936: the Turing machine (Turing (1936)). This model of computation is used for complexity theory and other theoretical issues (e.g. the Halting Problem (Lewis & Papadimitriou (1981)).

However, there are some small problems with the computer systems based on the imperative paradigm. It is hard to find a piece of software without any bugs. Computer scientists have learned to live with the *software crisis*, and they accept that most software products are unreliable, unmanageable and unprovable. Although hardware is much more reliable than software, most hardware systems appear to be designed in a hurry and even well-established processors now and then go down because of undocumented race-conditions. Clearly software and hardware systems have become very complex. So it seems to be understandable that these systems contain bugs. A good, orthogonal design costs many, many, man years of research and development. The good news is that hardware becomes cheaper and cheaper (thanks to the Very Large Scale Integration) and speed can be bought for prices never dreamed of. Of course it never goes fast enough. Yet it seems that the maximum speed that can be obtained with present day technology in Von Neumann architectures is beginning to reach its limit.

The two key problems that the computer science community has to solve are how to make reliable and user-friendly software at low costs and how to increase processing power at low costs. Researchers are looking for solutions for these problems: investigating software engineering techniques, to deal with problems related to the construction of very large software programs; designing new proof techniques to tackle the problems in proving the correctness of systems; developing program transformation techniques, to transform the specification of a problem to a program which solves it, designing new (parallel) computer architectures using many processors (up to thousands or more) to increase execution speed.

One approach which eventually may help to find a solution for the key problems, is based on the idea that these problems are fundamental problems which cannot be solved unless a totally different approach is taken and hardware and software is designed with a completely different model of computation in mind. We believe that this idea is true and that the solution lies in non-imperative paradigms.

An imperative programming style has the following drawbacks.

- It consists of a sequence of commands of which the dynamic behaviour must be known in order to understand how such a program works. Particularly the assignment causes problems, because it changes the value (and often the meaning) of a variable. Evaluating the same expression in succession may produce different answers. Reasoning about the correctness of an imperative program is therefore very difficult .
- In addition, due to the relatively low expressive power of the present high-level programming languages, programs become large and therefore hard to understand.
- Because of the command sequence, algorithms are more sequential than necessary. Therefore it is hard to detect which parts of the algorithm can or cannot be executed concurrently.

So, we believe that the software crisis and the speed problem are inherent to the nature of imperative programming languages and the underlying model of computation.

Object-oriented

The introduction of multiprocessing on a single machine, led to a new paradigm for parallel languages, which is also used for programming parallel machines:

In the *object-oriented* (or *actor-based*) paradigm a program describes the behaviour of a system in terms of its constituents, the objects. In object-oriented programming languages each object has some internal data and the ability to act on (change) these data. Objects may have an internal activity of their own. Objects only interact by sending messages to each other. One could say that each object is a process controlled by an imperative subprogram communicating via message passing with other processes.

An object-oriented program (written in POOL2 (America (1988))) that generates all prime numbers using the sieve method:

```
IMPL UNIT Sieve
USE File_IO
GLOBAL driver := Driver.new ()

CLASS Driver
VAR first := Sieve.new ()
BODY FOR i FROM 2 DO first ! input (i) OD
YDOB
END Driver

CLASS Sieve

VAR myprime, current : Int
    next : Sieve

METHOD input (n : Int) : Sieve
BEGIN current := n;
      RESULT SELF
END input

BODY ANSWER (input);
     myprime := current; %% the first input is a prime number
     standard_out ! write_Int (myprime, 0) ! new_line ();
     next := Sieve.new ();
     DO ANSWER (input);
        IF current // myprime ~= 0
        THEN next ! input (current)
        FI
     OD %% forever
YDOB
END Sieve
```

The main parts of the program are the class `Driver` of which there is only one object (created with `Driver.new ()`) and the class `Sieve` of which there is in principle an infinite number of objects (every sieve starts a new sieve). The `driver` generates all natural numbers greater than 2. The class `Sieve` contains a method `input` which if called (by other objects) and answered saves the argument in a local variable. Every `Sieve` has an activity of its own which is defined in the body. Its prime number is printed on the standard output file. It starts a new `Sieve` and it loops forever sieving out multiples of its primenumber.

The most commonly used basic computational model for object-oriented machine architectures is the Calculus of Communicating Systems (Hoare (1985)). This model of computation is used for complexity theory and other theoretical issues (e.g. fairness, proof theory). Process Algebra (Baeten (1986)) might be an alternative when one is more interested in describing the behaviour of processes.

Modularisation is an important technique which is used in the development of large programs. In the object-oriented paradigm a program is modularised by the introduction of the objects. One can imagine that in parallel machines different objects might live on different processors.

The activity of a single object is usually defined in an imperative language. In that case the object-oriented language essentially inherits the disadvantages of the imperative paradigm. In chapter 7 of this thesis we present some promising fundamental research in combining the object-oriented view with functional languages.

Dataflow

In the *dataflow* paradigm the data moves to the instructions instead of the other way around. An instruction may start executing as soon as its data is available. So, the flow of data determines the execution order. This approach has gained a lot of interest because of envisaged fast parallel hardware designs.

Dataflow languages or *single assignment* languages are languages in which variables are assigned only once: i.e. a variable is undefined until it gets a value and then it can never change afterwards. Single assignment languages are made to fit the dataflow model of computation.

A dataflow program (written in SISAL (Glauert (1978)) calculating an array with the first `N` Fibonacci numbers:

```
fibnumbers :=
   FOR
      INITIAL
         fib1 := 1; fib2 := 1
      REPEAT
         fib1, fib2 := OLD fib2, OLD fib1 + OLD fib2
      WHILE
         fib2 < N
      RETURNS ARRAY OF fib2
   END FOR
```

Also imperative and functional languages can be translated more or less efficiently to dataflow languages (Veen (1985)). It seems that functional languages are better suited to be efficiently translated to dataflow than imperative languages.

To exploit the advantages of the dataflow model special machine architectures are necessary which realize the inherent parallelism. Unfortunately, actual dataflow architectures (Gurd *et al.* (1985), (Arvind *et al.* (1987)) are very complex and not yet commercially available.

Declarative

In the *declarative* paradigm a desired computation is expressed in a static fashion as a list of declarations and an expression to be evaluated. A program is considered to be an executable specification.

A declarative language has the following advantages which are common in any mathematical notation.

- Mathematics is static (no assignments, side effects): a function will always give the same answer if it is applied on the same arguments.
- There is consistency in the use of names (like in $x^2 - 2x + 1$). Variables do not vary, they stand for a, perhaps not yet known, constant value throughout their scope.
- An expression always has the same meaning independent of the history of the computation (*referential transparency*).
- Because equal expressions are always and everywhere interchangeable, declarative languages are convenient to reason about.

The class of languages using the declarative paradigm can be split up in grammar, logic and functional languages.

Grammar languages

Grammar languages are based on the idea that a program essentially just parses its input and produces a result accordingly. The programmer specifies in a grammar what the input may look like and which semantic actions must be taken. In the field of compiler development these languages have lifted software development to a higher level and they have greatly increased the programming productivity.

A grammar program (written in enhanced Extended Affix Grammars (Meijer 1986)) that appends two lists into another list:

```
append (>empty, >list, list>): ;
append (>elt⊕list1, >list2, elt⊕list3>): append (>list1, >list2, list3>).
```

The > signs define the flow of the arguments: before the argument they denote an input parameter, after the argument they denote an output parameter.

Although all grammar languages are based on grammars they do not all have the same computational model. The kind of grammars that are used, identifies the computational model: e.g. affix grammars (Koster (1971)) or graph grammars (Nagl (1979)).

Grammar languages originated as special purpose languages but it has been proven to be worthwhile to investigate whether they can also serve as general purpose languages. Problems occur in userfriendliness, debugging facilities and programming environments.

Logic languages

Logic languages are based on the idea of a program being a set of rules which state that predicates are true if certain conditions are met. These rules are multi-directional. So, you can ask whether a predicate is true with certain values for specific logical variables but you can also ask for which values of these variables the predicate holds.

A logic program (written in PROLOG (Clocksin & Mellish (1984))) that appends two lists into another list:

```
append ([], List, List).
append ([Elt|List1], List2, [Elt|List3]):- append (List1, List2, List3).
```

The logic program is very much like the grammar program. The main difference is the multi-directionality of the logic program. The logic program will give a result for the first parameter if the other two are specified but it will also give a result for the third parameter if the first two are specified. This difference is perhaps not really essential but it is typical for the general view on programming.

In logic languages evaluation means investigating whether a formula in first-order predicate logic can be true. The evaluation method uses unification and is based on the resolution method of Robinson (Robinson (1965)). In fact the languages are quite close to the model by contrast with the imperative languages and the Turing machine model. Logic languages can often be seen as a subset of the general model defined by certain restrictions and extended with some specific constructs for efficiency and software engineering purposes.

The Japanese Fifth Generation Computers project has had a great impact on the world-wide interest for logic languages. Logic languages are now widely used in the context of database systems. In the context of logic languages data is often called *knowledge*.

Functional languages

In a *functional language* every program is just a collection of function definitions. Each function can be seen as a kind of program which accepts input (its arguments) and produces output (its result). The concept of a function is one of the fundamental notions in mathematics.

Some functional programs (written in Miranda™* (Turner (1985)):

Fibonacci numbers:

```
fibs thisfib nextfib = thisfib : fibs nextfib (thisfib + nextfib)

take n (fibs 1 1)
```

A functional program for sieving prime numbers can be found in chapter 7.

Append two lists into another list:

```
append []           list = list
append (elt:list1) list2 = elt : (append list1 list2)
```

One of the greatest advantages of functional programming languages is that they are based on a sound and well understood mathematical model, the λ*-calculus* (Church (1932/1933)). In terms of denotational semantics one could say that functional programming languages are sugared versions of

*Miranda is a trademark of Research Software Limited.

this λ-calculus. This computational model is introduced more or less at the same time as the Turing model. The power of these models is the same (Turing (1937)). Beside the λ-calculus there are other related computational models which can be seen as a basis of functional programming languages, namely *combinatory logic* (Schönfinkel (1924), Curry (1930)) and *rewriting systems* (Klop (1987)). These computational models are very important because they have a great influence on the specification of the semantics of functional programming languages and on their implementations. In section 1.3 the traditional models are discussed and compared. In chapter 2 rewriting models will be discussed. Recently many researchers started to investigate sequential and parallel machine architectures especially suited for these models of computation. These architectures are called *reduction machines.*

The main advantages that are offered by functional languages, are a great expressive power, relatively easy correctness proofs and a relatively high suitability for parallel evaluation. Therefore functional languages are a very important research topic. An overview of the field of functional programming is given in Field & Harrison (1988).

1.2 FUNCTIONAL PROGRAMMING LANGUAGES

In Backus (1978) it is pointed out that the solution for the software problems should be searched in finding a new programming discipline. He proposed to investigate *functional programming languages*, also called *applicative programming languages* (sometimes the notion "functional" programming languages is used for languages which support higher order functions and the notion "applicative" programming languages for languages which have application as the overall basic concept; outside the functional programming community the notion "functional" is widely used as a synonym of useful).

To be able to program with functions a suitable rich set of basic functions has to be defined and then they have to be used to define new functions in terms of these. Thus a whole library of useful functions may be built. Some of those library functions are probably built upon layers of others.

The question arises, whether more is needed than just a repertoire of basic functions and the ability to combine them in order to define a library of functional programs? Fortunately, there is a mathematical thesis, known as the *Church's thesis* (Church & Rosser (1936)), which states that the class of *computable functions* is exactly the same as the class of *recursive functions*. This class of recursive functions is exactly the class of functions you can get by combining some basic functions via primitive recursion, composition and minimalisation.

In functional languages the programmer can only define functions which compute values uniquely determined by the values of their arguments. Consequently, many of the familiar concepts of conventional programming languages are missing in purely functional languages. Most important, assignment is missing. So is the heavily used programming notion of a variable, something which holds a value that is changed from time to time by an assignment. Rather, the variables that exist in purely functional language are used like in mathematics to name and refer to a yet unknown constant value. Once the value is known it cannot be altered anymore: in mathematics a variable does not vary.

FUNCTIONAL PROGRAMMING IN AN IMPERATIVE LANGUAGE

Perhaps a functional programming style is important, like avoiding goto's. But are new languages really needed? Imperative languages also have functions, so why not just use the functional subset of e.g. C, Algol or Modula. Well, even if only the functional subset of these languages would be used (this means leaving out the assignment) these languages are not as suitable as the new functional programming languages. The reason is that functions in the imperative language are often not treated as first-class citizens. This is a fact of life, not a fundamental problem. In some languages a function cannot be an argument of a function, only values can be arguments. In other languages functions cannot yield a function as result. Sometimes these restrictions are present because one did not know how to make an (efficient) implementation of such functions. Furthermore, the available type systems of the classical imperative languages makes it impossible to fill the gap without a complete redesign of these languages. Also nice features of functional programming languages such as infinite data structures (see the following section) are not possible in imperative languages because of the evaluation order which is used in these languages.

BASIC CONCEPTS OF FUNCTIONAL PROGRAMMING LANGUAGES

In this section the most common concepts of functional languages are introduced. When not explicitly stated otherwise, all examples are written in the functional programming language Miranda.

Function definitions

A program contains an expression to be computed and a collection of function definitions written in the form of recursive equations.

Some simple function definitions:

```
increase x    =    x + 1
square x      =    x * x
squareinc x   =    square (increase x)
constant      =    7
```

In the definitions of these functions, `x` is a *formal parameter* or *formal argument.* It is essentially like a *bound* variable in mathematical logic. Its scope is limited to the equation in which it occurs (whereas the other names introduced above have the whole program as scope).

The basic operation in functional programming languages is *function application*. Because function application is so fundamental in functional programming languages, the operator is generally not written down explicitly. Application is simply denoted by juxtaposition.

For example, in case the function `square` is applied to the value `3`, we simply write down:

```
square 3
```

In some languages however, application has to be written explicitly with a special binary function.

For example:

```
Apply (square, 3)
```

The expression on which a function is applied (in the example the value `3`) is called the *actual parameter* or *actual argument.* The notation where the application operator is hidden is called the *applicative style* and the notation in which the application is explicitly present as binary function is called the *functional style.*

As usual in programming languages, one can denote and manipulate objects of certain predefined types. The following basic data types, with appropriate basic operations, are usually available.

Data types:	**Operations:**	**Notation of constant values:**
numbers	+, -, *, ...	1, 34, 12, ...
truth values	and, or, ...	True, False.
characters	=, <, ...	'a', 'c', ...

The execution of a functional program simply consists of the evaluation of an initial expression in the context of the function definitions in the program. Evaluating means repeatedly performing *reduction* or *rewriting steps*. In each reduction step (indicated by an "→") a function application in the expression is replaced (*reduced, rewritten*) according to its definition (by the right-hand-side of the equation), substituting the formal arguments by the corresponding actual arguments. The subexpression that is rewritten is called a *redex* (*reducible ex*pression). The reduction process stops when none of the function definitions can be applied anymore (there are no redexes left). Then the initial expression is in its most simple form, the *normal form.* This result of the evaluation is printed.

For instance, given the function definitions above (the environment), the expression `squareinc constant` can be evaluated (reduced) as follows. The expression which will be rewritten, is underlined.

```
squareinc constant    →   square (increase constant)
                      →   square (constant + 1)
                      →   square (7 + 1)
                      →   square 8
                      →   8 * 8
                      →   64
```

Higher order functions

Compared with the traditional imperative languages, which normally allow also the declaration of functions, functional programming languages have a sound view on the concept of a function: functions are treated as first-class citizens.

A *first-order* function is a function which can only have basic types as argument and as result. A *higher order* function is a function which can have a function as argument or as result. Languages which support higher order functions in a general way are also called *higher order* languages. Functional languages are higher order languages.

The possibility to yield a function as result makes it unnecessary to consider functions with more than one argument. A function with n arguments can be constructed by a function with one argument that returns a function that can be applied to the next argument, and so on.

In general a function definition has the following form:

$\texttt{function-name arg}_1\ \texttt{arg}_2\ \texttt{...} = \texttt{expression}$

In these definitions each function actually has only one argument. The convention is used that function application is associative to the left, so the parentheses are left out. But actually this must be read as:

$\texttt{.. ((function-name arg}_1\texttt{) arg}_2\texttt{) ...} = \texttt{expression}$

This way of considering all functions as functions of one argument is called *Currying*. At first sight, Currying perhaps looks a bit strange, but keep in mind that the basic operation in functional programming languages is binary function application, which is hidden by the applicative notation using juxtaposition. With explicit application the definition would be:

$\texttt{.. Apply (Apply (function-name, arg}_1\texttt{), arg}_2\texttt{), ...} = \texttt{expression}$

Currying enables a familiar notation to be used, but furthermore it enables "the application of functions with any number of arguments", which gives an additional descriptive flexibility. This encourages a programmer to write "parametrized functions".

A simple Curried definition:

```
plus x y = x + y
```
|| the type of `plus` is: `num` → `num` → `num`. The → 's are right associative, so given an object of type `num`, the higher order function `plus` produces an object of type: `num` → `num`

So, also the following definition is allowed:

```
inc = plus 1
```
|| the type of `inc` is: `num` → `num`

This enables the following reductions:

<u>inc</u> 3 → <u>plus 1 3</u> → <u>1 + 3</u> → 4

Patterns

In functional programming languages functions are defined by a series of equations. In the left-hand-side of an equation one can specify that the equation in question can only be applied if the actual arguments of the function are of a certain shape. The execution mechanism must be given the "intelligence" to match actual parameters with the patterns used in the definition of a function so that it can decide which equation to use. Generally, the equations are tried in the textual order they are specified, from top to bottom. Within one equation the arguments are tried from left to right. The patterns may consist of expressions with variables (e.g. `n`, values (e.g. `0`) and data constructors (e.g. the infix `:`, which is used to represent a list; see also the next section on data structures). This facility, which is called *pattern-matching,* offers an alternative to conditional expressions. It often leads to clearer and more concise definitions.

Patterns and pattern-matching:

```
fac 0 = 1
fac n = n * fac (n - 1)

hd (a:b) = a
tl (a:b) = b

cond True  x y = x
cond False x y = y
```

'`0`' and 'n' are the patterns of the first two rules (a simple variable as pattern indicates that it does not matter what is on that position). Reducing '`fac 7`' will result in a call to the pattern-matching facility, which decides that only the second rule is applicable.

The functions `hd` and `tl` require a non-empty list as actual argument. The head of the list is mapped on `a`, the tail of the list is mapped on `b`. Again these variables indicate that the contents of the list are irrelevant.

A conditional can easily be defined. The choice between "then" and "else" part will depend on the actual value of the first argument as indicated by the pattern.

Data structures

Lists are a rather important data structure in many functional programming languages. Lists, like any data structure in a functional programming languages, are constructed with the help of so called *data constructors.* Such a constructor can be seen as a kind of *identification* or *tag* which uniquely identifies (and is part of) a "record" of a particular type. The *list-constructor* is usually named "`Cons`" (prefix notation) or "`:`" (infix notation). The type of a list could recursively be defined as (usually it is predefined):

```
list *   ::= Cons * (list *)      | Nil
```

List-types are denoted by `list *` or `[*]`. A list contains two elements and the list-constructor `Cons`, or, a list is an empty list denoted by `Nil` or `[]`. If the list is non-empty it contains a list element of a certain type `*` as head while the tail of the list is again a list of the same type. One could say that the head and the tail are "glued" together with help of the list-constructor. Lists are declared either by enumeration or by (recursive) definitions. The syntax we use for lists, is illustrated by the following examples.

Lists, explicit infix notation:

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

For convenience of the programmer there is a shorthand notation for lists, using square brackets.

Lists, shorthand notation:

```
[1, 2, 3, 4, 5]             || the same list of numbers as above
[True, False, False]        || again the same list with 3 booleans
0 : [1, 2, 3]               || same as [0, 1, 2, 3]
```

One of the powerful features of functional programming languages is the possibility to declare infinite data structures via recursive definitions.

Definition of an infinite list equal to [1, 1, 1, 1, ...]:

```
ones = 1 : ones
```

`ones` is a recursive function yielding an infinite list.

Programs having infinite data structures as result do not terminate of course. For practical reasons such a result is printed as soon as possible. For an infinite list this means that, from "left to right", the elements of the list are printed as soon as they are in normal form.

The possibility of defining infinite data structures puts some restrictions on the evaluation strategy which is explained in the following section.

Some predefined operations on lists are generally available. These are: length of a list (denoted by #), subscription (!) and concatenation (++). The operations are assumed to be predefined in a library for reasons of efficiency and convenience.

Predefined list-operations:

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

Evaluation

As explained in the previous section, the evaluation of a functional program has the intention to find a final non-reducible expression denoting the same value as the initial expression E by repeatedly performing reduction steps. Because there are in general many redexes in the expression to reduce, one can perform these steps in several ways. This is determined by the so called *reduction strategy* which controls the evaluation. A reduction strategy is sometimes also called an *order* because it is used to indicate the next redex to reduce. There are a couple of important things to know about the ordering of reduction steps.

In functional languages the final result of the computation (the normal form) does not depend on the order in which the redexes were reduced: the normal form is unique (see also section 1.3). However, there are restrictions that have to be put on the evaluation order. The specification of a pattern forces evaluation in order to decide whether or not a particular rule for a function can be applied.

Forced evaluation:

```
fac 0 = 1
fac n = n * fac (n - 1)
```

The expression `fac (1-1)` should of course not match the second rule, but the argument has to be evaluated first. After the evaluation of the argument the rules may be matched in order.

Furthermore, some reduction orders may not lead to the normal form at all (such a computation will not terminate).

Non-terminating reduction: define

```
inf = inf
```

then the following reduction order may be taken:

```
cond True 1 inf   →   cond True 1 inf   →   cond True 1 inf   →   ...
```

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

```
cond True 1 inf   →   1
```

The reduction strategy followed depends on the functional programming languages. In some languages, e.g. Lisp and Hope, the arguments of a function are always reduced before the function itself is considered as a redex. This is called *eager* evaluation and languages which use such an evaluation strategy, are sometimes called *eager* languages. Infinite data structures cannot be handled in these languages, because they are evaluated as soon as they are passed as arguments, which leads to infinite computations.

Nowadays, in many functional programming languages the rewriting is done *lazy*. That is, the value of a subexpression (redexes) is calculated if and only if this value must be known to do the rewriting. Lazy evaluation makes it possible to handle infinite data structures.

Lazy evaluation and infinite lists:

```
hd (tl ones)  →  hd (tl (1:ones))  →  hd ones  →  hd (1:ones)  → 1
```

In eager languages the evaluation of the parameter `ones` would not terminate:

```
hd (tl ones)  →  hd (tl (1:ones))  →  hd (tl (1:(1:ones)))  →  …
```

There is a trade-off in the choice between lazy and eager evaluation: on one hand lazy evaluation gives better expressiveness in the language and on the other hand eager evaluation simplifies the implementation of a language. We have chosen for lazy evaluation because we like to improve the expressiveness of programming languages.

Typing

There are untyped functional languages (Twentel, KRC and SASL) but most functional languages are typed. In untyped languages any kind of functions can be written which is at the same time their advantage and their disadvantage. After all, every function is written with intended argument and result domains. In practise, a programmer will want to restrict himself deliberately to using these functions with elements of the intended domains only. Essentially, the type of a function is precisely this information on the intended domains. The type information can not only be used as a filter to restrict the set of acceptable programs, but it can also be useful for a compiler to produce faster code for specific types.

Two ways of typing are distinguished: explicit typing and implicit typing.

Explicit typing (or type *checking*) means that the compiler or the interpreter checks the type which is explicitly given by the programmer. Type checking occurs e.g. in the languages FP, HOPE, PONDER, ML and Miranda. There are many different algorithms for type checking in functional languages, differing in the kind of types they allow.

Implicit typing (or type *inference* or type *deduction*) means that the compiler or interpreter of the language will try to infer the types of a program. So in principle, the programmer does not need to supply any type information.

From a software engineering point of view it can be argued that explicit typing is much better because it forces the user to think about what he really wants to do with the functions he defines. However, for simple functions a lot of (error prone) specifying is asked from the programmer and then it is of course very nice if the types are deduced by the system. Since Miranda (the language we use in our examples) has implicit as well as explicit typing we will discuss implicit typing some more and we will give some comments on this combination of both ways of typing.

The type inference algorithms are usually based on the Curry$^+$ type system of the λ-calculus. In the Curry$^+$ type system it is inherently undecidable to determine the type of a λ-term, because this type system contains a special type deduction rule (EQ: if two terms are β-equal they have the same type) which (in programming terms) means that in order to determine the type of an expression the expression first has to be fully evaluated. So any practical algorithm based on the Curry$^+$ type system must be an approximation.

Milner (Milner (1978)) has given such an approximating algorithm which is used in Miranda. This algorithm uses the Curry type system (i.e. Curry$^+$ without EQ) and with some small extensions. It can deal with polymorphism i.e. a function can be applied on objects of different types (e.g. the identity function delivers an expression of the same type as its argument). But it can not handle *internal* polymorphism i.e. polymorphism of locally defined functions or polymorphism of arguments in the definition itself.

Milner-untypable function definitions:

```
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"
```

In the right-hand-side of the definition of `missionimpossible` the type system does not allow the actual argument `length` to get two different types in the body of the function `fun`. `length` cannot get type `[num] -> num` on `[1,2,3]` and also type `[char] -> num` on `"abc"`. These types are consistent with the definition of `length` but they are not consistent with the definition of `fun`.

Although the deduction scheme can be improved for some cases, a fully satisfactory type inference algorithm is not yet found.

In Miranda type inference and type deduction are combined. A programmer may specify types but he does not need to do so. For this purpose Miranda uses the type checking algorithm of Mycroft (Mycroft (1984)) which only slightly differs from the type deduction algorithm of Milner. At first sight this is a very nice feature. However, in certain cases it may lead to not very admirable situations.

For instance, suppose a programmer gives the following definition:

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

The type deduction algorithm of Milner which is used in Miranda cannot deduce the type of `g` because of the internal polymorphism of `g`. `g` is used polymorphic in the right hand side of its own definition, namely with type instances: `char -> [num]` and `[num] -> [num]`. Hence, an error message will be given by the compiler.

However, when the user explicitly specifies the type of `g` as: `* -> [num]` the type checker has no problems with this case: the specified type is accepted and no error message is produced.

For a programmer it might be very confusing when on one hand a program has a type error if types are deduced while on the other hand if a type is explicitly given the program is typed correctly. This confusion might be prevented if the language has explicit typing only. The programming environment could then have a facility built in that on explicit request of the user does its best trying to infer the type. This would give the user a clearer view on type inference and type checking.

Examples

The power of functional programming languages is illustrated in two examples which use the general features of functional programming languages.

Sorting a list.

The function `sort` needs a list of any type as argument and delivers as result the sorted list which has the same type as the argument.

```
sort::              [*] → [*]

sort []             = []
sort (a:x)          = sort (smalleq a x) ++ [a] ++ sort (greater a x)
```

The functions `smalleq` and `greater` need two arguments: a list and an element. This element must have the same type as the elements of the list.

```
smalleq::           * → [*] → [*]

smalleq a []        = []
smalleq a (b:x)     = cond (b <= a) (b:(smaller a x)) (smaller a x)

greater::           * → [*] → [*]

greater a []        = []
greater a (b:x)     = cond (b > a)  (b:(greater a x)) (greater a x)
```

Obviously, the type of the elements of the list must be such, that the operations '<=' and '>' are well defined.

Roman numbers.

Roman numbers are built up from the characters M, D, C, L, X, V and I. Each of these characters has its own value. 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 which contains no abbreviations, 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 abbreviations make it less simple to calculate the value of a Roman number because now the value of the character depends on its relevant position in the string. Negative numbers, or the number zero cannot be expressed in Roman numbers. The values of the Roman characters and the values of the common abbreviations are:

M	=	1000	CM	=	DCCCC	=	M	-	C
D	=	500	CD	=	CCCC	=	D	-	C
C	=	100	XC	=	LXXXX	=	C	-	X
L	=	50	XL	=	XXXX	=	L	-	X
X	=	10	IX	=	VIIII	=	X	-	I
V	=	5	IV	=	IIII	=	V	-	I
I	=	1							

A Roman number is represented as a string. It is assumed that the specified functions are applied on correct arguments only.

```
|| value defines the value of a Roman digit.

value::       char → num

value 'M'     =   1000
value 'D'     =   500
value 'C'     =   100
value 'L'     =   50
value 'X'     =   10
value 'V'     =   5
value 'I'     =   1

|| rconvert converts a Roman number to a decimal number.

rconvert::              [char] → num

rconvert ('C':'M':x)   =    value 'M' - value 'C'   + rconvert x
rconvert ('C':'D':x)   =    value 'D' - value 'C'   + rconvert x
rconvert ('X':'C':x)   =    value 'C' - value 'X'   + rconvert x
rconvert ('X':'L':x)   =    value 'L' - value 'X'   + rconvert x
rconvert ('I':'X':x)   =    value 'X' - value 'I'   + rconvert x
rconvert ('I':'V':x)   =    value 'V' - value 'I'   + rconvert x
rconvert ( a : x)      =    value a                 + rconvert x
rconvert []            =    0
```

DISCUSSION

The following claims have been made in favour of functional programming.

- Functional languages have a sound mathematical basis, with function definition and application as the essential concepts.
- Because of the lack of side-effects, which leads to referential transparency, program correctness proofs are easier. Proofs can be constructed with classical mathematical techniques like induction. Proofs are compositional i.e. properties of a function which are proven, can be used directly in other functions. Also it makes program transformations easier: a programmer can start

with a straightforward solution of a problem and via transformations convert this solution to a more efficient one. Therefore it is less difficult to develop reliable software.

- Mainly because of the use of higher order functions the expressive power of functional programming languages is higher than of conventional languages. Programming is more like mathematically specifying the algorithm. Programs are therefore in generally shorter than their conventional counterparts and thus easier to enhance and maintain.

- A functional programming style is a mathematical style. Therefore a functional language is in particular suited as a specification language. The fact that it is also a programming language can be seen as a convenient additional feature. For instance, in Boute (1986) it is shown that the functional paradigm can be used for the description of digital and analog circuits.

- In the evaluation process of an expression it may occur that there is more then one redex in that expression. So, the evaluation process may continue in different ways. The Church-Rosser property states that this evaluation order is unimportant if the choice does not lead to an infinite evaluation process. The Church-Rosser property is a property of the set of definitions, which forms the program of a functional programming language. The Church-Rosser property is also valid in (subclasses of) the models that are used for functional languages (see also section 1.3 and chapter 2). Due to the Church-Rosser property, alternative evaluation orders, such as parallel evaluation, will never produce a wrong result, although special care has to be taken in order to avoid non-termination. So, functional languages seem to be very suited for parallel evaluation.

There also are the following serious disadvantages which however may be taken away by further research.

- Some algorithms seem to be difficult to express in a functional programming style. Although it is in principle not impossible to do, those programs which have a strongly imperative nature (process control, operating systems, concurrently accessed databases) look not elegant. There is some hope for improvement because functional languages are still under development and because one still is learning how to express algorithms better in these languages.
- Efficient implementations are now becoming available (Johnson (1984), Brus *et al.* (1987)) but they are still somewhat slower than implementations of imperative languages. However, space and time efficiency can still be improved by further applying and improving implementation techniques e.g. strictness analysis, avoidance of space leaks and introduction and implementation of arrays with fast creation, access and update. Furthermore, one has to keep in mind that a certain loss of efficiency was also accepted when high level imperative languages were started to be used.
- From the software engineering point of view functional languages still lack several essential aspects:

- Firstly, there is no well established programming style. Only the first attempts are made towards establishing such a programming style.
- Secondly, there is no well established proposal for modularity, although several attempts have been made.
- Thirdly, there are hardly any debugging facilities yet. Although programs tend to contain relatively few errors, debugging cannot totally be avoided. Especially in lazy languages it is difficult for the programmer to be able to get an idea of the exact execution order which influences the real-time behaviour of a program.
- Lastly, early programmers training and education is just getting started in an experimental way.

The balance between advantages and disadvantages is such that further research on functional programming languages is more than justified.

1.3 TRADITIONAL MODELS OF COMPUTATION FOR FUNCTIONAL LANGUAGES

In this section we will discuss some traditional models of computation for functional programming languages. For these models some properties are given in relation to other models. All these models are capable of representing every computable function. Each model can be used as computational model for functional languages. Our choice for a non-traditional model is motivated.

GENERAL TERMINOLOGY: ABSTRACT REDUCTION SYSTEMS

The general terminology which is used in functional languages and in all models for these languages is contained in Abstract Reduction Systems (ARS's). ARS's abstract from the precise structure of the objects and from the way reduction takes place.

An *Abstract Reduction System* (*ARS*) is a pair $(E,\rightarrow)$, where E is a set of *elements* and $\rightarrow$ is a binary relation on E. The transitive reflexive closure of $\rightarrow$ is denoted by $\rightarrow^*$ or $\twoheadrightarrow$. A similar definitions or ARS's can be found in Klop (1987).

The intuitive idea is that each element represents a program at some state of execution. If $x \rightarrow y$ then a next step in the execution could be transforming the element x to y.

We say that x can be *reduced* or *rewritten* to y in one *step*. We also say x *reduces* to y and y is called a *one-step reduct* of x. An element x of an ARS is a *redex* (*red*ucible *ex*pression) if there exists an y such that $x \rightarrow y$. A *reduction sequence* of an ARS is a sequence $x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$. The *length* of this sequence is n. A sequence of length 0 is *empty*.

An element x of an ARS is a *normal form* if for there is no y such that $x \rightarrow y$. An element x *has a normal form* if there is an element y such that $x \twoheadrightarrow y$ and y is a normal form.

Given an ARS (E,→), a *(reduction) strategy* for this system is a function S which takes each $x \in E$ to a set S(x) of nonempty finite reduction sequences, each beginning with x. Note that S(x) can be empty.

A strategy S is *deterministic* if, for all x, S(x) contains at most one element. A strategy S is a *one-step strategy* (or *1-strategy*) if for every x in E, every member of S(x) has length 1.

Write $x \rightarrow_S y$ if S(x) contains a reduction sequence ending with y. By abuse of notation, we may write $x \rightarrow_S y$ to denote some particular but unspecified member of S(x). An S-*sequence* is a reduction sequence of the form $x_0 \rightarrow_S x_1 \rightarrow_S x_2 \rightarrow_S$ .... A strategy S is *normalising* if for all x_0 having a normal form any S-sequence $x_0 \rightarrow_S x_1 \rightarrow_S x_2 \rightarrow_S$ .... must eventually terminate with a normal form.

A *reducer* with strategy S is a process which starts with an element x, chooses a reduction sequence of S(x) and repeats this with the end of the chosen reduction sequence as the new element x. A *result* x of a reducer is reached when S(x) is empty. Reducers are deterministic or non-deterministic. A reducer is *deterministic* if a reduction sequence in S(x) is chosen via a function.

An ARS is *confluent* or *has the Church-Rosser property* (*is Church-Rosser*) if for all elements x, y and z for which $x \twoheadrightarrow y$ and $x \twoheadrightarrow z$, there exists a c such that $y \twoheadrightarrow c$ and $z \twoheadrightarrow c$. It can be proven that confluent ARS's have the *unique normal form property* i.e. each element has atmost one normal form. The unique normal form property implies that different normalising reducers all will have the same result if there is a normal form. This is useful for implementations which can improve efficiency by using other (possibly parallel) reduction orders.

In Klop (1987) many other interesting properties of ARS's can be found.

LAMBDA-CALCULUS

The theory of the λ-calculus (an overview is given in Barendregt (1984)) as a model of computation has been introduced by Church in 1936 (the very same year in which Turing introduced the Turing machine model of computation) and is further investigated since. The basic concepts of λ-calculus are application and abstraction. The set of terms Λ is the following.

λ-calculus: terms

Every constant `c` of the set of constants `C` is a term. Every variable `v` of the set of variables `V` is a term.
If `M` and `N` are terms then `M N` is also a term. (application)
If `M` is a term and `x` is a variable then `λx.M` is also a term. (abstraction)

Reduction is done by substitution of the variables (like in e.g. `f(x)= x + x`; `f(2)= 2 + 2`). The reduction relation is defined as follows.

λ-calculus: reduction relation (β-reduction)

$$(\lambda x.M)\ N \quad \rightarrow_\beta \quad M\ [x:=N]$$

where `[x:=N]` denotes substitute `N` for `x`. This reduction relation is closed under contexts, e.g. if $M \rightarrow_\beta N$ then also $\lambda x.M\ Z \rightarrow_\beta \lambda x.N\ Z$.

The abstract reduction system corresponding to λ-calculus is of course $(\Lambda,\rightarrow_\beta)$. The substitution mechanism is not at all trivial so in implementations it has to be taken care of in a special way. Not all occurrences of the variable may be substituted (only so called *free* occurrences). Furthermore it might be necessary to change the names of some variables during substitution. Changing the name of a variable is also called α-conversion.

λ-calculus: substitution

$$(\lambda x.x\ (\lambda x.x)\ x)\ c \rightarrow_\beta x\ (\lambda x.x)\ x\ [x:=c] = c\ (\lambda x.x)\ c$$
$$(\lambda x.(\lambda y.y\ x))\ y \rightarrow_\beta (\lambda y.y\ x)\ [x:=y] =_\alpha (\lambda z.z\ x)\ [x:=y] = \lambda z.z\ y$$

Lazy evaluation (leftmost) is normalising for the λK-calculus. Eager evaluation is not normalising for the λK-calculus. In λI-calculus terms which have a normal form, have no non-terminating reduction order. So, eager evaluation is normalising for λI-calculus. Definitions of λK- and λI-calculus can be found in Barendregt (1984).

λ-calculus is confluent. So alternative (possibly parallel) orders of evaluation can never yield a wrong result. Extending λ-calculus with so called delta rules which act on normal forms and use some internal representation to produce a result (e.g. arithmetical functions), is still confluent (Barendregt (1988)).

λ-calculus can be implemented using sharing (Wadsworth (1971)). It is very difficult to do this optimally avoiding copying of redexes always. The following term communicated to us by J.J. Lévy, illustrates the essential issues for this optimality problem:

$$(\lambda z.z\ (\lambda x.x)\ z)\ (\lambda f.(\lambda z.z\ ((\lambda x.x)\ z))\ (f\ c))$$

Traditionally, the λ-calculus is considered to be a suitable model for functional languages (Peyton Jones (1988b)). However, certain aspects of functional languages and the way they are usually implemented, cannot be modelled within this calculus. In particular, the λ-calculus lacks explicit recursion, pattern matching and the explicit notion of sharing of computations. With λ-calculus it is impossible to reason about these aspects which, in our opinion, are essential for the languages and their implementation.

COMBINATORY LOGIC

Combinatory logic (Schönfinkel (1924)) is closely related to λ-calculus. In λ-calculus terms a *combinator* is a λ-term without free variables. All recursive functions can be defined with applications of only two basic combinators `S` and `K` which are defined below.

SK-combinatory logic: terms

S is a term and K is a term; if M and N are terms then M N is also a term (application).

SK-combinatory logic: the reduction relation follows from the definitions of S and K in λ-calculus: S ≡ λx.λy.λz.x z (y z) and K ≡ λx.λy.x.

```
S x y z        → x z (y z)
K x y          → x
```

Of course, it is impossible to make an efficient implementation based on the combinators S and K only. A simple function needs already lots of S's and K's.

A simple algorithm for translating λ-calculus to S- and K-combinators is the following. This is one of the several algorithms which are called *bracket-abstraction* because the translation of a λ-term x to a combinator term is indicated by [x].

A bracket-abstraction algorithm:

```
[λx.x]        ≡    S K K
[λx.y]        ≡    K y
[λx.P Q]      ≡    S [λx.P] [λx.Q]
[P Q]         ≡    [P] [Q]
```

The abstraction algorithm reveals an important intuitive aspect of the combinators. They distribute an argument over the function body; K means the argument is not needed here and S means distribute the argument over both parts of the application. This intuitive idea is applied more clearly in the concept of Director Strings (Kennaway & Sleep (1988)).

David Turner was the first to introduce an implementation of a functional language using combinators as an intermediate language (Turner (1979a)). He did not use the set of two combinators mentioned above but he used an extended set of about 15 combinators and of course some delta-rules. His paper induced much research on which set of combinators was best for implementation purposes and it also induced research on machine architectures based on such sets of combinators. This research was very colourful with concepts as supercombinators and superdoopercombinators and machines with names such as NORMA and SKIM. However it turned out that using an intermediate language with graph primitives and special compiler techniques it was possible to get at least an order of magnitude increase in efficiency beating on an ordinary VAX the expected performance of special-purpose hardware based on these combinators.

An interesting more recently proposed combinator model is built with the so called *categorical combinators* (Curien (1986)) based on a category theoretical model of the λ-calculus. These combinators encode an environmental implementation of the λ-calculus using eager evaluation.

Leftmost-outermost reduction implemented with sharing is normalising and optimal for combinator systems.

Just as λ-calculus, combinatory logic lacks explicit recursion, pattern matching and the explicit notion of sharing of computations. With combinatory logic it is impossible to reason about these aspects which, in our opinion, are essential for the languages and their implementation.

1.4 CONCLUSION

We believe that the use of the declarative paradigm (and functional languages in particular) might contribute to solving the software crisis.

In our opinion both λ-calculus and combinatory logic are not the best suited models of computation for functional languages and their implementation because they lack recursion, pattern matching and sharing. Therefore, we will consider term and graph rewriting systems in the next chapter.