1
Standard ML

The first ML compiler was built in 1974. As the user community grew, various dialects began to appear. The ML community then got together to develop and promote a common language, Standard ML — sometimes called SML, or just ML. Good Standard ML compilers are available.

Standard ML has become remarkably popular in a short time. Universities around the world have adopted it as the first programming language to teach to students. Developers of substantial applications have chosen it as their implementation language. One could explain this popularity by saying that ML makes it easy to write clear, reliable programs. For a more satisfying explanation, let us examine how we look at computer systems.

Computers are enormously complex. The hardware and software found in a typical workstation are more than one mind can fully comprehend. Different people understand the workstation on different levels. To the user, the workstation is a word processor or spreadsheet. To the repair crew, it is a box containing a power supply, circuit boards, etc. To the machine language programmer, the workstation provides a large store of bytes, connected to a processor that can perform arithmetic and logical operations. The applications programmer understands the workstation through the medium of the chosen programming language.

Here we take 'spreadsheet', 'power supply' and 'processor' as ideal, abstract concepts. We think of them in terms of their functions and limitations, but not in terms of how they are built. Good abstractions let us use computers effectively, without being overwhelmed by their complexity.

Conventional 'high level' programming languages do not provide a level of abstraction significantly above machine language. They provide convenient notations, but only those that map straightforwardly to machine code. A minor error in the program can make it destroy other data or even itself. The resulting behaviour can be explained only at the level of machine language — if at all!

ML is well above the machine language level. It supports ***functional programming***, where programs consist of functions operating on simple data structures. Functional programming is ideal for many aspects of problem solving, as argued briefly below and demonstrated throughout the book. Programming tasks can be approached mathematically, without preoccupation with the computer's internal workings. ML also provides ***mutable*** variables and arrays. Mutable objects can be updated using an assignment command; using them, any piece of conventional code can be expressed easily. For structuring large systems, ML provides ***modules***: parts of the program can be specified and coded separately.

Most importantly of all, ML protects programmers from their own errors. Before a program may run, the compiler checks that all module interfaces agree and that data are used consistently. For example, an integer may not be used as a store address. (It is a myth that real programs must rely on such tricks.) As the program executes, further checking ensures safety: even a faulty ML program continues to behave as an ML program. It might run forever and it might return to the user with an error message. But it cannot crash.

ML supports a level of abstraction that is oriented to the requirements of the programmer, not those of the hardware. The ML system can preserve this abstraction, even if the program is faulty. Few other languages offer such assurances.

Functional Programming

Programming languages come in several varieties. Languages like Fortran, Pascal and C are called ***procedural***: their main programming unit is the procedure. A popular refinement of this approach centres on objects that carry their own operations about with them. Such ***object-oriented*** languages include C++ and Modula-3. Both approaches rely on commands that act upon the machine state; they are both ***imperative*** approaches.

Just as procedural languages are oriented around commands, functional languages are oriented around expressions. Programming without commands may seem alien to some readers, so let us see what lies behind this idea. We begin with a critique of imperative programming.

1.1 *Expressions versus commands*

Fortran, the first high-level programming language, gave programmers the arithmetic expression. No longer did they have to code sequences of additions, loads and stores on registers: the FORmula TRANslator did this for them. Why are expressions so important? Not because they are familiar: the Fortran syntax for

$$\sqrt{\frac{\sin^2\theta}{1+|\cos\phi|}}$$

has but a passing resemblance to that formula. Let us consider the advantages of expressions in detail. Expressions in Fortran can have ***side effects***: they can change the state. We shall focus on pure expressions, which merely compute a value.

Expressions have a recursive structure. A typical expression like

$$f(E_1 + E_2) - g(E_3)$$

is built out of other expressions E_1, E_2 and E_3, and may itself form part of a larger expression.

The value of an expression is given recursively in terms of the values of its subexpressions. The subexpressions can be evaluated in any order, or even in parallel.

Expressions can be transformed using mathematical laws. For instance, replacing $E_1 + E_2$ by $E_2 + E_1$ does not affect the value of the expression above, thanks to the commutative law of addition. This ability to substitute equals for equals is called ***referential transparency***. In particular, an expression may safely be replaced by its value.

Commands share most of these advantages. In modern languages, commands are built out of other commands. The meaning of a command like

$$\texttt{while}\ B_1\ \texttt{do}\ (\texttt{if}\ B_2\ \texttt{then}\ C_1\ \texttt{else}\ C_2)$$

can be given in terms of the meanings of its parts. Commands even enjoy referential transparency: laws like

$$(\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2); C \equiv \texttt{if}\ B\ \texttt{then}\ (C_1; C)\ \texttt{else}\ (C_2; C)$$

can be proved and applied as substitutions.

However, the meaning of an expression is simply the result of evaluating it, which is why subexpressions can be evaluated independently of each other. The meaning of an expression can be extremely simple, like the number 3. The meaning of a command is a state transformation or something equally complicated. To understand a command, you have to understand its full effect on the machine's state.

1.2 *Expressions in procedural programming languages*

How far have programming languages advanced since Fortran? Consider Euclid's Algorithm, which is defined by recursion, for computing the

Greatest Common Divisor (GCD) of two natural numbers:

$$\begin{aligned} gcd(0, n) &= n \\ gcd(m, n) &= gcd(n \text{ mod } m, m) \qquad \text{for } m > 0 \end{aligned}$$

In Pascal, a procedural language, most people would code the GCD as an imperative program:

```
function gcd(m,n: integer): integer;
  var prevm: integer;
begin
  while m<>0 do
    begin prevm := m;  m := n mod m;  n := prevm end;
  gcd := n
end;
```

Here it is in Standard ML as a functional program:

```
fun gcd(m,n) =
      if m=0 then n
             else gcd(n mod m, m);
```

The imperative program, though coded in a 'high-level' language, is hardly clearer or shorter than a machine language program. It repeatedly updates three quantities, one of which is just a temporary storage cell. Proving that it correctly implements Euclid's algorithm requires Floyd-Hoare proof rules — a tedious enterprise. In contrast, the functional version obviously implements Euclid's Algorithm.

A recursive program in Pascal would be only a slight improvement. Recursive procedure calls are seldom implemented efficiently. Thirty years after its introduction to programming languages, recursion is still regarded as something to eliminate from programs. Correctness proofs for recursive procedures have a sad history of complexity and errors.

Pascal expressions do not satisfy the usual mathematical laws. An optimizing compiler might transform $f(z) + u/2$ into $u/2 + f(z)$. However, these expressions may not compute the same value if the 'function' f changes the value of u. The meaning of an expression in Pascal involves states as well as values. For all practical purposes, referential transparency has been lost.

In a purely functional language there is no state. Expressions satisfy the usual mathematical laws, up to the limitations of the machine (for example, real arithmetic is approximate). Purely functional programs can also be written in Standard ML. However, ML is not pure because of its assignments and input/output commands. The ML programmer whose style is 'almost' functional had better not be lulled into a false sense of referential transparency.

1.3 *Storage management*

Expressions in procedural languages have progressed little beyond Fortran; they have not kept up with developments in data structures. Suppose we have employee records consisting of name, address, and other details. We cannot write record-valued expressions, or return an employee record from a function; even if the language permits this, copying such large records is prohibitively slow.

To avoid copying large objects, we can refer to them indirectly. Our record-valued function could allocate storage space for the employee record, and return its address. Instead of copying the record from one place to another, we copy its address instead. When we are finished with the record, we deallocate (release) its storage. (Presumably the employee got sacked.) Addresses used in this way are called ***references*** or ***pointers***.

Deallocation is the bugbear of this approach. The program might release the storage prematurely, when the record is still in use. Once that storage is reallocated, it will be used for different purposes at the same time. Anything could happen, leading (perhaps much later) to a mysterious crash. This is one of the most treacherous programming errors.

If we never deallocate storage, we might run out of it. Should we then avoid using references? But many basic data structures, such as the linked list, require references.

Functional languages, and some others, manage storage automatically. The programmer does not decide when to deallocate a record's storage. At intervals, the run-time system scans the store systematically, marking everything that is accessible and reclaiming everything that is not. This operation is called ***garbage collection***, although it is more like recycling. Garbage collection can be slow and may require additional space, but it pays dividends.

Languages with garbage collection typically use references heavily in their internal representation of data. A function that 'returns' an employee record actually returns only a reference to it, but the programmer does not know or care. The language gains in expressive power. The programmer, freed from the chore of storage management, can work more productively.

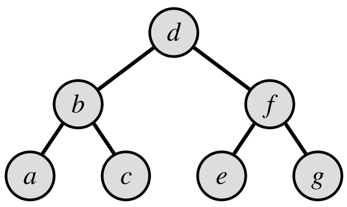
1.4 *Elements of a functional language*

Functional programs work with values, not states. Their tools are expressions, not commands. How can assignments, arrays and loops be dispensed with? Does not the outside world have state? These questions pose real challenges. The functional programmer can exploit a wide range of techniques to solve problems.

Lists and trees. Collections of data can processed as lists of the form

$$[a, b, c, d, e, \ldots].$$

Lists support sequential access: scanning from left to right. This suffices for most purposes, even sorting and matrix operations. A more flexible way of organizing data is as a tree:



Balanced trees permit random access: any part can be reached quickly. In theory, trees offer the same efficiency as arrays; in practice, arrays are often faster. Trees play key rôles in symbolic computation, representing logical terms and formulæ in theorem provers. Lists and trees are represented using references, so the runtime system must include a garbage collector.

Functions. Expressions consist mainly of function applications. To increase the power of expressions, functions must be freed from arbitrary restrictions. Functions may take any type of arguments and return any type of result. As we shall see, 'any type' includes functions themselves, which can be treated like other data; making this work also requires a garbage collector.

Recursion. Variables in a functional program obtain their values from outside (when a function is called) or by declaration. They cannot be updated, but recursive calls can produce a changing series of argument values. Recursion is easier to understand than iteration — if you do not believe this, recall our two GCD programs. Recursion eliminates the baroque looping constructs of procedural languages.[1]

Pattern-matching. Most functional languages allow a function to analyse its argument using pattern-matching. A function to count the elements of a list looks like this in ML:

[1] Recursion does have its critics. Backus (1978) recommends providing iteration primitives to replace most uses of recursion in function definitions. However, his style of functional programming has not caught on.

```
fun length []      = 0
  | length (x::xs) = 1 + length xs;
```

We instantly see that the length of the empty list ([]) is zero, and that the length of a list consisting of the element x prefixed to the list xs is the length of xs plus one. Here is the equivalent definition in Lisp, which lacks pattern-matching:

```
(define (length x)
  (if (null? x)
      0
      (+ 1 (length (cdr x)))))
```

ML function declarations often consider half a dozen cases, with patterns much more complicated than $x::xs$. Expressing such functions without using patterns is terribly cumbersome. The ML compiler does this internally, and can do a better job than the programmer could.

Polymorphic type checking. Programmers, being human, often err. Using a nonexistent part of a data structure, supplying a function with too few arguments, or confusing a reference to an object with the object itself are serious errors: they could make the program crash. Fortunately, the compiler can detect them before the program runs, provided the language enforces a type discipline. ***Types*** classify data as integers, reals, lists, etc., and let us ensure that they are used sensibly.

Some programmers resist type checking because it can be too restrictive. In Pascal, a function to compute the length of a list must specify the — completely irrelevant! — type of the list's elements. Our ML length function works for all lists because ML's type system is ***polymorphic***: it ignores the types of irrelevant components. Our Lisp version also works for all lists, because Lisp has no compile-time type checking. Lisp is more flexible than ML; a single list can mix elements of different types. The price of this freedom is hours spent hunting errors that might have been caught automatically.

Higher-order functions. Functions themselves are computational values. Even Fortran lets a function be passed as an argument to another function, but few procedural languages let function values play a full rôle as data structures.

A ***higher-order*** function (or ***functional***) is a function that operates on other functions. The functional map, when applied to a function f, returns another function; that function takes

$$[x_1, \ldots, x_n] \quad \text{to} \quad [f(x_1), \ldots, f(x_n)].$$

Another higher-order function, when applied to a function f and value e, returns

$$f(x_1, f(x_2, \ldots, f(x_n, e) \ldots)).$$

If $e = 0$ and $f = +$ (yes, the addition operator is a function) then we get the sum of $x_1, \ldots, x_n$, computed by

$$x_1 + (x_2 + \cdots + (x_n + 0) \cdots).$$

If $e = 1$ and $f = \times$ then we get their product, computed by

$$x_1 \times (x_2 \times \cdots \times (x_n \times 1) \cdots).$$

Other computations are expressed by suitable choices for f and e.

Infinite data structures. Infinite lists like $[1, 2, 3, \ldots]$ can be given a computational meaning. They can be of great help when tackling sophisticated problems. Infinite lists are processed using ***lazy evaluation***, which ensures that no value — or part of a value — is computed until it is actually needed to obtain the final result. An infinite list never exists in full; it is rather a process for computing successive elements upon demand.

The search space in a theorem prover may form an infinite tree, whose success nodes form an infinite list. Different search strategies produce different lists of success nodes. The list can be given to another part of the program, which need not know how it was produced.

Infinite lists can also represent sequences of inputs and outputs. Many of us have encountered this concept in the ***pipes*** of the Unix operating system. A chain of processes linked by pipes forms a single process. Each process consumes its input when available and passes its output along a pipe to the next process. The outputs of intermediate processes are never stored in full. This saves storage, but more importantly it gives us a clear notation for combining processes. Mathematically, every process is a function from inputs to outputs, and the chain of processes is their composition.

Input and output. Communication with the outside world, which has state, is hard to reconcile with functional programming. Infinite lists can handle sequential input and output (as mentioned above), but interactive programming and process communication are thorny issues. Many functional approaches have been investigated; ***monads*** are one of the most promising (Peyton Jones and Wadler, 1993). ML simply provides commands to perform input and output; thus, ML abandons functional programming here.

Functional languages: a survey. The mainstream functional languages adopt lazy evaluation, pattern-matching and ML-style polymorphic types. Miranda is an elegant language by David A. Turner (1990a). Lazy ML is a dialect of ML with lazy evaluation; its compiler generates efficient code (Augustsson and Johnsson, 1989). Haskell was designed by a committee of researchers as a common language (Hudak *et al.*, 1992); it has been widely adopted.

John Backus (1978) introduced the language FP in a widely publicized lecture. FP provides many higher-order functions (called 'combining forms'), but the programmer may not define new ones. Backus criticized the close coupling between programming languages and the underlying hardware, coining the phrase ***von Neumann bottleneck*** for the connection between the processor and the store. Many have argued that functional languages are ideal for parallel hardware. Sisal has been designed for parallel numerical computations; Cann (1992) claims that Sisal sometimes outperforms Fortran.

Many implementation techniques for functional programming, such as garbage collection, originated with Lisp (McCarthy *et al.*, 1962). The language includes low-level features that can be misused to disastrous effect. Later dialects, including Scheme (Abelson and Sussman, 1985) and Common Lisp, provide higher-order functions. Although much Lisp code is imperative, the first functional programs were written in Lisp. Most ML dialects include imperative features, but ML is more disciplined than Lisp. It has compile-time type checking, and allows updates only to mutable objects.

1.5 *The efficiency of functional programming*

A functional program typically carries a large run-time system with a resident compiler. The garbage collector may require a uniform representation of data, making it occupy additional storage. The functional programmer is sometimes deprived of the most efficient data structures, such as arrays, strings and bit vectors. A functional program may therefore be less efficient than the corresponding C program, particularly in its storage demands.

ML is best suited for large, complex applications. Type checking, automatic storage allocation and other advantages of functional programming can make the difference between a program that works and one that doesn't. Efficiency becomes a secondary issue; besides, with a demanding application, the difference will be less pronounced. Most functional programs ought to run nearly as fast as their procedural counterparts — perhaps five times slower in the worst case.

Efficiency is regarded with suspicion by many researchers, doubtless because many programs have been ruined in its pursuit. Functional programmers have sometimes chosen inefficient algorithms for the sake of clarity, or have sought to enrich their languages rather than implement them better. This attitude, more than technical reasons, has given functional programming a reputation for inefficiency.

We must now redress the balance. Functional programs must be efficient, or nobody will use them. Algorithms, after all, are designed to be efficient. The Greatest Common Divisor of two numbers can be found by searching through all possible candidates. This exhaustive search algorithm is clear, but useless. Euclid's Algorithm is fast and simple, having sacrificed clarity.

The exhaustive search algorithm for the GCD is an example of an ***executable specification***. One approach to program design might start with this and apply ***transformations*** to make it more efficient, while preserving its correctness. Eventually it might arrive at Euclid's Algorithm. Program transformations can indeed improve efficiency, but we should regard executable specifications with caution. The Greatest Common Divisor of two integers is, by definition, the largest integer that exactly divides both; the specification does not mention search at all. The exhaustive search algorithm is too complicated to be a good specification.

Functional programming and logic programming are instances of ***declarative programming***. The ideal of declarative programming is to free us from writing programs — just state the requirements and the computer will do the rest. Hoare (1989c) has explored this ideal in the case of the Greatest Common Divisor, demonstrating that it is still a dream. A more realistic aim for declarative programming is to make programs easier to understand. Their correctness can be justified by simple mathematical reasoning, without thinking about bytes. Declarative programming is still programming; we still have to code efficiently.

This book gives concrete advice about performance and tries to help you decide where efficiency matters. Most natural functional definitions are also reasonably efficient. Some ML compilers offer ***execution profiling***, which measures the time spent by each function. The function that spends the most time (never the one you would expect) becomes a prime candidate for improvement. Such bottom-up optimization can produce dramatic results, although it may not reveal global causes of waste. These considerations hold for programming generally — be it functional, procedural, object-oriented or whatever.

Correctness must come first. Clarity must usually come second, and efficiency third. Any sacrifice of clarity makes the program harder to maintain, and must be justified by a significant efficiency gain. A judicious mixture of realism and principle, with plenty of patience, makes for efficient programs.

Applications of functional programming. Functional programming techniques are used in artificial intelligence, formal methods, computer aided design, and other tasks involving symbolic computation. Substantial compilers have been written in (and for) Standard ML (Appel, 1992) and Haskell (Peyton Jones, 1992). Networking software has been written in ML (Biagioni *et al.*, 1994), in a project to demonstrate ML's utility for systems programming. A major natural language processing system, called LOLITA, has been written in Haskell (Smith *et al.*, 1994); the authors adopted functional programming in order to manage the complexity of their system. Hartel and Plasmeijer (1996) describe six major projects, involving diverse applications. Wadler and Gill (1995) have compiled a list of real world applications; these cover many domains and involve all the main functional languages.

Standard ML

Every successful language was designed for some specific purpose: Lisp for artificial intelligence, Fortran for numerical computation, Prolog for natural language processing. Conversely, languages designed to be general purpose — such as the 'algorithmic languages' Algol 60 and Algol 68 — have succeeded more as sources of ideas than as practical tools.

ML was designed for theorem proving. This is not a broad field, and ML was intended for the programming of one particular theorem prover — a specific purpose indeed! This theorem prover, called Edinburgh LCF (Logic for Computable Functions) spawned a host of successors, all of which were coded in ML. And just as Lisp, Fortran and Prolog have applications far removed from their origins, ML is being used in diverse problem areas.

1.6 *The evolution of Standard ML*

As ML was the Meta Language for the programming of proof strategies, its designers incorporated the necessary features for this application:

- The inference rules and proof methods were to be represented as functions, so ML was given the full power of higher-order functional programming.
- The inference rules were to define an abstract type: the type of theorems. Strong type checking (as in Pascal) would have been too restrictive, so ML was given polymorphic type checking.
- Proof methods could be combined in complex ways. Failure at any point had to be detected so that another method could be tried. So ML was allowed to raise and trap exceptions.
- Since a theorem prover would be useless if there were loopholes, ML was designed to be secure, with no way of corrupting the environment.

The ML system of Edinburgh LCF was slow: programs were translated into Lisp and then interpreted. Luca Cardelli wrote an efficient compiler for his version of ML, which included a rich set of declaration and type structures. At Cambridge

University and INRIA, the ML system of LCF was extended and its performance improved. ML also influenced HOPE; this purely functional language adopted polymorphism and added recursive type definitions and pattern-matching.

Robin Milner led a standardization effort to consolidate the dialects into Standard ML. Many people contributed. The module language — the language's most complex and innovative feature — was designed by David MacQueen and refined by Milner and Mads Tofte. In 1987, Milner won the British Computer Society Award for Technical Excellence for his work on Standard ML. The first compilers were developed at the Universities of Cambridge and Edinburgh; the excellent Standard ML of New Jersey appeared shortly thereafter.

Several universities teach Standard ML as the students' first programming language. ML provides a level base for all students, whether they arrive knowing C, Basic, machine language or no language at all. Using ML, students can learn how to analyse problems mathematically, breaking the bad habits learned from low-level languages. Significant computations can be expressed in a few lines. Beginners especially appreciate that the type checker detects common errors, and that nothing can crash the system!

Section 1.5 has mentioned applications of Standard ML to networking, compiler construction, etc. Theorem proving remains ML's most important application area, as we shall see below.

Further reading. Gordon *et al.* (1979) describe LCF. Landin (1966) discusses the language ISWIM, upon which ML was originally based. The formal definition of Standard ML has been published as a book (Milner *et al.*, 1990), with a separate volume of commentary (Milner and Tofte, 1990).

Standard ML has not displaced all other dialects. The French, typically, have gone their own way. Their language CAML provides broadly similar features with the traditional ISWIM syntax (Cousineau and Huet, 1990). It has proved useful for experiments in language design; extensions over Standard ML include lazy data structures and dynamic types. CAML Light is a simple byte-code interpreter that is ideal for small computers. Lazy dialects of ML also exist, as mentioned previously. HOPE continues to be used and taught (Bailey, 1990).

1.7 *The ML tradition of theorem proving*

Theorem proving and functional programming go hand in hand. One of the first functional programs ever written is a simple theorem prover (McCarthy *et al.*, 1962). Back in the 1970s, when some researchers were wondering what functional programming was good for, Edinburgh LCF was putting it to work.

Fully automatic theorem proving is usually impossible: for most logics, no automatic methods are known. The obvious alternative to automatic theorem proving, proof checking, soon becomes intolerable. Most proofs involve long, repetitive combinations of rules.

Edinburgh LCF represented a new kind of theorem prover, where the level of automation was entirely up to the user. It was basically a programmable proof checker. Users could write proof procedures in ML — the Meta Language — rather than typing repetitive commands. ML programs could operate on expressions of the Object Language, namely Scott's Logic of Computable Functions.

Edinburgh LCF introduced the idea of representing a logic as an abstract type of theorems. Each axiom was a primitive theorem while each inference rule was a function from theorems to theorems. Type checking ensured that theorems could be made only by axioms and rules. Applying inference rules to already known theorems constructed proofs, rule by rule, in the forward direction.

Tactics permitted a more natural style, backward proof. A tactic was a function from goals to subgoals, justified by the existence of an inference rule going the other way. The tactic actually returned this inference rule (as a function) in its result: tactics were higher-order functions.

Tacticals provided control structures for combining simple tactics into complex ones. The resulting tactics could be combined to form still more complex tactics, which in a single step could perform hundreds of primitive inferences. Tacticals were even more 'higher-order' than tactics. New uses for higher-order functions turned up in rewriting and elsewhere.

Further reading. Automated theorem proving originated as a task for artificial intelligence. Later research applied it to reasoning tasks such as planning (Rich and Knight, 1991). Program verification aims to prove software correct. Hardware verification, although a newer field, has been more successful; Graham (1992) describes the verification of a substantial VLSI chip and surveys other work.

Offshoots of Edinburgh LCF include HOL88, which uses higher-order logic (Gordon and Melham, 1993) and Nuprl, which supports constructive reasoning (Constable *et al.*, 1986).

Other recent systems adopt Standard ML. LAMBDA is a hardware synthesis tool, for designing circuits and simultaneously proving their correctness using higher-order logic. ALF is a proof editor for constructive type theory (Magnusson and Nordström, 1994).

1.8 *The new standard library*

The ML definition specifies a small library of standard declarations, including operations on numbers, strings and lists. Many people have found this library inadequate. For example, it has nothing to convert a character string such as `"3.14"` into a real number. The library's shortcomings have become more apparent as people have used ML for systems programming and other unforeseen areas. A committee, comprising several compiler writing teams, has drafted a new ML standard library (Gansner and Reppy, 1996). As of this writing it is still under development, but its basic outlines are known.

The library requires some minor changes to ML itself. It introduces a type of characters, distinct from character strings of length one. It allows the coexistence of numeric types that differ in their internal representations, and therefore in their precisions; this changes the treatment of some numerical functions.

The library is organized using ML modules. The numerous functions are components of ML ***structures***, whose contents is specified using ML ***signatures***. The functions are invoked not by their name alone, but via the name of their structure; for example, the sign function for real numbers is *Real*.*sign*, not just *sign*. Many function names occur in more than one structure; the library also provides *Int*.*sign*. When we later discuss modules, the library will help motivate the key concepts. Here is a summary of the library's main components, with the relevant structures:

- Operations on lists and lists of pairs belong to the structures *List* and *ListPair*. Some of these will be described in later chapters.
- Integer operations belong to the structure *Int*. Integers may be available in various precisions. These may include the usual hardware integers (structure *FixedInt*), which are efficient but have limited size. They could include unlimited precision integers (structure *IntInf*), which are essential for some tasks.
- Real number operations belong to the structure *Real*, while functions such as *sqrt*, *sin* and *cos* belong to *Math*. The reals may also be available in various precisions. Structures have names such as *Real*32 or *Real*64, which specify the number of bits used.
- Unsigned integer arithmetic is available. This includes bit-level operations such as logical 'and', which are normally found only in low-level languages. The ML version is safe, as it does not allow the bits to be converted to arbitrary types. Structures have names such as *Word*8.
- Arrays of many forms are provided. They include mutable arrays like those of imperative languages (structure *Array*), and immutable arrays (structure *Vector*). The latter are suitable for functional programming, since they cannot be updated. Their initial value is given by some calculation — one presumably too expensive to perform repeatedly.
- Operations on characters and character strings belong to structures *Char* and *String* among others. The conversion between a type and its textual representation is defined in the type's structure, such as *Int*.

- Input/output is available in several forms. The main ones are text I/O, which transfers lines of text, and binary I/O, which transfers arbitrary streams of bytes. The structures are *TextIO* and *BinIO*.
- Operating system primitives reside in structure *OS*. They are concerned with files, directories and processes. Numerous other operating system and input/output services may be provided.
- Calendar and time operations, including processor time measurements, are provided in structures *Date*, *Time* and *Timer*.
- Declarations needed by disparate parts of the library are collected into structure *General*.

Many other packages and tools, though not part of the library, are widely available. The resulting environment provides ample support for the most demanding projects.

1.9 *ML and the working programmer*

Software is notoriously unreliable. Wiener (1993) describes countless cases where software failures have resulted in loss of life, business crises and other calamities. Software products come not with a warranty, but with a warranty disclaimer. Could we prevent these failures by coding in ML instead of C? Of course not — but it would be a step in the right direction.

Part of the problem is the prevailing disdain for safety. Checks on the correct use of arrays and references are costly, but they can detect errors before they do serious harm. C. A. R. Hoare has said,

> . . . it is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea? (Hoare, 1989b, page 198)

This quote, from a lecture first given in 1973, has seldom been heeded. Typical compilers omit checks unless specifically commanded to include them. The C language is particularly unsafe: as its arrays are mere storage addresses, checking their correct usage is impractical. The standard C library includes many procedures that risk corrupting the store; they are given a storage area but not told its size! In consequence, the Unix operating system has many security loopholes. The Internet Worm exploited these, causing widespread network disruption (Spafford, 1989).

ML supports the development of reliable software in many ways. Compilers do not allow checks to be omitted. Appel (1993) cites its safety, automatic storage allocation, and compile-time type checking; these eliminate some major errors altogether, and ensure the early detection of others. Appel shares the view that functional programming is valuable, even in major projects.

Moreover, ML is defined formally. Milner *et al.* (1990) is not the first formal definition of a programming language, but it is the first one that compiler writers can understand.[2] Because the usual ambiguities are absent, compilers agree to a remarkable extent. The new standard library will strengthen this agreement. A program ought to behave identically regardless of which compiler runs it; ML is close to this ideal.

A key advantage of ML is its module system. System components, however large, can be specified and coded independently. Each component can supply its specified services, protected from external tampering. One component can take other components as parameters, and be compiled separately from them. Such components can be combined in many ways, configuring different systems.

Viewed from a software engineering perspective, ML is an excellent language for large systems. Its modules allow programmers to work in teams, and to reuse components. Its types and overall safety contribute to reliability. Its exceptions allow programs to respond to failures. Comparing ML with C, Appel admits that ML programs need a great deal of space, but run acceptably fast. Software developers have a choice of commercially supported compilers.

We cannot soon expect to have ML programs running in our digital watches. With major applications, however, reliability and programmer productivity are basic requirements. Is the age of C drawing to a close?

[2] This is possible thanks to recent progress in the theory of programming languages. The ML definition is an example of a structural operational semantics (Hennessy, 1990).