Chapter 1

Introduction

Programs in traditional languages, such as FORTRAN, Algol, C and Modula-3, rely heavily on modifying the values of a collection of variables, called the *state*. If we neglect the input-output operations and the possibility that a program might run continuously (e.g. the controller for a manufacturing process), we can arrive at the following abstraction. Before execution, the state has some initial value σ, representing the inputs to the program, and when the program has finished, the state has a new value σ' including the result(s). Moreover, during execution, each command changes the state, which has therefore proceeded through some finite sequence of values:

$$\sigma = \sigma_0 \to \sigma_1 \to \sigma_2 \to \cdots \to \sigma_n = \sigma'$$

For example in a sorting program, the state initially includes an array of values, and when the program has finished, the state has been modified in such a way that these values are sorted, while the intermediate states represent progress towards this goal.

The state is typically modified by *assignment* commands, often written in the form `v = E` or `v := E` where `v` is a variable and `E` some expression. These commands can be executed in a sequential manner by writing them one after the other in the program, often separated by a semicolon. By using statements like `if` and `while`, one can execute these commands conditionally, and repeatedly, depending on other properties of the current state. The program amounts to a set of instructions on how to perform these state changes, and therefore this style of programming is often called *imperative* or *procedural*. Correspondingly, the traditional languages intended to support it are known as imperative or procedural languages.

Functional programming represents a radical departure from this model. Essentially, a functional program is simply an expression, and execution means evaluation of the expression.[1] We can see how this might be possible, in gen-

[1]Functional programming is often called 'applicative programming' since the basic mecha-

eral terms, as follows. Assuming that an imperative program (as a whole) is deterministic, i.e. the output is completely determined by the input, we can say that the final state, or whichever fragments of it are of interest, is some function of the initial state, say $\sigma' = f(\sigma)$.[2] In functional programming this view is emphasized: the program is actually an expression that corresponds to the mathematical function f. Functional languages support the construction of such expressions by allowing rather powerful functional constructs.

Functional programming can be contrasted with imperative programming either in a negative or a positive sense. Negatively, functional programs do not use variables — there *is* no state. Consequently, they cannot use assignments, since there is nothing to assign to. Furthermore the idea of executing commands in sequence is meaningless, since the first command can make no difference to the second, there being no state to mediate between them. Positively however, functional programs can use functions in much more sophisticated ways. Functions can be treated in exactly the same way as simpler objects like integers: they can be passed to other functions as arguments and returned as results, and in general calculated with. Instead of sequencing and looping, functional languages use recursive functions, i.e. functions that are defined in terms of themselves. By contrast, most traditional languages provide poor facilities in these areas. C allows some limited manipulation of functions via pointers, but does not allow one to create new functions dynamically. FORTRAN does not even support recursion at all.

To illustrate the distinction between imperative and functional programming, the factorial function might be coded imperatively in C (without using C's unusual assignment operations) as:

```
int fact(int n)
{ int x = 1;
  while (n > 0)
   { x = x * n;
     n = n - 1;
   }
  return x;
}
```

whereas it would be expressed in ML, the functional language we discuss later, as a recursive function:

```
let rec fact n =
  if n = 0 then 1
  else n * fact(n - 1);;
```

nism is the *application* of functions to arguments.

[2] Compare Naur's remarks (Raphael 1966) that he can write any program in a single statement $Output = Program(Input)$.

In fact, this sort of definition can be used in C too. However for more sophisticated uses of functions, functional languages stand in a class by themselves.

1.1 The merits of functional programming

At first sight, a language without variables or sequencing might seem completely impractical. This impression cannot be dispelled simply by a few words here. But we hope that by studying the material that follows, readers will gain an appreciation of how it is possible to do a lot of interesting programming in the functional manner.

There is nothing sacred about the imperative style, familiar though it is. Many features of imperative languages have arisen by a process of abstraction from typical computer hardware, from machine code to assemblers, to macro assemblers, and then to FORTRAN and beyond. There is no reason to suppose that such languages represent the most palatable way for humans to communicate programs to a machine. After all, existing hardware designs are not sacred either, and computers are supposed to do our bidding rather than conversely. Perhaps the right approach is not to start from the hardware and work upwards, but to start with programming languages as an abstract notation for specifying algorithms, and then work *down* to the hardware (Dijkstra 1976). Actually, this tendency can be detected in traditional languages too. Even FORTRAN allows arithmetical expressions to be written in the usual way. The programmer is not burdened with the task of linearizing the evaluation of subexpressions and finding temporary storage for intermediate results.

This suggests that the idea of developing programming languages quite different from the traditional imperative ones is certainly defensible. However, to emphasize that we are not merely proposing change for change's sake, we should say a few words about why we might prefer functional programs to imperative ones.

Perhaps the main reason is that functional programs correspond more directly to mathematical objects, and it is therefore easier to reason about them. In order to get a firm grip on exactly what programs mean, we might wish to assign an abstract mathematical meaning to a program or command — this is the aim of *denotational semantics* (semantics = meaning). In imperative languages, this has to be done in a rather indirect way, because of the implicit dependency on the value of the state. In simple imperative languages, one can associate a command with a function $\Sigma \to \Sigma$, where Σ is the set of possible values for the state. That is, a command takes some state and produces another state. It may fail to terminate (e.g. `while true do x := x`), so this function may in general be partial. Alternative semantics are sometimes preferred, e.g. in terms of *predicate transformers* (Dijkstra 1976). But if we add features that can pervert the execution sequence in more complex ways, e.g. `goto`, or C's `break` and `continue`, even these interpretations no longer work, since one command can cause the later commands to be skipped. Instead, one typically uses a more complicated semantics based on *continuations*.

By contrast functional programs, in the words of Henson (1987), 'wear their semantics on their sleeves'.[3] We can illustrate this using ML. The basic datatypes have a direct interpretation as mathematical objects. Using the standard notation of $[\![X]\!]$ for 'the semantics of X', we can say for example that $[\![\text{int}]\!] = \mathbb{Z}$. Now the ML function `fact` defined by:

```
let rec fact n =
  if n = 0 then 1
  else n * fact(n - 1);;
```

has one argument of type `int`, and returns a value of type `int`, so it can simply be associated with an abstract partial function $\mathbb{Z} \to \mathbb{Z}$:

$$[\![\text{fact}]\!](n) = \begin{cases} n! & \text{if } n \geq 0 \\ \bot & \text{otherwise} \end{cases}$$

(Here $\bot$ denotes undefinedness, since for negative arguments, the program fails to terminate.) This kind of simple interpretation, however, fails in non-functional programs, since so-called 'functions' might not be functions at all in the mathematical sense. For example, the standard C library features a function `rand()`, which returns different, pseudo-random values on successive calls. This sort of thing can be implemented by using a local static variable to remember the previous result, e.g:

```
int rand(void)
{ static int n = 0;
  return n = 2147001325 * n + 715136305;
}
```

Thus, one can see the abandonment of variables and assignments as the logical next step after the abandonment of `goto`, since each step makes the semantics simpler. A simpler semantics makes reasoning about programs more straightforward. This opens up more possibilities for correctness proofs, and for provably correct transformations into more efficient programs.

Another potential advantage of functional languages is the following. Since the evaluation of expressions has no side-effect on any state, separate subexpressions can be evaluated in any order without affecting each other. This means that functional programs may lend themselves well to parallel implementation, i.e. the computer can automatically farm out different subexpressions to different

[3]More: denotational semantics can be seen as an attempt to turn imperative languages into functional ones by making the state explicit.

processors. By contrast, imperative programs often impose a fairly rigid order of execution, and even the limited interleaving of instructions in modern pipelined processors turns out to be complicated and full of technical problems.

Actually, ML is not a purely functional programming language; it does have variables and assignments if required. Most of the time, we will work inside the purely functional subset. But even if we do use assignments, and lose some of the preceding benefits, there are advantages too in the more flexible use of functions that languages like ML allow. Programs can often be expressed in a very concise and elegant style using higher-order functions (functions that operate on other functions).[4] Code can be made more general, since it can be parametrized even over other functions. For example, a program to add up a list of numbers and a program to multiply a list of numbers can be seen as instances of the same program, parametrized by the pairwise arithmetic operation and the corresponding identity. In one case it is given $+$ and 0 and in the other case, $*$ and 1.[5] Finally, functions can also be used to represent *infinite* data in a convenient way — for example we will show later how to use functions to perform exact calculation with real numbers, as distinct from floating point approximations.

At the same time, functional programs are not without their problems. Since they correspond less directly the the eventual execution in hardware, it can be difficult to reason about their exact usage of resources such as time and space. Input-output is also difficult to incorporate neatly into a functional model, though there are ingenious techniques based on infinite sequences.

It is up to readers to decide, after reading this book, on the merits of the functional style. We do not wish to enforce any ideologies, merely to point out that there *are* different ways of looking at programming, and that in the right situations, functional programming may have considerable merits. Most of our examples are chosen from areas that might loosely be described as 'symbolic computation', for we believe that functional programs work well in such applications. However, as always one should select the most appropriate tool for the job. It may be that imperative programming, object-oriented programming or logic programming are more suited to certain tasks. Horses for courses.

1.2 Outline

For those used to imperative programming, the transition to functional programming is inevitably difficult, whatever approach is taken. While some will be im-

[4] Elegance is subjective and conciseness is not an end in itself. Functional languages, and other languages like APL, often create a temptation to produce very short tricky code which is elegant to cognoscenti but obscure to outsiders.

[5] This parallels the notion of abstraction in pure mathematics, e.g. that the additive and multiplicative structures over numbers are instances of the abstract notion of a monoid. This similarly avoids duplication and increases elegance.

patient to get quickly to real programming, we have chosen to start with lambda calculus, and show how it can be seen as the theoretical underpinning for functional languages. This has the merit of corresponding quite well to the actual historical line of development.

So first we introduce lambda calculus, and show how what was originally intended as a formal logical system for mathematics turned out to be a completely general programming language. We then discuss why we might want to add types to lambda calculus, and show how it can be done. This leads us into ML, which is essentially an extended and optimized implementation of typed lambda calculus with a certain evaluation strategy. We cover the practicalities of basic functional programming in ML, and discuss polymorphism and most general types. We then move on to more advanced topics including exceptions and ML's imperative features. We conclude with some substantial examples, which we hope provide evidence for the power of ML.

Further reading

Numerous textbooks on 'functional programming' include a general introduction to the field and a contrast with imperative programming — browse through a few and find one that you like. For example, Henson (1987) contains a good introductory discussion, and features a similar mixture of theory and practice to this text. A detailed and polemical advocacy of the functional style is given by Backus (1978), the main inventor of FORTRAN. Gordon (1994) discusses the problems of incorporating input-output into functional languages, and some solutions. Readers interested in denotational semantics, for imperative and functional languages, may look at Winskel (1993).