Programming in Standard ML '97: A Tutorial Introduction

Stephen Gilmore
Laboratory for Foundations of Computer Science
The University of Edinburgh

September 1997
(Revised: July 1998)

Copyright notice

Disclaimer of warranty

About this document

These are the lecture notes from an eighteen-lecture Master of Science course given in the Department of Computer Science at The University of Edinburgh between 1992 and 1997. An on-line version was developed in co-operation with Harlequin Ltd. It can be obtained from the following WWW locations:

- http://www.harlequin.co.uk
- http://www.dcs.ed.ac.uk/home/stg

The following people have found errors in previous versions of these notes or suggested improvements: Anthony Bailey, Jo Brook, Elaine Farrow, Jon Hanson, Stefan Kahrs, Dan Russell and Don Sannella. Any remaining errors are solely the fault of the author. If you have comments or suggestions for improvements or clarifications please contact the author at the address below.

Laboratory for Foundations of Computer Science
The James Clerk Maxwell Building
The King's Buildings
University of Edinburgh
Edinburgh EH9 3JZ, UK

Email: stg@dcs.ed.ac.uk

Contents

1 Introduction **1**

1.1 Standard ML . . . 2
1.2 Programming in practice . . . 3
1.3 Reading material . . . 3
1.4 Other information . . . 4

2 Simple applicative programming **5**

2.1 Types, values and functions . . . 5
2.2 Defining a function by cases . . . 6
2.3 Scope . . . 7
2.4 Recursion . . . 8
2.5 Scoping revisited . . . 9
2.6 The Standard ML library . . . 10
2.6.1 The Bool structure . . . 10
2.6.2 The Byte structure . . . 10
2.6.3 The Char structure . . . 10
2.6.4 The Int structure . . . 11
2.6.5 The Real structure . . . 11
2.6.6 The String structure . . . 11
2.6.7 The StringCvt structure . . . 12
2.6.8 The Word and Word8 structures . . . 12

3 Higher-order programming **13**

3.1 Higher-order functions . . . 14
3.2 Curried functions . . . 15
3.3 Function composition . . . 16
3.4 Derived forms . . . 16

4 Types and type inference **19**

4.1 Type inference . . . 19
4.2 Pairs and record types . . . 20
4.3 Function types and type abbreviations . . . 20
4.4 Defining datatypes . . . 21
4.5 Polymorphism . . . 23
4.5.1 Function composition . . . 24
4.5.2 Default overloading . . . 24
4.6 Ill-typed functions . . . 25
4.7 Computing types . . . 26

5 Aggregates **27**
5.1 Lists . . . 27
5.2 Induction for lists . . . 29
5.3 List processing . . . 30
5.3.1 Selecting from a list . . . 31
5.3.2 Sorting lists . . . 32
5.3.3 List functions . . . 33
5.4 The tree datatype . . . 35
5.5 Converting trees to lists . . . 35
5.6 Induction for trees . . . 35
5.7 The vector datatype . . . 36
5.8 The Standard ML library . . . 37
5.8.1 The List structure . . . 37
5.8.2 The ListPair structure . . . 37
5.8.3 The Vector structure . . . 37

6 Evaluation **38**
6.1 Call-by-value, call-by-name and call-by-need . . . 38
6.2 Delaying evaluation . . . 39
6.3 Forcing evaluation . . . 40
6.4 From call-by-value to call-by-name . . . 41
6.5 Lazy datatypes . . . 41
6.6 An example: Computing the digits of e . . . 42

7 Abstract data types **44**
7.1 Programming with abstract data types . . . 45

8 Imperative programming **47**
8.1 References . . . 47
8.2 Assignment . . . 49
8.3 Sequential composition . . . 49
8.4 Iteration . . . 50
8.5 Types and imperative programming . . . 50
8.5.1 Type safety conditions . . . 51
8.5.2 Implementing type safety . . . 51
8.6 Arrays . . . 52
8.7 Memoisation . . . 53
8.8 Input/output . . . 53

9 Introducing Standard ML Modules **57**
9.1 Signatures . . . 57
9.2 Structures . . . 57
9.3 Representation independence and equality . . . 58

Bibliography **60**

Index **62**

Chapter 1

Introduction

Standard ML is a functional programming language, and it is more than that. Functional programs have many virtues. They are concise, secure and elegant. They are easier to understand and easier to prove correct than imperative programs.

Functional—or applicative—languages relieve the programmer of many of the difficulties which regrettably occupy much of the attention of a programmer working in imperative—or procedural—languages. These difficulties range from implementing re-usable routines to conserving memory to mastering the representation of data values. Some of these have attendant responsibilities. Once programmers understand how values are represented in memory they must subsequently ensure that they are inspected and updated in a manner which is consistent with that representation. Even after the difficulties of managing the machine's primary memory have been comprehended, there are still the difficulties of transferring data from primary memory to secondary memory and the complications of disk access mechanisms and input/output libraries.

A functional programmer can transcend these matters. Applicative languages have secure polymorphic type systems which simplify the task of writing re-usable, general-purpose routines. Applicative programmers delegate the task of memory management to "garbage collection" routines which need only be written once for the implementation of the language; not every time that another program is written in the language. Applicative languages hide the machine representation of data, making it impossible to write programs which are sensitive to the byte order of the underlying machine or to introduce other unintended dependencies. Using disk input routines can be avoided because of the interactive nature of applicative languages which allows data to be entered with the minimum of fuss. Using disk output routines can be avoided by allowing the user to export the environment bindings as easily as checkpointing a database. Of course any realistic programming language must offer input and output routines for disk, screen and keyboard but their use can be avoided for many applicative programming problems.

Functional languages do have shortcomings. It might be said that some programming problems appear to be inherently state-based. The uniformity of representation which applicative programming languages offer then becomes a handicap. Although an applicative solution to such a problem would be possible it might have an obscure or unnatural encoding. Another possible worry is that the functional implementation might be inefficient when compared with a straightforward imperative one. This would be a shame; a programming language should not make it difficult for a programmer to write efficient programs.

Imperative programming languages also have their strengths. They often provide explicit support for the construction of programs on a large scale by offering simple, robust modules which allow a large programming task to be decomposed into self-contained units to be implemented in isolation. If these units are carefully crafted and general they may be included in a library, facilitating their re-use in other programs. Modules also enable programs to be efficiently recompiled by avoiding the need to recompile parts of the program which have not changed. By adding high-level constructs to an imperative language one can elevate the practice of programming. The only reason not to provide high-level constructs in the language itself is that optimisations can sometimes be made if a programmer tailors every instance of a general high-level routine to take account of the particular idiosyncracies of the programming task at hand. Where is the profit in this? One is often only exchanging a saving in a relatively cheap resource, such as memory, for an increased amount of an expensive service, that of application programmer time. The present state of programming is that well-designed modern imperative languages such as Java [AG96] come close to offering the type security and programming convenience which functional programming languages offered in the 1970's. The interest in and increasing adoption of this language should be viewed as an encouraging, if slow, shift away from the miserliness which has bedevilled programming practice since the inception of the profession.

Is there a way to combine the virtues of well-designed applicative programming languages with the virtues of well-designed imperative ones? Of course, the resulting language might never become the world's most frequently used programming language. Perhaps programming simply is not a virtuous activity or perhaps the resulting language would be a hideous mutant; sterile and useless for programming. We think that computer scientists should feel that they have a moral duty to investigate the combination of the best of these two kinds of programming languages. In this author's opinion, the Standard ML programming language provides the most carefully designed and constructed attempt so far to develop a language to promote the relative virtues embodied in well-designed applicative and imperative programming languages.

1.1 Standard ML

The Standard ML programming language is defined formally. This definition is presented as a book of rules [MTHM97] expressed in so-called Natural Semantics, a powerful but concise formalism which records the essential essence of the language while simultaneously abstracting away from the uninformative detail which would inevitably be needed in a programming language implementation. As a comparison, the published model implementation of Standard Pascal [WH86] is five times longer than the definition of Standard ML. This is a shocking fact because Standard ML is a much more sophisticated language than Pascal. In these notes we follow the 1997 revision of the language definition.

Standard ML consists of a core language for small-scale programming and a module system for large-scale programming. The Standard ML core language is not a pure applicative programming language, it is a higher-order procedural language with an applicative subset. For most of these notes the focus will be on using the applicative subset of the language so we will spend some time discussing functions, in particular recursive functions, and giving verification techniques for these. In the early sections of these notes the examples are all small integer functions. Later, Standard ML's sophisticated type system is presented. The design of the Standard ML programming language enables the type of any value to be computed, or inferred, by the Standard ML system. Thus, the language has a strong, safe type system but does not force the programmer to state the type of every value before use. This type system facilitates the detection of programming errors before a program is ever executed.

John Hughes in [Hug89] suggests that one of the advantages of functional programming is the ability to glue programs together in many different ways. For this, we require the ability to manipulate functions as data. We will pass them as arguments and even return them as results. In order for such a powerful mechanism to be exploited to the fullest, the language must provide a means for functions to be defined in a general, re-usable way which allows values of different types to be passed to the same function. As we shall see, Standard ML provides such a feature without compromising the security of the type system of the language.

The mechanism by which applicative programs are evaluated is then discussed. This leads into the consideration of alternative evaluation strategies one of which is so-called lazy evaluation. This strategy has a profound effect on the style of programming which must be deployed.

The other elements of the Standard ML core language which are discussed are the mechanism used to signal that an exceptional case has been detected during processing and no consistent answer can be returned and the imperative features such as references and input/output.

The core language offers a comfortable and secure environment for the development of small programs. However, a large Standard ML program would contain many definitions (of values, functions or types) and we begin to require a method of packaging together related definitions; for example, a type and a useful collection of functions which process elements of this type. Standard ML has modules called *structures*. Structures facilitate the division of a large program into a number of smaller, independent units with well-defined, explicit connections. A large programming task may then be broken up so that several members of a programming team may independently produce structures which are assembled to form a single program.

Another reason for tidying collections of definitions into a structure is that we can pass structures as arguments and return them as results. In Standard ML the entity which plays the role of a function mapping structures to structures is called a *functor*.

Finally, we may wish to take the opportunity when collecting definitions into a structure to hide some of the declarations which played a minor role in helping the implementor to construct the major definitions of the structure. Thus, we require a *interface* for our structure. The Standard ML term for such an interface is a *signature*.

1.2 Programming in practice

Let's not kid ourselves. Switching to a better programming language or following a prescribed methodical approach to programming is no panacea to solve all of the problems which can arise in software development. For example, it is perfectly easy to imagine a correct program which is difficult or inconvenient to use. It is also perfectly easy to imagine users finding fault with a correct program which has some missing functionality which they would like. If a feature is never specified in the first place in an initial, abstract specification then nothing in a methodical program development approach will go wrong as the program is being developed. The forgotten feature, in Knuth's terminology [Knu89], will remain forgotten.

A clean, high-level programming language is simply a powerful tool in a programmer's toolset. With it the creation of saleable, efficient, secure and well-engineered programs will still remain a demanding intellectual activity which requires original, creative and careful thought.

1.3 Reading material

Two excellent textbooks to help anyone learning Standard ML are Paulson's "ML for the Working Programmer (Second Edition)" [Pau96] and Ullman's "Elements of ML Programming (ML '97 edition)" [Ull98]. These textbooks use the 1997 revision of the Standard ML language—sometimes called SML '97. Other textbooks refer to the 1990 issue of the language standard, and some even pre-date that. Another on-line tutorial on SML '97 is Harper's "Programming in Standard ML" which is available from his Web page at Carnegie Mellon University at `http://www.cs.cmu.edu`.

Good books which refer to the 1990 revision of the language are Sokołowski's "Applicative High-Order Programming" [Sok91] and Reade's "Elements of Functional Programming" [Rea89].

Tofte's "Four lectures on Standard ML" [Tof89] concentrates primarily on modules. A careful account of the static semantics of modules is given which makes clear the crucial notions of *sharing* and *signature matching*.

Textbooks suitable for someone with no programming experience are Bosworth's "A practical course in programming using Standard ML", Michaelson's "Elementary Standard ML", Wikström's "Functional programming using Standard ML" and "Programming with Standard ML" by Myers, Clack and Poon. Wikström's book is quite dated and some of the code fragments which appear in the book will no longer be accepted by Standard ML because of revisions to the language.

The definitive definition of the Standard ML programming language is given in "The Definition of Standard ML (Revised 1997)" by Milner, Tofte, Harper and MacQueen. This is not a reference manual for the language: rather it is a formal definition giving rules which define the simple constructs of the language in terms of familiar mathematical objects and define the complex constructs of the language in terms of the simpler ones. The definition is complemented by Milner and Tofte's "Commentary on Standard ML" which relates the definition to the user's view of the language.

1.4 Other information

Other material on Standard ML is available from the accompanying World-Wide Web page located at `http://www.dcs.ed.ac.uk/home/stg/tutorial/`. It is updated frequently.

Chapter 2

Simple applicative programming

Standard ML is an interactive language. Expressions are entered, compiled and then evaluated. The result of evaluation is displayed and the next expression may then be entered. This interactive style of working combines well with Standard ML's *type inference* mechanism to empower the programmer to work in a flexible, experimental way, moving freely from defining new functions to trying the function on some test data and then either modifying the function or moving on to define another.

The fact that types are assigned by the compiler also has the favourable consequence that Standard ML functions are usually shorter than comparable routines implemented in languages in which the types of variables must be supplied when the variable is declared.

We will begin to investigate applicative programming by inspecting values, expressions and functions. The functions defined are short and we will not spend much time describing the tasks to be computed since they will often be self-evident. Some simple and generally useful functions are pre-defined in Standard ML; these include arithmetic functions and others for processing strings and characters. These pre-defined functions are said to be in the initial SML basis, or environment. In addition, the language also provides a library which makes available other functions, values and types. We will briefly mention this at the end of this chapter.

2.1 Types, values and functions

A pre-defined type in Standard ML is the type of truth values, called `bool`. This has exactly two values, `true` and `false`. Another simple type is `string`. Strings are enclosed in double quotes (as in `"Hello"`) and are joined by "`^`" (the caret or up arrow symbol). As expected, the expression `"Hello" ^ " world!"` evaluates to `"Hello world!"`. There also is a type `char` for single characters. A character constant is a string constant of length one preceded by a hash symbol. Thus `#"a"` is the letter a, `#"\n"` is the newline character, `#"\t"` is tab and the backslash character can be used in the expected way to quote itself or the double quote character. Eight-bit characters can be accessed by their ASCII code, with `#"\163"` yielding `#"£"`, `#"\246"` yielding `#"ö"` and so forth. The function `explode` turns a string into a list of single characters and `implode` goes the other way. The function `ord` turns a character into its ASCII code and `chr` goes the other way. The function `size` returns the number of characters in a string. Because strings and characters are values of different types we need a function to convert a character into a string of length one. The function `str` is used for this.

The numeric types in Standard ML are the integers of type `int`, the real numbers of type `real`, and unsigned integers (or words) of type `word`. In addition to these, an implementation of the language may provide other numeric types in a library, for example perhaps arbitrary precision integers or double precision reals or even bytes (8-bit unsigned integers). The integers may be represented in decimal or hexadecimal notation thus `255` and `0xff` (zero x ff) both represent the integer 255. The numbers have one striking oddity "`~`" (tilde) is used for unary minus (with the exception of words of course, which cannot be negative). Reals must have either a fraction part—as in `4000.0`—or an exponent part—as in `4E3`—or both. A word or byte constant is written as a decimal numeral if following the characters `0w` (zero w) or as a hexadecimal numeral if following the characters `0wx` (zero w x). Thus `0w255` and `0wxff` are two different ways of writing the word constant 255. Integers, reals and words are thus lexically distinct and when we see a numeric constant we can infer its type, if we are not explicitly told what it is. Operators such as +, −, `*`, `div` and `mod` are provided for integers and words: +, −, `*` and `/` are provided for the reals. Functions such as absolute value, `abs`, and unary negation are provided for the signed numeric types only. Relational operators `=`, `<>`, `<`, `<=`, `>` and `>=` are provided for the numeric types.

The numeric types which we have mentioned are *separate types* and we must use functions to convert an integer to a real or a word or a byte. There is no implicit coercion between types as found in other programming languages. Standard ML provides two library functions to convert between integers and reals. Arbitrary precision integer arithmetic is provided by some implementations of the language. In order to determine which do and which do not, consult the reference manual for the implementation or calculate a large integer value, say by multiplying a million by itself.

The integer successor function may be denoted by `fn x => x+1`. Function application is indicated by juxtaposition of the function—or function expression—and the argument value—or argument expression. Parentheses are introduced where necessary. If the argument to the function is to be obtained by evaluating an expression then parentheses will be obligatory. They can be used at other times just to clarify the structure of the function application. As might be expected, the function application `(fn x => x+1) 4` evaluates to `5`. Function application associates to the left in Standard ML so an expression which uses the successor function twice must use parentheses, as in `(fn x => x+1) ((fn x => x+1) 4)`. Since unary minus is a function the parentheses are necessary in the expression `~ (~ x)`.

Of course, a mechanism is provided for binding names to values; even titchy programs would be unreadable without it. The declaration `val succ = fn x => x+1` binds the name `succ` to the successor function and we may now write `succ (succ 4)` as an abbreviation for the somewhat lengthy expression `(fn x => x+1) ((fn x => x+1) 4)`.

Standard ML is a case-sensitive language. All of the reserved words of the language, such as *val* and *fn*, must be in lower case and occurrences of a program identifier must use capitalization consistently.

2.2 Defining a function by cases

Standard ML provides a mechanism whereby the notation which introduces the function parameter may constrain the type or value of the parameter by requiring the parameter to match a given pattern (so-called "pattern matching"). The following function, `day`, maps integers to strings.

```
val day = fn 0 => "Monday"
           | 1 => "Tuesday"
           | 2 => "Wednesday"
           | 3 => "Thursday"
           | 4 => "Friday"
           | 5 => "Saturday"
           | _ => "Sunday"
```

The final case in the list is a catch-all case which maps any value other than those listed above it to `"Sunday"`. Be careful to use double quotes around strings rather than single quotes. Single quote characters are used for other purposes in Standard ML and you may receive a strange error message if you use them incorrectly.

2.3 Scope

Armed with our knowledge about integers and reals and the useful "pattern matching" mechanism of Standard ML we are now able to implement a simple formula. A Reverend Zeller discovered a formula which calculates the day of the week for a given date. If d and m denote day and month and y and c denote year and century with each month decremented by two (January and February becoming November and December of the previous year) then the day of the week can be calculated according to the following formula.

$$(\lfloor 2.61m - 0.2 \rfloor + d + y + y \div 4 + c \div 4 - 2c) \bmod 7$$

This is simple to encode as a Standard ML function, `zc`, with a four-tuple as its argument if we know where to obtain the conversion functions in the Standard ML library. We will bind these to concise identifier names for ease of use. We choose the identifier `floor` for the real-to-integer function and `real` for the integer-to-real function. Note the potential for confusion because the name `real` is simultaneously the name of a function and the name of a type. Standard ML maintains different name spaces for these.

```
val floor = Real.floor
val real = Real.fromInt
val zc =
  fn (d, m, y, c) =>
    (floor (2.61 * real m - 0.2) + d + y + y div 4 + c div 4 - 2 * c) mod 7
```

Now we may use the pattern matching mechanism of Standard ML to perform the adjustment required for the formula to calculate the days correctly.

```
val zeller = fn (d, 1, y)  => zc (d, 11, (y - 1) mod 100, (y - 1) div 100 + 1)
              | (d, 2, y)  => zc (d, 12, (y - 1) mod 100, (y - 1) div 100 + 1)
              | (d, m, y) => zc (d, m - 2, y mod 100, y div 100 + 1)
```

Although the `zeller` function correctly calculates days of the week its construction is somewhat untidy since the `floor`, `real` and `zc` functions have the same status as the `zeller` function although they were intended to be only sub-components. We require a language construct which will bundle the four functions together, making the inferior ones invisible. In Standard ML the *local* .. *in* .. *end* construct is used for this purpose.

```
local
  val floor = Real.floor
  val real = Real.fromInt
  val zc =
    fn (d, m, y, c) =>
      (floor (2.61 * real m - 0.2) + d + y + y div 4 + c div 4 - 2 * c) mod 7
in
  val zeller = fn (d, 1, y)  => zc (d, 11, (y - 1) mod 100, (y - 1) div 100 + 1)
                | (d, 2, y)  => zc (d, 12, (y - 1) mod 100, (y - 1) div 100 + 1)
                | (d, m, y) => zc (d, m - 2, y mod 100, y div 100 + 1)
end
```

Here we have three functions declared between the keyword *local* and the keyword *in* and one function defined between the keyword *in* and the keyword *end*. In general we could have a sequence of declarations in either place so a utility function could be shared between several others yet still not be made generally available. Either or both of the declaration sequences in *local* .. *in* .. *end* can be empty.

2.4 Recursion

Consider the simple problem of summing the numbers from one to n. If we know the following equation then we can implement the function easily using only our existing knowledge of Standard ML.

$$1 + 2 + \cdots + n \quad = \quad \frac{n(n+1)}{2}$$

The function is simply `fn n => (n * (n+1)) div 2`. We will call this function `sum`. But what if we had not known the above equation? We would require an algorithmic rather than a formulaic solution to the problem. We would have been forced to break down the calculation of the sum of the numbers from one to n into a number of smaller calculations and devise some strategy for recombining the answers. We would immediately use the associative property of addition for this purpose.

$$1 + 2 + \cdots + n \quad = \quad \underbrace{(\cdots(1+2) + \cdots + n-1)}_{\text{Sum of 1 to } n-1} + n$$

We now have a trivial case—when n is one—and a method for decomposing larger cases into smaller ones. These are the essential ingredients for a recursive function.

$$sum'(n) = \begin{cases} 1 & \text{if } n \text{ is one,} \\ sum'(n-1) + n & \text{otherwise.} \end{cases} \tag{2.1}$$

In our implementation of this definition we must mark our function declaration as being recursive in character using the Standard ML keyword *rec*. The only values which can be defined recursively in Standard ML are functions.

```
val rec sum' = fn 1 => 1
                | n => sum'(n - 1) + n
```

If you have not seen functions defined in this way before then it may seem somewhat worrying that the `sum'` function is being defined in terms of itself but there is no trickery here. The equation $n = 5n - 20$ defines the value of the natural number n precisely through reference to itself and the definition of the `sum'` function is as meaningful.

We would assert for positive numbers that `sum` and `sum'` will always agree but at present this is only a claim. Fortunately, the ability to encode recursive definitions of functions comes complete with its own proof method for checking such claims. The method of proof is called induction and the form of induction which we are using here is simple integer induction.

The intent behind employing induction here is to construct a convincing argument that the functions `sum` and `sum'` will always agree for positive numbers. The approach most widely in use in programming practice to investigate such a correspondence is to choose a set of numbers to use as test data and compare the results of applying the functions to the numbers in this set. This testing procedure may uncover errors if we have been fortunate enough to choose one of the positive integers—if there are any—for which `sum` and `sum'` disagree. One of the advantages of testing programs in this way is that the procedure is straightforwardly amenable to automation. After some initial investment of effort, different versions of the program may be executed with the same input and the results compared mechanically. However, in this problem we have very little information about which might be the likely values to uncover differences. Sadly, this is often the case when attempting to check the fitness of a computer program for a given purpose.

Rather than simply stepping through a selection of test cases, a proof by induction constructs a general argument. We will show that the equivalence between `sum` and `sum'` holds for the smallest allowable value—one in this case—and then show that if the equivalence holds for n it must also hold for $n + 1$. The first step in proving any result, even one as simple as this, is to state the result clearly. So we will do that next.

Proposition 2.4.1 *For every positive n, we have that $1 + 2 + \cdots + n = n(n + 1)/2$.*

Proof: Considering $n = 1$, we have that LHS $= 1 = 1(1+1)/2 =$ RHS. Now assume the proposition is true for $n = k$ and consider $n = k + 1$.

$$
\begin{aligned}
\text{LHS} &= 1 + 2 + \cdots + k + (k + 1) \\
&= k(k + 1)/2 + (k + 1) \\
&= (k^2 + 3k + 2)/2 \\
&= (k + 1)(k + 2)/2 \\
&= (k + 1)((k + 1) + 1)/2 \\
&= \text{RHS}
\end{aligned}
$$

It would be very unwise to appeal to the above proof and claim that the functions sum and sum′ are indistinguishable. In the first place, this is simply not true since they return different answers for negative numbers. What we may claim based on the above proof is that when both functions return a result for a positive integer, it will be the same result. More information on using induction to prove properties of functions may be found in [MNV73].

2.5 Scoping revisited

The construction *local* .. *in* .. *end* which we saw earlier is used to make one or more declarations local to other declarations. At times we might wish to make some declarations local to an expression so Standard ML provides *let* .. *in* .. *end* for that purpose. As an example of its use consider a simple recursive routine which converts a non-negative integer into a string. For a little extra effort we also make the function able to perform conversions into bases such as binary, octal and hexadecimal. The function is called radix and, for example, an application of the form radix (15, "01") returns "1111", the representation of fifteen in binary.

```
(* This function only processes non-negative integers.  *)

val rec radix = fn (n, base) =>
  let
    val b = size base
    val digit = fn n => str (String.sub (base, n))
    val radix' =
        fn (true, n) => digit n
         | (false, n) => radix (n div b, base) ^ digit (n mod b)
  in
    radix' (n < b, n)
  end
```

This implementation uses a function called sub to subscript a string by numbering beginning at zero. The sub function is provided by the String component of the Standard ML library and is accessed by the long identifier String.sub.

Notice one other point about this function. It is only the function radix which must be marked as a recursive declaration—so that the call from radix′ will make sense—the two functions nested inside the body of radix are not recursive.

Exercise 2.5.1 (Zeller's congruence) *The implementation of the function* zeller *can be slightly simplified if* zeller *is a recursive function. Implement this simplification.*

Exercise 2.5.2 *Devise an integer function* sum′′ *which agrees with* sum *everywhere that they are both defined but ensure that* sum′′ *can calculate sums for numbers larger than the largest* *argument to* sum *which produces a defined result. Use only the features of the Standard ML language introduced so far. You may wish to check your solution on an implementation of the language which does not provide arbitrary precision integer arithmetic.*

Exercise 2.5.3 *Consider the following function,* sq.

```
val rec sq = fn 1 => 1
              | n => sq(n - 1) + (2 * n - 1)
```

Prove by induction that $1 + \cdots + (2n - 1) = n^2$*, for positive* n.

2.6 The Standard ML library

As promised, let us consider some of the components in the Standard ML library. We will focus on those functions which work with the Standard ML types which we have met and defer inspection of more advanced features of the library until later.

The components of the Standard ML library are Standard ML modules, called *structures*. We can think of these as boxed-up collections of functions, values and types.

2.6.1 The Bool structure

The Bool structure is a very small collection of simple functions to manipulate boolean values. It includes the boolean negation function, Bool.not, and the string conversion functions Bool.toString and Bool.fromString. The latter function is not quite as simple as might be expected because it must signal that an invalid string does not represent a boolean. This is achieved though the use of the option datatype and the results from the Bool.fromString function are either NONE or SOME b, where b is a boolean value. This use of the optional variant of the result is common to most of the library functions which convert values from strings.

2.6.2 The Byte structure

The Byte structure is another small collection of simple functions, this time for converting between characters and eight-bit words. The function Byte.byteToChar converts an eight-bit word to a character. The function Byte.charToByte goes the other way.

2.6.3 The Char structure

Operations on characters are found in the Char structure. These include the integer equivalents of the word-to-character conversion functions, Char.chr and Char.ord. Also included are successor and predecessor functions, Char.succ and Char.pred. Many functions perform obvious tests on characters, such as Char.isAlpha, Char.isUpper, Char.isLower, Char.isDigit, Char.isHexDigit and Char.isAlphaNum. Some exhibit slightly more complicated behaviour, such as Char.isAscii which identifies seven-bit ASCII characters; and Char.isSpace which identifies whitespace (spaces, tabulate characters, line break and page break characters); Char.isPrint identifies printable characters and Char.isGraph identifies printable, non-whitespace characters. The functions Char.toUpper and Char.toLower behave as expected, leaving non-lowercase (respectively non-uppercase) characters unchanged. The pair of complementary functions Char.contains and Char.notContains may be used to determine the presence or absence of a character in a string. Relational operators on characters are also provided in the Char structure.

2.6.4 The Int structure

The Int structure contains many arithmetic operators. Some of these, such as Int.quot and Int.rem, are subtle variants on more familiar operators such as div and mod. Quotient and remainder differ from divisor and modulus in their behaviour with negative operands. Other operations on integers include Int.abs and Int.min and Int.max, which behave as expected. This structure also provides conversions to and from strings, named Int.toString and Int.fromString, of course. An additional formatting function provides the ability to represent integers in bases other than decimal. The chosen base is specified using a type which is defined in another library structure, the string convertors structure, StringCvt. This permits very convenient formatting of integers in binary, octal, decimal or hexadecimal, as shown below.

```
Int.fmt StringCvt.BIN 1024  = "10000000000"
Int.fmt StringCvt.OCT 1024  = "2000"
Int.fmt StringCvt.DEC 1024  = "1024"
Int.fmt StringCvt.HEX 1024  = "400"
```

The Int structure may define largest and smallest integer values. This is not to say that structures may arbitrarily shrink or grow in size, Int.maxInt either takes the value NONE or SOME i, with i being an integer value. Int.minInt has the same type.

2.6.5 The Real structure

It would not be very misleading to say that the Real structure contains all of the corresponding real equivalents of the integer functions from the Int structure. In addition to these, it provides conversions to and from integers. For the former we have Real.trunc, which truncates; Real.round, which rounds up; Real.floor which returns the greatest integer less than its argument; and Real.ceil, which returns the least integer greater than its argument. For the latter we have Real.fromInt, which we have already seen. One function which is necessarily a little more complicated than the integer equivalent is the Real.fmt function which allows the programmer to select between scientific and fixed-point notation or to allow the more appropriate format to be used for the value being formatted. In addition to this a number may be specified; it will default to six for scientific or fixed-point notation and twelve otherwise. The fine distinction between the import of the numbers is that it specifies decimal places for scientific and fixed-point notation and significant digits otherwise. Functions cannot vary the number of arguments which they take so once again the option type is used to signify the number of required digits. Here are some examples of the use of this function. Note that the parentheses are needed in all cases.

```
Real.fmt (StringCvt.FIX NONE) 3.1415       = "3.141500"
Real.fmt (StringCvt.SCI NONE) 3.1415       = "3.141500E00"
Real.fmt (StringCvt.GEN NONE) 3.1415       = "3.1415"
Real.fmt (StringCvt.FIX (SOME 3)) 3.1415   = "3.142"
Real.fmt (StringCvt.SCI (SOME 3)) 3.1415   = "3.142E00"
Real.fmt (StringCvt.GEN (SOME 3)) 3.1415   = "3.14"
```

2.6.6 The String structure

The String structure provides functions to extract substrings and to process strings at a character-by-character level. These include String.substring which takes a triple of a string and two integers to denote the start of the substring and its length. Indexing is from zero, a convention which is used throughout the library. The String.extract function enables the programmer to leave the end point of the substring unspecified using the option NONE. The meaning of this is that there is to be no limit on the amount of the string which is taken, save that imposed by the string itself. Thus for example, String.extract (s, 0, NONE) is always the same as s itself, even if s is empty. The String.sub function may be used to subscript a string to obtain the character at a given position. Lists of strings may be concatenated using String.concat.

Several high-level functions are provided for working with strings. Two of the most useful are a tokeniser, `String.tokens`, and a separator, `String.fields`. These are parameterised by a function which can be used to specify the delimiter which comes between tokens or between fields. The principal distinction between a token and a field is that a token may not be empty, whereas a field may. Used together with the character classification functions from the `Char` structure these functions provide a simple method to perform an initial processing step on a string, such as dividing it into separate words. Another useful function is `String.translate` which maps individual characters of a string to strings which are then joined. Used together with `String.tokens`, this function provides a very simple method to write an accurate and efficient lexical analyser for a formal language. The method is simply to pad out special characters with spaces and then to tokenize the result naïvely.

2.6.7 The StringCvt structure

In the string convertors structure, `StringCvt`, are several of the specifiers for forms of conversion from numbers to strings but also some simple functions which work on strings, such as `StringCvt.padLeft` and `StringCvt.padRight`. Both functions require a padding character, a field width and a string to be padded.

2.6.8 The Word and Word8 structures

The `Word` and `Word8` structures provide the same collection of functions which differ in that they operate on different versions of a type called, respectively, `Word.word` and `Word8.word`. Almost all of the functions are familiar to us now from the `Int` and `Real` structures. The `Word` and `Word8` structures also provide bitwise versions of the operators and, or, exclusive or and not—`andb`, `orb`, `xorb` and `notb`.

Chapter 3

Higher-order programming

As we have seen, functions defined by pattern matching associate patterns and expressions. Functions defined in this way are checked by Standard ML. One form of checking is to ensure that all of the patterns describe values from the same data type and all of the expressions produce values of the same type. Both of the functions below will be rejected because they do not pass these checks.

```
(* error *)
val wrong_pat = fn 1    => 1
                 | true => 1
```

```
(* error *)
val wrong_exp = fn 1 => true
                 | n => "false"
```

Pattern matching provides subsequent checking which, if failed, will not cause the function to be rejected but will generate warning messages to call attention to the possibility of error. This subsequent checking investigates both the extent of the patterns and the overlap between them. The first version of the boolean negation function below is incomplete because the matches in the patterns are not exhaustive; there is no match for true. The second version has a redundant pattern; the last one. Both produce compiler warning messages.

```
(* warning *)
val not1 = fn false => true
```

```
(* warning *)
val not2 = fn false => true
            | true  => false
            | false => false
```

Both functions can be used. The first will only produce a result for false but because of the first-fit pattern matching discipline the second will behave exactly like the Bool.not function in the Standard ML library. The warning message given for the second version signals the presence of so-called dead code which will never be executed.

The checking of pattern matching by Standard ML detects flawed and potentially flawed definitions. However, this checking is only possible because the allowable patterns are much simpler than the expressions of the language. Patterns may contain constants and variables and the wild card which is denoted by the underscore symbol. They may also use constructors from a data type—so far we have only met a few of these including false and true from the bool data type and SOME and NONE from the option data type. Constructors are distinct from variables in patterns because constructors can denote only themselves, not other values. Patterns place a restriction on the use of variables; they may only appear once in each pattern. Thus the following function definition is illegal.

```
(* error *)
val same = fn (x, x) => true
            | (x, y) => false
```

Similarly, patterns cannot contain uses of functions from the language. This restriction means that neither of the attempts to declare functions which are shown below are permissible.

```
(* error *)
val abs = fn (~x) => x
           | 0    => 0
           | x    => x
```

```
(* error *)
val even = fn (x + x)     => true
            | (x + x + 1) => false
```

Neither can we decompose functions by structured patterns, say in order to define a test on functions. Attempts such as the following are disallowed.

```
(* error *)
val is_identity = fn (fn x => x) => true
                   | _           => false
```

However, we can use pattern matching to bind functions to a local name because one of the defining characteristics of functional programming languages is that they give the programmer the ability to manipulate functions as easily as manipulating any other data item such as an integer or a real or a word. Functions may be passed as arguments or returned as results. Functions may be composed in order to define new functions or modified by the application of higher-order functions.

3.1 Higher-order functions

In the previous chapter two small functions were defined which performed very similar tasks. The sum′ function added the numbers between 1 and n. The sq function added the terms $2i - 1$ for i between 1 and n. We will distill out the common parts of both functions and see that they can be used to simplify the definitions of a family of functions similar to sum′ and sq. The common parts of these two functions are:

- a contiguous range with a lower element and upper element;
- a function which is applied to each element in turn;
- an operator to combine the results; and
- an identity element for the operator.

We will define a function which takes a five-tuple as its argument. Two of the elements in the five-tuple are themselves functions and it is for this reason that the function is termed higher-order. Functions which take functions as an argument—or, as here, as part of an argument—are called higher-order functions.

The function we will define will be called reduce. Here is the task we wish the reduce function to perform.

```
reduce (g, e, m, n, f)   ≡   g(e, g(f(m), g(f(m+1), ··· g(f(n−1), g(f(n))) ··· )))
```

The function g may be extremely simple, perhaps addition or multiplication. The function f may also be extremely simple, perhaps the identity function, *fn* x => x.

In order to implement this function, we need to decide when the reduction has finished. This occurs when the value of the lower element exceeds the value of the upper element, m > n. If this condition is met, the result will be the identity element e. If this condition is not met, the function g is applied to the value obtained by reducing the remainder of the range and f n.

We would like the structure of the implementation to reflect the structure of the analysis of termination given above so we shall implement a sub-function which will assess whether or not the reduction has finished. The scope of this definition can be restricted to the expression in the body of the function.

```
val rec reduce = fn (g, e, m, n, f) =>
    let val finished  = fn true => e
                         | false => g (reduce (g, e, m, n−1, f), f n)
    in  finished (m > n)
    end
```

The Standard ML language has the property that if all occurrences in a program of value identifiers defined by nonrecursive definitions are replaced by their definitions then an equivalent program is obtained. We shall apply this expansion to the occurrence of the finished function used in the reduce function as a way of explaining the meaning of the *let* .. *in* .. *end* construct. The program which is produced after this expansion is performed is shown below.

```
val rec reduce = fn (g, e, m, n, f) =>
        (fn true => e
          | false => g (reduce(g, e, m, n−1, f), f n)) (m > n)
```

The two versions of reduce implement the same function but obviously the first version is much to be preferred; it is much clearer. If the finished function had been used more than once in the body of the reduce function the result after expansion would have been even less clear. Now we may redefine the sum′ and sq functions in terms of reduce.

```
val sum′ = fn n => reduce (fn (x, y) => x+y, 0, 1, n, fn x => x)
val  sq  = fn n => reduce (fn (x, y) => x+y, 0, 1, n, fn x => 2*x−1)
```

Note that the function `fn (x, y) => x+y` which is passed to the reduce function does nothing more than use the predefined infix addition operation to form an addition function. Standard ML provides a facility to convert infix operators to the corresponding prefix function using the *op* keyword. Thus the expressions `(op +)` and `fn (x, y) => x+y` denote the same function. We will use the *op* keyword in the definition of another function which uses reduce, the factorial function, fac. The factorial of n is simply the product of the numbers from 1 to n.

```
val fac = fn n => reduce (op *, 1, 1, n, fn x => x)
```

Notice that if it is parenthesized, "`(op * )`" must be written with a space between the star and the bracket to avoid confusion with the end-of-comment delimiter.

3.2 Curried functions

The other question which arises once we have discovered that functions may take functions as arguments is "Can functions return functions as results?" (Such functions are called curried functions after Haskell B. Curry.) Curried functions seem perfectly reasonable tools for the functional programmer to request and we can encode any curried function using just the subset of Standard ML already introduced, e.g. `fn x => fn y => x`.

This tempting ability to define curried functions might leave us with the difficulty of deciding if a new function we wish to write should be expressed in its curried form or take a tuple as an argument. Perhaps we might decide that every function should be written in its fully curried form but this decision has the unfortunate consequence that functions which return tuples as results are sidelined. However, the decision is not really so weighty since we may define functions to curry or uncurry a function after the fact. We will define these after some simple examples.

A simple example of a function which returns a function as a result is a function which, when given a function f, returns the function which applies f to an argument and then applies f to the result. Here is the Standard ML implementation of the twice function.

```
val twice = fn f => fn x => f (f x)
```

For idempotent functions, twice simply acts as the identity function. The integer successor function `fn x => x+1` can be used as an argument to twice.

```
val addtwo = twice (fn x => x+1)
```

The function twice is simply a special case of a more general function which applies its function argument a number of times. We will define the iter function for iteration. It is a curried function which returns a curried function as its result. The function will have the property that iter 2 is twice. Here is the task we wish the iter function to perform.

$$\texttt{iter n f x} \quad \equiv \quad \texttt{f}^{\texttt{n}}\texttt{(x)} \quad \equiv \quad \underbrace{\texttt{f(f(}\cdots\texttt{f}}_{\texttt{n times}}\texttt{(x)}\cdots\texttt{))}$$

In the simplest case we have $f^0 = \mathrm{id}$, the identity function. When n is positive we have that $f^n(x) = f(f^{n-1}(x))$. We may now implement the iter function in Standard ML.

```
val rec iter =
        fn 0 => (fn f => fn x => x)
         | n => (fn f => fn x => f(iter (n-1) f x))
```

As promised above, we now define two higher-order, curried Standard ML functions which, respectively, transform a function into its curried form and transform a curried function into tuple-form.

```
  val curry = fn f => fn x => fn y => f(x, y)
val uncurry = fn f => fn (x, y) => f x y
```

If x and y are values and f and g are functions then we always have:

```
      (curry f) x y ≡ f(x, y)
(uncurry g)(x, y) ≡ g x y
```

3.3 Function composition

Let us now investigate a simple and familiar method of building functions: composing two existing functions to obtain another. The function composition $f \circ g$ denotes the function with the property that $(f \circ g)(x) = f(g(x))$. This form of composition is known as composition in functional order. Another form of composition defines $g; f$ to be $f \circ g$. This is known as composition in diagrammatic order. This is found in mathematical notation but not in programming language notation. Standard ML has a semicolon operator but it does not behave as described above. In fact, for two functions g and f we have instead that g ; f ≡ f.

Notice that function composition is associative, that is: $f \circ (g \circ h) = (f \circ g) \circ h$. The identity function is both a left and a right identity for composition; $\mathrm{id} \circ f = f \circ \mathrm{id} = f$. Notice also the following simple correspondence between function iteration and function composition.

$$f^n = \underbrace{f \circ f \circ \cdots \circ f}_{f \text{ occurs } n \text{ times}}$$

Function composition in functional order is provided in Standard ML by the predefined operator "o"—the letter O in lower case. If f and g are Standard ML functions then f o g is their composition. However, we will define a compose function which is identical to (op o).

```
val compose = fn (f, g) => fn x => f(g x)
```

3.4 Derived forms

Standard ML takes pattern matching and binding names to values as essential primitive operations. It provides additional syntactic constructs to help to make function declarations compact and concise. This additional syntax does not add to the power of the language, it merely sweetens the look of the functions which are being defined. Syntactic constructs which are added to a programming language in order to make programs look neater or simpler are sometimes called syntactic sugar but Standard ML calls them derived forms. The use of this more dignified term can be justified because the language has a formal semantic definition and that terminology seems appropriate in that context.

The derived form notation for functions uses the keyword *fun*. After this keyword comes the function identifier juxtaposed with the patterns in their turn. For example, the integer successor function can be declared thus: `fun succ x = x + 1`. The *fun* keyword applies also to recursive functions so we might re-implement the sum function from the previous chapter as shown here.

```
fun sum 1 = 1
  | sum n = sum (n - 1) + n
```

We begin to see that derived forms are needed when we consider curried functions with several arguments. The definitions of curry, uncurry and compose are much more compact when written as shown below.

```
fun curry f x y = f (x, y)
fun uncurry f (x, y) = f x y
fun compose (f, g) x = f (g x)
```

Other notation in the language is defined in terms of uses of functions. The most evident is the *case* .. *of* form which together with the *fun* keyword can be used to clarify the implementation of the reduce function to a significant extent. Compare the function declaration below with the previous version in order to understand how the *case* construct is defined as a derived form.

```
fun reduce (g, e, m, n, f) =
          case m > n of true => e
                      | false => g (reduce(g, e, m, n−1, f), f n)
```

The division of function definitions on the truth or falsehood of a logical condition occurs so frequently in the construction of computer programs that most programming languages provide a special form of the case statement for the type of truth values. Standard ML also provides this. Again, compare the function declaration below with the previous version in order to understand how the conditional expression is defined as a derived form.

```
fun reduce (g, e, m, n, f) =
          if m > n then e else g (reduce(g, e, m, n−1, f), f n)
```

Other keywords are derived forms which obtain their meanings from expressions which use a conditional. The logical connectives *andalso* and *orelse* are the short-circuiting versions of conjunction and disjunction. This means that they only evaluate the expression on the right-hand side if the expression on the left-hand side does not determine the overall result of the expression. That is just the behaviour which would be expected from a conditional expression and hence that is why the definition works.

Note in particular that *andalso* and *orelse* are not infix functions because they are not strict in their second argument—that is, they do not always force the evaluation of their second argument—and such functions cannot be defined in a strict programming language such as Standard ML. Thus we cannot apply the *op* keyword to *andalso* or *orelse*.

Exercise 3.4.1 *By using only constructors in pattern matching we could write four-line functions for the binary logical connectives which simply mimicked their truth tables. By allowing variables in patterns the declarations could all be shorter. Write these shorter versions of* conj, disj, impl *and* equiv*.*

Exercise 3.4.2 *Would it be straightforward to rewrite the* reduce *function from page 14 using* local *rather than* let*? What would be the difficulty?*

Exercise 3.4.3 *The semifactorial of a positive integer is $1 \times 3 \times 5 \times \cdots \times n$ if n is odd and $2 \times 4 \times 6 \times \cdots \times n$ if n is even. Use the* reduce *function to define a* semifac *function which calculates semifactorials.*

Exercise 3.4.4 *Which, if either, of the following are well defined?*

(1) compose (compose, uncurry compose)
(2) compose (uncurry compose, compose)

Exercise 3.4.5 *Use the predefined Standard ML version of function composition to define a function,* iter′*, which behaves identically to the function* iter *given earlier.*

Exercise 3.4.6 *How would you define exp_1* `andalso` *exp_2 and exp_1* `orelse` *exp_2 in terms of exp_1, exp_2, conditional expressions and the constants* `true` *and* `false`*? Try out your definitions against the expressions with the derived forms when exp_1 is "*`true`*" and exp_2 is "*`10 div 0 = 0`*". Then change exp_1 to "*`false`*" and compare again.*

Chapter 4

Types and type inference

Standard ML is a strongly and statically typed programming language. However, unlike many other strongly typed languages, the types of literals, values, expressions and functions in a program will be calculated by the Standard ML system when the program is compiled. This calculation of types is called type inference. Type inference helps program texts to be both lucid and succinct but it achieves much more than that because it also serves as a debugging aid which can assist the programmer in finding errors before the program has ever been executed.

Standard ML's type system allows the use of typed data in programs to be checked when the program is compiled. This is in contrast to the approach taken in many other programming languages which generate checks to be tested when the program is running. Lisp is an example of such a language. Other languages serve the software developer even less well than this since they neither guarantee to enforce type-correctness when the program is compiled nor when it is running. The C programming language is an example of a language in that class. The result of not enforcing type correctness is that data can become corrupted and the unsafe use of pointers can cause obscure errors. A splendid introduction to this topic is [Car96].

The approach of checking type correctness as early as possible has two clear advantages: no extra instructions are generated to check the types of data during program execution; and there are no insecurities when the program is executed. Standard ML programs can be executed both efficiently and safely and will never 'dump core' no matter how inexperienced the author of the program might have been. The design of the language ensures that this can never happen. (Of course, any particular compiler might be erroneous: compilers are large and complex programs. Such errors should be seen to be particular to one of the implementations of the language and not general flaws in design of the language.)

4.1 Type inference

Standard ML supports a form of polymorphism. Before going further, we should clarify the precise nature of the polymorphism which is permitted. It is sometimes referred to as "let-polymorphism". This name derives from the fact that in this system the term

```
let val Id = fn x => x in (Id 3, Id true) end
```

is a well-typed term whereas the very similar

```
(fn Id => (Id 3, Id true)) (fn x => x)
```

is not. The *let* .. *in* .. *end* construct is not just syntactic sugar for function application, it is essential to provide the polymorphism without compromising the type security of the language. This polymorphic type system has a long history; the early work was done by Roger Hindley [Hin69] but his work did not become well-known nor was its importance realised until the type system was re-discovered and extended by Robin Milner [Mil78].

We can distinguish between two kinds of bound variables: those which are bound by the keyword *fn* and those which are bound by the keyword *let*. The distinction is this:

- all occurrences of a *fn*-bound identifier must have the same type; but
- each occurrence of a *let*-bound identifier may have a different type provided it is a instance of the principal—or most general—type inferred for that identifier.

4.2 Pairs and record types

We have used pairs and tuples without stating their type. A pair with an integer as the left element and a boolean as the right has type "int * bool". Note that int * bool * real is neither (int * bool) * real nor int * (bool * real). Pairs and tuples are themselves simply records with numbered fields. The label for a field can also be a name such as age. Each field has an associated projection function which retrieves the corresponding value. The projection function for the age field is called #age. The following record value has type { initial : char, surname : string, age : int }.

```
val lecturer = { initial = #"S", surname = "Gilmore", age = 36 }
```

Then #surname (lecturer) is "Gilmore", as expected.

This has shown us another of the derived forms of the language. The pair and tuple notation is a derived form for the use of record notation. Thus the meaning of the second of the declarations below is the first.

```
val n : { 1 : int, 2 : bool } = { 1 = 13, 2 = false }
val n : int * bool = (13, false)
```

4.3 Function types and type abbreviations

On rare occasions, we need to tell Standard ML the type of a function parameter which it cannot itself infer. If we have to do that then it is convenient to be able to give a name to the type, rather than including the expression for the type in a constraint on a parameter. One time when we need to specify a type is when we write a function which projects information from a record. The following function is not an acceptable Standard ML function.

```
fun initials p = (#initial p, String.sub (#surname p, 0))          (‡)
```

The problem is that the type of the parameter p is underdetermined. We can see that it must be a record type with fields for initial letter and surname but what other fields does it have? Does it have age? Does it have date_of_birth? We cannot tell from the function definition and Standard ML does not support a notion of subtyping. No relation holds between tuples which are not identical: int * real * bool and int * real are not related. This has the consequence that it is impossible to define a function such as the function above without making explicit the type of the parameter, which we now do with the help of a type abbreviation.

```
type person = { initial : char, surname : string, age : int }

fun initials (p : person) = (#initial p, String.sub (#surname p, 0))
```

Type abbreviations are purely cosmetic. The type name person simply serves as a convenient abbreviation for the record type expression involving initial, surname and age.

As another example of the use of a type abbreviation, consider the possibility of representing sets by functions from the type of the elements of the set to the booleans. These functions have the obvious behaviour that the function returns true when applied to an element of the set and false otherwise. If we are using functions in this way it would be reasonable to expect to be able to state the fact that these functions represent sets. The complication here is that a family of type abbreviations are being defined; integer sets, real sets, word sets, boolean sets and others. One or more type variables may be used to parameterise a type abbreviation, as shown below.

$$\textit{type}\ \alpha\ \text{set} = \alpha \to \text{bool}$$

This would be the way to enter this declaration into Standard ML except that stupidly someone left many mathematical symbols and all of the Greek letters out of the ASCII character set so α is actually entered as a primed identifier, `'a`, and $\to$ is actually entered as the "->" keyword. Type variables such as `'a`, `'b`, `'c`, are pronounced 'alpha', 'beta', 'gamma'.

4.4 Defining datatypes

The *type* mechanism cannot be used to produce a fresh type: only to re-name an existing type. A Standard ML programmer can introduce a new type, distinct from all the others, through the use of datatypes.

datatype colour = red | blue | green

This introduces a new type, `colour`, and three *constructors* for that type, `red`, `blue` and `green`. Equality is defined for this type with the expected behaviour that, for example, `red = red` and `red <> green`. No significance is attached to the order in which the constructors were listed in the type definition and no ordering is defined for the type. Constructors differ from values because constructors may be used to form the patterns which appear in the definition of a function by pattern matching, as in the following function: (*fn* red => 1 | blue => 2 | green => 3). The pre-defined type `bool` behaves as if defined thus.

datatype bool = true | false

In Standard ML it is illegal to rebind the constructors of built-in datatypes such as `bool`. The motivation for this is to prevent confusion about the interaction between the derived-forms translation and runt datatypes such as this—*datatype* bool = true—intended to replace the built-in booleans. Thus the constructors of built-in datatypes have an importance which places them somewhere between the constructors of programmer-defined datatypes and reserved.

In constrast to the reverence accorded to the built-in constructors, programmer-defined constructors can be re-defined and these new definitions hide the ones which can before. So imagine that after elaborating the definition of the `colour` datatype we elaborate this definition.

datatype traffic_light = red | green | amber

Now we have available four constructors of two different types.

amber: traffic_light	blue: colour
green: traffic_light	red: traffic_light

The name `blue` is still in scope but the two other names of colours are not.

Another distinctive difference between datatype definitions and type abbreviations is that the type abbreviation mechanism cannot be used to describe recursive data structures; the type name is not in scope on the right-hand side of the definition. This is the real reason why we need another keyword, "*datatype*", to mark our type definitions as being [potentially] recursive just as we needed a new keyword, *rec*, to mark our function definitions as being recursive. One recursive datatype we might wish to define is the datatype of binary trees. If we wished to store integers in the tree we could use the following definition.

datatype inttree = empty | node *of* int * inttree * inttree

Note the use of yet another keyword, "*of*". The declaration introduces the empty binary tree and a constructor function which, when given an integer `n` and two integer trees, t_1 and t_2, builds a tree with `n` at the root and with t_1 and t_2 as left and right sub-trees. This tree is simply node(n, t_1, t_2). The reason for the use of the term "constructor" now becomes clearer, larger trees are really being constructed from smaller ones by the use of these functions. The constructors of the `colour` datatype are a degenerate form of constructors since they are nullary constructors.

The mechanism for destructing a constructed value into its component parts is to match it against a pattern which uses the constructor and, in so doing, bind the value identifiers which occur in the pattern. This convenient facility removes the need to implement 'destructors' for every new type and thereby reduces the amount of code which must be produced, enabling more effort to be expended on the more taxing parts of software development.

Since Standard ML type abbreviations may define families of types, it would seem natural that the datatypes of the language should be able to define families of datatypes. A datatype definition with a type parameter may be used to build objects of different types. The following `tree` definition generalises the integer trees given above.

```
datatype α tree = empty | node of α * α tree * α tree
```

We see that the `node` constructor is of type `(α * (α tree) * (α tree)) → (α tree)`. There is a peculiar consequence of allowing datatypes to be defined in this way since we might make the type increase every time it is passed back to the type constructor thus making a so-called "stuttering" datatype.

```
datatype α ttree = empty | node of α * (α * α) ttree * (α * α) ttree
```

Standard ML functions cannot be used within their own definitions on values of different types so there is no way to write a recursive Standard ML function which can process these trees, say to count the number of values stored in the tree or even to calculate its depth. We could comment that allowing the parameter in a datatype definition to be inflated in this way when it is passed back has created a slight imbalance in the language because it is possible to define recursive datatypes which cannot be processed recursively. This is not anything more serious than an imbalance; it is not a serious flaw in the language.

A built-in parameterised datatype of the language is the type of lists. These are ordered collections of elements of the same type. The pre-defined Standard ML type constructor `list` is a parameterised datatype for representing lists. The parameter is the type of elements which will appear in the list. Thus, `int list` describes a list of integers, `char list` describes a list of characters and so on.

The list which contains no elements is called `nil` and if `h` is of type `α` and `t` is of type `α list` then `h :: t`—pronounced "h cons t"—is also of type `α list` and represents the list with first element `h` and following elements the elements of `t` in the order that they appear in `t`. Thus `1 :: nil` is a one-element integer list; `2 :: 1 :: nil` is a two-element integer list and so on. Evidently to be correctly typed an expression with multiple uses of cons associates to the right. Thus the datatype definition for `α list` is as shown below. It declares the cons symbol to be used infix with right associativity and priority five. The keywords *infix* and *infixr* specify left and right associativity respectively.

```
infixr 5 ::
datatype α list = nil | :: of α * α list
```

Lists come with derived forms. The notation `[2, 1]` is the derived form for `2 :: 1 :: nil`; and similarly. For consistency, `[ ]` is the derived form for `nil`. As with the `bool` datatype we cannot redefine `::` or `nil` although, rather curiously, we can tinker with their fixity status and associativity—perhaps a small oversight by the language designers.

All of the parameterised datatypes which we have declared so far have been parameterised by a single type variable but they can be parameterised by a tuple of type variables. We can define lookup tables to be lists of pairs as shown below.

```
type (α, β) lookup = (α * β) list
```

Exercise 4.4.1 *This amusing puzzle is due to Bruce Duba of Rice University. At first it does not seem that it is possible at all. [Hint: you will need a tuple of type variables.]*

Define the constructors, `Nil` *and* `Cons`*, such that the following code type checks.*

```
fun length (Nil)          = 0
  | length (Cons (_, x)) = 1 + length (x)
val heterogeneous = Cons (1, Cons (true, Cons (fn x => x, Nil)))
```

Exercise 4.4.2 *It is possible to introduce two values at once by introducing a pair with the values as the elements, e.g.* `val (x, y) = (6, 7)` *defines* `x` *to be six and* `y` *to be seven. Why is it not possible to get around the need to use the keyword* `and` *by defining functions in pairs as shown below?*

```
val (odd, even) = (fn 0 => false | n => even (n - 1) ,
                   fn 0 => true  | n => odd (n - 1))
```

Datatype definitions can also be mutually recursive. An example of an application where this arises is in defining a programming language with integer expressions where operations such as addition, subtraction, multiplication and division can be used together with parentheses. A Standard ML datatype for integer expressions is shown here.

```
datatype int_exp    = plus of int_term * int_term
                    | minus of int_term * int_term
     and int_term   = times of int_factor * int_factor
                    | divide of int_factor * int_factor
                    | modulo of int_factor * int_factor
     and int_factor = int_const of int
                    | paren of int_exp
```

Exercise 4.4.3 *Define the following functions.*

```
eval_int_exp: int_exp → int
eval_int_term: int_term → int
eval_int_factor: int_factor → int
```

4.5 Polymorphism

Many functions which we have defined do not need to know the type of their arguments in order to produce meaningful results. Such functions are thought of as having many forms and are thus said to be polymorphic. Perhaps the simplest example of a polymorphic function is the identity function, `id`, defined by `fun id x = x`. Whatever the type of the argument to this function, the result will obviously be of the same type; it is a homogeneous function. All that remains is to assign it a homogeneous function type such as $X \to X$. But what if the type X had previously been defined by the programmer? The clash of names would be at best quite confusing. We shall give the `id` function the type $\alpha \to \alpha$ and prohibit the programmer from defining types called α, β, γ and so on.

Exercise 4.5.1 *Define a different function with type* $\alpha \to \alpha$.

The pairing function, `pair`, is defined by `fun pair x = (x, x)`. This function returns a type which is different from the type of its argument but still does not need to know whether the type of the argument is `int` or `bool` or a record or a function. The type is of course $\alpha \to (\alpha * \alpha)$. Given a pair it may be useful to project out either the left-hand or the right-hand element. We can define the functions `fst` and `snd` for this purpose thus: `fun fst (x, _) = x` and `fun snd (_, y) = y`. The functions have types $(\alpha * \beta) \to \alpha$ and $(\alpha * \beta) \to \beta$ respectively.

Exercise 4.5.2 *The function* `fn x => fn y => x` *has type* $\alpha \to (\beta \to \alpha)$. *Without giving an explicit type constraint, define a function with type* $\alpha \to (\alpha \to \alpha)$.

Notice that parentheses cannot be ignored when computing types. The function `paren` below has type $\alpha \to ((\alpha \to \beta) \to \beta)$ whereas the function `paren'` has type $\alpha \to \alpha$.

```
fun paren n  = fn g => g n
fun paren' n = (fn g => g) n
```

Exercise 4.5.3 *What is the type of* `fn x => x (fn x => x)`*?*

Standard ML will compute the type $\alpha \to \beta$ for the following function.

```
fun loop x = loop x
```

The type $\alpha \to \beta$ is the most general type for any polymorphic function. In contrasting this with $\alpha \to \alpha$, the type of the polymorphic identity function, it is simple to realise that nothing could be determined about the result type of the function. This is because no application of this function will ever return a result.

In assigning this type to the function, the Standard ML type system is indicating that the execution of the `loop` function will not terminate since there are no interesting functions of type $\alpha \to \beta$. Detecting (some) non-terminating functions is an extremely useful service for a programming language to provide.

Of course, the fact that an uncommon type has been inferred will only be a useful error detection tool if the type which was expected is known beforehand. For this reason, it is usually very good practice to compute by hand the type of the function which was written then allow Standard ML to compute the type and then compare the two types for any discrepancy. Some authors (e.g. Myers, Clack and Poon in [MCP93]) recommend embedding the type information in the program once it has been calculated, either by hand or by the Standard ML system, but this policy means that the program text can become rather cluttered with type information which obscures the intent of the program. However, in some implementations of the language the policy of providing the types for the compiler to check rather than requiring the compiler to infer the types may shorten the compilation time of the program. This might be a worthwhile saving for large programs.

4.5.1 Function composition

For the Standard ML function composition operator to have a well-defined type it is necessary for the source of the first function to be identical to the target of the second. For both functions, the other part of the type is not constrained. Recall the definition of the `compose` function which is equivalent to `(op o)`.

```
val compose = fn (f, g) => fn x => f(g(x))
```

We can calculate the type of `compose` as follows. It is a function which takes a pair so we may say that it is of the form $(\bigcirc * \bigcirc) \to \bigcirc$. We do not wish to restrict the argument `f` and thus we assign it a type $\alpha \to \beta$ since this is the worst possible type it can have. This forces `g` to be of the form $\bigcirc \to \alpha$ and we have $((\alpha \to \beta) * (\bigcirc \to \alpha)) \to \bigcirc$ as our current type for `compose`. Of course, there is no reason to restrict the type of `g` either so we assign it the type $\gamma \to \alpha$ and thus calculate $((\alpha \to \beta) * (\gamma \to \alpha)) \to (\gamma \to \beta)$ as the type of the `compose` function.

Exercise 4.5.4 *Define a function with type* $((\alpha \to \alpha) * (\alpha \to \alpha)) \to (\alpha \to \alpha)$ *without using a type constraint.*

Exercise 4.5.5 *What is the type of* `curry`*? What is the type of* `uncurry`*?*

4.5.2 Default overloading

In Standard ML programs, types are almost always inferred and there are only a few cases where additional information must be supplied by the programmer in order for the system to be able to compute the type of an expression or a value in a declaration. These cases arise because of underdetermined record types—as we saw with the version of the `initials` function which is marked (‡) on page 20. Another complication is overloading.

Overloading occurs when an identifier has more than one definition and these definitions have different types. For example, "+" and "–" are overloaded since they are defined for the numeric types: `int`, `word` and `real`. The "`~`" function is overloaded for `int` and `real`. The relational operators are overloaded for the numeric types and the text types, `char` and `string`. Overloading is not polymorphism: there is no way for the Standard ML programmer to define overloaded operators. To see this, consider the following simple `square` function.

```
fun square x = x * x
```

Would it be possible to assign the type $\alpha \rightarrow \alpha$ to this function? No, we should not because then `square` could be applied to strings, or even functions, for which no notion of multiplication is defined. So, Standard ML must choose one of the following possible types for the function.

```
square: int → int
square: word → word
square: real → real
```

Without being too pedantic, a good rule of thumb is that default overloading will choose numbers in favour of text and integers in favour of words or reals. Thus the type of `square` is `int → int`. We can force a different choice by placing a type constraint on the parameter to the function.

The type of the function `ordered` shown below is `(int * int) → bool` where here there were five possible types.

```
fun ordered (x, y) = x < y
```

Default overloading is not restricted to functions, it also applies to constants of non-functional type. Thus in an implementation of the language which provides arbitrary precision integers we might write `(100:BigInt.int)` in order to obtain the right type for a constant.

Our conclusion then is that overloading has a second-class status in Standard ML. Other programming languages, notably Ada [Bar96], provide widespread support for overloading but do not provide type inference. The Haskell language provides both.

4.6 Ill-typed functions

The Standard ML type discipline will reject certain attempts at function definitions. Sometimes these are obviously meaningless but there are complications. Mads Tofte writes in [Tof88]:

> At first it seems a wonderful idea that a type checker can find programming mistakes even before the program is executed. The catch is, of course, that the typing rules have to be simple enough that we humans can understand them and make the computers enforce them. Hence we will always be able to come up with examples of programs that are perfectly sensible and yet illegal according to the typing rules. Some will be quick to say that far from having been offered a type discipline they have been lumbered with a type bureaucracy.

It is Mads Tofte's view that rejecting some sensible programs which would never go wrong is inevitable but not everyone is so willing to accept a loss such as this. Stefan Kahrs in [Kah96] discusses the notion of completeness—programs which never go wrong can be type-checked—which complements Milner's notion of soundness—type-checked programs never go wrong [Mil78].

We will now consider some programs which the type discipline of Standard ML will reject. We have already noted above that the function `(fn g => (g 3, g true))` is not legal. Other pathological functions also cannot be defined in Standard ML. Consider the "`absorb`" function.

```
fun absorb x = absorb
```

This function is attempting to return itself *as its own result.* The underlying idea is that the `absorb` function will greedily gobble up any arguments which are supplied. The arguments may be of any type and there may be any number of them. Consider the following evaluation of an application of `absorb`.

```
absorb true 1 "abc" ≡ (((absorb true) 1) "abc")
                    ≡ ((absorb 1) "abc")
                    ≡ (absorb "abc")
                    ≡ absorb
```

Such horrifying functions have no place in a reasonable programming language. The Standard ML type system prevents us from defining them.

The `absorb` function cannot be given a type, because there is no type which we could give to it. However, `absorb` has a near-relative—`create`, shown below—which could be given type $\alpha \to \beta$ in some type systems, but will be rejected by Standard ML.

```
fun create x = create x x
```

As with `absorb`, there seems to be no practical use to which we could put this function. Once again consider an application.

```
create 6 ≡ (create 6) 6
         ≡ ((create 6) 6) 6
         ≡ (((create 6) 6) 6) 6
         ≡ ...
```

4.7 Computing types

Perhaps we might appear to have made too much of the problem of computing types. It may seem to be just a routine task which can be quickly performed by the Standard ML system. In fact this is not true. Type inference is computationally hard [KTU94] and there can be no algorithm which guarantees to find the type of a value in a time proportional to its "size". Types can increase exponentially quickly and their representations soon become textually much longer than an expression which has a value of that type. Fortunately the worst cases do not occur in useable programs. Fritz Henglein states in [Hen93],

> ... in practice, programs have "small types", if they are well typed at all, and Milner-Mycroft type inference for small types is tractable. This, we think, also provides insight into why ML type checking is usable and used in practice despite its theoretical intractability.

Exercise 4.7.1 *Compute the type of $y \circ y$ if y is $x \circ x$ and x is $pair \circ pair$.*

Exercise 4.7.2 *[This exercise is due to Stuart Anderson.] Work out by hand the type of the function* `fun app g f = f(f g)`. *[It may be helpful also to see this without the use of derived forms as* `val app = fn g => fn f => f(f g)`*]. What is the type of* `app app` *and what is the type of* `app (app app)`*?*

Exercise 4.7.3 *Why can the following function* $\mathtt{app}_2$ *not be typed by the Hindley-Milner type system? [The formal parameter* `f` *is intended to be a curried function.]*

$$\mathit{fun}\ \mathtt{app}_2\ \mathtt{f}\ \mathtt{x}_1\ \mathtt{x}_2 = (\mathtt{f}\ \mathtt{x}_1)\ (\mathtt{f}\ \mathtt{x}_2)$$

Chapter 5

Aggregates

Through the use of the datatype mechanism of Standard ML we can equip our programs with strong and relevant structure which mirrors the natural structure in the data values which they handle. When writing integer processing functions we were pleased that natural number induction was available to us to help us to check the correctness of our recursive function definitions and now the use of datatypes such as lists and trees might seem to make reasoning about a Standard ML function more difficult. We would be unable to use the induction principle for the natural numbers without re-formulating the function to be an equivalent function which operated on integers. The amount of work involved in the re-formulation would be excessive and it is preferable to have an induction principle which operates on datatypes directly. This is the basis of structural induction. This chapter introduces some functions which process lists, trees and vectors and shows how to use structural induction to check properties of these functions.

5.1 Lists

This pleasant datatype is to be found in almost all functional programming languages. In untyped languages lists are simply collections but in typed languages they are collections of values of the same type and so a list is always a list *of something*. Properly speaking, in Standard ML `list` is not a type, it is a type constructor. When we choose a particular type for the variable used by the type constructor then we have a type; so `char list` is a type and `int list` is a type and so forth. As we saw when we considered the definition (page 22) a list can be built up by using two value constructors, one for empty lists [`nil` of type α `list`] and one for non-empty lists [`::` of type $\alpha * \alpha$ `list` $\rightarrow \alpha$ `list`]. Some languages also provide destructors often called head and tail (`car` and `cdr` in LISP.) The definition of Standard ML does not insist that these destructors should be available since they are not needed; a list value may be decomposed by matching it against a pattern which uses the constructor. If we wished to use these destructors, how could we implement them? A difficulty which we would encounter in any typed language is that these functions are undefined for empty lists and so they must fail in these cases. Standard ML provides exceptions as a mechanism to signal failure. Raising an exception is a different activity from returning a value. For the purposes of type-checking it is similar to non-termination because no value is returned when an exception is raised.

An exception is introduced using the *exception* keyword. Here are declarations for three exceptions, `Empty`, `Overflow` and `Subscript`.

```
exception Empty
exception Overflow
exception Subscript
```

These declarations provide us with three new constructors for the built-in `exn` type. Like constructors of a datatype `Empty`, `Overflow` and `Subscript` may be used in patterns to denote themselves.

Again like constructors of a datatype they may be handled as values—passed to functions, returned as results, stored in lists and so forth. Exception constructors differ from datatype constructors in that they may be raised to signal that an exceptional case has been encountered and raised exceptions may subsequently be handled in order to recover from the effect of encountering an exceptional case. Thus exceptions are not fatal errors, they are merely transfers of program control. Now to return to implementing list destructors.

Definition 5.1.1 (Head) *An empty list has no head. This is an exceptional case. The head of a non-empty list is the first element. This function has type* α list → α.

```
fun hd []       = raise Empty
  | hd (h :: t) = h
```

Definition 5.1.2 (Last) *An empty list has no last element. This is an exceptional case. The last element of a one-element list is the first element. The last element of a longer list is the last element of its tail. This function has type* α list → α.

```
fun last []       = raise Empty
  | last [x]      = x
  | last (h :: t) = last t
```

Definition 5.1.3 (Tail) *An empty list has no tail. This is an exceptional case. The tail of a non-empty list is that part of the list following the first element. This function has type* α list → α list.

```
fun tl []       = raise Empty
  | tl (h :: t) = t
```

Definition 5.1.4 (Testers) *We might instead choose to use versions of head, last and tail functions which are of type* α list → α option *and* α list → α list option. *The option datatype is defined by* datatype α option = NONE | SOME *of* α. *The definitions of these 'tester' functions follow.*

```
fun hd_tst []       = NONE
  | hd_tst (h :: t) = SOME h

fun last_tst []       = NONE
  | last_tst [x]      = SOME x
  | last_tst (h :: t) = last_tst t

fun tl_tst []       = NONE
  | tl_tst (h :: t) = SOME t
```

These functions never raise exceptions and might be used in preference to the exception-producing versions given above. The conversion from one set to the other is so systematic that we can write a general purpose function to perform the conversion from an exception-producing function to one with an optional result. The tester *function shown below achieves this effect. Any exception which is raised by the application of* f *to* x *is handled and the value* NONE *is returned.*

```
fun tester f x = SOME (f x) handle _ => NONE
```

Thus hd_tst *is equivalent to* tester hd, last_tst *is equivalent to* tester last *and* tl_tst *is equivalent to* tester tl.

Definition 5.1.5 (Length) *The* length *function for lists has type* α list → int. *The empty list has length zero; a list with a head and a tail is one element longer than its tail.*

```
fun length []       = 0
  | length (h :: t) = 1 + length t
```

Definition 5.1.6 (Append) *The append function has type* (α list * α list) → α list. *In fact this is a pre-defined right associative operator,* @, *in Standard ML. If* l_1 *and* l_2 *are two lists of the same type then* l_1 @ l_2 *is a list which contains all the elements of both lists in the order they occur. This append operator has the same precedence as cons.*

```
infixr 5 @

fun [] @ l2        = l2
  | (h :: t) @ l2  = h :: t @ l2
```

Exercise 5.1.1 *Consider the situation where we had initially mistakenly set the precedence of the append symbol to be four, and corrected this immediately afterwards.*

```
infixr 4 @

fun [] @ l2        = l2
  | (h :: t) @ l2  = h :: t @ l2

infixr 5 @
```

Could this mistake be detected subsequently? If so, how?

Definition 5.1.7 (Reverse) *Using the append function we can easily define the function which reverses lists. This function has type* α list → α list. *The* rev *function is pre-defined but we will give a definition here which is identical to the pre-defined function. The base case for the recursion will be the empty list which reverses to itself. Given a list with head* h *and tail* t *then we need only reverse* t *and append the single-element list* [h] *(equivalently,* h :: nil*).*

```
fun rev []        = []
  | rev (h :: t)  = (rev t) @ [h]
```

Definition 5.1.8 (Reverse append) *One some occasions, the order in which elements appear in a list is not very important and we do not care about having the order of the inputs preserved in the results (as the append function does). The* revAppend *function joins lists by reversing the first onto the front of the second.*

```
fun revAppend ([], l2)       = l2
  | revAppend (h :: t, l2)   = revAppend(t, h :: l2)
```

Exercise 5.1.2 *Provide a definition of a reverse function by using reverse appending.*

5.2 Induction for lists

We will now introduce an induction principle for lists. It is derived directly from the definition of the list datatype.

Induction Principle 5.2.1 (Lists)

$$\frac{P\,[\,] \qquad P\,(t) \Rightarrow P\,(h :: t)}{\forall l.\, P\,(l)}$$

Proposition 5.2.1 (Interchange) *The* rev *function and the append operator obey an interchange law since the following holds for all* α *lists* l_1 *and* l_2.

$$\texttt{rev}\ (l_1\ @\ l_2) = \texttt{rev}\, l_2\ @\ \texttt{rev}\, l_1$$

Proof: The proof is by induction on l_1. The initial step is to show that this proposition holds when l_1 is the empty list, `[]`. Using properties of the append operator, we conclude $\texttt{rev}\,(\texttt{[]} \mathbin{@} l_2) = \texttt{rev}\, l_2 = \texttt{rev}\, l_2 \mathbin{@} \texttt{[]} = \texttt{rev}\, l_2 \mathbin{@} \texttt{rev}\,\texttt{[]}$ as required.

Now assume $\texttt{rev}\ (\texttt{t} \mathbin{@} l_2) = \texttt{rev}\, l_2 \mathbin{@} \texttt{rev}\,\texttt{t}$ and consider $\texttt{h} :: \texttt{t}$.

$$\begin{array}{rll} \text{LHS} & = \texttt{rev}\,((\texttt{h} :: \texttt{t}) \mathbin{@} l_2) & \\ & = \texttt{rev}\,(\texttt{h} :: (\texttt{t} \mathbin{@} l_2)) & [\text{defn of } @] \\ & = (\texttt{rev}\,(\texttt{t} \mathbin{@} l_2)) \mathbin{@} [\,\texttt{h}\,] & [\text{defn of } \textit{rev}] \\ & = \texttt{rev}\, l_2 \mathbin{@} \texttt{rev}\,\texttt{t} \mathbin{@} [\,\texttt{h}\,] & [\text{induction hypothesis}] \\ & = \texttt{rev}\, l_2 \mathbin{@} \texttt{rev}\,(\texttt{h} :: \texttt{t}) & [\text{defn of } \textit{rev} \text{ and } @] \\ & = \text{RHS} & \end{array}$$

□

Exercise 5.2.1 *Prove by structural induction that for all* α *lists* l_1 *and* l_2

$$\texttt{length}\ (l_1 \mathbin{@} l_2) = \texttt{length}\ l_1 + \texttt{length}\ l_2.$$

Proposition 5.2.2 (Involution) *The* `rev` *function is an involution, i.e. it always undoes its own work, since* $\texttt{rev}\,(\texttt{rev}\, l) = l$.

Proof: The initial step is to show that this proposition holds for the empty list, `[]`. From the definition of the function, $\texttt{rev}\,(\texttt{rev}\,\texttt{[]}) = \texttt{rev}\,\texttt{[]} = \texttt{[]}$ as required.

Now assume that $\texttt{rev}\,(\texttt{rev}\,\texttt{t}) = \texttt{t}$ and consider $\texttt{h} :: \texttt{t}$.

$$\begin{array}{rll} \text{LHS} & = \texttt{rev}\,(\texttt{rev}\,(\texttt{h} :: \texttt{t})) & \\ & = \texttt{rev}\,((\texttt{rev}\,\texttt{t}) \mathbin{@} [\,\texttt{h}\,]) & [\text{defn of } \textit{rev}] \\ & = (\texttt{rev}\,[\,\texttt{h}\,]) \mathbin{@} (\texttt{rev}\,(\texttt{rev}\,\texttt{t})) & [\text{interchange law}] \\ & = [\,\texttt{h}\,] \mathbin{@} \texttt{t} & [\text{induction hypothesis and defn of } \textit{rev}] \\ & = \texttt{h} :: \texttt{t} & [\text{defn of } @] \\ & = \text{RHS} & \end{array}$$

□

5.3 List processing

In this section we will look at a collection of simple list processing metaphors. Most of the functions defined are polymorphic. A simple function which we might define initially is a membership test. The empty list has no members. A non-empty list has `x` as a member if `x` is the head or it is a member of the tail. The following `member` function tests for membership in a given list.

```
fun member (x, [])      = false
  | member (x, h :: t)  = x = h orelse member (x, t)
```

We might like this function to have type $\alpha * \alpha$ `list` → `bool` but it does not. This cannot be a fully polymorphic function since we make an assumption about the values and lists to which it can be applied: we assume that equality is defined upon them. Our experience of the Standard ML language so far would lead us to conclude that this matter would be settled by the default overloading rule which would assign to this function the type `int * int list → bool`. This reasoning, although plausible, is flawed.

The equality operator has a distinguished status in Standard ML. It is not an overloaded operator, it is a qualified polymorphic function. The reason that we make this distinction is that where possible equality is made available on new types which we define. This does not happen with overloaded operators because overloaded functions are those which select a different algorithm to apply depending on the type of values which they are given and it is not possible for the Standard ML language to 'guess' how we wish to have overloaded operators extended to our new types.

The Standard ML terminology is that a type either *admits* equality or it does not. Those which do are *equality types*. When equality type variables are printed by the Standard ML system they are printed with two leading primes and so the type of the member function is displayed as ''a * ''a list → bool. Types which do not admit equality in Standard ML include function types and structured types which contain function types, such as pairs of functions or lists of functions. The consequence is that a function application such as member (Math.sin, [Math.cos, Math.atan, Math.tan]) will be rejected as being incorrectly typed. Exceptions do not admit equality either so a function application such as member (Empty, [Overflow]) will also be rejected as being incorrectly typed.

Exceptions are defined not to admit equality but why should function types not admit equality? The answer is that the natural meaning of equality for functions is extensional equality; simply that when two equal functions are given equal values then they return equal results. It is an elevated view. Extensional equality does not look inside the functions to see how they work out their answers and neither does it time them to see how long they take. A programming language cannot implement this form of equality. The type of equality which it could implement is pointer equality (also called intensional equality) and that is not the kind which we want.

Equality types can arise in one slightly unexpected place, when testing if a list is empty. A definition which uses pattern matching will assign to null the fully polymorphic type α list → bool.

```
fun null [] = true
  | null _  = false
```

However, if instead we use the equality on a value of a polymorphic datatype, the type system of Standard ML will assume that an equality exists for the elements also. Any parametric datatype α t will admit equality only if α does. Thus a definition which uses equality will assign to null_eq the qualified polymorphic type ''a list → bool.

```
fun null_eq s = s = []
```

Searching for an element by a key will allow us to retrieve a function from a list of functions. The retrieve function has type ''a * (''a * β) list → β.

```
exception Retrieve
fun retrieve (k1, [])             = raise Retrieve
  | retrieve (k1, (k2, v2) :: t) = if k1 = k2 then v2 else retrieve (k1, t)
```

5.3.1 Selecting from a list

It is useful to have functions which select elements from a list, perhaps selecting the nth element numbered from zero. This function will have type α list * int → α and should raise the exception Subscript whenever the nth element cannot be found (either because there are fewer than n elements or because n is negative).

```
fun nth ([], _)     = raise Subscript
  | nth (h :: t, 0) = h
  | nth (_ :: t, n) = nth (t, n - 1) handle Overflow => raise Subscript
```

Note the excruciatingly complicated final case. We could program the test for a negative index explicitly with a conditional expression but this would cost us the test every time that the function was called whereas the present expression of this function allows the negative number to be reduced successively until either the list runs out and the Subscript exception is raised or the subtraction operation underflows (raising Overflow!) and this is handled in order that the Subscript exception may be raised instead.

The selection criteria might be moderately more complex:

1. select the first n elements; or

2. select all but the first n elements.

Call the function which implements the first criterion `take` and the function which implements the second `drop`. The selection could be slightly more exacting if we supply a predicate which characterises the required elements. The selection might then be:

1. select the leading elements which satisfy the criterion; or

2. select all but the leading elements which satisfy the criterion.

Call the function which implements the first criterion `takewhile` and the function which implements the second `dropwhile`. We will implement `take` and `takewhile`.

The *take* function

There are two base cases for this function. Either the number of elements to be taken runs out (i.e. becomes zero) or the list runs out (i.e. becomes `[]`). The first case is good but the second is bad. In the recursive call we attach the head to the result of taking one less element from the tail.

```
fun take (_, 0)      = []
  | take ([], _)     = raise Subscript
  | take (h :: t, n) = h :: take (t, n - 1) handle Overflow => raise Subscript
```

Note the excruciatingly familiar final case, again checking for underflow.

Exercise 5.3.1 *Construct a* `drop` *function with the same type as* `take`*. Your* `drop` *function should preserve the property for all lists* `l` *and non-negative integers* `n` *not greater than* `length l` *that* `take (l, n) @ drop (l, n)` *is* `l`.

The *takewhile* function

The base case for this function occurs when the list is empty. If it is not then there are two sub-cases.

1. The head element satisfies the predicate.

2. The head element does not satisfy the predicate.

If the former, the element is retained and the tail searched for other satisfactory elements. If the latter, the selection is over.

```
fun takewhile (p, [])     = []
  | takewhile (p, h :: t) = if p h then h :: takewhile (p, t) else []
```

Exercise 5.3.2 *Construct the analogous* `dropwhile` *function.*

Exercise 5.3.3 *Construct a* `filter` *function which returns* all *the elements of a list which satisfy a given predicate.*

5.3.2 Sorting lists

The sorting routine which we will develop is simple insertion sort. There are two parts to this algorithm. The first is inserting an element into a sorted list in order to maintain the ordering; the second is repeatedly applying the insertion function.

Inserting an element into a sorted list has a simple base case where the empty list gives us a singleton (one-element) list. All singleton lists are sorted. For non-empty lists we compare the new element with the head. If the new element is smaller it is placed at the front. If larger, it is inserted into the tail.

```
fun insert (x, [])        = [x]
  | insert (x, h :: t)    = if    x <= h
                            then x :: h :: t
                            else h :: insert (x, t)
```

This function has type `int * int list → int list`, due to default overloading resolving the use of '<=' to take integer operands. Notice the inconvenience of having to reconstruct the list in the expression `x :: h :: t` after deconstructing it by pattern matching. We could bind the non-empty list to a single variable, say l, and then use *let .. in .. end* to destruct it, binding `h` and `t` as before but the `as` keyword does this much more conveniently. The following implementation is equivalent to the previous one.

```
fun insert (x, [])           = [x]
  | insert (x, l as h :: t)  = if    x <= h
                               then x :: l
                               else h :: insert (x, t)
```

To sort a list we need only keep inserting the head element into the sorted tail. All empty lists are sorted.

```
fun sort []        = []
  | sort (h :: t)  = insert (h, sort t)
```

Exercise 5.3.4 *Implement the Quicksort algorithm due to C.A.R. Hoare. [You may find it useful to use the* `filter` *function from Exercise 5.3.3.]*

Exercise 5.3.5 *[This exercise is due to Stuart Anderson.] Where is the error in the following attempt to define a* prefix *function of type* `''a list → (''a list → bool)`. *The function should ensure that* prefix l_1 l_2 *returns* `true` *exactly when there is some list* l_3 *such that* l_1 @ l_3 *is identical to* l_2.

```
fun prefix []        l         = true
  | prefix (a :: l)  (b :: m)  = a = b andalso prefix l m
```

Exercise 5.3.6 *Write a permutations function,* `perm`, *of type* α list → (α list) list *which generates all permutations of a list. The following example illustrates this.*

```
perm [1, 2] = [[1, 2], [2, 1]]
```

If the input list is of length n then how long is the result?

5.3.3 List functions

Many simple list processing problems fall into one of two forms. They may involve simply applying a function to each element of the list in turn or they involve accumulating a result by applying a function to pairs consisting of an element of the list and the result of a recursive application. The first form is known as "mapping" a function across a list; the second is known as "folding" a list.

The *map* function

The `map` function has type (α → β) → ((α list) → (β list)). It behaves as though implemented by the following definition.

```
fun map f []        = []
  | map f (h :: t)  = f h :: map f t
```

The `map` function preserves totality: if `f` is a total function—meaning that it never goes into an infinite loop and never raises an exception—then so is `map f`. However, `map` itself is not a total function since we can find `f` and l such that `map f l` is not defined.

Note the rather lack-lustre role of the function parameter `f` in the implementation of `map` above: it is simply passed back into the recursive call of the function unchanged. This suggests that it can be factored out using a *let .. in .. end* as shown below.

```
fun map f = let fun    map_f []       = []
                  |    map_f (h :: t) = f h :: map_f t
            in  map_f  end
```

Such definitions are not always easy to read! However, depending on the implementation technique employed by the Standard ML system being used, factoring out the parameter in this way may lead to a more efficient implementation of map.

The *mapPartial* function

A common next step after mapping a function across a list is to filter out some unwanted results. These two steps can be combined in one by using the mapPartial function of type $(\alpha \to \beta$ option$) \to ((\alpha$ list$) \to (\beta$ list$))$.

```
fun mapPartial f []       = []
  | mapPartial f (h :: t) = case f h of
                              NONE => mapPartial f t
                            | SOME v => v :: mapPartial f t
```

Left and right folding

The foldr function, pronounced 'fold right', is also usually implemented as a curried function and has an even more sophisticated type than map. Its type is $((\alpha * \beta) \to \beta) \to (\beta \to ((\alpha$ list$) \to \beta))$.

```
fun foldr f e []       = e
  | foldr f e (h :: t) = f(h, foldr f e t)
```

As with map we could factor out the function argument f (and here also the value e) using *let* .. *in* .. *end*. Also just as with map, the foldr function preserves totality: if f is a total function, then so is foldr f. However, again foldr itself is not a total function since we can find a function f and a list l such that foldr f e l is not defined.

The utility of foldr can be seen since we may now implement the sort function with much less effort given the insert function. The following function, sort′, is equivalent.

```
fun sort' s = foldr insert [] s
```

Another use of foldr is found in the concat function of type $(\alpha$ list$)$ list $\to \alpha$ list.

```
fun concat s = foldr (op @) [] s
```

We may define a length′ function equivalent to the length function on page 28.

```
fun length' s = foldr (fn (x, y) => 1 + y) 0 s
```

Now we define an alternative to foldr, called foldl, pronounced 'fold left'. Its type is also $((\alpha * \beta) \to \beta) \to (\beta \to ((\alpha$ list$) \to \beta))$.

```
fun foldl f e []       = e
  | foldl f e (h :: t) = foldl f (f(h, e)) t
```

If the foldl function is applied to an associative, commutative operation then the result will be the same as the result produced using foldr. However, if the operator is not commutative then the result will in general be different. For example we can define listrev as follows.

```
fun listrev s = foldl (op ::) [] s
```

Whereas the definition using foldr gives us the list identity function.

```
fun listid s = foldr (op ::) [] s
```

The names for foldl and foldr arise from the idea of making the operator (called f in the function definition above) associate to the right or to the left.

5.4 The tree datatype

We will introduce a few important definitions for trees as defined by the parameterised `tree` datatype which was given earlier (on page 22).

Definition 5.4.1 (Nodes) *The nodes of an* α `tree` *are the values of type* α *which it contains. We can easily define a function which counts the number of nodes in a tree.*

```
fun nodes (empty)            = 0
  | nodes (node (_, t1, t2)) = 1 + nodes t1 + nodes t2
```

Definition 5.4.2 (Path) *A* path *(from tree* t_1 *to its subtree* t_k*) is the list* $t_1, t_2, \ldots, t_k$ *of trees where, for all* $1 \leq i < k$*, either* $t_i = node\,(n, t_{i+1}, t')$ *or* $t_i = node\,(n, t', t_{i+1})$*.*

Definition 5.4.3 (Leaf) *A tree* t *is a* leaf *if it has the form* $node\,(n, empty, empty)$*.*

Definition 5.4.4 (Depth) *We will now describe two Standard ML functions which calculate the depth of a tree. They both have type* α `tree` $\rightarrow$ `int`*.*

```
fun maxdepth (empty)            = 0
  | maxdepth (node (_, t1, t2)) = 1 + Int.max (maxdepth t1, maxdepth t2)
```

Above, `Int.max` *is the integer maximum function. The* `mindepth` *function just uses* `Int.min` *instead of* `Int.max`*.*

Definition 5.4.5 (Perfectly balanced) *A tree* t *is perfectly balanced if its maximum and minimum depths are equal.*

5.5 Converting trees to lists

There are many ways to convert a tree into a list. These are termed traversal strategies for trees. Here we will look at three: *preorder*, *inorder* and *postorder*.

Definition 5.5.1 (Preorder traversal) *First visit the root, then traverse the left and right subtrees in preorder.*

Definition 5.5.2 (Inorder traversal) *First traverse the left subtree in inorder, then visit the root and finally traverse the right subtree in inorder.*

Definition 5.5.3 (Postorder traversal) *First traverse the left and right subtrees in postorder and then visit the root.*

Exercise 5.5.1 *Define* α `tree` $\rightarrow$ α `list` *functions,* `preorder`*,* `inorder` *and* `postorder`*.*

5.6 Induction for trees

In order to prove using structural induction some properties of functions which process trees we must first give an induction principle for the tree datatype. Such a principle can be derived directly from the definition of the datatype.

Induction Principle 5.6.1 (Trees)

$$\frac{P\,(empty) \qquad (P(l) \wedge P(r)) \Rightarrow P(node\,(n, l, r))}{\forall t.\, P\,(t)}$$

Proposition 5.6.1 *A perfectly balanced tree of depth k has $2^k - 1$ nodes.*

Proof: The empty tree is perfectly balanced and we have $nodes\,(empty) = 0$ and $maxdepth\,(empty) = 0$ and $0 = 2^0 - 1$ as required.

Now consider a perfectly balanced tree t of depth $k + 1$. It is of the form $node\,(n, l, r)$ where the depth of l is k and the depth of r is also k. By the induction hypothesis we have that $nodes\,(l) = 2^k - 1$ and $nodes\,(r) = 2^k - 1$.

$$
\begin{aligned}
\text{LHS} &= nodes\,(node\,(n, l, r)) \\
&= 1 + nodes\,(l) + nodes\,(r) \\
&= 1 + (2^k - 1) + (2^k - 1) \\
&= 2^k + 2^k - 1 \\
&= 2^{k+1} - 1 \\
&= \text{RHS}
\end{aligned}
$$

□

5.7 The vector datatype

The Standard ML library provides vectors. A value of type `α vector` is a fixed-length collection of values of type α. Vectors differ from lists in their access properties. A vector is a random access data structure, whereas lists are strictly sequential access. With a list of a thousand elements it takes longer to access the last element than the first but with a vector the times are the same. The penalty to be paid for this convenience is that we are not able to subdivide a vector as efficiently as a list into its 'head' and its 'tail' and thus when programming with vectors we do not write pattern matching function definitions. Indeed, we *cannot* write pattern matching function definitions because we do not have access to the constructors of the vector datatype, they are not exported from the `Vector` structure in the Standard ML library. If we ever have occasion to wish to split a vector into smaller parts then we work with vector slices which are triples of a vector, a start index and a length.

Vector constants resemble list constants, only differing in the presence of a hash before the opening bracket. Thus this is a vector constant of length five and type `int vector`.

```
#[2,4,8,16,32]
```

We can obtain a sub-vector by extracting a slice from this one.

```
Vector.extract (#[2,4,8,16,32], 2, SOME 2) ≡ #[8,16]
```

We can convert a list into a vector.

```
Vector.fromList [2,4,8,16,32] ≡ #[2,4,8,16,32]
```

A suitable use of a vector is in implementing a lookup table. We could revisit our `day` function from page 6 and re-implement it using a vector in place of pattern matching.

```
fun day' d = Vector.sub (#["Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"], d) handle Subscript => "Sunday"
```

The effect is entirely the same. The handled exception provides a catch-all case just as the wild card in the last pattern caught all other arguments, including negative numbers. As we have noted, the subscripting function `Vector.sub` provides constant-time access into the vector unlike the indexing function `List.nth` for lists and thus it is appropriate that it has a different name to remind us of this different execution behaviour.

5.8 The Standard ML library

5.8.1 The List structure

Many of the functions which we defined in Section 5.1 are in the library structure List. For convenience we have used the same names for functions as the List structure does and our functions behave in the same way as the corresponding library functions. Thus for example, where we defined hd and tl the library provides List.hd and List.tl and these raise the exception List.Empty when applied to empty lists. Similarly the List structure provides List.@, List.concat, List.drop, List.foldl, List.foldr, List.last, List.length, List.mapPartial, List.nth, List.null, List.rev, List.revAppend and List.take. In addition to these functions List structure also provides others listed below. Most of these are almost self-explanatory when considering the type of the function together with its descriptive identifier.

List.all	:	$(\alpha \to \text{bool}) \to \alpha\ \text{list} \to \text{bool}$
List.exists	:	$(\alpha \to \text{bool}) \to \alpha\ \text{list} \to \text{bool}$
List.filter	:	$(\alpha \to \text{bool}) \to \alpha\ \text{list} \to \alpha\ \text{list}$
List.find	:	$(\alpha \to \text{bool}) \to \alpha\ \text{list} \to \alpha\ \text{option}$
List.partition	:	$(\alpha \to \text{bool}) \to \alpha\ \text{list} \to \alpha\ \text{list} * \alpha\ \text{list}$
List.tabulate	:	$\text{int} * (\text{int} \to \alpha) \to \alpha\ \text{list}$

5.8.2 The ListPair structure

Often we find ourselves working with pairs of lists or lists of pairs. The ListPair structure in the standard library provides a useful collection of operations on values of such data types. Once again, the functions provided are self-explanatory.

ListPair.all	:	$(\alpha * \beta \to \text{bool}) \to \alpha\ \text{list} * \beta\ \text{list} \to \text{bool}$
ListPair.foldl	:	$(\alpha * \beta * \gamma \to \gamma) \to \gamma \to \alpha\ \text{list} * \beta\ \text{list} \to \gamma$
ListPair.foldr	:	$(\alpha * \beta * \gamma \to \gamma) \to \gamma \to \alpha\ \text{list} * \beta\ \text{list} \to \gamma$
ListPair.map	:	$(\alpha * \beta \to \gamma) \to \alpha\ \text{list} * \beta\ \text{list} \to \gamma\ \text{list}$
ListPair.unzip	:	$(\alpha * \beta)\ \text{list} \to \alpha\ \text{list} * \beta\ \text{list}$
ListPair.zip	:	$\alpha\ \text{list} * \beta\ \text{list} \to (\alpha * \beta)\ \text{list}$
ListPair.exists	:	$(\alpha * \beta \to \text{bool}) \to \alpha\ \text{list} * \beta\ \text{list} \to \text{bool}$

5.8.3 The Vector structure

In addition to the functions which we have seen the Vector structure in the Standard ML library also provides the following. The functions Vector.foldli and Vector.foldri differ from familiar left and right folding in that they also make use of the integer index into the vector and thus are near relatives of the **for** loops in Pascal-like programming languages. Such loops supply an integer loop control variable which is automatically incremented on each loop iteration (and may not be altered within the loop body). The Vector.foldli and Vector.foldri functions operate on vector slices.

Vector.concat	:	$\alpha\ \text{vector list} \to \alpha\ \text{vector}$
Vector.foldl	:	$(\alpha * \beta \to \beta) \to \beta \to \alpha\ \text{vector} \to \beta$
Vector.foldr	:	$(\alpha * \beta \to \beta) \to \beta \to \alpha\ \text{vector} \to \beta$
Vector.foldli	:	$(\text{int} * \alpha * \beta \to \beta) \to \beta \to \alpha\ \text{vector} * \text{int} * \text{int option} \to \beta$
Vector.foldri	:	$(\text{int} * \alpha * \beta \to \beta) \to \beta \to \alpha\ \text{vector} * \text{int} * \text{int option} \to \beta$
Vector.length	:	$\alpha\ \text{vector} \to \text{int}$
Vector.tabulate	:	$\text{int} * (\text{int} \to \alpha) \to \alpha\ \text{vector}$

Chapter 6

Evaluation

Some functional programming languages are lazy, meaning that an expression will not be evaluated unless its value is needed. This approach seems to be a very sensible one: the language implementation is attempting to optimize the execution of programs by avoiding any unnecessary computation. Perhaps surprisingly, this evaluation strategy will not always improve the efficiency of programs since it may involve some extra work in managing the delayed evaluation of expressions.

Lazy programming languages are difficult to implement efficiently (see [PJL92]) and economically ([Jon92] describes the 'space leaks' which occur in lazy languages when dynamically allocated memory is lost, never to be reclaimed). There are also the pragmatic difficulties with lazy programming which are mentioned in Robin Milner's "How ML Evolved": the difficulty of debugging lazy programs and the difficulty of controlling state-change or (perhaps interactive) input and output in lazy languages which are not purely functional. Because of some of these difficulties and the desire to include imperative features in the language, Standard ML uses a call-by-value evaluation strategy: expressions are evaluated irrespective of whether or not the result is ever needed. The lazy evaluation of expressions is then achieved by the programmer rather than by the language. We now discuss techniques for implementing the lazy evaluation of expressions.

6.1 Call-by-value, call-by-name and call-by-need

Unlike many other programming languages, functions in Standard ML can be designated by arbitrarily complex expressions. The general form of an application is $e\ e'$, which is evaluated by first evaluating e to obtain some function and then evaluating e' to obtain some value and finally applying the function to the value. In general, the expression e can be quite complex and significant computation may be required before a function is returned. This rule for evaluation of function application uses the call-by-value parameter passing mechanism because the argument to a function is evaluated before the function is applied.

An alternative strategy is call-by-name. Here the expression e' is substituted for all the occurrences of the formal parameter. The resulting expression is then evaluated as normal. This might mean that we evaluate some expressions more than once. Clearly, call-by-value is more efficient. The following important theorems make precise the relationship between the two forms of evaluation.

> **Theorem 6.1.1 (Church Rosser 1)** *For a* purely functional *language, if call-by-value evaluation and call-by-name evaluation both yield a well-defined result then they yield the same result.*

> **Theorem 6.1.2 (Church Rosser 2)** *If a well-defined result exists for an expression then the call-by-name evaluation strategy will find it where, in some cases, call-by-value evaluation will not.*

Lazy languages do not use call-by-name evaluation; they use call-by-need. Here when the value of an expression is computed it is also stored so that it need never be re-evaluated, only retrieved. In practice, a lazy language might do more computation than strictly necessary, due to definition of functions by pattern matching. In these cases it would not meet the applicability criteria of the Second Church Rosser theorem because it imperfectly implements call-by-name.

6.2 Delaying evaluation

One form of delayed evaluation which we have already seen is conditional evaluation. In an *if* .. *then* .. *else* .. expression only two of the three sub-expressions are evaluated. The boolean expression will always be evaluated and then, depending on the outcome, one of the other sub-expressions will be evaluated. The effect of a conditional expression would be different if all three of the sub-expressions were always evaluated. This explains why there is no `cond` function [of type (bool $* \alpha * \alpha) \to \alpha$] in Standard ML.

Similarly, a recursive computation sometimes depends upon the outcome of evaluating a boolean expression [as with the `takewhile` function (on page 32)]. In cases such as these, the evaluation of expressions can be delayed by placing them in the body of a function. By packaging up expressions in this way, we can program in a 'non-strict' way in Standard ML and we can describe recursive computations and we can define infinite objects such as the list of all natural numbers or the list of all primes. Consider the following function.

```
fun FIX f x = f (FIX f) x
```

Exercise 6.2.1 *What is the type of* FIX*? You might benefit from seeing this function with the derived form removed and some redundant parentheses inserted for clarity.*

```
val rec FIX = fn f => (fn x => (f (FIX f)) x)
```

The purpose of the `FIX` function is to compute fixed points of other functions. [Meaning: x is a fixed point of the function f if $f(x) = x$.] How is this function used? Consider the `facbody` function below. No derived forms are used here in order to make explicit that this is not a recursive function [not a `val rec` ..].

```
val facbody = fn f => fn 0 => 1
                       | x => x * f(x - 1)
```

If this function were to be given the factorial function as the argument `f` then it would produce a function as a result which was indistinguishable from the factorial function. That is, the following equivalence would hold.

$$\text{fac} \equiv \text{facbody}\,(\text{fac})$$

But this is just the equivalence we would expect to hold for a fixed point. What would then be the result if we defined the `fac` function as shown below.

```
val fac = FIX (facbody)
```

The `fac` function will then compute the factorials of the integers which it is given as its argument. Notice that neither the declaration of `fac` nor the declaration of `facbody` were recursive declarations; of the three functions which were used only `FIX` is a recursive function.

This effect is not specific to computing factorials, it would work with any recursive function. The functions below use `FIX` to define the usual map function for lists.

```
val mapbody = fn m => fn f =>
    fn [] => [] | h :: t => f h :: m f t

fun map f l = (FIX mapbody) f l
```

This method of defining functions succeeds because the FIX function delays a part of the computation. In its definition the parenthesised sub-expression FIX f which appears on the right-hand side is an unevaluated function term [sometimes called a suspension] equivalent to *fn* x => FIX f x. Crucially, this is the role of x in the definition; to delay evaluation. Without it the function would compute forever.

```
(* Always diverges when used *)
fun FIX' f = f (FIX' f)
```

Exercise 6.2.2 *What is the type of* FIX′*?*

This version of the function makes it much easier to see that the fixed point equation is satisfied—that f (FIX′ f) ≡ FIX′ f—and in a lazy variant of the Standard ML language the FIX′ function would be perfectly acceptable and would operate just as FIX does.

6.3 Forcing evaluation

Non-strict programming in Standard ML uses the pre-defined unit type. This peculiar type has only one element, written "()", and also called "unit". This representation for unit is a derived form for the empty record, "{}". Horrifyingly, that is also the way to represent the *type* of the empty record so we find that we have {} : {} in Standard ML!

The use of the unit type serves to convey the idea that the parameter which we pass to the function will never be used in any way since there are no operations on the unit element and it also conveys no information because there is only one value of this type. It is possible to delay the evaluation of expressions with unused values of any type but we would not wish to do this. The type of a Standard ML function acts as important documentation about its behaviour and we would not wish it to have a misleading source type, say int or α, since the resulting confusion about the type would make our program harder to understand.

The type which delayed expressions have is called a *delayed* type. This is a parameterised type constructor as defined below.

```
type α delayed = unit → α
```

We could then try to label integer expressions as being delayed and thereby turn them into functions of type "unit → int". If we need the integer value which they would compute then we can force the evaluation of the integer expression by applying the function to (). We will now attempt to define the functions which force evaluation and delay evaluation. The force function has type α delayed → α. This is a simple function to implement.

```
val force : α delayed → α = fn d => d ()
```

The function delay below has type α → α delayed. It should be the inverse of force and for all expressions *exp* we should have that force (delay (*exp*)) evaluates to the same value as *exp* itself.

```
val delay : α → α delayed = fn d => (fn () => d) defn.of.delay
```

This function has the correct type and has achieved the aim that force (delay (*exp*)) evaluates to the same value as *exp* for all expressions but it has not achieved the effect we wanted. The additional requirement was that it should delay the evaluation of an expression. However, consider the evaluation of a typical application of delay using Standard ML's call-by-value semantics.

```
delay (14 div 2) ≡ (fn d => (fn () => d)) (14 div 2)
                 ≡ (fn d => (fn () => d)) (7)
                 ≡ fn () => 7
```

This is not the desired effect since we wished to have the expression `delay (14 div 2)` be identical to `(fn () => 14 div 2)`. It is not possible to implement a `delay` function of type $\alpha \to \alpha$ `delayed` in Standard ML since the expression will always be evaluated and the resulting value passed to the function. We will write "`fn () =>` *exp*" from now on—or in some circumstances use an equivalent derived form—but continue to pronounce this as "delay *exp*". Our functions `delay` and `force` implement call-by-name expressions because repeated applications of `force` repeat previous computations.

6.4 From call-by-value to call-by-name

Now that we have all the machinery available to delay the evaluation of expressions we may ask whether a call-by-name variant can always be found for an existing call-by-value function. This question can be answered positively and we will now sketch out a recipe. Consider a function `f` with the following form:

```
fun f x = ··· x ··· x ···
```

and assume that this function has type $X \to Y$. We can provide a call-by-name variant which has type X `delayed` $\to Y$ where every occurrence of `x` is replaced by `force (x)`:

```
fun f x = ··· (force (x)) ··· (force (x)) ···
```

finally replace any applications of the function, `f (`*exp*`)`, by `f (fn () =>` *exp*`)`.

6.5 Lazy datatypes

Although any call-by-value function can be transformed into a call-by-name variant the chief interest in delaying evaluation in functional programming is the ability to create datatypes such as infinite sequences and infinitely branching trees. In a shocking and inexcusable abuse of technical terminology we will call these 'lazy' datatypes even though they simulate call-by-name evaluation instead of call-by-need evalution. The most important point about these datatypes is the representation ability which they offer; not that they optimise computations.

Consider the datatype of infinite sequences of integers. This can be described as a Standard ML datatype with a single constructor, `cons`.

```
datatype seq = cons of int * (seq delayed)
```

This definition provides us with a constructor, `cons`, of type `(int * (seq delayed))` $\to$ `seq`. The functions to return the head and tail of an infinite sequence are much simpler than those to return the head and tail of a list. Since the sequence can never be exhausted there is no exceptional case behaviour. However, note that the `tail` function is partial since the evaluation of `force t` may fail to terminate.

```
fun head (cons (h, _)) = h
fun tail  (cons (_, t)) = force t
```

These functions have types `seq` $\to$ `int` and `seq` $\to$ `seq` respectively.

Exercise 6.5.1 *Write a function to give a list of the first n integers in a sequence.*

We can construct a simple function which returns the infinite sequence which has the number one in every place.

```
fun ones () = cons (1, ones)
```

Unfortunately the `cons` constructor does not compose since `cons (first, cons (second, tail))` is not well-typed. Life is much easier if we define a lazy version of `cons` which does compose.

```
fun lcons (h, t) () = cons (h, t)
```

We may now easily define the infinite sequence which has one in first place and in every other odd place with every other digit being zero.

```
fun oneszeroes () = cons (1, lcons (0, oneszeroes))
```

In general we may define more interesting sequences by thinking of defining a whole family of sequences simultaneously. For example, the sequences which start at n and move up in steps of one.

```
fun from n () = cons (n, from (n + 1))
```

Using the `from` function we may define the sequence of all natural numbers quite easily. These are simply all the numbers from zero upwards.

```
val nats = force (from 0)
```

Given a sequence constructed in this way we could produce another infinite sequence by supplying a function which can be applied to each element of the sequence, thereby generating a new sequence. The function `tentimes`, when applied to a sequence s, will return a sequence where the elements are the corresponding elements of s multiplied by ten.

```
fun tentimes (cons (h, t)) = cons (10 * h, tentimes o t)
```

Using this function we may define `tens` as the result of the composition `(tentimes o ones)` and `hundreds` as the result of the composition `(tentimes o tentimes o ones)`.

Exercise 6.5.2 *Given the following definitions, what is* `next (force zeroes)`*?*

```
fun zeroes ()          = cons (0, zeroes)
fun next (cons (h, t)) = cons (h + 1, next o next o t)
```

6.6 An example: Computing the digits of *e*

(This example is taken from [Tur82] and it was first implemented in David Turner's excellent lazy functional programming language Miranda™ (a trademark of Research Software Ltd.). Trivia fans might like to know that it was proposed to Turner as a challenge by the famous Dutch computer scientist Edsger W. Dijkstra. It is a folk theorem in computer science that all challenge problems were initially proposed by Edsger W. Dijkstra.)

As an example of a program which uses infinite sequences, consider the problem of computing the digits of the transcendental number e. We would like to calculate as many digits of e as we wish. Notice that the decimal expansion of e is an infinite sequence of integers. (Each integer being a single decimal digit.) We could then use in our implementation the infinite sequence datatype which we have just defined.

The number e can be defined as the sum of a series. The terms in the series are the reciprocals of the factorial numbers.

$$
\begin{aligned}
e &= \sum_{i=0}^{\infty} \frac{1}{i!} \\
&= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots \\
&= 2.7182818284590\ldots \qquad \text{(base 10)} \\
&\qquad\qquad \text{a base in which the } i\text{th digit has weight } 1/10^{i-1} \\
&= 2.1111111111111\ldots \\
&\qquad\qquad \text{a funny base in which the } i\text{th digit has weight } 1/i!
\end{aligned}
$$

Both the decimal expansion and the expansion in the funny base where the ith digit has weight $1/i!$ can be expressed as infinite integer sequences. The problem is then to convert from this funny base to decimal.

For any base we have:

- take the integer part as a decimal digit;
- take the remaining digits, multiply them all by ten and renormalise (using the appropriate carry factors);
- repeat the process with the new integer part as the next decimal digit.

Note: The *carry factor* from the ith digit to the $(i-1)$th digit is i. I.e. when the ith digit is $\geq i$ we add 1 to the $(i-1)$th digit and subtract i from the ith digit.

```
fun carry (i, cons (x, u))     = carry' (i, x, u ())
and carry' (i, x, cons (y, v)) = cons (x+y div i, lcons (y mod i, v));

fun norm (i, cons (x, u)) ()   = norm' (i, x, u ())
and norm' (i, x, cons (y, v))  = norm" (i, y+9<i, cons (x, norm (i+1, cons (y, v))))
and norm" (i, nocarry, s)      = if nocarry then s else carry (i, s);

fun normalise s                = norm (2, cons (0, tentimes o s)) ();

fun convert (cons (x, u)) ()   = cons (x, (convert o normalise) u);

val e                          = convert (cons (2, ones)) ();
```

Chapter 7

Abstract data types

Thus far our programs have been small and simple and it would have seemed excessive to have structured them into modular units. When we come to write larger programs we will want to encapsulate some functions together with a datatype in order to control the access to the elements of the datatype. The construction we use for this purpose is *abstype* .. *with* .. *end*.

We will implement an abstract data type for sets. These are unordered collections of values. A membership test is provided. Duplications are not significant: there is no way to test how many times a value occurs in a set. The problem is then to provide a way to construct sets and test for membership without giving away other information such as the number of times a value appears in the set.

The following abstract data type introduces a type constructor, set, a value emptyset and two functions, addset and memberset. The constructors null and ins are hidden, they are not visible.

```
abstype α set = null | ins of α * α set
with
     val emptyset = null
     val addset = ins
     fun memberset (x, null) = false
        | memberset (x, ins(v,s)) = x = v orelse memberset (x, s)
end
```

It might seem somewhat futile to hide the names null and ins and then provide emptyset and addset. The point is that in so doing, we take away the *constructor* status of null and ins and that means that they cannot be used in pattern matching to destruct the constructed value and see inside.

But it would seem that nothing we have described could not be achieved with the features of the Standard ML language which we knew already, albeit in a slightly more complicated definition.

```
local
     datatype α set = null | ins of α * α set
in
     type α set = α set
     val emptyset = null
     val addset = ins
     fun memberset (x, null) = false
        | memberset (x, ins(v,s)) = x = v orelse memberset (x, s)
end
```

So in what sense is the abstract data type more abstract than the type which is defined here?

The problem is that *equality* is available on the sets which are defined using the second form of the definition and the equality which is provided is not the one we want. No equality test is permitted if we use an *abstype* definition. If we want to allow equality we must implement it ourselves.

```
abstype α set = null | ins of α * α set
with
     val emptyset = null
     val addset = ins
     fun memberset (x, null) = false
       | memberset (x, ins(v,s)) = x = v orelse memberset (x, s)
     local
       fun subset (null, _) = true
         | subset (ins(x, s1), s2) = memberset (x, s2) andalso subset (s1, s2)
     in
       fun equal (s1, s2) = subset (s1, s2) andalso subset (s2, s1)
     end
end
```

We have made the equal function available but not the subset function which we used in its implementation.

Abstract data types are *first-class* values in Standard ML because they may be passed to functions as arguments, as shown below.

```
fun allmembers ([], _)      = true
  | allmembers (h :: t, s) = memberset(h,s) andalso allmembers (t, s)
```

This function has type (''a list * ''a set) → bool. They may also be returned from functions as results. The following function has type (α list * α set) → α set.

```
fun addmembers ([], s)      = s
  | addmembers (h :: t, s) = addset (h, addmembers (t, s))
```

7.1 Programming with abstract data types

More than with any other part of Standard ML programming with abstypes requires a certain discipline. We have in the abstype concept a means of localising the creation of values of a particular type and the reason for wishing to do this is that we can validate the creation of these values, rejecting certain undesirable ones, perhaps by raising exceptions. Moreover, we can exploit the validation within the abstype definition itself. Here we implement sets as ordered lists without repetitions.

```
abstype α ordered_set = Set of ((α * α) → bool) * α list
with
   fun emptyset (op <=) = Set (op <=, [])

   fun addset (x, Set (op <=, s)) =
     let
       fun ins [] = [x]
         | ins (s as h::t) =
               if x = h then s else
               if x <= h then x :: s else h :: ins t
     in Set (op <=, ins s)
     end

   fun memberset (x, Set (_, [])) = false
     | memberset (x, Set (op <=, h::t)) =
       h <= x andalso (x = h orelse memberset (x, Set (op <=, t)))
end
```

The abstype mechanism ensures that sets have been created either by the function `emptyset` or by the function `addset`. Both of these functions construct sorted lists and thus it is safe to exploit this ordering in `memberset`. By this time, because we are imposing a particular order on the values in the list it is crucially important we do not provide access to the `Set` constructor (say via *val* `mk_set = Set`) because otherwise a user of this `ordered_set` abstype could create an unordered set thus.

```
val myset = mk_set (op <=, [1, 5, 3, 2, 8])
```

This directly constructed set value will then give unexpected results, as detailed below.

```
memberset (1, myset) = true
memberset (5, myset) = true
memberset (3, myset) = false
memberset (2, myset) = false
memberset (8, myset) = true
```

In the context of a larger program development these occasional unexpected results might sometimes lead to observable errors which would be difficult to diagnose.

Exercise 7.1.1 *Would it be prudent to add the following definition of* `equal` *within the definition of ordered set? What are the circumstances in which its use might give a misleading result?*

$$\textit{fun}\ \texttt{equal (Set (\_, s}_1\texttt{), Set (\_, s}_2\texttt{))} = \texttt{s}_1 = \texttt{s}_2$$

Having seen a simple implementation of sets as ordered lists we could improve this to an implementation which used binary trees and improve that to an implementation which used balanced trees. Fortunately this has already been done for us by Stephen Adams [Ada93] whose paper describes his implementation.

Chapter 8

Imperative programming

Standard ML is not a pure functional language, it is a higher-order imperative language. We have already [briefly] considered exceptions and now we consider assignment and input and output. Consider the following sequence of value declarations.

```
val x = 0;
val x = x + 1;
```

This resembles a sequence of assignments in an imperative program where first the variable `x` is given an initial value of zero and then the value of `x` is incremented. The final effect is, of course, that `x` holds the value one. On the basis of this example it might seem that changes to the environment are like changes of state. To clarify the difference between a sequence of assignments and a sequence of value declarations consider the declarations which appear below.

```
val x = 0;
val x = x < 1;
val x = if x then 1 else 0;
```

Once again the final effect is to leave `x` bound to one but in the middle of this process the identifier `x` was re-used to denote a boolean value. In a typed programming language which distinguishes between integers and booleans no sequence of assignments could ever achieve this effect and so we see that the value declaration mechanism brings about a *re-declaration* or a *re-binding* of the identifier `x`. We conclude that it is necessary to distinguish between the evironment and the state.

8.1 References

In Standard ML, updatable cells are accessed via references. There is a type constructor, `ref` and a value constructor also called `ref` of type $\alpha \to \alpha$ `ref`. Thus `ref 1` is an `int ref`, `ref 1.0` is a `real ref`, `ref #"A"` is a `char ref`, `ref Empty` is an `exn ref` and so on. We may have references to functions and even references to other references. In order to see that `ref` is non-functional we need only evaluate the expression `(ref 0) = (ref 0)` in order to discover that it evaluates to `false`. This is the value we should expect since we are comparing two freshly generated reference values.

Given a reference we must have a means of dereferencing it to retrieve the value which it references. If `r` is a `real ref` then `!r` is the real number which it references. The dereferencing operator behaves as though defined by `fun ! (ref x) = x`.

If r_1 and r_2 are references to values of the same type then we may test them for equality. This rule applies even when r_1 and r_2 are references to values of a type which does not admit equality (such as a function type or `exn` or an abstype). Notice though that in those cases where equality is defined upon the values that $r_1 = r_2$ is a distinctly different test from $!r_1 = !r_2$. We have that $r_1 = r_2$ implies $!r_1 = !r_2$ but not the other way around. Equality and dereferencing are the only built-in operations on references. In particular, arithmetic operations are not provided: it is not possible in Standard ML to increment a reference in order to move on to the next memory location.

There is a significant distinction to be made between comparing references and comparing values. Given the following bindings, `a` and `b` are equal but not `a` and `c`.

```
val a = ref 1; val b = a; val c = ref 1;
```

Notice that in Standard ML there is no way to declare a reference and omit the initial value. A declaration such as `val n : int ref` is *not* legal.

We now begin to see that the imperative features of the language can have an impact of the type-checking of programs. Carl and Elsa Gunter and Dave MacQueen write in [GGM91]:

> In the Definition of Standard ML, a unary type constructor α F is said to *admit equality* if t F is an equality type whenever the parameter t is. A constructed type t F admits equality only if both t and F admit equality. This extends to *n*-ary type constructors in the obvious way. Unfortunately, this definition is incomplete for the inference of equality properties because of the presence of certain special type constructors that have stronger equality properties. For example, the type `t ref` admits equality regardless of whether `t` does.

The paper then gives a treatment of equality types which uses a more discriminating test for admission of equality than that in the Definition of Standard ML [MTHM97].

Armed with this knowledge about references we are now able to tackle programming tasks which defeated us before. One use of references is to enable us to embellish our abstract data type for ordered sets with an equality test which is implemented as equality on sorted lists without repetitions. As before we commit to a particular ordering when we use the `emptyset` function. Adding elements to a set inserts them into a list, ordered by our chosen ordering. The problem of implementing an equality test on two sets as an equality test on two ordered lists is knowing whether the lists have been sorted using the same ordering. We cannot test the ordering functions with equality so we must adopt an indirect solution. Instead of storing only the function when we create an empty set we store a reference to the function. The benefit to be gained from this additional indirection is that in the equality test we can compare these references. When we cannot guarantee by this approach that the sets are ordered in the same way we raise an exception to signal their incompatibility.

```
abstype α ordered_set = Set of ((α * α) → bool) ref * α list
with
   fun emptyset (op <=) = Set (ref (op <=), [])

   fun addset (x, Set (r as ref (op <=), s)) =
      let
         fun ins [] = [x]
           | ins (s as h::t) =
                 if x = h then s else
                 if x <= h then x :: s else h :: ins t
      in Set (r, ins s)
      end

   fun memberset (x, Set (_, [])) = false
     | memberset (x, Set (r as ref (op <=), h::t)) =
         h <= x andalso (x = h orelse memberset (x, Set (r, t)))

   exception Incompatible
   fun equal (Set (r1, s1), Set (r2, s2)) =
      if r1 = r2 then s1 = s2 else raise Incompatible
end
```

Exercise 8.1.1 *Raising an exception when the two references are different is not necessary, it is enough to sort one of the sets using the ordering function from the other. Implement this extension.*

8.2 Assignment

Creating references to values and comparing references is all very jolly but the real reason to introduce references into the language is to enable the programmer to store values in a particular location for later retrieval and possible modification. Modification of the stored value is achieved by the use of the infix ":=" assignment operator. For example, if `n` is an `int ref` then `n := 1` and `n := !n + 1` are legal assignments but we will never see `n := n + 1`.

The assignment operator has type `(α ref * α) → unit`. The result type of `unit` indicates that this operator achieves its effect by side-effect because we know in advance that the result returned by the function will be `()`. This value is needed for the result because an operator is only an infix function and all Standard ML functions must return some value, even if their purpose is to change the state.

Exercise 8.2.1 *For the language designers, the other possible choice for the type of the assignment operator would be* `(α ref * α) → α`. *The effect of allowing this choice would be to allow the use of the value from an assignment. Define an infix operator "::=" of type* `(α ref * α) → α` *which has this property. [Exercise care, there is a potential pitfall.]*

The assignment operation genuinely increases the power of the language, as references did. We can now program call-by-need versions of the `delay` and `force` functions from page 40. These functions use references to an auxiliary datatype of suspended evaluations.

```
local
   datatype α hitchcock =
      mcguffin of unit → α
      | corpse of α
in
   abstype α susp = depalma of α hitchcock ref
   with
      fun delay f = depalma (ref (mcguffin f))
      fun force (depalma (ref (corpse x))) = x
        | force (depalma (loc as ref (mcguffin f))) =
              let val c = f ()
              in loc := corpse c; c
              end
   end
end
```

8.3 Sequential composition

In imperative programming it is essential to have a method of controlling the order in which the evaluation of the components of a compound expression takes place. We wish to be able to build a sequence of expressions into a single compound expression. Given expressions e_1, e_2 and e_3 we could force them to be evaluated in the correct order by using *let* .. *in* .. *end* as shown below. We are not interested in the results of the expressions, we simply throw them away by using the wild card as the pattern in the value binding.

```
let val _ = e1
    val _ = e2
in  e3
end
```

This seems rather cumbersome. Fortunately Standard ML provides a neater way to control the order of evaluation of expressions. The composition operator in Standard ML is a semicolon and parentheses may be used to build a single expression from a sequence of expressions. There is no requirement for all of the subexpressions to return the unit value as their result or to have the same type. The expression `(n := 5; true)` evaluates to `true`. The expression `(true; n := 5)` evaluates to `()`.

8.4 Iteration

A natural companion for assignment and sequential composition is an iteration mechanism. It is not essential since any expression which uses an iterative expression could be reformulated as a recursive function. The general form of a loop in Standard ML is the reserved word *while* followed by a boolean expression involving a pointer (or other counter) followed by the word *do* followed by an expression which performs some computation, changing the state and advancing the pointer.

As an example of the use of references and the *while* loop we will supply an imperative version of the factorial function.

```
fun ifac N =
    let val n = ref N and i = ref 1
    in
      while !n <> 0 do
        (i := !i * !n; n := !n - 1);
      !i
    end
```

The function `ifac` has type `int -> int` just as the function `fac` has. Given just the type of a Standard ML function, there is no systematic method which can determine whether the function will ever cause a change to the program state.

Exercise 8.4.1 *The* `while` *loop is a derived form. Can you work out how it is defined?*

8.5 Types and imperative programming

Thus far everything seems to have gone very well but there are problems just ahead when we consider the interaction of references and polymorphism. There is an ordering "$\succ$" corresponding to "degree of polymorphism" such that the following relation holds between the types of polymorphic functions.

$$\forall\alpha.\forall\beta.(\alpha \to \beta) \succ \forall\alpha.(\alpha \to \alpha) \succ \forall().(\mathtt{int} \to \mathtt{int})$$

Given a reference to a polymorphic function, an assignment could make the type of the function which is referenced less polymorphic as allowed by the "$\succ$" ordering. However, such behaviour could lead to expressions which can be statically type-checked but which would produce a run-time type error when executed. A short example is given below.

```
let val r = ref(fn x => x) in (r := (fn x => x + 1); !r true) end
```

If the function `r` was assigned the polymorphic type $\forall\alpha.((\alpha \to \alpha)\ \mathtt{ref})$ then the assignment and the dereferenced function application would both be correctly typed but the program would "go wrong" at run-time by attempting to add a boolean value to an integer. The type system of Standard ML does not allow programs to go wrong in this way and thus the example must be rejected by a compiler.

8.5.1 Type safety conditions

We desire a type system which permits secure, type-safe, implementation of imperative routines without simply imposing the unnecessarily harsh restriction that the programmer may only make references to monomorphic values. Such a restriction would mean that the previous ordered set and lazy expression examples would not be allowable. The problem of designing a permissive type system for imperative programming has proved to be one of the most difficult in the history of programming language design. The essential tension comes from several conflicting desires:

1. to detect all violations of types;
2. to compile as many programs as possible; and
3. to provide a type system which is intuitive for programmers.

This last complication is a serious concern. A programming language which infers types should not make the types of functions so complex that a programmer can no longer have reasonable conviction about the type to expect to see reported by the compiler. Why do we consider this to be important? Because the type information computed by the system is excellent diagnostic information which can be used to debug programs without ever executing them. Consider the following function which is intended to dereference a list of references and apply a function to each. This is a version of `map` for reference values, called `rap`.

```
fun rap f []       = []
  | rap f (h :: t) = f !h :: rap f t
```

We would expect the compiler to calculate the type $(\alpha \to \beta) \to ((\alpha\ \mathtt{ref}\ \mathsf{list}) \to (\beta\ \mathsf{list}))$ for the `rap` function. Instead the type calculated is $((\alpha\ \mathtt{ref} \to \alpha) \to \beta \to \gamma) \to \beta\ \mathsf{list} \to \gamma\ \mathsf{list}$. This type is obtained because `!` is a function and its application does not bind more tightly than the application of the function `f`. Function application associates to the left in Standard ML and so the subexpression `f !h` denotes `(f !) h` rather than `f (!h)` as intended. It is difficult to see how this error could have been made more evident to the programmer than by the calculation of the very unusual type for the function.

8.5.2 Implementing type safety

A number of methods of combining polymorphic definition and imperative features were proposed by Mads Tofte [Tof90] and Standard ML adopts the simplest of them. A persuasive argument is provided by Andrew Wright [Wri95] that relatively little expressiveness is lost in making this choice. Tofte's approach divides expressions into the categories of *expansive* and *non-expansive*. The essence of the idea is that only non-expansive expressions may be polymorphic.

What is a non-expansive expression? We may characterise it as a value and thus Standard ML is said to have 'value polymorphism'. More precisely, a non-expansive expression is

- a constant;
- an identifier;
- a record where the labels are associated with non-expansive expressions;
- an application of any constructor except `ref` to a non-expansive expression; and
- any *fn* expression

and anything of this form supplemented with parentheses, type constraints and derived forms. Any other expression is deemed expansive, even if it does not use references.

The following value declarations are not permitted since they contain expansive expressions of polymorphic type.

```
(* All rejected by the compiler *)
val r = rev []
val r = ref []
val r = ref (fn x => x)
val concat = foldr (op @) []
```

Notice that the value polymorphism applies even for purely functional programs which make no use of the imperative features of the language. When confronted with a polymorphic expression which is rejected because it contains a free type variable (at the top level) there are several possible solutions. One is to simplify the expression to remove any unneeded function applications—such as replacing rev [] with []—and another solution is to add an explicit type constraint if the desired result is monomorphic. For expressions which denote polymorphic functions the introduction of an explicit *fn* abstraction solves the problem. For example, concat can be written thus

```
val concat = fn s => foldr (op @) [] s
```

or using the derived form for this which we saw on page 34.

The following value declarations *are* legal because their (monomorphic) types can be determined.

```
let  val r = rev []  in  val s = implode r  end
let  val x = ref []  in  val s = 1 :: !x  end
```

The following imperative version of the rev function (on page 29) contains only occurrences of expansive expressions within the body of a *fn* abstraction (recall that the *fun* keyword is a derived form which includes a *fn* abstraction). This function reverses lists in linear time and is thus called fastrev.

```
fun fastrev l =
    let val left = ref l and right = ref []
    in while not (null (!left)) do
           ( right := hd (!left) :: !right;
             left := tl (!left)             );
      !right
    end
```

The Standard ML system will compute the type α list → α list for this function, just as it did for the functional version of list reversal.

8.6 Arrays

Lists are the central datatype in applicative programming. In imperative programming the array is the central datatype. Arrays are provided in a type-safe way in Standard ML through the use of the following operations of the Array structure in the Standard ML library.

- the type α Array.array;
- the creation functions Array.array of type (int * α) → α Array.array and Array.fromList of type α list → α Array.array;
- the Array.update operation of type (α Array.array * int * α) → unit; and
- the operation Array.sub with type (α Array.array * int) → α with exception Subscript. Subscripting begins from 0.

The `Array.array` type constructor admits equality and the equality which is provided is equality of reference. Thus, given the following declarations,

```
val a = Array.fromList [1]
val b = Array.fromList [1]
```

then `a` and `b` are *not* equal.

8.7 Memoisation

Given the imperative features of Standard ML and the array datatype we may now construct an extremely useful function which allows the Standard ML programmer to achieve greater efficiency from programs without damaging the clarity of the functional style of programming.

The simple technique of *memoisation* involves storing a value once it has been computed to avoid re-evaluating the expression. We present a very simple version of the memoisation function which assumes:

1. the function to be memoised returns integers greater than zero;
2. the only arguments of the function are between zero and fifty.

A more general version which does not have these restrictions was implemented by Kevin Mitchell. It can be found in the Edinburgh ML library [Ber91].

```
fun memo f =
let
  val ans = Array.array (50,0)
  fun compute ans i =
    case Array.sub (ans, i) of
        0 => (Array.update (ans, i, f (compute ans) i);
              Array.sub (ans, i))
      | v => v
in
  compute ans
end
```

The function which is to be memoised is the fibonacci function. This is presented as a "lifted" variant of its usual (exponential running time) version. The lifted function is called `mf`.

```
fun mf fib 0 = 1
  | mf fib 1 = 1
  | mf fib n = fib (n - 1) + fib (n - 2)
```

The memoised version of fibonacci is then simply `memo mf`.

8.8 Input/output

The final imperative features of Standard ML which we will present are the facilities for imperative input and output which are available in the language.

Pre-defined streams are `TextIO.stdIn` of type `TextIO.instream` and `TextIO.stdOut` of type `TextIO.outstream`. A new input stream can be created by using the function `TextIO.openIn` of type `string -> TextIO.instream`. A new output stream can be created by using the `TextIO.openOut` function of type `string -> TextIO.outstream`. There are `TextIO.closeIn` and `TextIO.closeOut` functions as well.

The result of attempting to open a file which is not present is an exceptional case and raises the exception `Io`, which carries a record describing the nature of the I/O failure. This exception may be handled and alternative action taken.

The functions for text I/O are the following.

```
      TextIO.input : TextIO.instream → string
     TextIO.inputN : TextIO.instream * int → string
  TextIO.lookahead : TextIO.instream → char option
TextIO.endOfStream : TextIO.instream → bool
     TextIO.output : TextIO.outstream * string → unit
```

A familiar C programming metaphor for processing files may be easily implemented in Standard ML. The function below simulates the behaviour of the UNIX `cat` command.

```
fun cat s =
    let val f = TextIO.openIn s and c = ref ""
    in
        while (c := TextIO.inputN (f, 1) ; !c <> "") do
            TextIO.output (TextIO.stdOut, !c);
        TextIO.closeIn f
    end
```

This function simulates the behaviour of the UNIX `strings` command, that is, it reads in a binary file and prints out those strings of printable characters which have length four or more.

```
fun strings s =
  let
    local
      val is = BinIO.openIn s
    in
      val binfile = BinIO.inputAll is
      val _= BinIO.closeIn is
    end
    val ws = String.str o Char.chr o Word8.toInt
    val fold = Word8Vector.foldr (fn (w, s) => ws w ^ s) ""
    val tokenise = String.tokens (Bool.not o Char.isPrint)
    val select = List.filter (fn s => String.size s >= 4)
  in
    (select o tokenise o fold) binfile
  end
```

We can present another C programming metaphor: a pre-processor which includes files as specified by a `#include` directive. It searches for the include files in one of a list of directories, handling possible exceptions and trying the next directory in its turn. The implementation of the function is in Figure 8.1.

Finally we show that we can combine text input and binary output by implementing a text-to-binary file translator which decodes a Base 64 encoded file. The Base 64 standard is the one which is used by for Internet mail in order to safeguard data from unintentional corruption. It operates by encoding three eight-bits characters using four six-bits ones. These six bits can be mapped onto the uppercase letters, the lowercase letters, the digits and the symbols plus and divide in that order, from 0 to 63. The Base 64 translator is presented in Figure 8.2 and uses auxiliary functions `charToWord` and `wordListToVector` together with infixed versions of the functions `Word.<<`, `Word.>>`, `Word.orb` and `Word.andb`.

Exercise 8.8.1 *The* `base64decode` *functions uses masks to select out the middle and low bytes in a word. Why could these not be obtained by shifting up sixteen bits and down eight and shifting down sixteen bits respectively?*

```
fun mlpp dir is os =
let val os = TextIO.openOut os
     fun findAndOpen [] f = TextIO.openIn f
        | findAndOpen (h::t) f = TextIO.openIn f
             handle _ => TextIO.openIn (h^f)
                  handle _ => findAndOpen t f
     fun inc f =
          let val is = findAndOpen dir f
          in
             while not (TextIO.endOfStream is) do
                let val line = TextIO.inputLine is
                     val len = String.size line
                in
                     if len > 8 andalso
                        String.substring (line, 0, 8) = "#include"
                     then inc (String.substring (line, 10, len - 12))
                     else TextIO.output (os, line)
                end;
             TextIO.closeIn is
          end
in
  inc is;
  TextIO.closeOut os
end;
```

Figure 8.1: The mlpp pre-processor

```
fun base64decode infile outfile =
let
  val is = TextIO.openIn infile
  val os = BinIO.openOut outfile
  fun decode #"/" = 0wx3F
    | decode #"+" = 0wx3E
    | decode c =
        if Char.isDigit c then charToWord c + 0wx04
        else if Char.isLower c then charToWord c - 0wx47
        else if Char.isUpper c then charToWord c - 0wx41
        else 0wx00
  fun convert (w0::w1::w2::w3::_) =
    let
      val w = (w0 << 0wx12) orb (w1 << 0wx0C) orb (w2 << 0wx06) orb w3
    in
      [w >> 0wx10, (w andb 0wx00FF00) >> 0wx08, w andb 0wx0000FF]
    end
  | convert _ = []
  fun next is = (convert o map decode o explode) (TextIO.inputN (is, 4))
in
  while not (TextIO.endOfStream is) do
    if TextIO.lookahead is = SOME #"\n"
    then (TextIO.input1 is; ())
    else (BinIO.output (os, wordListToVector (next is)));
  TextIO.closeIn is;
  BinIO.closeOut os
end
```

Figure 8.2: A Base64 translator

Chapter 9

Introducing Standard ML Modules

9.1 Signatures

One use of a signature is as a specification of a structure. That is, it may be used to describe a structure which is later to be provided. The signature states the types which will be declared in the structure and gives the type information for the values and functions in the structure.

Another use of a signature is as an interface which will hide some parts of the structure while allowing other parts to remain visible. This is achieved because it is possible for a structure to *match* a signature even though it declares more types and values than are required by the signature. These additional types and values are not visible.

Here is a simple signature which describes sets.

```
signature Set =
sig
   type ''a set
   val emptyset : ''a set
   val addset : ''a * ''a set → ''a set
   val memberset : ''a * ''a set → bool
end
```

Now we provide an implementation of this in the form of a structure. The signature acts as a *constraint* on the structure in the sense that it might hide identifiers or make a polymorphic function less polymorphic and perhaps even monomorphic. It might be said that a signature constraint is used for a structure in a similar way to the way that a type constraint is used for a value.

9.2 Structures

We will implement sets as boolean-valued functions which return `true` if applied to an element in the set and `false` otherwise.

```
structure Set :> Set =
struct
   type ''a set = ''a → bool
   fun emptyset _ = false
   fun addset (x, s) = fn e => e = x orelse s e
   fun memberset (x, s) = s x
end
```

This structure declaration has collected the type and the three functions under the umbrella name, `Set`. They are given *long identifiers* which are formed by prefixing the identifier with the name of the structure and a dot.

$$\begin{aligned}\mathsf{Set.emptyset} &: {}''\mathsf{a\ Set.set}\\ \mathsf{Set.addset} &: {}''\mathsf{a} * {}''\mathsf{a\ Set.set} \to {}''\mathsf{a\ Set.set}\\ \mathsf{Set.memberset} &: {}''\mathsf{a} * {}''\mathsf{a\ Set.set} \to \mathsf{bool}\end{aligned}$$

In order to understand the effect of the signature constraint above one should compare the results with the results obtained when the signature constraint [the "`:> Set`" part] is omitted. We then obtain a structure which has its *principal signature* and the types for the three functions are as given below.

$$\begin{aligned}\mathsf{Set.emptyset} &: {}'\mathsf{a} \to \mathsf{bool}\\ \mathsf{Set.addset} &: {}''\mathsf{a} * ({}''\mathsf{a} \to \mathsf{bool}) \to {}''\mathsf{a} \to \mathsf{bool}\\ \mathsf{Set.memberset} &: {}'\mathsf{a} * ({}'\mathsf{a} \to {}'\mathsf{b}) \to {}'\mathsf{b}\end{aligned}$$

These types seem to provide much less information about the intended use of the functions than do the type constraints imposed by using the signature `Set`. In particular it seems somewhat hard to understand how the `Set.memberset` function should be used.

There are other candidate signatures for the `Set` structure which lie 'between' the principal signature and the `Set` signature. One of these is given below.

```
signature Set =
sig
   type 'a set
   val emptyset : 'a set
   val addset : ''a * ''a set -> ''a set
   val memberset : 'a * 'a set -> bool
end
```

This does not seem to be better than the previous `Set` signature because it fails to require equality types throughout. In so doing it rules out implementations of the `Set` structure which are allowed by the previous signature and all but forces the use of functions to represent sets.

Exercise 9.2.1 *(Addicts only.) Provide a* `Set` *structure which matches the signature given above but stores the set elements in a list.*

9.3 Representation independence and equality

We will now consider replacing the implementation of the `Set` structure which uses functions by one which uses lists as the underlying concrete representation for the set. For convenience we will assume that we already have a structure containing utilities for lists such as a membership predicate.

```
structure Set :> Set =
struct
   type ''a set = ''a list
   val emptyset = []
   val addset = op ::
   val memberset = ListUtils.member
end
```

We might feel fearful of making this change because—unlike functions over equality types—lists of values from equality types can themselves be tested for equality. Equality on lists is not the same as equality on sets and so we might fear that this implementation of `Set` would have the disadvantage that it allows sets to be tested for equality, giving inappropriate answers. Such fears are misplaced. The equality on the type `''a set` is hidden by the use of the `Set` signature. Tight lipped, the signature refers to `''a set` as a *`type`*, with no indication that it admits equality. Thus we see that signatures are to be interpreted literally and not supplemented by other information which is obtained from peeking inside the structure to see how types have been defined. The terminology for this in Standard ML is that signatures which are attached using a coercion '`:>`' are *opaque*, not *transparent*.

The question of whether or not signatures should be opaque is typical of many questions which arise in programming language design. The exchange being made is between the purity of a strict interpretation of an abstraction and the convenience of software re-use. Transparent signatures may save a significant amount of work since the software developer is able to exploit some knowledge about how a structure has been implemented. This saving may have to be paid for later when a change of data representation causes modifications to be made to other structures.

Exercise 9.3.1 *Reimplement the α* `susp` *datatype from page 49 as a structure* `Susp`*. You will notice that in the body of the structure neither the* `local .. in .. end` *nor the* `abstype .. with .. end` *are necessary. The effects of hiding the α* `hitchcock` *datatype and hiding the equality on α* `susp` *values can both be achieved through the use of a signature.*

Bibliography

[Ada93] Stephen Adams. Functional Pearls: Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[Bar96] J. Barnes. *Programming in Ada 95*. Addison-Wesley, 1996.

[Ber91] Dave Berry. The Edinburgh SML Library. Technical Report ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, University of Edinburgh, April 1991.

[Car96] Luca Cardelli. Type systems, 1996. *CRC Handbook of Computer Science and Engineering*.

[GGM91] Carl A. Gunter, Elsa L. Gunter, and David B. MacQueen. An abstract interpretation for ML equality kinds. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 112–130. Springer-Verlag, September 1991.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[Hin69] J.R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, 146:29–60, 1969.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[Jon92] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[Kah96] Stefan Kahrs. Limits of ML-definability. In *Proceedings of Eighth International Symposium on Programming Languages, Implementations, Logics, and Programs*, September 1996.

[Knu89] Donald Knuth. The errors of TeX. *Software—Practice and Experience*, 19:607–685, 1989.

[KTU94] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, March 1994.

[MCP93] Colin Myers, Chris Clack, and Ellen Poon. *Programming with Standard ML*. Prentice-Hall, 1993.

[Mil78] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[MNV73] Zohar Manna, Stephen Ness, and Jean Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM*, 16(8):491–502, August 1973.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.

[Pau96] Larry Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

[PJL92] Simon L. Peyton-Jones and David Lester. *Implementing Functional Languages: A Tutorial*. International Series in Computer Science. Prentice-Hall, 1992.

[Rea89] Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[Sok91] S. Sokołowski. *Applicative High-Order Programming: The Standard ML Perspective*. Chapman and Hall, 1991.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988. Also published as Technical Report CST-52-88, Department of Computer Science.

[Tof89] Mads Tofte. Four lectures on Standard ML. Technical Report ECS-LFCS-89-73, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[Tur82] D. A. Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and its Applications*. Cambridge University Press, 1982.

[Ull98] J. D. Ullman. *Elements of ML Programming (ML97 edition)*. Prentice-Hall, 1998.

[WH86] Jim Welsh and Atholl Hay. *A model implementation of standard Pascal*. International Series in Computer Science. Prentice-Hall, 1986.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

Index

- :=, 49, 50, 52, 54
- :>, 57–59
- ”a, 31, 33, 45, 57–59
- ’a, 21, 58
- ’b, 21, 58
- ’c, 21
- 0w255, 6
- 0wxff, 6
- 0xff, 5
- 255, 5
- ::=, 49
- @, *see* List.@
- Array, 52
  - Array.array, 52, 53
  - Array.fromList, 52, 53
  - Array.sub, 52, 53
  - Array.update, 52, 53
- BigInt.int, 25
- BinIO
  - BinIO.closeIn, 54
  - BinIO.closeOut, 56
  - BinIO.inputAll, 54
  - BinIO.openIn, 54
  - BinIO.openOut, 56
  - BinIO.output, 56
- Bool.fromString, 10
- Bool.not, 10, 13
- Bool.toString, 10
- Bool, 10
  - Bool.not, 54
- Byte.byteToChar, 10
- Byte.charToByte, 10
- Byte, 10
- Char.chr, 10
- Char.contains, 10
- Char.isAlphaNum, 10
- Char.isAlpha, 10
- Char.isAscii, 10
- Char.isDigit, 10
- Char.isGraph, 10
- Char.isHexDigit, 10
- Char.isLower, 10
- Char.isPrint, 10
- Char.isSpace, 10
- Char.isUpper, 10
- Char.notContains, 10
- Char.ord, 10
- Char.pred, 10
- Char.succ, 10
- Char.toLower, 10
- Char.toUpper, 10
- Char, 10, 12
  - Char.chr, 54
  - Char.isDigit, 56
  - Char.isLower, 56
  - Char.isPrint, 54
  - Char.isUpper, 56
- Empty, 27, 28
- FIX’, 40
- FIX, 39, 40
- Int.abs, 11
- Int.fmt, 11
- Int.fromString, 11
- Int.maxInt, 11
- Int.max, 11
- Int.minInt, 11
- Int.min, 11
- Int.quot, 11
- Int.rem, 11
- Int.toString, 11
- Int, 11, 12
  - Int.max, 35
  - Int.min, 35
- Io, 53
- ListPair, 37
  - ListPair.all, 37
  - ListPair.exists, 37
  - ListPair.foldl, 37
  - ListPair.foldr, 37
  - ListPair.map, 37
  - ListPair.unzip, 37
  - ListPair.zip, 37
- ListUtils, 58
- List, 37
  - List.@, 37
  - List.Empty, 37
  - List.all, 37
  - List.concat, 37
  - List.drop, 37
  - List.exists, 37

- List.filter, 37, 54
- List.find, 37
- List.foldl, 37
- List.foldr, 37
- List.hd, 37
- List.last, 37
- List.length, 37
- List.mapPartial, 37
- List.nth, 36, 37
- List.null, 37
- List.partition, 37
- List.revAppend, 37
- List.rev, 37
- List.tabulate, 37
- List.take, 37
- List.tl, 37

Math
- Math.atan, 31
- Math.cos, 31
- Math.sin, 31
- Math.tan, 31

NONE, 10, 11, 28, 34
Overflow, 27, 31, 32
Real.ceil, 11
Real.floor, 7, 11
Real.fmt, 11
Real.fromInt, 7, 11
Real.round, 11
Real.trunc, 11
Real, 11, 12
Retrieve, 31
SOME, 10, 11, 28, 34
Set.addset, 58
Set.emptyset, 58
Set.memberset, 58
Set.set, 58
Set, 45, 46, 57–59
String.concat, 11
String.extract, 11
String.fields, 12
String.substring, 11
String.sub, 9, 11, 20
String.tokens, 12
String.translate, 12
StringCvt.BIN, 11
StringCvt.DEC, 11
StringCvt.FIX, 11
StringCvt.GEN, 11
StringCvt.HEX, 11
StringCvt.OCT, 11
StringCvt.SCI, 11
StringCvt.padLeft, 12
StringCvt.padRight, 12
StringCvt, 11, 12
String, 9, 11
- String.size, 54, 55
- String.str, 54
- String.substring, 55
- String.tokens, 54

Subscript, 27, 31, 32, 36, 52
Susp, 59
TextIO
- TextIO.closeIn, 53–56
- TextIO.closeOut, 53, 55
- TextIO.endOfStream, 54–56
- TextIO.input1, 56
- TextIO.inputLine, 55
- TextIO.inputN, 54, 56
- TextIO.input, 54
- TextIO.instream, 53, 54
- TextIO.lookahead, 54, 56
- TextIO.openIn, 53–56
- TextIO.openOut, 53, 55
- TextIO.output, 54, 55
- TextIO.outstream, 53, 54
- TextIO.stdIn, 53
- TextIO.stdOut, 53, 54

Vector, 36, 37
- Vector.concat, 37
- Vector.extract, 36
- Vector.foldli, 37
- Vector.foldl, 37
- Vector.foldri, 37
- Vector.foldr, 37
- Vector.fromList, 36
- Vector.length, 37
- Vector.sub, 36
- Vector.tabulate, 37

Word.word, 12
Word8.word, 12
Word8Vector
- Word8Vector.foldr, 54

Word8, 12
- Word8.toInt, 54

Word, 12
- Word.andb, 54
- Word.orb, 54
- Word.<<, 54
- Word.>>, 54

absorb, 25, 26
abs, 6, 14
addset, 44, 46, 57, 58
addtwo, 15
age, 20
amber, 21
andb, 12
app, 26
a, 53

base64decode, 54
blue, 21
bool, 5, 21, 22, 54
b, 53
cat, 54
charToWord, 54
char, 5, 54
chr, 5
colour, 21
compose, 16, 17, 24
concat, 34, 52
cond, 39
cons, 41, 42
corpse, 49
create, 26
curry, 16, 17
date_of_birth, 20
day', 36
day, 6, 36
defn.of.delay, 40
delayed, 40
delay, 40, 41, 49
depalma, 49
divide, 23
div, 11
dropwhile, 32
drop, 32
emptyset, 44, 46, 48, 57, 58
equal, 45, 46
eval_int_exp, 23
eval_int_factor, 23
eval_int_term, 23
even, 14, 23
exn, 27
explode, 5
e, 43
facbody, 39
fac, 15, 39, 50
false, 5, 46
fastrev, 52
fib, 53
filter, 32, 33
finished, 14
floor, 7
foldl, 34
foldr, 34
force, 40–42, 49
from, 42
fst, 23
green, 21
hd_tst, 28
hd, 28
head, 41
heterogeneous, 22
hitchcock, 49, 59
hundreds, 42
ifac, 50
implode, 5
initials, 20, 24
initial, 20
inorder, 35
insert, 33, 34
ins, 44
int_const, 23
int_exp, 23
int_factor, 23
int_term, 23
inttree, 21
int, 5, 23
is_identity, 14
iter', 17
iter, 15–17
last_tst, 28
last, 28
lcons, 42
length', 34
length, 22, 28, 30, 32, 34
listid, 34
listrev, 34
list, 22, 35
loop, 24
mapPartial, 34
map_f, 34
mapbody, 39
map, 33, 34
maxdepth, 35
mcguffin, 49
memberset, 44, 46, 57, 58
member, 30, 31, 58
memo, 53
mf, 53
mindepth, 35
minus, 23
mk_set, 46
modulo, 23
mod, 11
myset, 46
nats, 42
next, 42
nil, 22
nodes, 35
notb, 12
nth, 31
null_eq, 31
null, 44
odd, 23
oneszeroes, 42
ones, 42

option, 10, 11, 28, 54
orb, 12
ordered_set, 45, 46
ordered, 25
ord, 5
o, 16
pair, 23
paren', 23
paren, 23
perm, 33
person, 20
plus, 23
postorder, 35
prefix, 33
preorder, 35
radix', 9
radix, 9
rap, 51
real, 5, 7
rec, 39
reduce, 14, 15, 17
red, 21
ref, 47, 50, 51
retrieve, 31
revAppend, 29
rev, 29, 30, 52
same, 13
set, 21, 44, 57–59
size, 5
snd, 23
sort', 34
sort, 33, 34
square, 25
sq, 10, 14, 15
string, 5, 53, 54
str, 5
subset, 45
sub, 9
succ, 6, 16
sum", 9
sum', 8, 9, 14, 15
sum, 8
surname, 20
susp, 49, 59
tail, 41
takewhile, 32, 39
take, 32
tens, 42
tentimes, 42
tester, 28
times, 23
tl_tst, 28
tl, 28
traffic_light, 21
tree, 22, 35
true, 5, 46
ttree, 22
twice, 15
uncurry, 16, 17
unit, 40, 49, 54
val, 39
vector, 36
wordListToVector, 54
word, 5
wrong_exp, 13
wrong_pat, 13
xorb, 12
zc, 7
zeller, 7, 9
α, 49

Anderson, Stuart, 26, 33

Base 64, 54
Bosworth, Richard, 3
byte, 5

call-by-name, 38
call-by-need, 39
call-by-value, 38
case-sensitive language, 6
Church Rosser Theorems
  First Theorem, 38
  Second Theorem, 38
Clack, Chris, 3
composition, 16
  in diagrammatic order, 16
  in functional order, 16
conditional expression, 17
  short-circuit, 17
constructors, 13, 21
  nullary, 21
curried functions, 15
Curry, Haskell B., 15

dead code, 13
default overloading, 30
dereferencing, 47
derived forms, 16, 20, 22
destructors, 27
Dijkstra, Edsger W., 42
Duba, Bruce, 22

equality types, 31
exception, 27
expressions
  expansive, 51
  non-expansive, 51
extensional equality, 31

factorial function, 15
first-fit pattern matching, 13
for loops, 37
function
  fibonacci, 53
  integer maximum, 35
functions
  composition of, 16
  curried, 15
  factorial, 15
  higher-order, 14
  homogeneous, 23
  idempotent, 15
  identity, 14–16
  polymorphic, 23
  successor, 6, 15

handled, 28
Harper, Robert, 3
Henglein, Fritz, 26
higher-order function, 14
Hindley, Roger, 19
Hoare, C.A.R., 33
Hughes, John, 2

idempotent functions, 15
identity function, 14–16
induction, 8
  structural, 27
intensional equality, 31
interchange law, 29
involution, 30

Kahrs, Stefan, 25

leaf, 35
long identifier, 9

MacQueen, Dave, 3
masks, 54
memoisation, 53
Michaelson, Greg, 3
Milner, Robin, 3, 19, 38
Mitchell, Kevin, 53
ML keywords
  `abstype`, 44, 45, 48, 49, 59
  `andalso`, 17, 18, 33, 45, 48, 55
  `and`, 23, 43, 50, 52, 54
  `as`, 33, 45, 48, 49
  `case`, 17, 34, 53
  `datatype`, 21–23, 28, 41, 44, 49
  `do`, 50, 52, 54–56
  `else`, 17, 31–33, 39, 43, 45, 47, 48, 55, 56
  `end`, 7, 9, 14, 15, 19, 33, 34, 44, 45, 48–50, 52–59
  `exception`, 27, 31, 48
  `fn`, 6–10, 13–16, 19–26, 34, 39–41, 50–52, 54, 57
  `fun`, 16, 17, 20, 22–26, 28–36, 39–57
  `handle`, 28, 31, 32, 36, 55
  `if`, 17, 31–33, 39, 43, 45, 47, 48, 55, 56
  `infixr`, 22, 29
  `infix`, 22
  `in`, 7, 9, 14, 15, 19, 33, 34, 44, 45, 48–50, 52–56, 59
  `let`, 9, 14, 15, 17, 19, 20, 33, 34, 45, 48–50, 52–56
  `local`, 7, 9, 17, 44, 45, 49, 54, 59
  `of`, 17, 21–23, 28, 34, 41, 44, 45, 48, 49, 53
  `op`, 15–17, 24, 34, 45, 46, 48, 52, 58
  `orelse`, 17, 18, 30, 44, 45, 48, 57
  `raise`, 28, 31, 32, 48
  `rec`, 8–10, 14–16, 21, 39
  `then`, 17, 31–33, 39, 43, 45, 47, 48, 55, 56
  `type`, 20–22, 40, 44, 57–59
  `val`, 6–10, 13–16, 19, 20, 22–24, 26, 39, 40, 42–50, 52–58
  `while`, 50, 52, 54–56
  `with`, 44, 45, 48, 49, 59
ML library units
  `Array`
    `Array.array`, 52, 53
    `Array.fromList`, 52, 53
    `Array.sub`, 52, 53
    `Array.update`, 52, 53
  `BinIO`
    `BinIO.closeIn`, 54
    `BinIO.closeOut`, 56
    `BinIO.inputAll`, 54
    `BinIO.openIn`, 54
    `BinIO.openOut`, 56
    `BinIO.output`, 56
  `Bool`
    `Bool.not`, 54
  `Char`
    `Char.chr`, 54
    `Char.isDigit`, 56
    `Char.isLower`, 56
    `Char.isPrint`, 54
    `Char.isUpper`, 56
  `Int`
    `Int.max`, 35
    `Int.min`, 35
  `ListPair`
    `ListPair.all`, 37
    `ListPair.exists`, 37
    `ListPair.foldl`, 37

ListPair.foldr, 37
ListPair.map, 37
ListPair.unzip, 37
ListPair.zip, 37
List
List.Empty, 37
List.all, 37
List.concat, 37
List.drop, 37
List.exists, 37
List.filter, 37, 54
List.find, 37
List.foldl, 37
List.foldr, 37
List.hd, 37
List.last, 37
List.length, 37
List.mapPartial, 37
List.nth, 36, 37
List.null, 37
List.partition, 37
List.revAppend, 37
List.rev, 37
List.tabulate, 37
List.take, 37
List.tl, 37
Math
Math.atan, 31
Math.cos, 31
Math.sin, 31
Math.tan, 31
String
String.size, 54, 55
String.str, 54
String.substring, 55
String.tokens, 54
TextIO
TextIO.closeIn, 53–56
TextIO.closeOut, 53, 55
TextIO.endOfStream, 54–56
TextIO.input1, 56
TextIO.inputLine, 55
TextIO.inputN, 54, 56
TextIO.input, 54
TextIO.instream, 53, 54
TextIO.lookahead, 54, 56
TextIO.openIn, 53–56
TextIO.openOut, 53, 55
TextIO.output, 54, 55
TextIO.outstream, 53, 54
TextIO.stdIn, 53
TextIO.stdOut, 53, 54
Vector
Vector.concat, 37
Vector.extract, 36
Vector.foldli, 37
Vector.foldl, 37
Vector.foldri, 37
Vector.foldr, 37
Vector.fromList, 36
Vector.length, 37
Vector.sub, 36
Vector.tabulate, 37
Word8Vector
Word8Vector.foldr, 54
Word8
Word8.toInt, 54
Word
Word.andb, 54
Word.orb, 54
Word.<<, 54
Word.>>, 54
ML modules keywords
signature, 57, 58
sig, 57, 58
structure, 57, 58
struct, 57, 58
Myers, Colin, 3

nullary constructors, 21

operators
overloaded, 24
overloading, 24

path, 35
pattern matching, 13
pattern matching, 6
wild card, 13
Paulson, Larry, 3
perfectly balanced, 35
polymorphic, 23
Poon, Ellen, 3

raised, 28
Reade, Chris, 3
records, 20
references, 47

singleton, 32
SML basis, 5
SML library, 5
Sokołowski, Stefan, 3
sorting
insertion sort, 32
Standard ML library, 10
statically typed, 19
strict, 17
strongly typed, 19

structures in the Standard ML library, 10
subtyping, 20
successor function, 6, 15
syntactic sugar, 16

testing, 8
Tofte, Mads, 3, 25, 51
tokeniser, 12
traversal
    inorder, 35
    postorder, 35
    preorder, 35
traversal strategies, 35
type coercions, 6
type inference, 19
type variable, 20

Ullman, Jeffrey, 3

value polymorphism, 51
vector, 36
vector slice, 36

Wikström, Åke, 3
word, 5
Wright, Andrew, 51

Zeller's congruence, 7