

3

Lists

In a public lecture, C. A. R. Hoare (1989a) described his algorithm for finding the i th smallest integer in a collection. This algorithm is subtle, but Hoare described it with admirable clarity as a game of solitaire. Each playing card carried an integer. Moving cards from pile to pile by simple rules, the required integer could quickly be found.

Then Hoare changed the rules of the game. Each card occupied a fixed position, and could only be moved if exchanged with another card. This described the algorithm in terms of arrays. Arrays have great efficiency, but they also have a cost. They probably defeated much of the audience, as they defeat experienced programmers. Mills and Linger (1986) claim that programmers become more productive when arrays are restricted to stacks, queues, etc., without subscripting.

Functional programmers often process collections of items using lists. Like Hoare's stacks of cards, lists allow items to be dealt with one at a time, with great clarity. Lists are easy to understand mathematically, and turn out to be more efficient than commonly thought.

Chapter outline

This chapter describes how to program with lists in Standard ML. It presents several examples that would normally involve arrays, such as matrix operations and sorting.

The chapter contains the following sections:

Introduction to lists. The notion of list is introduced. Standard ML operates on lists using pattern-matching.

Some fundamental list functions. A family of functions is presented. These are instructive examples of list programming, and are indispensable when tackling harder problems.

Applications of lists. Some increasingly complicated examples illustrate the variety of problems that can be solved using lists.

The equality test in polymorphic functions. Equality polymorphism is intro-

duced and demonstrated with many examples. These include a useful collection of functions on finite sets.

Sorting: A case study. Procedural programming and functional programming are compared in efficiency. In one experiment, a procedural program runs only slightly faster than a much clearer functional program.

Polynomial arithmetic. Computers can solve algebraic problems. Lists are used to add, multiply and divide polynomials in symbolic form.

Introduction to lists

A *list* is a finite sequence of elements. Typical lists are `[3, 5, 9]` and `["fair", "Ophelia"]`. The empty list, `[]`, has no elements. The order of elements is significant, and elements may appear more than once. For instance, the following lists are all different:

```
[3, 4]    [4, 3]    [3, 4, 3]    [3, 3, 4]
```

The elements of a list may have any type, including tuples and even other lists. Every element of a list must have the same type. Suppose this type is τ ; the type of the list is then τ *list*. Thus

```
[(1, "One"), (2, "Two"), (3, "Three")] : (int*string) list
[ [3.1], [], [5.7, ~0.6] ]           : (real list) list
```

The empty list, `[]`, has the polymorphic type α *list*. It can be regarded as having any type of elements.

Observe that the type operator *list* has a postfix syntax. It binds more tightly than \times and \rightarrow . So $int \times string\ list$ is the same type as $int \times (string\ list)$, not $(int \times string)\ list$. Also, $int\ list\ list$ is the same type as $(int\ list)\ list$.

3.1 Building a list

Every list is constructed by just two primitives: the constant *nil* and the infix operator `::`, pronounced ‘cons’ for ‘construct.’

- *nil* is a synonym for the empty list, `[]`.
- The operator `::` makes a list by putting an element in front of an existing list.

Every list is either *nil*, if empty, or has the form $x :: l$ where x is its **head** and l its **tail**. The tail is itself a list. The list operations are not symmetric: the first element of a list is much more easily reached than the last.

If l is the list $[x_1, \dots, x_n]$ and x is a value of correct type then $x :: l$ is the list $[x, x_1, \dots, x_n]$. Making the new list does not affect the value of l . The list

[3, 5, 9] is constructed as follows:

$$\begin{aligned} \text{nil} &= [] \\ 9 :: [] &= [9] \\ 5 :: [9] &= [5, 9] \\ 3 :: [5, 9] &= [3, 5, 9] \end{aligned}$$

Observe that the elements are taken in reverse order. The list [3, 5, 9] can be written in many ways, such as $3 :: (5 :: (9 :: \text{nil}))$, or $3 :: (5 :: [9])$, or $3 :: [5, 9]$. To save writing parentheses, the infix operator ‘cons’ groups to the right. The notation $[x_1, x_2, \dots, x_n]$ stands for $x_1 :: x_2 :: \dots :: x_n :: \text{nil}$. The elements may be given by expressions; a list of the values of various real expressions is

```
[ Math.sin 0.5, Math.cos 0.5, Math.exp 0.5 ];
> [0.479425539, 0.877582562, 1.64872127] : real list
```

List notation makes a list with a fixed number of elements. Consider how to build the list of integers from m to n :

$$[m, m + 1, \dots, n]$$

First compare m and n . If $m > n$ then there are no numbers between m and n ; the list is empty. Otherwise the head of the list is m and the tail is $[m+1, \dots, n]$. Constructing the tail recursively, the result is obtained by

$$m :: [m + 1, \dots, n].$$

This process corresponds to a simple ML function:

```
fun upto (m, n) =
  if m > n then [] else m :: upto(m+1, n);
> val upto = fn : int * int -> int list
upto(2, 5);
> [2, 3, 4, 5] : int list
```



Lists in other languages. Weakly typed languages like Lisp and Prolog represent lists by pairing, as in $(3, (5, (9, \text{"nil"})))$. Here “nil” is some end marker and the list is [3, 5, 9]. This representation of lists does not work in ML because the type of a ‘list’ would depend on the number of elements. What type could *upto* have?

ML’s syntax for lists differs subtly from Prolog’s. In Prolog, $[5 | [6]]$ is the same list as $[5, 6]$. In ML, $[5 :: [6]]$ is the same list as $[[5, 6]]$.

3.2 Operating on a list

Lists, like tuples, are structured values. In ML, a function on tuples can be written with a pattern for its argument, showing its structure and naming the components. Functions over lists can be written similarly. For example,

```
fun prodof3 [i,j,k] : int = i*j*k;
```

declares a function to take the product of a list of numbers — but only if there are exactly three of them!

List operations are usually defined by recursion, treating several cases. What is the product of a list of integers?

- If the list is empty, the product is 1 (by convention).
- If the list is non-empty, the product is the head times the product of the tail.

It can be expressed in ML like this:

```
fun prod [] = 1
  | prod (n::ns) = n * (prod ns);
> val prod = fn : int list -> int
```

The function consists of two clauses separated by a vertical bar (`|`). Each clause treats one argument pattern. There may be several clauses and complex patterns, provided the types agree. Since the patterns involve lists, and the result can be the integer 1, ML infers that `prod` maps a list of integers to an integer.

```
prod [2,3,5];
> 30 : int
```

Empty versus non-empty is the commonest sort of case analysis for lists. Finding the maximum of a list of integers requires something different, for the empty list has no maximum. The two cases are

- The maximum of the one-element list `[m]` is `m`.
- To find the maximum of a list with two or more elements `[m, n, ...]`, remove the smaller of `m` or `n` and find the maximum of the remaining numbers.

This gives the ML function

```
fun maxl [m] : int = m
  | maxl (m::n::ns) = if m>n then maxl (m::ns)
                     else maxl (n::ns);
> ***Warning: Patterns not exhaustive
> val maxl = fn : int list -> int
```

Note the warning message: ML detects that *maxl* is undefined for the empty list. Also, observe how the pattern $m :: n :: ns$ describes a list of the form $[m, n, \dots]$. The smaller element is dropped in the recursive call.

The function works — except for the empty list.

```
maxl [ ~4, 0, ~12];
> 0 : int
maxl [];
> Exception: Match
```

An **exception**, for the time being, can be regarded as a run-time error. The function *maxl* has been applied to an argument for which it is undefined. Normally exceptions abort execution. They can be trapped, as we shall see in the next chapter.

Intermediate lists. Lists are sometimes generated and consumed within a computation. For instance, the factorial function has a clever definition using *prod* and *upto*:

```
fun factl (n) = prod (upto (1, n));
> val factl = fn : int -> int
factl 7;
> 5040 : int
```

This declaration is concise and clear, avoiding explicit recursion. The cost of building the list $[1, 2, \dots, n]$ may not matter. However, functional programming should facilitate reasoning about programs. This does not happen here. The trivial law

$$\text{factl}(m + 1) = (m + 1) \times \text{factl}(m)$$

has no obvious proof. Opening up its definition, we get

$$\text{factl}(m + 1) = \text{prod}(\text{upto}(1, m + 1)) = ?$$

The next step is unclear because the recursion in *upto* follows its first argument, not the second. The honest recursive definition of factorial seems better.

Strings and lists. Lists are important in string processing. Most functional languages provide a type of single characters, regarding strings as lists of characters. With the new standard library, ML has acquired a character type — but it does not regard strings as lists. The built-in function *explode* converts a string to a list of characters. The function *implode* performs the inverse operation, joining a list of characters to form a string.

```

explode "Banquo";
> ["#B", "#a", "#n", "#q", "#u", "#o"] : char list
implode it;
> "Banquo" : string

```

Similarly, the function *concat* joins a list of strings to form a string.

Some fundamental list functions

Given a list we can find its length, select the *n*th element, take a prefix or suffix, or reverse the order of its elements. Given two lists we can append one to the other, or, if they have equal length, pair corresponding elements. The functions declared in this section are indispensable, and will be taken for granted in the rest of the book. All of these functions are polymorphic.

Efficiency becomes a central concern here. For some functions, a naïve recursive definition is less efficient than an iterative version. For others, an iterative style impairs both readability and efficiency.

3.3 Testing lists and taking them apart

The three basic functions on lists are *null*, *hd* and *tl*.

The function null. This function tests whether a list is empty:

```

fun null [] = true
  | null (_:_) = false;
> val null = fn : 'a list -> bool

```

The function is polymorphic: testing whether a list is empty does not examine its elements. The underscores (*_*) in the second pattern take the place of components whose values are not needed in the clause. These underscores, called *wildcard* patterns, save us from inventing names for such components.

The function hd. This function returns the head (the first element) of a non-empty list:

```

fun hd (x::_) = x;
> ***Warning: Patterns not exhaustive
> val hd = fn : 'a list -> 'a

```

This pattern has a wildcard for the tail, while the head is called *x*. Since there is no pattern for the empty list, ML prints a warning. It is a partial function like *maxl*.

Here we have a list of lists. Its head is a list and the head of that is an integer. Each use of *hd* removes one level of brackets.

```

hd([[1,2], [3]], [[4]]);
> [[1, 2], [3]] : (int list) list
hd it;
> [1, 2] : int list
hd it;
> 1 : int

```

What if we type `hd it;` once more?

The function tl. This returns the tail of a non-empty list. The tail, remember, is the list consisting of the all elements but the first.

```

fun tl (_::xs) = xs;
> ***Warning: Patterns not exhaustive
> val tl = fn : 'a list -> 'a list

```

Like `hd`, this is a partial function. Its result is always another list:

```

tl ["Out", "damned", "spot!"];
> ["damned", "spot!"] : string list
tl it;
> ["spot!"] : string list
tl it;
> [] : string list
tl it;
> Exception: Match

```

Attempting to take the tail of the empty list is an error.

Through `null`, `hd` and `tl`, all other list functions can be written without pattern-matching. The product of a list of integers can be computed like this:

```

fun prod ns = if null ns then 1
              else (hd ns) * (prod (tl ns));

```

If you prefer this version of `prod`, you might as well give up ML for Lisp. For added clarity, Lisp primitives have names like `CAR` and `CDR`. Normal people find pattern-matching more readable than `hd` and `tl`. A good ML compiler analyses the set of patterns to generate the best code for the function. More importantly, the compiler prints a warning if the patterns do not cover all possible arguments of the function.

Exercise 3.1 Write a version of `maxl` using `null`, `hd` and `tl`, instead of pattern-matching.

Exercise 3.2 Write a function to return the last element of a list.

3.4 List processing by numbers

We now declare the functions *length*, *take* and *drop*, which behave as follows:

$$l = \underbrace{[x_0, \dots, x_{i-1}]}_{\text{take}(l, i)} \underbrace{[x_i, \dots, x_{n-1}]}_{\text{drop}(l, i)} \quad \text{length}(l) = n$$

The function *length*. The length of a list can be computed by a naïve recursion:

```
fun nlength [] = 0
  | nlength (x::xs) = 1 + nlength xs;
> val nlength = fn : 'a list -> int
```

Its type, $\alpha \text{ list} \rightarrow \text{int}$, permits *nlength* to be applied to a list regardless of the type of its elements. Let us try it on a list of lists:

```
nlength[[1,2,3], [4,5,6]];
> 2 : int
```

Did you think the answer would be 6?

Although correct, *nlength* is intolerably wasteful for long lists:

$$\begin{aligned} \text{nlength}[1, 2, 3, \dots, 10000] &\Rightarrow 1 + \text{nlength}[2, 3, \dots, 10000] \\ &\Rightarrow 1 + (1 + \text{nlength}[3, \dots, 10000]) \\ &\vdots \\ &\Rightarrow 1 + (1 + 9998) \\ &\Rightarrow 1 + 9999 \Rightarrow 10000 \end{aligned}$$

The ones pile up, wasting space proportional to the length of the list, and could easily cause execution to abort. Much better is an iterative version of the function that accumulates the count in another argument:

```
local
  fun addlen (n, []) = n
    | addlen (n, x::l) = addlen (n+1, l)
in
  fun length l = addlen (0, l)
end;
> val length = fn : 'a list -> int
length (explode"Throw physic to the dogs!");
> 25 : int
```

The function *addlen* adds the length of a list to another number, initially 0. Since *addlen* has no other purpose, it is declared `local` to *length*. It executes

as follows:

$$\begin{aligned} \text{addlen}(0, [1, 2, 3, \dots, 10000]) &\Rightarrow \text{addlen}(1, [2, 3, \dots, 10000]) \\ &\Rightarrow \text{addlen}(2, [3, \dots, 10000]) \\ &\vdots \\ &\Rightarrow \text{addlen}(10000, []) \Rightarrow 10000 \end{aligned}$$

The greatly improved efficiency compensates for the loss of readability.

The function take. Calling *take* (*l*, *i*) returns the list of the first *i* elements of *l*:

```
fun take ([], i)      = []
  | take (x::xs, i) = if i>0 then x::take(xs, i-1)
                      else [];
> val take = fn : 'a list * int -> 'a list
take (explode"Throw physic to the dogs!", 5);
> [#"T", #"h", #"r", #"o", #"w"] : char list
```

Here is a sample computation:

$$\begin{aligned} \text{take}([9, 8, 7, 6], 3) &\Rightarrow 9 :: \text{take}([8, 7, 6], 2) \\ &\Rightarrow 9 :: (8 :: \text{take}([7, 6], 1)) \\ &\Rightarrow 9 :: (8 :: (7 :: \text{take}([6], 0))) \\ &\Rightarrow 9 :: (8 :: (7 :: [])) \\ &\Rightarrow 9 :: (8 :: [7]) \\ &\Rightarrow 9 :: [8, 7] \\ &\Rightarrow [9, 8, 7] \end{aligned}$$

Observe that $9 :: (8 :: (7 :: []))$ above is an expression, not a value. Evaluating it constructs the list $[9, 8, 7]$. Allocating the necessary storage takes time, particularly if we consider its contribution to the cost of later garbage collections. Indeed, *take* probably spends most of its time building its result.

The recursive calls to *take* get deeper and deeper, like *nlength*, which we have recently deplored. Let us try to make an iterative version of *take* by accumulating the result in an argument:

```
fun rtake ([], _, taken) = taken
  | rtake (x::xs, i, taken) =
      if i>0 then rtake(xs, i-1, x::taken)
      else taken;
> val rtake = fn : 'a list * int * 'a list -> 'a list
```

The recursion is nice and shallow ...

```
rtake([9, 8, 7, 6], 3, []) ⇒ rtake([8, 7, 6], 2, [9])
                          ⇒ rtake([7, 6], 1, [8, 9])
                          ⇒ rtake([6], 0, [7, 8, 9])
                          ⇒ [7, 8, 9]
```

... but the output is reversed!

If a reversed output is acceptable, *rtake* is worth considering. However, the size of the recursion in *take* is tolerable compared with the size of the result. While *nlength* returns an integer, *take* returns a list. Building a list is slow, which is usually more important than the space temporarily consumed by deep recursion. Efficiency is a matter of getting the costs into proportion.

The function drop. The list *drop* (*l*, *i*) contains all but the first *i* elements of *l*:

```
fun drop ([], _) = []
  | drop (x::xs, i) = if i>0 then drop (xs, i-1)
                    else x::xs;
> val drop = fn : 'a list * int -> 'a list
```

Luckily, the obvious recursion is iterative.

```
take (["Never", "shall", "sun", "that", "morrow", "see!"], 3);
> ["Never", "shall", "sun"] : string list
drop (["Never", "shall", "sun", "that", "morrow", "see!"], 3);
> ["that", "morrow", "see!"] : string list
```

Exercise 3.3 What do *take* (*l*, *i*) and *drop* (*l*, *i*) return when *i* > *length*(*l*), and when *i* < 0? (The library versions of *take* and *drop* would raise exceptions.)

Exercise 3.4 Write a function *nth* (*l*, *n*) to return the *n*th element of *l* (where the head is element 0).

3.5 Append and reverse

The infix operator @, which appends one list to another, and *rev*, which reverses a list, are built-in functions. Their definitions deserve close attention.

The append operation. Append puts the elements of one list after those of another list:

$$[x_1, \dots, x_m] @ [y_1, \dots, y_n] = [x_1, \dots, x_m, y_1, \dots, y_n]$$

What sort of recursion accomplishes this? The traditional name *append* suggests that the action takes place at the end of a list, but lists are always built from the front. The following definition, in its essentials, dates to the early days of Lisp:

```
infixr 5 @;
fun ([] @ ys) = ys
  | ((x::xs) @ ys) = x :: (xs@ys);
> val @ = fn : 'a list * 'a list -> 'a list
```

Its type, $\alpha \text{ list} \times \alpha \text{ list} \rightarrow \alpha \text{ list}$, accepts any two lists with the same element type — say, lists of strings and lists of lists:

```
["Why", "sinks"] @ ["that", "cauldron?"];
> ["Why", "sinks", "that", "cauldron?"] : string list
[[2,4,6,8], [3,9]] @ [[5], [7]];
> [[2, 4, 6, 8], [3, 9], [5], [7]] : int list list
```

The computation of $[2, 4, 6] @ [8, 10]$ goes like this:

$$\begin{aligned} [2, 4, 6] @ [8, 10] &\Rightarrow 2 :: ([4, 6] @ [8, 10]) \\ &\Rightarrow 2 :: (4 :: ([6] @ [8, 10])) \\ &\Rightarrow 2 :: (4 :: (6 :: ([] @ [8, 10]))) \\ &\Rightarrow 2 :: (4 :: (6 :: [8, 10])) \\ &\Rightarrow 2 :: (4 :: [6, 8, 10]) \\ &\Rightarrow 2 :: [4, 6, 8, 10] \\ &\Rightarrow [2, 4, 6, 8, 10] \end{aligned}$$

The last three steps put the elements from the first list on to the second. As with *take*, the cost of building the result exceeds that of the deep recursion; an iterative version is not needed. The cost of evaluating $xs @ ys$ is proportional to the length of xs and is completely independent of ys . Even $xs @ []$ makes a copy of xs .

In Pascal and C you can implement lists using pointer types, and join them by updating the last pointer of one list to point towards another. Destructive updating is faster than copying, but if you are careless your lists could end up in knots. What happens if the two lists happen to be the same pointer? ML lists involve internal pointers used safely. If you like living dangerously, ML has explicit pointer types — see Chapter 8.

The function rev. List reversal can be defined using *append*. The head of the list becomes the last element of the reversed list:

```

fun nrev [] = []
  | nrev (x::xs) = (nrev xs) @ [x];
> val nrev = fn : 'a list -> 'a list

```

This is grossly inefficient. If *nrev* is given a list of length $n > 0$, then *append* calls *cons* (*::*) exactly $n - 1$ times to copy the reversed tail. Constructing the list $[x]$ calls *cons* again, for a total of n calls. Reversing the tail requires $n - 1$ more *conses*, and so forth. The total number of *conses* is

$$0 + 1 + 2 + \dots + n = \frac{n(n + 1)}{2}.$$

This cost is quadratic: proportional to n^2 .

We have already seen, in *rtake*, another way of reversing a list: repeatedly move elements from one list to another.

```

fun revAppend ([], ys) = ys
  | revAppend (x::xs, ys) = revAppend (xs, x::ys);
> val revAppend = fn : 'a list * 'a list -> 'a list

```

Append is never called. The number of steps is proportional to the length of the list being reversed. The function resembles *append* but reverses its first argument:

```

revAppend (["Macbeth", "and", "Banquo"], ["all", "hail!"]);
> ["Banquo", "and", "Macbeth", "all", "hail!"] : string list

```

The efficient reversal function calls *revAppend* with an empty list:

```

fun rev xs = revAppend (xs, []);
> val rev = fn : 'a list -> 'a list

```

Here a slightly longer definition pays dramatically. Reversing a 1000-element list, *rev* calls *::* exactly 1000 times, *nrev* 500,500 times. Furthermore, the recursion in *revAppend* is iterative. Its key idea — of accumulating list elements in an extra argument rather than appending — applies to many other functions.

Exercise 3.5 Modify the *append* function to handle $xs @ []$ efficiently.

Exercise 3.6 What would happen if we changed $[x]$ to x in the definition of *nrev*?

Exercise 3.7 Show the computation steps to reverse the list $[1, 2, 3, 4]$ using *nrev* and then *rev*.

3.6 Lists of lists, lists of pairs

Pattern-matching and polymorphism cope nicely with combinations of data structures. Observe the types of these functions.

The function concat. This function makes a list consisting of all the elements of a list of lists:

```
fun concat [] = []
  | concat (l::ls) = l @ concat ls;
> val concat = fn : 'a list list -> 'a list
concat [ ["When", "shall"], ["we", "three"], ["meet", "again"] ];
> [ "When", "shall", "we", "three", "meet", "again" ]
> : string list
```

The copying in $l @ \text{concat } ls$ is reasonably fast, for l is usually much shorter than $\text{concat } ls$.

The function zip. This function pairs corresponding members of two lists:

$$\text{zip}([x_1, \dots, x_n], [y_1, \dots, y_n]) = [(x_1, y_1), \dots, (x_n, y_n)]$$

If the two lists differ in length, let us ignore surplus elements. The declaration requires complex patterns:

```
fun zip (x::xs, y::ys) = (x, y) :: zip (xs, ys)
  | zip _ = [];
> val zip = fn : 'a list * 'b list -> ('a*'b) list
```

The second pattern in the definition of *zip*, with its wildcard, matches all possibilities. But it is only considered if the first pattern fails to match. ML considers a function's patterns in the order given.

The function unzip. The inverse of *zip*, called *unzip*, takes a list of pairs to a pair of lists:

$$\text{unzip}[(x_1, y_1), \dots, (x_n, y_n)] = ([x_1, \dots, x_n], [y_1, \dots, y_n])$$

Building two lists simultaneously can be tricky in functional languages. One approach uses an extra function:

```
fun conspair ((x, y), (xs, ys)) = (x::xs, y::ys);
fun unzip [] = ([], [])
  | unzip (pair::pairs) = conspair (pair, unzip pairs);
```

A `let` declaration, where pattern-matching takes apart the result of the recursive call, eliminates the function *conspair*:

```

fun unzip [] = ([], [])
  | unzip ((x, y) :: pairs) =
    let val (xs, ys) = unzip pairs
    in (x :: xs, y :: ys) end;
> val unzip = fn : ('a*'b) list -> 'a list * 'b list

```

An iterative function can construct several results in its arguments. This is the simplest way to unzip a list, but the resulting lists are reversed.

```

fun rev_unzip([], xs, ys) = (xs, ys)
  | rev_unzip((x, y) :: pairs, xs, ys) =
    rev_unzip(pairs, x :: xs, y :: ys);

```



Lists and the standard library. The standard library provides most of the functions described above. Append (the infix @), reverse (*rev*), *null*, *hd*, *tl* and *length* are available at top level. *List* provides *take*, *drop* and *concat*, among others. *ListPair* provides *zip* and *unzip*.

Please use the library versions of these functions. Those given here omit error handling. The library versions respond to erroneous inputs by raising exceptions such as *List.Empty* (if you request the head of the empty list) and *Subscript* (if you attempt to *take* more elements than exist). Library versions will also be tuned for efficiency.

Exercise 3.8 Compare the following function with *concat*, considering its effect and efficiency:

```

fun f [] = []
  | f ([] :: ls) = f(ls)
  | f ((x :: l) :: ls) = x :: f(l :: ls);

```

Exercise 3.9 Give an equivalent definition of *zip* that does not depend upon the order in which patterns are considered.

Exercise 3.10 Is *rev (rtake (l, i, []))* more efficient than *take (l, i)*? Consider all the costs involved.

Applications of lists

This section demonstrates how lists can perform sophisticated tasks, like binary arithmetic and matrix operations. Feel free to skip the harder ones if you like.

Two examples from the classic book *A Discipline of Programming* (Dijkstra, 1976) are also solved. Dijkstra presents programs in all their ‘compelling and deep logical beauty.’ His programs use arrays; do lists possess greater beauty?

3.7 Making change

Let us start with something simple: making change. The task is to express some amount of money in terms of coins drawn from a list of coin values. Naturally, we expect to receive change using the largest coins possible. This is easy if coin values are supplied in decreasing order:

```
fun change (coinvals, 0)          = []
  | change (c::coinvals, amount) =
      if amount < c then change(coinvals, amount)
      else c :: change(c::coinvals, amount-c);
> ***Warning: Patterns not exhaustive
> val change = fn : int list * int -> int list
```

The function definition could hardly be more intuitive. If the target amount is zero, no coins are required; if the largest coin value c is too large, discard it; otherwise use it and make change for the amount less c .

Let us declare ML identifiers for British and U.S. coin values:

```
val gb_coins = [50,20,10,5,2,1]
and us_coins = [25,10,5,1];
```

Thus, 43 pence is expressed differently from 43 cents:

```
change(gb_coins, 43);
> [20, 20, 2, 1] : int list
change(us_coins, 43);
> [25, 10, 5, 1, 1, 1] : int list
```

Making change is less trivial than it first appears. Suppose the only coin values we have are 5 and 2?

```
change([5,2], 16);
> Exception: Match
```

The compiler warned us of this possibility when we declared *change*. But 16 is easily expressed using fives and twos! Our algorithm is greedy: it always chooses the largest coin value, trying to express 16 as $5 + 5 + 5 + c$, only now $c = 1$ is not possible.

Can we design a better algorithm? **Backtracking** means responding to failure by undoing the most recent choice and trying again. One way of implementing backtracking involves exceptions, as shown below in Section 4.8. An alternative approach is to compute the list of all solutions. Observe the use of *coins* to hold the list of coins chosen so far.

```

fun allChange (coins, coinvals, 0)           = [coins]
  | allChange (coins, [], amount)           = []
  | allChange (coins, c::coinvals, amount) =
      if amount < 0 then []
      else allChange (c::coins, c::coinvals, amount - c) @
            allChange (coins, coinvals, amount);
> val allChange = fn
> : int list * int list * int -> int list list

```

The ‘patterns not exhaustive’ warning has disappeared; the function considers all cases. Given an impossible problem, it returns the empty list instead of raising an exception:

```

allChange([], [10,2], 27);
> [] : int list list

```


Let us try some more interesting examples:

```

allChange([], [5,2], 16);
> [[2, 2, 2, 5, 5], [2, 2, 2, 2, 2, 2, 2]]
> : int list list
allChange([], gb_coins, 16);
> [[1, 5, 10], [2, 2, 2, 10], [1, 1, 2, 2, 10], ...]
> : int list list

```

There are 25 ways of making change for 16 pence! At first sight, this approach looks untenable. To control the exponential growth in solutions we can use lazy evaluation, generating only the solutions required (see Section 5.14).

 *Further reading.* Making change is much harder than it first appears, in the general case. It is closely related to the *subset-sum* problem, which is NP-complete. This means it is highly unlikely that there exists an efficient algorithm to decide whether or not a given set of coins can be used to express a given sum. Cormen *et al.* (1990) discuss algorithms for finding approximate solutions.

Exercise 3.11 Write a function to express integers as Roman numerals. Supplied with suitable arguments, your function should be able to express 1984 as either MDCCCCLXXXIII or MCMLXXXIV.

Exercise 3.12 The change functions expect *coinvals* to consist of strictly decreasing positive integers. What happens if this precondition is violated?

Exercise 3.13 We are seldom fortunate enough to have an infinite supply of coins. Modify *allChange* to make change from a finite purse.

Exercise 3.14 Modify *allChange* to accumulate its result in an extra argument, eliminating the call to *append*. Compare its efficiency with the original version by making change for 99 pence.

3.8 Binary arithmetic

Functional programming may seem far removed from hardware, but lists are good for simulating digital circuits. Binary addition and multiplication are defined here for lists of zeros and ones.

Addition. If you have forgotten the rules for binary sums, have a look at the binary version of $11 + 30 = 41$:

$$\begin{array}{r} 11110 \\ + 1011 \\ \hline 101001 \end{array}$$

Addition works from right to left. The two bits plus any carry (from the right) give a sum bit for this position and a carry to the left. Right to left is the wrong direction for lists; the head of a list is its leftmost element. So the bits will be kept in reverse order.

The two binary numerals may have unequal lengths. If one bit list terminates then the carry must be propagated along the other bit list.

```
fun bincarry (0, ps)      = ps
  | bincarry (1, [])     = [1]
  | bincarry (1, p::ps) = (1-p) :: bincarry(p, ps);
> ***Warning: Patterns not exhaustive
> val bincarry = fn : int * int list -> int list
```

Yes, patterns may contain constants: integers, reals, booleans and strings. Function *bincarry* can propagate a carry of 0 or 1, the only sensible values. It is undefined for others.

The binary sum is defined for two bit lists and a carry. When either list terminates, *bincarry* deals with the other. If there are two bits to add, their sum and carry are computed:

```
fun binsum (c, [], qs)      = bincarry (c, qs)
  | binsum (c, ps, [])     = bincarry (c, ps)
  | binsum (c, p::ps, q::qs) =
      ((c+p+q) mod 2) :: binsum((c+p+q) div 2, ps, qs);
> val binsum = fn
> : int * int list * int list -> int list
```

Let us try $11 + 30 = 41$, remembering that the bits are kept in reverse order:

```
binsum(0, [1,1,0,1], [0,1,1,1,1]);
> [1, 0, 0, 1, 0, 1] : int list
```

Multiplication. The binary product is computed by shifting and adding. For instance, $11 \times 30 = 330$:

$$\begin{array}{r}
 11110 \\
 \times 1011 \\
 \hline
 11110 \\
 11110 \\
 + 11110 \\
 \hline
 101001010
 \end{array}$$

Here, shifting is performed by inserting a 0:

```
fun binprod ([], _) = []
  | binprod (0::ps, qs) = 0::binprod(ps, qs)
  | binprod (1::ps, qs) = binsum(0, qs, 0::binprod(ps, qs));
> ***Warning: Patterns not exhaustive
> val binprod = fn : int list * int list -> int list
```

Let us evaluate $11 \times 30 = 330$:

```
binprod([1,1,0,1], [0,1,1,1,1]);
> [0, 1, 0, 1, 0, 0, 1, 0, 1] : int list
```

A structure for binary arithmetic. In a large program, it is poor style to associate related functions merely by a naming convention such as the prefix *bin*. The binary arithmetic functions should be grouped into a structure, say *Bin*. Shorter names can be used inside the structure, making the code more readable. Outside, the structure's components have uniform compound names. The function declarations above can easily be packaged into a structure:

```
structure Bin =
  struct
    fun carry (0, ps) = ...
      fun sum (c, [], qs) = ...
        fun prod ([], _) = ...
  end;
```

With a little more effort, structure *Bin* can be made to match signature *ARITH* of Section 2.22. This would give the operators for binary numerals precisely the same interface as those for complex numbers. It would include binary arithmetic in our collection of structures that can be used in generic arithmetic packages. But binary arithmetic is quite different from complex arithmetic; division is not

exact, for example. Noting which properties are needed in a particular case is our responsibility.

Exercise 3.15 Write functions to compute the binary sum and product of a list of boolean values, using no built-in arithmetic.

Exercise 3.16 Write a function to divide one binary numeral by another.

Exercise 3.17 Using the results of the previous exercise, or by writing dummy functions, extend structure *Bin* so that it matches signature *ARITH*.

Exercise 3.18 Decimal numerals can be held as lists of integers from 0 to 9. Write functions to convert between binary and decimal: both directions. Compute the factorial of 100.

3.9 Matrix transpose

A matrix can be viewed as a list of rows, each row a list of matrix elements. The matrix $\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$, for instance, can be declared in ML by

```
val matrix = [ ["a", "b", "c"],
               ["d", "e", "f"] ];
> val matrix = [ ["a", "b", "c"], ["d", "e", "f"] ]
> : string list list
```

Matrix transpose works well with this list representation because it goes sequentially along rows and columns, with no jumping. The transpose function changes the list of rows

$$A = \begin{bmatrix} [x_{11}, & x_{12}, & \dots, & x_{1m}], \\ \vdots & \vdots & & \vdots \\ [x_{n1}, & x_{n2}, & \dots, & x_{nm}] \end{bmatrix}$$

to the list of the columns of A :

$$A^T = \begin{bmatrix} [x_{11}, & \dots, & x_{n1}], \\ [x_{12}, & \dots, & x_{n2}], \\ \vdots & & \vdots \\ [x_{1m}, & \dots, & x_{nm}] \end{bmatrix}$$

One way to transpose a matrix is by repeatedly taking columns from it. The heads of the rows form the first column of the matrix:

```

fun headcol [] = []
  | headcol ((x::_) :: rows) = x :: headcol rows;
> ***Warning: Patterns not exhaustive
> val headcol = fn : 'a list list -> 'a list

```

The tails of the rows form a matrix of the remaining columns:

```

fun tailcols [] = []
  | tailcols ((_::xs) :: rows) = xs :: tailcols rows;
> ***Warning: Patterns not exhaustive
> val tailcols = fn : 'a list list -> 'a list list

```

Consider their effect on our small matrix:

```

headcol matrix;
> ["a", "d"] : string list
tailcols matrix;
> [{"b", "c"}, {"e", "f"}] : string list list

```

Calling *headcol* and *tailcols* chops the matrix like this:

$$\left(\begin{array}{c|cc} a & b & c \\ \hline d & e & f \end{array} \right)$$

These functions lead to an unusual recursion: *tailcols* takes a list of n lists and returns a list of n shorter lists. This terminates with n empty lists.

```

fun transp ([]::rows) = []
  | transp rows = headcol rows :: transp (tailcols rows);
> val transp = fn : 'a list list -> 'a list list
transp matrix;
> [{"a", "d"}, {"b", "e"}, {"c", "f"}] : string list list

```

The transposed matrix is

$$\left(\begin{array}{cc} a & d \\ b & e \\ c & f \end{array} \right).$$



A neater way. Many of the programs presented here can be expressed more concisely using higher-order functions such as *map*, which applies a function to every element of a list. Higher-order functions are discussed later on. You may want to peek at Section 5.7, where matrix transpose is reconsidered.

Exercise 3.19 What input pattern do *headcol* and *tailcols* not handle? What does *transp* return if the rows of the ‘matrix’ do not have the same length?

Exercise 3.20 What does *transp* do given the empty list? Explain.

Exercise 3.21 Write an alternative transpose function. Instead of turning columns into rows, it should turn rows into columns.

3.10 Matrix multiplication

We begin with a quick review of matrix multiplication. The *dot product* (or inner product) of two vectors is

$$(a_1, \dots, a_k) \cdot (b_1, \dots, b_k) = a_1 b_1 + \dots + a_k b_k.$$

If A is an $m \times k$ matrix and B is a $k \times n$ matrix then their *product* $A \times B$ is an $m \times n$ matrix. For each i and j , the (i, j) element of $A \times B$ is the dot product of row i of A with column j of B . Example:

$$\begin{pmatrix} 2 & 0 \\ 3 & -1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 2 \\ 4 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 4 \\ -1 & 1 & 6 \\ 4 & -1 & 0 \\ 5 & -1 & 2 \end{pmatrix}$$

The (1,1) element of the product above is computed by

$$(2, 0) \cdot (1, 4) = 2 \times 1 + 0 \times 4 = 2.$$

In the dot product function, the two vectors must have the same length; ML prints a warning that some cases are not covered. Henceforth these warnings will usually be omitted.

```
fun dotprod([], []) = 0.0
  | dotprod(x::xs, y::ys) = x*y + dotprod(xs, ys);
> ***Warning: Patterns not exhaustive
> val dotprod = fn : real list * real list -> real
```

If A has just one row, so does $A \times B$. Function *rowprod* computes the product of a row with B . The matrix B must be given as its transpose: a list of columns, not a list of rows.

```
fun rowprod(row, []) = []
  | rowprod(row, col::cols) =
      dotprod(row, col) :: rowprod(row, cols);
> val rowprod = fn
> : real list * real list list -> real list
```

Each row of $A \times B$ is obtained by multiplying a row of A by the columns of B :

```
fun rowlistprod([], cols) = []
  | rowlistprod(row::rows, cols) =
      rowprod(row, cols) :: rowlistprod(rows, cols);
> val rowlistprod = fn
> : real list list * real list list -> real list list
```

The matrix product function makes *transp* construct a list of the columns of *B*:

```
fun matprod (rowsA, rowsB) = rowlistprod (rowsA, transp rowsB);
> val matprod = fn
> : real list list * real list list -> real list list
```

Here are the declarations of the sample matrices, omitting ML's response:

```
val rowsA = [ [2.0, 0.0],
               [3.0, ~1.0],
               [0.0, 1.0],
               [1.0, 1.0] ]
and rowsB = [ [1.0, 0.0, 2.0],
               [4.0, ~1.0, 0.0] ];
```

Here is their product:

```
matprod (rowsA, rowsB);
> [[2.0, 0.0, 4.0],
>  [~1.0, 1.0, 6.0],
>  [4.0, ~1.0, 0.0],
>  [5.0, ~1.0, 2.0]] : real list list
```

Exercise 3.22 A matrix is negated by negating each of its components; thus $-\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} -a & -b \\ -c & -d \end{pmatrix}$. Write a function to negate a matrix.

Exercise 3.23 Two matrices of the same dimensions are added by adding corresponding components; thus $\begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} a+a' & b+b' \\ c+c' & d+d' \end{pmatrix}$. Write a function to add two matrices.

3.11 Gaussian elimination

One of the classic matrix algorithms, Gaussian elimination may seem an unlikely candidate for functional programming. This algorithm (Sedgewick, 1988) can compute the determinant or inverse of a matrix, or solve systems of independent linear equations such as the following:

$$\begin{aligned} x + 2y + 7z &= 7 \\ -4w + 3y - 5z &= -2 \\ 4w - x - 2y - 3z &= 9 \\ -2w + x + 2y + 8z &= 2 \end{aligned} \tag{*}$$

Gaussian elimination works by isolating each of the variables in turn. Equation (*), properly scaled and added to another equation, eliminates *w* from it. Repeating this operation, which is called *pivoting*, eventually reduces the system

to a triangular form:

$$\begin{aligned} -4w + 3y - 5z &= -2 \\ x + 2y + 7z &= 7 \\ 3y - z &= 14 \\ 3z &= 3 \end{aligned}$$

Now the solutions come out, beginning with $z = 1$.

Equation (*) is a good choice for eliminating w because the absolute value (4) of its coefficient is maximal. Scaling divides the equation by this value; a small divisor (not to mention zero!) could cause numerical errors. Function *pivotrow*, given a list of rows, returns one whose head is greatest in absolute value.

```
fun pivotrow [row] = row : real list
  | pivotrow (row1::row2::rows) =
    if abs(hd row1) >= abs(hd row2)
    then pivotrow (row1::rows)
    else pivotrow (row2::rows);
> val pivotrow = fn : real list list -> real list
```

If the selected row has head p , then *delrow*(p , *rows*) removes it from the list of rows.

```
fun delrow (p, []) = []
  | delrow (p, row::rows) = if p = hd row then rows
    else row :: delrow (p, rows);
> val delrow = fn : 'a * 'a list list -> 'a list list
```

Function *scalarprod* multiplies a row or vector by a constant k :

```
fun scalarprod (k, []) = [] : real list
  | scalarprod (k, x::xs) = k*x :: scalarprod (k, xs);
> val scalarprod = fn : real * real list -> real list
```

Function *vectorsum* adds two rows or vectors:

```
fun vectorsum ([], []) = [] : real list
  | vectorsum (x::xs, y::ys) = x+y :: vectorsum (xs, ys);
> val vectorsum = fn : real list * real list -> real list
```

Function *elimcol*, declared inside *gausselim*, refers to the current pivot row by its head p (the leading coefficient) and tail *pro*. Given a list of rows, *elimcol* replaces each by its sum with *pro*, properly scaled. The first element of each sum is zero, but these zeros are never computed; the first column simply disappears.

```

fun gausseelim [row] = [row]
  | gausseelim rows =
    let val p::prow = pivotrow rows
        fun elimcol [] = []
          | elimcol ((x::xs)::rows) =
              vectorsum(xs, scalarprod(~x/p, prow))
              :: elimcol rows
        in (p::prow) :: gausseelim (elimcol (delrow (p, rows)))
        end;
> val gausseelim = fn : real list list -> real list list

```

Function *gausseelim* removes the pivot row, eliminates a column, and calls itself recursively on the reduced matrix. It returns a list of pivot rows, decreasing in length, forming an upper triangular matrix.

A system of n equations is solved by Gaussian elimination on an $n \times (n + 1)$ matrix, where the extra column contains the right-side values. The solutions are generated recursively from the triangular matrix. Known solutions are multiplied by their coefficients and added — this is a vector dot product — and divided by the leading coefficient. To subtract the right-side value we employ a trick: a spurious solution of -1 .

```

fun solutions [] = [~1.0]
  | solutions ((x::xs)::rows) =
    let val solns = solutions rows
        in ~(dotprod(solns, xs)/x) :: solns end;
> val solutions = fn : real list list -> real list

```

One way of understanding this definition is by applying it to the example above. We compute the triangular matrix:

```

gausseelim [[ 0.0, 1.0, 2.0, 7.0, 7.0],
            [~4.0, 0.0, 3.0, ~5.0, ~2.0],
            [ 4.0, ~1.0, ~2.0, ~3.0, 9.0],
            [~2.0, 1.0, 2.0, 8.0, 2.0]];
> [[~4.0, 0.0, 3.0, ~5.0, ~2.0],
> [ 1.0, 2.0, 7.0, 7.0],
> [ 3.0, ~1.0, 14.0],
> [ 3.0, 3.0]] : real list list

```

Ignoring the final -1 , the solutions are $w = 3$, $x = -10$, $y = 5$ and $z = 1$.

```

solutions it;
> [3.0, ~10.0, 5.0, 1.0, ~1.0] : real list%

```



Further reading. Researchers at the University of Wales have applied the language Haskell to computational fluid dynamics problems. The aim is to investigate the practical utility of functional programming. One paper compares different representations of matrices (Grant *et al.*, 1996). Another considers the possibility of ex-

exploiting parallelism, using a simulated parallel processor (Grant *et al.*, 1995). Compared with conventional Fortran implementations, the Haskell ones are much slower and need more space; the authors list some developments that could improve the efficiency.

Exercise 3.24 Show that if the input equations are linearly independent, then division by zero cannot occur within *gausselim*.

Exercise 3.25 Do *pivotrow* and *delrow* work correctly if the heads of several rows have the same absolute value?

Exercise 3.26 Write a function to compute the determinant of a matrix.

Exercise 3.27 Write a function to invert a matrix.

Exercise 3.28 Write a structure *Matrix* that matches signature *ARITH*. You can either use the previous exercise and those of Section 3.10, or write dummy functions. This adds matrices to our collection of arithmetic structures.¹

3.12 Writing a number as the sum of two squares

Dijkstra (1976) presents a program that, given an integer r , finds all integer solutions of $x^2 + y^2 = r$. (Assume $x \geq y \geq 0$ to suppress symmetries.) For instance, $25 = 4^2 + 3^2 = 5^2 + 0^2$, while 48,612,265 has 32 solutions.

Brute force search over all (x, y) pairs is impractical for large numbers, but fortunately the solutions have some structure: if $x^2 + y^2 = r = u^2 + v^2$ and $x > u$ then $y < v$. If x sweeps downwards from \sqrt{r} as y sweeps upwards from 0, then all solutions can be found in a single pass.

Let $Bet(x, y)$ stand for the set of all solutions between x and y :

$$Bet(x, y) = \{(u, v) \mid u^2 + v^2 = r \wedge x \geq u \geq v \geq y\}$$

The search for suitable x and y is guided by four observations:

- 1 If $x^2 + y^2 < r$ then $Bet(x, y) = Bet(x, y + 1)$. There are plainly no solutions of the form (u, y) for $u < x$.
- 2 If $x^2 + y^2 = r$ then $Bet(x, y) = \{(x, y)\} \cup Bet(x - 1, y + 1)$. A solution! There can be no other for the same x or y .
- 3 If $x^2 + y^2 > r > x^2 + (y - 1)^2$ then $Bet(x, y) = Bet(x - 1, y)$. There can be no solutions of the form (x, v) .

¹ There is a serious problem: what is the component *zero*? The obvious choice is [], but the matrix operations will require careful modification in order to treat this correctly as the zero matrix.

4 Finally, $Bet(x, y) = \emptyset$ if $x < y$.

These suggest a recursive — indeed iterative — search method. Case 3 requires special care if it is to be used efficiently. At the start, make sure $x^2 + y^2 < r$ holds. Increase y until $x^2 + y^2 \geq r$. If $x^2 + y^2 > r$ then y must be the least such, and so Case 3 applies. Decreasing x by one re-establishes $x^2 + y^2 < r$.


Initially $y = 0$ and $x = \sqrt{r}$ (the integer square root of r), so the starting condition holds. Since $x > y$, we know that y will be increased several times when x is decreased. As a further concession to efficiency, therefore, the program takes the computation of x^2 outside the inner recursion:

```
fun squares r =
  let fun between (x,y) = (*all pairs between x and y*)
        let val diff = r - x*x
            fun above y = (*all pairs above y*)
                  if y>x then []
                  else if y*y<diff then above (y+1)
                  else if y*y=diff then (x,y)::between(x-1,y+1)
                  else (* y*y>diff *) between(x-1,y)
            in above y end;
        val firstx = floor(Math.sqrt(real r))
    in between (firstx, 0) end;
  > val squares = fn : int -> (int*int) list
```

Execution is fast, even for large r :

```
squares 50;
> [(7, 1), (5, 5)] : (int*int) list
squares 1105;
> [(33, 4), (32, 9), (31, 12), (24, 23)] : (int*int) list
squares 48612265;
> [(6972, 59), (6971, 132), (6952, 531), (6948, 581),
> (6944, 627), (6917, 876), (6899, 1008), (6853, 1284),
> (6789, 1588), (6772, 1659), ...] : (int*int) list
```

Dijkstra's program has a different search method: x and y start with equal values, then sweep apart. Our method could well be the one he rejected because 'the demonstration that no solutions had been omitted always required a drawing.'

 *A smarter way?* A number is the sum of two squares precisely if, in its prime factorization, every factor of the form $4k + 3$ appears with an even exponent. For example, $48, 612, 265 = 5 \times 13 \times 17 \times 29 \times 37 \times 41$ and none of these primes has the form $4k + 3$. The criterion itself merely tells us whether solutions exist, but the theory also provides a means of enumerating the solutions (Davenport, 1952, Chapter V). A program exploiting this theory would pay off only for huge numbers.

3.13 The problem of the next permutation

Given a list of integers, we are asked to rearrange the elements to produce the permutation that is next greater under lexicographic ordering. The new permutation should be greater than the one given, with no other permutation in between.

Let us modify the problem slightly. Lexicographic order means the head of the list has the most significance. The next greater permutation will probably differ in the least significant elements. Since the head of a list is the easiest element to reach, let us make it least significant. Fighting the natural order of lists would be foolish. We therefore compute the next permutation under reverse lexicographic ordering.

The problem is hard to visualize — even Dijkstra gives an example. Here are the next eight permutations after 4 3 2 1 (the initial permutation):

3 4 2 1
4 2 3 1
2 4 3 1
3 2 4 1
2 3 4 1
4 3 1 2
3 4 1 2
4 1 3 2

The affected part of each is underlined. The sequence of permutations terminates at 1 2 3 4, which has no successor.

To make a greater permutation, some element of the list must be replaced by a larger element to its left. To make the very next permutation, this replacement must happen as far to the left — the least significant position — as possible. The replacement value must be as small as possible, and the elements to the left of the replacement must be arranged in descending order. All this can be done in two steps:

- 1 Find the leftmost element y that has a greater element to its left. The elements to its left will therefore be an increasing sequence $x_1 \leq \dots \leq x_n$. (We are really speaking of positions rather than elements, but this only matters if the elements are not distinct.)
- 2 Replace y by the smallest x_i , with $1 \leq i \leq n$, such that $y < x_i$, and arrange $x_1, \dots, x_{i-1}, y, x_{i+1}, \dots, x_n$ in descending order. This can be accomplished by scanning x_n, x_{n-1}, \dots, x_1 until the correct value is found for x_i , placing larger elements in front of the final result.

Calling *next(xlist, ys)* finds the *y* in *ys* to replace, while *xlist* accumulates the elements passed over. When *xlist* holds the reversed list $[x_n, \dots, x_1]$, the function *swap* performs the replacement and rearrangement. The list manipulations are delicate.

```
fun next(xlist, y::ys) : int list =
  if hd xlist <= y then next(y::xlist, ys)
  else (*swap y with greatest xk such that x>=xk>y*)
    let fun swap [x] = y::x::ys
        | swap (x::xk::xs) = (*x >= xk*)
          if xk>y then x::swap(xk::xs)
            else (y::xk::xs)@(x::ys)
          (*x > y >= xk >= xs*)
    in swap(xlist) end;
> val next = fn : int list * int list -> int list
```

Function *nextperm* starts the scan.

```
fun nextperm (y::ys) = next([y], ys);
> val nextperm = fn : int list -> int list
nextperm [1,2,4,3];
> [3, 2, 1, 4] : int list
nextperm it;
> [2, 3, 1, 4] : int list
nextperm it;
> [3, 1, 2, 4] : int list
```

It also works when the elements are not distinct:

```
nextperm [3,2,2,1];
> [2, 3, 2, 1] : int list
nextperm it;
> [2, 2, 3, 1] : int list
nextperm it;
> [3, 2, 1, 2] : int list
```

Exercise 3.29 Write the steps to compute *nextperm*[2, 3, 1, 4].

Exercise 3.30 Does *next* still work if the \leq comparison is replaced by $<$ in its second line? Justify your answer in terms of the two steps described above.

Exercise 3.31 What does *nextperm(ys)* return if there is no next permutation of *ys*? Modify the program so that it returns the initial permutation in that case.

The equality test in polymorphic functions

Polymorphic functions like *length* and *rev* accept lists having elements of any type because they do not perform any operations on those elements. Now

consider a function to test whether a value e is a member of a list l . Is this function polymorphic? Each member of l must be tested for equality with e . Equality testing is polymorphic, but in a restricted sense.

3.14 Equality types

An **equality type** is a type whose values admit equality testing. Equality testing is forbidden on function types and abstract types:

- The equality test on functions is not computable because f and g are equal just when $f(x)$ equals $g(x)$ for every possible argument x . There are other ways of defining equality of functions, but there is no escaping the problem.
- An abstract type provides only those operations specified in its definition. ML hides the representation's equality test, for it seldom coincides with the desired abstract equality.²

Equality is defined for the basic types: integers, reals, characters, strings, booleans. For structured values, the equality test compares corresponding components; equality is thus defined for tuples, records, lists and datatypes (introduced in the next chapter) built over the basic types. It is not defined for values containing functions or elements of abstract types.

Standard ML provides **equality type variables** $\alpha^=, \beta^=, \gamma^=, \dots$ ranging over the equality types. Equality types contain no type variables other than equality type variables. For example, int , $bool \times string$ and $(int\ list) \times \beta^=$ are equality types, while $int \rightarrow bool$ and $bool \times \beta$ are not.

Here is the type of the equality test itself, the infix operator (=):

```
op= ;
> fn : ('a * 'a) -> bool
```

In mathematical notation this type is $\alpha^= \times \alpha^= \rightarrow bool$. In ML, an equality type variable begins with two ' characters.

Now let us declare the membership testing function:

```
infix mem;
fun (x mem []) = false
  | (x mem (y::l)) = (x=y) orelse (x mem l);
> val mem = fn : 'a * 'a list -> bool
```

The type $\alpha^= \times (\alpha^=list) \rightarrow bool$ means that *mem* may be applied to any list whose elements permit equality testing.

² Chapter 7 describes abstract types in detail.

```
"Sally" mem ["Regan", "Goneril", "Cordelia"];
> false : bool
```

3.15 Polymorphic set operations

A function's type contains equality type variables if it performs polymorphic equality testing, even indirectly, for instance via *mem*. The function *newmem* adds a new element to a list, provided it is really new:

```
fun newmem(x, xs) = if x mem xs then xs else x::xs;
> val newmem = fn : 'a * 'a list -> 'a list
```

Lists constructed by *newmem* can be regarded as finite sets.³ Let us declare some set operations and note their types. If equality type variables appear, then equality tests are involved.

The function *setof* converts a list to a 'set' by eliminating repeated elements:

```
fun setof [] = []
  | setof (x::xs) = newmem(x, setof xs);
> val setof = fn : 'a list -> 'a list
setof [true, false, false, true, false];
> [true, false] : bool list
```

Observe that *setof* may perform many equality tests. To minimize the use of *setof*, the following functions can be applied to 'sets' — lists of distinct elements — to ensure that their result is a 'set.'

Union. The list *union(xs, ys)* includes all elements of *xs* not already in *ys*, which is assumed to consist of distinct elements:

```
fun union([], ys) = ys
  | union(x::xs, ys) = newmem(x, union(xs, ys));
> val union = fn : 'a list * 'a list -> 'a list
```

The type variable 'a indicates equality testing, here via *newmem*.

```
union([1, 2, 3], [0, 2, 4]);
> [1, 3, 0, 2, 4] : int list
```

Intersection. Similarly, *inter(xs, ys)* includes all elements of *xs* that also belong to *ys*:

³ Section 3.22 below considers more deeply the question of representing one data structure using another.

```

fun inter([], ys)      = []
  | inter(x::xs, ys) = if x mem ys then x::inter(xs, ys)
                       else      inter(xs, ys);
> val inter = fn : 'a list * 'a list -> 'a list

```

A baby's name can be chosen by intersecting the preferences of both parents ...

```

inter(["John", "James", "Mark"], ["Nebuchadnezzar", "Bede"]);
> [] : string list

```

... although this seldom works.

The subset relation. Set T is a **subset** of S if all elements of T are also elements of S :

```

infix subs;
fun ([]      subs ys) = true
  | ((x::xs) subs ys) = (x mem ys) andalso (xs subs ys);
> val subs = fn : 'a list * 'a list -> bool

```

Recall that equality types may involve tuples, lists and so forth:

```

[("May", 5), ("June", 6)] subs [("July", 7)];
> false : bool

```

Equality of sets. The built-in list equality test is not valid for sets. The lists $[3, 4]$ and $[4, 3]$ are not equal, yet they denote the same set, $\{3, 4\}$. Set equality ignores order. It can be defined in terms of subsets:

```

infix seq;
fun (xs seq ys) = (xs subs ys) andalso (ys subs xs);
> val seq = fn : 'a list * 'a list -> bool
[3, 1, 3, 5, 3, 4] seq [1, 3, 4, 5];
> true : bool

```

Sets ought to be declared as an abstract type, hiding the equality test on lists.

Powerset. The **powerset** of a set S is the set consisting of all the subsets of S , including the empty set and S itself. It can be computed by removing some element x from S and recursively computing the powerset of $S - \{x\}$. If T is a subset of $S - \{x\}$ then both T and $T \cup \{x\}$ are subsets of S and elements of the powerset. The argument *base* accumulates items (like x) that must be included in each element of the result. In the initial call, *base* should be empty.

```

fun powset ([], base) = [base]
  | powset (x::xs, base) =
      powset(xs, base) @ powset(xs, x::base);
> val powset = fn : 'a list * 'a list -> 'a list list

```

The ordinary type variables indicate that *powset* does not perform equality tests.

```
powset (rev ["the","weird","sisters"], []);
> [[], ["the"], ["weird"], ["the", "weird"], ["sisters"],
>  ["the", "sisters"], ["weird", "sisters"],
>  ["the", "weird", "sisters"]] : string list list
```

Using set notation, the result of *powset* can be described as follows, ignoring the order of list elements:

$$\text{powset}(S, B) = \{T \cup B \mid T \subseteq S\}$$

Cartesian product. The **Cartesian product** of S and T is the set of all pairs (x, y) with $x \in S$ and $y \in T$. In set notation,

$$S \times T = \{(x, y) \mid x \in S, y \in T\}.$$

Several functional languages support some set notation, following David Turner; see Bird and Wadler (1988) for examples. Since ML does not, we must use recursion over lists. The function to compute Cartesian products is surprisingly complex.

```
fun cartprod ([], ys) = []
  | cartprod (x::xs, ys) =
    let val xsprod = cartprod (xs, ys)
        fun pairx [] = xsprod
          | pairx (y::ytail) = (x, y) :: (pairx ytail)
        in pairx ys end;
> val cartprod = fn : 'a list * 'b list -> ('a * 'b) list
```

The function *cartprod* does not perform equality tests.

```
cartprod ([2,5], ["moons","stars","planets"]);
> [(2, "moons"), (2, "stars"), (2, "planets"),
>  (5, "moons"), (5, "stars"), (5, "planets")]
> : (int * string) list
```

Section 5.10 will demonstrate how higher-order functions can express this function. For now, let us continue with simple methods.

Exercise 3.32 How many equality tests does ML perform when evaluating the following expressions?

```
1 mem upto (1, 500)
setof (upto (1, 500))
```


Exercise 3.33 Compare *union* with the function *itunion* declared below. Which function is more efficient?

```
fun itunion ([], ys) = ys
  | itunion (x::xs, ys) = itunion (xs, newmem (x, ys));
```

Exercise 3.34 Write a function *choose* such that *choose*(*k*, *xs*) generates the set of all *k*-element subsets of *xs*. For instance, *choose*(29, *upto*(1, 30)) should return a list containing 30 subsets.

Exercise 3.35 The following function is simpler than *cartprod*. Is it better for computing Cartesian products?

```
fun cprod ([], ys) = []
  | cprod (x::xs, ys) =
    let fun pairx [] = cprod (xs, ys)
        | pairx (y::ytail) = (x, y) :: (pairx ytail)
    in pairx ys end;
```

3.16 Association lists

A dictionary or table can be represented by a list of pairs. Functions to search such tables involve equality polymorphism. To store the dates of history's greatest battles we could write

```
val battles =
  [("Crecy", 1346), ("Poitiers", 1356), ("Agincourt", 1415),
   ("Trafalgar", 1805), ("Waterloo", 1815)];
```

A list of (key, value) pairs is called an **association list**. The function *assoc* finds the value associated with a key by sequential search:


```
fun assoc ([], a) = []
  | assoc ((x, y)::pairs, a) = if a=x then [y]
                               else assoc (pairs, a);
> val assoc = fn : ('a * 'b) list * 'a -> 'b list
```

Its type, $(\alpha^= \times \beta)list \times \alpha^= \rightarrow \beta list$, indicates that keys must have some equality type $\alpha^=$, while values may have any type β at all. Calling *assoc* (*pairs*, *x*) returns [] if the key *x* is not found, and returns [*y*] if *y* is found paired with *x*. Returning a list of results is a simple method of distinguishing success from failure.

```
assoc (battles, "Agincourt");
> [1415] : int list
assoc (battles, "Austerlitz");
```

```
> [] : int list
```

Searching can be slow, but updating is trivial: put a new pair in front. Since *assoc* returns the first value it finds, existing associations can be overridden. Pairing names with types in a block-structured language is a typical application. A name will be paired with several types in the association list if it is declared in nested blocks.

 *Equality types: good or bad?* Appel (1993) criticises ML's equality polymorphism on several grounds. They complicate the language definition. They complicate the implementation; data must have run-time tags to support equality testing, or else an equality test must be passed implicitly to functions. Sometimes the standard equality test is inappropriate, as in the case of a set of sets. The polymorphic equality test can be slow.

Part of the justification for equality polymorphism is historical. ML is related to Lisp, where functions like *mem* and *assoc* are among the most basic primitives. But even Lisp has to provide different versions of these functions, performing different sorts of equality tests. If ML did not have equality polymorphism, those functions could still be expressed by taking the testing function as an extra argument.

Equality is really overloaded: its meaning depends upon its type. Other overloaded functions are the arithmetic operators and functions to express values as strings. ML's treatment of overloading seems unsatisfactory, especially compared with Haskell's elegant *type classes* (Hudak *et al.*, 1992). But type classes also complicate the language. More seriously, a program cannot be executed — even in principle — without a full type checking. Odersky *et al.* (1995) discuss an alternative setup; more research is needed.

3.17 Graph algorithms

A list of pairs can also represent a directed graph. Each pair (x, y) stands for the edge $x \rightarrow y$. Thus the list

```
val graph1 = [("a", "b"), ("a", "c"), ("a", "d"),
              ("b", "e"), ("c", "f"), ("d", "e"),
              ("e", "f"), ("e", "g")];
```

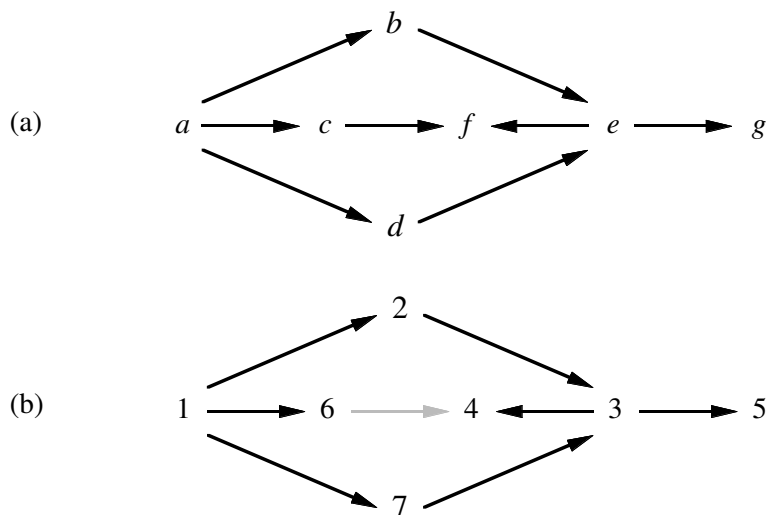
represents the graph shown in Figure 3.1(a).

The function *nexts* finds all successors of a node a — the destinations of all edges leading from a — in the graph:

```
fun nexts (a, []) = []
  | nexts (a, (x,y)::pairs) =
    if a=x then y :: nexts (a, pairs)
    else nexts (a, pairs);
> val nexts = fn : 'a * ('a * 'b) list -> 'b list
```

This function differs from *assoc* by returning all values that are paired with a , not just the first:

Figure 3.1 A directed graph, and a depth-first traversal



```

nexts("e", graph1);
> ["f", "g"] : string list

```

Depth-first search. Many graph algorithms work by following edges, keeping track of nodes visited so that a node is not visited more than once. In **depth-first search**, the subgraph reachable from the current node is fully explored before other nodes are visited. The function `depthf` implements this search strategy, using the argument `visited` to accumulate the nodes in reverse order:

```

fun depthf ([], graph, visited) = rev visited
  | depthf (x::xs, graph, visited) =
    if x mem visited then depthf (xs, graph, visited)
    else depthf (nexts(x, graph) @ xs, graph, x::visited);
> val depthf = fn
> : 'a list * ('a * 'a) list * 'a list -> 'a list

```

The nodes of a graph may have any equality type.

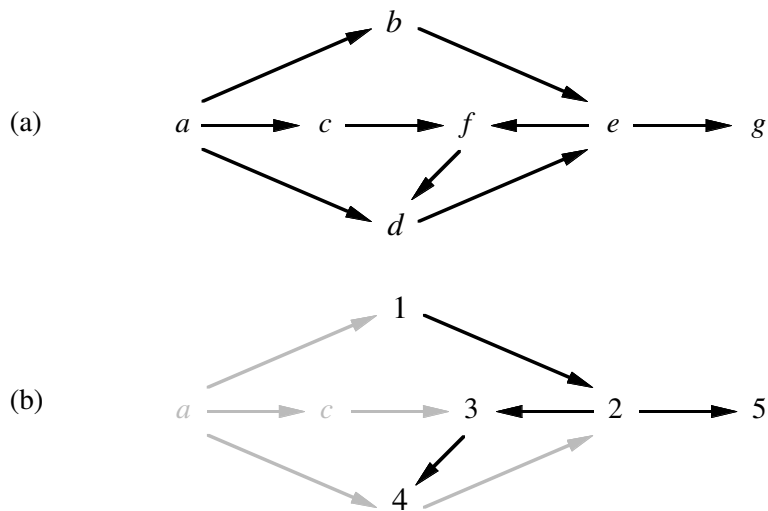
Depth-first search of `graph1` starting at `a` visits nodes in the order shown in Figure 3.1(b). One of the edges is never traversed. Let us check the traversal by calling our function:

```

depthf(["a"], graph1, []);

```

Figure 3.2 A cyclic graph, and a depth-first traversal



```
> ["a", "b", "e", "f", "g", "c", "d"] : string list
```

Adding an edge from *f* to *d* makes the graph cyclic. If that graph is searched starting at *b*, one of the edges in the cycle is ignored. Also, part of the graph is not accessible from *b* at all; see Figure 3.2.

```
depthf(["b"], ("f", "d")::graph1, []);
> ["b", "e", "f", "d", "g"] : string list
```

After visiting a node *x* that has not been visited before, depth-first search recursively visits each successor of *x*. In the list computed by `nexts(x, graph) @ xs`, the successors of *x* precede the other nodes *xs* that are awaiting visits. This list behaves as a stack. **Breadth-first search** results if the list of nodes to visit behaves as a queue.

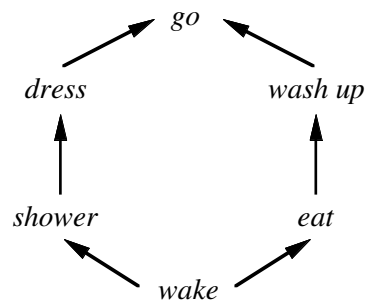
Depth-first search can also be coded as follows:

```
fun depth ([], graph, visited) = rev visited
| depth (x::xs, graph, visited) =
  depth (xs, graph,
        if x mem visited then visited
        else depth (nexts(x, graph), graph, x::visited));
```

A nested recursive call visits the successors of *x*, then another call visits the

other nodes, xs . The functions *depthf* and *depth* are equivalent, although the proof is subtle. By omitting a call to `append (@)`, *depth* is a bit faster. More importantly, since one call is devoted to visiting x , it is easily modified to detect cycles in graphs and perform topological sorting.

Topological sorting. Constraints on the order of events form a directed graph. Each edge $x \rightarrow y$ means ‘ x must happen before y .’ The graph



says everything about getting to work. Here it is as a list:

```

val grwork = [("wake", "shower"), ("shower", "dress"),
              ("dress", "go"),    ("wake", "eat"),
              ("eat", "washup"),  ("washup", "go")];
  
```

Finding a linear sequence of events from such a graph is called **topological sorting**. Sedgewick (1988) points out that depth-first search can do this if the visit to node x is recorded after its successors have been searched. Thus x comes after every node reachable from x : a topological sort in reverse.

This means a simple change to *depth*: put x on the result of the recursive call instead of the argument. The list forms in reverse so no other reversal is necessary.

```

fun topsort graph =
  let fun sort ([], visited) = visited
      | sort (x::xs, visited) =
          sort(xs, if x mem visited then visited
                  else x :: sort(nexts(x, graph), visited))
      val (starts, _) = ListPair.unzip graph
  in
    sort(starts, [])
  end;
> val topsort = fn : ('a * 'a) list -> 'a list
  
```

The `let` declaration of *sort* allows this function to refer to *graph*. It also de-

clares *starts*, the list of all starting nodes of edges, to ensure that every node in the graph is reached.

So how do we get to work?

```
topsort grwork;
> ["wake", "eat", "washup", "shower", "dress", "go"]
> : string list
```

Reversing the list of edges gives a different answer for the graph:

```
topsort (rev grwork);
> ["wake", "shower", "dress", "eat", "washup", "go"]
> : string list
```

Cycle detection. Now consider a further constraint: we must go before we eat. The resulting graph contains a cycle and admits no solution. The function call runs forever:

```
topsort (("go", "eat") :: grwork);
```

Looping is not acceptable; the function should somehow report that no solution exists. Cycles can be detected by maintaining a list of all nodes being searched. This list of nodes, called *path*, traces the edges from the start of the search.

```
fun pathsort graph =
  let fun sort ([], path, visited) = visited
      | sort (x::xs, path, visited) =
          if x mem path then hd[] (*abort!!*)
          else sort (xs, path,
                    if x mem visited then visited else
                    x :: sort (nexts (x, graph), x :: path, visited))
      val (starts, _) = ListPair.unzip graph
  in sort (starts, [], []) end;
> val pathsort = fn : ('a * 'a) list -> 'a list
```

It works on our original graph. Given a cycle it causes an error:

```
pathsort graph1;
> ["a", "d", "c", "b", "e", "g", "f"] : string list
pathsort (("go", "eat") :: grwork);
> Exception: Match
```

An error message is better than looping, but *pathsort* aborts by making an erroneous function call (namely *hd []*), an ugly trick. The next chapter explains how to declare an *exception* for such errors.

Exceptions are not the only way to report cycles. The following function returns two results: a list of visits, as before, and a list of nodes found in cycles. Maintaining two results, let us declare a function to add a visit:

```
fun newvisit (x, (visited, cys)) = (x::visited, cys);
> val newvisit = fn : 'a * ('a list * 'b) -> 'a list * 'b
```

With the help of this function, topological sorting is easily expressed:

```
fun cyclesort graph =
  let fun sort ([], path, (visited, cys)) = (visited, cys)
      | sort (x::xs, path, (visited, cys)) =
          sort (xs, path,
                if x mem path then (visited, x::cys)
                else if x mem visited then (visited, cys)
                else newvisit (x, sort (nexts (x, graph),
                                         x::path, (visited, cys))))
      val (starts, _) = ListPair.unzip graph
  in sort (starts, [], ([], [])) end;
> val cyclesort = fn
> : ('a * 'a) list -> 'a list * 'a list
```

If there is a cycle, then *cyclesort* says where it is:

```
cyclesort (("go", "eat")::grwork);
> (["wake", "shower", "dress", "go", "eat", "washup"],
> ["go"]) : string list * string list
```

And if not, then *cyclesort* sorts the graph:

```
cyclesort (rev graph1);
> (["a", "b", "c", "d", "e", "f", "g"], [])
> : string list * string list
```

These polymorphic graph functions are too slow for large graphs because of the list searches. Restricting the nodes to integers, more efficient functions can be written using the functional arrays of the next chapter.

Exercise 3.36 Modify *pathsort* to return [] if the graph has a cycle and the singleton list [*visited*] otherwise.

Exercise 3.37 Let (*visited*, *cys*) be the result of *cyclesort*. If the graph contains many cycles, will *cys* contain a node belonging to each? What can be said about *visited* if the graph contains cycles?

Sorting: A case study

Sorting is one of the most studied topics in the theory of computing. Several sorting algorithms are widely known. To sort n items, *insertion sort* takes order n^2 time; *merge sort* takes order $n \log n$ time; *quick sort* takes order $n \log n$ on average, n^2 in the worst case.

These algorithms usually sort an array. Apart from *heap sort*, where the array encodes a binary tree, they are easily coded as functions on lists. Their time complexity remains unchanged: not that a list sort will win a race against an array sort! A complexity estimate such as ‘order n^2 time’ means the execution time is proportional to n^2 . The list sort will have a higher constant of proportionality.

This section compares several sorting functions, giving the time taken to sort a list of 10,000 random numbers. These timings are informal but illustrate the practical performance of each algorithm.

The Pascal version of quick sort by Sedgewick (1988) can sort the numbers in 110 msec. This roughly equals the best time for functional sorting. Pascal beats ML if checking is disabled, but we give up the clarity and simplicity of functional programming (never mind safety). The overheads of lists would matter less for sorting, say, a bibliography, where the cost of comparisons would dominate.



How timings were measured. Timings were conducted on a Sun SuperSPARC Model 61 computer running Standard ML of New Jersey, version 108. The measurements were made using standard library facilities (structure *Timer*), and include garbage collection time. Thanks to improvements in hardware and software, ML programs run 20–40 times faster than they did in the first edition of this book.

The Pascal program was compiled using the Pascal 3.0 compiler. With array subscript checking disabled, the run-time drops to 75 msec. With full optimization the program runs in just 34 msec, but afterwards prints a warning that it ‘may have produced nonstandard floating-point results.’ What risks are worth taking in our quest for speed?

3.18 Random numbers

First, we must produce 10,000 random numbers. Park and Miller (1988), complaining that good random number generators are hard to find, recommend the following.

```
local val a = 16807.0 and m = 2147483647.0
in fun nextrand seed =
    let val t = a*seed
        in t - m * real(floor(t/m)) end
end;
> val nextrand = fn : real -> real
```

Calling *nextrand* with any *seed* between 1 and $m - 1$ yields another number in this range, performing the integer calculation

$$(a \times seed) \bmod m.$$

Real arithmetic is used to avoid integer overflow. The function works provided mantissæ are accurate to 46 bits. When trying this on your machine, check that the random numbers are exact integers.

Calling `randlist (n, seed, [])` generates a random list of length n starting from `seed`. Because the list accumulates in `tail`, its order is reversed:

```
fun randlist (n, seed, tail) =
  if n=0 then (seed, tail)
  else randlist(n-1, nextrand seed, seed::tail);
> val randlist = fn
> : int * real * real list -> real * real list
```

The list of 10,000 random numbers is called `rs`. Here are the first 15.

```
val (seed, rs) = randlist(10000, 1.0, []);
> val seed = 1043618065.0 : real
> val rs =
> [1484786315.0, 925166085.0, 1614852353.0, 721631166.0,
> 173942219.0, 1229443779.0, 789328014.0, 570809709.0,
> 1760109362.0, 270600523.0, 2108528931.0, 16480421.0,
> 519782231.0, 162430624.0, 372212905.0, ...] : real list
```

3.19 Insertion sort

Insertion sort works by inserting the items, one at a time, into a sorted list. It is slow but simple. Here is the insertion function:

```
fun ins (x, []): real list = [x]
  | ins (x, y::ys) =
    if x<=y then x::y::ys (*it belongs here*)
    else y::ins(x, ys);
> val ins = fn : real * real list -> real list
```

The type constraint `real list` resolves overloading of the comparison operator. All the sorting functions have a type constraint.

We insert some numbers into `[6.0]`, which is trivially sorted:

```
ins(4.0, [6.0]);
> [4.0, 6.0] : real list
ins(8.0, it);
> [4.0, 6.0, 8.0] : real list
ins(5.0, it);
> [4.0, 5.0, 6.0, 8.0] : real list
```

Insertion sort calls `ins` on every element of the input:

```
fun insort [] = []
  | insort (x::xs) = ins(x, insort xs);
> val insort = fn : real list -> real list
```

These functions require deep recursion. But this inefficiency is insignificant. Insertion, functional or imperative, does a lot of copying. The execution time of the sort is order n^2 . For our 10,000 integers it takes over 32 seconds, nearly 300 times slower than quick sort. Insertion sort can be considered only for short lists or those that are nearly sorted. The algorithm is worth noting because it is simple and because better sorting algorithms (merge sort and heap sort) are refinements of it.

3.20 Quick sort

Quick sort, invented by C. A. R. Hoare, was among the first efficient sorting algorithms. It works by divide and conquer:

- Choose some value a , called the *pivot*, from the input.
- Partition the remaining items into two parts: the items less than or equal to a , and the items greater than a .
- Sort each part recursively, then put the smaller part before the greater.

Quick sort is ideal for arrays — the partition step is extremely fast, moving few items. For lists, it copies all the items; *partition* is a good example of an iterative function that builds two results.

```
fun quick []      = []
  | quick [x]    = [x]
  | quick (a::bs) = (*the head "a" is the pivot*)
    let fun partition (left,right,[]): real list =
          (quick left) @ (a :: quick right)
        | partition (left,right, x::xs) =
          if x<=a then partition (x::left, right, xs)
            else partition (left, x::right, xs)
        in partition([],[],bs) end;
  > val quick = fn : real list -> real list
```

This function sorts our 10,000 numbers in about 160 msec:

```
quick rs;
> [1.0, 8383.0, 13456.0, 16807.0, 84083.0, 86383.0,
> 198011.0, 198864.0, 456291.0, 466696.0, 524209.0,
> 591308.0, 838913.0, 866720.0, ...] : real list
```

The append (@) can be eliminated by accumulating the sorted result in a second argument. This version of quick sort, which is left as an exercise, takes only about 110 msec.

Like its procedural counterpart, *quick* takes order $n \log n$ time in the average case. If the input is already in increasing or decreasing order, then quick sort takes order n^2 time.

Exercise 3.38 Express quick sort such that *quicker(xs, sorted)* accumulates the result in *sorted*, with no use of *append*.

Exercise 3.39 Write a function *find* such that *find(xs, i)* returns the *i*th smallest item in the list *xs*. This is called **selection**. Hoare's algorithm for selection is related to quick sort, and is much faster than sorting the list and returning the *i*th element.

Exercise 3.40 Generalize *find* above to *findrange(xs, i, j)*, returning the list of the *i*th to *j*th smallest items in the list *xs*.

3.21 Merge sort

Several algorithms work by merging sorted lists. The merging function repeatedly takes the smaller of the heads of two lists:

```
fun merge([], ys)      = ys : real list
  | merge(xs, [])     = xs
  | merge(x::xs, y::ys) =
      if x <= y then x::merge(xs, y::ys)
      else y::merge(x::xs, ys);
> val merge = fn : real list * real list -> real list
```

When sorting 10,000 items, the recursion in *merge* may be too deep for some ML systems. The fault lies with those ML systems, not with *merge*. As with *take* and *append*, the dominant cost is that of constructing the resulting list. An iterative merging function, although avoiding the deep recursion, would probably have to perform costly list reversals.

Merge sort can be **top-down** or **bottom-up**. Either way, merging is efficient only if the two lists have similar lengths. If a list has only one element, merging degenerates to insertion.

Top-down merge sort. In the top-down approach, the input list is divided into two roughly equal parts using *take* and *drop*. These are sorted recursively and the results merged.

```
fun tmergesort [] = []
  | tmergesort [x] = [x]
  | tmergesort xs =
      let val k = length xs div 2
      in merge (tmergesort (List.take(xs, k)),
               tmergesort (List.drop(xs, k)))
      end;
> val tmergesort = fn : real list -> real list
```

Unlike quick sort, the worst case execution time is order $n \log n$. But it is slower on average, taking about 290 msec to sort the 10,000 numbers. Its calls to *length*, *take* and *drop* scan the input list repeatedly. Here is one way to eliminate them:

```
fun tmergesort' xs =
  let fun sort (0, xs) = ([], xs)
      | sort (1, x::xs) = ([x], xs)
      | sort (n, xs) =
          let val (l1, xs1) = sort ((n+1) div 2, xs)
              val (l2, xs2) = sort (n div 2, xs1)
          in (merge (l1, l2), xs2)
          end
      val (l, _) = sort (length xs, xs)
  in l end;
```

Calling *sort* (n, xs) sorts the first n elements of xs and returns the remaining elements. One might expect this to be slow, since it builds so many pairs. But it needs only 200 msec to sort the random numbers. It is still slower than quick sort, but can be recommended as a simple and acceptably fast method.

Bottom-up merge sort. The basic bottom-up approach divides the input into lists of length 1. Adjacent pairs of lists are then merged, obtaining sorted lists of length 2, then 4, then 8 and so on. Finally one sorted list remains. This approach is easy to code but wasteful. Why should the 10,000 numbers be copied into a list of 10,000 singleton lists?

O'Keefe (1982) describes a beautiful way to merge the lists of various lengths simultaneously, never storing these lists in full.

$$\underline{\underline{A \ B \ C \ D \ E \ F \ G \ H \ I \ J \ K}}$$

The underlining shows how adjacent lists are merged. First A with B , then C with D , and now AB and CD have equal length and can be merged. O'Keefe accumulates the merges at all levels in one list. Rather than comparing the sizes of lists, he lets the count k of members determine how to add the next member. If k is even then there are two members of equal size s to merge. The resulting list is treated as member $k/2$ of size $2s$, which may cause further merging.

```
fun mergepairs ([l], k) = [l]
  | mergepairs (l1::l2::ls, k) =
    if k mod 2 = 1 then l1::l2::ls
    else mergepairs (merge (l1, l2)::ls, k div 2);
> val mergepairs = fn
> : real list list * int -> real list list
```

If $k = 0$ then *mergепairs* merges the entire list of lists into one list. Calling *sorting* (*xs*, [[]], 0) sorts the list *xs*. It takes 270 msec to sort the 10,000 numbers.

```
fun sorting([], ls, k) = hd(mergепairs(ls, 0))
  | sorting(x::xs, ls, k) =
    sorting(xs, mergепairs([x]::ls, k+1), k+1);
> val sorting = fn
> : real list * real list list * int -> real list
```

A *smooth* sort has a linear execution time (order n) if its input is nearly sorted, degenerating to $n \log n$ in the worst case. O’Keefe presents a ‘smooth applicative merge sort’ that exploits order in the input. Rather than dividing it into singleton lists, he divides the input into increasing runs. If the number of runs is independent of n (and so ‘nearly sorted’) then the execution time is linear.

The function *nextrun* returns the next increasing run from a list, paired with the list of unread items. (An imperative program would delete items as they were processed.) The run grows in reverse order, hence the call to *rev*.

```
fun nextrun(run, []) = (rev run, []: real list)
  | nextrun(run, x::xs) =
    if x < hd run then (rev run, x::xs)
    else nextrun(x::run, xs);
> val nextrun = fn
> : real list * real list -> real list * real list
```

Runs are repeatedly taken and merged.

```
fun samsorting([], ls, k) = hd(mergепairs(ls, 0))
  | samsorting(x::xs, ls, k) =
    let val (run, tail) = nextrun([x], xs)
    in samsorting(tail, mergепairs(run::ls, k+1), k+1)
    end;
> val samsorting = fn
> : real list * real list list * int -> real list
```

The main sorting function is

```
fun samsort xs = samsorting(xs, [[]], 0);
> val samsort = fn : real list -> real list
```

The algorithm is both elegant and efficient. Even for our random data, with its short runs, the execution time is 250 msec.



Historical notes. Sorting with lists is similar to sorting on tape. Random access is inefficient; data must be scanned in order, either forward or reverse. The fascinating but largely obsolete literature on tape sorting may contain useful ideas.

The technique used in *mergепairs* bears a strong resemblance to the oscillating sort developed during the 1960s (Knuth, 1973, §5.4.5).

Merge sorting is seldom done using arrays because it cannot be done in-place: it requires two arrays. Even so, its use was proposed as early as 1945; the idea of exploiting runs in the input is also quite old (Knuth, 1973, §5.2.4).

Exercise 3.41 Use the following function to code a new version of top-down merge sort, and measure its speed. Explain your findings, taking account of garbage collection time if you can measure it.

```
fun alts ([], xs, ys)      = (xs, ys)
  | alts ([x], xs, ys)    = (x::xs, ys)
  | alts (x::y::l, xs, ys) = alts (l, x::xs, y::ys);
```

Exercise 3.42 This is the same as the previous exercise, except that you should base the new sorting function on the following:

```
fun takedrop ([], n, xs)  = (xs, [])
  | takedrop (x::l, n, xs) =
    if n>0 then takedrop (l, n-1, x::xs)
    else (xs, x::l);
```

Exercise 3.43 Why call *sorting* (*xs*, [[]], 0) and not *sorting* (*xs*, [], 0)?

Exercise 3.44 Write a version of *samsort* that uses both increasing and decreasing runs.

Polynomial arithmetic

Computers were invented to perform numerical arithmetic. They excel at portraying data graphically. But sometimes nothing conveys more information than a symbolic formula. A graph of $E = mc^2$ is simply a straight line!

Computer algebra is concerned with automating symbolic mathematics, as used by scientists and engineers. Systems such as MACSYMA and REDUCE can do stupendous tasks involving differentiation, integration, power series expansions, etc. The most basic symbolic operations perform polynomial arithmetic. Even this is hard to do efficiently. We shall restrict ourselves to the simplest case of all, **univariate** polynomials. These are polynomials in one variable, say x ; they can be added and multiplied straightforwardly:

$$(x + 1) + (x^2 - 2) = x^2 + x - 1$$

$$(x + 1) \times (x^2 - 2) = x^3 + x^2 - 2x - 2$$

In developing a package for arithmetic on univariate polynomials, we touch upon the general problem of data representation. We implement addition and multiplication, using sorting ideas from the previous section. We arrive at another structure matching signature *ARITH* of Section 2.22. Finally we consider how to find greatest common divisors, a challenge even in the univariate case.

Our code will be fast enough to compute $(x^3 + 1)^{1000}$ in under two seconds. This demonstrates what can be accomplished with such simple tools as lists.

3.22 Representing abstract data

In Section 3.15 above we considered operations such as *union* and *subset* for finite sets. ML does not provide finite sets as a data structure; instead, we represent them by lists without repetitions. It is worth examining what this really involves. Although finite sets may seem trivial, they exhibit most of the issues involved in data representation.

A collection of abstract objects, here finite sets, is represented using a set of concrete objects, here certain lists. Every abstract object can be represented by at least one concrete object. There may be more than one: recall that $\{3, 4\}$ can be represented by $[3, 4]$ or $[4, 3]$. Some concrete objects, such as $[3, 3]$, represent no abstract object at all.

Operations on the abstract data are defined in terms of the representations. For example, the ML function *union* implements the abstract function \cup provided *union*(l, l') represents $A \cup A'$ for all lists l and l' that represent the sets A and A' , respectively. The ML predicate *subs* (which is an infix operator) implements the abstract relation \subseteq provided $l \text{ subs } l'$ equals *true* whenever $A \subseteq A'$ holds, for all l and l' that represent the sets A and A' . The equality relation is treated similarly; we do not expect equal sets to have equal representations.

These issues come up every time we use a computer, which ultimately represents all data in terms of zeros and ones. Some deeper issues can only be mentioned here. For example, the computer represents real numbers by floating point numbers, but most real numbers cannot be represented, and real arithmetic is implemented only approximately.

3.23 Representing polynomials

Now let us consider how to represent univariate polynomials of the form

$$a_n x^n + \cdots + a_0 x^0.$$

Since we only allow one variable, its name does not have to be stored. The **coefficients** a_n, \dots, a_0 in an abstract polynomial might be real numbers. But real arithmetic on a computer is approximate. We ought to represent coefficients

by rational numbers, pairs of integers with no common factor.⁴ But this would be a digression, and it really requires arbitrary-precision integers, which some ML systems lack. Therefore, strictly as an expedient, let us represent coefficients by the ML type *real*.

We could represent polynomials by the list of their coefficients, $[a_n, \dots, a_0]$. To see that this is unacceptable, consider the polynomial $x^{100} + 1$, and then consider squaring it! In a typical polynomial, most of the coefficients are zero. We need a *sparse* representation, in contrast to the *dense* representation that we used earlier for matrices.

Let us represent a polynomial by a list of pairs of the form (k, a_k) for every k such that a_k is non-zero. The pairs should appear in decreasing order in k ; this will help us collect terms having the same exponent. For example, $[(2, 1.0), (0, ~2.0)]$ represents $x^2 - 2$.

Our representation is better behaved than that of finite sets because every abstract polynomial has a unique representation. Two polynomials are equal just if their underlying lists are equal. But not every list of (integer, real) pairs represents a polynomial.

We ought to declare polynomials as an abstract type, hiding the underlying lists. For now, let us package the polynomial operations into a structure, following signature *ARITH*.

```
structure Poly =
  struct
    type t = (int*real) list;
    val zero = [];
    fun sum ...
    fun diff ...
    fun prod ...
    fun quo ...
  end;
```

Here t is the type that represents polynomials, and *zero* is the empty list, which represents the zero polynomial. The other components are described below.

3.24 Polynomial addition and multiplication

Computing the sum of two polynomials is like merging the corresponding lists; consider $(x^3 - x) + (2x^2 + 1) = x^3 + 2x^2 - x + 1$. But like terms must be combined, and zero terms cancelled; consider $(x^4 - x + 3) + (x - 5) = x^4 - 2$. The ML definition follows the method we should use on paper:

⁴ See Exercise 2.25 on page 63.


```

fun sum ([], us)                = us : t
  | sum (ts, [])                = ts
  | sum ((m, a) :: ts, (n, b) :: us) =
      if m > n then (m, a) :: sum (ts, (n, b) :: us)
      else if n > m then (n, b) :: sum (us, (m, a) :: ts)
      else (* m=n *)
          if a+b=0.0 then sum (ts, us)
          else (m, a+b) :: sum (ts, us);

```

The product of two polynomials is computed using the distributive law. Terms of one polynomial are multiplied against the other polynomial, and the results added:

$$\begin{aligned}
 (x^2 + 2x - 3) \times (2x - 1) &= x^2(2x - 1) + 2x(2x - 1) - 3(2x - 1) \\
 &= (2x^3 - x^2) + (4x^2 - 2x) + (-6x + 3) \\
 &= 2x^3 + 3x^2 - 8x + 3
 \end{aligned}$$

To implement this method, we first need a function to multiply a term by a polynomial:

```

fun termprod ((m, a), [])      = [] : t
  | termprod ((m, a), (n, b) :: ts) =
      (m+n, a*b) :: termprod ((m, a), ts);

```

The naïve multiplication algorithm closely follows the example above:

```

fun nprod ([], us)            = []
  | nprod ((m, a) :: ts, us) = sum (termprod ((m, a), us),
                                   nprod (ts, us));

```

Faster multiplication. Experiments show that *nprod* is too slow for large polynomials. It requires over two seconds and numerous garbage collections to compute the square of $(x^3 + 1)^{400}$. (Such large computations are typical of computer algebra.) The reason is that *sum* merges lists of greatly differing length. If *ts* and *us* have 100 terms each, then *termprod* $((m, a), us)$ has only 100 terms, while *nprod* (ts, us) could have as many as 10,000 terms. Their sum will have at most 10,100 terms, a growth of only 1%.

Merge sort inspires a faster algorithm. Divide one of the polynomials into equal halves, compute two products recursively, and add them. We form as many sums as before, but they are balanced: we compute $(p_1 + p_2) + (p_3 + p_4)$ instead of $p_1 + (p_2 + (p_3 + p_4))$. On average the summands are smaller, and each addition doubles the size of the result.

```

fun prod ([], us) = []
| prod ([(m, a)], us) = termprod ((m, a), us)
| prod (ts, us) =
  let val k = length ts div 2
  in sum (prod (List.take(ts, k), us),
         prod (List.drop(ts, k), us))
  end;

```

This is three times as fast as *nprod* for computing the square of $(x^3 + 1)^{400}$. The speedup appears to increase with larger polynomials.

Running some examples. Although we have not fully defined the structure *Poly* yet, suppose we have done so — including a function *show*, to display polynomials as strings. We compute the sum and product of $x + 1$ and $x^2 - 2$, which were shown at the start of this section:

```

val p1 = [(1, 1.0), (0, 1.0)]
and p2 = [(2, 1.0), (0, ~2.0)];

Poly.show (Poly.sum (p1, p2));
> "x^2 + x - 1.0" : string
Poly.show (Poly.prod (p1, p2));
> "x^3 + x^2 - 2.0x - 2.0" : string

```

Structure *Poly* also provides exponentiation. The function *power* is defined as in Section 2.14. For a larger example, let us compute $(x - 1)^{10}$:

```

val xminus1 = [(1, 1.0), (0, ~1.0)];

Poly.show (Poly.power (xminus1, 10));
> "x^10 - 10.0x^9 + 45.0x^8 - 120.0x^7 + 210.0x^6
> - 252.0x^5 + 210.0x^4 - 120.0x^3 + 45.0x^2
> - 10.0x + 1.0" : string

```

Here are the first few terms of $(x^2 - 2)^{150}$:

```

Poly.show (Poly.power (p2, 150));
> "x^300 - 300.0x^298 + 44700.0x^296
> - 4410400.0x^294 + 324164400.0x^292..." : string

```

Exercise 3.45 Code *diff*, a function that computes the difference between two polynomials. Using *termprod* it can be coded in one line, but not efficiently.

Exercise 3.46 Code *show*, the function that produces the output shown in the text. (Functions *Real.toString* and *Int.toString* convert numbers to strings.)

Exercise 3.47 Give a convincing argument that *sum* and *prod* respect the representation of polynomials.

Exercise 3.48 What does it mean to say that *sum* correctly computes the sum of two polynomials? How might you prove it?

3.25 The greatest common divisor

Many applications involve **rational functions**: fractions over polynomials. Efficiency demands that a fraction's numerator and denominator should have no common factor. Therefore, we need a function to compute the greatest common divisor (GCD) of two polynomials. This requires functions to compute polynomial quotients and remainders.

Polynomial division. The algorithm for polynomial division resembles ordinary long division. It is actually easier, for it requires no guessing. Each step removes the leading term of the dividend by dividing the leading term of the divisor into it. Each such step yields a term of the quotient. Let us divide $2x^2 + x - 3$ by $x - 1$:

$$\begin{array}{r} 2x + 3 \\ x - 1 \overline{) 2x^2 + x - 3} \\ \underline{2x^2 - 2x} \\ 3x - 3 \\ \underline{3x - 3} \\ 0 \end{array}$$

Here the remainder is zero. In the general case, the remainder is a polynomial whose leading exponent (called its **degree**) is less than that of the divisor. Let us divide $x^3 + 2x^2 - 3x + 1$ by $x^2 + x - 2$, obtaining a remainder of $-2x + 3$:

$$\begin{array}{r} x + 1 \\ x^2 + x - 2 \overline{) x^3 + 2x^2 - 3x + 1} \\ \underline{x^3 + x^2 - 2x} \\ x^2 - x + 1 \\ \underline{x^2 + x - 2} \\ -2x + 3 \end{array}$$

If the divisor's leading coefficient is not unity then fractions will probably appear. This can make the computation much slower, but it does not complicate the basic algorithm. The function *quoem* directly implements the method sketched above, returning the pair (quotient, remainder). The quotient forms in reverse.

```

fun quorem (ts, (n, b)::us) =
  let fun dividing ([], qs) = (rev qs, [])
      | dividing ((m, a)::ts, qs) =
          if m < n then (rev qs, (m, a)::ts)
          else dividing (sum (ts, termprod ((m-n, ~a/b), us)),
                        (m-n, a/b) :: qs)
  in dividing (ts, []) end;

```

Dividing $x^2 - 2$ by $x - 1$ yields a quotient of $x + 1$ and a remainder of -1 , since $(x - 1) \times (x + 1) = x^2 - 1$. Assume that *Poly* includes *quorem*, and also a function *showpair* for displaying a pair of polynomials:

```

Poly.quorem (p2, xminus1);
> ([ (1, 1.0), (0, 1.0) ], [ (0, ~1.0) ])
> : (int * real) list * (int * real) list
Poly.showpair it;
> "x + 1.0,      - 1.0" : string

```

Let us run the second example of division shown above.

```

Poly.showpair
  (Poly.quorem ([ (3, 1.0), (2, 2.0), (1, ~3.0), (0, 1.0) ],
                [ (2, 1.0), (1, 1.0), (0, ~2.0) ]));
> "x + 1.0,      - 2.0x + 3.0" : string

```

We can trivially define the quotient function *quo* in terms of *quorem*. Including the *diff* function of Exercise 3.45 gives structure *Poly* all the components it needs to match signature *ARITH*. Our collection of arithmetic structures now includes the complex numbers, binary numerals, matrices and polynomials! But note, again, that there are important differences among these structures; for each there are additional components that would not make sense for the others. Although *Poly* can match signature *ARITH* when we need it to, normally we take full advantage of its other components.

Euclid's Algorithm for polynomials. We can now compute GCDs using Euclid's Algorithm. Recall that #2 extracts the second component of a pair, here the remainder after polynomial division.

```

fun gcd ([], us) = us
  | gcd (ts, us) = gcd (#2 (quorem (us, ts)), ts);

```

Assuming that *Poly* contains *gcd* as a component, we can try it out on some examples. The GCD of $x^8 - 1$ and $x^3 - 1$ is $x - 1$:

```

Poly.show (Poly.gcd ([ (8, 1.0), (0, ~1.0) ],
                    [ (3, 1.0), (0, ~1.0) ]));
> "x - 1.0" : string

```

The GCD of $x^2 + 2x + 1$ and $x^2 - 1$ is ... $-2x - 2$?

```
Poly.show (Poly.gcd ([ (2, 1.0), (1, 2.0), (0, 1.0) ],
                    [ (2, 1.0), (0, ~1.0) ]));
> " - 2.0x - 2.0" : string
```

The GCD ought to be $x + 1$. This particular difficulty can be solved by dividing through by the leading coefficient, but there are other problems. We must use rational arithmetic: see the warning below. Then computing the GCD often requires operating on enormous integers, even if the initial polynomials have only single-digit coefficients. Many further refinements, typically using modular arithmetic, are essential.



Beware rounding errors. The use of floating point arithmetic (type *real*) is especially bad in *gcd*, because its first line tests for a remainder of zero. Rounding errors during division could give the remainder a small but non-zero coefficient; a common divisor would be missed. For other applications, polynomial arithmetic is fairly well-behaved, because the effects of rounding errors are predictable. (I am grateful to James Davenport for these observations.)



Further reading. Davenport *et al.* (1993) is an excellent introduction to computer algebra. Chapter 2 covers data representation. We learn, for example, that multivariate polynomials can be represented as univariate polynomials whose coefficients are themselves univariate polynomials in some other variable. Thus $y^2 + xy$ is a univariate polynomial in y whose coefficients are 1 and the polynomial x ; we could exchange the rôles of x and y . Chapter 4 covers the horrendous complications involved in computing GCDs efficiently.

Summary of main points

- Lists are constructed from the empty list (*nil* or `[]`), using `::` ('cons') to attach elements to the front.
- Important library functions include *length*, *take*, *drop*, the infix operator `@` (concatenation) and *rev* (reverse).
- A recursive function, by accumulating its results in an extra argument, may avoid inefficient list concatenation.
- Equality polymorphism allows functions to perform equality testing on their arguments. Examples include *mem*, which tests membership in a list, and *assoc*, which searches in a list of pairs.
- The usual sorting algorithms can be implemented using lists, with surprising efficiency.
- Lists can represent binary numerals, matrices, graphs, polynomials, etc., allowing operations to be expressed concisely.