8
Imperative Programming in ML

Functional programming has its merits, but imperative programming is here to stay. It is the most natural way to perform input and output. Some programs are specifically concerned with managing state: a chess program must keep track of where the pieces are! Some classical data structures, such as hash tables, work by updating arrays and pointers.

Standard ML's imperative features include references, arrays and commands for input and output. They support imperative programming in full generality, though with a flavour unique to ML. Looping is expressed by recursion or using a `while` construct. References behave differently from Pascal and C pointers; above all, they are secure.

Imperative features are compatible with functional programming. References and arrays can serve in functions and data structures that exhibit purely functional behaviour. We shall code sequences (lazy lists) using references to store each element. This avoids wasteful recomputation, which is a defect of the sequences of Section 5.12. We shall code functional arrays (where updating creates a new array) with the help of mutable arrays. This representation of functional arrays can be far more efficient than the binary tree approach of Section 4.15.

A typical ML program is largely functional. It retains many of the advantages of functional programming, including readability and even efficiency: garbage collection can be faster for immutable objects. Even for imperative programming, ML has advantages over conventional languages.

Chapter outline

This chapter describes reference types and arrays, with examples of their use in data structures. ML's input and output facilities are presented.

The chapter contains the following sections:

Reference types. References stand for storage locations and can be created, updated and inspected. Polymorphic references cannot be created, but polymorphic functions can use references.

References in data structures. Three large examples are presented. We modify our type of sequences to store computed elements internally. Ring buffers illustrate how references can represent linked data structures. V-arrays exploit imperative programming techniques in a functional data structure.

Input and output. Library functions convert between strings and basic types, such as $real$. Channels, carrying streams of characters, connect an ML program to input and output devices. Examples include date scanning, conversion to HTML, and pretty printing.

Reference types

References in ML are essentially store addresses. They correspond to the variables of C, Pascal and similar languages, and serve as pointers in linked data structures. For control structures, ML provides `while-do` loop commands; the `if-then-else` and `case` expressions also work for imperative programming. The section concludes by explaining the interaction between reference types and polymorphism.

8.1 *References and their operations*

All values computed during the execution of an ML program reside for some time in the machine store. To functional programmers, the store is nothing but a device inside the computer; they never have to think about the store until they run out of it. With imperative programming the store is visible. An ML reference denotes the address of a location in the store. Each location contains a value, which can be replaced using an assignment. A reference is itself a value; if x has type τ then a reference to x is written *ref* x and has type τ *ref*.

The constructor *ref* creates references. When applied to a value v, it allocates a new address with v for its initial contents, and returns a reference to this address. Although *ref* is an ML function, it is not a function in the mathematical sense because it returns a new address every time it is called.

The function !, when applied to a reference, returns its contents. This operation is called ***dereferencing***. Clearly ! is not a mathematical function; its result depends upon the store.

The assignment $E_1 := E_2$ evaluates E_1, which must return a reference p, and E_2. It stores at address p the value of E_2. Syntactically, := is a function and $E_1 := E_2$ is an expression, even though it updates the store. Like most functions that change the machine's state, it returns the value () of type $unit$.

Here is a simple example of these primitives:

```
val p = ref 5 and q = ref 2;
```

```
> val p = ref 5 : int ref
> val q = ref 2 : int ref
```

The references p and q are declared with initial contents 5 and 2.

```
(!p,!q);
> (5, 2) : int * int
p := !p + !q;
> () : unit
(!p,!q);
> (7, 2) : int * int
```

The assignment changes the contents of p to 7. Note the word 'contents'! The assignment does not change the value of p, which is a fixed address in the store; it changes the contents of that address. We may use p and q like integer variables in Pascal, except that dereferencing is explicit. We must write `!p` to get the contents of p, since p by itself denotes an address.

References in data structures. Because references are ML values, they may belong to tuples, lists, etc.

```
val refs = [p,q,p];
> val refs = [ref 7, ref 2, ref 7] : int ref list
q := 1346;
> () : unit
refs;
> [ref 7, ref 1346, ref 7] : int ref list
```

The first and third elements of *refs* denote the same address as p, while the second element is the same as q. ML compilers print the value of a reference as *ref* c, where c is its contents, rather than printing the address as a number. So assigning to q affects how *refs* is printed. Let us assign to the head of the list:

```
hd refs := 1415;
> () : unit
refs;
> [ref 1415, ref 1346, ref 1415] : int ref list
(!p,!q);
> (1415, 1346) : int * int
```

Because the head of *refs* is p, assigning to *hd refs* is the same as assigning to p.

References to references are also allowed:

```
val refp = ref p and refq = ref q;
> val refp = ref (ref 1415) : int ref ref
> val refq = ref (ref 1346) : int ref ref
```

The assignment below updates the contents (q) of *refq* with the contents (1415) of the contents (p) of $refp$. Here $refp$ and $refq$ behave like Pascal pointer variables.

```
!refq := !(!refp);
> () : unit
(!p,!q);
> (1415, 1415) : int * int
```

Equality of references. The ML equality test is valid for all reference types. Two references of the same type are equal precisely if they denote the same address. The following tests verify that p and q are distinct references, and that the head of $refs$ equals p, not q:

```
p=q;
> false : bool
hd refs = p;
> true : bool
hd refs = q;
> false : bool
```

In Pascal, two pointer variables are equal if they happen to contain the same address; an assignment makes two pointers equal. The ML notion of reference equality may seem peculiar, for if p and q are distinct references then nothing can make them equal (short of redeclaring them). In imperative languages, where all variables can be updated, a pointer variable really involves two levels of reference. The usual notion of pointer equality is like comparing the contents of $refp$ and $refq$, which are references to references:

```
!refp = !refq;
> false : bool
refq := p;
> () : unit
!refp = !refq;
> true : bool
```

At first, $refp$ and $refq$ contain different values, p and q. Assigning the value p to $refq$ makes $refp$ and $refq$ have the same contents; both 'pointer variables' refer to p.

When two references are equal, like p and $hd\ refs$, assigning to one affects the contents of the other. This situation, called ***aliasing***, can cause great confusion. Aliasing can occur in procedural languages; in a procedure call, a global variable and a formal parameter may denote the same address.

Cyclic data structures. Circular chains of references arise in many situations. Suppose that we declare cp to refer to the successor function on the integers, and dereference it in the function $cFact$.

```
val cp = ref (fn k => k+1);
> val cp = ref fn : (int -> int) ref
fun cFact n =  if n=0  then  1  else  n * !cp(n-1);
> val cFact = fn : int -> int
```

Each time *cFact* is called, it takes the current contents of *cp*. Initially this is the successor function, and $cFact(8) = 8 \times 8 = 64$:

```
cFact 8;
> 64 : int
```

Let us update *cp* to contain *cFact*. Now *cFact* refers to itself via *cp*. It becomes a recursive function and computes factorials:

```
cp := cFact;
> () : unit
cFact 8;
> 40320 : int
```

Updating a reference to create a cycle is sometimes called 'tying the knot.' Many functional language interpreters implement recursive functions exactly as shown above, creating a cycle in the execution environment.

Exercise 8.1 True or false: if $E_1 = E_2$ then $ref\ E_1 = ref\ E_2$.

Exercise 8.2 Declare the function `+:=` such that `+:=` *Id E* has the same effect as *Id* `:=` `!`*Id* `+` *E*, for integer *E*.

Exercise 8.3 With *p* and *q* declared as above, explain ML's response when these expressions are typed at top level:

```
p:=!p+1        2*!q
```

8.2 *Control structures*

ML does not distinguish commands from expressions. A command is an expression that updates the state when evaluated. Most commands have type *unit* and return (). Viewed as an imperative language, ML provides only basic control structures.

The conditional expression

```
if E then E1 else E2
```

can be viewed as a conditional command. It evaluates E, possibly updating the state. If the resulting boolean value is *true* then it executes E_1; otherwise it executes E_2. It returns the value of E_1 or E_2, though with imperative programming that value is normally ().

Note that this behaviour arises from ML's treatment of expressions in general; ML has only one `if` construct.

Similarly, the `case` expression can serve as a control structure:

`case` E `of` P_1 `=>` E_1 `|` $\cdots$ `|` P_n `=>` E_n

First E is evaluated, perhaps changing the state. Then pattern-matching selects some expression E_i in the usual way. It is evaluated, again perhaps changing the state, and the resulting value is returned.

In the function call $E_1\ E_2$ and the n-tuple $(E_1, E_2, \ldots, E_n)$, the expressions are evaluated from left to right. If E_1 changes the state, it could affect the outcome of E_2.

A series of commands can also be executed by the expression

$$(E_1;\ E_2;\ \ldots;\ E_n)$$

When this expression is evaluated, the expressions E_1, E_2, ..., E_n are evaluated from left to right. The result is the value of E_n; the values of the other expressions are discarded. Because of the other uses of the semicolon in ML, this construct must always be enclosed in parentheses unless it forms the body of a `let` expression:

`let` D `in` $E_1; E_2; \ldots; E_n$ `end`

For iteration, ML has a `while` command:

`while` E_1 `do` E_2

If E_1 evaluates to *false* then the `while` is finished; if E_1 evaluates to *true* then E_2 is evaluated and the `while` is executed again. To be precise, the `while` command satisfies the recursion

$$\texttt{while}\ E_1\ \texttt{do}\ E_2 \equiv \texttt{if}\ E_1\ \texttt{then}\ (E_2;\ \texttt{while}\ E_1\ \texttt{do}\ E_2)\ \texttt{else}\ ()$$

The value returned is (), so E_2 is evaluated just for its effect on the state.

Simple examples. ML can imitate procedural programming languages. The following procedures, apart from the explicit dereferencing (the `!` operation), could have been written in Pascal or C. The function *impFact* computes factorials using local references *resultp* and *ip*, returning the final contents of *resultp*. Observe the use of a `while` command to execute the body n times:

```
fun impFact n =
  let val resultp = ref 1
      and ip      = ref 0
  in  while !ip < n do (ip      := !ip + 1;
                        resultp := !resultp * !ip);
      !resultp
  end;
> val impFact = fn : int -> int
```

The body of the `while` contains two assignments. At each iteration it adds one to the contents of *ip*, then uses the new contents of *ip* to update the contents of *resultp*.

Although calling *impFact* allocates new references, this state change is invisible outside. The value of *impFact*(E) is a mathematical function of the value of E.

```
impFact 6;
> 720 : int
```

In procedural languages, a procedure may have reference parameters in order to modify variables in the calling program. In Standard ML, a reference parameter is literally a formal parameter of reference type. We can transform *impFact* into a procedure *pFact* that takes *resultp* as a reference parameter.

```
fun pFact (n, resultp) =
  let val ip = ref 0
  in  resultp := 1;
      while !ip < n do (ip      := !ip + 1;
                        resultp := !resultp * !ip)
  end;
> val pFact = fn : int * int ref -> unit
```

Calling *pFact*(n, *resultp*) assigns the factorial of n to *resultp*:

```
pFact (5,p);
> () : unit
p;
> ref 120 : int ref
```

These two functions demonstrate the imperative style, but a pure recursive function is the clearest and probably the fastest way to compute factorials. More realistic imperative programs appear later in this chapter.

Supporting library functions. The standard library declares some top level functions for use in imperative programs. The function *ignore* ignores its argument and returns (). Here is a typical situation:

```
if !skip then ignore (TextIO.inputLine file)
        else skip := true;
```

The input/output command returns a string, while the assignment returns (). Calling *ignore* discards the string, preventing a clash between types *string* and *unit*. The argument to *ignore* is evaluated only for its side-effects, here to skip the next line of a file.

Sometimes we must retain an expression's value before executing some command. For instance, if x contains 0.5 and y contains 1.2, we could exchange their contents like this:

```
y := #1 (!x, x := !y);
> () : unit
(!x, !y);
> (1.2, 0.5) : real * real
```

The exchange works because the arguments of the pair are evaluated in order. The function `#1` returns the first component,[1] which is the original contents of x. The library infix *before* provides a nicer syntax for this trick. It simply returns its first argument.

```
y := (!x before x := !y);
```

The list functional *app* applies a command to every element of a list. For example, here is a function to assign the same value to each member of a list of references:

```
fun initialize rs x = app (fn r => r:=x) rs;
> val initialize = fn : 'a ref list -> 'a -> unit
initialize refs 1815;
> () : unit
refs;
> [ref 1815, ref 1815, ref 1815] : int ref list
```

Clearly *app f l* is similar to *ignore* (*map f l*), but avoids building a list of results. The top level version of *app* comes from structure *List*. Other library structures, including *ListPair* and *Array*, declare corresponding versions of *app*.

Exceptions and commands. When an exception is raised, the normal flow of execution is interrupted. An exception handler is chosen, as described in Section 4.8, and control resumes there. This could be dangerous; an exception

[1] Section 2.9 explains selector functions of the form #k.

could occur at any time, leaving the state in an abnormal condition. The following exception handler traps any exception, tidies up the state, and re-raises the exception. The variable e is a trivial pattern (of type exn) to match all exceptions:

```
handle e => (...(*tidy up actions*)...; raise e)
```

Note: Most commands return the value () of type $unit$. From now on, our sessions will omit the boring response

```
> () : unit
```

Exercise 8.4 Expressions $(E_1; E_2; \ldots; E_n)$ and `while` E_1 `do` E_2 are derived forms in ML, which means they are defined by translation to other expressions. Describe suitable translations.

Exercise 8.5 Write an imperative version of the function $sqroot$, which computes real square roots by the Newton-Raphson method (Section 2.17).

Exercise 8.6 Write an imperative version of the function fib, which computes Fibonacci numbers efficiently (Section 2.15).

Exercise 8.7 The simultaneous assignment

$$V_1,\, V_2, \ldots,\, V_n := E_1, E_2, \ldots, E_n$$

first evaluates the expressions, then assigns their values to the corresponding references. For instance $x,\, y := !y, !x$ exchanges the contents of x and y. Write an ML function to perform simultaneous assignments. It should have the polymorphic type $(\alpha\ ref)list \times \alpha\ list \to unit$.

8.3 *Polymorphic references*

References have been a notorious source of insecurity ever since they were introduced to programming languages. Often, no type information was kept about the contents of a reference; a character code could be interpreted as a real number. Pascal prevents such errors, ensuring that each reference contains values of one fixed type, by having a distinct type ‘pointer to τ’ for each type τ. In ML, the problem is harder: what does the type $\tau\ ref$ mean if τ is polymorphic? Unless we are careful, the contents of this reference could change over time.

An imaginary session. This illegal session demonstrates what could go wrong if references were naïvely added to the type system. We begin by declaring the identity function:

```
fun I x = x;
> val I = fn : 'a -> 'a
```

Since I is polymorphic, it may be applied to arguments of any types. Now let us create a reference to I:

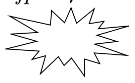
```
val fp = ref I;
> val fp = ref fn : ('a -> 'a) ref
```

With its polymorphic type $(\alpha \to \alpha)\mathit{ref}$, we should be able to apply the contents of *fp* to arguments of any types:

```
(!fp true, !fp 5);
> (true, 5) : bool * int
```

And its polymorphic type lets us assign a function of type $\mathit{bool} \to \mathit{bool}$ to *fp*:

```
fp := not;
!fp 5;
```



Applying *not* to the integer 5 is a run-time type error, but ML is supposed to detect all type errors at compile-time. Obviously something has gone wrong, but where?

Polymorphism and substitution. In the absence of imperatives, evaluating an expression repeatedly always yields the same result. The declaration `val` *Id* `=` *E* makes *Id* a synonym for this one result. Ignoring efficiency, we could just as well substitute *E* for *Id* everywhere.

For example, here are two polymorphic declarations, of a function and a list of lists:

```
let val I = fn x => x  in  (I true, I 5) end;
> (true, 5) : bool * int
let val nill = [[]]  in  (["Exeter"]::nill, [1415]::nill) end;
> ([["Exeter"], []], [[1415], []])
> : string list list * int list list
```

Substituting the declarations away affects neither the value returned nor the typing:

```
((fn x => x) true, (fn x => x) 5);
```

```
> (true, 5) : bool * int
(["Exeter"]::[[]], [1415]::[[]]);
> ([["Exeter"], []], [[1415], []])
> : string list list * int list list
```

Now let us see how ML reacts to our imaginary session above, when packaged as a `let` expression:

```
let val fp = ref I
in  ((!fp true, !fp 5), fp := not, !fp 5)  end;
> Error: Type conflict: expected int, found bool
```

ML rejects it, thank heavens — and with a meaningful error message too. What happens if we substitute the declarations away?

```
((!(ref I) true, !(ref I) 5), (ref I) := not, !(ref I) 5);
> ((true, 5), (), 5) : (bool * int) * unit * int
```

The expression is evaluated without error. But the substitution has completely altered its meaning. The original expression allocates a reference *fp* with initial contents *I*, extracts its contents twice, updates it and finally extracts the new value. The modified expression allocates four different references, each with initial contents *I*. The assignment is pointless, updating a reference used nowhere else.

The crux of the problem is that repeated calls to *ref* always yield different references. We declare `val` *fp* = *ref I* expecting that each occurrence of *fp* will denote the same reference: the same store address. Substitution does not respect the sharing of *fp*. Polymorphism treats each identifier by substituting the type of its defining expression, thereby assuming that substitution is valid.

The culprit is sharing, not side effects. We must regulate the creation of polymorphic references, not assignments to them.

Polymorphic value declarations. ***Syntactic values*** are expressions that are too simple to create references. They come in several forms:

- A literal constant such as 3 is a syntactic value.
- An identifier is one also, as it refers to some other declaration that has been dealt with already.
- A syntactic value can be built up from others using tupling, record notation and constructors (excluding *ref*, of course).
- A function in `fn` notation is a syntactic value, even if its body uses *ref*, as the body is not executed until the function is called.

Calls to *ref* and other functions are not syntactic values.

If E is a syntactic value then the polymorphic declaration `val` *Id* `=` E is equivalent to a substitution. The declaration is polymorphic in the usual way: each occurrence of *Id* may have a different instance of E's type. Every `fun` declaration is treated like this, for it is shorthand for a `val` declaration with `fn` notation, which is a syntactic value.

If E is not a syntactic value then the declaration `val` *Id* `=` E might create references. To respect sharing, each occurrence of *Id* must have the same type. If the declaration occurs inside a `let` expression, then each type variable in the type of E is frozen throughout the `let` body. The expression

```
let val fp = ref I
in  fp := not; !fp 5  end;
```

is illegal because *ref I* involves the type variable α, which cannot stand for *bool* and *int* at the same time. The expression

```
let val fp = ref I
in  (!fp true, !fp 5)  end;
```

is illegal for the same reason. Yet it is safe: if we could evaluate it, the result would be (*true*, 5) with no run-time error. A monomorphic version is legal:

```
let val fp = ref I
in  fp := not; !fp true  end;
> false : bool
```

A top level polymorphic declaration is forbidden unless E is a syntactic value; the type checker cannot predict how future occurrences of *Id* will be used:

```
val fp = ref I;
> Error: Non-value in polymorphic declaration
```

A monomorphic type constraint makes the top level declaration legal. The expression no longer creates polymorphic references:

```
val fp = ref (I: bool -> bool);
> val fp = ref fn : (bool -> bool) ref
```

Imperative list reversal. We now consider an example with real polymorphism. The function *irev* reverses a list imperatively. It uses one reference to scan down the list and another to accumulate the elements in reverse.

```
fun irev l =
  let val resultp = ref []
      and lp      = ref l
  in  while not (null (!lp)) do
          (resultp := hd(!lp) :: !resultp;
           lp      := tl(!lp));
      !resultp
  end;
> val irev = fn : 'a list -> 'a list
```

The variables *lp* and *resultp* have type $(\alpha\ list)ref$; the type variable α is frozen in the body of the `let`. ML accepts *irev* as a polymorphic function because it is declared using `fun`.

As we can verify, *irev* is indeed polymorphic:

```
irev [25,10,1415];
> [1415, 10, 25] : int list
irev (explode("Montjoy"));
> [#"y", #"o", #"j", #"t", #"n", #"o", #"M"]
> : char list
```

It can be used exactly like the standard function *rev*.

Polymorphic exceptions. Although exceptions do not involve the store, they require a form of sharing. Consider the following nonsense:

```
exception Poly of 'a;          (* illegal!! *)
(raise Poly true) handle Poly x => x+1;
```

If this expression could be evaluated, it would attempt to evaluate $true + 1$, a run-time error. When a polymorphic exception is declared, ML ensures that it is used with only one type, just like a restricted value declaration. The type of a top level exception must be monomorphic and the type variables of a local exception are frozen.

Limitations of value polymorphism. As noted in Section 5.4, the restriction to syntactic values bans some natural polymorphic declarations. In most cases they can be corrected easily, say by using `fun` instead of `val`:

```
val length   = foldl (fn (_,n) => n+1) 0;     (*rejected*)
fun length l = foldl (fn (_,n) => n+1) 0 l;   (*accepted*)
```

Compile-time type checking must make conservative assumptions about what could happen at run-time. Type checking rejects many programs that could execute safely. The expression $hd[5, true] + 3$ evaluates safely to 8 despite being ill-typed. Most modern languages employ compile-time type checking;

programmers accept these restrictions in order to be free from type errors at run-time.

The history of polymorphic references. Many people have studied polymorphic references, but Mads Tofte is generally credited with cracking the problem. Early ML compilers forbade polymorphic references altogether: the function *ref* could have only monomorphic types. Tofte's original proposal, adopted in the ML *Definition*, was more liberal than the one used now. Special 'weak' type variables tracked the use of imperative features, and only they were restricted in `val` declarations. Standard ML of New Jersey used an experimental approach where weak type variables had numerical degrees of weakness.

Weak type variables result in complicated, unintuitive types. They assign different types to *irev* and *rev*, and prevent their being used interchangeably. Worst of all, they make it hard to write signatures before implementing the corresponding structures.

Wright (1995) proposed treating all `val` declarations equally — in effect, making all type variables weak. Purely functional code would be treated like imperative code. Could programmers tolerate such restrictions? Using a modified type checker, Wright examined an immense body of ML code written by others. The restrictions turned out to cause very few errors, and those could be repaired easily. And so, after due consideration, this proposal has been recommended for ML.

The type checking of polymorphic references in Standard ML is probably safe. Tofte (1990) proved its correctness for a subset of ML and there is no reason to doubt that it is correct for the full language. Greiner (1996) has investigated a simplification of the New Jersey system. Harper (1994) describes a simpler approach to such proofs. Standard ML has been defined with great care to avoid insecurities and other semantic defects; in this regard, the language is practically in a class by itself.

Exercise 8.8 Is this expression legal? What does *WI* do?

```
let fun WI x = !(ref x)
in  (WI false, WI "Clarence")  end
```

Exercise 8.9 Which of these declarations are legal? Which could, if evaluated, lead to a run-time type error?

```
val funs = [hd];
val l    = rev [];
val l'   = tl [3];
val lp   = let fun nilp x = ref []  in  nilp()  end;
```

References in data structures

Any textbook on algorithms describes recursive data structures such as lists and trees. These differ from ML recursive datatypes (as seen in Chapter 4) in one major respect: the recursion involves explicit link fields, or pointers. These pointers can be updated, allowing re-linking of existing data structures and the creation of cycles.

Reference types, in conjunction with recursive datatypes, can implement such linked data structures. This section presents two such examples: doubly-linked circular lists and a highly efficient form of functional array. We begin with a simpler use of references: not as link fields, but as storage for previously computed results.

8.4 *Sequences, or lazy lists*

Under the representation given in Section 5.12, the tail of a sequence is a function to compute another sequence. Each time the tail is inspected, a possibly expensive function call is repeated. This inefficiency can be eliminated. Represent the tail of a sequence by a reference, which initially contains a function and is later updated with the function's result. Sequences so implemented exploit the mutable store, but when viewed from outside are purely functional.

An abstract type of sequences. Structure *ImpSeq* implements lazy lists; see Figure 8.1 on the next page. Type $\alpha\ t$ has three constructors: *Nil* for the empty sequence, *Cons* for non-empty sequences, and *Delayed* to permit delayed evaluation of the tail. A sequence of the form

$$Cons(x,\ ref(Delayed\ xf)),$$

where *xf* has type $unit \to \alpha\ t$, begins with x and has the sequence $xf()$ for its remaining elements. Note that *Delayed xf* is contained in a reference cell. Applying *force* updates it to contain the value of $xf()$, removing the *Delayed*. Some overhead is involved, but if the sequence element is revisited then there will be a net gain in efficiency.

The function *null* tests whether a sequence is empty, while *hd* and *tl* return the head and tail of a sequence. Because *tl* calls *force*, a sequence's outer constructor cannot be *Delayed*. Inside structure *ImpSeq*, functions on sequences may exploit pattern-matching; outside, they must use *null*, *hd* and *tl* because the constructors are hidden. An opaque signature constraint ensures that the structure yields an abstract type:

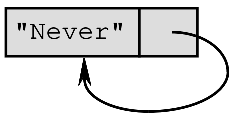
```
signature IMP_SEQUENCE =
  sig
  type 'a t
  exception Empty
  val empty    : 'a t
  val cons     : 'a * (unit -> 'a t) -> 'a t
```

Figure 8.1 *Lazy lists using references*

```
structure ImpSeq :> IMP_SEQUENCE =
  struct
  datatype 'a t = Nil
                | Cons    of 'a * ('a t) ref
                | Delayed of unit -> 'a t;

  exception Empty;

  fun delay xf = ref(Delayed xf);

  val empty = Nil;

  fun cons(x,xf) = Cons(x, delay xf);

  fun force xp =
          case !xp of
            Delayed f => let val s = f()
                         in  xp := s;  s  end
          | s => s;

  fun null Nil        = true
    | null (Cons _) = false;

  fun hd Nil            = raise Empty
    | hd (Cons(x,_)) = x;

  fun tl Nil            = raise Empty
    | tl (Cons(_,xp)) = force xp;

  fun take (xq, 0)            = []
    | take (Nil, n)           = []
    | take (Cons(x,xp), n) = x :: take (force xp, n-1);

  fun           Nil @ yq = yq
    | (Cons(x,xp)) @ yq =
          Cons(x, delay(fn()=> (force xp) @ yq));

  fun map f Nil            = Nil
    | map f (Cons(x,xp)) =
          Cons(f x, delay(fn()=> map f (force xp)));

  fun cycle seqfn =
      let val knot = ref Nil
      in  knot := seqfn (fn()=> !knot);  !knot  end;
  end;
```

```
val null       : 'a t -> bool
val hd         : 'a t -> 'a
val tl         : 'a t -> 'a t
val take       : 'a t * int -> 'a list
val toList     : 'a t -> 'a list
val fromList   : 'a list -> 'a t
val @          : 'a t * 'a t -> 'a t
val interleave : 'a t * 'a t -> 'a t
val concat     : 'a t t -> 'a t
val map        : ('a -> 'b) -> 'a t -> 'b t
val filter     : ('a -> bool) -> 'a t -> 'a t
val cycle      : ((unit -> 'a t) -> 'a t) -> 'a t
end;
```

Cyclic sequences. The function *cycle* creates cyclic sequences by tying the knot. Here is a sequence whose tail is itself:



This behaves like the infinite sequence `"Never"`, `"Never"`, ..., but occupies a tiny amount of space in the computer. It is created by

```
ImpSeq.cycle(fn xf => ImpSeq.cons("Never", xf));
> - : string ImpSeq.t
ImpSeq.take(5, it);
> ["Never", "Never", "Never", "Never", "Never"]
> : string list
```

When *cycle* is applied to some function *seqfn*, it creates the reference *knot* and supplies it to *seqfn* (packaged as a function). The result of *seqfn* is a sequence that, as its elements are computed, eventually refers to the contents of *knot*. Updating *knot* to contain this very sequence creates a cycle.

Cyclic sequences can compute Fibonacci numbers in an amusing fashion. Let *add* be a function that adds two sequences of integers, returning a sequence of sums. To illustrate reference polymorphism, *add* is coded in terms of a function to join two sequences into a sequence of pairs:

```
fun pairs(xq,yq) =
      ImpSeq.cons((ImpSeq.hd xq, ImpSeq.hd yq),
                  fn()=>pairs(ImpSeq.tl xq, ImpSeq.tl yq));
> val pairs = fn
> : 'a ImpSeq.t * 'b ImpSeq.t -> ('a * 'b) ImpSeq.t
fun add (xq,yq) = ImpSeq.map Int.+ (pairs(xq,yq));
> val add = fn
> : int ImpSeq.t * int ImpSeq.t -> int ImpSeq.t
```

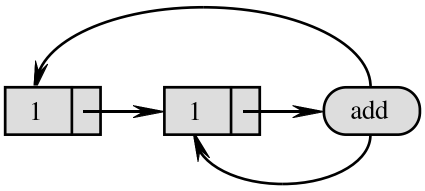
The sequence of Fibonacci numbers can be defined using *cycle*:

```
val fib = ImpSeq.cycle(fn fibf =>
    ImpSeq.cons(1, fn()=>
         ImpSeq.cons(1, fn()=>
              add(fibf(), ImpSeq.tl(fibf())))));
> val fib = - : int ImpSeq.t
```

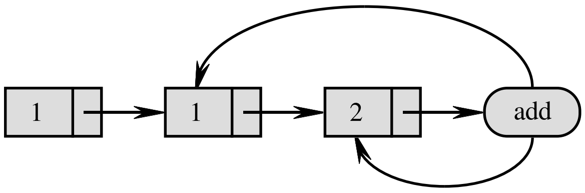
This definition is cyclic. The sequence begins 1, 1, and the remaining elements are obtained by adding the sequence to its tail:

$$add(fib,\ ImpSeq.tl\ fib)$$

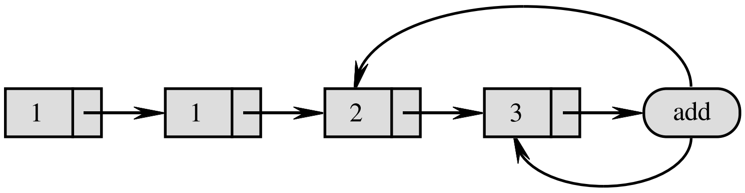
Initially, *fib* can be portrayed as follows:



When the third element of *fib* is inspected by *tl* (*tl fib*), the *add* call computes a 2 and *force* updates the sequence as follows:

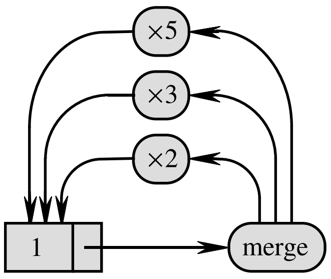


When the next element is inspected, *fib* becomes



Because the sequence is cyclic and retains computed elements, each Fibonacci number is computed only once. This is reasonably fast. If Fibonacci numbers were defined recursively using the sequences of Section 5.12, the cost of computing the nth element would be exponential in n.

Exercise 8.10 The Hamming problem is to enumerate all integers of the form $2^i 3^j 5^k$ in increasing order. Declare a cyclic sequence consisting of these numbers. Hint: declare a function to merge increasing sequences, and consider the following diagram:



Exercise 8.11 Implement the function *iterates*, which given f and x creates a cyclic representation of the sequence $[x, f(x), f(f(x)), \ldots, f^k(x), \ldots]$.

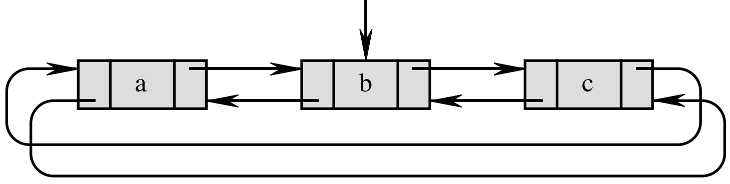
Exercise 8.12 Discuss the difficulty of showing whether a cyclic sequence is correct — that it generates a sequence of values satisfying a given specification. Comment on the following sequence:

```
val fib2 = ImpSeq.cycle(fn fibf =>
    ImpSeq.cons(1, fn()=> add(fibf(), ImpSeq.tl(fibf()))));
```

Exercise 8.13 Code the functions omitted from structure *ImpSeq* but specified in its signature, namely *toList*, *fromList*, *interleave*, *concat* and *filter*.

8.5 *Ring buffers*

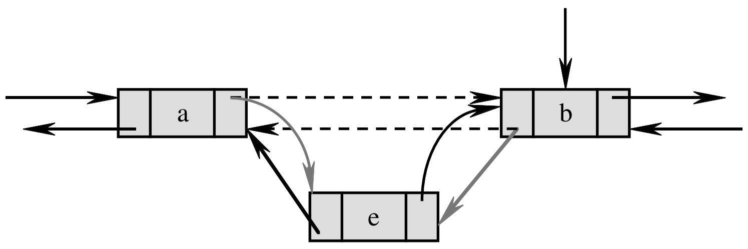
A doubly-linked list can be read forwards or backwards, and allow elements to be inserted or deleted at any point. It is cyclic if it closes back on itself:



This mutable data structure, sometimes called a ***ring buffer***, should be familiar to most programmers. We implement it here to make a comparison between references in Standard ML and pointer variables in procedural languages. Let us define an abstract type with the following signature:

```
signature RINGBUF =
  sig
  eqtype 'a t
  exception Empty
  val empty     :  unit -> 'a t
  val null      :  'a t -> bool
  val label     :  'a t -> 'a
  val moveLeft  :  'a t -> unit
  val moveRight :  'a t -> unit
  val insert    :  'a t * 'a -> unit
  val delete    :  'a t -> 'a
  end;
```

A ring buffer has type $\alpha\ t$ and is a reference into a doubly-linked list. A new ring buffer is created by calling the function *empty*. The function *null* tests whether a ring buffer is empty, *label* returns the label of the current node, and *moveLeft*/*moveRight* move the pointer to the left/right of the current node. As shown below, *insert*(*buf*, *e*) inserts a node labelled *e* to the left of the current node. Two links are redirected to the new node; their initial orientations are shown by dashed arrows and their final orientations by shaded arrows:



The function *delete* removes the current node and moves the pointer to the right. Its value is the label of the deleted node.

The code, which appears in Figure 8.2, is much as it might be written in Pascal. Each node of the doubly-linked list has type α *buf*, which contains a label and references to the nodes on its left and right. Given a node, the functions *left* and *right* return these references.

The constructor *Nil* represents an empty list and serves as a placeholder, like Pascal's `nil` pointer. If *Node* were the only constructor of type α *buf*, no value of that type could be created. Consider the code for *insert*. When the first node is created, its left and right pointers initially contain *Nil*. They are then updated to contain the node itself.

Bear in mind that reference equality in ML differs from the usual notion of pointer equality. The function *delete* must check whether the only node of a buffer is about to be deleted. It cannot determine whether *Node*(*lp*, *x*, *rp*) is the

Figure 8.2 *Ring buffers as doubly-linked lists*

```
structure RingBuf :> RINGBUF =
  struct
  datatype 'a buf = Nil | Node of 'a buf ref * 'a * 'a buf ref;
  datatype 'a t   = Ptr of 'a buf ref;
  exception Empty;

  fun left  (Node(lp,_,_)) = lp
    | left  Nil              = raise Empty;

  fun right  (Node(_,_,rp)) = rp
    | right  Nil              = raise Empty;

  fun empty() = Ptr(ref Nil);

  fun null (Ptr p) = case !p of
          Nil           => true
        | Node(_,x,_)  => false;

  fun label (Ptr p) = case !p of
          Nil           => raise Empty
        | Node(_,x,_)  => x;

  fun moveLeft  (Ptr p)    = (p := !(left(!p)));
  fun moveRight  (Ptr p)   = (p := !(right(!p)));

  fun insert (Ptr p, x) =
      case !p of
          Nil           =>
              let val lp    = ref Nil
                  and rp    = ref Nil
                  val new  = Node(lp,x,rp)
              in  lp := new;  rp := new;  p := new  end
        | Node(lp,_,_)  =>
              let val new = Node(ref(!lp), x, ref(!p))
              in  right(!lp) := new;  lp := new  end;

  fun delete (Ptr p) =
      case !p of
          Nil             => raise Empty
        | Node(lp,x,rp)  =>
             (if left(!lp) = lp then p := Nil
              else (right(!lp) := !rp;  left (!rp) := !lp;  p := !rp);
              x)
  end;
```

only node by testing whether $lp = rp$, as a Pascal programmer might expect. That equality will always be false in a properly constructed buffer; each link field must be a distinct reference so that it can be updated independently. The test $left(!lp) = lp$ is correct. If the node on the left (namely $!lp$) and the current node have the same left link, then they are the same node and that is the only node in the buffer.

Here is a small demonstration of ring buffers. First, let us create an empty buffer. Because the call to *empty* is not a syntactic value, we must constrain its result to some monotype, here *string*. (Compare with the empty sequence *ImpSeq*.*empty*, which contains no references and is polymorphic.)

```
val buf: string RingBuf.t = RingBuf.empty();
> val buf = - : string RingBuf.t
RingBuf.insert(buf, "They");
```

If only *insert* and *delete* are performed, then a ring buffer behaves like a mutable queue; elements can be inserted and later retrieved in the same order.

```
RingBuf.insert(buf, "shall");
RingBuf.delete buf;
> "They" : string
RingBuf.insert(buf, "be");
RingBuf.insert(buf, "famed");
RingBuf.delete buf;
> "shall" : string
RingBuf.delete buf;
> "be" : string
RingBuf.delete buf;
> "famed" : string
```

Exercise 8.14 Modify *delete* to return a boolean value instead of a label: *true* if the modified buffer is empty and otherwise *false*.

Exercise 8.15 Which of the equalities below are suitable for testing whether $Node(lp, x, rp)$ is the only node in a ring buffer?

$$!lp = !rp \qquad right(!lp) = lp \qquad right(!lp) = rp$$

Exercise 8.16 Compare the following insertion function with *insert*; does it have any advantages or disadvantages?

```
fun insert2 (Ptr p, x) =
    case !p of
        Nil           => p := Node(p,x,p)
      | Node(lp,_,_) =>
            let val new = Node(lp,x,p)
            in  right(!lp) := new;   lp := new end;
```

Exercise 8.17 Code a version of *insert* that inserts the new node to the right of the current point, rather than to the left.

Exercise 8.18 Show that if a value of type α *RingBuf* . *t* (with a strong type variable) could be declared, a run-time type error could ensue.

Exercise 8.19 What good is equality testing on type α *RingBuf* . *t*?

8.6 *Mutable and functional arrays*

The Definition of Standard ML says nothing about arrays, but the standard library provides a structure *Array* with the following signature:

```
signature ARRAY =
  sig
  eqtype 'a array
  val array      :  int * 'a -> 'a array
  val fromList   :  'a list -> 'a array
  val tabulate   :  int * (int -> 'a) -> 'a array
  val sub        :  'a array * int -> 'a
  val update     :  'a array * int * 'a -> unit
  val length     :  'a array -> int
  ⋮
  end;
```

Each array has a fixed size. An n-element array admits subscripts from 0 to $n-1$. The operations raise exception *Subscript* if the array bound is exceeded and raise *Size* upon any attempt to create an array of negative (or grossly excessive) size.[2]

Here is a brief description of the main array operations:

- *array*(n, x) creates an n-element array with x stored in each cell.
- *fromList*$[x_0, x_1, \ldots, x_{n-1}]$ creates an n-element array with x_k stored in cell k, for $k = 0, \ldots, n-1$.
- *tabulate*(n, f) creates an n-element array with $f(k)$ stored in cell k, for $k = 0, \ldots, n-1$.

[2] These exceptions are declared in the library structure *General*.

- $sub(A, k)$ returns the contents of cell k of array A.
- $update(A, k, x)$ updates cell k of array A to contain x.
- $length(A)$ returns the size of array A.

Array are mutable objects and behave much like references. They always admit equality: two arrays are equal if and only if they are the same object. Arrays of arrays may be created, as in Pascal, to serve as multi-dimensional arrays.

Standard library aggregate structures. Arrays of type α *array* can be updated. Immutable arrays provide random access to static data, and can make functional programs more efficient. The library structure *Vector* declares a type α *vector* of immutable arrays. It provides largely the same operations as *Array*, excluding *update*. Functions *tabulate* and *fromList* create vectors, while *Array* . *extract* extracts a vector from an array.

Because types α *array* and α *vector* are polymorphic, they require an additional indirection for every element. Monomorphic arrays and vectors can be represented more compactly. The library signature *MONO_ARRAY* specifies the type *array* of mutable arrays over another type *elem*. Signature *MONO_VECTOR* is analogous, specifying a type *vector* of immutable arrays. Various standard library structures match these signatures, giving arrays of characters, floating point numbers, etc.

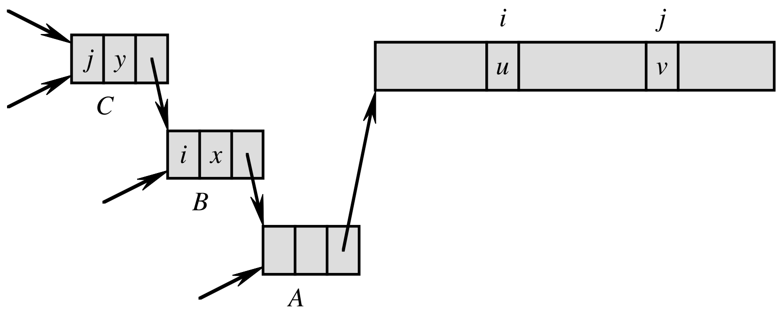
The library regards arrays, vectors and even lists as variations on one concept: aggregates. The corresponding operations agree as far as possible. Arrays, like lists, have *app* and fold functionals. The function *Array* . *fromList* converts a list to an array, and the inverse operation is easy to code:

```
fun toList l = Array.foldr op:: [] l;
```

Lists, like arrays, have a *tabulate* function. They both support subscripting, indexed from zero, and both raise exception *Subscript* if the upper bound is exceeded.

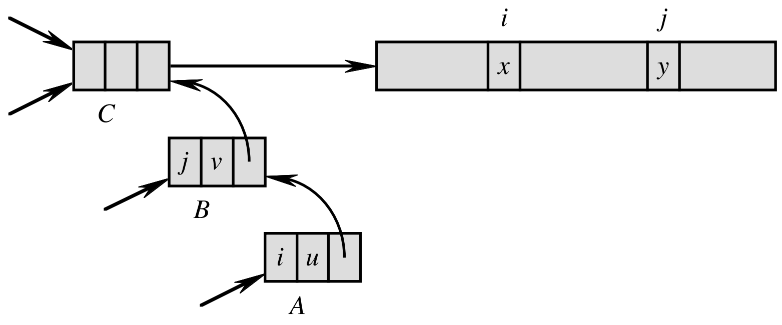
Representing functional arrays. Holmström and Hughes have developed a hybrid representation of functional arrays, exploiting mutable arrays and association lists. An association list consisting of $(index, contents)$ pairs has a functional update operation: simply add a new pair to the front of the list. Update is fast, but lookup requires an expensive search. Introducing a mutable array, called the ***vector***, makes lookups faster (Aasa *et al.*, 1988).

Initially, a functional array is represented by a vector. Update operations build an association list in front of the vector, indicating differences between the current contents of the vector and the values of various arrays. Consider two cells i and j of a functional array A, with $i \neq j$, and suppose $A[i] = u$ and $A[j] = v$. Now perform some functional updates. Obtain B from A by storing x in position i; obtain C from B by storing y in position j:



Other links into A, B and C are shown; these come from arrays created by further updating. The arrays form a tree, called a ***version tree*** since its nodes are ‘versions’ of the vector. Unlike ordinary trees, its links point towards the root rather than away from it. The root of the tree is A, which is a dummy node linked to the vector. The dummy node contains the only direct link into the vector, in order to simplify the re-rooting operation.

Re-rooting the version tree. Although C has the correct value, with $C[i] = x$, $C[j] = y$ and the other elements like in A, lookups to C are slower than they could be. If C is the most heavily used version of the vector, then the root of the version tree ought to be moved to C. The links from C to the vector are reversed; the updates indicated by those nodes are executed in the vector; the previous contents of those vector cells are recorded in the nodes.



This operation does not affect the values of the functional arrays, but lookups to A become slower while lookups to C become faster. The dummy node is now C. Nodes of the version tree that refer to A, B, or C likewise undergo a change in lookup time, but not in value. Re-rooting does not require locating those other nodes. If there are no other references to A or B then the ML storage allocator will reclaim them.

Figure 8.3 *Functional arrays as version trees*

```
structure Varray :> VARRAY =
  struct
  datatype 'a t = Modif of {limit  : int,
                            index  : int ref,
                            elem   : 'a ref,
                            next   : 'a t ref}
                | Main of 'a Array.array;

  fun array (n,x) =
        if n < 0  then  raise Size
        else  Modif{limit=n, index=ref 0, elem=ref x,
                    next=ref(Main(Array.array(n,x)))};

  fun reroot (va as Modif{index, elem, next,...}) =
      case !next of
         Main _  => va  (*have reached root*)
       | Modif _  =>
           let val Modif{index=bindex,elem=belem,next=bnext,...} =
                             reroot (!next)
               val Main ary = !bnext
           in  bindex := !index;
               belem  := Array.sub(ary, !index);
               Array.update(ary, !index, !elem);
               next   := !bnext;
               bnext  := va;
               va
           end;

  fun sub (Modif{index,elem,next,...}, i) =
       case !next of
          Main ary => Array.sub(ary,i)
        | Modif _  => if !index = i then !elem
                                    else sub(!next,i);

  fun justUpdate(va as Modif{limit,...}, i, x) =
        if  0<=i andalso i<limit
        then Modif{limit=limit, index= ref i, elem=ref x, next=ref va}
        else raise Subscript;

  fun update(va,i,x) = reroot(justUpdate(va,i,x));
  end;
```

An implementation. Figure 8.3 shows an ML structure declaration for version tree arrays, called v-arrays for short. It matches the following signature:

```
signature VARRAY =
  sig
  type 'a t
  val array       :  int * 'a -> 'a t
  val reroot      :  'a t -> 'a t
  val sub         :  'a t * int -> 'a
  val justUpdate  :  'a t * int * 'a -> 'a t
  val update      :  'a t * int * 'a -> 'a t
  end;
```

An opaque signature constraint hides the representation of v-arrays, including their equality test. The underlying equality compares identity of stored objects, which is not functional behaviour.

The type of v-arrays is $\alpha\ t$, which has constructors *Modif* and *Main*. A *Modif* (for modification) node is a record with four fields. The upper limit of the v-array is stored for subscript checking. The other fields are references to an index, an element and the next v-array; these are updated during re-rooting. A *Main* node contains the mutable array.

Calling $array(n, x)$ constructs a v-array consisting of a vector and a dummy node. The recursive function *reroot* performs re-rooting. The subscript operation $sub(va, i)$ searches in the nodes for i and if necessary looks up that subscript in the vector. The function *justUpdate* simply creates a new node, while *update* follows this operation by re-rooting at the new array. The library exceptions *Subscript* and *Size* can be raised explicitly and from the *Array* operations.

Programs frequently use functional arrays in a single-threaded fashion, discarding the previous value of the array after each update. In this case, we should re-root after each update. If many versions of a functional array are active then version trees could be inefficient; only one version can be represented by the vector. In this case, we should represent functional arrays by binary trees, as in Section 4.15. Binary trees would also allow an array to grow and shrink.

Experimental results for v-arrays. The code above is based upon Aasa *et al.* (1988). For several single-threaded algorithms, they found v-arrays to be more efficient than other representations of functional arrays. At best, v-arrays can perform lookups and updates in constant time, although more slowly than mutable arrays. Quick sort on v-arrays turns out to be no faster than quick sort on lists, suggesting that arrays should be reserved for tasks requiring random access. Lists are efficient for processing elements sequentially.

Exercise 8.20 Recall the function *allChange* of Section 3.7. With the help of arrays, write a function that can efficiently determine the value of

```
length(allChange([], [5,2], 16000));
```

Exercise 8.21 Add a function *fromList* to structure *Varray*, to create a v-array from a non-empty list.

Exercise 8.22 Add a function *copy* to structure *Varray*, such that $copy(va)$ creates a new v-array having the same value as va.

Exercise 8.23 Declare a structure *Array*2 for mutable arrays of 2 dimensions, with components analogous to those of *Array*.

Exercise 8.24 Declare a structure *Varray*2 for v-arrays of 2 dimensions, with components analogous to those of *Varray*.

Exercise 8.25 What are the contents of the dummy node? Could an alternative representation of v-arrays eliminate this node?

Input and output

Input/output can be a tiresome subject. Reading data in and printing results out seems trivial compared with the computation lying in between — especially as the operations must conform to the arbitrary features of common operating systems. Input/output brings our secure, typed and mainly functional world into contact with byte-oriented, imperative devices. Small wonder that the ML *Definition* specified a parsimonious set of primitives. The Algol 60 definition did not bother with input/output at all.

The ML library rectifies this omission, specifying several input/output models and dozens of operations. It also provides string processing functions for scanning inputs and formatting outputs. We shall examine a selection of these.

Input/output of linked data structures such as trees poses special difficulties. Flattening them to character strings destroys the usually extensive sharing of subtrees, causing exponential blowup. A persistent store, like the one in Poly/ML, can save arbitrary data structures efficiently. Such facilities are hard to find and can be inflexible.

8.7 *String processing*

The library provides extensive functions for processing strings and substrings. Structures *Int*, *Real* and *Bool* (among others) contain functions for translating between basic values and strings. The main functions are *toString*, *fromString*, *fmt* and *scan*. Instead of overloading these functions at top level, the library declares specialized versions in every appropriate structure. You might declare these functions in some of your own structures.

Converting to strings. The function *toString* expresses its argument as a string according to a default format:

```
Int.toString (~23 mod 10);
> "7" : string
Real.toString Math.pi;
> "3.14159265359" : string
Bool.toString (Math.pi = 22.0/7.0);
> "false" : string
```

Structure *StringCvt* supports more elaborate formatting. You can specify how many decimal places to display, and pad the resulting string to a desired length. You even have a choice of radix. For example, the DEC PDP-8 used octal notation, and padded integers to four digits:

```
Int.fmt StringCvt.OCT 31;
> "37" : string
StringCvt.padLeft #"0" 4 it;
> "0037" : string
```

Operations like *String*.*concat* can combine formatted results with other text.

Converting from strings. The function *fromString* converts strings to basic values. It is permissive; numeric representations go well beyond what is valid in ML programs. For instance, the signs + and − are accepted as well as ~:

```
Real.fromString "+.6626e-33";
> SOME 6.626E~34 : real option
```

The string is scanned from left to right and trailing characters are ignored. User errors may go undetected:

```
Int.toString "1o24";
> SOME 1 : int option
Bool.fromString "falsetto";
> SOME false : bool option
```

Not every string is meaningful, no matter how permissive we are:

```
Int.fromString "My master's mind";
> NONE : int option
```

Figure 8.4 *Scanning dates from strings*

```
val months = ["JAN", "FEB", "MAR", "APR", "MAY", "JUN",
              "JUL", "AUG", "SEP", "OCT", "NOV", "DEC"];

fun dateFromString s =
  let val sday::smon::syear::_ = String.tokens (fn c => c = #"-") s
      val SOME day  = Int.fromString sday
      val mon       = String.substring (smon, 0, 3)
      val SOME year = Int.fromString syear
  in  if List.exists (fn m => m=mon) months
      then SOME (day, mon, year)
      else NONE
  end
  handle Subscript => NONE
       | Bind      => NONE;
```

Splitting strings apart. Since *fromString* ignores leftover characters, how are we to translate a series of values in a string? The library structures *String* and *Substring* provide useful functions for scanning. The function *String.tokens* extracts a list of tokens from a string. Tokens are non-empty substrings separated by one or more delimiter characters. A predicate of type $char \to bool$ defines the delimiter characters; structure *Char* contains predicates for recognizing letters (*isAlpha*), spaces (*isSpace*) and punctuation (*isPunct*). Here are some sample invocations:

```
String.tokens Char.isSpace
    "What is thy name?  I know thy quality.";
> ["What", "is", "thy", "name?",
>  "I", "know", "thy", "quality."] : string list
String.tokens Char.isPunct
    "What is thy name?  I know thy quality.";
> ["What is thy name", "  I know thy quality"]
> : string list
```

We thus can split a string of inputs into its constituent parts, and pass them to *fromString*. Function *dateFromString* (Figure 8.4) decodes dates of the form *dd-MMM-yyyy*. It takes the first three hyphen-separated tokens of the input. It parses the day and year using *Int.fromString*, and shortens the month to three characters using *String.substring*. It returns *NONE* if the month is unknown, or if exceptions are raised; *Bind* could arise in three places.

```
dateFromString "25-OCTOBRE-1415-shall-live-forever";
```

```
> SOME (25, "OCT", 1415) : (int * string * int) option
dateFromString "2L-DECX-18o5";
> SOME (2, "DEC", 18) : (int * string * int) option
```

We see that *dateFromString* is as permissive as the other *fromString* functions.

Scanning from character sources. The *scan* functions, found in several library structures, give precise control over text processing. They accept any functional character source, not just a string. If you can write a function

$$getc : \sigma \to (char \times \sigma)option$$

then type σ can be used as a character source. Calling *getc* either returns *NONE* or else packages the next character with a further character source.

The *scan* functions read a basic value, consuming as many characters as possible and leaving the rest for subsequent processing. For example, let us define lists as a character source:

```
fun listGetc (x::l) = SOME (x,l)
  | listGetc []     = NONE;
> val listGetc = fn : 'a list -> ('a * 'a list) option
```

The *scan* functions are curried, taking the character source as their first argument. The integer *scan* function takes, in addition, the desired radix; *DEC* means decimal. Let us scan some faulty inputs:

```
Bool.scan listGetc (explode "mendacious");
> NONE : (bool * char list) option
Bool.scan listGetc (explode "falsetto");
> SOME (false, [#"t", #"t", #"o"])
> : (bool * char list) option
Real.scan listGetc (explode "6.626x-34");
> SOME (6.626, [#"x", #"-", #"3", #"4"])
> : (real * char list) option
Int.scan StringCvt.DEC listGetc (explode "1o24");
> SOME (1, [#"o", #"2", #"4"])
> : (int * char list) option
```

The mis-typed characters `x` and `o` do not prevent numbers from being scanned, but they remain in the input. Such errors can be detected by checking that the input has either been exhausted or continues with an expected delimiter character. In the latter case, delimiters can be skipped and further values scanned.

The *fromString* functions are easy to use but can let errors slip by. The *scan* functions form the basis for robust input processing.

Exercise 8.26 Declare function *writeCheque* for printing amounts on cheques. Calling *writeCheque* *w* (*dols*, *cents*) should express the given sum in dollars and cents to fit a field of width *w*. For instance, *writeCheque* 9 (57,8) should return the string "$***57.08"

Exercise 8.27 Write a function *toUpper* for translating all the letters in a string to upper case, leaving other characters unchanged. (Library structures *String* and *Char* have relevant functions.)

Exercise 8.28 Repeat the examples above using substrings instead of lists as the source of characters. (The library structure *Substring* declares useful functions including *getc*.)

Exercise 8.29 Use the *scan* functions to code a function for scanning dates. It should accept an arbitrary character source. (Library structure *StringCvt* has relevant functions.)

8.8 *Text input/output*

ML's simplest input/output model supports imperative operations on text files. A ***stream*** connects an external file (or device) to the program for transmitting characters. An ***input*** stream is connected to a producer of data, such as the keyboard; characters may be read from it until the producer terminates the stream. An ***output*** stream is connected to a consumer of data, such as a printer; characters may be sent to it until the program terminates the stream.

The input and output operations belong to structure *TextIO*, whose signature is in effect an extension of the following:

```
signature TEXT_IO =
  sig
  type instream and outstream
  exception Io of {name: string, function: string, cause: exn}
  val stdIn       : instream
  val stdOut      : outstream
  val openIn      : string -> instream
  val openOut     : string -> outstream
  val closeIn     : instream -> unit
  val closeOut    : outstream -> unit
  val inputN      : instream * int -> string
  val inputLine   : instream -> string
  val inputAll    : instream -> string
  val lookahead   : instream -> char option
  val endOfStream : instream -> bool
  val output      : outstream * string -> unit
  val flushOut    : outstream -> unit
  val print       : string -> unit
  end;
```

Here are brief descriptions of these items. Consult the library documentation for more details.

- Input streams have type *instream* while output streams have type *outstream*. These types do not admit equality.
- Exception *Io* indicates that some low-level operation failed. It bundles up the name of the affected file, a primitive function and the primitive exception that was raised.
- *stdIn* and *stdOut*, the standard input and output streams, are connected to the terminal in an interactive session.
- *openIn*(s) and *openOut*(s) create a stream connected to the file named s.
- *closeIn*(*is*) and *closeOut*(*os*) terminate a stream, disconnecting it from its file. The stream may no longer transmit characters. An input stream may be closed by its device, for example upon end of file.
- *inputN*(*is*, n) removes up to n characters from stream *is* and returns them as a string. If fewer than n characters are present before the stream closes then only those characters are returned.
- *inputLine*(*is*) reads the next line of text from stream *is* and returns it as a string ending in a newline character. If stream *is* has closed, then the empty string is returned.
- *inputAll*(*is*) reads the entire contents of stream *is* and returns them as a string. Typically it reads in an entire file; it is not suitable for interactive input.
- *lookahead*(*is*) returns the next character, if it exists, without removing it from the stream *is*.
- *endOfStream*(*is*) is *true* if the stream *is* has no further characters before its terminator.
- *output*(*os*, s) writes the characters of string s to the stream *os*, provided it has not been closed.
- *flushOut*(*os*) sends to their ultimate destination any characters waiting in system buffers.
- *print*(s) writes the characters in s to the terminal, as might otherwise be done using *output* and *flushOut*. Function *print* is available at top level.

The input operations above may ***block***: wait until the required characters appear or the stream closes.

Suppose the file `Harry` holds some lines by Henry V, from his message to the French shortly before the battle of Agincourt:

```
My people are with sickness much enfeebled,
my numbers lessened, and those few I have
almost no better than so many French ...
But, God before, we say we will come on!
```

Let *infile* be an input stream to `Harry`. We peek at the first character:

```
val infile = TextIO.openIn("Harry");
> val infile = ? : TextIO.instream
TextIO.lookahead infile;
> SOME #"M" : char option
```

Calling *lookahead* does not advance into the file. But now we extract ten characters as a string, then read the rest of the line.

```
TextIO.inputN (infile,10);
> "My people " : string
TextIO.inputLine infile;
> "are with sickness much enfeebled;\n" : string
```

Calling *inputAll* gets the rest of the file as a long and unintelligible string, which we then output to the terminal:

```
TextIO.inputAll infile;
> "my numbers lessened, and those few I have\nalmo#
print it;
> my numbers lessened, and those few I have
> almost no better than so many French ...
> But, God before, we say we will come on!
```

A final peek reveals that we are at the end of the file, so we close it:

```
TextIO.lookahead infile;
> NONE : char option
TextIO.inputLine infile;
> "" : string
TextIO.closeIn infile;
```

Closing streams when you are finished with them conserves system resources.

8.9 *Text processing examples*

A few small examples will demonstrate how to process text files in ML. The amount of actual input/output is surprisingly small; string processing functions such as *String.tokens* do most of the work.

Batch input/output. Our first example is a program to read a series of lines and print the initial letters of each word. Words are tokens separated by spaces; subscripting gets their initial characters and *implode* joins them to form a string:

```
fun firstChar s = String.sub(s,0);
> val firstChar = fn : string -> char
val initials = implode o (map firstChar) o
               (String.tokens Char.isSpace);
> val initials = fn : string -> string
initials "My ransom is this frail and worthless trunk";
> "Mritfawt" : string
```

The function *batchInitials*, given input and output streams, repeatedly reads a line from the input and writes its initials to the output. It continues until the input stream is exhausted.

```
fun batchInitials (is, os) =
  while not (TextIO.endOfStream is)
  do TextIO.output(os, initials (TextIO.inputLine is) ^ "\n");
> val batchInitials = fn
> : TextIO.instream * TextIO.outstream -> unit
```

Let *infile* be a fresh input stream to `Harry`. We apply *batchInitials* to it:

```
val infile = TextIO.openIn("Harry");
> val infile = ? : TextIO.instream
batchInitials(infile, TextIO.stdOut);
> Mpawsme
> mnlatfIh
> anbtsmF.
> BGbwswwco
```

The output appears at the terminal because *stdOut* has been given as the output stream.

Interactive input/output. We can make *batchInitials* read from the terminal just by passing *stdIn* as its first argument. But an interactive version ought to display a prompt before it pauses to accept input. A naïve attempt calls *output* just before calling *inputLine*:

```
  while not (TextIO.endOfStream is)
  do (TextIO.output(os, "Input line? ");
        TextIO.output(os, initials(TextIO.inputLine is) ^ "\n"));
```

But this does not print the prompt until after it has read the input! There are two mistakes. (1) We must call *flushOut* to ensure that the output really appears, instead of sitting in some buffer. (2) We must print the prompt before calling *endOfStream*, which can block; therefore we must move the prompting code between the `while` and `do` keywords. Here is a better version:

```
fun promptInitials (is, os) =
  while (TextIO.output(os, "Input line? ");
         TextIO.flushOut os;
         not (TextIO.endOfStream is))
  do TextIO.output(os, "Initials:   " ^
                         initials(TextIO.inputLine is) ^ "\n");
> val promptInitials = fn
> : TextIO.instream * TextIO.outstream -> unit
```

Recall that evaluating the expression $(E_1; E_2; \ldots; E_n)$ evaluates $E_1, E_2, \ldots, E_n$ in that order and returns the value of E_n. We can execute any commands before the testing the loop condition. In this sample execution, text supplied to the standard input is underlined:

```
promptInitials(TextIO.stdIn, TextIO.stdOut);
> Input line? If we may pass, we will;
> Initials:   Iwmpww
> Input line? If we be hindered ...
> Initials:   Iwbh.
> Input line?
```

The final input above was Control-D, which terminates the input stream. That does not prevent our reading further characters from *stdIn* in the future. Similarly, after we reach the end of a file, some other process could extend the file. Calling *endOfStream* can return *true* now and *false* later.

If the output stream is always the terminal, using *print* further simplifies the `while` loop:

```
  while (print "Input line? "; not (TextIO.endOfStream is))
  do print ("Initials:   " ^ initials(TextIO.inputLine is) ^ "\n");
```

Translating into HTML. Our next example performs only simple input/output, but illustrates the use of substrings. A value of type *substring* is represented by a string s and two integers i and n; it stands for the n-character segment of s starting at position i. Substrings support certain forms of text processing efficiently, with minimal copying and bounds checking. A substring can be

Figure 8.5 *Raw input prior to conversion*

```
Westmoreland. Of fighting men they have full three score thousand.

Exeter. There's five to one; besides, they all are fresh.

Westmoreland. O that we now had here
But one ten thousand of those men in England
That do no work to-day!

King Henry V. What's he that wishes so?
My cousin Westmoreland? No, my fair cousin:
If we are marked to die, we are enough
To do our country loss; and if to live,
The fewer men, the greater share of honour.
```

divided into tokens or scanned from the left or right; the results are themselves substrings.

Our task is to translate plays from plain text into HTML, the HyperText Markup Language used for the World Wide Web. Figure 8.5 shows a typical input. Blank lines separate paragraphs. Each speech is a paragraph; the corresponding output must insert the <P> markup tag. The first line of a paragraph gives the character's name, followed by a period; the output must emphasize this name by enclosing it in the <EM> and </EM> tags. To preserve line breaks, the translation should attach the <BR> tag to subsequent lines of each paragraph.

Function *firstLine* deals with the first line of a paragraph, separating the name from the rest of line. It uses three components of the library structure *Substring*, namely *all*, *splitl* and *string*. Calling *all s* creates a substring representing the whole of string *s*. The call to *splitl* scans this substring from left to right, returning in *name* the substring before the first period, and in *rest* the remainder of the original substring. The calls to *string* convert these substrings to strings so that they can be concatenated with other strings containing the markup tags.

```
fun firstLine s =
    let val (name, rest) =
          Substring.splitl (fn c => c <> #".") (Substring.all s)
    in  "\n<P><EM>" ^ Substring.string name ^
        "</EM>"     ^ Substring.string rest
    end;
```

Figure 8.6 *Displayed output from* HTML

Westmoreland. Of fighting men they have full three score thousand.

Exeter. There's five to one; besides, they all are fresh.

Westmoreland. O that we now had here
But one ten thousand of those men in England
That do no work to-day!

King Henry V. What's he that wishes so?
My cousin Westmoreland? No, my fair cousin:
If we are marked to die, we are enough
To do our country loss; and if to live,
The fewer men, the greater share of honour.

```
> val firstLine = fn : string -> string
```

In this example, observe the placement of the markup tags:

```
firstLine "King Henry V. What's he that wishes so?";
> "\n<P><EM>King Henry V</EM>. What's he that wishes so?"
> : string
```

Function *htmlCvt* takes a filename and opens input and output streams. Its main loop is the recursive function *cvt*, which translates one line at a time, keeping track of whether or not it is the first line of a paragraph. An empty string indicates the end of the input, while an empty line (containing just the newline character) starts a new paragraph. Other lines are translated according as whether or not they are the first. The translated line is output and the process repeats.

```
fun htmlCvt fileName =
    let val is  = TextIO.openIn fileName
        and os  = TextIO.openOut (fileName ^ ".html")
        fun cvt _ ""    = ()
          | cvt _ "\n" = cvt true (TextIO.inputLine is)
          | cvt first s  =
                (TextIO.output (os,
                                 if first then firstLine s
                                 else "<BR>" ^ s);
                 cvt false (TextIO.inputLine is));
    in  cvt true "\n";  TextIO.closeIn is;  TextIO.closeOut os
    end;
> val htmlCvt = fn : string -> unit
```

Finally, *htmlCvt* closes the streams. Closing the output stream ensures that text held in buffers actually reaches the file. Figure 8.6 on the preceding page shows how a Web browser displays the translated text.

Input/output and the standard library. Another useful structure is *BinIO*, which supports input/output of binary data in the form of 8-bit bytes. Characters and bytes are not the same thing: characters occupy more than 8 bits on some systems, and they assign special interpretations to certain codes. Binary input/output has no notion of line breaks, for example.

The functors *ImperativeIO*, *StreamIO* and *PrimIO* support input/output at lower levels. (The library specifies them as optional, but better ML systems will provide them.) *ImperativeIO* supports imperative operations, with buffering. *StreamIO* provides functional operations for input: items are not removed from an instream, but yield a new instream. *PrimIO* is the most primitive level, without buffering and implemented in terms of operating system calls. The functors can be applied to support specialized input/output, say for extended character sets.

Andrew Appel designed this input/output interface with help from John Reppy and Dave Berry.

Exercise 8.30 Write an ML program to count how many lines, words and characters are contained in a file. A word is a string of characters delimited by spaces, tabs, or newlines.

Exercise 8.31 Write a procedure to prompt for the radius of a circle, print the corresponding area (using $A = \pi r^2$) and repeat. If the attempt to decode a real number fails, it should print an error message and let the user try again.

Exercise 8.32 The four characters `< > & "` have special meanings in HTML. Occurrences of them in the input should be replaced by the escape sequences `&lt; &gt; &amp; &quot;` (respectively). Modify *htmlCvt* to do this.

8.10 *A pretty printer*

Programs and mathematical formulæ are easier to read if they are laid out with line breaks and indentation to emphasize their structure. The tautology checker of Section 4.19 includes the function *show*, which converts a proposition to a string. If the string is too long to fit on one line, we usually see something like this (for a margin of 30):

```
((((landed | saintly) | ((~lan
ded) | (~saintly))) & (((~rich
) | saintly) | ((~landed) | (~
saintly)))) & (((landed | rich
) | ((~landed) | (~saintly)))
```

Figure 8.7 *Output of the pretty printer*

```
((~(((~landed) | rich) &
    (~(saintly & rich)))) |
 ((~landed) | (~saintly)))

((((landed | saintly) |
   ((~landed) | (~saintly))) &
  (((~rich) | saintly) |
   ((~landed) |
    (~saintly)))) &
 (((landed | rich) |
   ((~landed) | (~saintly))) &
  (((~rich) | rich) |
   ((~landed) | (~saintly)))))


((~(((~landed) | rich) & (~(saintly & rich)))) |
 ((~landed) | (~saintly)))

((((landed | saintly) | ((~landed) | (~saintly))) &
  (((~rich) | saintly) | ((~landed) | (~saintly)))) &
 (((landed | rich) | ((~landed) | (~saintly))) &
  (((~rich) | rich) | ((~landed) | (~saintly)))))
```

```
& (((~rich) | rich) | ((~lande
d) | (~saintly)))))
```

Figure 8.7 shows the rather better display produced by a pretty printer. Two propositions (including the one above) are formatted to margins of 30 and 60. Finding the ideal presentation of a formula may require judgement and taste, but a simple scheme for pretty printing gives surprisingly good results. Some ML systems provide pretty-printing primitives similar to those described below.

The pretty printer accepts a piece of text decorated with information about nesting and allowed break points. Let us indicate nesting by angle brackets ($\langle\rangle$) and possible line breaks by a vertical bar ($|$). An expression of the form $\langle e_1 \ldots e_n \rangle$ is called a ***block***.

For instance, the block

$$\left\langle \left\langle \texttt{a *} \;\middle|\; \texttt{b} \right\rangle \texttt{-} \;\middle|\; \left\langle \texttt{(} \left\langle \texttt{c +} \;\middle|\; \texttt{d} \right\rangle \texttt{)} \right\rangle \right\rangle$$

represents the string `a*b-(c+d)`. It allows line breaks after the characters `*`, `-` and `+`.

When parentheses are suppressed according to operator precedences, correct pretty printing is essential. The nesting structure of the block corresponds to the formula

$$(a \times b) - (c + d) \quad \text{rather than} \quad a \times (b - (c + d)).$$

If `a*b-(c+d)` does not fit on one line, then it should be broken after the `-` character; outer blocks are broken before inner blocks.

The pretty printing algorithm keeps track of how much space remains on the current line. When it encounters a break, it determines how many characters there are until the next break in the same block or in an enclosing block. (Thus it ignores breaks in inner blocks.) If that many characters will not fit on the current line, then the algorithm prints a new line, indented to match the beginning of the current block.

The algorithm does not insist that a break should immediately follow every block. In the previous example, the block

$$\Big\langle\ \texttt{c +}\ \Big|\ \texttt{d}\ \Big\rangle$$

is followed by a `)` character; the string `d)` cannot be broken. Determining the distance until the next break is therefore somewhat involved.

The pretty printer has the signature

```
signature PRETTY =
  sig
  type t
  val blo : int * t list -> t
  val str : string -> t
  val brk : int -> t
  val pr  : TextIO.outstream * t * int -> unit
  end;
```

and provides slightly fancier primitives than those just described:

- t is the type of symbolic expressions, namely blocks, strings and breaks.
- $blo(i, [e_1, \ldots, e_n])$ creates a block containing the given expressions, and specifies that the current indentation be increased by i. This indentation will be used if the block is broken.
- $str(s)$ creates an expression containing the string s.
- $brk(l)$ creates a break of length l; if no line break is required then l spaces will be printed instead.
- $pr(os, e, m)$ prints expression e on stream os with a right margin of m.

Figure 8.8 presents the pretty printer. Observe that *Block* stores the total size of a block, as computed by *blo*. Also, *after* holds the distance from the end of the current block to the next break.

The output shown above in Figure 8.7 was produced by augmenting our tautology checker as follows:

```
local open Pretty
  in

  fun prettyshow (Atom a)      = str a
    | prettyshow (Neg p)       =
        blo(1, [str"(~", prettyshow p, str")"])
    | prettyshow (Conj(p,q))  =
        blo(1, [str"(", prettyshow p, str" &",
                brk 1, prettyshow q, str")"])
    | prettyshow (Disj(p,q))  =
        blo(1, [str"(", prettyshow p, str" |",
                brk 1, prettyshow q, str")"]);

  end;
> val prettyshow = fn : prop -> Pretty.t
```

Calling *Pretty*.*pr* with the result of *prettyshow* does the pretty printing.

Further reading. The pretty printer is inspired by Oppen (1980). Oppen's algorithm is complicated but requires little storage; it can process an enormous file, storing only a few linefuls. Our pretty printer is adequate for displaying theorems and other computed results that easily fit in store. Kennedy (1996) presents an ML program for drawing trees.

Exercise 8.33 Give an example of how a block of the form

$$\Big\langle\Big\langle E_1 \;\texttt{*}\; \Big| E_2\Big\rangle \;\texttt{-}\; \Big|\Big\langle \;\texttt{(}\; \Big\langle E_3 \;\texttt{+}\; \Big| E_4 \Big\rangle \;\texttt{)}\; \Big\rangle\Big\rangle$$

could be pretty printed with a line break after the $\star$ character and none after the $-$ character. How serious is this problem? Suggest a modification to the algorithm to correct it.

Exercise 8.34 Implement a new kind of block, with 'consistent breaks': unless the entire block fits on the current line, all of its breaks are forced. For instance, consistent breaking of

$$\Big\langle \;\texttt{if}\; E \Big| \;\texttt{then}\; E_1 \Big| \;\texttt{else}\; E_2 \Big\rangle$$

would produce

```
if E
then E1
else E2
```

and never

```
if E then E1
else E2
```

Figure 8.8 *The pretty printer*

```
structure Pretty : PRETTY =
  struct
  datatype t = Block  of t list * int * int
             | String of string
             | Break  of int;

  fun breakdist (Block(_,_,len)::es, after) = len + breakdist (es,after)
    | breakdist (String s :: es, after)     = size s + breakdist (es,after)
    | breakdist (Break _ :: es, after)      = 0
    | breakdist ([], after)                 = after;

  fun pr (os, e, margin) =
   let val space = ref margin

       fun blanks n = (TextIO.output(os, StringCvt.padLeft #" " n "");
                       space := !space - n)

       fun newline () = (TextIO.output(os,"\n");  space := margin)

       fun printing ([], _, _)              = ()
         | printing (e::es, blockspace, after) =
          (case e of
               Block(bes,indent,len) =>
                  printing(bes, !space-indent, breakdist (es,after))
             | String s  => (TextIO.output(os,s);  space := !space - size s)
             | Break len =>
                 if len + breakdist (es,after) <= !space
                 then blanks len
                 else (newline();  blanks(margin-blockspace));
            printing (es, blockspace, after))
   in  printing([e], margin, 0);  newline()  end;

  fun length (Block(_,_,len)) = len
    | length (String s)       = size s
    | length (Break len)      = len;

  val str = String  and  brk = Break;

  fun blo (indent,es) =
    let fun sum ([],    k) = k
          | sum (e::es, k) = sum(es, length e + k)
    in  Block(es, indent, sum(es,0))  end;
  end;
```

Exercise 8.35 Write a purely functional version of the pretty printer. Instead of writing to a stream, it should return a list of strings. Does the functional version have any practical advantages?

Exercise 8.36 The Fortran statement

```
FORMAT (' Input =', I6, '  Output =', F8.2)
```

describes a line of text beginning with the string `' Input ='`, followed by an integer taking up 6 characters, followed by the string `'  Output ='`, followed by a floating point (real) number taking up 8 characters, with 2 digits to the right of the decimal point. A file written under a Fortran format can be read under the same format. Discuss how this kind of formatted input/output could be implemented in ML. How would formats and data be represented?

Summary of main points

- References denote mutable cells in the store, like the variables and pointers of procedural languages.
- In ML, variables cannot be updated; only references and arrays can be updated.
- To prevent polymorphic references from causing run-time type errors, the expression in a polymorphic `val` declaration must be a syntactic value.
- Cyclic data structures, like ring buffers, can be constructed using references.
- A function can exploit imperative features while exhibiting purely functional behaviour.
- Input and output commands transmit characters between the program and external devices.