The structure `TextIO` contains a wide variety of I/O types, values and functions. You load these by entering:

```ml
open TextIO;
```

Among the values loaded are:

- **type instream**
  This is the type that represents input text files.

- **type outstream**
  This is the type that represents output text files.

- **type vector = string**
  Makes `vector` a synonym for `string`.

- **type elem = char**
  Makes `elem` a synonym for `char`.

- **val stdIn : instream**
- **val stdOut : outstream**
- **val stdErr : outstream**
  Predefined input and output streams.

- **val openIn : string -> instream**
- **val openOut : string -> outstream**
  Open an input or output stream.

  For example,
  ```ml
  val out = openOut("/tmp/test1");
  val out = - : outstream
  ```

- **val input : instream -> vector**
  Read a line of input into a `string` (vector is defined as equivalent to string). For example (user input is in red):

  ```ml
  val s = input(stdIn);
  "Hello!
  val s = "Hello!\n" : vector
  ```

- **val inputN : instream * int -> vector**
  Read the next `N` input characters into a `string`. For example,
  ```ml
  val t = inputN(stdIn,3);
  abcde
  val t = "abc" : vector
  ```

- **val inputAll : instream -> vector**
  Read the rest of the input file into a `string` (with newlines separating lines). For example,
  ```ml
  val u = inputAll(stdIn);
  Four score and seven years ago ...
  val u = "Four score and\nseven years ago ...\n" : vector
  ```

- **val endOfStream : instream -> bool**
  Are we at the end of this input stream?

- **val output : outstream * vector -> unit**
  Output a string on the specified output stream. For example,
  ```ml
  output(stdOut,
    "That’s all folks!\n");
  That’s all folks!
  ```
String Operations

ML provides a wide variety of string manipulation routines. Included are:

• The string concatenation operator, \^ "abc" ^ "def" = "abcdef"

• The standard 6 relational operators: < > <= >= = <>

• The string size operator:
  val size : string -> int
  size ("abcd");
  val it = 4 : int

• The string subscripting operator (indexing from 0):
  val sub =
  fn : string * int -> char
  sub("abcde", 2);
  val it = #"c" : char

• Concatenation of a list of strings into a single string:
  concat :
  string list -> string
  For example,
  concat ["What's", "up", "?"];
  val it = "What's up?" : string

• Convert a character into a string:
  str : char -> string
  For example,
  str(#"x");
  val it = "x" : string

• “Explode” a string into a list of characters:
  explode : string -> char list
  For example,
  explode("abcde");
  val it = [#"a", #"b", #"c", #"d", #"e"] : char list

• “Implode” a list of characters into a string.
  implode : char list -> string
  For example,
  implode [#"a", #"b", #"c", #"d", #"e"];
  val it = "abcde" : string

Structures and Signatures

In C++ and Java you can group variable and function definitions into classes. In Java you can also group classes into packages.

In ML you can group value, exception and function definitions into structures.

You can then import selected definitions from the structure (using the notation structure.name) or you can open the structure, thereby importing all the definitions within the structure.

(Examples used in this section may be found at ~cs538-1/public/sml/struct.sml)
The general form of a structure definition is
structure name = struct
  val, exception and fun definitions
end
For example,
structure Mapping = struct
  exception NotFound;
  val create = [];
  fun lookup(key,[]) = raise NotFound
  | lookup(key, (key1,value1)::rest) = 
    if key = key1
    then value1
    else lookup(key,rest);
end;

We can access members of this structure as Mapping.name. Thus
fun insert(key,value,[]) = [(key,value)]
  | insert(key,value, (key1,value1)::rest) = 
    if key = key1
    then (key,value)::rest
    else (key1,value1)::
      insert(key,value,rest);
end;

Signatures
Each structure has a signature, which is its type.
For example, Mapping’s signature is
structure Mapping:
  sig
    exception NotFound
    val create : 'a list
    val insert : 'a * 'b * ('a * 'b) list -> ('a * 'b) list
    val lookup : 'a * ('a * 'b) list -> 'b
  end

You can define a signature as
signature name = sig
  type definitions for values, functions and exceptions
end
For example,
signature Str2IntMapping = sig
  exception NotFound;
  val lookup: string * (string*int) list -> int;
end;
Signatures can be used to

- Restrict the type of a value or
  function in a structure.
- Hide selected definitions that appear
  in a structure

For example

```
structure Str2IntMap :
  Str2IntMapping = Mapping;
```

defines a new structure, Str2IntMap, created by restricting Mapping to the
Str2IntMapping signature. When we do this we get

```
open Str2IntMap;
exception NotFound
val lookup : string *
  (string * int) list -> int
```

Only lookup and NotFound are created, and lookup is limited to keys
that are strings.

---

**Extending ML's Polymorphism**

In languages like C++ and Java we must use types like `void*` or `Object` to simulate the polymorphism that ML provides. In ML whenever possible a general type (a polytype) is used rather than a fixed type. Thus in

```
fun len([]) = 0
|  len(a::b) = 1 + len(b);
```

we get a type of

```
'a list -> int
```

because this is the most general type possible that is consistent with `len`'s definition.

Is this form of polymorphism general enough to capture the general idea of making program definitions as type-independent as possible?

---

It isn't, and to see why consider the following ML definition of a merge sort. A merge sort operates by first splitting a list into two equal length sublists. The following function does this:

```
fun split [] = ([],[],[])
  |  split [a] = ([a],[],[])
  |  split (a::b::rest) =
    let val (left,right) =
      split(rest) in
    (a::left, b::right)
    end;
```

After the input list is split into two halves, each half is recursively sorted, then the sorted halves are merged together into a single list.

The following ML function merges two sorted lists into one:

```
fun merge([],[]) = []
  |  merge([],hd::tl) = hd::tl
  |  merge(hd::tl,[],) = hd::tl
  |  merge(hd::tl,h::t) =
    if hd <= h
    then hd::merge(tl,h::t)
    else h::merge(hd::tl,t)
```

With these two subroutines, a definition of a sort is easy:

```
fun sort [] = []
|  sort([a]) = [a]
|  sort(a::b::rest) =
  let val (left,right) =
    split(a::b::rest) in
  merge(sort(left),
       sort(right))
  end;
```
This definition looks very general—it should work for a list of any type. Unfortunately, when ML types the functions we get a surprise:

```ml
val split = fn : 'a list -> 'a list * 'a list
val merge = fn : int list * int list -> int list
val sort = fn : int list -> int list
```

`split` is polymorphic, but `merge` and `sort` are limited to integer lists! Where did this restriction come from?

The problem is that we did a comparison in `merge` using the <= operator, and ML typed this as an integer comparison.

We can make our definition of `sort` more general by adding a comparison function, `le(a,b)` as a parameter to `merge` and `sort`. If we curry this parameter we may be able to hide it from end users. Our updated definitions are:

```ml
fun merge(le,[],[]) = []
|  merge(le,[],hd::tl) = hd::tl
|  merge(le,hd::tl,[]) = hd::tl
|  merge(le,hd::tl,h::t) = if le(hd,h) then hd::merge(le,tl,h::t) else h::merge(le,hd::tl,t)
```

```ml
fun sort le [] = []
|  sort le [a] = [a]
|  sort le (a::b::rest) = let
|     val (left,right) = split(a::b::rest) in
|     merge(le, sort le left, sort le right)
|   end;
```

Now the types of `merge` and `sort` are:

```ml
val merge = fn : ('a * 'a -> bool) * 'a list * 'a list -> 'a list
val sort = fn : ('a * 'a -> bool) -> 'a list -> 'a list
```

We can now “customize” `sort` by choosing a particular definition for the `le` parameter:

```ml
fun le(a,b) = a <= b;
val le = fn : int * int -> bool
```

```ml
fun intsort L = sort le L;
val intsort = fn : int list -> int list
intsort([4,9,0,2,111,~22,8,~123]);
val it = [-123,-22,0,2,4,8,9,111] : int list
```

```ml
fun strle(a:string,b) = a <= b;
val strle = fn : string * string -> bool
fun strsort L = sort strle L;
val strsort = fn : string list -> string list
strsort(['aac','aaa','ABC','123']);
val it = ['123','ABC','aaa','aac'] : string list
Making the comparison relation an explicit parameter works, but it is a bit ugly and inefficient. Moreover, if we have several functions that depend on the comparison relation, we need to ensure that they all use the same relation. Thus if we wish to define a predicate `inOrder` that tests if a list is already sorted, we can use:

```ml
fun inOrder le [] = true
  | inOrder le [a] = true
  | inOrder le (a::b::rest) = le(a,b) andalso
    inOrder le (b::rest);
val inOrder = fn :
  ('a * 'a -> bool) -> 'a list -> bool
```

Now `sort` and `inOrder` need to use the same definition of `le`. But how can we enforce this?

The structure mechanism we studied earlier can help. We can put a single definition of `le` in the structure, and share it:

```ml
structure Sorting =
  struct
  fun le(a,b) = a <= b;

  fun split [] = ([],[])
    | split [a] = ([a],[])
    | split (a::b::rest) =
        let val (left,right) = split rest in
        (a::left,b::right)
        end;

  fun merge([],[]) = []
    | merge([],hd::tl) = hd::tl
    | merge(hd::tl,[],) = hd::tl
    | merge(hd::tl,h::t) =
        if le(hd,h)
        then hd::merge(tl,h::t)
        else h::merge(hd::tl,t);
  end;

  fun sort [] = []
    | sort([a]) = [a]
    | sort(a::b::rest) =
        let val (left,right) = split(a::b::rest) in
        merge(sort(left),
            sort(right))
        end;

  fun inOrder [] = true
    | inOrder [a] = true
    | inOrder (a::b::rest) =
        le(a,b) andalso
        inOrder (b::rest);
  end;
structure Sorting :
  sig
    val inOrder : int list -> bool
    val le : int * int -> bool
    val merge : int * int list * int list
    val sort : int list -> int list
```

To sort a type other than integers, we replace the definition of `le` in the structure.

But rather than actually edit that definition, ML gives us a powerful mechanism to parameterize a structure. This is the `functor`, which allows us to use one or more structures as parameters in the definition of a structure.