**Imperative Features of ML**

ML provides references to heap locations that may be updated. This is essentially the same as access to heap objects via references (Java) or pointers (C and C++).

The expression

```ml
ref val
```

creates a reference to a heap location initialized to `val`. For example,

```ml
ref 0;
val it = ref 0 : int ref
```

The prefix operator `!` fetches the value contained in a heap location (just as `*` dereferences a pointer in C or C++).

Thus

```ml
! (ref 0);
val it = 0 : int
```

The expression

```ml
ref := val
```

updates the heap location referenced by `ref` to contain `val`. The unit value, `()`, is returned.

Hence

```ml
val x = ref 0;
val x = ref 0 : int ref
!x;
val it = 0 : int
x:=1;
val it = () : unit
!x;
val it = 1 : int
```

**Sequential Composition**

Expressions or statements are sequenced using `;`. Hence

```ml
val a = (1+2;3+4);
val a = 7 : int
(x:=1;!x);
val it = 1 : int
```

**Iteration**

```ml
while expr1 do expr2
```

implies iteration (and returns unit); Thus

```ml
(while false do 10);
val it = () : unit
while !x > 0 do x:= !x-1;
val it = () : unit
!x;
val it = 0 : int
```

**Simple I/O**

The function

```ml
print;
val it = fn : string -> unit
```

prints a string onto standard output.

For example,

```ml
print("Hello World\n");
Hello World
```

The conversion routines

```ml
Real.toString;
val it = fn : real -> string
```

```ml
Int.toString;
val it = fn : int -> string
```

```ml
Bool.toString;
val it = fn : bool -> string
```
convert a value (real, int or bool) into a string. Unlike Java, the call must be explicit.

For example,

```plaintext
print(Int.toString(123));
```

123

Also available are

```plaintext
Real.fromString;
val it = fn : string -> real option
```

```plaintext
Int.fromString;
val it = fn : string -> int option
```

```plaintext
Bool.fromString;
val it = fn : string -> bool option
```

which convert from a string to a real or int or bool if possible. (That's why the option type is used).

For example,

```plaintext
case (Int.fromString("123")) of
    SOME(i) => i | NONE => 0;
val it = 123 : int
```

```plaintext
case (Int.fromString("One two three")) of
    SOME(i) => i | NONE => 0;
val it = 0 : int
```

**Text I/O**

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

```plaintext
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`.

```plaintext
• val stdIn : instream
  val stdOut : outstream
  val stdErr : outstream
  Predefined input & output streams.

• val openIn :
   string -> instream
val openOut :
   string -> outstream
Open an input or output stream. For example,

val out = openOut("/tmp/test1");
val out = - : outstream
```

```plaintext
• 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):

val s = input(stdIn);
Hello!
val s = "Hello!\n" : vector
```
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
  val it = 4 : int

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

• The substring function
  val substring : string * int * int -> string
  This function is called as substring(string, start, len)
  start is the starting position, counting from 0.
  len is the length of the desired substring. For example,
  substring("abcdefgij", 3, 4) val it = "defg" : string

• Concatenation of a list of strings into a single string:
  val concat : string list -> string
  For example,
  concat ["What’s"," up","?"] val it = "What’s up?" : string
• Convert a character into a string:
  \[ \text{str: char \to string} \]
  For example,
  \[ \text{str('x')} \];
  \[ \text{val it = "x" : string} \]

• “Explode” a string into a list of characters:
  \[ \text{explode: string \to char list} \]
  For example,
  \[ \text{explode("abcde")} \];
  \[ \text{val it = ["a", "b", "c", "d", "e"] : char list} \]

• “Implode” a list of characters into a string.
  \[ \text{implode: char list \to string} \]
  For example,
  \[ \text{implode ["a", "b", "c", "d", "e"]} \];
  \[ \text{val it = "abcde" : string} \]

---

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 \textit{structures}.

You can then import selected definitions from the structure (using the notation \texttt{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 \texttt{-cs538-1/public/sml/struct.sml})

---

The general form of a structure definition is

\[
\text{structure name = struct val, exception and fun definitions end}
\]

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);
  val insert(key,value,[]) = [(key,value)]
  | insert(key,value,(key1,value1)::rest) = 
    if key = key1 then (key1,value1)::rest
    else (key1,value1)::
        insert(key,value,rest);
end;

We can access members of this structure as \texttt{Mapping.name}. Thus

\[
\text{Mapping.insert(538,"languages",[])};
\]

\[ \text{val it = [(538,"languages") : (int * string) list} \]

open Mapping;

exception NotFound
val create : 'a list
val insert : `'a * 'b * (''a * 'b) list \to (''a * 'b) list
val lookup : `'a * (''a * 'b) list \to 'b
Signatures

Each structure has a signature, which is its type.

For example, Mapping's signature is

```ocaml
structure Mapping :
  sig
    exception NotFound
    val create : 'a list
    val insert : ('a * 'b) list -> ('a * 'b) list
    val lookup : 'a *
      ('a * 'b) list -> 'b
  end
```

You can define a signature as

```ocaml
signature name = sig
  type definitions for values, functions and exceptions
end
```

For example,

```ocaml
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

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

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

```ocaml
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

```ml
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:

```ml
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:

```ml
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:

```ml
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:

```haskell
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:

```haskell
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)
```

```
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:

```
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:

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

```
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
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
```