Functions

Functions take a single argument (which can be a tuple).

Function calls are of the form
function_name argument;

For example
size "xyz";
cos 3.14159;

The more conventional form
size("xyz"); or cos(3.14159);
is OK (the parentheses around the argument are allowed, but unnecessary).
The form (size "xyz") or (cos 3.14159)
is OK too.

Note that the call
plus(1,2);
passes one argument, the tuple (1,2) to plus.
The call dummy();
passes one argument, the unit value, to dummy.
All parameters are passed by value.

Function Types

The type of a function in ML is denoted as T1→T2. This says that a parameter of type T1 is mapped to a result of type T2.
The symbol fn denotes a value that is a function.
Thus
size;
val it = fn : string → int
not;
val it = fn : bool → bool
Math.cos;
val it = fn : real → real
(Math is an ML structure—an external library member that contains separately compiled definitions).

User-Defined Functions

The general form is
fun name arg = expression;
ML answers back with the name defined, the fact that it is a function (the fn symbol) and its inferred type.
For example,
fun twice x = 2*x;
val twice = fn : int → int
fun twotimes(x) = 2*x;
val twotimes = fn : int → int
fun fact n =
  if n=0
  then 1
  else n*fact(n-1);
val fact = fn : int → int
fun plus(x,y) : int = x+y;
val plus = fn : int * int -> int
The : int suffix is a type constraint.
It is needed to help ML decide that +
is integer plus rather than real plus.

Patterns In Function Definitions

The following defines a predicate that
tests whether a list, L is null (the
predefined null function already
does this).
fun isNull L =
  if L=[] then true else
  false;
val isNull = fn : 'a list -> bool

However, we can decompose the
definition using patterns to get a
simpler and more elegant definition:
fun isNull [] = true
  | isNull(_::_) = false;
val isNull = fn : 'a list -> bool

The "|" divides the function
definition into different argument
patterns; no explicit conditional logic
is needed. The definition that matches
a particular actual parameter is
automatically selected.
fun fact(1) = 1
  | fact(n) = n*fact(n-1);
val fact = fn : int -> int

If patterns that cover all possible
arguments aren’t specified, you may
get a run-time Match exception.
If patterns overlap you may get a
warning from the compiler.

fun append([],L) = L
  | append(hd::tl,L) = hd::append(tl,L);
val append = fn :
  'a list * 'a list -> 'a list

If we add the pattern
append(L,[]) = L
we get a redundant pattern warning
(Why?)

fun append ([],L) = L
  | append(hd::tl,L) = hd::append(tl,L)
  | append(L,[]) = L;
stdIn:151.1-153.20 Error: match
redundant
  (nil,L) => ... (hd :: tl,L) => ...
  -->  (L,nil) => ...
But a more precise decomposition is fine:

```plaintext
fun append ([],L) = L
| append(hd::tl, hd2::tl2) = hd::append(tl,hd2::tl2)
| append(hd::tl, []) = hd::tl;
val append = fn :
 'a list * 'a list -> 'a list
```

**Function Types Can be Polytypes**

Recall that 'a, 'b, ... represent type variables. That is, any valid type may be substituted for them when checking type correctness.

ML said the type of `append` is

```plaintext
val append = fn :
 'a list * 'a list -> 'a list
```

Why does 'a appear in three places?

We can define `eitherNull`, a predicate that determines whether either of two lists is null as

```plaintext
fun eitherNull(L1,L2) = null(L1) orelse null(L2);
val eitherNull = fn :
 'a list * 'b list -> bool
```

Why are both 'a and 'b used in `eitherNull`'s type?

---

**Currying**

ML chooses the most general (least-restrictive) type possible for user-defined functions.

Functions are first-class objects, as in Scheme.

The function definition

```plaintext
fun f x y = x :: [y];
val f = fn : 'a -> 'a -> 'a list
```

says that `f` takes a parameter `x`, of type 'a, and returns a function (of `y`, whose type is 'a) that returns a list of 'a.

Contrast this with the more conventional

```plaintext
fun g(x,y) = x :: [y];
val g = fn : 'a * 'a -> 'a list
```

Here `g` takes a pair of arguments (each of type 'a) and returns a value of type 'a list.

The advantage of currying is that we can bind one argument and leave the remaining argument(s) free.
For example

\[ f(1); \]

is a legal call. It returns a function of type

\[ \text{fn} : \text{int} \rightarrow \text{int list} \]

The function returned is equivalent to

\begin{align*}
\text{fun } & \text{h b = 1 :: [b];} \\
\text{val } & \text{h = fn : int} \rightarrow \text{int list}
\end{align*}

Map Revisited

ML supports the map function, which can be defined as

\begin{align*}
\text{fun } & \text{map(f,[]} \text{[]} = \text{[]} \\
| & \text{map(f,x::y) =} \\
& \quad (f \ x) \ :: \ \text{map(f,y);} \\
\text{val map } = \\
\text{fn : (}'a \rightarrow 'b) * 'a \text{ list} \rightarrow 'b \text{ list}
\end{align*}

This type says that map takes a pair of arguments. One is a function from type \('a\) to type \('b\). The second argument is a list of type \('a\). The result is a list of type \('b\).

In curried form map is defined as

\begin{align*}
\text{fun } & \text{map f } \text{[]} = \text{[]} \\
| & \text{map f (x::y) =} \\
& \quad (f \ x) \ :: \ \text{map f y;} \\
\text{val map } = \\
\text{fn : (}'a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}
\end{align*}

This type says that map takes one argument that is a function from type \('a\) to type \('b\). It returns a function that takes an argument that is a list of type \('a\) and returns a list of type \('b\).

The advantage of the curried form of map is that we can now use map to create "specialized" functions in which the function that is mapped is fixed.

For example,

\begin{align*}
\text{val } & \text{neg = map not;} \\
\text{val neg } = \\
& \quad \text{fn : bool list} \rightarrow \text{bool list} \\
\text{neg } & \quad [\text{true, false, true}]; \\
\text{val } & \text{it } = [\text{false, true, false}] : \text{bool list}
\end{align*}

Power Sets Revisited

Let's compute power sets in ML.

We want a function pow that takes a list of values, viewed as a set, and which returns a list of lists. Each sublist will be one of the possible subsets of the original argument.

For example,

\[ \text{pow } [1,2] = \text{[[1,2],[1],[2],[[]]]} \]

We first define a version of cons in curried form:

\begin{align*}
\text{fun } & \text{cons h t } = \text{h::t;} \\
\text{val } & \text{cons } = \text{fn :} \\
& \quad 'a \rightarrow 'a \text{ list} \rightarrow 'a \text{ list}
\end{align*}
Now we define \( \text{pow} \). We define the powerset of the empty list, \([\,]\), to be \([\,\,][\,]\). That is, the power set of the empty set is set that contains only the empty set.

For a non-empty list, consisting of \(h::t\), we compute the power set of \(t\), which we call \(\text{pset} \). Then the power set for \(h::t\) is just \(h\) distributed through \(\text{pset} \) appended to \(\text{pset} \).

We distribute \(h\) through \(\text{pset} \) very elegantly: we just map the function \((\text{cons } h)\) to \(\text{pset} \). \((\text{cons } h)\) adds \(h\) to the head of any list it is given. Thus mapping \((\text{cons } h)\) to \(\text{pset} \) adds \(h\) to all lists in \(\text{pset} \).

Therefore \(\text{pow}\ [1,2] = \)

\[
  (\text{map } (\text{cons } 1) \ [2], \ [1]) \times \ [2], \ [1] =
  [[1,2],[1],[2],[1]]
\]

The complete definition is simply

\[
\begin{align*}
\text{fun pow } & [\,] = \ [\,\,][\,] \\
| \text{pow } (h::t) = \\
& \text{let} \\
& \quad \text{val pset } = \text{pow } t \\
& \quad \text{in} \\
& \quad (\text{map } (\text{cons } h) \ pset) \ @ \ pset \\
& \quad \text{end;}
\end{align*}
\]

\[
\text{val pow } =
\text{fn : 'a list } \to \ 'a \ \text{list list}
\]

Let’s trace the computation of \(\text{pow } [1,2]\).

Here \(h = 1\) and \(t = [2]\). We need to compute \(\text{pow } [2]\).

Now \(h = 2\) and \(t = [\,]\).

We know \(\text{pow } [\,] = \ [\,\,][\,] \),
\(\text{so } \text{pow } [2] = (\text{map } (\text{cons } 2) \ [\,]\)@\([\,]\) = (\[2]\)@\([\,]\) = \([2],[\,]\)

\[
\text{Composing Functions}
\]

We can define a composition function that composes two functions into one:

\[
\begin{align*}
\text{fun comp } & (f,g) (x) = f (g (x)) \\
\text{val comp } &= \text{fn : ('a } \to \ 'b) \times ('c } \to \ 'a) \to 'c } \to \ 'b
\end{align*}
\]

In curried form we have

\[
\begin{align*}
\text{fun comp } f \ g \ x &= f (g (x)) \\
\text{val comp } &= \text{fn : ('a } \to \ 'b) \to ('c } \to \ 'a) \to 'c } \to \ 'b
\end{align*}
\]

For example,

\[
\begin{align*}
\text{fun sqr } x : \text{int} &= x \times x \\
\text{val sqr } &= \text{fn : int } \to \ \text{int}
\end{align*}
\]

\[
\text{comp } \text{sqr } \text{sqr};
\]

\[
\text{val it } = \text{fn : int } \to \ \text{int}
\]
comp sqr sqr 3;
val it = 81 : int

In SML $\circ$ (lower-case O) is the infix composition operator.
Hence
sqr $\circ$ sqr $\equiv$ comp sqr sqr