Examples

val x = 1;
val x = 1 : int
val z = (x,x,x);
val z = (1,1,1) : int * int * int
val L = [z,z];
val L = [(1,1,1),(1,1,1)] : (int * int * int) list
val r = {a=L};
val r = {a=[(1,1,1),(1,1,1)]} : {a:(int * int * int) list}

After rebinding, the “nearest” (most recent) binding is used.
The \textit{and} symbol (not \textit{boolean and}) is used for simultaneous binding:

\begin{verbatim}
val x = 10;
val x = 10 : int
val x = true and y = x;
val x = true : bool
val y = 10 : int
\end{verbatim}

Local definitions are temporary value definitions:

\begin{verbatim}
local
    val x = 10
in
    val u = x*x;
end;
val u = 100 : int
\end{verbatim}
Let bindings are used in expressions:

```ml
let
  val x = 10
in
  5*x
end;
val it = 50 : int
```
Patterns

Scheme (and most other languages) use access or decomposition functions to access the components of a structured object.

Thus we might write

\[
\text{let } ( (h \ (\text{car } L)) \ (t \ (\text{cdr } L)) \ ) \ \\
\text{body } \)
\]

Here \text{car} and \text{cdr} are used as access functions to locate the parts of \text{L} we want to access.

In ML we can access components of lists (or tuples, or records) directly by using patterns. The context in which the identifier appears tells us the part of the structure it references.
val x = (1,2);
val x = (1,2) : int * int
val (h,t) = x;
val h = 1 : int
val t = 2 : int
val L = [1,2,3];
val L = [1,2,3] : int list
val [v1,v2,v3] = L;
val v1 = 1 : int
val v2 = 2 : int
val v3 = 3 : int
val [1,x,3] = L;
val x = 2 : int
val [1,rest] = L;
(* This is illegal. Why? *)
val yy::rest = L;
val yy = 1 : int
val rest = [2,3] : int list
Wildcards

An underscore (\_\_) may be used as a “wildcard” or “don’t care” symbol. It matches part of a structure without defining a new binding.

val zz::_ = L;
val zz = 1 : int

Pattern matching works in records too.

val r = {a=1,b=2};
val r = {a=1,b=2} : 
  {a:int, b:int}
val \{a=va,b=vb\} = r;
val va = 1 : int
val vb = 2 : int
val \{a=wa,b=_\}=r;
val wa = 1 : int
val \{a=za, ...\}=r;
val za = 1 : int
Patterns can be nested too.

val x = ((1,3.0),5);
val x = ((1,3.0),5) : (int * real) * int
val ((1,y),_)=x;
val y = 3.0 : real
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
\texttt{plus}(1,2);
passes one argument, the tuple \((1,2)\) to \texttt{plus}.

The call \texttt{dummy}();
passes one argument, the unit value, to \texttt{dummy}.

All parameters are passed by value.
Function Types

The type of a function in ML is denoted as $T_1 \rightarrow T_2$. This says that a parameter of type $T_1$ is mapped to a result of type $T_2$.

The symbol $\text{fn}$ denotes a value that is a function.

Thus

```plaintext
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 \( \texttt{fn} \) symbol) and its inferred type.

For example,

\[
\begin{align*}
\text{fun twice x = } & 2 \times x; \\
\text{val twice = fn : int } & \rightarrow \text{ int} \\
\text{fun twotimes(x) = } & 2 \times x; \\
\text{val twotimes = fn : int } & \rightarrow \text{ int} \\
\text{fun fact n = } & \\
\text{ if n=0 } & \text{ then 1} \\
\text{ else n \times fact(n-1); } & \\
\text{val fact = fn : int } & \rightarrow \text{ int}
\end{align*}
\]
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 \texttt{null} function already does this).

\[
\text{fun isNull } L = \\
\quad \text{if } L=\texttt{[]} \text{ then true else false;}
\]

\[
\text{val isNull } = \text{ fn : } \texttt{'a list \to bool}
\]

However, we can decompose the definition using patterns to get a simpler and more elegant definition:

\[
\text{fun isNull } \texttt{[]} = \texttt{true} \\
\quad \mid \text{isNull}(\_::\_\texttt{)} = \texttt{false;}
\]

\[
\text{val isNull } = \text{ fn : } \texttt{'a list \to bool}
\]
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.

```plaintext
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:

```ml
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

```ml
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

```ml
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

\[
\text{fun } f \ x \ y = \text{expression;}
\]

defines a function \( f \) (of \( x \)) that returns a function (of \( y \)).

Reducing multiple argument functions to a sequence of one argument functions is called currying (after Haskell Curry, a mathematician who popularized the approach).
Thus

```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

\[
\text{fun h b = 1 :: [b];}
\]
\[
\text{val h = fn : int \rightarrow int list}
\]