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 and symbol (not boolean and) is used for simultaneous binding:

val x = 10;
val x = 10 : int
val x = true and y = x;
val x = true : bool
val y = 10 : int

Local definitions are temporary value definitions:

local
val x = 10
in
val u = x*x;
end;
val u = 100 : int

Patterns

Let bindings are used in expressions:

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

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

Thus we might write

(let ( (h (car L) (t (cdr L)) )
body )

Here car and cdr are used as access functions to locate the parts of 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

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

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

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 $T_1 \rightarrow T_2$. 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
size;
val it = fn : string $\rightarrow$ int
not;
val it = fn : bool $\rightarrow$ bool
Math.cos;
val it = fn : real $\rightarrow$ 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 $\text{fn}$ symbol) and its inferred type.
For example,
fun twice x = 2*x;
val twice = fn : int $\rightarrow$ int
fun twotimes(x) = 2*x;
val twotimes = fn : int $\rightarrow$ int
fun fact n =
  if n=0
  then 1
  else n*fact(n-1);
val fact = fn : int $\rightarrow$ int

fun plus(x,y):int = x+y;
val plus = fn : int * int $\rightarrow$ 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).

```haskell
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:

```haskell
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.

```haskell
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.

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

But a more precise decomposition is fine:

```haskell
fun append([],L) = L
| append(hd::tl,L) = hd::append(tl,L)
| append(L,[]) = L;
stdIn:151.1-153.20 Error: match redundant

fun append([],L) = L
| append(hd::tl,L) = hd::append(tl,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
```

If we add the pattern

```haskell
append(L,[]) = L
```
we get a redundant pattern warning (Why?)

```haskell
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) => ...
```
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

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

Why does 'a appear in three places?

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

\[
\text{fun eitherNull}(L1,L2) = \text{null}(L1) \text{ orelse null}(L2);
\]

\[
\text{val eitherNull} = \text{fn} : \text{'a list * 'b list} \to \text{bool}
\]

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

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

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

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

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

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

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

\[
\text{fun } g (x, y) = x :: [y];
\]

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

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

\[
\text{f(1)};
\]

is a legal call. It returns a function of type

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

The function returned is equivalent to

\[
\text{fun } h \text{ b} = 1 :: [b];
\]

\[
\text{val } h = \text{fn} : \text{int} \to \text{int list}
\]