Redefinition of an identifier is OK, but this is redefinition *not* assignment;

Thus

```
val x = 100;
val x = (x=100);
```

is fine; there is no type error even though the first **x** is an integer and then it is a boolean.

```
val x = 100 : int
val x = true : bool
```


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
```

Let bindings are used in expressions:

```
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

```
(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
```


Wildcards

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

```
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:

```
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

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

Why does **'a** appear three times?

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

```
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

```
fun f x y = 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

```
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

```
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

```
fn : int -> int list
```

The function returned is equivalent to

```
fun h b = 1 :: [b];
```

```
val h = fn : int -> int list
```

Map Revisited

ML supports the **map** function, which can be defined as

```
fun map(f,[]) = []
  | map(f,x::y) =
      (f x) :: map(f,y);
```

```
val map =
 fn : ('a -> 'b) * 'a list -> 'b list
```

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

```
fun map f [] = []
  | map f (x::y) =
      (f x) :: map f y;
```

```
val map =
 fn : ('a -> 'b) ->
        'a list -> 'b list
```


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,

```
val neg = map not;
```

```
val neg =
  fn : bool list -> bool list
```

```
neg [true,false,true];
```

```
val it = [false,true,false] :
 bool list
```


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,

```
pow [1,2] =  [[1,2],[1],[2],[]]
```

We first define a version of **cons** in curried form:

```
fun cons h t = h::t;
```

```
val cons = fn :
  'a -> 'a list -> 'a list
```


Now we define **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 **pset**. Then the power set for **h::t** is just **h** distributed through **pset** appended to **pset**.

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

The complete definition is simply

```
fun pow [] = [[]]
 |  pow (h::t) =
    let
     val pset = pow t
    in
    (map (cons h) pset) @ pset
    end;
val pow =
  fn : 'a list -> 'a list list
```

Let's trace the computation of `pow [1,2]`.

Here `h` = `1` and `t` = `[2]`. We need to compute `pow [2]`.

Now `h` = `2` and `t` = `[]`.

We know `pow []` = `[[]]`, so `pow [2] =`
`(map (cons 2) [[]])@[[]]` =
`([[2]])@[[]] = [[2],[]]`

Therefore `pow [1,2] =`

```
(map (cons 1) [[2],[]])
 @[[2],[]] =
[[1,2],[1]]@[[2],[]] =
[[1,2],[1],[2],[]]
```