

Patterns

In Scheme (and most other languages) we need *access* or *decomposition* functions to access the components of a structured object.

Thus we might have

```
(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 $\tau_1 \rightarrow \tau_2$. This says that a parameter of type τ_1 is mapped to a result of type τ_2 .

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::t1,hd2::t12) =
    hd::append(t1,hd2::t12)
  | append(hd::t1,[]) =
    hd::t1;
val append = fn :
  'a list * 'a list -> 'a list
```