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).

\begin{verbatim}
fun isNull L = 
    if L = [] then true
    else false;

val isNull = fn : 'a list -> bool
\end{verbatim}

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

\begin{verbatim}
fun isNull [] = true
    | isNull(_::_:_) = false;

val isNull = fn : 'a list -> bool
\end{verbatim}
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.

```latex
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 \( \alpha, \beta, \ldots \) 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 :
  \alpha \text{ list} * \alpha \text{ list} \rightarrow \alpha \text{ list}
```

Why does \( \alpha \) 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 : \alpha \text{ list} * \beta \text{ list} \rightarrow \text{ bool}
```

Why are both \( \alpha \) and \( \beta \) 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

```ml
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

```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

\[ \text{fn : int -> int list} \]

The function returned is equivalent to

\[ \text{fun h b = 1 :: [b];} \]
\[ \text{val h = fn : int -> int list} \]
Map Revisited

ML supports the map function, which can be defined as

```ml
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

```ml
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 \texttt{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 \texttt{map} is that we can now use \texttt{map} to create “specialized” functions in which the function that is mapped is fixed.

For example,

\begin{verbatim}
val neg = map not;
val neg =
    fn : bool list -> bool list
neg [true,false,true];
val it = [false,true,false] : bool list
\end{verbatim}
Power Sets Revisited

Let’s compute power sets in ML.
We want a function $\text{pow}$ 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,

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

We first define a version of $\text{cons}$ in curried form:

fun $\text{cons}$ $h$ $t$ = $h::t$;
val $\text{cons}$ = fn :
    'a -> 'a list -> 'a list
Now we define \texttt{pow}. We define the powerset of the empty list, \texttt{[]}, to be \texttt{[][]}. That is, the power set of the empty set is set that contains only the empty set.

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

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

```ocaml
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 $\text{pow } [2]$.

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

We know $\text{pow } [] = [[]]$, so $\text{pow } [2] =$

$\text{(map (cons 2) [[]])@[]} =
\text{([[[2]]])@[]} = [[2],[[]]]$
Therefore $\text{pow } [1,2] = \\
(map \ (\text{cons } 1) \ [[2],[[]]]) \\
@[[2],[[]]] = \\
[[[1,2],[1]]@[[2],[[]]] = \\
[[1,2],[1],[2],[[]]]
Composing Functions

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

```plaintext
fun comp (f, g)(x) = f(g(x));
val comp = fn :
('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

In curried form we have

```plaintext
fun comp f g x = f(g(x));
val comp = fn :
('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

For example,

```plaintext
fun sqr x : int = x*x;
val sqr = fn : int -> int
comp sqr sqr;
val it = fn : int -> int
```
\begin{verbatim}
comp sqr sqr 3;
val it = 81 : int
\end{verbatim}

In SML \texttt{o} (lower-case \texttt{O}) is the infix composition operator.

Hence

\begin{verbatim}
sqr o sqr \equiv \text{comp} \ sqr \ s qr
\end{verbatim}
Lambda Terms

ML needs a notation to write down unnamed (anonymous) functions, similar to the lambda expressions Scheme uses.

That notation is

\[ \text{fn arg => body;} \]

For example,

\[ \text{val sqr = fn x: int => x*x;} \]
\[ \text{val sqr = fn : int -> int} \]

In fact the notation used to define functions,

\[ \text{fun name arg = body;} \]

is actually just an abbreviation for the more verbose

\[ \text{val name = fn arg => body;} \]
An anonymous function can be used wherever a function value is needed.

For example,

```plaintext
map (fn x => [x]) [1,2,3];
val it = 
[[[1],[2],[3]] : int list list
```

We can use patterns too:

```plaintext
(fn [] => []
 | (h::t) => h::h::t);
val it = fn : 'a list -> 'a list
```

(What does this function do?)