Functions

1. Keyword `fun`.
2. Function name and formal parameters.
3. `=`
4. Expression giving the value returned by the function.

Example:

```plaintext
fun cube(x:int) = x*x*x;
val cube = fn : int -> int
```

- Note description of the value of `cube`.
  - It is described as a “fn” (function).
  - Its type is given as `int -> int`. The `->` means “function from...to.” I.e., “`cube` is a function from integers to integers.”

- Note inconsistency: for a nonfunction, ML responds with its value; for a function, the type is given (but how could ML describe the “value” of a function save by repeating the code?).

- To avoid “overloading” the `*` operator, we have to tell ML that `x` is an integer.
  - The colon operator attaches the type `int` to `x`.
  - Beware: the colon has very low precedence, so parentheses surrounding the variable and the type are needed.
  - Otherwise, the parentheses around the parameter of `cube` are optional; the following is legal:

```plaintext
fun cube x = (x:int)*x*x;
```

Invoking Functions

After defining a function, it can be used as an operand of an expression.
cube(9);
val it = 729 : int

Patterns

Much of ML’s power comes from its ability to describe functions as a series of patterns that its input arguments might meet, with an expression describing the result in each case.

- The first pattern that matches the input “wins” and determines the result.

Example:

```ml
fun member(x, nil) = false
| member(x, y::ys) = 
  if x=y then true
  else member(x, ys);
val member = fn : "a * "a list → bool
```

- Note type: input (domain) is a pair consisting of an element of some type , and a list of elements of that type. Output (range) is a boolean.

- The double apostrophes in the type name indicates it is an equality type, one for which “=” must make sense.

- Warning: it is tempting to write the pattern member(x,x::xs) to catch the case where the element x is found at the head.

- But we may not use a variable twice in a pattern.

Example:

```ml
fun fact 0 = 1
| fact n = n*fact(n-1);
val fact = fn : int → int
```

- Note that a pattern can be an integer constant.

- Because function application has higher precedence than binary operators like -, we need parens in fact(n-1) although they are not needed in fact n.
(Fairly) General Form of Function Declarations

1. Keyword \texttt{fun}.

2. One or more expressions of the form "pattern = expression," separated by vertical bars.
   a) Pattern = function name + parameters. Each parameter may be an expression.
   b) The expression may use the variables that appear in the parameters.

Local Environments

the \texttt{let...in...end} construct allows us to make local, or temporary declarations using \texttt{val} or \texttt{fun}.

- These declarations go away after the \texttt{end}.

Example: The power set of a set \( S \) is the set of all subsets of \( S \). If sets are represented by lists, then the power set of a set of integers is of type \texttt{int list list}.

The following is a useful function that prepends \( x \) to each list on a list of lists \( L \).

\begin{verbatim}
fun pre(x,nil) = nil
  | pre(x,L::Ls) = (x::L)::pre(x,Ls);
val pre = fn 'a * 'a list list \rightarrow 'a list list
pre(1, [[2,3], [4,5], []]);
val it = [[1,2,3], [1,4,5], [1]] : int list list
\end{verbatim}

The following power-set function uses \texttt{pre} and computes the power set of the tail (a set with one fewer element) recursively. It uses that power set twice, once as-is and once with the head element prepended.

\begin{verbatim}
fun pow([]) = [[]]
  | pow(x::xs) =
    let
      val ps = pow(xs);
    in
      ps @ pre(x,ps)
    end;
val pow = fn 'a list \rightarrow 'a list list
\end{verbatim}
In general, a list of declarations, optionally ended by semicolons, may appear between let and in.

- Common errors: omitting val or end.

Variables may also be defined by a pattern — see split on p. 66, EMLP.

**Example:** Given a list, produce the minimum and maximum of the list of integers.

- Note the result is a pair; most languages only let you produce pointers to structures such as pairs.

```ocaml
fun minmax([x:int]) = (x,x)
| minmax(x::xs) =
  let
    val (low, high) = minmax(xs);
  in
    if x<low then (x,high)
    else if x>high then (low,x)
    else (low,high)
  end;
```

```ocaml
val minmax = fn : int list -> int * int
minmax([3,4,5,1,6,2,7,5]);
val it = (1,7) : int * int
```

- Notice that the pattern \([x:int]\) (or just \([x]\)) matches only a list of length 1 and binds \(x\) to the one element of that list.

- ML correctly discovers that minmax has no pattern that covers the empty list.

\[\Box\] Since minmax makes no sense on [], we should handle this problem with an “exception” as in Ch. 8.