1 Lists

Though we can nest pairs or tuples as deep as we want, their types always specify exactly how many parts they have. That is often too restrictive: we might need a list of data and the length of the list might be unknown at the time the program is written. ML has lists, which help solve this problem. Lists are more flexible than pairs, because they can have any length; but also less flexible, because all the elements of a list must have the same type.

The empty list, with syntax [], has 0 elements. It is a value, so like all values it evaluates to itself immediately. It can have type \texttt{t list} for any type \texttt{t}, which ML writes as \texttt{'a list} (pronounced “quote a list” or “tick a list” or “alpha list”); the latter pronunciation is good style). In general, the type \texttt{t list} describes lists where all the elements in the list have type \texttt{t}. The empty list [] has no elements, so it has type \texttt{t list} no matter what \texttt{t} is.

A non-empty list with \textit{n} values is written \texttt{[v1,v2,\ldots,vn]}. You can make a list with syntax \texttt{[e1,\ldots,en]}. Each expression is evaluated to a value, yielding a list of values, which is itself a value.

A more common syntax to create a list is \texttt{e1 :: e2}, pronounced “\texttt{e1 cons e2}.” (Or “\texttt{e1 consed onto e2}”. The word \texttt{cons} is from LISP, the granddaddy of all functional languages, including ML.) If \texttt{e1} evaluates to an \texttt{element} of type \texttt{t}, and \texttt{e2} evaluates to a list of elements of type \texttt{t}, then the result of evaluating \texttt{e1 :: e2} is a new list whose first element is the result of \texttt{e1} and whose remaining elements are the elements of \texttt{e2}.

How can we get elements out of lists? For now we will use three functions provided by ML:\footnote{ML programmers pronounce \texttt{'a} as “alpha”, \texttt{'b} as “beta”, \texttt{'c} as “gamma”, etc.}

- \texttt{null} evaluates to \texttt{true} for empty lists and \texttt{false} for nonempty lists.
- \texttt{hd} returns the \textit{head}—that is, the first element—of a list, raising an exception if the list is empty.
- \texttt{tl} returns the \textit{tail} of a list (everything except the head), raising an exception if the list is empty.

That’s the first we’ve heard of \textit{exceptions} in ML. We’ll learn more about them later.

Here are some simple examples of functions that take or return lists:

\begin{verbatim}
fun sum_list (lst : int list) = 
  if null lst 
  then 0 
  else hd(lst) + sum_list(tl(lst))

fun countdown (x : int) = 
  if x=0 
  then [] 
  else x :: countdown(x-1)

fun append (lst1 : int list, lst2 : int list) = 
  if null lst1 
  then lst2 
  else hd(lst1) :: append(tl(lst1), lst2)
\end{verbatim}

\footnote{As with pairs, we’ll later learn a much better way to \textit{destruct} (take apart) lists by using pattern matching.}
Functions that make and use lists are almost always recursive because a list has an unknown length. To write a recursive function, the thought process involves thinking about the base case (what should the answer be for an empty list?) and the recursive case (how can the answer be expressed in terms of the answer for the rest of the list?).

When you think this way, many problems become much simpler in a way that surprises people who are used to thinking about using lists with while loops and assignment statements. A great example is the append function above that takes two lists and produces one list appended to the other. This code implements an elegant recursive algorithm: If the first list is empty, then we can append by just evaluating to the second list. Otherwise, we can append the tail of the first list to the second list. That is almost the right answer, but we need to “cons on” the first element of the first list. There is nothing magical here. We keep making recursive calls with shorter and shorter first lists, then, as the recursive calls complete, we add back on the list elements removed for the recursive calls.

We can combine pairs and lists however we want without having to learn any new features of ML. For example, here are several functions that take a list of pairs of integers:

```ml
fun sum_pair_list (lst : (int * int) list) = 
  if null lst
  then 0
  else #1 (hd(lst)) + #2 (hd(lst)) + sum_pair_list(tl(lst))

fun firsts (lst : (int * int) list) = 
  if null lst
  then []
  else (#1 (hd lst))::(firsts(tl lst))

fun seconds (lst : (int * int) list) = 
  if null lst
  then []
  else (#2 (hd lst))::(seconds(tl lst))

fun sum_pair_list2 (lst : (int * int) list) = 
  (sum_list (firsts lst)) + (sum_list (seconds lst))
```

The last function reuses earlier functions to allow for a very short solution. This is very common in functional programming. In fact, it should bother you that firsts and seconds are so similar but don’t share any code. We’ll learn how to fix that, soon.

## 2 Let expressions

A let expression enables local variables. In fact, it enables have local bindings of any sort, including function bindings. Because let is an expression, it can appear anywhere other expressions can.

Syntactically, a let expression is:

```
let b1 b2 ... bn in e end
```

where each $b_i$ is a binding and $e$ is an expression.

The type-checking and evaluation rules of a let expression are much like the those of the top-level bindings we’ve seen so far. For evaluation, each binding is evaluated in order, progressively creating a larger environment for the subsequent bindings. So we can use all the earlier bindings in the later ones, and we can use them all in $e$. The scope of a binding is where it can be used, so the scope of a binding in a let expression
is the later bindings in that \texttt{let} expression and the \texttt{body} \texttt{e} of the \texttt{let} expression. The value \texttt{e} evaluates to is the value of the entire \texttt{let} expression. For type checking, each binding is checked in order. Each extends the static environment with a new name and its type. Finally, \texttt{e} is type checked, and the type of the entire \texttt{let} expression is the type of \texttt{e}.

For example, this expression evaluates to 7; notice how one inner binding for \texttt{x} \textit{shadows} an outer one.

\begin{verbatim}
let val x = 1
 in (let val x = 2 in x+1 end) + (let val y = x+2 in y+1 end)
end
\end{verbatim}

Also notice how \texttt{let} expressions are expressions so they can appear as a subexpression in an addition—though this example is silly and bad style.

We can use \texttt{let} expressions to bind functions too, since functions are just another kind of binding. For example, here we use a local helper function to help produce the list $[1,2,\ldots,x]$:

\begin{verbatim}
fun countup_from1 (x:int) = 
 let fun count (from:int, to:int) = 
   if from=to then [to] 
   then [to]
   else from :: count(from+1,to)
 in count(1,x)
end
\end{verbatim}

If a helper function is needed by only one other function and is unlikely to be useful elsewhere, it is good style to bind it locally, as we did with \texttt{count}. Inside that locally-bound function, the bindings from the outer function are in scope. The code above didn’t make use of that: it didn’t use \texttt{x}. But it could! That means \texttt{count} doesn’t need \texttt{to} at all, since (in the code above) \texttt{to} always has the same value as \texttt{x}. So we can rewrite \texttt{count}:

\begin{verbatim}
fun countup_from1_better (x:int) = 
 let fun count (from:int) = 
   if from=x then [x] 
   then [x]
   else from :: count(from+1)
 in count 1
end
\end{verbatim}

This technique—defining a local function that uses other variables in scope—is a hugely convenient thing to do. It’s very common in functional programming. Unfortunately, many non-functional languages have little or no support for doing something like it.

Local variables are often good style for keeping code readable. And they can be even more useful when they bind to the results of potentially expensive computations. For example, consider this code, which does not use \texttt{let} expressions:

\begin{verbatim}
fun bad_max (lst : int list) = 
 if null lst then 0
 else if null (tl lst)
then hd lst
else if hd lst > bad_max(tl lst)
  then hd lst
  else bad_max(tl lst)

If you call bad_max with countup_from1 30, it will make approximately $2^{30}$ (over one billion) recursive calls to itself. The reason is an “exponential blowup” — the code calls bad_max(tl lst) twice and each of those calls call bad_max two more times (so four total) and so on. This sort of programming “error” can be difficult to detect because it can depend on your test data (if the list counts down, the algorithm makes only 30 recursive calls instead of $2^{30}$).

We can use let expressions to avoid repeated computations. This version computes the max of the tail of the list once and stores the resulting value in tl_ans.

fun good_max (lst : int list) = 
  if null lst 
  then 0 
  else if null (tl lst) 
  then hd lst 
  else 
    (* for even better style, bind hd lst *) 
    let val tl_ans = good_max(tl lst) 
    in 
    if hd lst > tl_ans 
    then hd lst 
    else tl_ans 
    end

3 Options

The two max functions we just wrote don’t properly handle the empty list—they return 0, which isn’t really the maximum value of 0 numbers. Could we do better? Mathematically, the best answer is negative infinity. In a program, another possibility is to raise an exception.

Still another possibility is to change the return type to somehow indicate whether the function is returning the maximum element, or whether the list was empty. We could “code this up” by returning an int list, using [] if the input was the empty list and a list with one integer (the maximum) if the input list was not empty.

That works, but it’s a “hack”. A list is not really a precise description of what we are returning. ML has options which are a precise description: an option value has either 0 or 1 “things” inside it. NONE is an option value “containing” nothing, SOME e evaluates e to a value v and becomes the option carrying the single value v. The type of NONE is 'a option, and the type of SOME e is t option if e has type t.

Just like ML has null to determine whether a list is empty, it has isSome, which evaluates to false if its argument is NONE, and true otherwise. And just like ML has hd and tl to get parts of lists (raising an exception for the empty list), it has valOf to get the value carried by SOME, and raising an exception for NONE.

Using options, here is a better version of max. Its return type is int option.

fun better_max (lst : int list) = 
  if null lst 
  then NONE 
  else
let val tl_ans = better_max(tl lst)
in if isSome tl_ans andalso valOf tl_ans > hd lst
  then tl_ans
  else SOME (hd lst)
end

The version above works just fine and is a reasonable recursive function because it does not repeat any potentially expensive computations. But it is both awkward and a little inefficient to have each recursive call except the last one create an option with SOME just to have its caller access the value underneath. Here is an alternative approach where we use a local helper function for non-empty lists and then just have the outer function return an option. Notice the helper function would raise an exception if called with [], but since it is defined locally, we can guarantee that will never happen.

fun better_max2 (lst : int list) =
  if null lst
  then NONE
  else let (* fine to assume argument nonempty because it is local *)
    fun max_nonempty (lst : int list) =
      if null (tl lst) (* lst better not be [] *)
      then hd lst
      else let val tl_ans = max_nonempty(tl lst)
        in
          if hd lst > tl_ans
          then hd lst
          else tl_ans
        end
      in
        SOME (max_nonempty lst)
      end
end