Main topics of the week:
- Patterns in function definitions
- Local environments, scope and storage
- Data types in ML
- Recursion

Patterns in Function Definitions

As we have seen, functions can use conditional expressions to deal with various cases. However, ML has a very powerful construct to allow it to deal with various cases, and this technique is particularly useful for quickly coding interpreters. The construct is patterns. In a way, it is similar to function overloading in C++ or Java, where the signatures for functions consist of the name, parameter type list, and return type. However, in ML functions, patterns are only used to parse into different forms – the types of all the patterns must still be the same. (When we get to data types we’ll see how to combine different types into a single type in a C union like way.)

A common pattern is a list – where we represent the parameter as x::xs, i.e., an item prepended to a list of items. To match this pattern we would have to have a list, and this pattern then essentially parses the list to allow us to refer to the first element of the list. Another pattern might be the empty list. Other patterns could be tuples of various size and various types. Again the power of the pattern is having the interpreter match the pattern and parse it into named parameters that appear in the function body. Patterns need not be mutually exclusive – they are matched in the order given. Completely redundant patterns are an error. All cases need not be covered, although this might generate a warning, and at runtime could result in an error.

Patterns can be constant values or variables or list constructors. There is even a wild card, denoted by ‘_’ to match a value we don’t care about.

The general form of patterns is:

fun identifier (pattern1) = expression1
  | identifier (pattern2) = expression2
  | identifier (pattern3) = expression3
  ...

An example of patterns in function definitions:

- fun reverse(nil) = nil
  
  =   | reverse(x::xs) = reverse(xs) @ [x];
  val reverse = fn : 'a list -> 'a list
  - reverse([]);
  Warning: type vars not generalized because of
     value restriction are instantiated to dummy types
  val it = [] : ?.X1 list
  - reverse([1,2,3,4]);
  val it = [4,3,2,1] : int list
  - reverse(reverse([1,2,3,4]));
  val it = [1,2,3,4] : int list

- fun elt(1, x::xs) = x
  
  =   | elt(i, L) = elt(i-1, tl(L));
val elt = fn : int * 'a list -> 'a
- elt(1, [1,2,3,4]);
val it = 1 : int
- elt(3, [5,6,7,8]);
val it = 7 : int

Note that in the second example, we use the constant integer value 1 in the pattern for the base case.

Local Environments

Just as in C, Java, C++, and many other programming languages, ML permits the creation of blocks. We can think of these as local contexts in which variables are scoped just to the block and hide variables in an outer scope. In the ML world, we think of this as adding to the environment whenever we see a variable declaration. With the construct of blocks, the variable declaration only has a limited lifetime, and cannot be seen beyond the end of the block, so this essentially permits the temporary addition of a variable to the environment and its subsequent removal.

The construct is implemented with the keywords let, in, and end, denoting the beginning of declarations, the body in which they are effective, and the end. This is not a statement block as in imperative languages – after all, ML is functional and everything is expression based. The let construct is an expression – the value of the expression is the body between the in and end. The declarations part can have multiple declarations; termination by semi-colons is optional.

Let expressions can be used to provide convenient local names for variables or functions. They can also be used to perform common subexpression calculations and give names that make a complex expression easier to understand.

- fun volume(r,h) =
  =   let fun square(x:real) = x*x;
  =     val pi = 3.14159;
  =   in
  =     pi * square(r) * h
  =   end;
val volume = fn : real * real -> real
- volume(3.0, 2.0);
val it = 56.54862 : real
- pi;
Error: unbound variable or constructor: pi
- square(2.0);
Error: unbound variable or constructor: square

Note that the function square and the variable pi are not able to be referenced outside of the let expression. They are only temporarily in the environment for the volume calculation.

Variable Scope

Consider the following
1   let val x = 5
2       fun f y = x - y
3   in
4     let val x = 3

Atkins – Spring 2006
Note we have labeled each line so we can refer to it. In the right hand side of line 2, we say that y is a **bound** variable (it is bound to the formal parameter of f on the left side) and that x is a **free** variable, since it is not bound to anything. However, when we evaluate the expression in line 6, we must first figure out what the value f is, that is, we must resolve the free variable x in the definition of f (from line 2). We look at line 2 and find the closest enclosing let block and we see the value for x in line 1. This is called **static binding**. Likewise, the x that appears in line 6 will bind to the nearest x, the one on line 4. This behavior is called **static scope**. In fact, the code above evaluates to 2, showing the different bindings of x. The choice of name for x in lines 4 and 6 is not important, we will get the same results if we replace those occurrences of x by x1.

Contrast this to **dynamic scope**. If ML implemented dynamic scope, then the x in line 2 would actually be bound to the x in line 4. That is, the binding is deferred to execution. Thus the program would return 0 if dynamic scope was used. Lisp is a dynamic scope language (which was actually an error in its design) and Scheme fixed this to have static scope. If we tried the renaming of x suggested, we would see the result change under dynamic scoping. It is generally accepted that static scope has fewer surprises to the programmer and is the better way to go, although there are some who argue for dynamic scope. In fact, objects in Java have some notion of dynamic scope.

**Storage**

We’ll now consider some aspects of storage. In block structured languages (like Algol, ML, C, Java,...) there is a notion of a block. (let..end in ML, {} in C). And nesting is permitted for blocks. Questions come up about when we allocate space for variables that appear inside blocks. The basic idea here is that there is an activation frame for each block and the blocks are chained (actually a stack) as we nest blocks, with the chain being adjusted as blocks are entered and exited. Consider the following bit of code:

```
1   let val z = 25
2       val w = z+1
3   in
4       let val y = z + w
5           val w = 1
6   in
7       y + w
8   end
9 end
```

Note that w appears in both blocks. The appearance of w in the inner block creates a **hole in scope** for the outer w, i.e., the outer w visibility is obscured by the inner w. However, note that this hole begins with the inner w declaration. That is, at line 4, the outer w is visible and used in the initialization of y. But from lines 5 through 8, the outer w is hidden. In fact the multiple declarations in the let block (lines 4 and 5) are really syntactic sugar for nested let blocks. Anyway, the compiler builds activation frames for each let block and pushes these on a stack as execution proceeds. When a variable’s value is needed at runtime, we look back through the activation frames for the variable, stopping as soon as we find it, and this effectively hides the outer w from line 7.
If you were going to implement an interpreter, what sort of information would you need to keep in each activation frame? For the simple let blocks above where we have no functions, we need only keep the variables declared and their values (and their type, of course). These become part of the environment and when we need to resolve a variable, we just look back through the activation records constituting the environment to find the desired variable – that is, if we are implementing static scope. Do we actually need to do this search through the activation records at runtime? Not really, for if we don’t have procedures, and we have static scoping, at “compile” time, we can just encode the mapping by way of offsets into the activation frames – e.g., number of frames back and offset into that frame. Size of types is of course important here – if we have dynamically sized types, e.g., dynamic arrays, then the activation frame could be used for a pointer or some mechanism to refer to heap storage for the dynamically determined part. Essentially the activation frame would have a control structure.

What about if we add functions? With functions, we’ll need to add some additional information to the activation frame. We will need the parameter list so we can store the values passed to the function. We will also need some storage for the return value of the function. In C, where a function is a new context block, we would need the local variables for the function as well.

Data Types in ML

Types are defined in ML in a way similar to unions in C. There are simple synonyms for types using the keyword type that is like the typedef of C. But the more interesting thing is actually defining a new type using the datatype keyword. A datatype definition consists of type constructors and data constructors. The type constructor is just the name of the datatype being defined. The data constructors are identifiers that behave as symbolic operators to build values of the new datatype. First we look at a simple use of datatype to define enumerated types as in C. These are also called algebraic types or union types. Here the data constructors can be thought of as functions that take no arguments and just return a value. These are disjunctive values and such disjuncts are also called tags.

```ml
- datatype Color = Red | Yellow | Blue;

Consider a function using this defined type.

```ml
- fun f Red = true
  = | f Yellow = false
  = | f Blue = true;

This can also be written using a case construct (or nested if then else):

```ml
- fun f x = case x of Red => true
  = | Yellow => false
  = | Blue => true;

val it = true : bool
```
If we eliminate the last case, then we do not have exhaustive cases for color. This is not a compile error, but produces a runtime error if \( f \) is applied to a value \( \text{Blue} \).

- \( \text{fun } f \text{ Red} = \text{true } | f \text{ Yellow} = \text{false} ; \)

  Warning: match nonexhaustive
  Red => ...
  Yellow => ...

  \( \text{val } f = fn : \text{Color} \rightarrow \text{bool} \)
  - \( f \text{ Blue} ; \)

  uncaught exception nonexhaustive match failure

Representation of disjuncts will use some bits for keeping track of things, but has no numerical interpretation (unlike C). I.e., types are very important and distinct and there is no relation between color and \( \text{int} \). A compiler needs to know types for memory allocation purposes and static type checking, but the runtime also needs type information to check things like the legality of disjuncts.

Data constructors need not be so simple; they may take arguments. This provides a mechanism where we could have, for example, integer values associated with the new type, but in a way that could not be confused with plain integers. That is, in the case of enums in C, one can use integers whenever an enum is expected and this can lead to program problems. Even when the C compiler enforces explicit type matching, we can still use type casting to “create” an enum from an integer. In ML, we must always be explicit – there is only a correlation between an integer and a type if there is a constructor for the type that takes an integer. Such data constructor expressions use the keyword \textit{of}. They may be thought of as functions taking an argument of the type indicated in the constructor expression – essentially becoming a wrapper for the value.

Here is a simple example of defining a numeric type with constructors for \( \text{int} \) and \( \text{real} \) and then defining a function that takes and returns the numeric type.

- \( \text{datatype } \text{num} = \text{Int of int } | \text{Real of real} ; \)
  \( \text{datatype } \text{num} = \text{Int of int } | \text{Real of real} \)
- \( \text{fun } \text{square } (\text{Int } n) = \text{Int}(n*n) \)
  = \text{square } (\text{Real } x) = \text{Real}(x*x) ; \)
  \( \text{val } \text{square} = fn : \text{num} \rightarrow \text{num} \)
- \( \text{square } 2 ; \)

  Error: operator and operand don't agree
  - \( \text{square } (\text{Int } 2) ; \)
  val it = \text{Int} 4 : \text{num} \\
  - \( \text{square } (\text{Real } 3.0) ; \)
  val it = \text{Real} 9.0 : \text{num} \\

Note that we must be explicit in using the constructor when we pass a value to the \text{square} function. That is, ML wants to make sure we know what we are doing in terms of types and will not take shortcuts (as in an overloaded function in C++) of matching a type to a constructor. However, this simple example does show that we can define a function to deal with a generalized notion of a numeric type – to use it we will always have to be passing this generalized type \( \text{num} \), but in the context of a larger body of code, that would tend to happen naturally.

We can also use certain operators as type constructors. We could define a list of integers by

- \( \text{datatype } \text{List} = \text{nil } | \text{cons of int } * \text{List} ; \)
  \( \text{datatype } \text{List} = \text{cons of int } * \text{List} | \text{nil} \)
and we could define a type to capture the idea of single values or pairs of heterogeneous values as follows:

```
datatype ('a,'b) element = Pair of 'a * 'b | Single of 'a

fun count(nil) = 0
= count(Single(x)::L) = count(L)
= count(Pair(x,n)::L) = n + count(L);
val count = fn : ('a,int) element list -> int
val it = 6 : int
```

Again, it may look cumbersome to have to specify the type constructors, but in the context of a larger program, these types would be likely to be constructed by reading some input, so would not appear explicitly. This example shows how we might have a list of elements that sometimes have counts associated with the elements, but not always, yet we are able to design a function that takes that difference into account.

### Recursive Functions

Although ML does have a looping construct, the functional paradigm of the language means that recursion is the preferred way for iterative flow control. Recall that recursion simply means a function calling itself, and as noted earlier, this works by reduction. That is, we establish a base case behavior for a function (i.e., behavior for parameter value(s) in some minimal boundary situation) and also implement behavior for other values that reduce to a “lesser” value so that we eventually terminate in the base case.

### Linear Recursion

Linear recursion is the simplest form of recursion, where the recursive case reduces directly to the function with a lesser value. A good example of this is the factorial function, where we want to compute the factorial value for an integer. This value is defined as the product of all the integers from 1 up to and including the argument integer value. The base case is for the simple value of 1, where the factorial is simply 1. For any other integer n, we can reduce the computation of its factorial to the product of the integer with the factorial of the previous integer. Another example is integer exponentiation – finding the value of a number raised to a power. This is very much like proof by induction (and in fact could be called an inductive definition of factorial) where we assume the truth of all previous cases and use these facts to prove the desired case. The code in ML to implement factorial looks like:

```
fun fact(n) = if n = 0 then 1 else n * fact(n-1);
val fact = fn : int -> int
val it = 1 : int
val it = 6 : int
val it = 3628800 : int

fun pow(x, n) = if n = 0 then 1 else x * pow(x, n-1);
val pow = fn : int * int -> int
```
Certainly factorial numbers and powers increase very quickly – we’ll overlook that issue for the time being. Recursion is useful for dealing with lists and generally any problem that can be viewed as reducing to simpler cases.

**Non-linear Recursion**

Not all recursion need be this simple, modeled after straight induction. As in some induction proofs, we need to appeal not just to the truth of the immediate predecessor case, but to the truth of several of the predecessor cases. For example, the choose function that calculates the number of ways you can choose \( m \) objects out of a set of \( n \) objects can be given by the formula \( \frac{n!}{m!*(n-m)!} \). Let’s see what this is like as an ML function:

```ml
- fun comb(n,m) = fact(n) div (fact(m)*fact(n-m));
  val comb = fn : int * int -> int
- comb(4,2);
  val it = 6 : int
- comb(10,3);
  val it = 120 : int
- comb(20,5);
  val it = uncaught exception overflow
```

```
- fun comb(n,m) = if m=0 orelse m=n then 1 else comb(n-1,m) + comb(n-1,m-1);
  val comb = fn : int * int -> int
- comb(10,3);
  val it = 120 : int
- comb(20,5);
  val it = 15504 : int
- comb(30,6);
  val it = 593775 : int
```

Note that our first simplistic approach just tries to use the formula and the already defined factorial function, but quickly runs out of gas. We observe that to choose \( m \) things out of \( n \), we could set the first item aside and choose \( m \) out of the remaining, and include the first item while choosing \( m-1 \) items out of the remaining. This reduces in two ways to lesser combinations, so we make our problem simpler. By breaking the problem down this way, we do not hit the limits of the integer storage as quickly since we are only performing addition, not multiplication to a massive number that we subsequently divide. By the way, you can code this in C just as easily as in ML (and get faster execution).

**Mutual Recursion**

Another form of recursion is mutual recursion. This is a more complicated scenario where you reduce from one function to another and vice-versa. The reduction might not be as apparent here as it seems the real issue is cyclic calls: if \( f \) calls \( g \) and \( g \) calls \( f \), how do we ever get one or the other defined? Well, ML has a mechanism – this is the use of the keyword **and** (the reason
we have to use the awkward and also for conditionals). In the following example, we want to define a function ‘odds’ that takes a list and produces a list with all the odd elements, i.e., the first, third, fifth, etc. Now this would be easy recursion if we had a function to get the even elements since we could just take the first element and prepend it to the even elements of the tail of the list. Likewise we could define the evens in a similar way, taking the odd elements of the tail of a list. In each case we would have reduced to a smaller list, with the base case for both being the empty list. Note the construct in ML used to get the interpreter to intertwine the definitions of the two functions:

- fun odds(L) = if L = nil then nil else hd(L)::evens(tl(L));
  Error: unbound variable or constructor: evens
- fun evens(L) = if L = nil then nil else odds(tl(L));
  Error: unbound variable or constructor: odds

- fun odds(L) = if L = nil then nil else hd(L)::evens(tl(L))
  = and evens(L) = if L = nil then nil else odds(tl(L));
val odds = fn : ''a list -> ''a list
val evens = fn : ''a list -> ''a list
- odds([1,2,3,4,5]);
val it = [1,3,5] : int list
- evens([1,2,3,4,5]);
val it = [2,4] : int list