Functional Programming

In languages like SML, functions are considered *first class* citizens. This means functions can be treated like any other variable. They can be passed as parameters to functions, returned by functions. They can be elements in a tuple/record, computed, stored, etc. Functions that accept functions as arguments and returns functions are known as *higher order functions*.

Functions can also refer to variables defined outside it self, but the variables are defined within the scope of the function. This type of functions are called *function closures*. Functional programming a style which involves the following:

1. Variables are immutable
2. Higher order functions and function closures
3. Recursive functions
4. Programming which is closer to math (`i = i + 1` would not make sense in math)
5. Lazy evaluation
6. etc…

A *functional programming language* is then a language in which functional programming is natural. Any language that enables immutable variables, higher order functions and function closures can be considered "functional". Most modern programming languages like Scala, Java, Python, etc., support functional programming along with other paradigms like object oriented programming.

Functions as Arguments

Consider the situation where a bunch of functions need to be implemented. The functions are very similar in the sense that they all take the same input parameters, perform a similar sequence of operations but only differ in how the return value is computed. Take a simple example where the salary increment needs to be compiled: smallIncrement, mediumIncrement, largeIncrement. See the initial implementations:

```
fun smallIncrement(ls: real list) =
  if null ls then []
  else
    ((hd ls) * 1.1) :: smallIncrement(tl ls);

fun mediumIncrement(ls: real list) =
  if null ls then []
  else
    ((hd ls) * Math.log10(hd ls)) :: mediumIncrement(tl ls);

fun largeIncrement(ls: real list) =
  if null ls then []
  else
    (hd ls * hd ls) :: largeIncrement(tl ls);
```

All the above functions accept a list of `real` values. They all check if the list is empty and returns an empty list if true. If the list passed in is not empty, the increment for the first element in the list is calculated and prepended to the list of increments for the tail of the list computed

recursively. There is a lot of duplication of code. The caller is limited to 3 algorithms to calculate the increment. If a new algorithms are needed by the caller, then additional functions will need to be implemented, so this is not very scalable solution. So what if we could write a function that not only takes the list but also an additional parameter that passes in the calculation to be done? This is where the higher order functions come in. The above functions can be re-written as:

```
fun increment(incrementer, ls) =
  if null ls then []
  else incrementer(hd ls) :: increment(incrementer, tl ls);

fun smallIncrementer(x: real) = x * 1.1;

fun mediumIncrementer(x: real) = x * Math.log10(x);

fun largeIncrementer(x: real) = x * x;
```

In the above implementation the incrementing calculation is captured in function `smallIncrementer, mediumIncrementer` and `largeIncrementer`. The actual increment function accepts an "incrementer" in addition to the actual list of salaries. Based on the type of increment needed, caller can pass the appropriate incremented as below:

```
increment (smallIncrementer, [3000.0, 1000.0, 2200.0]);
increment (mediumIncrementer, [3000.0, 1000.0, 2200.0]);
increment (largeIncrementer, [3000.0, 1000.0, 2200.0]);
```

If needed aliases can be created for the above as below:

```
fun smallIncrement2(ls: real list) = increment(smallIncrementer, ls);
fun mediumIncrement2(ls: real list) = increment(mediumIncrementer,
ls);
fun largeIncrement2(ls: real list) = increment(largeIncrementer, ls);
```

The power in the above implementations is that the caller does not need to be limited to the above incrementers. The caller can pass in custom incrementers to based on their need. Note that the power in these implementations comes from being able to write a function that accepts another function as a parameter (`increment` function) aka **Higher Order functions**. And functions that can reference bindings that are outside themselves as in (smallIncrement2, etc referencing smallIncrementer in their body (**function closures!**)

Polymorphic Types:
Try out the above examples in SML and notice the types associated with each function. Particularly the type of function `increment` will be shown as `fn : ('a -> 'b) * 'a list -> 'b list`. The function is pretty generic and is not even aware that it's being used to do calculations on real numbers. So in theory it would be passed a list of strings and return a list of numbers! We don't even have to write an "incrementer" we can just pass in the builtin `Real.fromString` as below:

```
increment (Real.fromString, ["3000.0", "1000.0", "2200.0"]);
```

which will return a `real option list`:

```
[SOME 3000.0,SOME 1000.0,SOME 2200.0]
```

Such functions are called polymorphic. Higher order functions need not be polymorphic and not all polymorphic functions are higher order functions! The function below (used for illustrative purpose) is higher order but not polymorphic. The REPL will show that the type for this function is `fn : (int -> int) * int -> int`.

```
fun do_something(f, i) =
  if i = 0 then 0
  else 1 + do_something(f, f i);
```

Anonymous Functions

In the above example, the helper functions `smallIncrementer, mediumIncrementer,` and `largeIncrementer` do not need to be written as stand alone functions. Instead anonymous functions can be used to implement smallIncrement2, mediumIncrement2 and largeIncrement2 as below:

```
fun smallIncrement3(ls: real list) = increment(fn x => x * 1.1, ls);
fun mediumIncrement3(ls: real list) = increment(fn x => x *
Math.log10(x), ls);
fun largeIncrement3(ls: real list) = increment(fn x => x * x, ls);
```

Take smallIncrement3 further analyze the anonymous function. An anonymous function is a function without a name, aka a **lambda**. In SML the syntax for declaring an anonymous function is as:

```
fn (<arguments>) => <expression>
```
e.g.:
```
val a = fn (i, j) => if i = 0 then 0 else i + j;
```

When using anonymous functions sometimes there is a tendency to unnecessarily use anonymous function to wrap another function. As in:

increment (fn x => Real.fromString(x), ["3000.0", "1000.0", “2200.0"]);

While the above code will work as expected, there is no need to wrap call to `Real.fromString`. Just call in the function from the builtin library as is. Functions are first class citizens in SML.

Maps and Filters

Map and filter are the most popular use cases for higher order functions. The increment function above is essentially a map function. It essentially “maps” each value in a list to another value and creates a new list of the mapped values. The map function can be re-written using case expression as below:

```
fun map(f, l) =
  case l of
  [] => []
  | ls::ls' => f ls :: map(f, ls')
```

Note: It's common convention to use a variable and it's "prime" to represent head and tail of a list in case expressions, as above with ls and ls' (read as "ls prime")

The above map function can be used to map a list of integers to their doubles and squares using anonymous functions as below:

```
val doubles = map (fn x => x + x, [12, 31, 55, 16, 613]);
val squares = map (fn x => x * x, [12, 31, 55, 16, 613]);
```

You can also nest calls to the map function to apply multiple mappings:

```
val logs = map (Math.log10, map (
  Real.fromInt,
  [12, 31, 55, 16, 613]
  ))
```

In the code above the inner call to map function (in bold) converts the integers to real numbers. The outer map in turn maps each real value to the logs of their numbers. As will be seen later this can be made even more concise by using function composition, given below for an initial look:

```
val logs = map (Math.log10 o Real.fromInt,
  [12, 31, 55, 16, 613]
  )
```

The other popular use of higher order functions is the filter function. A filter function is used to select only elements in a list that meet a certain requirement. For instance there might be a need to extract only even numbers from a list of integers. Given below is such a filter implemented in SML:

```
fun filter(f, l) =
  case l of
  [] => []
  | l::l' => if f l then l :: filter(f, l')
    else filter(f, l');
```

The type for the function filter above is `fn : ('a -> bool) * 'a list -> 'a list`. This function is polymorphic as we can see from the presence of the generic type `'a` above. The first argument is a function that take a value of type `'a` and returns a boolean value. The second parameter is an `'a list`. and the return value is also an `'a list`. As this function is generic, it can be used to filter lists of any type. Given below is an example where the filter function is used to filter for even numbers:

```
val evens = filter (fn x => x mod 2 = 0, [12, 31, 55, 16, 613]);
```

The anonymous function contains the logic used for the filter. `x mod 2` gives the remainder from the integer division with 2. If the remainder is 0 then the number is even and hence is a good check for evenness of the number. To show the generic nature of the filter function we can filter strings using the same filter:

```
val a_words = filter (fn s => String.isPrefix "a" s orelse
String.isPrefix "A" s,
  ["Fox", "Asterdam", "Atlas", "Box", "Fall", "Beetle"]);
```

Functions that return functions: If functions can accept functions as arguments then functions can also return functions. Given below is a bit of a contrived example for the illustration purpose:

```
exception InvalidArgument of real;

fun multiplier(r) =
  if r <= 0.0 then raise InvalidArgument(r)
  else if r > 1.0 then
    fn x => x * r
  else
    fn x => x / r;
```

The function accepts a real number. If the value of the real number is > 1.0 then the function returns a function that accepts one parameter (x) and return the product of it's own argument and the argument that is passed to the wrapper function. The multiplier can be used as below:

```
val res1 = multiplier(2.0)(3.0); (* return value of 6.0 *)
val res2 = multiplier(0.2)(3.0); (* return value of 15.0 *)
```

Note: The map and filter functions are the most popular uses of higher order functions but these can be used in any situation where part of the algorithm needs to be either passed into the function or returned from the function.

Lexical Scope

Most of programming languages use what is known as "Lexical Scope". To understand this, consider the code below:

```
val x = 1
fun f y = x + y
val x = 2
val y = 3
val z = f (x + y)
```

What would be the value of z be? It might appear that the answer should be 7 as below

```
val z = f (x + y)
        (* becomes f (2 + 3) -> 5 gets passed to function and in the
           body of the function x is added to the argument hence
           should be evaluated to 7, as x = 2 (?) *)
```

Contrary to what one might expect, the value assigned to z is 6. This is because though the variable x defined in line 1 is shadowed by the binding in line 3, the function f still keeps track of the initial binding of x, and uses that binding when evaluating the body. So as far as the body of the function f is concerned, the variable x is still bound to the value 1! This is the essence of "Lexical Scope". Formally it might be stated as:

*The body of a function is evaluated in the environment where the function is **defined**, not the environment where the function is **called**.*

Most modern programming use this type of scoping, which is also called static scoping. "Dynamic scoping" reverses this rule. Most modern languages shun "Dynamic scoping" as this

means that any function should account for all possible environments that this function can be called from! In the code above the value returned by function f can't be predicted, as binding of x might be changed in any way before the function is called. Lexical scoping makes it easier to reason about the function. We can say exactly that the function increments the value of y by 1 and returns the incremented value, because value of x is 1 for the function.

Environments and Closures

For the all the functions defined so far, only the type of the function has been discussed. The "value" of the functions is not discussed. The "value" of a function consists of the code and also the environment where the function is defined. These 2 aspects of the "value" of a function is called a *Closure* or *function closure*.

So for the function f above:

1. The code part of the function is `fn y => x + y` and
2. The environment has a binding of x to the value 1.

Passing Closures:

Consider a situation where you want to use the leverage the filter function above to filter numbers greater than a number specified by the caller, as below:

```
fun allGreaterThan (xs, n) = filter (fn x => x > n, xs)
```

The above function works because of lexical scoping. This is because though the variable n is not present in the filter function, it gets passed as a *closure* via the anonymous function which is bold above.

Fold

Another popular uses of higher functions is the fold function. This is also known as a reduce function. The map/filter/reduce functions go together in a lot of applications related to processing data, where these functions can be used to clean data, do aggregations, etc before passing on to other stages in the application. Given below is an implementation of the fold function:

```
fun fold (f, acc, ls) =
  case ls of
    [] => acc
    | ls::ls' => fold(f, f(ls, acc), ls');
```

The function above takes a function (f), and initial value (acc) that is built on during the process of recursion, and the list to apply the fold on. The function f is used to combine the head value with acc.

The above fold (or reduce) function can be used to implement a function that sums up all the numbers in a list as below:

```
fun sum(ls) = fold (fn (x, y) => x + y, 0, ls);
```

Calling the sum function as below gives the result 45:

```
val s = sum([1, 2, 3, 4, 5, 6, 7, 8, 9]);
```

Function Composition

Function composition is a way to combine 2 or more functions. Given two function `f()` and `g()` and there is a need to call these in a nested manner, as in `f(g())`. Using composition it's possible to create a composite function that does this, as in:

fun compose (f, g) = fn x => f (g x)

SML provides a builtin infix operator `o` to do function composition. The code below:

```
fun sqrt_of_abs i = Math.sqrt(Real.fromInt (abs i))
```

can be re-written as below:

```
fun sqrt_of_abs i = (Math.sqrt o Real.fromInt o abs) i
```

The above can be written more concisely as:

```
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

There is another way to compose functions, which is using using *pipeline* operators. In SML it's possible to define our own infix operator, as below:

```
infix |>
fun x |> f = f x
```

Once the infix operator is defined, sqrt_of_abs can be redefined in a more intuitive way as below.

```
fun sqrt_of_abs i = i |> abs |> Real.fromInt |> Math.sqrt;
```

This way of composing functions is more intuitive because it's easier think of passing results of one function to the next function and so on.

Creating fallback functions:
Consider a situation where 2 functions f, and g are used as below:

1. Call g with the argument passed
2. If it returned SOME value then return the value
3. If it returns NONE, then return the value returned by f

This may be implemented as below:

```
fun fallback(f, g) = fn x => case g x of
  NONE => f x
  | SOME y => y;
```

To illustrate the usage of the above fallback function, see the code below:

```
fun sqrt_options x = if x > 0.0 then
  SOME (Math.sqrt x)
```

```
  else NONE;
val v1 = fallback (abs, sqrt_options) (2.0);
val v2 = fallback (abs, sqrt_options) (~2.0);
```

v1 will get the value of the square root of 2.0. But as negative numbers don't have a square root, the function falls back to returning the absolute value of number.

Function Currying and Partial Functions

The term *currying* is named after Haskell Curry who was a mathematician who studied combinatory logic. Function currying comes from the idea of function that return functions. SML accepts multiple arguments as a tuple. There is however another elegant way to implement functions that accept multiple arguments not as tuples but multiple arguments. This is implemented by implementing functions that accept only one parameter but returns a function that accepts second parameter and so on. Look at the following 2 implementations of a function that multiplies 2 numbers:

```
fun multiplier1 (i, j) = i * j;

fun multiplier2 i = fn j => i * j;
```

Both return the same result but the way they are called is different. The types for both the functions are respectively given below:

```
fn : int * int -> int
fn : int -> int -> int
```

The code in bold highlights the difference. The function multiplier2 does not result a numeric result, but rather it returns an anonymous function which accepts the second parameter. Because of this, they are called differently as below:

```
val val1 = multiplier1 (3, 4);
val val2 = multiplier2 3 4;
```

SML provides syntactic sugar to make writing curried functions even more concise. The multiplier can re-written as:

```
fun multiplier2 i j = i * j;
```

In SML both the ways of writing the `multiplier2` are equivalent.

The advantage of the curried function is that they can be used to write *partial functions*, as below:

```
val doubler = multiplier2 2;
val d = doubler 4;
```

The function doubler can be also written as:

```
fun doubler x = multiplier2 2 x;
```

But this is considered as "Unnecessary function wrapping". This style is not preferred.

The advantages of currying can be summarized as below:

1. The parameters passed to a function can be logically grouped
2. The secondary (partial) functions can be re-used to create specialized functions
3. Functions can be re-used as above, without using lambdas

Note: SML provide built in map and filter functions. They are `List.map` and `List.filter`. They work just like the map and filter functions implemented above, but these use currying.

In most situations the function wrapping is not needed but sometimes SML might not be able to determine the types of the input parameters. e.g.:

```
val pair_with_one = List.map (fn x => (x, 1));
```

The above function will give a warning but is unusable. The work around to this is either

1. Use wrapper functions! as below:

```
fun pair_with_one_fun xs = List.map (fn x => (x, 1)) xs;
```

2. Or specify the function type explicitly:

```
val pair_with_one: int list -> (int * int) list = List.map (fn x =>
(x, 1));
```

The first work around will keep the generic nature of the function. The second work around ties the function to a specific type of list.