1 More ML

Q: How does `let` syntax work?

The `let` allows you to define variables and functions for a limited scope.

```
let
    val x = 5;        (* Define a variable *)
    fun f a = a + 1; (* Define a function *)
in
    f x
end
```

In the above code we are defining a variable `x` and a function `f` between the `let` and the `in` (Note the semicolons delimiting different expressions). We can then use them in the body of the let expression after the `in` and before the `end`. The value of the let expression will be the value of its body, in the above case it will equal 6.

The formatting is not significant so you may see the let used in a way that looks different, however the usage is the same. The below is equivalent to what is above.

```
let val x = 5; fun f a = a + 1; in f x end
```

Q: What is currying?

Currying is the process of taking a function that accepts multiple arguments and converting it to a function that accepts arguments one at a time. To do this, the function accepts a single argument and returns a new function waiting for the next argument.

```
fun addTuple (x, y) = x + y;    (* Not curried *)
fun addAnon x = fn y => x + y; (* Curried *)
fun add x y = x + y;            (* Curried with syntactic sugar *)
```

The function `addTuple` accepts two arguments in the form of a tuple (Technically still a single argument). The function `addAnon` is a curried version of `addTuple`. Notice how `addAnon` accepts a single argument `x` and returns a new single-argument function that will add `x` to the next argument. The `add` function is the same as `addAnon` using syntactic sugar provided by SML. With currying we can create functions on the fly and use them as needed. For example if we need a function that adds 5 to a number we can do:

```
fun add x y = x + y;
val add5 = add 5;
```

Now the `add5` function will add 5 to whatever is passed in as the argument.

Q: How do I read ML types?

Learning to read ML types can help with writing and debugging code. Seeing the type of the function you have written will help you understand if you are attempting to return the an expression of the wrong type or if a function has an error in it. Below is the compose function. The type is given in a comment above the function.

```sml
(* Takes in two functions, f and g, and returns a new function
    h(x) = f(g(x)) *)

(* ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b *)
fun compose f g = fn x => f (g x);
```

The single characters preceded by a single quote (`'a`, `'b`, etc...) are *generic types* these could be any concrete type (e.g. int, bool, ...) as long as all instances of the same generic type are the same.

The function type uses the arrow symbol (`->`) to separate the type of the argument from the type of the output. The arrow binds to the right so `int -> bool -> int` can be read as `int -> (bool -> int)` or "a function that takes in an `int` and returns a function that takes in `bool` and returns an `int`". More tersely: "A function from `int` to functions from `bool` to `int`.

In the compose function above the parenthesis are our guide. The first argument is a function, ML gives the type of the function as `('a -> 'b)` because it doesn't know what the input and output types are, this is the parameter `f` in the function definition.

Since the function is curried, after receiving the first argument, a new function is returned. The argument to the new function (or the second argument if you prefer) argument has type `('c -> 'a)`. Notice that SML figured out that the output of the second argument must be of type `'a` because it is being input into the first argument: `f(g(x))`.

Finally the compose function returns a new function, `'c -> 'b`. SML has elided the parenthesis here because they bind to the write implicitly.

Putting everything together the type of `compose` in English is: "A function that takes in a function from a to b and another function from c to a and returns a function from c to b".

Q: What is the unit type?

The unit type is a special type that has one value `()`. This is useful for functions that don't need an input and just produce a value.

Q: How does lazy evaluation work?

In order to simulate lazy evaluation in SML we need to hide the evaluation behind a function. We can use the `unit` type for this.

```sml
datatype 'a Seq = Cons of 'a * (unit -> 'a Seq);
```

Here we have an infinite sequence of values. The `Cons` constructor contains a tuple where the first value is the head of the sequence and the second value is a function that will produce the tail. By hiding the tail behind a function we can choose when to evaluate it thus simulating lazy evaluation.

```
(* An infinite sequence of 1s *)
val ones =
    let
        fun f() = Cons(1, f)
    in
        f()
    end;
```

Above is an infinite sequence of ones. We define a function `f` that takes in a argument of type unit and returns a new `Seq` with `1` as the head and itself as the tail function. Finally the `f` is called by passing the unit into it (`f()`) to produce the sequence.

```
(* The natural numbers *)
val natseq =
    let
        fun f n () = Cons(n, f(n + 1))
    in
        f 0 ()
    end;
```

Above is the list of natural numbers. Here the function `f` has two parameters, the first, `n`, becomes the new head of the list (`Cons(n, ...)`. While the second, the unit allows us to control when the function is evaluated. The body of `f` creates a new sequence using `n` as the head, and calls `f(n + 1)` as the tail function. This still has the appropriate type because `f` is curried so passing in the argument `n` gives us a function of type `unit -> int Seq`.

Finally the sequence is given by calling `f 0 ()` because 0 is the first natural number.

Now, remember the goal is to have lazy evaluation. In order to use this lazy list we will need to force the evaluation of our tail function. The following function gets a value at the nth index of the sequence s

```
fun get n (Cons (x, xs)) =
    if n = 0
    then x
    else get (n - 1) (xs ());
```

This function travels the sequence from front to back until `n` is 0 then it returns the head. Notice when we recur in the else clause when we not only decrement `n` but also pass `xs ()` as the new sequence. Since `xs` is of type `unit -> Seq` we can't pass it directly into our `get` function which is expecting a `Seq`. We need to call the function and get its return value, so we pass in the unit type. This is how we force the evaluation.