StateBuilder

This exercise works through the `StateBuilder`, in which we want to manage some state of a computation. The `StateBuilder` is a useful example as it is the generalized version of several of the popular computation expressions you'll find in F# libraries, e.g. Freya and Saturn.

1. Create a new file, `StateBuilder.fs`.
2. Add the file to your `.fsproj` with `<Compile Include="StateBuilder.fs" />` just below `ChoiceBuilder.fs`.
3. Add the following lines to the `StateBuilder.fs` file:

```
module States

open Expecto

type StateBuilder() = class end

let state = StateBuilder()
```

State

Let's first define the `State` type that will define our interactions. In order to understand how this type should be defined, let's look at an example of how we might use some state without a computation expression:

```
open System.Text

[<Tests>]
let tests =
    testList "states" [
        test "StringBuilder as state" {
            let printA (sb:StringBuilder) = sb.Append("A")
            let printB (sb:StringBuilder) = sb.Append("B")
            let printC (sb:StringBuilder) = sb.Append("C")
            let run (sb:StringBuilder) = sb.ToString()
            let sb = StringBuilder()
            let actual = sb |> printA |> printB |> printC |> run
            Expect.equal actual "ABC" "Expected ABC."
        }
    ]
```

We've defined four functions, each taking a `StringBuilder` and returning a `StringBuilder`, or in the case of `run`, a `string`. This allows us to nicely chain these functions together and then produce a result at the end. Your primary **observation** here should be that there's a pattern of passing a specific instance and getting it back in the result.

This pattern is often called a *Fluent Interface* in C# designs. In F#, the use of a pipeline typically indicates getting one thing and returning a new, modified instance of the same type, whereas the above sample nafariously hides state mutation with the same pattern. (See Fluent Interfaces are Evil for a relevant discussion.)

However, this is overly simplified, as we may not ***always*** want to `Append` to the `StringBuilder`, and we may want to compute a value, which may require retrieving the current value from the `StringBuilder`. With this in mind, we can define our type as:

```
/// State is a function type that takes a state and
/// returns a value and the new state.
type State<'a, 's> = 's -> 'a * 's
```

or

```
/// Single case union of the same type.
type State<'a, 's> = State of ('s -> 'a * 's)
```

depending on whether you prefer the single-case union style. The latter makes the type much more explicit, but it may also prevent you from using existing functions without first wrapping and unwrapping them. For this exercise, we'll use the first version as a means of demonstrating that approach.

> **NOTE:** you may not always need to produce a value to follow this pattern. A great example is the `HttpHandler` from Suave and Giraffe: `type HttpHandler = HttpContext -> Async<HttpContext option>`. This type mixes several concepts together but is ultimately similar to what we are building here, only without a value produced aside from the state.

`State` module

Many of our computation expression member implementations will be very similar. However, we were able to leverage existing module functions for the `OptionBuilder`, and having created a new type, we have no functions with which to work. Let's define those now.

> **NOTE:** creating a module of functions allows you to work without a computation expression, as well. This can be quite useful for debugging or for writing simple computations where the CE may not be quite as useful. You may also find it easier to think of each function separately rather than trying to define the behavior within a class member.

`Return`

In order to return a value, we'll need to wrap it in our `'s -> 'a * 's` function.

```
module State =
    // Explicit
    //let result x : State<'a, 's> = fun s -> x, s
    // Less explicit but works better with other, existing functions:
    let result x s = x, s
```

The `result` definition has been expressed in two ways. Interstingly, the first form with the `State<'a, 's>` does not actually force F# to use that type definition, so the latter may be preferred as it is shorter and gives you a good idea of the form of functions that could be used without modification. Were we to use the single-case union, you could enforce the type signature to be `State<'a,'s>` and not the type it aliases.

`Bind`

Next we want to look at how we compose two `State<'a, 's>` types. Here's the signature we have to work with

```
val bind : ('a -> State<'b, 's>) -> State<'a, 's> -> State<'b, 's>

// expanded:
val bind : ('a -> ('s -> 'b * 's)) -> ('s -> 'a * 's) -> ('s -> 'b * 's)
```

The expanded signature shows us that we need a state value to retrieve the `'a` value needed to pass to the function:

```
    let bind (f:'a -> State<'b, 's>) (m:State<'a, 's>) : State<'b, 's> =
        // return a function that takes the state
        fun s ->
            // Get the value and next state from the m parameter
            let a, s' = m s
            // Get the next state computation by passing a to the f 
parameter
            let m' = f a
            // Apply the next state to the next computation
            m' s'
```

In this case, we used the explicit lambda as it allows us to specify the types in the `bind` definition to help us guide our implementation.

We now have enough to begin implementing our `StateBuilder`:

```
type StateBuilder() =
    member __.Return(value) : State<'a, 's> = State.result value
    member __.Bind(m:State<'a, 's>, f:'a -> State<'b, 's>) : State<'b, 's> 
= State.bind f m
    member __.ReturnFrom(m:State<'a, 's>) = m
```

> **NOTE:** the type declarations are not required, but they can prove helpful when verifying the types are what you want.

Running a `State` Computation

While the `maybe` and `choose` computations returned an `option` value, this one returns a computation. You'll generally find it helpful to create helper functions to run the computation. In the case of `State`, since we have a tuple in the result, it is helpful to create two:

```
module State =
    // ... previously defined functions

    /// Evaluates the computation, returning the result value.
    let eval (m:State<'a, 's>) (s:'s) = m s |> fst

    /// Executes the computation, returning the final state.
    let exec (m:State<'a, 's>) (s:'s) = m s |> snd
```

We'll also need a pair of functions to get and set the state:

```
    /// Returns the state as the value.
    let getState (s:'s) = s, s

    /// Ignores the state passed in favor of the provided state value.
    let setState (s:'s) = fun _ -> (), s
```

Next add some tests to verify the basics work. The following test pairs are almost identical, but they return a result in two different ways:

1. as a value without changing state
2. as a state update returning `()`

```
        test "returns value" {
            let c = state {
                let! (s : string) = State.getState
                return System.String(s.ToCharArray() |> Array.rev)
            }
            let actual = State.eval c "Hello"
            Expect.equal actual "olleH" "Expected \"olleH\" as the value."
        }

        test "returns without changing state" {
            let c = state {
                let! (s : string) = State.getState
                return System.String(s.ToCharArray() |> Array.rev)
            }
            let actual = State.exec c "Hello"
            Expect.equal actual "Hello" "Expected \"Hello\" as the state."
        }

        test "returns unit" {
            let c = state {
                let! (s : string) = State.getState
                let s' = System.String(s.ToCharArray() |> Array.rev)
                do! State.setState s'
            }
            let actual = State.eval c "Hello"
            Expect.equal actual () "Expected return value of unit."
        }

        test "returns changed state" {
            let c = state {
                let! (s : string) = State.getState
                let s' = System.String(s.ToCharArray() |> Array.rev)
                do! State.setState s'
            }
            let actual = State.exec c "Hello"
            Expect.equal actual "olleH" "Expected state of \"elloH\"."
        }
```

Run `dotnet test`, and your tests should pass.

Handle `if ... then` without an `else`

```
        test "state supports if ... then with no else" {
            let c : State<unit, string> = state {
                if true then
                    printfn "Hello"
            }
            let actual = State.eval c ""
            Expect.equal actual () "Expected the value to be ()."
        }
```

The compiler should complain that the `StateBuilder` is missing the `Zero` method:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/StateBuilder.fs(100,17): error FS0708: This control
construct may only be used if the computation expression builder defines a
'Zero' method
```

Let's add it:

```
    member __.Zero() = State.result ()
```

With this in place, we can run this test successfully with `dotnet test`. Running with F# Interactive, however, reveals an issue:

```
>              let c : State<unit, string> = state {
-                  if true then
-                      printfn "Hello"
-              };;
Hello
val c : State<unit,string>
```

Why is `Hello` printed above just by creating the `State` value? Shouldn't that only happen when we run the computation? We need to implement `Delay` to delay execution until we are ready. Here's an implementation like we had before:

```
    member __.Delay(f) = f
    member __.Run(f) = f()
```

This satisfies the compiler (if you include `Run`) but does not resolve the issue, as `Hello` is still printed. What about the previous attempt of calling the `f` that was passed immediately?

```
    member __.Delay(f) = f()
```

Unfortunately, this also satisfies the compiler but fails to actually delay the computation. We need something a bit stronger:

```
    member __.Delay(f) = State.bind f (State.result ())
```

This at last prevents the computation from running before we are ready:

```
>              let c : State<unit, string> = state {
-                  if true then
-                      printfn "Hello"
-              };;
val c : State<unit,string>
```

Our tests also continue to pass when running `dotnet test`.

Returning Multiple Values

Add the following test and observe the compiler error:

```
        test "state supports returning multiple values" {
            let c : State<string, string> = state {
                return "one"
                return "two"
            }
            let actual = State.eval c ""
            Expect.equal actual "onetwo" "Expected all returns to be
concatenated."
        }
```

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/StateBuilder.fs(193,17): error FS0708: This control
construct may only be used if the computation expression builder defines a
'Combine' method
```

Returning multiple values requires `Combine`, so let's implement it ... but how? Here, we once again land in difficult territory. This is a reasonable implementation:

```
    member __.Combine(m1:State<unit, 's>, m2:State<'a, 's>) =
        // Carry state from m1 through to m2 while ignoring the value from
m1.
        State.bind (fun () -> m2) m1
```

Here's another:

```
    member inline __.Combine(m1:State<'a, 's>, m2:State<'a, 's>) =
        fun s ->
            let v1, s1 = m1 s
            let v2, s2 = m2 s
            v1 + v2, s1 + s2
```

For our test above, the second option works, whereas the first option fails, as it requires that the first `return` returns a `()`.

> **Observation:** you can implement **both** member definitions side-by-side within the same builder without a problem. Since CE builders are just classes, you can implement overload methods for different types of value and state as appropriate, which can give you great freedom to cover a multitude of types.

Review

We've implemented another builder with the following members:

- `Return`
- `ReturnFrom`
- `Bind`
- `Zero`
- `Delay`
- `Combine`

We did not implement `Run` because it doesn't help us in this case. However, we ***did*** implement a module of functions to define the functionality we used in the `StateBuilder` and that help us run the computation produced by the `state` expression, including:

- `eval`
- `exec`
- `getState`
- `setState`

We observed an even clearer example of how builder implementations may require member overloads in order to correctly cover different types of expressions.

Lastly, we have an example of how to resolve a common F# design question, "Where do I put this common state parameter?" If you have a set of functions that work together, and you are trying to determine whether to put the state instance as the first parameter -- so you can partially apply it -- or as the last -- so you can pipe it -- to the several, related functions, you may just want a CE that allows you to define the computation and then feed the state at the end.

In the next exercise, we will take returning multiple values to the next level.