Sequences

This exercise will take us on a slightly different path of building computation expressions where the the goal is to produce and consume sequences. We've looked at multiple `return` statements and `Combine` in previous exercises, but we have not yet covered anything like what you find with `seq { }` expressions.

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

```
module Sequences

open Expecto

type Stack<'a> =
    | Empty
    | Cons of top:'a * rest:Stack<'a>

module Stack =
    /// Pushes a new value on top of the stack
    let push v s = Cons(v, s)

    /// Pops the top value off the stack,
    /// returning both the value and remaining stack.
    /// Throws an error if there are no remaining values.
    let pop s =
        match s with
        | Cons(v, c) -> v, c
        | _ -> failwith "Nothing to pop!"

    /// Converts the Stack<'a> to an 'a list.
    let toList s =
        let rec loop s cont =
            match s with
            | Cons(head, tail) ->
                loop tail (fun rest -> cont(head::rest))
            | Empty -> cont []
        loop s id

    /// Pushes a value onto a new stack.
    let lift v = push v Empty

type StackBuilder() = class end

let stack = StackBuilder()

[<Tests>]
let tests =
    testList "sequences" [
```

```
        test "Stack.toList generates a matching list" {
            let actual = Cons(1, Cons(2, Cons(3, Empty))) |> Stack.toList
            Expect.equal actual [1;2;3] "Expected list containing [1;2;3]"
        }
    ]
```

You'll likely notice immediately that this is essentially the exact same thing as `'a list`. We could easily have used that type for our implementation. However, the goal is to show how to build computation expressions, not use existing `list` comprehensions.

Returning a Value

As noted above, we know how to return multiple values. However, we will do so using the `yield` syntax this time:

```
[<Tests>]
let tests =
    testList "sequences" [
        test "stack can return one item" {
            let actual = stack { yield 1 }
            Expect.equal (Stack.toList actual) [1] "Expected a stack
containing 1"
        }
    ]
```

Building the project fails with:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/Sequences.fs(42,34): error FS0708: This control
construct may only beused if the computation expression builder defines a
'Yield' method
```

We could have used `return` as we did previously, but this time we are using `yield`; therefore, we need to implement the `Yield` member. What's the difference?

```
type StackBuilder() =
    member __.Yield(value) = Stack.lift value
```

Basically nothing. Compare `Stack.lift` with `State.result`, and you'll find they are essentially the same, except for the differences in data structure. `Yield` is typically used when you ***want*** multiple return values as it better matches with the syntax found in most iterator or generator implementations in other languages. You are free to use either.

Once you implement the member, the build should succeed and tests should pass when running `dotnet test`.

> **Observation:** `Yield` = `Return`, *most* of the time.

We might as well cover `YieldFrom`, which is the same as `ReturnFrom` covered earlier. Again, these essentially enable syntactic sugar of a different form, so you don't need to implement both unless you want them to enable different behavior in some way.

```
// in StackBuilder
    member __.YieldFrom(m:Stack<'a>) = m

// in tests
        test "stack can yield an empty stack" {
            let actual = stack { yield! Empty }
            Expect.equal actual Empty "Expected an empty stack"
        }
```

Running `dotnet test` should succeed.

Returning Multiple Values

What about returning multiple values?

```
        test "stack can return multiple items" {
            let actual = stack {
                yield 1
                yield 2
                yield 3
            }
            Expect.equal (Stack.toList actual) [1;2;3] "Actual should
match expected"
        }
```

Building the project fails with:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/Sequences.fs(43,17): error FS0708: This control
construct may only be used if the computation expression builder defines a
'Combine' method
```

Our old friend `Combine` wants to join in the fun. This shouldn't come as a surprise given what we learned about the relationship between `Return` and `Yield`. The only outstanding issue is how to combine these two?

You have two options:

1. Load the first stack first, though they are older
2. Load the second stack first, as they are newer

This greatly depends on how you want your computation expression to work. `push` will always push a new value onto the stack, meaning the last in is on top. However, our test above defines the first listed as the top. Once again, we find that the implementor defines the CE implementation strategy; there's no prescribed way to implement this member. We'll follow the test above, where the CE represents creating a stack from top to bottom rather than a sequence of pushing items into the stack:

```
// in module Stack =
    let append s1 s2 =
        let rec loop s1 cont =
            match s1 with
            | Cons(top, rem) ->
                loop rem (fun rest -> cont(Cons(top, rest)))
            | Empty -> cont s2
        loop s1 id

// in StackBuilder =
    member __.Combine(s1:Stack<'a>, s2:Stack<'a>) = Stack.append s1 s2
```

Once again, module functions come in handy. The `append` function is essentially the same as the `List.append`, and that's exactly what we want to do here: replace the `Empty` at the end of the first stack, `s1`, with `s2`.

Running `dotnet test` once again fails:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/Sequences.fs(58,17): error FS0708: This control
construct may only beused if the computation expression builder defines a
'Delay' method
```

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

is enough to satisfy the compiler and allow our tests to run. However, if you add `printfn` statements between the `yield` statements, you'll see they are eagerly evaluated as before. You'll see the same with the following implementation, with `Run`:

```
    member __.Combine(s1, s2) = Stack.append s1 (s2())
    member __.Delay(f) = f
    member __.Run(f) = f()
```

Delayed Execution?

In order to correctly delay execution, you'll need something similar to what we did for the `StateBuilder`. However, we don't want `Bind` in this case as we have an iterable collection.

```
        test "stack can iterate and yield" {
            let expected = stack { yield 1; yield 2; yield 3 }
            let actual = stack {
                for x in expected do
                    yield x
            }
            Expect.equal actual expected "Expected iterating and yielding
to return the same result"
        }
```

A `for` loop is a much better way of expressing iteration for a collection than `let!`, and iteration is the execution of a collection, after all.

The compiler helpfully informs us:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/Sequences.fs(83,17): error FS0708: This control
construct may only beused if the computation expression builder defines a
'For' method
```

The documentation specifies the following types for `For`:

```
seq<'T> * ('T -> M<'U>) -> M<'U>
// or
seq<'T> * ('T -> M<'U>) -> seq<M<'U>>
```

The first would be useful in our previous CEs as means of aggregating a sequence. However, the list is incomplete. Another, valid signature is:

```
M<'T> -> ('T -> M<'U>) -> M<'U>
```

This should be familar from our implementations of `Bind` in previous CEs. The same signature is represented by the common collection function `collect`, as defined in F# Collection Types: "Applies the given function to each element of the collection, concatenates all the results, and returns the combined list."

> **Observation:** `For` = `Bind`, for some use cases.

```
// in module Stack =
    /// Applies the function f to each element in the stack and
concatenates the results.
    let collect (f:'a -> Stack<'b>) (m:Stack<'a>) =
        let rec loop s cont =
            match s with
```

```
            | Cons(top, rem) ->
                loop rem (fun rest -> cont(append (f top) rest))
            | Empty -> cont Empty
        loop m id

// in StackBuilder =
    member __.For(m, f) = Stack.collect f m
```

Running tests with `dotnet test` should show everything passing.

What about the delayed execution? Now that we have a `Bind` in the form of a `For` and `Return` in the form of `Yield`, will the strategy used in `StateBuilder` now allow our builder to delay execution? Change `Delay`:

```
    member __.Delay(f) = Stack.collect f (Stack.lift ())
```

Run the code:

```
val stack : StackBuilder

>             let actual = stack {
-                 yield 1
-                 printfn "printed 1"
-                 yield 2
-                 printfn "printed 2"
-                 yield 3
-             };;
printed 1
printed 2
val actual : Stack<int> = Cons (1,Cons (2,Cons (3,Empty)))
```

The `printfn` statements still succeed. Just like `'a list`, our `Stack` is eagerly evaluated to produce a result. Attempting to block this with iteration fails. We would need to go to greater lengths and create a true `LazyList` type using either `Lazy<'T>` or a `unit -> 'T` as the internal type. For the purposes of `StackBuilder`, however, we should adopt the simplest option possible, which was the original:

```
    member __.Delay(f:unit -> Stack<'a>) = f()
```

While it effectively does nothing, it's also the simplest implementation and satisfies the CE requirements.

Review

We've now implemented three types of builders:

1. containers
2. computations

3. sequences

and looked at many of the members required to create these builders. In this exercise, we implemented:

- `Yield`
- `YieldFrom`
- `For`

and saw how they were similar to other members we implemented previously except for the syntax provided. We also briefly covered variations for the `For` member.

Only a few members remain for the core Computation Expression syntax, and we'll review those in the next exercise.