ChoiceBuilder

In this next exercise, we'll look at how we can chain `'a option` computations together by returning multiple values. This is similar to allow the computation to make a choice, e.g.

```
let actual : string option =
    choice {
        return! None
        return "Use this instead."
    }
```

Our implemetation will allow the computation to *choose* a path based on the values returned. Following our example from the previous exercise, successfully writing the file and returning `None` could then be a way of signaling to continue, whereas a value of `Some "some error message"` would indicate the computation should halt.

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

```
module Choose

open Expecto

type ChoiceBuilder() = class end
```

Interlude

Before we get to far, observe that due to CE's use of classes, we can re-use functionality by means of inheritance. I don't advise doing this as a regular practice, but it can be useful in some cases. We'll use it here only to illustrate that you *can*.

```
open Options

type ChoiceBuilder() =
    inherit OptionBuilder()

let choose = ChoiceBuilder()

[<Tests>]
let tests =
    testList "choices" [
        test "ChoiceBuilder returns value" {
            let expected = 1
            let actual = choose { return expected }
            Expect.equal actual (Some expected) "Expected Some 1"
        }

        test "ChoiceBuilder can bind option values" {
            let actual = choose {
                let! w = opt1
                let! x = opt2
                let! y = opt3
                let! z = opt4
                let result = sum4 w x y z
                printfn "Result: %d" result // print if a result was
computed.
                return result
            }
            Expect.equal actual nested "Actual should sum to the same
value as nested."
        }

        test "ChoiceBuilder instance can be used directly" {
            let actual =
                choose.Bind(opt1, fun w ->
                    choose.Bind(opt2, fun x ->
                        choose.Bind(opt3, fun y ->
                            choose.Bind(opt4, fun z ->
                                let result = sum4 w x y z
                                printfn "Result: %d" result
                                choose.Return(result)
                            )
                        )
                    )
                )
            Expect.equal actual composed "Actual should sum to the same
value as nested."
        }

        test "ChoiceBuilder can exit without returning a value" {
            let dirExists path =
                let fileInfo = System.IO.FileInfo(path)
                let fileName = fileInfo.Name
                let pathDir = fileInfo.Directory.FullName.TrimEnd('~')
                if System.IO.Directory.Exists(pathDir) then
                    Some (System.IO.Path.Combine(pathDir, fileName))
                else None

            let choosePath = Some "~/test.txt"

            let actual =
                choose {
                    let! path = choosePath
                    let! fullPath = dirExists path
                    System.IO.File.WriteAllText(fullPath, "Test
succeeded")
```

```
                }

            Expect.equal actual None "Actual should be Some unit"
        }

        test "ChoiceBuilder supports if then without an else" {
            let choosePath = Some "~/test.txt"

            let actual =
                choose {
                    let! path = choosePath
                    let pathDir = System.IO.Path.GetDirectoryName(path)
                    if not(System.IO.Directory.Exists(pathDir)) then
                        return "Select a valid path."
                }

            Expect.equal actual (Some "Select a valid path.") "Actual
should return Some(\"Select a valid path.\")"
        }

        test "ChoiceBuilder allows for early escape with return!" {
            let actual =
                choose {
                    if true then
                        return! None
                    else
                        let! w = opt1
                        let! x = opt2
                        let! y = opt3
                        let! z = opt4
                        let result = sum4 w x y z
                        printfn "Result: %d" result // print if a result
was computed.
                        return result
                }
            Expect.equal actual None "Should return None immediately"
        }
    ]
```

Running `dotnet test` with this implementation should succeed and show that we are now ready to extend our builder. This is *one* way to extend a builder if you don't want to accidentally break code using `maybe` elsewhere.

Combining Computations

```
        test "choose returns first value if it is Some" {
            let actual = choose {
                return! Some 1
                printfn "returning first value?"
                return! Some 2
            }
            Expect.equal actual (Some 1) "Expected the first value to be
returned."
        }
```

Run `dotnet test`. The project will once again fail to compile with a helpful error message:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ChoiceBuilder.fs(19,17): error FS0708: This control
construct may only be used if the computation expression builder defines a
'Combine' method
```

The standard signature for `Choose` from the specification would appear to indicate the correct implementation for our `ChoiceBuilder` is:

```
member Combine : unit option * (unit -> 'a option) -> 'a option
```

This would work just fine if we wanted to continue in the case of a `Some ()`, as we initially started with in our `OptionBuilder`. However, we want to support a choice, where a `None` leads to the second path, not a `Some ()`. With that in mind, our signature needs to be slightly different:

```
member Combine : 'a option * (unit -> 'a option) -> 'a option
```

This looks wrong at first blush, but it's a necessary evil to support a `None` return value:

```
    member __.Combine(m:'a option, f:unit -> 'a option) =
        printfn "choose.Combine(%A, %A)" m f
        match m with
        | Some _ -> m
        | None -> f()
```

Try to compile, and you'll find that the compiler is still not happy:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ChoiceBuilder.fs(125,17): error FS0708: This control
construct may only be used if the computation expression builder defines a
'Delay' method
```

Delaying Execution

When running a computation expression, results are eagerly evaluated. `Delay` allows the computation to pause. `Delay` is generally defined as:

```
member Delay : (unit -> 'a option) -> 'a option
```

or for our computation,

```
    member __.Delay(f:unit -> 'a option) =
        printfn "choose.Delay(%A)" f
        f()
```

`Delay` essentially wraps your expression in a function that takes a `unit` parameter in order to force something to request execution. There is no requirement on the return type, and we'll see why this is useful shortly.

The compiler will now inform you that the signature for `Combine` is incorrect:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ChoiceBuilder.fs(32,17): error FS0001: This expression
was expected to have type
    'unit -> int option'
but here has type
    'int option'
```

```
    member __.Combine(m1:'a option, m2:'a option) =
        printfn "choose.Combine(%A, %A)" m1 m2
        match m1 with
        | Some _ -> m1
        | None -> m2
```

`Combine` now takes two `'a option` types. Because of our implementation, the type arguments must match. This isn't required for the signature of `Combine`, as should be clear from our initial attempt.

The compiler is finally happy. Run `dotnet test` to see your test pass, as well as the print out of what methods were called:

```
choose.Delay(<fun:actual@63>)
maybe.ReturnFrom(Some 1)
choose.Delay(<fun:actual@64-1>)
returning first value?
maybe.ReturnFrom(Some 2)
choose.Combine(Some 1, Some 2)
val actual : int option = Some 1
```

The output betrays a few issues. Here's the expanded form:

```
        test "expanding choose for the second attempt runs the same way" {
            let actual =
                choose.Delay(fun () ->
                    choose.Combine(
                        choose.ReturnFrom(Some 1),
                        choose.Delay(fun () ->
                            printfn "returning first value?"
                            choose.ReturnFrom(Some 2)
                        )
                    )
                )
            Expect.equal actual (Some 1) "Expected the first value to be
returned."
        }
```

Notice that in order to return the result, the computation *will* evaluate both paths because `Delay` evaluates immediately, even though the second path need not be evaluated in this case. Aside from the order of the `printfn` output from the method calls, `"returning first value?"` is also printed.

It's possible to have the computation expression delay execution until you are ready by changing the implementation of the `Delay` member:

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

`Delay` takes a function and returns it without doing anything with it. However, the compiler is unhappy:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ChoiceBuilder.fs(361,17): error FS0001: This expression
was expected to have type
    'int option'
but here has type
    'unit -> int option'
```

`Combine` is now incorrect. Fortunately, we can just restore it to what we had before:

```
    member __.Combine(m:'a option, f:unit -> 'a option) =
        printfn "choose.Combine(%A, %A)" m f
        match m with
        | Some _ -> m
        | None -> f()
```

Running `dotnet test` fails to compile and produces the following result:

```
/Users/ryan/Code/computation-expressions-
workshop/solutions/ChoiceBuilder.fs(69,26): error FS0001: The type '(unit
-> int option)' does not support the 'equality' constraint because it is a
function type
```

Our `actual` value is now a function that must be called. We can do several things with this now:

- Let the user call this after returning the value
- Add a `runMaybe` function that accepts a `unit -> 'a option` and runs it
- Wrap this as a `Lazy<_>` value to avoid repeat executions
- Implement the `Run` member for computation expressions

Because the implementation is valid, the compiler won't tell you to implement `Run`. However, you'll likely want this to avoid having to explicitly evaluate the result at the end of each computation. From the specification, `Run`, if implemented, is wrapped around the computation expression and can be used to do almost anything. For our purposes, we'll have it just execute the delayed expression:

```
    member __.Run(f:unit -> 'a option) =
        printfn "choose.Run(%A)" f
        f()
```

The previous test we added to compare the expanded form will now break, so just comment it out or remove it and run `dotnet test`. Your tests should pass, and you should no longer see `"returning first value?"` printed in the output.

```
choose.Delay(<fun:actual@427-14>)
choose.Run(<fun:actual@427-14>)
maybe.ReturnFrom(Some 1)
choose.Delay(<fun:actual@428-15>)
choose.Combine(Some 1, <fun:actual@428-15>)
val actual : int option = Some 1
```

Here's the expanded form now that we have delayed the computation:

```
        test "expanding choose for the fourth attempt runs the same way" {
            let actual =
                choose.Run(
                    choose.Delay(fun () ->
                        choose.Combine(
                            choose.ReturnFrom(Some 1),
                            choose.Delay(fun () ->
                                printfn "returning first value?"
                                choose.ReturnFrom(Some 2)
                            )
                        )
                    )
                )
            Expect.equal actual (Some 1) "Expected the first value to be
returned."
        }
```

Notice that `Run` wraps the entire expression now. Also, the second argument to `Combine` is now the delayed expression rather than the evaluated result of the delayed expression because `Delay` *does not* immediately invoke its provided function. `Combine` is now responsible for invoking the provided function, just as `Run` is responsible for invoking the top-level delayed function.

For good measure, let's add tests to verify:

1. the second path *will* be evaluated if the first is `None`
2. more than two returns are supported

```
            test "choose returns second value if first is None" {
                let actual = choose {
                    return! None
                    printfn "returning second value?"
                    return! Some 2
                }
                Expect.equal actual (Some 2) "Expected the second value to be
returned."
            }

            test "choose returns the last value if all previous are None" {
                let actual = choose {
                    return! None
                    return! None
                    return! None
                    return! None
                    return! None
                    return! None
                    return! Some 7
                }
                Expect.equal actual (Some 7) "Expected the seventh value to be
returned."
            }
```

Running `dotnet test` should compile and pass all tests.

Chaining Computations

Can we now go back to our file writing computation and chain additional work to the end of a successful, indicated by a `None`, file write?

```
            test "ChoiceBuilder can chain a computation onto another returning
None, where None indicates success" {
```

```
            let dirExists path =
                let fileInfo = System.IO.FileInfo(path)
                let fileName = fileInfo.Name
                let pathDir = fileInfo.Directory.FullName.TrimEnd('~')
                if System.IO.Directory.Exists(pathDir) then
                    Some (System.IO.Path.Combine(pathDir, fileName))
                else None

            let choosePath = Some "~/test.txt"

            let writeFile =
                choose {
                    let! path = choosePath
                    let! fullPath = dirExists path
                    System.IO.File.WriteAllText(fullPath, "Test succeeded")
                }
            let actual =
                choose {
                    return! writeFile
                    return "Successfully wrote file"
                }

            Expect.equal actual (Some "Successfully wrote file") "Actual should indicate success"
        }
```

Running `dotnet test` should successfully run all tests.

Review

In this exercise, we implemented the following members, many in several, different ways:

- `Combine`
- `Delay`
- `Run`

You should be getting the idea that computation expressions *do not* adhere to strict or rigid type signatures but can be implemented to solve different problems depending on your use case.

We also observed that, since computation expression builders are just .NET classes, you can also use inheritance and the full range of OO programming.

So far, we've only looked at "wrapper" or "container" types, types that wrap a value in some way. Another example would be the `Result` type. In the next section, we'll look at writing computation expressions around function types.