Query Expressions

In this section we will look at Query Expressions. Query Expressions were introduced in F# 3.0, alongside Type Providers.

From the docs,

> Query expressions enable you to query a data source and put the data in a desired form. Query expressions provide support for LINQ in F#.

In earlier versions of F#, LINQ queries had to be expressed in the form of quotations, e.g.

```
query <@ for x in source do yield x @>
```

With query expressions, the same could be expressed in a form more familiar to .NET developers of other languages, e.g.

```
query {
    for x in source do
    where x < 1
    select x
}
```

Compare with C#:

```
from x in source
where x < 1
select x
```

Fortunately, query expressions may be created for any data source, just like computation expressions and LINQ providers. Also like these, you are allowed to add as many or as few operators as you want to support or enable within your query.

In this exercise, we'll look at the `QueryBuilder` provided by `FSharp.Core`, implement a few more extensions to support `'a option` results, and explore the range of `CustomOperation`s by implementing several members of the `rxquery` expression for the Reactive Extensions found in FSharp.Control.Reactive.

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

```
module Queries

open Expecto

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

QueryBuilder

You can find the implementation of the F# `QueryBuilder` in the Microsoft/visualfsharp repository on GitHub. Open the link and make the following observations:

1. `QueryBuilder` works against a `QuerySource` type
2. `For` and `Yield`, as well as `Zero` and `YieldFrom`, form the basic syntax for iterating and returning values
3. `Quote` is implemented, meaning the results will be transformed into F# quotations
4. There are a bunch of methods implemented that look like some of the syntax we would expect to see, e.g. `Select`, `Where`, etc.
5. Toward the bottom of the `QueryBuilder` definition, at line 203, you'll see several methods relating to `Run`, as well as the actual `Run` method
6. Below the `QueryBuilder` you'll see a `Query` module that processes the `Quotation.Expr<_>` results

You may be wondering whether all these methods like `Select`, `Where`, etc. are all detected by name to support query expressions. Or you may know that these methods have attributes specified in a Query.fsi signature file. Open this link, and you'll see the following:

```
        /// <summary>A query operator that determines whether the selected
elements contains a specified element.
        /// </summary>
        [<CustomOperation("contains")>]
        member Contains : source:QuerySource<'T,'Q> * key:'T -> bool

        /// <summary>A query operator that returns the number of selected
elements.
        /// </summary>
        [<CustomOperation("count")>]
        member Count : source:QuerySource<'T,'Q> -> int

        /// <summary>A query operator that selects the last element of
those selected so far.
        /// </summary>
        [<CustomOperation("last")>]
        member Last : source:QuerySource<'T,'Q> -> 'T

        // ...

        /// <summary>A query operator that projects each of the elements
```

```
selected so far.
        /// </summary>
        [<CustomOperation("select",AllowIntoPattern=true)>]
        member Select : source:QuerySource<'T,'Q> *
[<ProjectionParameter>] projection:('T -> 'Result) ->
QuerySource<'Result,'Q>

        /// <summary>A query operator that selects those elements based on
a specified predicate.
        /// </summary>

[<CustomOperation("where",MaintainsVariableSpace=true,AllowIntoPattern=tru
e)>]
        member Where : source:QuerySource<'T,'Q> * [<ProjectionParameter>]
predicate:('T -> bool) -> QuerySource<'T,'Q>

        // ...

        /// <summary>A query operator that correlates two sets of selected
values based on matching keys.
        /// Normal usage is 'join y in elements2 on (key1 = key2)'.
        /// </summary>
        [<CustomOperation("join",IsLikeJoin=true,JoinConditionWord="on")>]
        member Join : outerSource:QuerySource<'Outer,'Q> *
innerSource:QuerySource<'Inner,'Q> * outerKeySelector:('Outer -> 'Key) *
innerKeySelector:('Inner -> 'Key) * resultSelector:('Outer -> 'Inner ->
'Result) -> QuerySource<'Result,'Q>

        /// <summary>A query operator that correlates two sets of selected
values based on matching keys and groups the results.
        /// Normal usage is 'groupJoin y in elements2 on (key1 = key2)
into group'.
        /// </summary>

[<CustomOperation("groupJoin",IsLikeGroupJoin=true,JoinConditionWord="on")
>]
        member GroupJoin : outerSource:QuerySource<'Outer,'Q> *
innerSource:QuerySource<'Inner,'Q> * outerKeySelector:('Outer -> 'Key) *
innerKeySelector:('Inner -> 'Key) * resultSelector:('Outer -> seq<'Inner>
-> 'Result) -> QuerySource<'Result,'Q>

        // ...

        /// <summary>
        /// When used in queries, this operator corresponds to the LINQ
Zip operator.
        /// </summary>
        [<CustomOperation("zip",IsLikeZip=true)>]
        member Zip : firstSource:QuerySource<'T1> *
secondSource:QuerySource<'T2> * resultSelector:('T1 -> 'T2 -> 'Result) ->
QuerySource<'Result>

        // ...
```

Query expressions are enabled by the `CustomOperationAttribute`. Observe that this query expression uses many variations of the available parameters, though `MaintainsVariableSpaceUsingBind` is missing. Nevertheless, you can learn a lot about the behavior enforced by these parameters based on the methods in this builder.

The use of `Quote` means the builder ***should*** also define `Run`. While not required, some form of quotation processor should be provided. In the case of `QueryBuilder`, several `Run` options are provided:

```
        /// <summary>
        /// A method used to support the F# query syntax.  Indicates that
the query should be passed as a quotation to the Run method.
        /// </summary>
        member Quote  : Quotations.Expr<'T> -> Quotations.Expr<'T>

        /// <summary>
        /// A method used to support the F# query syntax.  Runs the given
quotation as a query using LINQ IQueryable rules.
        /// </summary>
        member Run : Quotations.Expr<QuerySource<'T,IQueryable>> ->
IQueryable<'T>
        member internal RunQueryAsQueryable :
Quotations.Expr<QuerySource<'T,IQueryable>> -> IQueryable<'T>
        member internal RunQueryAsEnumerable :
Quotations.Expr<QuerySource<'T,IEnumerable>> -> seq<'T>
        member internal RunQueryAsValue : Quotations.Expr<'T> -> 'T
```

These `Run` methods allow the `query` expression to support several data sources at once, including those exposed by Entity Framework and simple `seq<'a>` values.

> **Observation:** the `QueryBuilder` works with `Enumerable` internally, then exposes its contents as `Quotations.Expr<_>`, which are processed by the `Run` methods. While you may not necessarily want to use the `Quote` feature, this highlights an important concept of differntiating the internal type of the computation from the result type.

Extending `QueryBuilder` to support `'a option`

Another observation you may have made is how closely the `QueryBuilder` has stuck to the common LINQ expressions, including the `exactlyOneOrDefault`, `headOrDefault`, etc. However, it's difficult to make sense of a `default` for many F# types. The more common approach for F# is to represent something that may not exist as an `'a option`. The following test will pass, but what is represented by the `actual : TestRec` value?

```
type TestRec = { Value : int }

[<Tests>]
let tests =
    testList "queries" [
        test "query supports F# types with headOrDefault" {
```

```
            let actual : TestRec =
                query {
                    for x in Seq.empty<TestRec> do
                    headOrDefault
                }
            Expect.equal actual (Unchecked.defaultof<TestRec>) "Expected
default value of TestRec"
        }
    ]
```

The following test better expresses what we would like from the query:

```
        test "query supports F# types with headOrNone" {
            let actual =
                query {
                    for x in Seq.empty<TestRec> do
                    headOrNone
                }
            Expect.equal actual None "Expected None"
        }
```

The compiler will complain that this operator is not available on the `QueryBuilder`:

```
This is not a known query operator. Query operators are identifiers such
as 'select', 'where', 'sortBy', 'thenBy', 'groupBy', 'groupValBy', 'join',
'groupJoin', 'sumBy' and 'averageBy', defined using corresponding methods
on the 'QueryBuilder' type.
```

Let's fix this with an extension:

```
open System
open System.Linq
open System.Reactive.Linq
open System.Reactive.Concurrency
open Microsoft.FSharp.Linq
open Expecto

type Microsoft.FSharp.Linq.QueryBuilder with

    [<CustomOperation("headOrNone")>]
    member __.HeadOrNone(source:QuerySource<'T,'Q>) =
        Seq.tryHead source.Source
```

Running `dotnet test` should now succeed. This extension didn't need any of the additonal `CustomOperation` parameters as it works directly on the input data source.

We can do something similar to support `exactlyOneOrNone`:

```
// ... extension:
type Microsoft.FSharp.Linq.QueryBuilder with

    [<CustomOperation("exactlyOneOrNone")>]
    member __.ExactlyOneOrNone(source:QuerySource<'T,'Q>) =
        if Seq.length source.Source = 1 then
            Enumerable.Single(source.Source) |> Some
        else None

// ... tests:

        test "query exactlyOneOrNone returns the single value for a seq
with one element" {
            let source = seq { yield { Value = 1 } }
            let actual =
                query {
                    for x in source do
                    exactlyOneOrNone
                }
            Expect.equal actual (Seq.tryHead source) "Expected { Value = 1
}"
        }

        test "query exactlyOneOrNone returns None for an empty seq" {
            let source = Seq.empty<TestRec>
            let actual =
                query {
                    for x in source do
                    exactlyOneOrNone
                }
            Expect.equal actual None "Expected None"
        }

        test "query exactlyOneOrNone returns None for a seq with more than
one element" {
            let source = seq { yield { Value = 1 }; yield { Value = 2 } }
            let actual =
                query {
                    for x in source do
                    exactlyOneOrNone
                }
            Expect.equal actual None "Expected None"
        }
```

`dotnet test` should run successfully for all tests.

`CustomOperation`s in `rxquery`

The FSharp.Control.Reactive project provides modules and builders that make working with the Reactive Extensions for .NET better fit F# idioms. In the next exercise, we'll re-implement a simple Rx query builder to better understand the `CustomOperationAttribute` and what each of its parameters can enable.

At a minimum, we'll need to implement `For`, `Yield`, and `Zero`, as we saw when looking at the `QueryBuilder`. These are required for iterating over the input data source, yielding results, and allowing for an empty return value:

```
type RxQueryBuilder() =
    member __.For (s:IObservable<_>, body : _ -> IObservable<_>) =
s.SelectMany(body)
    member __.Yield (value) = Observable.Return(value)
    member __.Zero () = Observable.Empty(Scheduler.CurrentThread :>
IScheduler)

let rxquery = RxQueryBuilder()
```

> **NOTE:** this query expression uses Rx's `Observable` type for its implementation and follows the LINQ syntax for determining which methods to use. For example, `SelectMany` is the common implementation member for combining two `from ...` expressions in a LINQ query, so it's used here. This means that the first observable must be exhausted before the second would be used. Another valid implementation might be to use the `Observable.Merge` member. The difference is that where `SelectMany` waits for the first observable to complete, `Merge` combines the combined observables as they produce values. This may be a better implementation for your use case.

Select

While `yield` will allow us to return a value, the more common idiom in query expressions is `select`:

```
        test "rxquery can select values from a source" {
            let expected = [|1..10|]
            let actual = Array.zeroCreate<int> 10
            let source = Observable.Range(1, 10)
            use disp =
                rxquery {
                    for x in source do
                    select x
                }
                |> Observable.subscribe (fun i -> actual.[i - 1] <- i)
            Expect.equal actual expected "Expected observable to populate
empty array with selected values"
        }
```

We can support this syntax with a `CustomOperation`. We'll use the Rx `Select` extension method for `IObservable<_>`. This method takes a `selector` function, so we'll need to pass that in somehow:

```
type RxQueryBuilder with
    [<CustomOperation("select")>]
    member __.Select(s:IObservable<_>, selector: _ -> _) =
        s.Select(selector)
```

This *almost* works. However, the compiler complains that, "The value or constructor 'x' is not defined." We can make a simple change to the test to get this to compile and run:

```
        test "rxquery can select values from a source" {
            let expected = [|1..10|]
            let actual = Array.zeroCreate<int> 10
            let source = Observable.Range(1, 10)
            use disp =
                rxquery {
                    for x in source do
                    select (fun x -> x) // or the `id` function
                }
                |> Observable.subscribe (fun i -> actual.[i - 1] <- i)
            Expect.equal actual expected "Expected observable to populate
empty array with selected values"
        }
```

That's not quite what we wanted, but it will work. However, we can do better. Remember the `ProjectionParameter` attribute from the `Extensions` exercise? **The `ProjectionParameter` attribute allows us to pick up the variable or value from earlier in the expression and use it.**

```
type RxQueryBuilder with
    [<CustomOperation("select")>]
    member __.Select(s:IObservable<_>, [<ProjectionParameter>] selector: _
-> _) =
        s.Select(selector)
```

After this change, the test `rxquery` is okay, but it still ignores the `x` in the `for` and complains in the `Expect` expression that the return type is an `obj`. We can now remove the lambda and simply reference the `x` value as before. Running `dotnet test` should succeed.

`Head` and `exactlyOne`

While on the subject of returning values, we should implement operators that return a single value from the expression, such as `head` and `exactlyOne`. These correlate to the LINQ extension methods `First` and `Single`, respectively.

```
type RxQueryBuilder with
    [<CustomOperation("head")>]
    member __.Head (s:IObservable<_>) = s.FirstAsync()
```

```
    [<CustomOperation("exactlyOne")>]
    member __.ExactlyOne (s:IObservable<_>) = s.SingleAsync()
```

The relevant Rx extension methods do not require any parameters, so these are quite easy to implement, assuming you know what methods to call.

We can add tests for these, as well:

```
        test "rxquery can return the first value from a source" {
            let mutable actual : int = -1
            let source = Observable.Range(1, 10)
            use disp =
                rxquery {
                    for x in source do
                    head
                }
                |> Observable.subscribe (fun i -> actual <- i)
            Expect.equal actual 1 "Expected head to return 1"
        }

        test "rxquery can return the single value from a source of one
element" {
            let mutable actual : int = -1
            let source = Observable.Return(1)
            use disp =
                rxquery {
                    for x in source do
                    exactlyOne
                }
                |> Observable.subscribe (fun i -> actual <- i)
            Expect.equal actual 1 "Expected exactlyOne to return 1"
        }
```

You should now be able to implement operators such as `count`, `max`, `min`, and more using the same approach.

Where

Another common operation is to filter data. We can expose this with `where`:

```
        test "rxquery can filter an observable with where" {
            let expected = [|1..5|]
            let actual = Array.zeroCreate<int> 5
            let source = Observable.Range(1, 10)
            use disp =
                rxquery {
                    for x in source do
                    where (x <= 5)
                    select x
                }
```

```
                |> Observable.subscribe (fun i -> actual.[i - 1] <- i)
            Expect.equal actual expected "Expected exactlyOne to return 1"
        }
```

The related Rx extension method is also `Where` which takes a `predicate` function. We know how to handle projecting the variable from our implementation of `Select`:

```
type RxQueryBuilder with
    [<CustomOperation("where")>]
    member __.Where(s:IObservable<_>, [<ProjectionParameter>] predicate: _
-> bool) =
        s.Where(predicate)
```

Unfortunately, the compiler doesn't think this is enough.

```
/Users/ryan/Code/ceworkshop/solutions/Queries.fs(195,21): error FS0001:
This expression was expected to have type    'int'    but here has type
'unit'
```

In order for `where` to work successfully, we need to tell inform the compiler that this operation will not change the return type. **The `MaintainsVariableSpace` parameter in the `CustomOperation` attribute will inform the compiler the operation is a chained operation with the same type as the input.**

```
type RxQueryBuilder with
    [<CustomOperation("where", MaintainsVariableSpace=true)>]
    member __.Where(s:IObservable<_>, [<ProjectionParameter>] predicate: _
-> bool) =
        s.Where(predicate)
```

The compiler is once again happy, and `dotnet test` should run your tests successfully. Additional operators that use this approach include `Skip`, `Take`, `Sort` (though you may want to avoid this with `IObsevable`s), etc.

GroupBy

The next operator we should investigate is `groupBy`. Unlike `where`, `groupBy` transforms the result type. Unlike `select`, `groupBy` doesn't return a result set. Instead, its result is assigned to a new value:

```
        test "rxquery can group an observable" {
            let expected = [|"a";"b"|]
            let actual = ResizeArray<string>()
            let source =
                Observable.Generate(1,
                    (fun x -> x < 10),
                    (fun x -> x + 1),
```

```
                    (fun x ->
                        if x < 2 then "a", x
                        else "b", x))
            use disp =
                rxquery {
                    for (k, _) in source do
                    groupBy k into g
                    select g.Key
                }
                |> Observable.subscribe actual.Add
            Expect.equal (actual.ToArray()) expected "Expected where to
filter input"
        }
```

Rx again provides the extension method required to implement this behavior, and we once again allow use of the value assigned in the `for` with the `ProjectionParameter`:

```
type RxQueryBuilder with
    [<CustomOperation("groupBy")>]
    member __.GroupBy (s:IObservable<_>,[<ProjectionParameter>]
keySelector : _ -> _) =
        s.GroupBy(Func<_,_>(keySelector))
```

The compiler helpfully informs us, `The operator 'groupBy' does not accept the use of 'into'`. This matches another parameter of the `CustomOperation` attribute. **The `AllowIntoPattern` parameter allows an operator to transform the source input into a new type by specifying a new value with `into`.**

Join

Joins are used to find the intersection of two data sets based on a `predicate` that links the two data sets to gether. The results of joins may be grouped using `into` like `groupBy`. However, joins are special enough to require their own parameter. **The `IsLikeJoin` parameter in the `CustomOperation` attribute provides all the context necessary for the compiler to understand how to process a join. You also need to specify the `JoinConditionWord`, e.g. "on".**

> **NOTE:** the `JoinConditionWord` is important to remember. At some point in the past, this appears to have been defaulted to `"on"`; however, the current `FSharp.Core` defaults this parameter to `""`. You will only see a type error at present.

```
    [<CustomOperation("join", IsLikeJoin=true, JoinConditionWord="on")>]
    member __.Join (s1:IObservable<_>, s2:IObservable<_>,
                    [<ProjectionParameter>] leftDurationSelector : _ -> _,
                    [<ProjectionParameter>] rightDurationSelector : _ ->
_,
                    [<ProjectionParameter>] resultSelector : _ -> _) =
        s1.Join(s2,
                Func<_,_>(leftDurationSelector),
```

```
                Func<_,_>(rightDurationSelector),
                Func<_,_,_>(resultSelector))
```

The `Join` method takes two observables, a key selector function for each observable, and finally the projected result selector, or the rest of the query expression. Whereas the types of the observables may be different, the types of the key selectors should match, as they will be equated.

```
        test "rxquery can join two observables" {
            let actual = ResizeArray<int * int>()
            let source1 = Observable.Range(1, 5)
            let source2 = Observable.Range(3, 5)
            use disp =
                rxquery {
                    for x in source1 do
                    join y in source2 on
((Observable.Timer(TimeSpan.FromMilliseconds 0.)) =
(Observable.Timer(TimeSpan.FromMilliseconds 0.)))
                    select (x,y)
                }
                |> Observable.subscribe actual.Add
            Expect.isGreaterThan (Array.length (actual.ToArray())) 1
"Expected join to produce a sequence longer than 1 value"
        }
```

This test should compile and pass with `dotnet test`.

Zip

Joins are not the only way to combine values. You can also zip two sources together or even split them apart and reassemble them, i.e. forkjoin. Rx helpfully provides methods for both of these tasks. `CustomOperation` also provides an `IsLikeZip` parameter to indicate this behavior should be used. **The `IsLikeZip` parameter, seen earlier in `Extensions`, informs the compiler that each element of two sources should be paired together.**

Unlike `Join`, setting the `IsLikeZip=true` is sufficient to trigger the correct behavior:

```
    [<CustomOperation("zip", IsLikeZip=true)>]
    member __.Zip (s1:IObservable<_>, s2:IObservable<_>,
                   [<ProjectionParameter>] resultSelector : _ -> _) =
        s1.Zip(s2, Func<_,_,_>(resultSelector))
```

This successfully allows the following test to pass with `dotnet test`:

```
        test "rxquery can zip two observables" {
            let expected = [|1,3;2,4;3,5;4,6;5,7|]
            let actual = ResizeArray<int * int>()
```

```
            let source1 = Observable.Range(1, 5)
            let source2 = Observable.Range(3, 5)
            use disp =
                rxquery {
                    for x in source1 do
                    zip y in source2
                    select (x,y)
                }
                |> Observable.subscribe actual.Add
            Expect.equal (actual.ToArray()) expected "Expected join to
produce [|1,3;2,4;3,5;4,6;5,7|]"
        }
```

Understanding Restrictions

There are a handful of rules about `CustomOperation`s that are not well defined except by error messages. You can find those restrictions in the `FSComp.txt` file in the visualfsharp repository (or by trying and failing to build).

One such restriction states, "The implementations of custom operations may not be overloaded." Another states, "A custom operation may not be used in conjunction with 'use', 'try/with', 'try/finally', 'if/then/else' or 'match' operators within this computation expression."

Review

This section completes our exploration into the features provided by computation and query expressions. In this section, we reviewed the following:

- `QueryBuilder`
- Extending Query Expressions
- `CustomOperation` parameter uses

You have now seen all the tools available for creating useful, reusable abstractions to simplify your programs. However, we can take this even farther by leveraging `CustomOperation`s to embed domain specific languages within F# and provide a even more expressive abstractions for writing simple programs to solve complex problems.