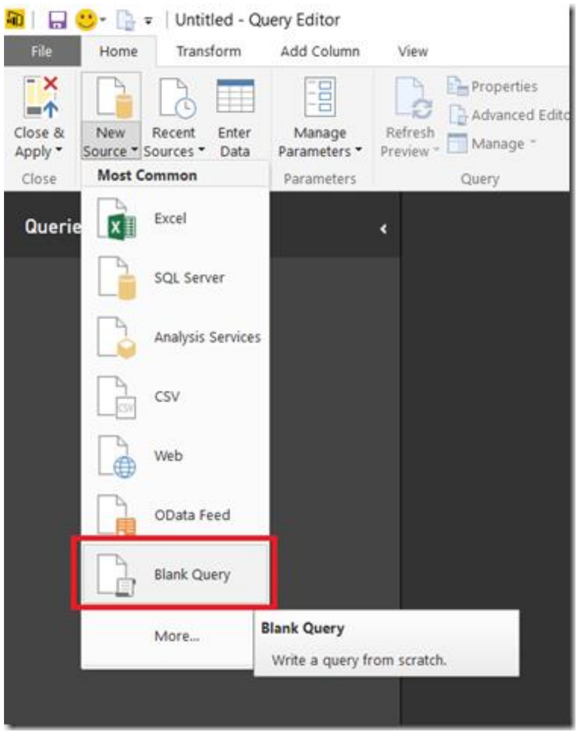
Understanding Let Expressions In M For Power BI And Power Query

When you start writing M code for loading data in Power Query or Power BI, one of the first things you'll do is open up the Advanced Editor for a query you've already built using the UI.

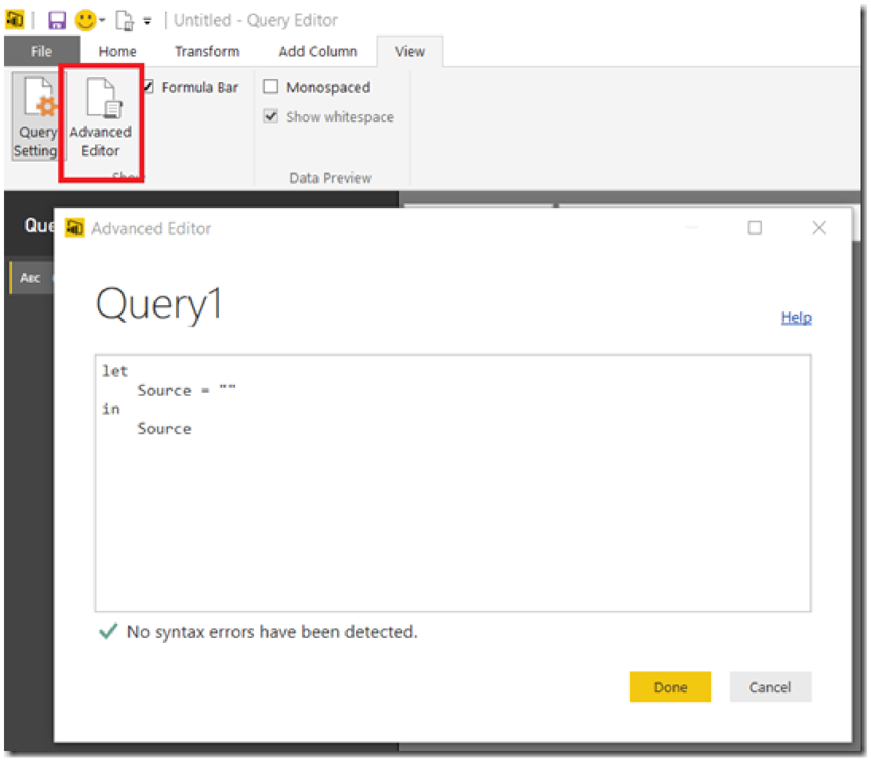
The first step to doing so is to understand how *let* expressions work in M.

Each query that you create in Power BI Desktop or Power Query is a single expression that, when evaluated, returns a single value – and that single value is usually, but not always, a table that then gets loaded into the data model.

To illustrate this, open up Power BI Desktop (the workflow is almost the same in Power Query), click the Edit Queries button to open the Query Editor window and then click New Source/Blank Query to create a new query.



Next, go to the View tab and click on the Advanced Editor button to open the Advanced Editor dialog:

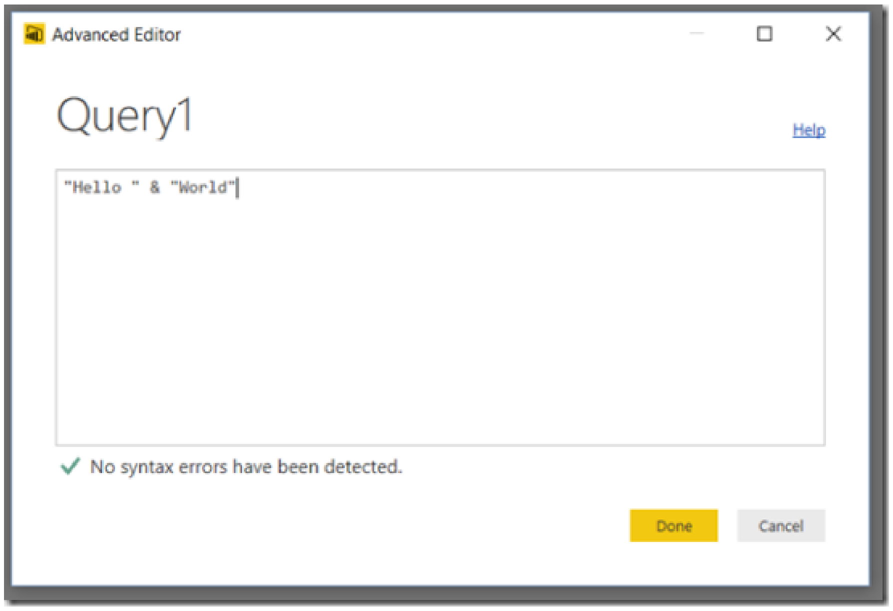


```
let
    Source = ""
in
    Source
```

No syntax errors have been detected.

You'll notice that this doesn't actually create a blank query at all, because there is some code visible in the Advanced Editor when you open it. Delete everything there and replace it with the following M expression:

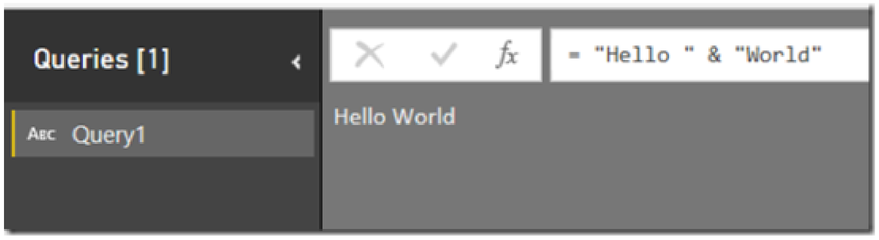
1"Hello " & "World"



```
"Hello " & "World"
```

No syntax errors have been detected.

Hit the Done button and the expression will be evaluated, and you'll see that the query returns the text value "Hello World":



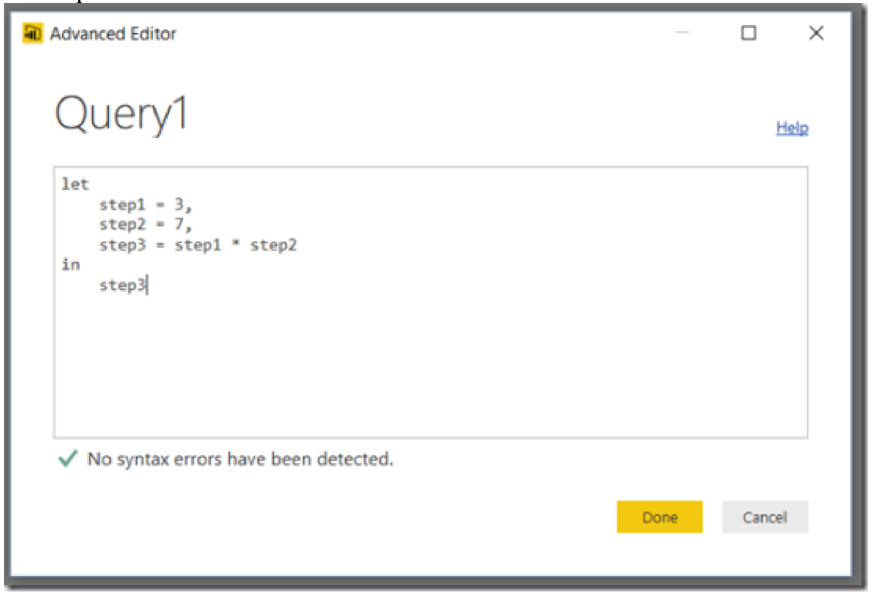
```
= "Hello " & "World"
```

```
Hello World
```

Notice how the ABC icon next to the name of the Query – Query1 – indicates that the query returns a text value. Congratulations, you have written the infamous "Hello World" program in M!

You might now be wondering how the scary chunk of code you see in the Advanced Editor window for your real-world query could possibly be a single expression – but in fact it is. This is where *let* expressions come in: they allow you to break a single expression down into multiple parts. Open up the Advanced Editor again and enter the following expression:

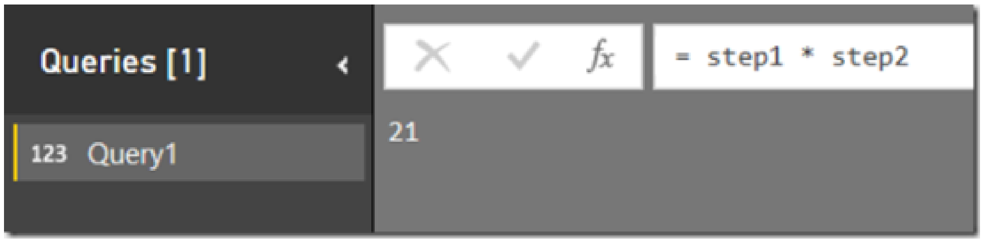
```
let
  step1 = 3,
  step2 = 7,
  step3 = step1 * step2
in
  step3
```



```
let
    step1 = 3,
    step2 = 7,
    step3 = step1 * step2
in
    step3
```

✓ No syntax errors have been detected.

Without knowing anything about M it's not hard to guess that this bit of code returns the numeric value 21 (notice again that the 123 icon next to the name of the query indicates the data type of the value the query returns):

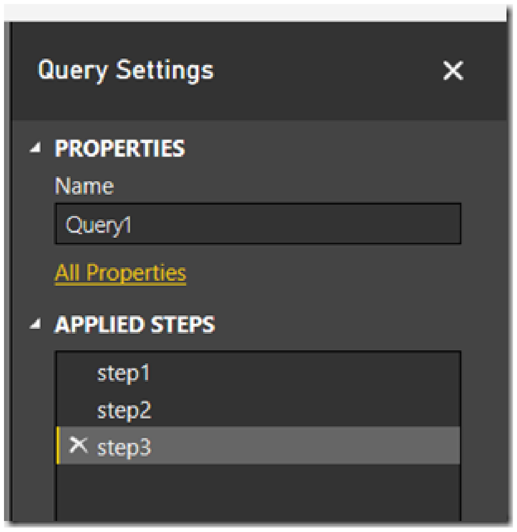


```
= step1 * step2
```

```
21
```

In the M language a *let* expression consists of two sections. After the *let* comes a list of variables, each of which has a name and an expression associated with it. In the previous example there are three variables:

step1, step2 and step3. Variables can refer to other variables; here, step3 refers to both step1 and step2. Variables can be used to store values of any type: numbers, text, dates, or even more complex types like records, lists or tables; here, all three variables return numbers. The Query Editor is usually clever enough to display these variables as steps in your query and so displays then in the Applied Steps pane on the right-hand side of the screen:

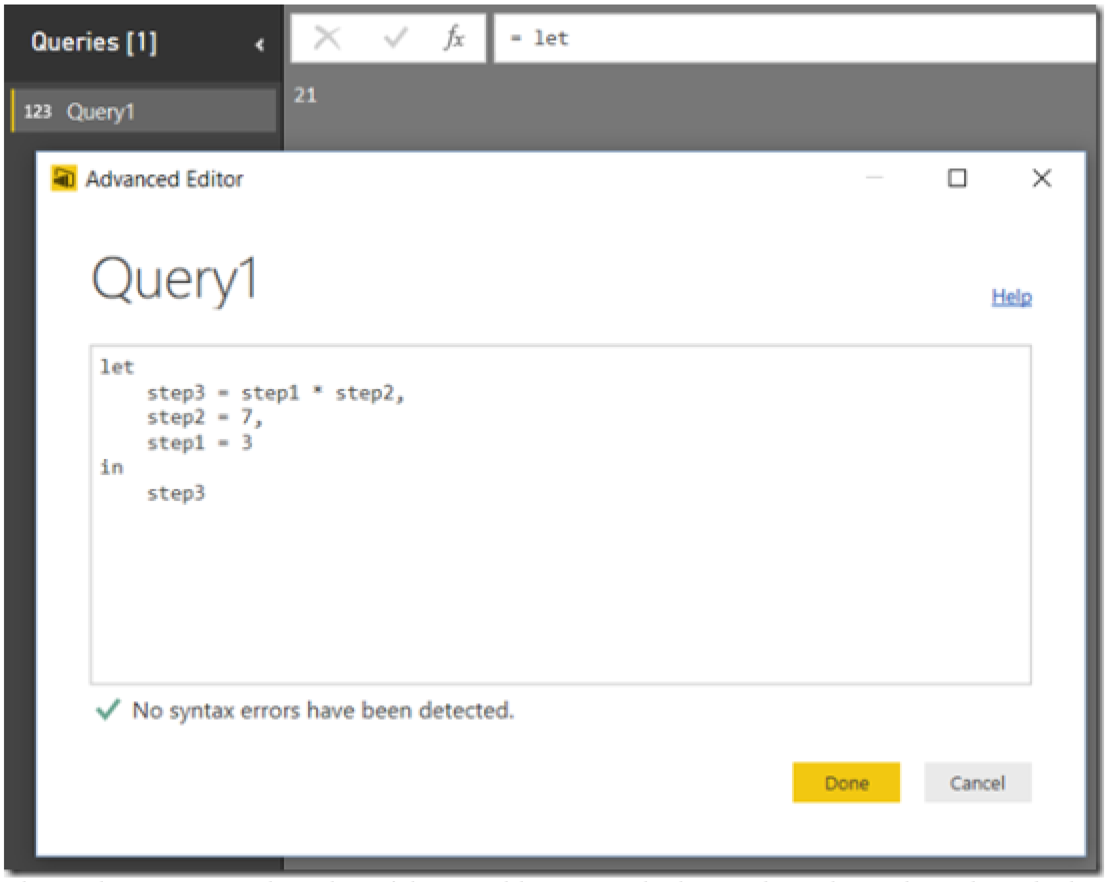


The value that the *let* expression returns is given in the *in* clause. In this example the *in* clause returns the value of the variable step3, which is 21.

It's important to understand that the *in* clause can reference any or none of the variables in the variable list. It's also important to understand that, while the variable list might look like procedural code it isn't, it's just a list of variables that can be in any order. The UI will always generate code where each variable/step builds on the value returned by the previous variable/step but when you're writing your own code the variables can be in whatever order that suits you.

For example, the following query also returns the value 21:

```
let
  step3 = step1 * step2,
  step2 = 7,
  step1 = 3
in
  step3
```

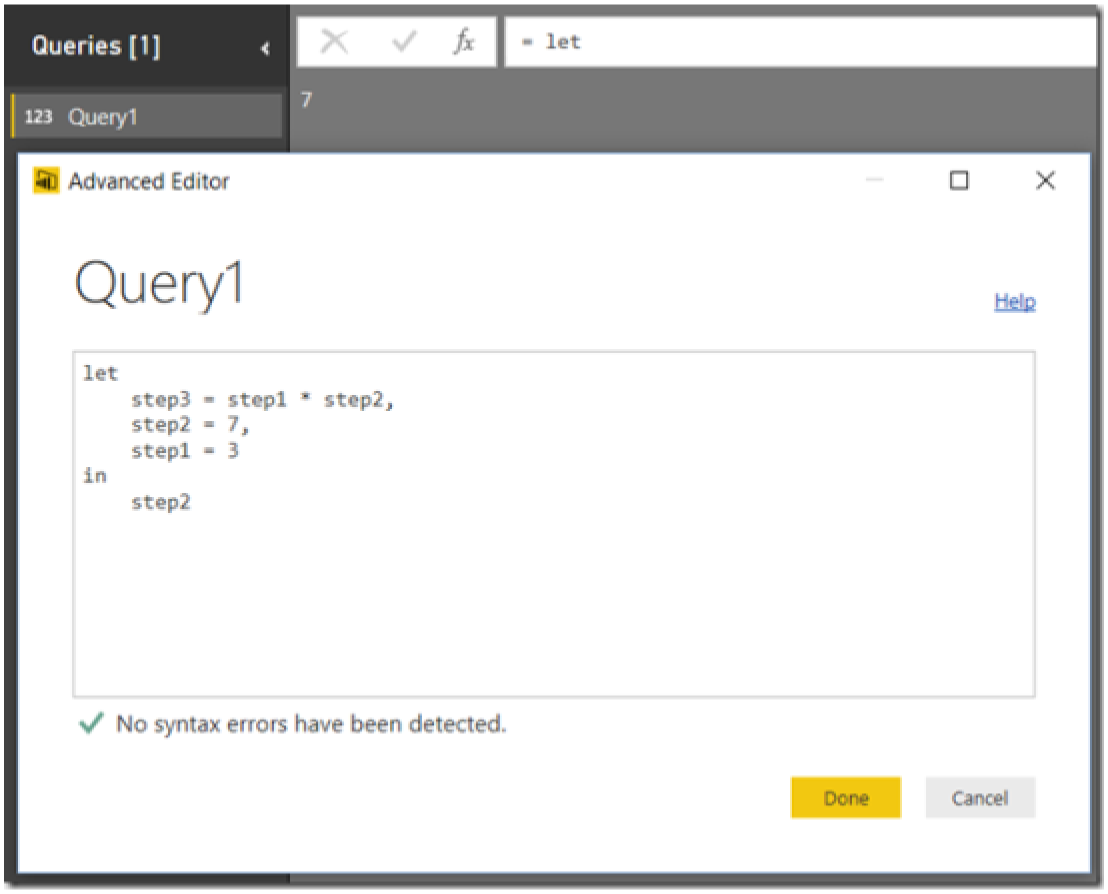


```
let
    step3 = step1 * step2,
    step2 = 7,
    step1 = 3
in
    step3
```

No syntax errors have been detected.

The *in* clause returns the value of the variable step3, which in order to be evaluated needs the variables step2 and step1 to be evaluated; the order of the variables in the list is irrelevant (although it does mean the Applied Steps no longer displays each variable name). What is important is the chain of dependencies that can be followed back from the *in* clause.
To give another example, the following query returns the numeric value 7:

```
let
  step3 = step1 * step2,
  step2 = 7,
  step1 = 3
in
  step2
```



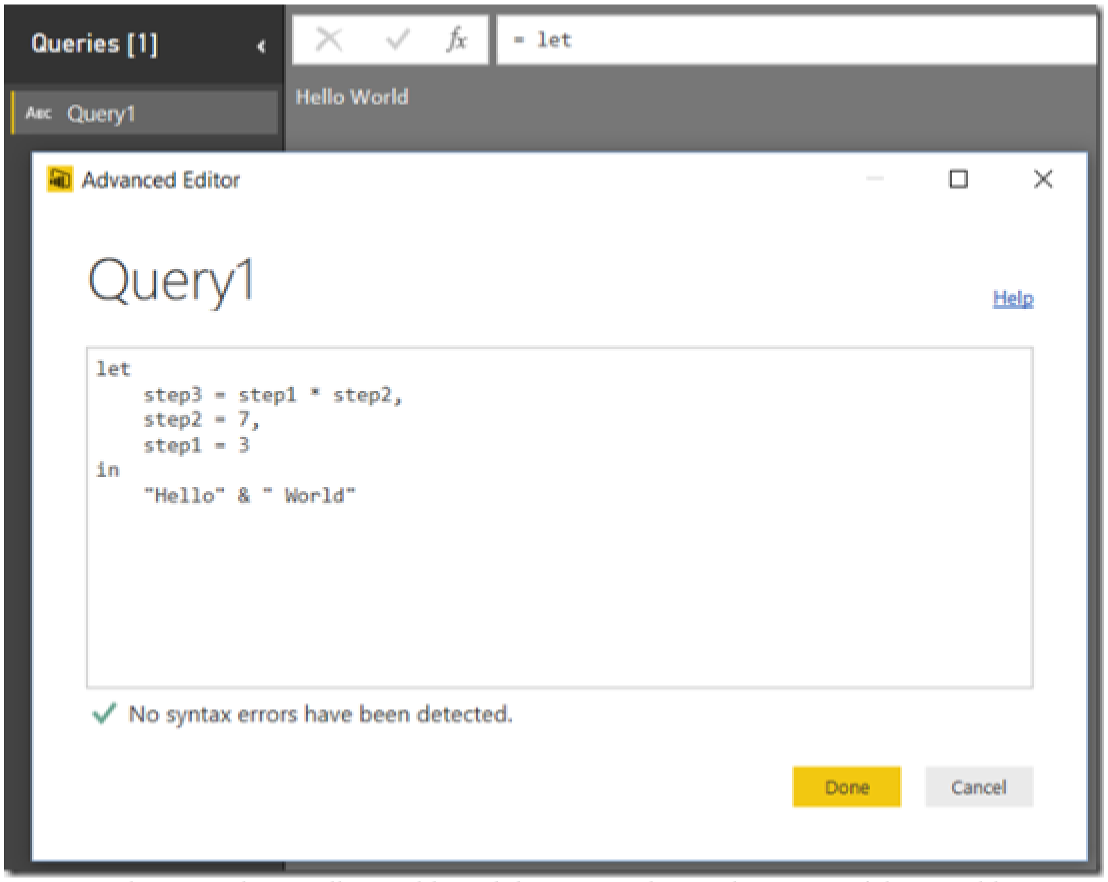
Query1

```
let
    step3 = step1 * step2,
    step2 = 7,
    step1 = 3
in
    step2
```

No syntax errors have been detected.

In this case, step2 is the only variable that needs to be evaluated for the entire *let* expression to return its value. Similarly, the query

```
let
  step3 = step1 * step2,
  step2 = 7,
  step1 = 3
in
  "Hello" & " World"
```



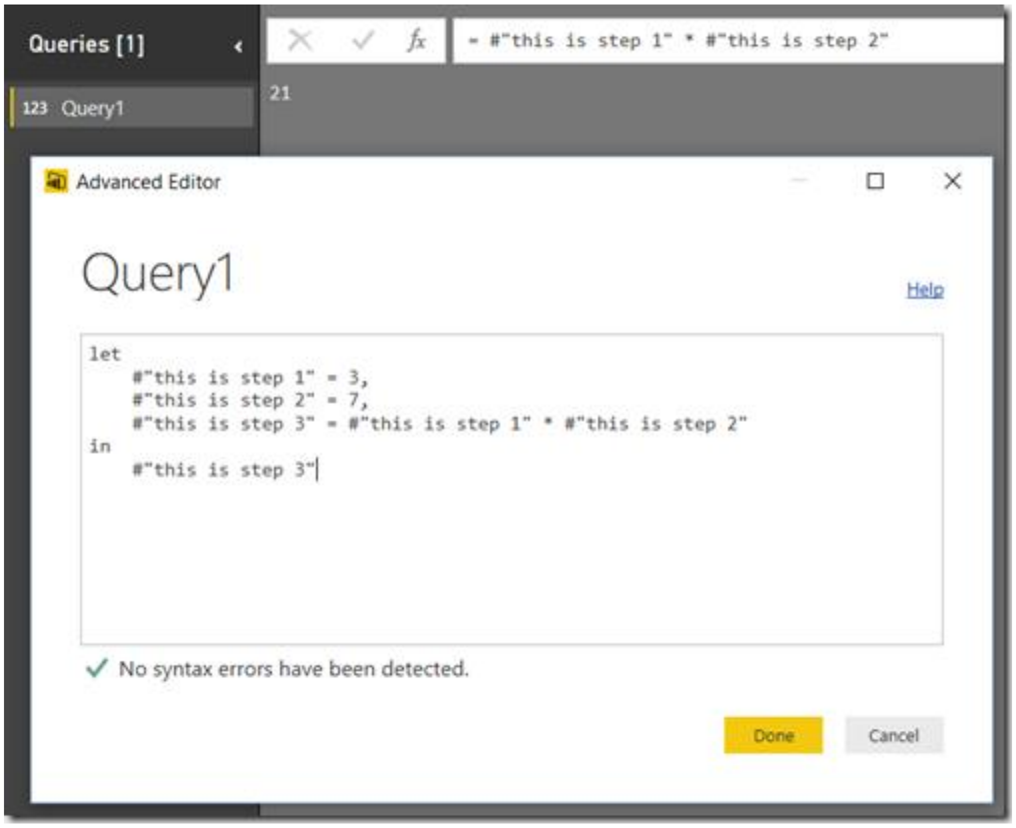
```
let
    step3 = step1 * step2,
    step2 = 7,
    step1 = 3
in
    "Hello" & " World"
```

No syntax errors have been detected.

...returns the text value "Hello World" and doesn't need to evaluate any of the variables step1, step2 or step3 to do this.

The last thing to point out is that if the names of the variables contain spaces, then those names need to be enclosed in double quotes and have a hash # symbol in front. For example here's a query that returns the value 21 where all the variables have names that contain spaces:

```
let
  #"this is step 1" = 3,
  #"this is step 2" = 7,
  #"this is step 3" = #"this is step 1" * #"this is step 2"
in
  #"this is step 3"
```



```
= #"this is step 1" * #"this is step 2"
```

21

Query1

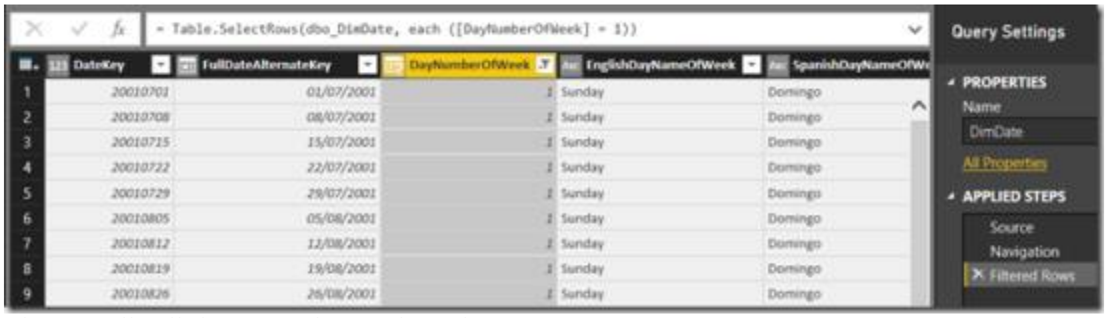
```
let
    #"this is step 1" = 3,
    #"this is step 2" = 7,
    #"this is step 3" = #"this is step 1" * #"this is step 2"
in
    #"this is step 3"
```

✓ No syntax errors have been detected.

How does all this translate to queries generated by the UI? Here's the M code for a query generated by the UI that connects to SQL Server and gets filtered data from the DimDate table in the Adventure Works DW database:

```
let
  Source = Sql.Database("localhost", "adventure works dw"),
  dbo_DimDate = Source{[Schema="dbo",Item="DimDate"]}[Data],
  #"Filtered Rows" = Table.SelectRows(dbo_DimDate,
                          each ([DayNumberOfWeek] = 1))
in
  #"Filtered Rows"
```

Regardless of what the query actually does, you can now see that there are three variables declared here, #"Filtered Rows", dbo_DimDate and Source, and the query returns the value of the #"Filtered Rows" variable. You can also see that in order to evaluate the #"Filtered Rows" variable the dbo_DimDate variable must be evaluated, and in order to evaluate the dbo_DimDate variable the Source variable must be evaluated. The Source variable connects to the Adventure Works DW database in SQL Server; dbo_DimDate gets the data from the DimDate table in that database, and #"Filtered Rows" takes the table returned by dbo_DimDate and filters it so that you only get the rows here the DayNumberOfWeek column contains the value 1.



```
= Table.SelectRows(dbo_DimDate, each ([DayNumberOfWeek] = 1))
```

	DateKey	FullDateAlternateKey	DayNumberOfWeek	EnglishDayNameOfWeek	SpanishDayNameOfWe
1	20010701	01/07/2001	1	Sunday	Domingo
2	20010708	08/07/2001	1	Sunday	Domingo
3	20010715	15/07/2001	1	Sunday	Domingo
4	20010722	22/07/2001	1	Sunday	Domingo
5	20010729	29/07/2001	1	Sunday	Domingo
6	20010805	05/08/2001	1	Sunday	Domingo
7	20010812	12/08/2001	1	Sunday	Domingo
8	20010819	19/08/2001	1	Sunday	Domingo
9	20010826	26/08/2001	1	Sunday	Domingo

Query Settings

PROPERTIES

Name: DimDate

All Properties

APPLIED STEPS

- Source
- Navigation
- Filtered Rows

That's really all there is to know about *let* expressions.

A Power Query M formula language query is composed of formula **expression** steps that create a mashup query. A formula expression can be evaluated (computed), yielding a value. The **let** expression encapsulates a set of values to be computed, assigned names, and then used in a subsequent expression that follows the **in** statement. For example, a let expression could contain a **Source** variable that equals the value of **Text.Proper()** and yields a text value in proper case.

Let expression

powerquery-mCopy

```
let
    Source = Text.Proper("hello world")
in
    Source
```

In the example above, Text.Proper("hello world") is evaluated to "Hello World".

The next sections describe value types in the language.

Primitive value

A **primitive** value is single-part value, such as a number, logical, text, or null. A null value can be used to indicate the absence of any data.

Type	Example value
Binary	00 00 00 02 // number of points (2)

Type	Example value
Date	5/23/2015
DateTime	5/23/2015 12:00:00 AM
DateTimeZone	5/23/2015 12:00:00 AM -08:00
Duration	15:35:00
Logical	true and false
Null	null
Number	0, 1, -1, 1.5, and 2.3e-5
Text	"abc"
Time	12:34:12 PM

Function value

A **Function** is a value which, when invoked with arguments, produces a new value. Functions are written by listing the function's **parameters** in parentheses, followed by the goes-to symbol =>, followed by the expression defining the function. For example, to create a function called "MyFunction" that has two parameters and performs a calculation on parameter1 and parameter2:

powerquery-mCopy

```
let
    MyFunction = (parameter1, parameter2) => (parameter1 + parameter2) / 2
in
    MyFunction

Calling the MyFunction() returns the result:

let
    Source = MyFunction(2, 4)
in
```

```
    Source
```

This code produces the value of 3.

Structured data values

The M language supports the following structured data values:

- List
- Record
- Table
- Additional structured data examples

Note

Structured data can contain any M value. To see a couple of examples, see Additional structured data examples.

List

A List is a zero-based ordered sequence of values enclosed in curly brace characters { }. The curly brace characters { } are also used to retrieve an item from a List by index position. See [List value](#_List_value).

Note

Power Query M supports an infinite list size, but if a list is written as a literal, the list has a fixed length. For example, {1, 2, 3} has a fixed length of 3.

The following are some **List** examples.

Value	Type
{123, true, "A"}	List containing a number, a logical, and text.
{1, 2, 3}	List of numbers

Value	Type
{ {1, 2, 3}, {4, 5, 6} }	List of List of numbers
{ [CustomerID = 1, Name = "Bob", Phone = "123-4567"], [CustomerID = 2, Name = "Jim", Phone = "987-6543"] }	List of Records
{123, true, "A"}{0}	Get the value of the first item in a List. This expression returns the value 123.
{ {1, 2, 3}, {4, 5, 6} }{0}{1}	Get the value of the second item from the first List element. This expression returns the value 2.

Record

A **Record** is a set of fields. A **field** is a name/value pair where the name is a text value that is unique within the field's record. The syntax for record values allows the names to be written without quotes, a form also referred to as **identifiers**. An identifier can take the following two forms:

- identifier_name such as OrderID.
- #"identifier name" such as #"Today's data is: ".

The following is a record containing fields named "OrderID", "CustomerID", "Item", and "Price" with values 1, 1, "Fishing rod", and 100.00. Square brace characters [ ] denote the beginning and end of a record expression, and are used to get a field value from a record. The follow examples show a record and how to get the Item field value.

Here's an example record:

powerquery-mCopy

```
let Source =
        [
              OrderID = 1,
              CustomerID = 1,
              Item = "Fishing rod",
              Price = 100.00
        ]
in Source
```

To get the value of an Item, you use square brackets as Source[Item]:

powerquery-mCopy

```
let Source =
    [
          OrderID = 1,
          CustomerID = 1,
          Item = "Fishing rod",
          Price = 100.00
    ]
in Source[Item] //equals "Fishing rod"
```

Table

A **Table** is a set of values organized into named columns and rows. The column type can be implicit or explicit. You can use #table to create a list of column names and list of rows. A **Table** of values is a List in a **List**. The curly brace characters { } are also used to retrieve a row from a **Table** by index position (see Example 3 – Get a row from a table by index position).

Example 1 - Create a table with implicit column types

powerquery-mCopy

```
let
  Source = #table(
    {"OrderID", "CustomerID", "Item", "Price"},
      {
          {1, 1, "Fishing rod", 100.00},
          {2, 1, "1 lb. worms", 5.00}
      })
in
    Source
```

Example 2 – Create a table with explicit column types

powerquery-mCopy

```
let
    Source = #table(
    type table [OrderID = number, CustomerID = number, Item = text, Price =
number],
        {
                {1, 1, "Fishing rod", 100.00},
             {2, 1, "1 lb. worms", 5.00}
        }
    )
in
    Source
```

Both of the examples above creates a table with the following shape:

OrderID	CustomerID	Item	Price
1	1	Fishing rod	100.00
2	1	1 lb. worms	5.00

Example 3 – Get a row from a table by index position

powerquery-mCopy

```
let
    Source = #table(
    type table [OrderID = number, CustomerID = number, Item = text, Price =
number],
        {
              {1, 1, "Fishing rod", 100.00},
              {2, 1, "1 lb. worms", 5.00}
         }
    )
in
    Source{1}
```

This expression returns the follow record:

OrderID	2
CustomerID	1
Item	1 lb. worms
Price	5

Additional structured data examples

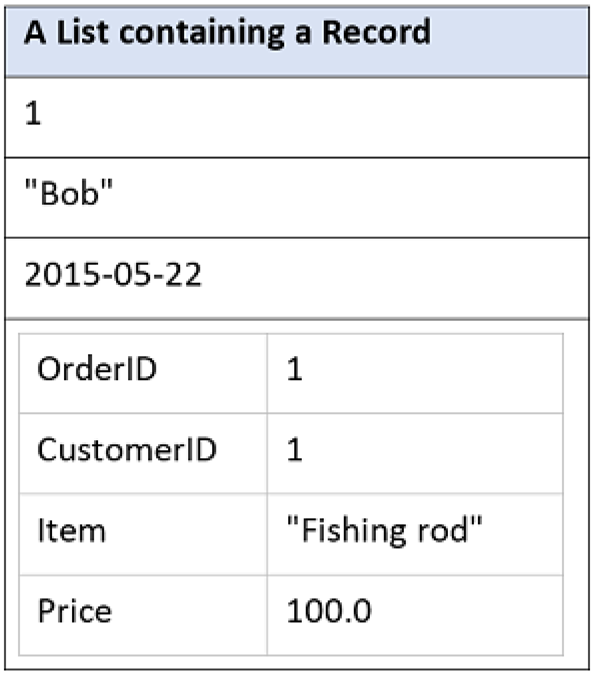
Structured data can contain any M value. Here are some examples:

Example 1 - List with [Primitive](#_Primitive_value_1) values, [Function](#_Function_value), and [Record](#_Record_value)

powerquery-mCopy

```
let
    Source =
{
   1,
   "Bob",
   DateTime.ToText(DateTime.LocalNow(), "yyyy-MM-dd"),
   [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0]
}
in
    Source
```

Evaluating this expression can be visualized as:



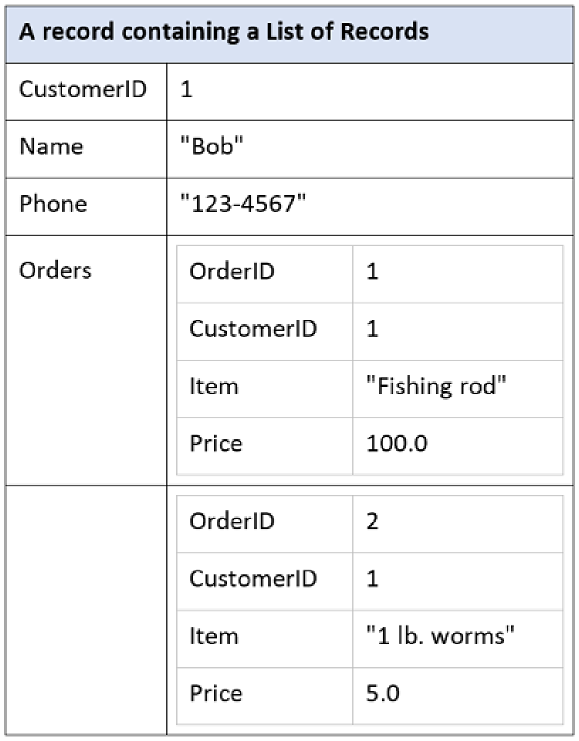
A List containing a Record	
1	
"Bob"	
2015-05-22	
OrderID	1
CustomerID	1
Item	"Fishing rod"
Price	100.0

Example 2 - Record containing Primitive values and nested Records

powerquery-mCopy

```
let
    Source = [CustomerID = 1, Name = "Bob", Phone = "123-4567", Orders =
        {
              [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price =
100.0],
            [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0]
        }]
in
    Source
```

Evaluating this expression can be visualized as:



A record containing a List of Records			
CustomerID	1		
Name	"Bob"		
Phone	"123-4567"		
Orders		OrderID	1
		CustomerID	1
		Item	"Fishing rod"
		Price	100.0
		OrderID	2
		CustomerID	1
		Item	"1 lb. worms"
		Price	5.0

Note: Although many values can be written literally as an expression, a value is not an expression. For example, the expression 1 evaluates to the value 1; the expression 1+1 evaluates to the value 2. This distinction is subtle, but important. Expressions are recipes for evaluation; values are the results of evaluation.

If expression

The **if** expression selects between two expressions based on a logical condition. For example:

powerquery-mCopy

```
if 2 > 1 then
    2 + 2
else
    1 + 1
```

The first expression (2 + 2) is selected if the logical expression (2 > 1) is true, and the second expression (1 + 1) is selected if it is false. The selected expression (in this case 2 + 2) is evaluated and becomes the result of the **if** expression (4).

References

1. Understanding Let Expressions In M For Power BI And Power Query:
    a. https://blog.crossjoin.co.uk/2016/05/22/understanding-let-expressions-in-m-for-power-bi-and-power-query/
2. Expressions, values, and let expression:
    a. https://docs.microsoft.com/en-us/powerquery-m/expressions-values-and-let-expression
3. space