

C# Expression Trees in the Real World

Spencer Schneidenbach



@schneidenbach



@schneidenbach



AVIRON

spencer@avironsoftware.com

Have you ever...

- Query a database with a LINQ expression?

```
from product in db.Products
where product.Price < 2.55
select new {
    product.Name,
    product.Price
}
```



Have you ever...

- Render a field in an ASP.NET Core app like this?

```
@Html.EditorFor(e => e.FirstName)
```



Have you ever...

- Select one or more members from an object?

```
ForMember(x => x.Name)
```



If you've ever done any of those

- Then you've used expression trees

```
ForMember(x => x.Name)
```



**So what are expression
trees anyways?**



@schneidenbach

Let's talk lambdas first



@schneidenbach

Lambda expressions

```
Func<string, string> toUpper = str => str.ToUpper();  
var result = toUpper("spencer");
```



Lambda expressions

```
Func<string, string> toUpper = str => str.ToUpper();  
var result = toUpper("spencer");           //SPENCER
```



Lambda expressions

```
Expression<Func<string, string>> toUpper = str => str.ToUpper();  
var result = toUpper("spencer");
```



Lambda expressions

```
Expression<Func<string, string>> toUpper = str => str.ToUpper();  
var result = toUpper("spencer");    //does not compile
```



What the heck?

```
Expression<Func<string, string>> toUpper = str => str.ToUpper();  
var result = toUpper("spencer");
```

```
Func<string, string> toUpper = str => str.ToUpper();  
var result = toUpper("spencer");
```



Key difference

- Lambdas do the thing
- Expressions describe the lambda that does the thing



Homoiconicity

```
Expression<Func<string, string>> toUpper = str => str.ToUpper();  
var result = toUpper("spencer");
```

```
Func<string, string> toUpper = str => str.ToUpper();  
var result = toUpper("spencer");
```



So what can we do with expressions?

- **Read them**
- Create them
- Use them



Lambda expressions

```
Expression<Func<string, string>> toUpper = str => str.ToUpper();
```



```
str => str.ToUpper()
```



@schneidenbach

`str => str.ToUpper()`

`ParameterExpression`

`=>`

`str`



`str => str.ToUpper()`

ParameterExpression

`str`

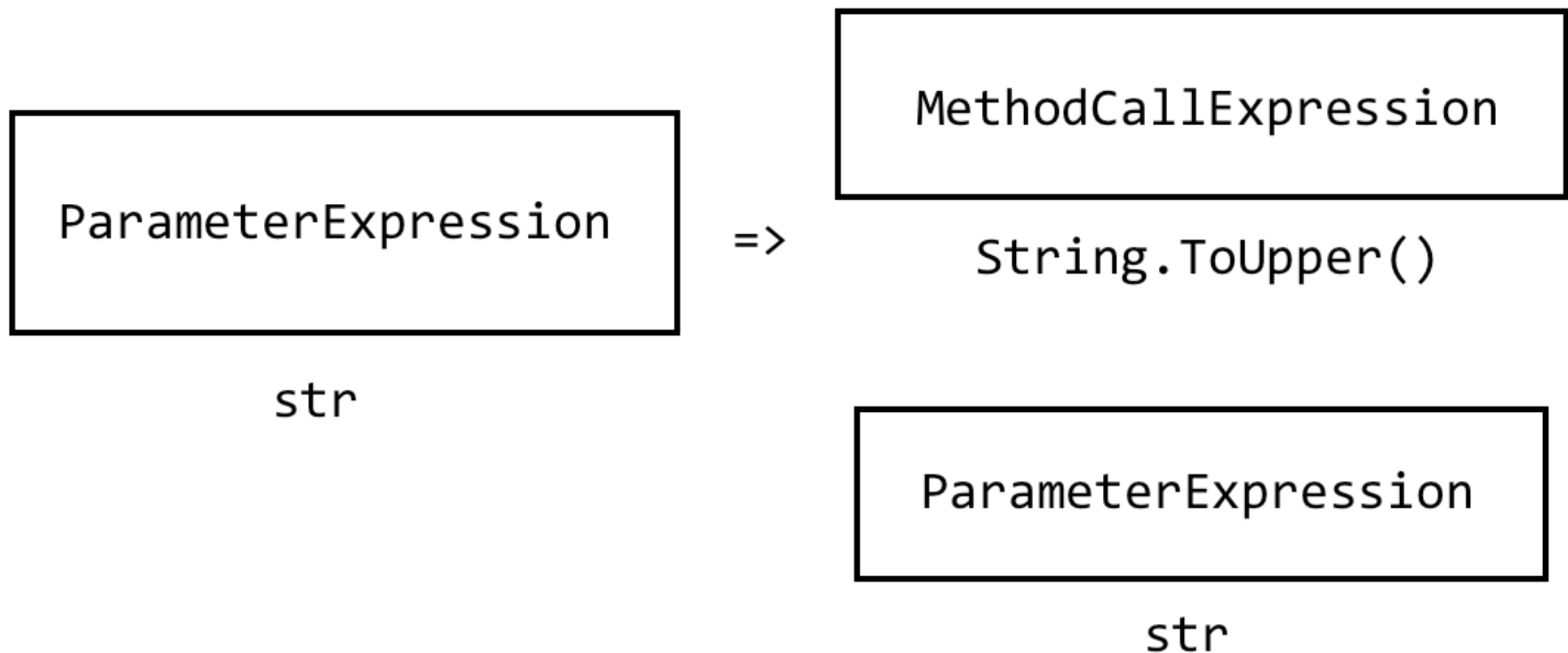
`=>`

MethodCallExpression

`String.ToUpper()`



`str => str.ToUpper()`



So what can we do with expressions?

- Read them
- **Create them**
- Use them



Lambda expressions

```
Expression<Func<string, string>> toUpper = str => str.ToUpper();
```



Lambda expressions

```
Expression<Func<string, string>> toUpper = str => str.ToUpper();
```

Useful, but mainly for libraries to use them



@schneidenbach

Expression API



@schneidenbach

ParameterExpression

- Represents a parameter to a function

```
Expression.Parameter(typeof(string));
```

```
Expression.Parameter(typeof(string), "myStr"); //with a name
```



ConstantExpression

- Represents a declared value (e.g. 1, “str”, etc)

```
Expression.Constant("spencer");
```



MethodCallExpression

- Represents a call to a method

```
var prm = Expression.Parameter(typeof(string));  
var toUpper = typeof(string).GetMethod("ToUpper", Type.EmptyTypes);  
  
Expression.Call(prm, toUpper);
```



Let's build this guy!

```
Expression<Func<string, string>> toUpper = str => str.ToUpper();
```



@schneidenbach

str => str.ToUpper()

```
var prm = Expression.Parameter(typeof(string), "str");  
var toUpper = typeof(string).GetMethod("ToUpper", Type.EmptyTypes);  
  
var body = Expression.Call(prm, toUpper);  
  
var lambda = Expression.Lambda(body, prm);
```



So what can we do with expressions?

- Read them
- Create them
- **Use them**



Your favorite libraries

- Entity Framework/EF Core
- AutoMapper



Entity Framework

- Translates LINQ expressions to SQL



Entity Framework

```
var products = db.Products
    .Where(p => p.Name == "eggs")
    .OrderByDescending(p => p.Price);
```



Entity Framework

```
var products = db.Products  
    .Where(p => p.Name == "eggs")  
    .OrderByDescending(p => p.Price);
```

```
SELECT * FROM Products  
WHERE Name = 'eggs'  
ORDER BY Price DESC
```



Entity Framework

```
var products = db.Products  
    .Where(p => p.Name == "eggs")  
    .OrderByDescending(p => p.Price);
```

?

```
SELECT * FROM Products  
WHERE Name = 'eggs'  
ORDER BY Price DESC
```





@schneidenbach

Entity Framework

```
var products = db.Products  
    .Where(p => p.Name == "eggs")  
    .OrderByDescending(p => p.Price);
```

db.Products is an IQueryable



Enumerable

```
Enumerable<T>.Where(Func<T, bool> predicate)
```



@schneidenbach

Queryable

```
Queryable<T>.Where(Expression<Func<T, bool>> predicate)
```



@schneidenbach

Queryable

```
Queryable<T>.Where(Expression<Func<T, bool>> predicate)
```

This one can be interpreted at runtime!



@schneidenbach

ExpressionVisitor

Used to read and operate on expressions



@schneidenbach

Entity Framework

```
var products = db.Products  
    .Where(p => p.Name == "eggs")  
    .OrderByDescending(p => p.Price);
```



Entity Framework

```
.Where(p => p.Name == "eggs")
```

```
WHERE Name = 'eggs'
```



Entity Framework

```
.Where(p => p.Name == "eggs")
```

This is what is known as a binary expression



BinaryExpression

Represents an operation with a left side, right side, and an operator



@schneidenbach

BinaryExpression

Property	What is it?
Left	Expression
Right	Expression
NodeType	Equal, NotEqual, etc.



ExpressionVisitor

Reads each part of the expression and does *something*
based on what it's reading



@schneidenbach

`p.Name == "Eggs"`

Part of function	SQL
<code>p.Name</code>	<code>Name</code>
<code>==</code>	<code>=</code>
<code>"eggs"</code>	<code>eggs'</code>



Entity Framework

```
var products = db.Products
    .Where(p => p.Name == "eggs")
    .OrderByDescending(p => p.Price);
```

ExpressionVisitor

```
SELECT * FROM Products
WHERE Name = 'eggs'
ORDER BY Price DESC
```



```

[Project1].[downtimeId] AS [downtimeId],
CASE WHEN ([Extent12].[downtimeStart] > @p__linq__7) THEN [Extent13].[downtimeStart] ELSE @p__linq__8 END AS [C1],
CASE WHEN ([Extent14].[equipmentID] IS NULL) THEN 0 ELSE [Extent15].[equipmentID] END AS [C2],
CASE WHEN ([Extent16].[equipmentID] IS NULL) THEN N''Unit Overhead'' ELSE [Extent18].[equipmentCode] END AS [C3],
CASE WHEN ( CAST( [Project1].[downtimeEquipmentStart] AS datetime2) > @p__linq__9) THEN CAST( [Project1].[downtimeEquipmentStart] AS
datetime2) ELSE @p__linq__10 END AS [C4],
CASE WHEN ( CAST( [Project1].[downtimeEquipmentEnd] AS datetime2) < @p__linq__11) THEN CAST( [Project1].[downtimeEquipmentEnd] AS
datetime2) ELSE @p__linq__12 END AS [C5],
CASE WHEN ([Extent19].[standardHourRate] IS NULL) THEN cast(0 as decimal(18)) ELSE [Extent20].[standardHourRate] END AS [C6],
CASE WHEN ([Extent21].[equipmentID] IS NULL) THEN 0 ELSE [Filter2].[reportingSequence] END AS [C7]
FROM
        (SELECT
        @p__linq__0 AS [p__linq__0],
        [Extent1].[downtimeId] AS [downtimeId],
        [Extent1].[equipmentID] AS [equipmentID],
        [Extent1].[downtimeEquipmentStart] AS [downtimeEquipmentStart],
        [Extent1].[downtimeEquipmentEnd] AS [downtimeEquipmentEnd]
        FROM [dbo].[DowntimeEquipment] AS [Extent1] ) AS [Project1]
OUTER APPLY (SELECT [Extent2].[reportingSequence] AS [reportingSequence]
FROM [dbo].[ProcessUnitEquipment] AS [Extent2]
INNER JOIN [dbo].[Downtime] AS [Extent3] ON [Extent3].[equipmentID] = [Extent2].[equipmentID]
LEFT OUTER JOIN (SELECT
        [Extent4].[downtimeId] AS [downtimeId]
        FROM [dbo].[Downtime] AS [Extent4]
        WHERE [Project1].[downtimeId] = [Extent4].[downtimeId] ) AS [Project2] ON 1 = 1
WHERE ([Project1].[downtimeId] = [Extent3].[downtimeId]) AND ([Extent2].[processUnitID] = @p__linq__0) AND (@p__linq__0 IS NOT NULL) )
AS [Filter2]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent5] ON [Project1].[downtimeId] = [Extent5].[downtimeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent6] ON [Project1].[downtimeId] = [Extent6].[downtimeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent7] ON [Project1].[downtimeId] = [Extent7].[downtimeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent8] ON [Project1].[downtimeId] = [Extent8].[downtimeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent9] ON [Project1].[downtimeId] = [Extent9].[downtimeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent10] ON [Project1].[downtimeId] = [Extent10].[downtimeId]
LEFT OUTER JOIN [dbo].[DownTimeType] AS [Extent11] ON [Extent10].[downTimeTypeId] = [Extent11].[downTimeTypeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent12] ON [Project1].[downtimeId] = [Extent12].[downtimeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent13] ON [Project1].[downtimeId] = [Extent13].[downtimeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent14] ON [Project1].[downtimeId] = [Extent14].[downtimeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent15] ON [Project1].[downtimeId] = [Extent15].[downtimeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent16] ON [Project1].[downtimeId] = [Extent16].[downtimeId]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent17] ON [Project1].[downtimeId] = [Extent17].[downtimeId]
LEFT OUTER JOIN [dbo].[Equipment] AS [Extent18] ON [Extent17].[equipmentID] = [Extent18].[equipmentID]
LEFT OUTER JOIN [dbo].[Equipment] AS [Extent19] ON [Project1].[equipmentID] = [Extent19].[equipmentID]
LEFT OUTER JOIN [dbo].[Equipment] AS [Extent20] ON [Project1].[equipmentID] = [Extent20].[equipmentID]
LEFT OUTER JOIN [dbo].[Downtime] AS [Extent21] ON [Project1].[downtimeId] = [Extent21].[downtimeId]

```



@schneidenbach

ExpressionVisitor

Used to read and operate on expressions
...or even modify them... kind of



```
public class ToUpperVisitor : ExpressionVisitor
{
    public override Expression Visit(Expression node)
    {
        if (node.NodeType == ExpressionType.Parameter)
        {
            return base.Visit(node);
        }

        if (node.Type == typeof(string))
        {
            var toUpper = typeof(string).GetMethod("ToUpper", Type.EmptyTypes);
            var methodCallExpression = Expression.Call(node, toUpper);
            return methodCallExpression;
        }
        return base.Visit(node);
    }
}
```



```

public class ToUpperVisitor : ExpressionVisitor
{
    public override Expression Visit(Expression node)
    {
        if (node.NodeType == ExpressionType.Parameter)
        {
            return base.Visit(node);
        }

        if (node.Type == typeof(string))
        {
            var toUpper = typeof(string).GetMethod("ToUpper", Type.EmptyTypes);
            var methodCallExpression = Expression.Call(node, toUpper);
            return methodCallExpression;
        }
        return base.Visit(node);
    }
}

```

```

Expression<Func<string, string>> spencyString =
    s => s + " belongs to Spencer";

```

```

var toUpperVisitor = new ToUpperVisitor();
var expressed = toUpperVisitor.VisitAndConvert(spencyString, null);

```

```

Console.WriteLine(expressed.Compile().DynamicInvoke("the cheese"));

```

```

//output: THE CHEESE BELONGS TO SPENCER

```



@schneidenbach

My Real World



@schneidenbach

SQL Model Mapper

- Needed to map one entity to another
- Stored procs vs. code



SQL Model Mapper

- Salesforce Customer -> Quickbooks Customer



```
public class SalesforceCustomer
{
    public string CustomerName { get; set; }
    public DateTime? CreateDate { get; set; }
}
```

```
public class QuickbooksCustomer
{
    public string Name { get; set; }
    public DateTime? OpenDate { get; set; }
}
```



```
INSERT QuickbooksCustomers (Name, OpenDate)
SELECT CustomerName, CreateDate
FROM SalesForceCustomers
```



```
INSERT QuickbooksCustomers (Name, OpenDate)
SELECT CustomerName, CreateDate
FROM SalesForceCustomers
```

Now do this for 100's of objects and 1,000's of properties, and never make a mistake



@schneidenbach

```
public class SfCustomerToQbCustomer
{
    public SfCustomerToQbCustomer()
    {
        SourceField(sfc => sfc.CustomerName)
            .IsEqualTo(qbc => qbc.Name.Trim());
        SourceField(sfc => sfc.CreateDate)
            .IsEqualTo(qbc => qbc.OpenDate ?? DateTime.Now);
    }
}
```



Tasks

- Write an expression visitor
- Handle any type of expression we want to translate



qbc => qbc.Name.Trim()



@schneidenbach


```
qbc => qbc.Name.Trim()
```

LTRIM(RTRIM(Name))



qbc => qbc.CreateDate ?? DateTime.Now



@schneidenbach

qbc => qbc.CreateDate ?? DateTime.Now

ISNULL(CreateDate, GETDATE())



@schneidenbach

```
INSERT QuickbooksCustomers (Name, OpenDate)
SELECT LTRIM(RTRIM(CustomerName)), ISNULL(CreateDate, GETDATE())
FROM SalesForceCustomers
```



Benefits

- Predictable
- Gave developers a natural way to express mappings
- Saved 1,000's of hours of developer time



Very simple example



@schneidenbach

<https://dotnetfiddle.net/PUij3K>



@schneidenbach

Order by...string?



@schneidenbach

api/Customers?orderBy=name



@schneidenbach

```
db.Customers.OrderBy(c => c.Name)
db.Customers.OrderBy("Name")      //doesn't exist
```



```
public async Task<IActionResult> GetCustomers(string orderBy){
    var customersQuery = _context.Customers;
    switch (orderBy)
    {
        case "Name":
            customersQuery = customersQuery.OrderBy(c => c.Name);
            break;
        case "Age":
            customersQuery = customersQuery.OrderBy(c => c.Age);
            break;
        case "Address":
            customersQuery = customersQuery.OrderBy(c => c.Address);
            break;
        case "Id":
            customersQuery = customersQuery.OrderBy(c => c.Id);
            break;
        default:
            break;
    }

    return Ok(customers);
}
```



**Solution: cook an
expression!**



@schneidenbach

```
db.Customers.OrderBy(c => c.Name)
```



@schneidenbach

Goals

- Cook our expression
- Apply it to our IQueryable



```
IQueryable<T> OrderByPropertyOrField<T>(
    this IQueryable<T> queryable,
    string propertyOrFieldName,
    bool ascending
)
```



```
IQueryable<T> OrderByPropertyOrField<T>(
    this IQueryable<T> queryable,
    string propertyOrFieldName,
    bool ascending
)
{
    var elementType = typeof (T);
    var parameter = Expression.Parameter(elementType);
    var prop = Expression.PropertyOrField(parameter, propertyOrFieldName);
    var selector = Expression.Lambda(prop, parameter);
}
```




```
Queryable.OrderBy<TSource, TKey>(
    IQueryable<TSource>,
    Expression<Func<TSource, TKey>>
)
```



```
Queryable.OrderBy<TSource, TKey>(IQueryable<TSource>, Expression<Func<TSource, TKey>>)
```

```
var selector = Expression.Lambda(prop, parameter);

var orderByMethodName = ascending ? "OrderBy" : "OrderByDescending";
var orderByExpression = Expression.Call(
    typeof (Queryable),           //the type whose function we want to call
    orderByMethodName,           //the name of the method
    new[] {elementType, prop.Type}, //the generic type signature
    queryable.Expression,        //parameter
    selector);                   //parameter
```



```
Queryable.OrderBy<TSource, TKey>(IQueryable<TSource>, Expression<Func<TSource, TKey>>)
```

```
var selector = Expression.Lambda(prop, parameter);
```

```
var orderByMethodName = ascending ? "OrderBy" : "OrderByDescending";
```

```
var orderByExpression = Expression.Call(  
    typeof (Queryable),           //the type whose function we want to call  
    orderByMethodName,           //the name of the method  
    new[] {elementType, prop.Type}, //the generic type signature  
    queryable.Expression,        //parameter  
    selector);                   //parameter
```



```
Queryable.OrderBy<TSource, TKey>(IQueryable<TSource>, Expression<Func<TSource, TKey>>)
```

```
var selector = Expression.Lambda(prop, parameter);
```

```
var orderByMethodName = ascending ? "OrderBy" : "OrderByDescending";  
var orderByExpression = Expression.Call(  
    typeof (Queryable),           //the type whose function we want to call  
    orderByMethodName,           //the name of the method  
    new[] {elementType, prop.Type}, //the generic type signature  
    queryable.Expression,        //parameter  
    selector);                   //parameter
```



```
Queryable.OrderBy<TSource, TKey>(IQueryable<TSource>, Expression<Func<TSource, TKey>>)
```

```
var selector = Expression.Lambda(prop, parameter);
```

```
var orderByMethodName = ascending ? "OrderBy" : "OrderByDescending";  
var orderByExpression = Expression.Call(  
    typeof (Queryable),           //the type whose function we want to call  
    orderByMethodName,           //the name of the method  
    new[] {elementType, prop.Type}, //the generic type signature  
    queryable.Expression,        //parameter  
    selector);                   //parameter
```



```
Queryable.OrderBy<TSource, TKey>(IQueryable<TSource>, Expression<Func<TSource, TKey>>)
```

```
var selector = Expression.Lambda(prop, parameter);
```

```
var orderByMethodName = ascending ? "OrderBy" : "OrderByDescending";  
var orderByExpression = Expression.Call(  
    typeof (Queryable),           //the type whose function we want to call  
    orderByMethodName,           //the name of the method  
    new[] {elementType, prop.Type}, //the generic type signature  
    queryable.Expression,        //parameter  
    selector);                   //parameter
```



```
Queryable.OrderBy<TSource, TKey>(IQueryable<TSource>, Expression<Func<TSource, TKey>>)
```

```
var selector = Expression.Lambda(prop, parameter);

var orderByMethodName = ascending ? "OrderBy" : "OrderByDescending";
var orderByExpression = Expression.Call(
    typeof (Queryable),           //the type whose function we want to call
    orderByMethodName,           //the name of the method
    new[] {elementType, prop.Type}, //the generic type signature
    queryable.Expression,        //parameter
    selector);                   //parameter
```



```
Queryable.OrderBy<TSource, TKey>(
    IQueryable<TSource>,
    Expression<Func<TSource, TKey>>
)
```



@schneidenbach

```
Queryable.OrderBy<TSource, TKey>(IQueryable<TSource>, Expression<Func<TSource, TKey>>)
```

```
var selector = Expression.Lambda(prop, parameter);
```

```
var orderByMethodName = ascending ? "OrderBy" : "OrderByDescending";  
var orderByExpression = Expression.Call(  
    typeof (Queryable),           //the type whose function we want to call  
    orderByMethodName,           //the name of the method  
    new[] {elementType, prop.Type}, //the generic type signature  
    queryable.Expression,        //parameter  
    selector);                   //parameter
```



<https://dotnetfiddle.net/5PlIF>



@schneidenbach

Rules engine?



@schneidenbach

Define structure

```
public class Rule
{
    public string PropertyName { get; set; }
    public Operation Operation { get; set; }
    public object Value { get; set; }
}
```

```
public enum Operation
{
    GreaterThan,
    LessThan,
    Equal
}
```



Employee search criteria

```
new Rule {  
    PropertyName = "Name",  
    Operation = Operation.Equal,  
    Value = "gary"  
},  
new Rule {  
    PropertyName = "HireDate",  
    Operation = Operation.GreaterThan,  
    Value = new DateTime(2016, 1, 1)  
}
```



Cook an expression

```
var parameter = Expression.Parameter(typeof(Employee));
BinaryExpression binaryExpression = null;

foreach (var rule in rules)
{
    var prop = Expression.Property(parameter, rule.PropertyName);
    var value = Expression.Constant(rule.Value);
    var newBinary = Expression.MakeBinary(rule.Operation, prop, value);

    binaryExpression =
        binaryExpression == null
        ? newBinary
        : Expression.MakeBinary(Also, binaryExpression, newBinary);
}
```



Cook an expression

```
var parameter = Expression.Parameter(typeof(Employee));
BinaryExpression binaryExpression = null;

foreach (var rule in rules)
{
    var prop = Expression.Property(parameter, rule.PropertyName);
    var value = Expression.Constant(rule.Value);
    var newBinary = Expression.MakeBinary(rule.Operation, prop, value);

    binaryExpression =
        binaryExpression == null
        ? newBinary
        : Expression.MakeBinary(AndAlso, binaryExpression, newBinary);
}
```



Cook an expression

```
var parameter = Expression.Parameter(typeof(Employee));
BinaryExpression binaryExpression = null;

foreach (var rule in rules)
{
    var prop = Expression.Property(parameter, rule.PropertyName);
    var value = Expression.Constant(rule.Value);
    var newBinary = Expression.MakeBinary(rule.Operation, prop, value);

    binaryExpression =
        binaryExpression == null
        ? newBinary
        : Expression.MakeBinary(AndAlso, binaryExpression, newBinary);
}
```



Cook an expression

```
var parameter = Expression.Parameter(typeof(Employee));
BinaryExpression binaryExpression = null;

foreach (var rule in rules)
{
    var prop = Expression.Property(parameter, rule.PropertyName);
    var value = Expression.Constant(rule.Value);
    var newBinary = Expression.MakeBinary(rule.Operation, prop, value);

    binaryExpression =
        binaryExpression == null
        ? newBinary
        : Expression.MakeBinary(Also, binaryExpression, newBinary);
}
```



Cook an expression

```
var parameter = Expression.Parameter(typeof(Employee));
BinaryExpression binaryExpression = null;

foreach (var rule in rules)
{
    var prop = Expression.Property(parameter, rule.PropertyName);
    var value = Expression.Constant(rule.Value);
    var newBinary = Expression.MakeBinary(rule.Operation, prop, value);

    binaryExpression =
        binaryExpression == null
        ? newBinary
        : Expression.MakeBinary(AndAlso, binaryExpression, newBinary);
}
```



Employee search criteria

```
new Rule {  
    PropertyName = "Name",  
    Operation = Operation.Equal,  
    Value = "gary"  
},  
new Rule {  
    PropertyName = "HireDate",  
    Operation = Operation.GreaterThan,  
    Value = new DateTime(2016, 1, 1)  
}
```

```
e => (e.Name == "gary") && (e.HireDate > new DateTime(2016, 1, 1))
```



<https://dotnetfiddle.net/iobiuW>



@schneidenbach

Making it more complex

```
e => (e.Name == "gary") && (e.HireDate > new DateTime(2016, 1, 1))
```



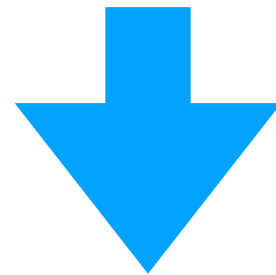
@schneidenbach

Making it more complex

```
e => (e.Name == "gary" || e.Name == "spencer") &&  
      (e.HireDate > new DateTime(2015, 1, 1))
```



Making it more complex



```
e => (e.Name == "gary" || e.Name == "spencer") &&  
      (e.HireDate > new DateTime(2015, 1, 1))
```



```
public abstract class Rule
{
    public abstract Expression BuildExpression(ParameterExpression parameter);
}
```



@schneidenbach

```
public class RuleGroup : Rule
{
    public List<Rule> SubRules { get; set; }
    public ExpressionType CombineWith { get; set; }

    public RuleGroup(List<Rule> subRules, ExpressionType combineWith)
    {
        SubRules = subRules;
        CombineWith = combineWith;
    }

    public override Expression BuildExpression(ParameterExpression parameter)
    {
        var expressions = SubRules.Select(rule => rule.BuildExpression(parameter));
        var combined = expressions.Aggregate((left, right) => Expression.MakeBinary(CombineWith, left, right));
        return combined;
    }
}
```




```
public class RulePrimitive : Rule
{
    public string PropertyName { get; set; }
    public ExpressionType Operation { get; set; }
    public object Value { get; set; }

    public RulePrimitive(string propertyName, ExpressionType operation, object value)
    {
        PropertyName = propertyName;
        Operation = operation;
        Value = value;
    }

    public override Expression BuildExpression(ParameterExpression parameter)
    {
        var left = Expression.Property(parameter, PropertyName);
        var right = Expression.Constant(Value);
        return Expression.MakeBinary(Operation, left, right);
    }
}
```



<https://dotnetfiddle.net/XuF4ci>



@schneidenbach

Some other things



@schneidenbach

Expression.Block



@schneidenbach

Expression.Block

- Used for grouping expressions in a “block”
- Can declare variables for use inside of this block

```
Expression.Variable(typeof(int), "x");
```



```
// Define a variable
var variable = Expression.Variable(typeof(int), "x");

// Create an expression block
var blockExpr = Expression.Block(
    new[] { variable },
    Expression.Assign(variable, Expression.Constant(5)),
    Expression.Add(variable, Expression.Constant(10))
);

// Compile and execute the expression block
var lambda = Expression.Lambda<Func<int>>(blockExpr).Compile();
var result = lambda();

Console.WriteLine("Result: " + result); // Output: Result: 15
```



<https://dotnetfiddle.net/LOe2iO>



@schneidenbach

Here Be Dragons

(Problems and limitations)



@schneidenbach

C# compiler == magic



@schneidenbach

```
Expression<Func<string, string, string>> combineStringsExp =  
    (str1, str2) => str1 + str2;
```



```
var str1Param = Expression.Parameter(typeof(string));  
var str2Param = Expression.Parameter(typeof(string));  
  
var combineThem = Expression.MakeBinary(ExpressionType.Add, str1Param, str2Param);
```



```
var str1Param = Expression.Parameter(typeof(string));  
var str2Param = Expression.Parameter(typeof(string));  
  
var combineThem = Expression.MakeBinary(ExpressionType.Add, str1Param, str2Param);
```

EXCEPTION: The binary operator Add is not defined for the types
'System.String' and 'System.String'



```
Expression<Func<string, string, string>> combineStringsExp =  
    (str1, str2) => str1 + str2;
```

This uses string.Concat



@schneidenbach

Conversions need to be done explicitly



@schneidenbach

**Doesn't support every
language feature**



@schneidenbach

Homoiconicity



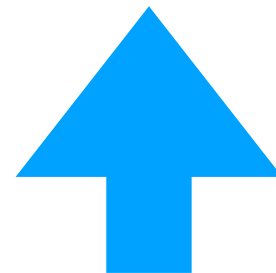
@schneidenbach


```
customers.Where(c => c.Address?.ZipCode == "12345")
```



@schneidenbach

```
customers.Where(c => c.Address?.ZipCode == "12345")
```



(Spencer is paraphrasing from memory)

“We’re not going to update the Expression API. We looked at the work that would be required and we’d basically have to stop developing other features for an entire year and focus on that and that alone.”



@schneidenbach

How to start

- Experiment
- LINQPad
- Google
- Intellisense!



**Most importantly,
EXPERIMENT!**



@schneidenbach

Thank you!



schneids.net



[@schneidenbach](https://twitter.com/schneidenbach)

