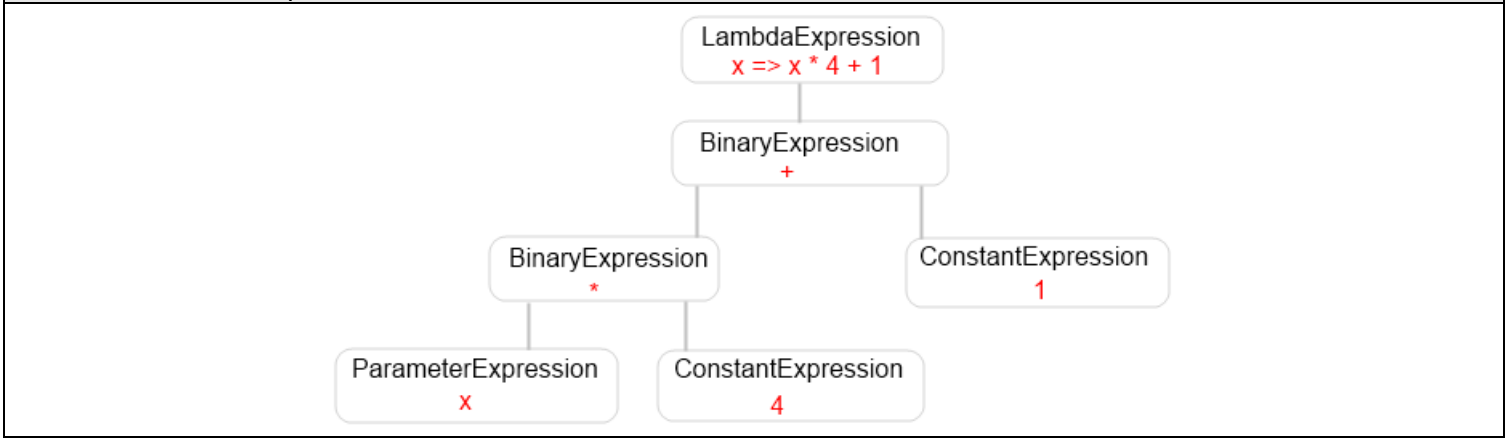
Convirtiendo arboles de expresión lambda en diferentes strings.

Por: Horacio Aldo Tore, escrito en abril del 2010 en Buenos Aires Argentina

Objetivo: Exponer en forma práctica con un ejemplo, como un mismo árbol de expresión (Expression tree) puede ser interpretado y convertido en distintas formas. Es con esta misma lógica como se desarrollo linq to SQL, linq to Dataset, linq to Entities y así varios proveedores más de linq to algo. La idea es escribir una sola expresión lambda de consulta en CSharp y que luego el proveedor de linq se encargue de transformarlo en una sentencia SQL válida para Oracle o para SQL Server dependiendo del proveedor de linq utilizado. Cabe destacar que no es objetivo de este documento detallar las propiedades de LINQ.

Introducción: Las lambda pueden ser utilizadas como *métodos anónimos* o como *arboles de expresión*, trataremos aquí a las mismas como arboles de expresión, en otras palabras *una estructura que representa en forma de árbol las operaciones y el orden en que deben realizarse*. Por ejemplo el árbol de expresión dado por la lambda x => x * 4 + 1 no solo nos daría las operaciones matemáticas, factores y sumandos que la componen, sino también el orden en que dichas operaciones debe efectuarse para obtener el resultado correcto, en este caso primero el producto y luego la suma.

Grafico del arbol de exprecion lambda x=> x * 4 + 1



Así la línea de código `Expression<Func<int, int>> expression = x => x*4+1` genera una estructura de árbol de expresión en la cual lo importante no es darle un valor a x y obtener el resultado sino que lo importante es poder interpretar y obtener información de cómo está escrita esta operación matemática, que hay una multiplicación cuyos factores son un parámetro llamado x del tipo entero y cuyo otro factor es una constante, también entera, cuyo valor es 4, que la primera operación matemática que se debe realizar es el producto y así sucesivamente. Este árbol de expresión nos permitirá a los desarrolladores interpretarlo, navegarlo, serializarlo, ejecutarlo o transformarlo según nuestra conveniencia. La lambda x => x*4+1 por sí sola no nos brinda información alguna sobre su tipo, ya que puede ser del tipo `Expression<Func<int, int>>` o `Func<int, int>` razón por la cual el siguiente código fallará en tiempo de compilación e incluso antes, en tiempo de edición ya que el motor de IntelliSense detecta la ambigüedad establecida por la sobrecarga del método “Convert” o en otras palabras no se puede determinar cuál de las dos definiciones del método “Convert” debe ser invocada.

El siguiente código emite este error: “The call is ambiguous between the following methods or properties”

```
private void MainX()
{
      Convert(x => x * 4 + 1);
}

private void Convert(Expression<Func<int, int>> expressionX) { }
private void Convert(Func<int, int> funX) { }
```

Otro hecho de ambigüedad de características similares al anterior, es el siguiente:

```
var lambdaX = x => x * 4 + 1;
```

La variable “lambdaX” es de tipo implícita (implicit type) no es del tipo explicito (explicit type), esto quiere decir que su tipo será inferido del tipo de dato que se le esta asignado, por ejemplo en `var nombre = "sadosky manuel";` la variable “nombre” tomara el tipo string dado que se le está asignado una constante de este tipo, cabe destacar que desde el momento en que la variable adopta el tipo, este es conservado durante todo el tiempo de vida de la misma sin poder cambiarse, ahora no se puede inferir el tipo de “lambdaX” ya que no se puede determinar si es `Expression<Func<int, int>>` o `Func<int, int>`, razón que genera el siguiente error “Cannot assign lambda expression to an implicitly-typed local variable” .

Desarrollo: Dada la breve introducción ya podemos ingresar al tema que nos proponemos tratar que es cómo convertir un mismo árbol de expresión lambda en diferentes strings. Convertiremos el mismo árbol de expresión lambda x = > x + 1 en dos string uno que nos de cómo resultado "x.Add(1)" y otro "(x) + (1)", en la siguiente tabla veremos unos ejemplos más, para de esta forma lograr entender lo que se quiere plantear.

árbol de expresión lambda	Resultado de la conversión dada por el método ToNaturalFormat	Resultado de la conversión dada por el método ToAnotherFormat
`Expression<Func<int, int>> expression = x => x + 1;`	"x.Add(1)"	"(x) + (1)"
`Expression<Func<int, int>> expression = x => x + 1 - 4;`	"x.Add(1).Subtract(4)"	"((x) + (1)) - (4)"
`Expression<Func<int, int>> expression = x => x + (1 - (x + 1));`	"x.Add(1.Subtract(x.Add(1)))"	"(x) + ((1) - ((x) + (1)))"
`Expression<Func<int, int, int>> expression = (x,y) => y + (1 - (x + 1));`	"y.Add(1.Subtract(x.Add(1)))"	"(y) + ((1) - ((x) + (1)))"

El siguiente fragmento de código muestra la creación de una expresión lambda y la posterior invocación a los métodos ToAnotherFormat y ToNaturalFormat responsables de las diferentes interpretaciones y conversiones:

Main

```
/// <summary>
/// Método principal responsable de invocar a los métodos
/// encargados de convertir la misma expresión lambda a dos formatos distintos.
/// </summary>
private void Main()
```

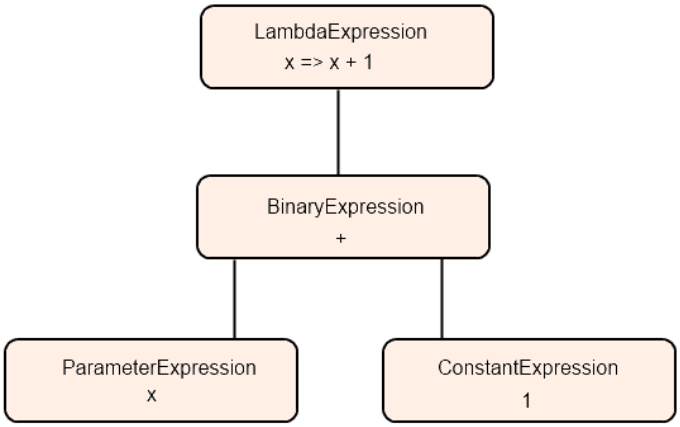
```
{
    // Creacion de la exprecion lambda a convertir
    Expression<Func<int, int>> expression = x => x + 1; // Linea 0

    // retorno esperado: x.Add(1)
    string textInAnotherFormat = ToAnotherFormat(expression);

    // retorno esperado: (x) + (1)
    string textInNaturalFormat = ToNaturalFormat(expression);
}
```

Los métodos ToAnotherFormat y ToNaturalFormat son los responsables de interpretar el árbol de expresión y de convertirlo en un string, podemos ver este árbol en forma gráfica como sigue:

Grafico del arbol de exprecion lambda x=> x + 1



Los métodos de conversión son recursivos y saben cómo convertir cada uno de los nodos que componen el árbol. Es necesario que el parámetro del método de conversión sea del tipo System.Linq.Expressions.Expression ya que es de este del que heredan todas las expresiones, como en este caso LambdaExpression, BinaryExpression etc. Podemos notar que los métodos de conversión aquí presentados no pueden resolver cualquier tipo de expresión lambda, por ejemplo si se pasa como parámetro la siguiente x => x / 1, el programa compilaría pero en tiempo de ejecución se dispararía una excepción del tipo “NotSupportedException” y cuya propiedad “Message” contendría el siguiente texto {" El siguiente tipo no está soportado: Divide "} , esto se debe a que el switch no contempla nodos del tipo ExpressionType.Divide, es buena práctica crear un caso de “default” en la sentencia switch para que contemple los tipos de nodo no implementados ya que de otra forma nuestra aplicación no daría excepción alguna pero si funcionaria en forma anómala.

Podemos reemplazar la "`// Linea 0`" del método “Main” por alguna otra expresión lambda más compleja como ser `Expression<Func<int, int, int>> expression = (x,y) => y + (1 - (x + 1));` y el programa seguirá funcionando correctamente dado el carácter genérico y recursivo de los métodos de conversión, la salida sería la que siguiente : "(y) + ((1) - ((x) + (1)))" y "y.Add(1.Subtract(x.Add(1)))" causada por los métodos ToNaturalFormat y ToAnotherFormat respectivamente.

Código del método ToNaturalFormat

```
/// <summary>
```

```
/// Convierte la expresión lambda x = > x + 1 en un string de la forma "(x) + (1)"
/// </summary>
/// <param name="expression">
/// Arbol de expresión a convertir.
/// </param>
/// <returns>
/// Resultado de la conversion.
/// </returns>
public string ToNaturalFormat(Expression expression)
{
    string text = "";
    BinaryExpression binaryExpression;
    string textLeft;
    string textRight;

    switch (expression.NodeType)
    {
        case ExpressionType.Lambda:
            LambdaExpression lambdaExpression = (LambdaExpression)expression;
            text = ToNaturalFormat(lambdaExpression.Body);
            break;
        case ExpressionType.Add:
            text = "({0}) + ({1})";
            binaryExpression = ((BinaryExpression)expression);
            textLeft = ToNaturalFormat(binaryExpression.Left);
            textRight = ToNaturalFormat(binaryExpression.Right);
            text = string.Format(text, textLeft, textRight);
            break;
        case ExpressionType.Subtract:
            text = "({0}) - ({1})";
            binaryExpression = ((BinaryExpression)expression);
            textLeft = ToNaturalFormat(binaryExpression.Left);
            textRight = ToNaturalFormat(binaryExpression.Right);
            text = string.Format(text, textLeft, textRight);
            break;
        case ExpressionType.Parameter:
            ParameterExpression parameterExpression = ((ParameterExpression)expression);
            text = parameterExpression.Name;
            break;
        case ExpressionType.Constant:
            ConstantExpression constantExpression = ((ConstantExpression)expression);
            text = constantExpression.Value.ToString();
            break;
        default:
            throw new NotSupportedException("El siguiente tipo no esta soportado: " +
expression.NodeType.ToString());
            break;
    }
    return text;
}
```

Código del método ToAnotherFormat

```
/// <summary>
/// Convierte la expresión lambda x = > x + 1 en un string de la forma "x.Add(1)"
/// </summary>
```

```
/// <param name="expression">
/// Arbol de expresión a convertir.
/// </param>
/// <returns>
/// Resultado de la conversion.
/// </returns>
public string ToAnotherFormat(Expression expression)
{
    string text = "";
    BinaryExpression binaryExpression;
    string textLeft;
    string textRight;

    switch (expression.NodeType)
    {
        case ExpressionType.Lambda:
            LambdaExpression lambdaExpression = (LambdaExpression)expression;
            text = ToAnotherFormat(lambdaExpression.Body);
            break;
        case ExpressionType.Add:
            text = "{0}.Add({1})";
            binaryExpression = ((BinaryExpression)expression);
            textLeft = ToAnotherFormat(binaryExpression.Left);
            textRight = ToAnotherFormat(binaryExpression.Right);
            text = string.Format(text, textLeft, textRight);
            break;
        case ExpressionType.Subtract:
            text = "{0}.Subtract({1})";
            binaryExpression = ((BinaryExpression)expression);
            textLeft = ToAnotherFormat(binaryExpression.Left);
            textRight = ToAnotherFormat(binaryExpression.Right);
            text = string.Format(text, textLeft, textRight);
            break;
        case ExpressionType.Parameter:
            ParameterExpression parameterExpression = ((ParameterExpression)expression);
            text = parameterExpression.Name;
            break;
        case ExpressionType.Constant:
            ConstantExpression constantExpression = ((ConstantExpression)expression);
            text = constantExpression.Value.ToString();
            break;
        default:
            throw new NotSupportedException("El siguiente tipo no esta soportado: " +
expression.NodeType.ToString());
            break;
    }
    return text;
}
```

Conclusión del desarrollo: Volviendo a nuestro ejemplo, tratar de hacer un intérprete (parser) de expresiones matemáticas con solo sumas, restas y cualquier nivel de paréntesis que indican la prioridad de las operaciones a realizar hubiese sido muy difícil de desarrollar sin la existencia de los arboles de expresión lambda, imaginemos tener que interpretar algo tan simple como lo que sigue: y + (1 - (x + 1)) teniendo que identificar las operaciones, el orden en que se deben realizar las mismas y los posibles errores de sintaxis o escritura que pudieran haberse cometido. Es evidente el orden de los sumandos

no altera la suma (Ley conmutativa), pero recordemos que el objetivo de los arboles de expresión no es ejecutar el código sino describir con precisión como ha sido escrito este. Si la operación matemática está mal escrita, el código no compilará, razón por la cual no es necesario que verifiquemos la correcta escritura de la misma como ser por ejemplo la falta del cierre de un paréntesis o intentar sumar un valor que no es del tipo numérico, la definición de la expresión lambda dada por `Expression<Func<int, int>>` nos garantiza lo antes mencionado.

Conclusión: Los arboles de expresión lambda son la solución al problema de interpretar pequeños trozos de código nativo, y hacer algún proceso a partir de dicha interpretación, como transformarlo en otros códigos compatibles con otras plataformas como lo hace LINQ to SQL que trasforma las lambda en sentencias de Transact-SQL o LINQ to Oracle que las transforma en PL/SQL, de esta forma las consultas a la base de datos serian independientes de la misma ya que estarían escritas en código nativo (por ejemplo CSharp de .net) lo que nos permitiría cambiar con facilidad de proveedor de base de datos si fuera necesario, con un impacto mínimo en el código de nuestro sistema. Otro punto de vista es que estamos escribiendo las consultas en forma *declarativa* (describimos la consulta) y no *imperativa* (como de debe hacerse la consulta), nuestro proveedor de LINQ será el encargado de transformar dicha descripción de la consulta en cómo debe hacerse la misma ya que la descripción de la consulta es la misma y es independientemente de la base de datos en que se debe ejecutar dicha consulta. Por ejemplo, describiremos un filtro con todos los usuarios cuya edad es mayor a 90 años y nuestro proveedor de LINQ to SQL se encargaría de trasformar esto a algo del estilo de "SELECT * FROM usuarios WHERE edad > 90" y nuestro proveedor de LINQ to TXT se encargaría de … y etc.

Links utilizados para desarrollar este documento:

Link	Descripción	Idioma
http://www.clikear.com/C_Desmitificando_expresiones_18846.aspx	C#: Desmitificando las expresiones lambda (I)	Español
http://www.variablenotfound.com/2009/03/c-desmitificando-las-expresiones-lambda_29.html	C#: Desmitificando las expresiones lambda (II)	Español
http://www.variablenotfound.com/2009/03/c-desmitificando-las-expresiones-lambda_2829.html	C#: Desmitificando las expresiones lambda (y III)	Español
http://msdn.microsoft.com/spain/library/bb384061.aspx	Variables locales con asignación implícita de tipos (Guía de programación de C#)	Español
http://marlongrech.wordpress.com/2008/01/08/working-with-expression-trees-part-1/	Working with Expression Trees	Ingles