Expression Trees

XML Serialisation & Deserialisation, Evaluation, Postfix

This assignment will give you practical experience with serialisation topics and lambda-based expression trees.

For a general view on **.NET Serialization**, see this doc:

https://docs.microsoft.com/en-us/dotnet/standard/serialization/

> Serialization is the process of converting the state of an object into a form that can be persisted or transported. The complement of serialization is deserialization, which converts a stream into an object. Together, these processes allow data to be stored and transferred.

These topics are very important for distributed applications. Here we use XML serialisation. The appendix shows basic code for basic serialisation and deserialization.

Linq expression trees are essential for the versatility of Linq, allowing the targeted translation of higher-order functions involving lambda expressions, to a wide variety of formats, e.g. to methods, to SQL, or to URL.

These trees have been briefly discussed in the lectures. For their detailed structure, see these docs:

https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.expression?view=netframework-4.8

https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.lambdaexpression?view=netframework-4.8

These are the latest versions, but for our purposes there are no critical differences.

Linqpad is also great in visualising the structure of expression trees. The Appendix shows a Linqpad dump of such a tree structure.

This assignment has a required part (**A#6**) and a bonus part (**A#6B**), both in **C#**. There will be an additional bonus part for a somehow similar **F#** version (**A#6C**), with a separate handout and other time limits.

Each part carries the same weight: **1 course mark**, thus the total can be 2 marks (including the C# bonus) or 3 marks (also including the F# bonus).

A#6 "what to do" description

At each run, your program reads a text from stdin, representing one single XML serialisation of a simple lambda expression of this format:

```
Expression<Func<int, int, … int>>
```

Our simple lambdas use only int parameters, int constants, and the four basic int binary operators (+, -, *, /). No free variables, no closures, nothing else.

Your task is to recreate the original lambda expression and evaluate it in one of the following ways, and print the result on one single line:

- If the lambda has no parameters, then just evaluate it.
- If the lambda has one parameter, then evaluate it with the argument (1).
- If the lambda has two parameters, then evaluate it with the arguments (1, 2).
- If the lambda has three parameters, then evaluate it with the arguments (1,2,3).
- Otherwise, print a question mark (?), w/o any surrounding spaces.

A#6B "what to do" description

Your program reads the same XML serialisations as above (**A#6**)

However, the output must be an equivalent postfix (aka reverse Polish) expression for the tree body (the body is the expression after the equal sign).

The postfix notation must only consist of: parameter names (e.g. x), int constants (e.g. 2, -3), operator symbols (+, -, *, /) - not the enumeration names appearing in the tree, Add, Subtract, Multiply, Divide - separated by single spaces and nothing else.

Caveat

Standard Linq expressions trees cannot be directly serialised and deserialised. There are several reasons for this, but we do not go into details.

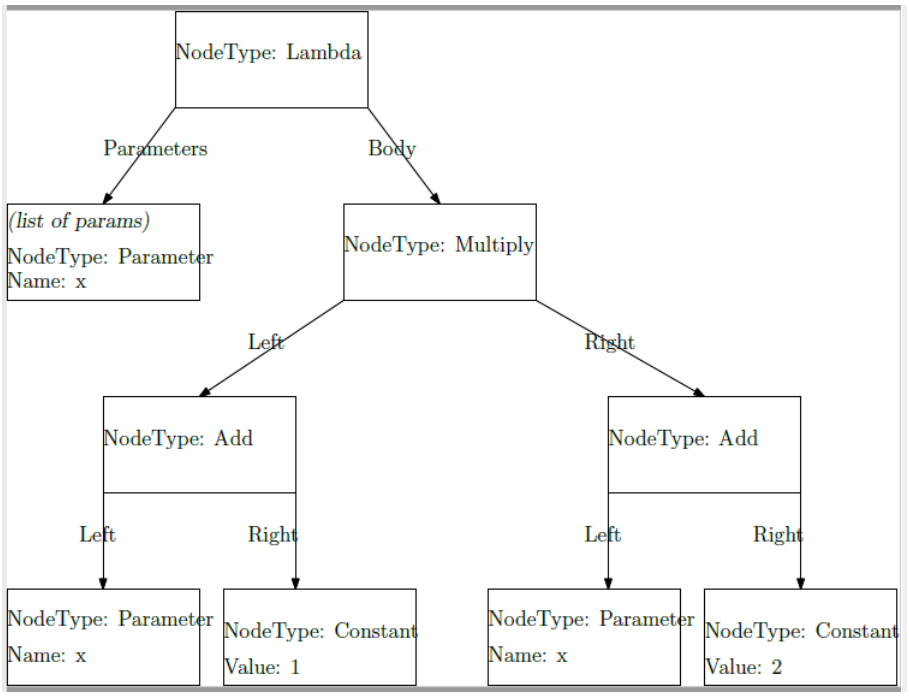
The workaround is to convert Linq expressions trees to an isomorphic third party format, called MetaLinq editable expressions, that allows serialisation, deserialisation, and editing (under some constraints). We use a library called MetaLinq.dll, which we have recompiled as MetaLinqUoA.dll, to match the software that we currently use.

This MetaLinqUoA.dll library will be given to you in the A#6 samples folder, and is also present on the automaker, so you'll be able to use it w/o worries.

An **A#6** scenario (and how to do hints). Assume that, before compiling and running your programs, the automarker (AM) starts with this lambda:

```
Expression<Func<int, int>> lambda1 = x => (x + 1) * (x + 2);
```

It is useful to consider its abstract syntax tree (AST), here annotated with attributes close to those used by the Linq (and MetaLinq) expression trees:



Then, AM will convert this to an isomorphic MetaLinq editable expression tree, that will be serialised and sent to stdin. All this is done by AM, w/o your help.

Your task starts now, by reading the XML sent to your stdin:

- deserialise the XML into a MetaLinq editable expression (see Appendix),
- convert this to a standard Linq expression,
- compile this to method, and
- invoke this with the right number of parameters (you need to find this)
- print the result

Recall that, when invoking this method, the 1st parameter takes value 1, the 2nd value 3, the 3rd value 3. In our scenario, print one line with the number 6, b/c:

```
lambda1 (1) = (1 + 1) * (1 + 2) = 6
```

You could, if you wish, obtain the result directly from parsing and interpreting the XML text, w/o deserialisation etc – there is no penalty for this, but this could be a longer way. Up to you.

The **Appendix** show a (bit simplified) XML serialisation of the MetaLinq editable expression created from our lambda1.

A#6B scenario (and how to do hints). Consider the same lambda as above:

```
Expression<Func<int, int>> lambda1 = x => (x + 1) * (x + 2);
```

You will get the same XML text and, by visiting the highlighted body of the expression tree in postfix order (see 105 or a good basic book on algorithms), you can write the following output line (equivalent reverse Polish):

```
x 1 + x 2 + *
```

You can obtain this postfix notation by other means, but these alternatives may prove longer. There is no penalty though, up to you again.

Testing before submission

Besides the given lambda1, please test your programs extensively, locally, with many other lambda expressions of the expected type:

```
Expression<Func<int, int, … int>>
```

The assignment support folder, **03-A6-Samples**, contains code for both serialisation and deserialisation, so you can serialise yourself as many other test lambdas as necessary for a reasonable test coverage. You can also adapt and reuse the testing harness (batch) given for A#5.

Programs:

(A#5) a C# solution (required).

(A#5b) a C# solution (optionally, bonus).

Essentially, each program must be totally contained in one single file and use only standard libraries extant in the labs, plus MetaLinqUoA.dll

These programs will run on automarker:

- The input xml must be read from stdin, i.e. Console.In in C#.
- The output must be written to stdout, i.e. Console[.Out] in C#.

Submission

Please submit to the automarker (https://www.automarker.cs.auckland.ac.nz/student.php):

APPENDIX

Visualise the actual Linq expression tree of our sample lambda1 - Linqpad dump!

Top, Parameters, and **Body / Left** :

lambda1

Expression<Func<Int32,Int32>>

x => ((x + 1) * (x + 2))

CanReduce	False
Type	typeof(Func<Int32,Int32>)
NodeType	Lambda
Parameters	**IList<ParameterExpression> (1 item)**

CanReduce	Type	NodeType	Name	IsByRef
False	typeof(Int32)	Parameter	x	False

Name	*null*
Body	**SimpleBinaryExpression**

((x + 1) * (x + 2))

CanReduce	False
Right	**SimpleBinaryExpression**

(x + 2)

CanReduce	False
Right	**ConstantExpression**

2

CanReduce	False
Type	typeof(Int32)
NodeType	Constant
Value	2

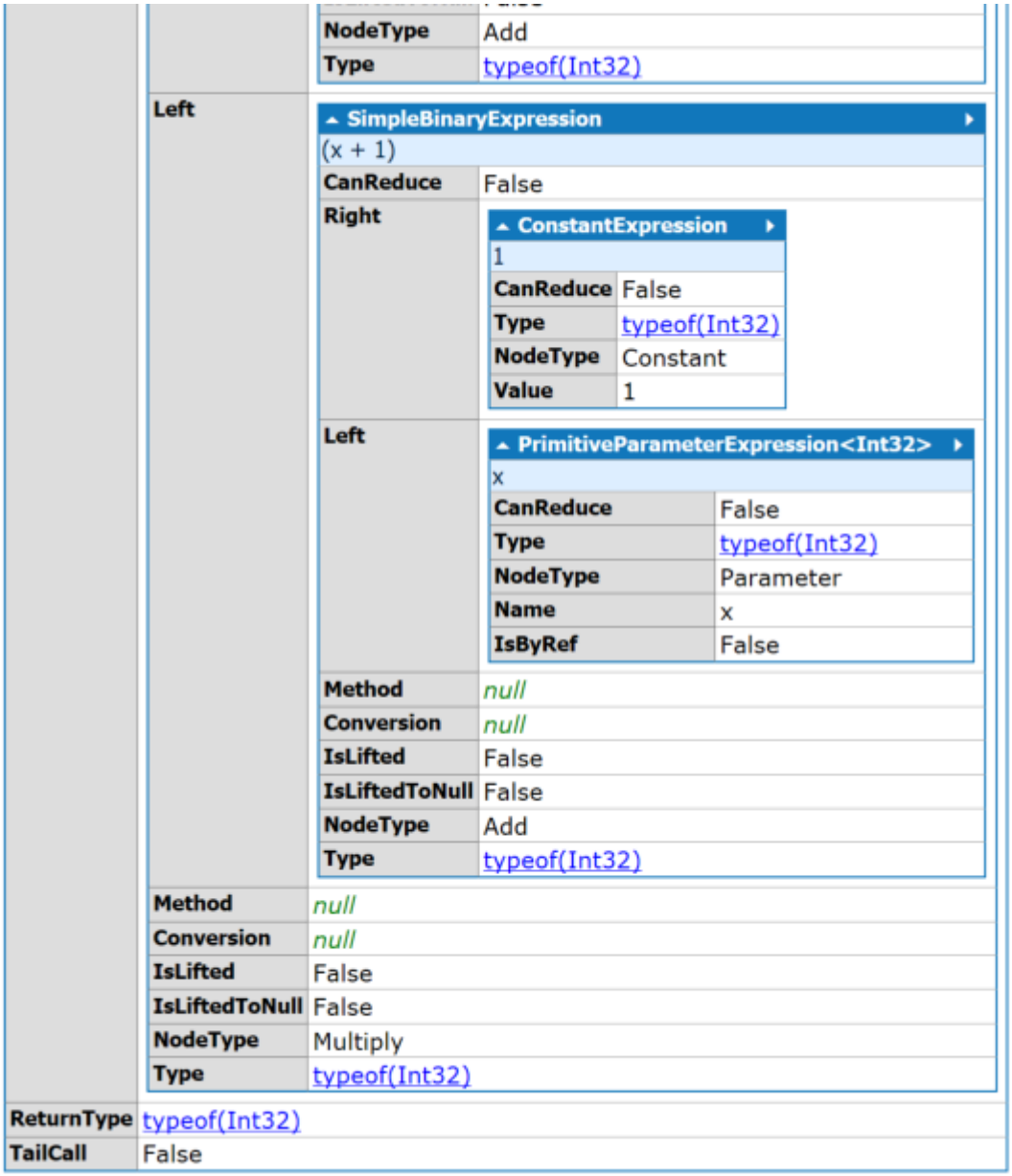
Left	**PrimitiveParameterExpression<Int32>**

x

CanReduce	False
Type	typeof(Int32)
NodeType	Parameter
Name	x
IsByRef	False

Method	*null*
Conversion	*null*
IsLifted	False
IsLiftedToNull	False
NodeType	Add
Type	typeof(Int32)
Left	**SimpleBinaryExpression**

Body / **Right**, bottom :



The next page shows a (bit simplified) version of the **XML serialisation** of the equivalent MetaLinq editable expression.

```
<?xml version="1.0" encoding="utf-16"?>
<EditableLambdaExpression xmlns:xsi="…" xmlns:xsd="…">
  <NodeType>Lambda</NodeType>
  <Parameters>
    <EditableExpression xsi:type="EditableParameterExpression">
      <NodeType>Parameter</NodeType>
      <Name>x</Name>
    </EditableExpression>
    <EditableExpression xsi:type="EditableParameterExpression">
      <NodeType>Parameter</NodeType>
      <Name>y</Name>
    </EditableExpression>
  </Parameters>
  <Body xsi:type="EditableBinaryExpression">
    <NodeType>Multiply</NodeType>
    <Left xsi:type="EditableBinaryExpression">
      <NodeType>Add</NodeType>
      <Left xsi:type="EditableParameterExpression">
        <NodeType>Parameter</NodeType>
        <Name>x</Name>
      </Left>
      <Right xsi:type="EditableConstantExpression">
        <NodeType>Constant</NodeType>
        <Value xsi:type="xsd:int">3</Value>
      </Right>
    </Left>
    <Right xsi:type="EditableBinaryExpression">
      <NodeType>Add</NodeType>
      <Left xsi:type="EditableParameterExpression">
        <NodeType>Parameter</NodeType>
        <Name>y</Name>
      </Left>
      <Right xsi:type="EditableConstantExpression">
        <NodeType>Constant</NodeType>
        <Value xsi:type="xsd:int">7</Value>
      </Right>
    </Right>
  </Body>
</EditableLambdaExpression>
```

Sample serialisation code, using MetaLinq as intermediary:

```
Expression<Func<int, int>> lambda1 = x => (x + 1) * (x + 2);
var exp = lambda1 as Expression
var metexp =
    MetaLinq.EditableExpression.CreateEditableExpression (exp);

var serializer = new XmlSerializer (typeof(EditableLambdaExpression));
var sw = new StringWriter ();
serializer.Serialize (sw, metexp);
var xml = sw.ToString ();
```

Sample deserialisation code, using MetaLinq as intermediary:

```
var sr = new StringReader (xml);
var serializer = new XmlSerializer (typeof(EditableLambdaExpression));
var metexp = serializer.Deserialize (sr) as EditableLambdaExpression;
var exp = metexp.ToExpression () as LambdaExpression;
if (exp is Expression<Func<int, int>>) …
```

The lecture handouts show **how to internally compile and then evaluate** an expression tree.

To **compile with CSC**, include the external MetaLinqUoA.dll, here assumed in the current folder (source file name irrelevant):

```
csc  -r:MetaLinqUoA.dll  jbon007.cs
```

Your sources will probably need to reference these namespaces:

```
System; System.IO;System.Linq;
System.Linq.Expressions; System.Xml.Serialization;
MetaLinq.Expressions;
```