前段时间写了一个词法分析器，最近在这个词法分析器的基础上写了一个表达式分析器，它可以由词法分析的结果构建出一个DLR表达式树，最终可以编译为一个.NET Method.

定义表达式块：

```
using System;
using System.Collections.Generic;

namespace Zxf.ExpressionBuilder
{
    public class ExpressionBlock
    {
        public ExpressionBlock(string blockType)
        {
            BlockType = blockType;
            SubBlocks = new List<ExpressionBlock>();
        }

        public ExpressionBlock(string blockType, LexicalBlock lexicalBlock)
        {
            BlockType = blockType;
            LexicalBlock = lexicalBlock;
            SubBlocks = new List<ExpressionBlock>();
        }

        public String BlockType { get; set; }

        public LexicalBlock LexicalBlock { get; set; }

        public List<ExpressionBlock> SubBlocks { get; set; }
    }
}
```

定义表达式分析的异常类：

```
using System;

namespace Zxf.ExpressionBuilder
{
    public class ExpressionAnalysisException : ApplicationException
    {
        public ExpressionAnalysisException(string message, LexicalBlock lexicalBlock)
            : base(message)
        {
            LexicalBlock = lexicalBlock;
        }

        public ExpressionAnalysisException(string message, ExpressionBlock expressionBlock)
            : base(message)
        {
            ExpressionBlock = expressionBlock;
        }

        public LexicalBlock LexicalBlock { get; set; }

        public ExpressionBlock ExpressionBlock { get; set; }
    }
}
```

用来支持各种数据类型间的数据运算：

```
using System;

namespace Zxf.ExpressionBuilder
```

```
{
    public static class DynamicCalculate
    {
        public static bool GreaterThanOrEqual(dynamic a, dynamic b)
        {
            return GetValue(a, b) >= GetValue(b, a);
        }

        public static bool GreaterThan(dynamic a, dynamic b)
        {
            return GetValue(a, b) > GetValue(b, a);
        }

        public static bool LessThanOrEqual(dynamic a, dynamic b)
        {
            return GetValue(a, b) <= GetValue(b, a);
        }

        public static bool LessThan(dynamic a, dynamic b)
        {
            return GetValue(a, b) < GetValue(b, a);
        }

        public static bool Equal(dynamic a, dynamic b)
        {
            if (a is DBNull && b is DBNull)
            {
                return true;
            }
            else if (a is DBNull || b is DBNull)
            {
                return false;
            }
            else
            {
                return GetValue(a, b) == GetValue(b, a);
            }
        }

        public static bool NotEqual(dynamic a, dynamic b)
        {
            if (a is DBNull && b is DBNull)
            {
                return false;
            }
            else if (a is DBNull || b is DBNull)
            {
                return true;
            }
            else
            {
                return GetValue(a, b) != GetValue(b, a);
            }
        }

        public static bool And(dynamic a, dynamic b)
        {
            return a && b;
        }

        public static bool Or(dynamic a, dynamic b)
        {
            return a || b;
        }
```

```
        public static Object Add(dynamic a, dynamic b)
        {
            return GetValue(a, b) + GetValue(b, a);
        }

        public static Object Subtract(dynamic a, dynamic b)
        {
            return GetValue(a, b) - GetValue(b, a);
        }

        public static Object Multiply(dynamic a, dynamic b)
        {
            return GetValue(a, b) * GetValue(b, a);
        }

        public static Object Divide(dynamic a, dynamic b)
        {
            return GetValue(a, b) / GetValue(b, a);
        }

        public static Object Modulo(dynamic a, dynamic b)
        {
            return GetValue(a, b) % GetValue(b, a);
        }

        private static dynamic GetValue(dynamic o1, dynamic other)
        {
            if (o1.GetType() == other.GetType())
            {
                return o1;
            }
            else
            {
                if (o1 is string || other is string)
                {
                    return Convert.ToString(o1);
                }
                else if (o1 is ValueType && other is ValueType)
                {
                    if (o1 is decimal || other is decimal)
                    {
                        return (decimal)o1;
                    }
                    else
                    {
                        return o1;
                    }
                }
                else
                {
                    return o1;
                }
            }
        }
    }
}
```

表达式分析类：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Runtime.CompilerServices;
```

```
using Microsoft.CSharp.RuntimeBinder;

namespace Zxf.ExpressionBuilder
{
    public static class ExpressionAnalysis
    {
        #region Private Memebers

        private static readonly List<List<String>> s_LeveledOperate = new List<List<String>>();

        #endregion

        #region Static Constructor
        static ExpressionAnalysis()
        {
            s_LeveledOperate.Add(new List<String> { "Viariable", "Paras", "Exp", "StringLiteral", "IntegerLiteral", "NULLLit
            s_LeveledOperate.Add(new List<String> { "And", "Or" });
            s_LeveledOperate.Add(new List<String>
                                    {
                                        "GreaterThanOrEqual",
                                        "LessThanOrEqual",
                                        "GreaterThan",
                                        "LessThan",
                                        "Equal",
                                        "NotEqual"
                                    });
            s_LeveledOperate.Add(new List<String> { "Add", "Subtract" });
            s_LeveledOperate.Add(new List<String> { "Multiply", "Divide", "Modulo" });
            s_LeveledOperate.Add(new List<String> { "Dot", "Function", "Index" });
        }
        #endregion

        #region Public Method
        public static TDelegate Analysis<TDelegate>(List<LexicalBlock> lexicalBlocks, Dictionary<string, Type> parametersTyp
        {
            Dictionary<string, ParameterExpression> parameterExpressions = parametersType.ToDictionary(
                item => item.Key, item => Expression.Parameter(item.Value));

            ExpressionBlock block = AnalysisBlock(BlockAnalysis("Exp", lexicalBlocks));
            Expression buildExpression = Expression.Convert(BuildExpression(block, parameterExpressions),
                                                            typeof(object));
            Expression<TDelegate> dynamicLambdaExp = Expression.Lambda<TDelegate>(buildExpression, parameterExpressions.Valu

            return dynamicLambdaExp.Compile();
        }
        #endregion

        #region Block Process

        private static ExpressionBlock BlockAnalysis(string blockType, List<LexicalBlock> lexicalBlocks)
        {
            var returnBlock = new ExpressionBlock(blockType);

            if (blockType == "Paras")
            {
                int curIndex = 0;
                while (curIndex < lexicalBlocks.Count)
                {
                    int endIndex = GetNotMatchedEndIndex(lexicalBlocks, curIndex, lexicalBlocks.Count);

                    ExpressionBlock paraBlock = BlockAnalysis("Exp", lexicalBlocks.GetRange(curIndex, endIndex - curIndex));
                    returnBlock.SubBlocks.Add(paraBlock);

                    curIndex = endIndex + 1;
                }
```

```
    }
    else
    {
        int curIndex = 0;
        while (curIndex < lexicalBlocks.Count)
        {
            string blockPattern = GetBlockPattern(lexicalBlocks, curIndex, lexicalBlocks.Count);
            if (blockPattern == "Dot|Viariable|OpenParen")
            {
                var funcBlock = new ExpressionBlock("Function") { LexicalBlock = lexicalBlocks[curIndex + 1] };
                returnBlock.SubBlocks.Add(funcBlock);

                int endIndex = GetMatchedEndIndex(lexicalBlocks, curIndex + 2, lexicalBlocks.Count);

                ExpressionBlock parasBlock = BlockAnalysis("Paras", lexicalBlocks.GetRange(curIndex + 3, endIndex -
                returnBlock.SubBlocks.Add(parasBlock);

                curIndex = endIndex + 1;
            }
            else if (blockPattern.StartsWith("OpenBracket"))
            {
                var indexBlock = new ExpressionBlock("Index");
                returnBlock.SubBlocks.Add(indexBlock);

                int endIndex = GetMatchedEndIndex(lexicalBlocks, curIndex, lexicalBlocks.Count);

                ExpressionBlock parasBlock = BlockAnalysis("Paras", lexicalBlocks.GetRange(curIndex + 1, endIndex -
                returnBlock.SubBlocks.Add(parasBlock);

                curIndex = endIndex + 1;
            }
            else if (blockPattern.StartsWith("OpenParen"))
            {
                int endIndex = GetMatchedEndIndex(lexicalBlocks, curIndex, lexicalBlocks.Count);

                ExpressionBlock expBlock = BlockAnalysis("Exp", lexicalBlocks.GetRange(curIndex + 1, endIndex - curI
                returnBlock.SubBlocks.Add(expBlock);

                curIndex = endIndex + 1;
            }
            else
            {
                returnBlock.SubBlocks.Add(new ExpressionBlock(lexicalBlocks[curIndex].BlockType, lexicalBlocks[curIn

                curIndex++;
            }
        }
    }
    return returnBlock;
}

private static String GetBlockPattern(List<LexicalBlock> lexicalBlocks, int start, int end)
{
    string blockPattern = lexicalBlocks[start].BlockType;

    if (start + 1 < end)
    {
        blockPattern += "|" + lexicalBlocks[start + 1].BlockType;
    }

    if (start + 2 < end)
    {
        blockPattern += "|" + lexicalBlocks[start + 2].BlockType;
    }
```

```
    return blockPattern;
}

private static Int32 GetMatchedEndIndex(List<LexicalBlock> lexicalBlocks, int start, int end)
{
    var stack = new Stack<string>();
    for (int i = start; i < end; i++)
    {
        if (lexicalBlocks[i].Text == "("
            || lexicalBlocks[i].Text == "[")
        {
            stack.Push(lexicalBlocks[i].Text);
        }
        else if (lexicalBlocks[i].Text == ")"
                 || lexicalBlocks[i].Text == "]")
        {
            string startFlag = lexicalBlocks[i].Text == ")" ? "(" : "[";
            if (stack.Peek() == startFlag)
            {
                stack.Pop();
            }
            else
            {
                throw new ExpressionAnalysisException("Missing ')' or ']'", lexicalBlocks[start]);
            }
        }

        if (stack.Count == 0)
        {
            return i;
        }
    }
    throw new ExpressionAnalysisException("Missing ')' or ']'", lexicalBlocks[start]);
}

private static Int32 GetNotMatchedEndIndex(List<LexicalBlock> lexicalBlocks, int start, int end)
{
    int curIndex = start;
    while (curIndex < end)
    {
        if (lexicalBlocks[curIndex].Text == "("
            || lexicalBlocks[curIndex].Text == "[")
        {
            curIndex = GetMatchedEndIndex(lexicalBlocks, curIndex, lexicalBlocks.Count) + 1;
        }
        else if (lexicalBlocks[curIndex].Text == ",")
        {
            return curIndex;
        }
        curIndex++;
    }
    return curIndex;
}

private static ExpressionBlock AnalysisBlock(ExpressionBlock block)
{
    //计算分区块
    for (int i = 0; i < block.SubBlocks.Count; i++)
    {
        ExpressionBlock curBlock = block.SubBlocks[i];
        if (curBlock.BlockType == "Paras"
            || curBlock.BlockType == "Exp"
            )
        {
            block.SubBlocks[i] = AnalysisBlock(curBlock);
```

```
    }
}

//计算表达式块
if (block.BlockType != "Paras")
{
    while (block.SubBlocks.Count > 1)
    {
        var blockStack = new Stack<ExpressionBlock>();
        for (int i = 0; i < block.SubBlocks.Count; i++)
        {
            ExpressionBlock curBlock = block.SubBlocks[i];

            if (blockStack.Count == 0)
            {
                blockStack.Push(curBlock);
            }
            else
            {
                int curLevel = GetOperateLevel(curBlock);
                int topLevel = GetOperateLevel(blockStack.Peek());

                if (curLevel > topLevel)
                {
                    ExpressionBlock topBlock = blockStack.Pop();
                    int topTopLevel = GetOperateLevel(blockStack.Count == 0 ? null : blockStack.Peek());

                    if (curLevel > topTopLevel)
                    {
                        curBlock.SubBlocks.Add(topBlock);
                        blockStack.Push(curBlock);
                    }
                    else if (curLevel == topTopLevel)
                    {
                        ExpressionBlock topTopBlock = blockStack.Pop();
                        topTopBlock.SubBlocks.Add(topBlock);

                        var expBlock = new ExpressionBlock("Exp");
                        expBlock.SubBlocks.Add(topTopBlock);

                        curBlock.SubBlocks.Add(expBlock);
                        blockStack.Push(curBlock);
                    }
                    else
                    {
                        ExpressionBlock topTopBlock = blockStack.Pop();
                        topTopBlock.SubBlocks.Add(topBlock);

                        var expBlock = new ExpressionBlock("Exp");
                        expBlock.SubBlocks.Add(topTopBlock);

                        if (blockStack.Count == 0)
                        {
                            curBlock.SubBlocks.Add(expBlock);
                            blockStack.Push(curBlock);
                        }
                        else
                        {
                            blockStack.Push(expBlock);

                            blockStack.Push(curBlock);
                        }
                    }
                }
                else if (curLevel < topLevel)
```

```
                    {
                        if (block.SubBlocks.Count == i + 1)
                        {
                            if (blockStack.Peek().SubBlocks.Count == 1)
                            {
                                ExpressionBlock topBlock = blockStack.Pop();
                                topBlock.SubBlocks.Add(curBlock);

                                var expBlock = new ExpressionBlock("Exp");
                                expBlock.SubBlocks.Add(topBlock);

                                blockStack.Push(expBlock);
                            }
                            else
                            {
                                blockStack.Push(curBlock);
                            }
                        }
                        else
                        {
                            blockStack.Push(curBlock);
                        }
                    }
                    else
                    {
                        throw new ExpressionAnalysisException("Not supported LexicalBlock", curBlock.LexicalBlock);
                    }
                }
            }

            block.SubBlocks.Clear();
            while (blockStack.Count > 0)
            {
                block.SubBlocks.Add(blockStack.Pop());
            }
            block.SubBlocks.Reverse();
        }
    }

    return block;
}

private static Int32 GetOperateLevel(ExpressionBlock block)
{
    if (block == null) return -1;
    for (int i = 0; i < s_LeveledOperate.Count; i++)
    {
        if (s_LeveledOperate[i].Contains(block.BlockType))
        {
            return i;
        }
    }
    throw new ExpressionAnalysisException("Not supported LexicalBlock", block.LexicalBlock);
}

#endregion

#region Build Expression

private static Expression BuildExpression(ExpressionBlock block, Dictionary<string, ParameterExpression> parameterEx
{
    switch (block.BlockType)
    {
        case "Function":
            return BuildFunctionExpression(block, parameterExpressions);
```

```
            case "Index":
                return BuildIndexExpression(block, parameterExpressions);
            case "Exp":
                return BuildExpression(block.SubBlocks[0], parameterExpressions);
            case "CharLiteral":
            case "NULLLiteral":
            case "StringLiteral":
            case "IntegerLiteral":
                return BuildConstExpression(block);
            case "Dot":
                return BuildDotExpression(block, parameterExpressions);
            case "GreaterThanOrEqual":
            case "GreaterThan":
            case "LessThanOrEqual":
            case "LessThan":
            case "Equal":
            case "NotEqual":
            case "And":
            case "Or":
                return BuildLogicExpression(block, parameterExpressions);
            case "Add":
            case "Subtract":
            case "Multiply":
            case "Divide":
            case "Modulo":
                return BuildAriguExpression(block, parameterExpressions);
            default:
                return null;
        }
    }

    private static Expression BuildFunctionExpression(ExpressionBlock block,
                                                      Dictionary<string, ParameterExpression> parameterExpressions)
    {
        if (block.SubBlocks.Count != 2)
        {
            throw new ExpressionAnalysisException("Can not build expression", block);
        }

        if (block.SubBlocks[0].BlockType != "Viariable"
            && block.SubBlocks[0].BlockType != "Exp")
        {
            throw new ExpressionAnalysisException("Can not build expression", block);
        }

        if (block.SubBlocks[1].BlockType != "Paras")
        {
            throw new ExpressionAnalysisException("Can not build expression", block);
        }

        var CSharpArgus = new List<CSharpArgumentInfo> { CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null) };

        var ExpreArgus = new List<Expression>();
        if (block.SubBlocks[0].BlockType == "Viariable")
        {
            if (!parameterExpressions.ContainsKey(block.SubBlocks[0].LexicalBlock.Text))
            {
                throw new ExpressionAnalysisException("Can not build expression", block);
            }
            ExpreArgus.Add(parameterExpressions[block.SubBlocks[0].LexicalBlock.Text]);
        }
        else
        {
            ExpreArgus.Add(BuildExpression(block.SubBlocks[0], parameterExpressions));
        }
```

```
    foreach (ExpressionBlock t in block.SubBlocks[1].SubBlocks)
    {
        CSharpArgus.Add(CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null));
        ExpreArgus.Add(BuildExpression(t, parameterExpressions));
    }

    CallSiteBinder binderInvokeMember = Binder.InvokeMember(CSharpBinderFlags.None, block.LexicalBlock.Text,
                                                            null, typeof(Object), CSharpArgus.ToArray());
    return Expression.Dynamic(binderInvokeMember, typeof(object), ExpreArgus.ToArray());
}

private static Expression BuildIndexExpression(ExpressionBlock block, Dictionary<string, ParameterExpression> parame
{
    if (block.SubBlocks.Count != 2)
    {
        throw new ExpressionAnalysisException("Can not build expression", block);
    }

    if (block.SubBlocks[0].BlockType != "Viariable"
        && block.SubBlocks[0].BlockType != "Exp")
    {
        throw new ExpressionAnalysisException("Can not build expression", block);
    }

    if (block.SubBlocks[1].BlockType != "Paras")
    {
        throw new ExpressionAnalysisException("Can not build expression", block);
    }

    var CSharpArgus = new List<CSharpArgumentInfo> { CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null) }

    var ExpreArgus = new List<Expression>();
    if (block.SubBlocks[0].BlockType == "Viariable")
    {
        if (!parameterExpressions.ContainsKey(block.SubBlocks[0].LexicalBlock.Text))
        {
            throw new ExpressionAnalysisException("Can not build expression", block);
        }
        ExpreArgus.Add(parameterExpressions[block.SubBlocks[0].LexicalBlock.Text]);
    }
    else
    {
        ExpreArgus.Add(BuildExpression(block.SubBlocks[0], parameterExpressions));
    }

    foreach (ExpressionBlock t in block.SubBlocks[1].SubBlocks)
    {
        CSharpArgus.Add(CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null));
        ExpreArgus.Add(BuildExpression(t, parameterExpressions));
    }

    CallSiteBinder binderGetIndex = Binder.GetIndex(CSharpBinderFlags.None, typeof(Object), CSharpArgus.ToArray());
    return Expression.Dynamic(binderGetIndex, typeof(object), ExpreArgus.ToArray());
}

private static Expression BuildAriguExpression(ExpressionBlock block, Dictionary<string, ParameterExpression> parame
{
    if (block.SubBlocks.Count != 2)
    {
        throw new ExpressionAnalysisException("Can not build expression", block);
    }

    if (block.SubBlocks[0].BlockType != "Block"
        && block.SubBlocks[0].BlockType != "Exp"
```

```
            && block.SubBlocks[0].BlockType != "CharLiteral"
            && block.SubBlocks[0].BlockType != "NULLLiteral"
            && block.SubBlocks[0].BlockType != "StringLiteral"
            && block.SubBlocks[0].BlockType != "IntegerLiteral")
        {
            throw new ExpressionAnalysisException("Can not build expression", block);
        }

        if (block.SubBlocks[1].BlockType != "Block"
            && block.SubBlocks[1].BlockType != "Exp"
            && block.SubBlocks[1].BlockType != "CharLiteral"
            && block.SubBlocks[1].BlockType != "NULLLiteral"
            && block.SubBlocks[1].BlockType != "StringLiteral"
            && block.SubBlocks[1].BlockType != "IntegerLiteral")
        {
            throw new ExpressionAnalysisException("Can not build expression", block);
        }

        Expression left = BuildExpression(block.SubBlocks[0], parameterExpressions);
        Expression right = BuildExpression(block.SubBlocks[1], parameterExpressions);
        if (left.Type != typeof(Object))
        {
            left = Expression.Convert(left, typeof(object));
        }
        if (right.Type != typeof(Object))
        {
            right = Expression.Convert(right, typeof(object));
        }

        switch (block.BlockType)
        {
            case "Add":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("Add"), new[] { left, right });
            case "Subtract":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("Subtract"), new[] { left, right });
            case "Multiply":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("Multiply"), new[] { left, right });
            case "Divide":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("Divide"), new[] { left, right });
            case "Modulo":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("Modulo"), new[] { left, right });
            default:
                return null;
        }
    }

    private static Expression BuildLogicExpression(ExpressionBlock block,
                                                   Dictionary<string, ParameterExpression> parameterExpressions)
    {
        if (block.SubBlocks.Count != 2)
        {
            throw new ExpressionAnalysisException("Can not build expression", block);
        }

        if (block.SubBlocks[0].BlockType != "Block"
            && block.SubBlocks[0].BlockType != "Exp"
            && block.SubBlocks[0].BlockType != "CharLiteral"
            && block.SubBlocks[0].BlockType != "StringLiteral"
            && block.SubBlocks[0].BlockType != "NULLLiteral"
            && block.SubBlocks[0].BlockType != "IntegerLiteral")
        {
            throw new ExpressionAnalysisException("Can not build expression", block);
        }

        if (block.SubBlocks[1].BlockType != "Block"
```

```
            && block.SubBlocks[1].BlockType != "Exp"
            && block.SubBlocks[1].BlockType != "CharLiteral"
            && block.SubBlocks[1].BlockType != "NULLLiteral"
            && block.SubBlocks[1].BlockType != "StringLiteral"
            && block.SubBlocks[1].BlockType != "IntegerLiteral")
        {
            throw new ExpressionAnalysisException("Can not build expression", block);
        }

        Expression left = BuildExpression(block.SubBlocks[0], parameterExpressions);
        Expression right = BuildExpression(block.SubBlocks[1], parameterExpressions);
        if (left.Type != typeof(Object))
        {
            left = Expression.Convert(left, typeof(object));
        }
        if (right.Type != typeof(Object))
        {
            right = Expression.Convert(right, typeof(object));
        }

        switch (block.BlockType)
        {
            case "GreaterThanOrEqual":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("GreaterThanOrEqual"), new[] { left, rig
            case "GreaterThan":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("GreaterThan"), new[] { left, right });
            case "LessThanOrEqual":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("LessThanOrEqual"), new[] { left, right
            case "LessThan":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("LessThan"), new[] { left, right });
            case "Equal":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("Equal"), new[] { left, right });
            case "NotEqual":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("NotEqual"), new[] { left, right });
            case "And":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("And"), new[] { left, right });
            case "Or":
                return Expression.Call(null, typeof(DynamicCalculate).GetMethod("Or"), new[] { left, right });
            default:
                return null;
        }
    }

    private static Expression BuildConstExpression(ExpressionBlock block)
    {
        switch (block.BlockType)
        {
            case "CharLiteral":
                return Expression.Constant(block.LexicalBlock.Text[1], typeof(char));
            case "NULLLiteral":
                return Expression.Constant(DBNull.Value, typeof(DBNull));
            case "StringLiteral":
                return Expression.Constant(block.LexicalBlock.Text.Substring(1, block.LexicalBlock.Text.Length - 2), typ
            case "IntegerLiteral":
                {
                    if (block.LexicalBlock.Text.IndexOf('.') != -1)
                    {
                        Single singleResult;
                        if (Single.TryParse(block.LexicalBlock.Text, out singleResult))
                        {
                            return Expression.Constant(Convert.ToSingle(block.LexicalBlock.Text), typeof(Single));
                        }

                        Double doubleResult;
                        if (Double.TryParse(block.LexicalBlock.Text, out doubleResult))
```

```
                    {
                        return Expression.Constant(Convert.ToDouble(block.LexicalBlock.Text), typeof(Double));
                    }

                    Decimal decimalResult;
                    if (Decimal.TryParse(block.LexicalBlock.Text, out decimalResult))
                    {
                        return Expression.Constant(Convert.ToDecimal(block.LexicalBlock.Text), typeof(Decimal));
                    }

                    throw new Exception("Not support number: " + block.LexicalBlock.Text);
                }
                else
                {
                    Int32 int32Result;
                    if (Int32.TryParse(block.LexicalBlock.Text, out int32Result))
                    {
                        return Expression.Constant(Convert.ToInt32(block.LexicalBlock.Text), typeof(Int32));
                    }

                    Int64 int64Result;
                    if (Int64.TryParse(block.LexicalBlock.Text, out int64Result))
                    {
                        return Expression.Constant(Convert.ToInt64(block.LexicalBlock.Text), typeof(Int64));
                    }

                    throw new Exception("Not support number: " + block.LexicalBlock.Text);
                }
            }
          default:
            return null;
    }
}

private static Expression BuildDotExpression(ExpressionBlock block, Dictionary<string, ParameterExpression> paramete
{
    if (block.SubBlocks.Count != 2)
    {
        throw new ExpressionAnalysisException("Can not build expression", block);
    }

    if (block.SubBlocks[0].BlockType != "Viariable"
        && block.SubBlocks[0].BlockType != "Exp")
    {
        throw new ExpressionAnalysisException("Can not build expression", block);
    }

    if (block.SubBlocks[1].BlockType != "Viariable")
    {
        throw new ExpressionAnalysisException("Can not build expression", block);
    }

    if (block.SubBlocks[0].BlockType == "Viariable")
    {
        if (!parameterExpressions.ContainsKey(block.SubBlocks[0].LexicalBlock.Text))
        {
            throw new ExpressionAnalysisException("Can not build expression", block);
        }

        CallSiteBinder binderGetProperty = Binder.GetMember(CSharpBinderFlags.None,
                                                            block.SubBlocks[1].LexicalBlock.Text,
                                                            typeof(Object),
                                                            new[]{CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.
        return Expression.Dynamic(binderGetProperty, typeof(object), parameterExpressions[block.SubBlocks[0].Lexical
    }
```

```
            else
            {
                Expression exp = BuildExpression(block.SubBlocks[0], parameterExpressions);

                CallSiteBinder binderGetProperty = Binder.GetMember(CSharpBinderFlags.None,
                                                                    block.SubBlocks[1].LexicalBlock.Text,
                                                                    typeof(Object),
                                                                    new[]{CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.
                return Expression.Dynamic(binderGetProperty, typeof(object), exp);
            }
        }

        #endregion
    }
}
```