UniversalExpressionParser Documentation

Release 1.0.1

Artak Hakobyan

May 14, 2022

CONTENTS

CHAPTER

ONE

SUMMARY

UniversalExpressionParser is a library for parsing expressions like the one demonstrated below into expression items for functions, literals, operators, etc.

```
var z = x1*y1+x2*y2;

var matrixMultiplicationResult = [[x1_1, x1_2, x1_3], [x2_1, x2_2, x2_3]]*[[y1_1, x1_2],␣
↪[y2_1, x2_2], [y3_1, x3_2]];

println(matrixMultiplicationResult);

[NotNull] [PublicName("Calculate")]
public F1(x, y) : int =>
{

    /*This demos multiline
    comments that can be placed anywhere*/
    ++y /*another multiline comment*/;
    return 1.3EXP-2.7+ ++x+y*z; // Line comments.
}

public abstract class ::metadata{description: "Demo prefix"} ::types[T1, T2, T3] Animal
   where T1: IType1 where T2: T1, IType2 whereend
   where T3: IType3 whereend
{
    public abstract Move() : void;
}

public class Dog : (Animal)
{
    public override Move() : void => println("Jump");
}
```

- Below is the demo code used to parse this expression:

```
using TextParser;
using UniversalExpressionParser.DemoExpressionLanguageProviders;

namespace UniversalExpressionParser.Tests.Demos
{
    public class SummaryDemo
    {
```

(continues on next page)

(continued from previous page)

```
        private readonly IExpressionParser _expressionParser;
        private readonly IExpressionLanguageProvider _nonVerboseLanguageProvider = new␣
↪NonVerboseCaseSensitiveExpressionLanguageProvider();
        private readonly IExpressionLanguageProvider _verboseLanguageProvider = new␣
↪VerboseCaseInsensitiveExpressionLanguageProvider();
        private readonly IParseExpressionOptions _parseExpressionOptions = new␣
↪ParseExpressionOptions();

        public SummaryDemo()
        {
            IExpressionLanguageProviderCache expressionLanguageProviderCache =
                new ExpressionLanguageProviderCache(new␣
↪DefaultExpressionLanguageProviderValidator());

            _expressionParser = new ExpressionParser(new TextSymbolsParserFactory(),␣
↪expressionLanguageProviderCache);

            expressionLanguageProviderCache.RegisterExpressionLanguageProvider(_
↪nonVerboseLanguageProvider);
            expressionLanguageProviderCache.RegisterExpressionLanguageProvider(_
↪verboseLanguageProvider);
        }

        public IParseExpressionResult ParseNonVerboseExpression(string expression)
        {
            /*
            The same instance _expressionParser of UniversalExpressionParser.
↪IExpressionParser can be used
            to parse multiple expressions using different instances of␣
↪UniversalExpressionParser.IExpressionLanguageProvider
            Example:

            var parsedExpression1 = _expressionParser.ParseExpression(_
↪nonVerboseLanguageProvider.LanguageName, "var x=2*y; f1() {++x;} f1();");
            var parsedExpression2 = _expressionParser.ParseExpression(_
↪verboseLanguageProvider.LanguageName, "var x=2*y; f1() BEGIN ++x;END f1();");
            */
            return _expressionParser.ParseExpression(_nonVerboseLanguageProvider.
↪LanguageName, expression, _parseExpressionOptions);
        }
    }
}
```

- Expression is parsed to an instance of **UniversalExpressionParser.IParseExpressionResult** by calling the method **IParseExpressionResult ParseExpression(string expressionLanguageProviderName, string expressionText, IParseExpressionOptions parseExpressionOptions)** in **UniversalExpressionParser.IExpressionParser**.
- The interface **UniversalExpressionParser.IParseExpressionResult** (i.e., result of parsing the expression) has a property **UniversalExpressionParser.ExpressionItems.IRooxExpressionItem RootExpressionItem { get; }** that stores the root expression item of a tree structure of parsed expression items.
- The code that evaluates the parsed expression can use the following properties in **UniversalExpressionParser.ExpressionItems.IRootExpressionItem** to iterate through all parsed expression items:

  - **IEnumerable<IExpressionItemBase> AllItems { get; }**
  - **IReadOnlyList<IExpressionItemBase> Prefixes { get; }**
  - **IReadOnlyList<IKeywordExpressionItem> AppliedKeywords { get; }**
  - **IReadOnlyList<IExpressionItemBase> RegularItems { get; }**
  - **IReadOnlyList<IExpressionItemBase> Children { get; }**
  - **IReadOnlyList<IExpressionItemBase> Postfixes { get; }**
  - **IReadOnlyList<IExpressionItemBase> Prefixes { get; }**

- All expressions are parsed either to expressions items of type **UniversalExpressionParser.ExpressionItems.IExpressionItemBase** or one of its subclasses for simple expressions or to expressions items of type **UniversalExpressionParser.ExpressionItems.IComplexExpressionItem** (which is a sub-interface of UniversalExpressionParser.ExpressionItems.IExpressionItemBase) or one of its subclasses for expression items that consists of other expression items.
- Some examples simple expression items are: **UniversalExpressionParser.ExpressionItems.ICommaExpressionItem** for commas, **UniversalExpressionParser.ExpressionItems.IOpeningBraceExpressionItem** and **UniversalExpressionParser.ExpressionItems.IClosingBraceExpressionItem** for opening and closing braces "(" and ")"
- Some complex expression items are: **UniversalExpressionParser.ExpressionItems.IBracesExpressionItem** for functions like "f1 (x1, x2)", **UniversalExpressionParser.ExpressionItems.IOperatorExpressionItem** for operators like the binary operator with operands "f1(x)" and "y" in "f1(x) + y".
- All expressions are currently parsed to one of the following expression items (or intances of other sub-interfaces of these interfaces) in namespaces **UniversalExpressionParser.ExpressionItems** and **UniversalExpressionParser.ExpressionItems.Custom**: **ILiteralExpressionItem**, **ILiteralNameExpressionItem**, **IConstantTextExpressionItem**, **IConstantTextValueExpressionItem**, **INumericExpressionItem**, **INumericExpressionValueItem**, **IBracesExpressionItem**, **IOpeningBraceExpressionItem**, **IClosingBraceExpressionItem**, **ICommaExpressionItem**, **ICodeBlockExpressionItem**, **ICustomExpressionItem**, **IKeywordExpressionItem**, **ICodeBlockStartMarkerExpressionItem**, **ICodeBlockEndMarkerExpressionItem**, **ISeparatorCharacterExpressionItem**, **IOperatorExpressionItem**, **IOperatorInfoExpressionItem**, **IKeywordExpressionItem**, **UniversalExpressionParser.ExpressionItems.Custom.ICustomExpressionItem**, **IRootExpressionItem**, **IComplexExpressionItem**, **ITextExpressionItem**, **IExpressionItemBase**. The state of this expression items can be analyzed when evaluating the parsed expression.
- The format of valid expressions is defined by properties and methods in interface **UniversalExpressionParser.IExpressionLanguageProvider**. The expression language name **UniversalExpressionParser.IExpressionLanguageProvider.LanguageName** of some instance **UniversalExpressionParser.IExpressionLanguageProvider** is passed as a parameter to method **ParseExpression(...)** in **UniversalExpressionParser.IExpressionParser**, as demonstrated in example above. Most of the properties and methods of this interface are demonstrated in examples in sections below.
- The default abstract implementation of this interface in this package is **UniversalExpressionParser.ExpressionLanguageProviderBase**. In most cases, this abstract class can be extended and abstract methods and properties can be implemented, rather than providing a brand new implementation of **UniversalExpressionParser.IExpressionLanguageProvider**.
- The test project **UniversalExpressionParser.Tests** in git repository has a number of tests for testing successful parsing, as well as tests for testing expressions that result in errors (see section **Error Reporting** below). These tests generate random expressions as well as generate randomly configured instances of **UniversalExpressionParser.IExpressionLanguageProvider** to validate parsing of thousands of all possible languages and expressions (see the test classes **UniversalExpressionParser.Tests.SuccessfulParseTests.ExpressionParserSuccessfulTests** and **UniversalExpressionParser.Tests.ExpressionParseErrorTests.ExpressionParseErrorTests**).

- The demo expressions and tests used to parse the demo expressions in this documentation are in folder **Demos** in test project **UniversalExpressionParser.Tests**. This documentation uses implementations of **UniversalExpressionParser.IExpressionLanguageProvider** in project **UniversalExpressionParser.DemoExpressionLanguageProviders** in **git** repository.
- The parsed expressions in this documentation (i.e., instances of **UniversalExpressionParser.ExpressionItems.IParseExpressionResult**) are visualized into xml texts, that contain values of most properties of the parsed expression. However, to make the files shorter, the visualized xml files do not include all the property values.

CHAPTER

TWO

LITERALS

- Literals are names that can be used either alone (say in operators) or can be part of more complex expressions. For example literals can precede opening square or round braces (e.g., **f1** in **f1(x)**, or **m1** in **m1[1, 2]**), or code blocks (e.g., **Dog** in expression **public class Dog {}**).
- Literals are parsed into expression items of type **UniversalExpressionParser.ExpressionItems.ILiteralExpressionItem** objects.
- If literal precedes round or square braces, it will be stored in value of property **ILiteralExpressionItem Name { get; }** of **UniversalExpressionParser.ExpressionItems.IBracesExpressionItem**.
- If literal precedes a code block staring marker (e.g., **Dog** in expression **public class Dog {}**), then the code block is added to the list **UniversalExpressionParser.ExpressionItems.IComplexExpressionItem.Postfixes** in expression item for the literal (see section **Postfixes** for more details on postfixes).
- Literals are texts that cannot have space and can only contain characters validated by method **UniversalExpressionParser.IExpressionLanguageProvider.IsValidLiteralCharacter(char character, int positionInLiteral, ITextSymbolsParserState textSymbolsParserState)** in interface **UniversalExpressionParser.IExpressionLanguageProvider**. In other words, a literal can contain any character (including numeric or operator characters, a dot '.', '_', etc.), that is considered a valid literal character by method **IExpressionLanguageProvider.IsValidLiteralCharacter(...)**.

2.1 Examples of literals

```
1  // In example below _x, f$, x1, x2, m1, and x3, Dog, Color, string, println, and Dog.
   ↪Color are literals.
2  var _x = f$(x1, x2) + m1[1, x3];
3
4  public class Dog
5  {
6      public Color : string => "brown";
7  }
8
9  // println is a literal and it is part of "println(Dog.Color)" braces expression item.␣
   ↪In this example,
10 // println will be parsed to an expression item UniversalExpressionParser.
   ↪ExpressionItems.INamedComplexExpressionItem
```

(continues on next page)

(continued from previous page)

```
// and will be the value of property NamedExpressionItem in UniversalExpressionParser.
↪ExpressionItems.IBracesExpressionItem
println(Dog.Color);
```

CHAPTER

THREE

FUNCTIONS AND BRACES

- Functions are round or square braces preceded with a literal (e.g., **F1(x)**, **F1(x) {}**, **m1[i,j]**, **m1[i,j]{}**). Functions are parsed to expression items of type **UniversalExpressionParser.ExpressionItems.IBracesExpressionItem** with the value of property **ILiteralExpressionItem NameLiteral { get; }** equal to a literal that precedes the braces.
- Braces are a pair of round or square braces ((e.g., **(x)**, **(x) {}**, **[i,j]**, **[i,j]{}**)). Braces are parsed to expression items of type **UniversalExpressionParser.ExpressionItems.IBracesExpressionItem** with the value of property **ILiteralExpressionItem NameLiteral { get; }** equal to null.

3.1 Examples of functions

```
// The statements below do not make much sense.
// They just demonstrate different ways the square and round braces can be used
// in expressions.
var x = x1+f1(x2, x3+x4*5+x[1])+
         matrix1[[y1+3, f1(x4)], x3, f2(x3, m2[x+5])];

f1(x, y) => x+y;

f2[x, y]
{
   // Normally matrixes do not have bodies like functions doo. This is just to demo that
   // the parser allows this.
}
```

3.2 Examples of braces

```
// The statements below do not make much sense.
// They just demonstrate different ways the square and round braces can be used
// in expressions.
var x = ((x1, x2, x3), [x4, x5+1, x6], y);
x += (x2, x4) + 2*[x3, x4];
```

CHAPTER

FOUR

OPERATORS

- *Example of defining operators in an implementation of **UniversalExpressionParser.IExpressionLanguageProvider***
- *Example of considering priorities when parsing operators*
- *Example of using braces to change order of application of operators*
- *Example of operators with multiple parts in operator names*
- *Example of two operators (e.g., postfix operators, then a binary operator) used next to each other without spaces in between*
- *Example of unary prefix operator used to implement "return" statement*

- Operators that the valid expressions can use are defined in property **System.Collections.Generic.IReadOnlyList<UniversalExpressionParser.IOperatorInfo> Operators { get; }** in interface **UniversalExpressionParser.IExpressionLanguageProvider** (an instance of this interface is passed to the parser).
- The interface **UniversalExpressionParser.IOperatorInfo** has properties for operator name (i.e., a collection of texts that operator consists of, such as ["IS", "NOT", "NUL"] or ["+="]), priority, unique Id, operator type (i.e., binary, unary prefix or unary postfix).
- Two different operators can have similar names, as long as they have different operator. For example "++" can be used both as unary prefix as well as unary postfix operator.

4.1 Example of defining operators in an implementation of UniversalExpressionParser.IExpressionLanguageProvider

```
public class TestExpressionLanguageProviderBase : ExpressionLanguageProviderBase
{
    //...
    // Some other method and property implementations here
    // ...
    public override IReadOnlyList<IOperatorInfo> Operators { get; } = new IOperatorInfo[]
    {
            // The third parameter (e.g., 0) is the priority.
            new OperatorInfo(1, new [] {"!"}, OperatorType.PrefixUnaryOperator, 0),
            new OperatorInfo(2, new [] {"IS", "NOT", "NULL"}, OperatorType.
↪PostfixUnaryOperator, 0),
```

(continues on next page)

(continued from previous page)

```

            new OperatorInfo(3, new [] {"*"}, OperatorType.BinaryOperator, 10),
            new OperatorInfo(4, new [] {"/"}, OperatorType.BinaryOperator, 10),

            new OperatorInfo(5, new [] {"+"}, OperatorType.BinaryOperator, 30),
        new OperatorInfo(6, new [] {"-"}, OperatorType.BinaryOperator, 30),
    }
}
```

- Operator expression (e.g., "a * b + c * d") is parsed to an expression item of type **UniversalExpressionParser.ExpressionItems.IOperatorExpressionItem** (a subclass of **UniversalExpressionParser.ExpressionItems.IComplexExpressionItem**).

For example the expression "a * b + c * d", will be parsed to an expression logically similar to "*(+(a, b), +(x,d))*". *This is so since the binary operator "+" has lower priority (the value of **IOperatorInfo.Priority** is larger)*, than the binary operator "*".

In other words this expression will be parsed to a binary operator expression item for "+" (i.e., an instance of **IOperatorExpressionItem**) with Operand1 and Operand2 also being binary operator expression items of type **UniversalExpressionParser.ExpressionItems.IOperatorExpressionItem** for expression items "a * b" and "c * d".

4.2 Example of considering priorities when parsing operators

```
// The binary operator + has priority 30 and * has priority 20. Therefore,
// in expression below,  * is applied first and + is applied next.
// The following expression is parsed to an expression equivalent to
// "=(var y, +(x1, *(f1(x2, +(x3, 1)), x4)))"
var y = x1 + f1(x2,x3+1)*x4;
```

4.3 Example of using braces to change order of application of operators

```
// Without the braces, the expression below would be equivalent to x1+(x2*x3)-x4.
var y1 = [x1+x2]*(x3-x4);
```

4.4 Example of operators with multiple parts in operator names

```
// The expression below is similar to
// z = !((x1 IS NOT NULL) && (x2 IS NULL);
z = !(x1 IS NOT NULL && x2 IS NULL);
```

4.5 Example of two operators (e.g., postfix operators, then a binary operator) used next to each other without spaces in between

```
// The spaces between two ++ operators, and + was omitted intentionally to show that the␣
↪parser will parse the expression
// correctly even without the space.
// The expression below is similar to  println(((x1++)++)+x2). To avoid confusion, in␣
↪some cases it is better to use braces.
println(x1+++++x2)
```

4.6 Example of unary prefix operator used to implement "return" statement

```
// return has priority int.MaxValue which is greater then any other operator priority,␣
↪therefore
// the expression below is equivalent to "return (x+(2.5*x))";
return x+2.5*y;

// another example within function body
f1(x:int, y:int) : bool
{
   // return has priority int.MaxValue which is greater then any other operator priority,
↪ therefore
   // the expression below is equivalent to "return (x+(2.5*x))";
   return f(x)+y > 10;
}
```

CHAPTER

FIVE

NUMERIC VALUES

- **Universal Expression Parser** parses expression items that have numeric format to expression items of type **UniversalExpressionParser.ExpressionItems.INumericExpressionItem**. The format of expression items that will be parsed to **UniversalExpressionParser.ExpressionItems.INumericExpressionItem** is determined by property **IReadOnlyList<NumericTypeDescriptor> NumericTypeDescriptors { get; }** in interface **UniversalExpressionParser.IExpressionLanguageProvider**, an instance of which is passed to the parser.
- The parser scans the regular expressions in list in property **IReadOnlyList<string> RegularExpressions { get; }** in type **NumericTypeDescriptor** for each provided instance of **UniversalExpressionParser.NumericTypeDescriptor** to try to parse the expression to numeric expression item of type **UniversalExpressionParser.ExpressionItems.INumericExpressionItem**.
- The abstract class **UniversalExpressionParser.ExpressionLanguageProviderBase** that can be used as a base class for implementations of **UniversalExpressionParser.IExpressionLanguageProvider** in most cases, implements the property **NumericTypeDescriptors** as a virtual property. The implementation of property **NumericTypeDescriptors** in **UniversalExpressionParser.ExpressionLanguageProviderBase** is demonstrated below, and it can be overridden to provide different format for numeric values:

Note: The regular expressions used in implementation of property **NumericTypeDescriptors** should always start with '**^**' and should never end with '**$**'.

```
public virtual IReadOnlyList<NumericTypeDescriptor> NumericTypeDescriptors { get; } =␣
↪new List<NumericTypeDescriptor>
{
    new NumericTypeDescriptor(KnownNumericTypeDescriptorIds.ExponentFormatValueId,
    new[] { @"^(\d+\.\d+|\d+\.|\.\d+|\d+)(EXP|exp|E|e)[+|-]?(\d+\.\d+|\d+\.|\.\d+|\d+)"}
↪),

    new NumericTypeDescriptor(KnownNumericTypeDescriptorIds.FloatingPointValueId,
    new[] { @"^(\d+\.\d+|\d+\.|\.\d+)"}),

    new NumericTypeDescriptor(KnownNumericTypeDescriptorIds.IntegerValueId, new[] { @"^\
↪d+" })
}
```

- The first regular expression that matches the expression, is stored in properties **SucceededNumericTypeDescriptor** of type **UniversalExpressionParser.NumericTypeDescriptor** and **IndexOfSucceededRegularExpression** in parsed expression item of type **UniversalExpressionParser.ExpressionItems.INumericExpressionItem**.
- The numeric value is stored as text in property **INameExpressionItem Value** in text format. Therefore, there is no limit on numeric value digits.

- The expression evaluator that uses the **Universal Expression Parser** can convert the textual value in property **Value** of type **INameExpressionItem** in **UniversalExpressionParser.ExpressionItems.INumericExpressionItem** to a value of numeric type (int, long, double, etc.).
- Examples of numeric value expression items are demonstrated below:

```
// By default exponent notation can be used.
println(-0.5e-3+.2exp3.4+3.E2.7+2.1EXP.3);
println(.5e15*x);

// Numeric values can have no limitations on the number of digits. The value is stored as text in
// UniversalExpressionParser.ExpressionItems.INumericExpressionItem.
// The text can be validated farther and converted to numeric values by the expression evaluator that
// uses the parser.
var x = 2.3*x+123456789123456789123456789123456789;
```

CHAPTER

SIX

TEXTS

The interface **UniversalExpressionParser.IExpressionLanguageProvider** has a property **IReadOnlyList<char> ConstantTextStartEndMarkerCharacters { get; }** that are used by **Universal Expression Parser** to parse expression items that start or end with some character in **ConstantTextStartEndMarkerCharacters** to parse the expression item to **UniversalExpressionParser.ExpressionItems.IConstantTextExpressionItem**.

- The default implementation **UniversalExpressionParser.ExpressionLanguageProviderBase** of **UniversalExpressionParser.IExpressionLanguageProvider** has the following characters in property **UniversalExpressionParser.IExpressionLanguageProvider.ConstantTextStartEndMarkerCharacters** and the examples below will use these text marker characters.
- If an expression starts with some character listed in property **UniversalExpressionParser.IExpressionLanguageProvider.ConstantTextStartEndMarkerCharacters**, then the text expression should end with the same character. In other words, if text starts with character **‘** it should end with **‘**. Similarly, if text starts with character **“** it should end with **“**.
- The text in expression parsed to **UniversalExpressionParser.ExpressionItems.IConstantTextExpressionItem** can contain any of the text marker characters in **UniversalExpressionParser.IExpressionLanguageProvider.ConstantTextStartEndMarkerCharacters**. if the text contains a text marker character that marks the start of the text expression, it should be typed twice as displayed in examples below:
- Texts can span multiple lines (see the example below).

```
// Example of texts using text markers ',", and ` in IExpressionLanguageProvider.
↪ConstantTextStartEndMarkerCharacters property.
// The text marker that starts the specific text expression can be used in text as well␣
↪when it is
// typed twice (e.g., '', "", ``, etc).

// Example of using all text markers together in operators
x = "We can use this text expression marker "" if we type it twice. However, other text␣
↪marker chars do not need to be typed twice such as ' and `."
    +'We can use this text expression marker '' if we type it twice. However, other text␣
↪marker chars do not need to be typed twice such as " and `.'
    +`We can use this text expression marker `` if we type it twice. However, other text␣
↪marker chars do not need to be typed twice such as " and '.`;

println("This is a text that spans
 multiple
 lines.
" + x + ' Some other text.');
```

CHAPTER

SEVEN

CODE SEPARATORS AND CODE BLOCKS

- *More complex examples of code blocks and code separators*

- If the value of property **char ExpressionSeparatorCharacter { get; }** in **UniversalExpressionParser.IExpressionLanguageProvider** is not equal to character '0', multiple expressions can be used in a single expression.

For example if the value of property **ExpressionSeparatorCharacter** is ';' the expression "var x=f1(y);println(x)" will be parsed to a list of two expression items for "x=f1(y)" and "println(x)". Otherwise, the parser will report an error for this expression (see section on **Error Reporting** for more details on errors).

- If both values of properties **char CodeBlockStartMarker { get; }** and **string CodeBlockEndMarker { get; }** in **UniversalExpressionParser.IExpressionLanguageProvider** are not equal to character '0', code block expressions can be used to combine multiple expressions into code block expression items of type **UniversalExpressionParser.ExpressionItems.ICodeBlockExpressionItem**.
- For example if the values of properties **CodeBlockStartMarker** and **CodeBlockEndMarker** are '{' and '}', the expression below will be parsed to two code block expressions of type **UniversalExpressionParser.ExpressionItems.ICodeBlockExpressionItem**. Otherwise, the parser will report errors.

```
{
    var x = y^2;
    println(x);
}

{
    f1(x1, x2);
    println(x) // No need for ';' after the last expression
}
```

7.1 More complex examples of code blocks and code separators

```
var y = x1 + 2.5 * x2;

f1()
{
    // Code block used as function body (see examples in Postfixes folder)
}

```

(continues on next page)

(continued from previous page)

```
var z = e ^ 2.3;
{
    var x = 5 * y;
    println("x=" + x);

    {
        var y1 = 10 * x;
        println(getExp(y1));
    }

    {
        var y2 = 20 * x;
        println(getExp(y2));
    }

    getExp(x) : double
    {
        // another code block used as function body (see examples in Postfixes folder)
        return e^x;
    }
}

f2()
{
    // Another code block used as function body (see examples in Postfixes folder)
}

{
    // Another code block
}

public class Dog
{
    public Bark()
    {
        // Note, code separator ';' is not necessary, if the expression is followed by␣
↪code block end marker '}'.
        println("bark")
    }
}
```

CHAPTER

EIGHT

KEYWORDS

- *Examples of keywords*

- Keywords are special names (e.g., **var**, **public**, **class**, **where**) that can be specified in property **IReadOnlyList<ILanguageKeywordInfo> Keywords { get; }** in interface **UniversalExpressionParser.IExpressionLanguageProvider**, as shown in example below.

Note: Keywords are supported only if the value of property **SupportsKeywords** in **UniversalExpressionParser.IExpressionLanguageProvider** is true.

```
public class DemoExpressionLanguageProvider : IExpressionLanguageProvider
{
    ...
    public override IReadOnlyList<ILanguageKeywordInfo> Keywords { get; } = new []
    {
        new UniversalExpressionParser.UniversalExpressionParser(1, "where"),
        new UniversalExpressionParser.UniversalExpressionParser(2, "var"),
        ...
    };
}
```

- Keywords are parsed to expression items of type **UniversalExpressionParser.ExpressionItems.IKeywordExpressionItem**.
- Keywords have the following two applications.

1) One or more keywords can be placed in front of any literal (e.g., variable name), round or square braces expression, function or matrix expression, a code block. In this type of usage of keywords the parser parses the keywords and adds the list of parsed keyword expression items (i.e., list of **UniversalExpressionParser.ExpressionItems.IKeywordExpressionItem** objects) to list in property **IReadOnlyList<IKeywordExpressionItem> AppliedKeywords { get; }** in **UniversalExpressionParser.ExpressionItems.IComplexExpressionItem** for the expression item that follows the list of keywords.
2) Custom expression parser evaluates the list of parsed keywords to determine if the expression that follows the keywords should be parsed to a custom expression item. See section **Custom Expression Item Parsers** for more details on custom expression parsers.

8.1 Examples of keywords

```
// Keywords "public" and "class" will be added to list in property "AppliedKeywords" in class
// "UniversalExpressionParser.ExpressionItems.Custom.IComplexExpressionItem" for the parsed expression "Dog".
public class Dog;

// Keywords "public" and "static will be added to list in property "AppliedKeywords" in class
// "UniversalExpressionParser.ExpressionItems.Custom.IComplexExpressionItem" for the parsed expression "F1()".
public static F1();

// Keywords "public" and "static" will be added to list in property "AppliedKeywords" in class
// "UniversalExpressionParser.ExpressionItems.Custom.IComplexExpressionItem" for the parsed expression "F1()".
public static F1() {return 1; }

// Keyword "::codeMarker" will be added to list in property "AppliedKeywords" in class
// "UniversalExpressionParser.ExpressionItems.Custom.IComplexExpressionItem" for the parsed expression "(x1, x2)".
::codeMarker (x1, x2);

// Keyword "::codeMarker" will be added to list in property "AppliedKeywords" in class
// "UniversalExpressionParser.ExpressionItems.Custom.IComplexExpressionItem" for the parsed expression "m1[2, x1]".
::codeMarker m1[2, x1];

// Keyword "::codeMarker" will be added to list in property "AppliedKeywords" in class
// "UniversalExpressionParser.ExpressionItems.Custom.IComplexExpressionItem" for the parsed expression "[x1, x2]".
::codeMarker[x1, x2];

// Keyword "static" will be added to list in property "AppliedKeywords" in class
// "UniversalExpressionParser.ExpressionItems.Custom.IComplexExpressionItem" for the code block parsed expression "{}".
static
{
    var x;
}

// Keyword "::pragma" will be used by custom expression parser
// "UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.PragmaCustomExpressionItemParser" to
// parse expressions "::pragma x2" and "::pragma x3" to custom expression items of type
// "UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.PragmaCustomExpressionItem".
var y = x1 +::pragma x2+3*::pragma x3 +y;
```

CHAPTER

NINE

PREFIXES

Prefixes are one or more expression items that precede some other expression item, and are added to the list in property **Prefixes** in interface **UniversalExpressionParser.ExpressionItems.IComplexExpressionItem** for the expression item that follows the list of prefix expression items.

Note: Prefixes are supported only if the value of property **SupportsPrefixes** in interface **UniversalExpressionParser.IExpressionLanguageProvider** is true.

Currently **Universal Expression Parser** supports two types of prefixes:

9.1 1) Nameless brace as prefixes

Square or round braces expressions items without names (i.e. expression items that are parsed to **UniversalExpressionParser.ExpressionItems.IBracesExpressionItem** with property **NamedExpressionItem** equal to **null**) that precede another expression item (e.g., another braces expression, a literal, a code block, text expression item, numeric value expression item, etc) are parsed as prefixes and are added to expression item they precede.

- In the example below the braces expression items parsed from "[NotNull, ItemNotNull]" and "(Attribute("MarkedFunction"))" will be added as prefixes to expression item parsed from "F1(x, x2)".

```
[NotNull, ItemNotNull](Attribute("MarkedFunction")) F1(x, x2)
{
    // This code block will be added to expression item parsed from F1(x:T1, y:T2, z:
↪T3) as a postfix.
    retuens [x1, x2, x3];
}
```

9.2 2) Custom expressions as prefixes

If custom expression items of type **UniversalExpressionParser.ExpressionItems.Custom.ICustomExpressionItem** with property **CustomExpressionItemCategory** equal to **UniversalExpressionParser.ExpressionItems.Custom.CustomExpressionItemCategory.Prefix** are added as prefixes to expression item they precede.

Note: List of prefixes can include both nameless brace expression items as well as custom expression items, placed in any order.

- In the example below, the expression items "::types[T1,T2]" and "::types[T3]" are parsed to custom expression items of type **UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.IGenericTypeDataExpressionItem**, and are added as prefixes to braces expression item parsed from "F1(x:T1, y:T2, z: T3)".

Note: For more details on custom expression items see section **Custom Expression Item Parsers**.

```
::types[T1,T2] ::types[T3] F1(x:T1, y:T2, z: T3)
{
    // This code block will be added to expression item parsed from F1(x:T1, y:T2, z:
↪T3) as a postfix.
}
```

Note: The list of prefixes can include both types of prefixes at the same time (i.e., braces and custom expression items).

- Here is an example of prefixes used to model c# like attributes for classes and methods:

```
// [TestFixture] and [Attribute("IntegrationTest")] are added as prefixes to literal
↪MyTests.
[TestFixture]
[Attribute("IntegrationTest")]
// public and class are added as keywords to MyTests
public class MyTests
{
    // Brace expression items [SetupMyTests], [Attribute("This is a demo of multiple
↪prefixes")]
    // and custom expression item starting with ::metadata and ending with } are added
↪as prefixes to
    // expression SetupMyTests()
    [TestSetup]
    [Attribute("This is a demo of multiple prefixes")]
    ::metadata {
        Description: "Demo of custom expression item parsed to
                      UniversalExpressionParser.DemoExpressionLanguageProviders.
↪CustomExpressions.IMetadataCustomExpressionItem
                      used in prefixes list of expression parsed from 'SetupMyTests()'
↪";
        SomeMetadata: 1
    }
```

(continues on next page)

(continued from previous page)

```
    // public and static are added as keywords to expression SetupMyTests().
    public static SetupMyTests() : void
    {
        // Do some test setup here
    }
}
```

Note: The list of prefixes can include both types of prefixes at the same time (i.e., braces and custom expression items).

- Below is an example of using prefixes with different expression item types:

```
// Prefixes added to a literal "x".
[NotNull] [Attribute("Marker")] x;

// Prefixes added to named round braces. [NotNull] [Attribute("Marker")] will be added
// to prefixes in braces expression item parsed from "f1(x1)"
[NotNull] [Attribute("Marker")] f1(x1);

// Prefixes added to unnamed round braces. [NotNull] [Attribute("Marker")] will be added
// to prefixes in braces expression item parsed from "(x1)"
[NotNull] [Attribute("Marker")] (x1);

// Prefixes added to named square braces. [NotNull] [Attribute("Marker")] will be added
// to prefixes in named braces expression item parsed from "m1[x1]"
[NotNull] [Attribute("Marker")] m1[x1];

// Prefixes added to unnamed square braces. [NotNull] [Attribute("Marker")] will be added
// to prefixes in braces expression item parsed from "[x1]".
[NotNull] [Attribute("Marker")] [x1];

// Prefixes added to code block.
// Custom prefix expression item "::types[T1,T2]" will be added to list of prefixes in␣
↪code block expression item
// parsed from "{var i = 12;}".
// Note, if we replace "::types[T1,T2]" to unnamed braces, then the unnamed braces will␣
↪be used as a postfix for
// code block.
::types[T1,T2] {var i = 12;};

// Prefixes added to custom expression item parsed from "::pragma x".
// [Attribute("Marker")] will be added to list of prefixes in custom expression item
// parsed from "::pragma x".
[Attribute("Marker")] ::pragma x;

// Prefixes added text expression item.
// [Attribute("Marker")] will be added to list of prefixes in text expression item
// parsed from "Some text".
[Attribute("Marker")] "Some text";

// Prefixes added to numeric value item.
// [Attribute("Marker")] will be added to list of prefixes in numeric value expression␣
↪item
```

(continues on next page)

(continued from previous page)

```
// parsed from "0.5e-3.4".
[Attribute("Marker")] 0.5e-3.4;
```

CHAPTER

TEN

POSTFIXES

Postfixes are one or more expression items that are placed after some other expression item, and are added to the list in property **Postfixes** in interface **UniversalExpressionParser.ExpressionItems.IComplexExpressionItem** for the expression item that the postfixes are placed after.

Currently **Universal Expression Parser** supports two types of postfixes:

10.1 1) Code block expression items

Code block expression items that are parsed to **UniversalExpressionParser.ExpressionItems.ICodeBlockExpressionItem** that succeed another expression item are added as postfixes to the expression item they succeed.

Note: The following are expression types that can have postfixes: Literals, such a **x1** or **Dog**, braces expression items, such as **f(x1)**, **(y)**, **m1[x1]**, **[x2]**, or custom expression items for which property **CustomExpressionItemCategory** in interface **UniversalExpressionParser.ExpressionItems.Custom.ICustomExpressionItem** is equal to **UniversalExpressionParser.ExpressionItems.Custom.CustomExpressionItemCategory.Regular**.

- In the example below the code block expression item of type **UniversalExpressionParser.ExpressionItems.ICodeBlockExpressionItem** parsed from expression that starts with '{' and ends with '}'" will be added to the list **Postfixes** in **UniversalExpressionParser.ExpressionItems.IComplexExpressionItem** for the literal expression item parsed from expression **Dog**.

```
public class Dog
{
   // This code block will be added as a postfix to literal expression item parsed from
↪"Dog"
}
```

10.2 2) Custom postfix expression items

Custom expression items of type **UniversalExpressionParser.ExpressionItems.Custom.ICustomExpressionItem** with property **CustomExpressionItemCategory** equal to **UniversalExpressionParser.ExpressionItems.Custom.CustomExpressionItemCategory.Postfix** that succeed another expression item are added as postfixes to the expression item they succeed.

- In the example below the two custom expression items of type **UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.IWhereCustomExpressionItem** parsed from expressions that start with "where" and end with "whereend"" as well as the code block will be added as postfixes to literal expression item parsed from "Dog".

Note: For more details on custom expression items see section **Custom Expression Item Parsers**.

```
::types[T1,T2, T3] F1(x:T1, y:T2, z: T3)
// The where below will be added as a postfix to expression item parsed from "F1(x:T1,␣
↪y:T2, z: T3)
where T1:int where T2:double whereend
// The where below will be added as a postfix to expression item parsed from "F1(x:T1,␣
↪y:T2, z: T3)
where T3:T1  whereend
{
    // This code block will be added as a postfix to expression item parsed from
↪"F1(x:T1, y:T2, z: T3).
}
```

Note: The list of postfixes can include both types of postfixes at the same time (i.e., custom expression items as well as a code block postfix).

- Example of a code block postfix used to model function body:

```
// More complicated cases
// In the example below the parser will apply operator ':' to 'f2(x1:int, x2:int)' and
↪'int'
// and will add the code block after 'int' as a postfix to 'int'.
// The evaluator that processes the parsed expression can do farther transformation so␣
↪that the code block is assigned to
// some new property in some wrapper for an expression for 'f2(x1:int, x2:int)', so that␣
↪the code block belongs to the function, rather than
// to the returned type 'int' of function f2.
f2(x1:int, x2:int) : int
{
   f3() : int
   {
       var result = x1+x2;
           println("result='"+result+"'");
           return result;
   }

   return f3();
```

(continues on next page)

(continued from previous page)

```
}

var myFunc = f2(x1:int, x2:int) =>
{
    println(exp ^ (x1 + x2));
}
```

- Example of code block postfix used to model class definition:

```
// In the example below the parser will apply operator ':' to literal 'Dog' (with keywords public and class) and
// braces '(Anymal, IDog)' and will add the code block after '(Anymal, IDog)' as a postfix to '(Anymal, IDog)'.
// The evaluator that processes the parsed expression can do farther transformation so that the code block is assigned to
// some new property in some wrapper for an expression for 'Dog', so that the code block belongs to the 'Dog' class, rather than
// to the braces for public classes in '(Anymal, IDog)'.
public class Dog : (Anymal, IDog)
{
    public Bark() : void
    {
        println("Bark.");
    }
}
```

- Below are some more examples of postfixes with different expression items:

```
f1(x1)
{
    // Code block added to postfixes list for braces expression "f1(x1)"
    return x2*y1;
}

m1[x2]
{
    // Code block added to postfixes list for braces expression "m1[x2]"
    x:2*3
}

(x3)
{
    // Code block added to postfixes list for braces expression "(x3)"
    return x3*2;
}

[x4]
{
    // Code block added to postfixes list for braces expression "[x4]"
    x4:2*3
}

```

(continues on next page)

(continued from previous page)

```
class Dog
{
    // Code block added to postfixes list for literal expression "Dog"
}

::pragma x
{
    // Code block added to custom expression item IPragmaCustomExpressionItem parsed
↪from "::pragma x"
}
```

CHAPTER

ELEVEN

CUSTOM EXPRESSION ITEM PARSERS

Custom expression parsers allow to plugin into parsing process and provide special parsing of some portion of the parsed expression.

The expression parser (i.e., **UniversalExpressionParser.IExpressionParser**) iteratively parses keywords (see the section above on keywords), before parsing any other symbols.

Then the expression parser loops through all the custom expression parsers of type UniversalExpressionParser.ExpressionItems
ICustomExpressionItem ICustomExpressionItemParser.TryParseCustomExpressionItem(IParseExpressionItemContext context, IReadOnlyList<IExpressionItemBase> parsedPrefixExpressionItems, IReadOnlyList<IKeywordExpressionItem> keywordExpressionItems).

If method call **TryParseCustomExpressionItem(...)** returns non-null value of type **UniversalExpressionParser.ExpressionItems.Custom.ICustomExpressionItem**, the parser uses the parsed custom expression item.

Otherwise, if **TryParseCustomExpressionItem(...)** returns null, the parser tries to parse a non custom expression item at current position (i.e., operators, a literal, function, code block, etc.).

Interface ICustomExpressionItem has a property UniversalExpressionParser.ExpressionItems.Custom.CustomExpressionItem

- a prefix for subsequently parsed regular expression (i.e., literal, function, braces, etc.).
- should be treated as regular expression (which can be part of operators, function parameter, etc.).
- or should be used as a postfix for the previously parsed expression item.

In the example below the parser parses "::pragma x" to a regular custom expression item of type **UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.PragmaCustomExpressionItem** (i.e., the value of **CustomExpressionItemCategory property** in **UniversalExpressionParser.ExpressionItems.Custom.ICustomExpressionItem** is equal to **UniversalExpressionParser.ExpressionItems.Custom.CustomExpressionItemCategory.Regular**). As a result, the expression "::pragma x+y;" below is logically similar to "(::pragma x)+y;"

In a similar manner the expression "::types[T1, T2]" is parsed to a prefix custom expression item of type **UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.GenericTypesCustomExpressionItem** by custom expression item parser **UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.GenericTyp**

The custom expression item parsed from "::types[T1, T2]" is added as a prefix to an expression item parsed from **F1(x:T1, y:T2)**.

Also, the expression "where T1:int where T2:double whereend" is parsed to postfix custom expression item of type **UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.WhereCustomExpressionItem** by custom expression parser **UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.WhereCustomE**

The parser adds the parsed custom expression as a postfix to the preceding regular expression item parsed from text "F1(x:T1, y:T2)".

In this example, the code block after "whereend" (the expression "{...}") is parsed as a postfix expression item of type **UniversalExpressionParser.ExpressionItems.ICodeBlockExpressionItem** and is added as a postfix to regular expression item parsed from "F1(x:T1, y:T2)" as well, since the parser adds all the prefixes/postfixes to regular expression item it finds after/before the prefixes/postfixes.

```
::pragma x+y;
::types[T1,T2] F1(x:T1, y:T2) where T1:int where T2:double whereend
{
   // This code block will be added as a postfix to expression item parsed from "F1(x:T1,
↪ y:T2)".
}
```

- This is another example demonstrating that the parsed expression can have multiple prefix and postfix custom expressions items applied to the same regular expression item parsed from "F1(x:T1, y:T2, z:T3)".

```
// The expression below ("::metadata {...}") is parsed to a prefix custom expression␣
↪item and added to list of prefixes of regular
// expression item parsed from F1(x:T1, y:T2, z:T3)
::metadata {description: "F1 demoes regular function expression item to which multiple␣
↪prefix and postfix custom expression items are added."}

// ::types[T1,T2] is also parsed to a prefix custom expression item and added to list of␣
↪prefixes of regular
// expression item parsed from F1(x:T1, y:T2, z:T3)
::types[T1,T2]
F1(x:T1, y:T2, z:T3)

// The postfix custom expression item parsed from "where T1:int where T2:double whereend
↪" is added to list of postfixes of regular expression
// parsed from "F1(x:T1, y:T2, z:T3)".
where T1:int,class where T2:double whereend

// The postfix custom expression item parsed from "where T3 : T1 whereend " is also␣
↪added to list of postfixes of regular expression
// parsed from "F1(x:T1, y:T2, z:T3)".
where T3 : T1 whereend
{
   // This code block will be added as a postfix to expression item parsed from "F1(x:T1,
↪ y:T2, z:T3)".
}
```

11.1 Implementing Custom Expression Parsers

For examples of custom expression item parsers look at some examples in demo project **UniversalExpressionParser.DemoExpressionLanguageProviders**.

The following demo implementations of **UniversalExpressionParser.ExpressionItems.Custom.ICustomExpressionItemParserByKe** might be useful when implementing custom expression parses:

- UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.WhereCustomExpressionItemParserBase
- UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.PragmaCustomExpressionItemParser
- UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions.MetadataCustomExpressionItemParser

Also, these custom expression parser implementations demonstrate how to use the helper class **UniversalExpressionParser.IParseExpressionItemContext** that is passed as a parameter to method **DoParseCustomExpressionItem(IParseExpressionItemContext context,...)** in **UniversalExpressionParser.ExpressionItems.Custom.CustomExpressionItemParserByKeywordId** to parse the text at current position, as well as how to report errors, if any.

- To add a new custom expression parser, one needs to implement an interface **UniversalExpressionParser.ExpressionItems.Custom.ICustomExpressionItemParser** and make sure the property **CustomExpressionItemParsers** in interface **UniversalExpressionParser.IExpressionLanguageProvider** includes an instance of the implemented parser class.
- In most cases the default implementation **UniversalExpressionParser.ExpressionItems.Custom.AggregateCustomExpressionI** of **UniversalExpressionParser.ExpressionItems.Custom.ICustomExpressionItemParser** can be used to initialize the list of all custom expression parers that will be used by **Universal Expression Parser**.

UniversalExpressionParser.ExpressionItems.Custom.AggregateCustomExpressionItemParser has a dependency on **IEnumerable<ICustomExpressionItemParserByKeywordId>** (injected into constructor).

- Using a single instance of **AggregateCustomExpressionItemParser** in property **CustomExpressionItemParsers** in interface **UniversalExpressionParser.IExpressionLanguageProvider** instead of multiple custom expression parsers in this property improves the performance.

AggregateCustomExpressionItemParser keeps internally a mapping from keyword Id to all the instances of **UniversalExpressionParser.ExpressionItems.Custom.ICustomExpressionItemParserByKeywordId** injected in constructor. When the parser executes the method **TryParseCustomExpressionItem(...,IReadOnlyList<IKeywordExpressionItem> parsedKeywordExpressionItems,...)** in interface **UniversalExpressionParser.ExpressionItems.Custom**, the custom expression item parser of type **AggregateCustomExpressionItemParser** evaluates the last keyword in list in parameter **parsedKeywordExpressionItems** to retrieve all the parsers mapped to this keyword Id, to try to parse a custom expression item using only those custom expression item parsers.

- Below is some of the code from classes **AggregateCustomExpressionItemParser** and **ICustomExpressionItemParserByKeywordId**.

```
namespace UniversalExpressionParser.ExpressionItems.Custom;

public class AggregateCustomExpressionItemParser : ICustomExpressionItemParser
{
    public AggregateCustomExpressionItemParser(
        IEnumerable<ICustomExpressionItemParserByKeywordId> customExpressionItemParsers)
    {
        ...
    }

```

(continues on next page)

(continued from previous page)

```
    public ICustomExpressionItem TryParseCustomExpressionItem(IParseExpressionItemContext context,
            IReadOnlyList<IExpressionItemBase> parsedPrefixExpressionItems,
            IReadOnlyList<IKeywordExpressionItem> parsedKeywordExpressionItems)
    {
        ...
    }
}

public interface ICustomExpressionItemParserByKeywordId
{
    long ParsedKeywordId { get; }

    ICustomExpressionItem TryParseCustomExpressionItem(IParseExpressionItemContext context,
            IReadOnlyList<IExpressionItemBase> parsedPrefixExpressionItems,
            IReadOnlyList<IKeywordExpressionItem> parsedKeywordExpressionItemsWithoutLastKeyword,
            IKeywordExpressionItem lastKeywordExpressionItem);
}
```

- Here is the code from demo custom expression item parser **PragmaCustomExpressionItemParser**

```
using System.Collections.Generic;
using UniversalExpressionParser.ExpressionItems;
using UniversalExpressionParser.ExpressionItems.Custom;

namespace UniversalExpressionParser.DemoExpressionLanguageProviders.CustomExpressions
{
    /// <summary>
    ///  Example: ::pragma x
    /// </summary>
    public class PragmaCustomExpressionItemParser : CustomExpressionItemParserByKeywordId
    {
        public PragmaCustomExpressionItemParser() : base(KeywordIds.Pragma)
        {
        }

        /// <inheritdoc />
        protected override ICustomExpressionItem DoParseCustomExpressionItem(IParseExpressionItemContext context, IReadOnlyList<IExpressionItemBase> parsedPrefixExpressionItems,
                                                                             IReadOnlyList<IKeywordExpressionItem> parsedKeywordExpressionItemsWithoutLastKeyword,

                                                                             IKeywordExpressionItem pragmaKeywordExpressionItem)
        {
            var pragmaKeywordInfo = pragmaKeywordExpressionItem.LanguageKeywordInfo;

            var textSymbolsParser = context.TextSymbolsParser;

            if (!context.SkipSpacesAndComments() || !context.TryParseSymbol(out var literalExpressionItem))
```

(continues on next page)

(continued from previous page)

```
            {
                if (!context.ParseErrorData.HasCriticalErrors)
                {
                    // Example: print("Is in debug mode=" + ::pragma IsDebugMode)
                    context.AddParseErrorItem(new ParseErrorItem(textSymbolsParser.PositionInText,
                        () => $"Pragma keyword '{pragmaKeywordInfo.Keyword}' should be followed with pragma symbol. Example: println(\"Is in debug mode = \" + {pragmaKeywordInfo.Keyword} IsDebug);",
                        CustomExpressionParseErrorCodes.PragmaKeywordShouldBeFollowedByValidSymbol));
                }

                return null;
            }

            return new PragmaCustomExpressionItem(parsedPrefixExpressionItems, parsedKeywordExpressionItemsWithoutLastKeyword,
                pragmaKeywordExpressionItem,
                new NameExpressionItem(literalExpressionItem, textSymbolsParser.PositionInText - literalExpressionItem.Length));
        }
    }
}
```

CHAPTER

TWELVE

COMMENTS

The interface **UniversalExpressionParser.IExpressionLanguageProvider** has properties **string LineCommentMarker { get; }**, **string MultilineCommentStartMarker { get; }**, and **string MultilineCommentEndMarker { get; }** for specifying comment markers.

If the values of these properties are not null, line and code block comments can be used.

The abstract implementation **UniversalExpressionParser.ExpressionLanguageProviderBase** of **UniversalExpressionParser.IExpressionLanguageProvider** overrides these properties to return "//", "/*", *and* "*/" (the values of these properties can be overridden in subclasses).

The on commented out code data is stored in property **IReadOnlyList<UniversalExpressionParser.ICommentedTextData> SortedCommentedTextData { get; }** in **UniversalExpressionParser.IParsedExpressionResult**, an instance of which is returned by the call to method **UniversalExpressionParser.IExpressionParser.ParseExpression(...)**.

- Below are some examples of line and code block comments:

```
// Line comment
var x = 5*y; // another line comments

println(x +/*Code block
comments
can span multiple lines and can be placed anywhere.
*/y+10*z);

/*
Another code block comments
var y=5*x;
var z = 3*y;
*/
```

- Below is the definition of interface **UniversalExpressionParser.ICommentedTextData** that stores data on comments.

```
// Copyright (c) UniversalExpressionParser Project. All rights reserved.
// Licensed under the MIT License. See LICENSE in the solution root for license␣
↪information.

using UniversalExpressionParser.ExpressionItems;

namespace UniversalExpressionParser
{
    /// <summary>
    /// Info on commented out code block.
```

(continues on next page)

(continued from previous page)

```
    /// </summary>
    public interface ICommentedTextData: ITextItem
    {
        /// <summary>
        /// If true, the comment is a line comment. Otherwise, it is a block comment.
        /// </summary>
        bool IsLineComment { get; }
    }
}
```

CHAPTER

THIRTEEN

ERROR REPORTING

Parse error data is stored in property **UniversalExpressionParser.IParseErrorData ParseErrorData { get; }**. The class **UniversalExpressionParser.IParseErrorData** has a property **IReadOnlyList<UniversalExpressionParser.IParseErrorItem> AllParseErrorItems { get; }** that stores data on all parse errors.

The extensions class **UniversalExpressionParser.ParseExpressionResultExtensionMethods** has number of helper methods, among which is **string GetErrorTextWithContextualInformation(this IParsedExpressionResult parsedExpressionResult, int parsedTextStartPosition, int parsedTextEnd, int maxNumberOfCharactersToShowBeforeOrAfterErrorPosition = 50)** for returning a string with error details and contextual data (i.e., text before and after the position where error happened, along with arrow pointing to the error).

- Below is an expression which has several errors:

```
var x = y /*operator is missing here*/x;

{ // This code block is not closed
    f1(x, y, /*function parameter is missing here*/)
    {

        var z = ++x + y + /*' +' is not a postfix and operand is missing */;
        return + /*' +' is not a postfix and operand is missing */ z + y;
    }
// Closing curly brace is missing here
```

CHAPTER

FOURTEEN

PARSING SECTION IN TEXT

- Sometimes we want to parse a single braces expression at specific location in text (i.e., an expression starting with "(" or "[" and ending in ")" or "]" correspondingly) or single code block expression (i.e., an expression starting with **UniversalExpressionParser.IExpressionLanguageProvider.CodeBlockStartMarker** and ending in **UniversalExpressionParser.IExpressionLanguageProvider.CodeBlockEndMarker**). In these scenarios, we want the parser to stop right after fully parsing the braces or code block expression.
- The interface **UniversalExpressionParser.IExpressionParser** has two methods for doing just that.
- The methods for parsing single braces or code block expression are **UniversalExpressionParser.IParseExpressionResult ParseBracesExpression(string expressionLanguageProviderName, string expressionText, IParseExpressionOptions parseExpressionOptions)** and **UniversalExpressionParser.IParseExpressionResult ParseCodeBlockExpression(string expressionLanguageProviderName, string expressionText, IParseExpressionOptions parseExpressionOptions)**, and are demonstrated in sub-sections below.
- The parsed expression of type **UniversalExpressionParser.IParseExpressionResult** returned by these methods has a property **int PositionInTextOnCompletion { get; }** that stores the position in text, after the parsing is complete (i.e., the position after closing brace or code block end marker).

14.1 Example of parsing single braces expression

- Below is an an SQLite table definition in which we want to parse only the braces expression **(SALARY > 0 AND SALARY > MAX_SALARY/2 AND f1(SALARY) < f2(MAX_SALARY))**, and stop parsing right after the closing brace ')'.

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   MAX_SALARY     REAL,
   /* The parser will only parse expression
   (SALARY > 0 AND SALARY > MAX_SALARY/2 AND f1(SALARY)<f2(MAX_SALARY)) and will stop␣
↪right after the
   closing round brace ')' of in this expression. */
   AVG_SALARY     REAL
                   CHECK(SALARY > 0 AND
                             SALARY > MAX_SALARY/2 AND
```

(continues on next page)

(continued from previous page)

```
                                f1(SALARY) < f2(MAX_SALARY)),
   ADDRESS         CHAR(50)
);
```

- The method **ParseBracesAtCurrentPosition(string expression, int positionInText)** in class **UniversalExpressionParser.Tests.Demos.ParseSingleBracesExpressionAtPositionDemo** (shown below) demonstrates how to parse the braces expression **(SALARY > 0 AND SALARY > MAX_SALARY/2 AND f1(SALARY) < f2(MAX_SALARY))**, by passing the position of opening brace in parameter **positionInText**.
- Here is square braces expression **[f1()+m1[], f2{++i;}]** between texts 'any text before braces' and 'any text after braces…', which can also be parsed using the code in class **UniversalExpressionParser.Tests.Demos.ParseSingleBracesExpressionAtPositionDemo**.

```
any text before braces[f1()+m1[], f2
{
   ++i;
}]any text after braces including more braces expressions that will not be parsed
```

14.2 Example of parsing single code block expression

Below is a text with code block expression **{f1(f2()+m1[], f2{++i;})}** between texts 'any text before code block' and 'any text after code block…' that we want to parse.

```
any text before braces[f1()+m1[], f2
{
   ++i;
}]any text after braces including more braces expressions that will not be parsed
```

- The method **IParseExpressionResult ParseCodeBlockExpressionAtCurrentPosition(string expression, int positionInText)** in class **UniversalExpressionParser.Tests.Demos.ParseSingleCodeBlockExpressionAtPositionDemo** demonstrates how to parse the single code block expression **{f1(f2()+m1[], f2{++i;})}**, by passing the position of code block start marker '{' in parameter **positionInText**.

CHAPTER

FIFTEEN

CASE SENSITIVITY AND NON STANDARD LANGUAGE FEATURES

15.1 Case sensitivity

- Case sensitivity is controlled by property **bool IsLanguageCaseSensitive { get; }** in interface **UniversalExpressionParser.IExpressionLanguageProvider**.
- If the value of this property **IsLanguageCaseSensitive** is **true**, any two expressions are considered different, if the expressions are the same, except for capitalization of some of the text (say "public class Dog" vs "Public ClaSS DOg"). Otherwise, if the value of property **IsLanguageCaseSensitive** is **false**, the capitalization of any expression items does not matter (i.e., parsing will succeed regardless of capitalization in expression).
- For example C# is considered a case sensitive language, and Visual Basic is considered case insensitive.
- The value of property **IsLanguageCaseSensitive** in abstract implementation **UniversalExpressionParser.ExpressionLanguageProviderBase** of **UniversalExpressionParser.IExpressionLanguageProvider** returns **true**.
- The expression below demonstrates parsing the expression by **UniversalExpressionParser.IExpressionLanguageProvider** with overridden **IsLanguageCaseSensitive** to return **false**.

15.2 Non standard comment markers

- The properties **string LineCommentMarker { get; }**, **string MultilineCommentStartMarker { get; }**, and **string MultilineCommentEndMarker { get; }** in interface **UniversalExpressionParser.IExpressionLanguageProvider** determine the line comment marker as well as code block comment start and end markers.
- The default implementation **UniversalExpressionParser.ExpressionLanguageProviderBase** of **UniversalExpressionParser.IExpressionLanguageProvider** returns "//", "/*", *and* "/" for these properties to use C# like comments. However, other values can be used for these properties.

- The expression below demonstrates parsing the expression by an instance of **UniversalExpressionParser.IExpressionLanguageProvider** with overridden **LineCommentMarker**, **MultilineCommentStartMarker**, and **MultilineCommentEndMarker** to return "rem", "rem*", "*rem".

15.3 Non standard code separator character and code block markers

- The properties **char ExpressionSeparatorCharacter { get; }**, **string CodeBlockStartMarker { get; }**, and **string CodeBlockEndMarker { get; }** in interface **UniversalExpressionParser.IExpressionLanguageProvider** determine the code separator character, as well as the code block start and end markers.
- The default implementation **UniversalExpressionParser.ExpressionLanguageProviderBase** of **UniversalExpressionParser.IExpressionLanguageProvider** returns ";", "{", and "}" for these properties to use C# like code separator and code block markers. However, other values can be used for these properties.
- The expression below demonstrates parsing the expression by an instance of **UniversalExpressionParser.IExpressionLanguageProvider** with overridden **ExpressionSeparatorCharacter**, **CodeBlockStartMarker**, and **CodeBlockEndMarker** to return ";", "BEGIN", and "END".

15.4 Example demonstrating case insensitivity and non standard language features

- The expression below is parsed using the expression language provider **UniversalExpressionParser.DemoExpressionLanguageProviders.VerboseCaseInsensitiveExpressionLanguageProvider**, which overrides **IsLanguageCaseSensitive** to return **false**. As can bee seen in this example, the keywords (e.g., **var**, **public**, **class**, **::pragma**, etc), non standard code comment markers (i.e., "rem", "rem*", "*rem"), *code block markers (i.e., **BEGIN**, **END*) and operators **IS NULL**, **IS NOT NULL** can be used with any capitalization, and the expression is still parsed without errors.

```
rem This line commented out code with verbose line comment marker 'rem'
rem*this is a demo of verbose code block comment markers*rem

rem#No space is required between line comment marker and the comment, if the first
rem character is a special character (such as operator, opening, closing round or squer␣
↪braces, comma etc.)

BEGIN
    println(x); println(x+y)
    rem* this is an example of code block
    with verbose code block start and end markers 'BEGIN' and 'END'.
    *rem
END

Rem Line comment marker can be used with any capitalization

REm* Multi-line comment start/end markers can be used with
any capitalization *ReM

rem keywords public and class can be used with any capitalization.
PUBLIC Class DOG
```

(continues on next page)

(continued from previous page)

```
BEGIN Rem Code block start marker 'BEGIN' can be used with any capitalization
    PUBLIc static F1(); rem keywords (e.g., 'PUBLIC') can be used with any capitalization
end

REm keyword 'var' can be used with any capitalization.
VaR x=::PRagma y;

PRintLN("X IS NOT NULL=" + X Is noT Null && ::pRAGMA y is NULL);

f1(x1, y1)
BEGin Rem Code block start marker 'BEGIN'can be used with any capitalization.
   Rem Line comment marker 'rem' can be used with any capitalization
   rem Line comment marker 'rem' can be used with any capitalization

   REm* Multi line comment start/end markers can be used with
   any capitalization *rEM

   RETurN X1+Y1; rem unary prefix operator 'return' (and any other) operator  can be␣
↪used with any capitalization.
enD rem Code block end marker 'END' can be used  with any capitalization.
```

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search