XVIII Conference on Applied Mathematics
Novi Sad, 2010, pp. 95–100

EXTENSIBLE EXPRESSION EVALUATOR FOR THE DYNAMIC GEOMETRY SOFTWARE GEOMETRIJICA

Davorka Radaković, Đorđe Herceg[1], Markus Löberbauer[2]

Abstract

We present an extensible expression evaluator, which was developed as a component of our dynamic geometry software called *Geometrijica*. The evaluator supports numerical and text data types, as well as mathematical functions and a number of object data types, representing geometrical shapes. It maintains a list of defined expressions and their dependencies and also provides a notification mechanism which reports changes in expression values. We devised a method for adding new functions, which requires the developer only to implement a single method per function.

Key words and phrases: dynamic geometry, expression evaluation

1 Introduction

The dynamic geometry software *Geometrijica* can visualize geometrical drawings. A geometrical drawing consists of geometrical objects such as points, lines and curves. These objects can depend on other objects. For example, a line defined by two points depends on these points, and a circle defined with a point as its center and the radius defined by the length of a line depends on the point and the line. A change in an object causes cascading changes in all dependent objects, and subsequent repainting of the geometrical drawing on screen.

Geometrijica consists of an evaluation core and extensions like a graphical user interface, see Figure 1. The evaluation core contains an evaluation engine, a function library and a varibale storage. The evaluation engine evaluates expressions, the functions used in these expressions must be known in the function library. The function libarary provides built-in and custom functions, custom functions can be added with library files. Results of expressions can be stored in variables and used in other expressions, these variables are held in the variable storage. Outside the evaluation core

[1]Department of Mathematics and Informatics, Faculty of Science, University of Novi Sad, Serbia, e-mail: *davorkar@dmi.uns.ac.rs*, *herceg@dmi.uns.ac.rs*

[2]Institute for System Software, Johannes Kepler University, Altenberger Strasse 69, 4040 Linz, Austria, e-mail: *loeberbauer@ssw.jku.at*

Geometrijica provides a user interface that accepts text input and displays the geometrical drawings. The user interface uses a parser to convert the text input into expression trees for the evaluation engine. Further extensions can be attached to *Geometrijica* as customers, e.g., a software component that draws geometrical objects on the screen.

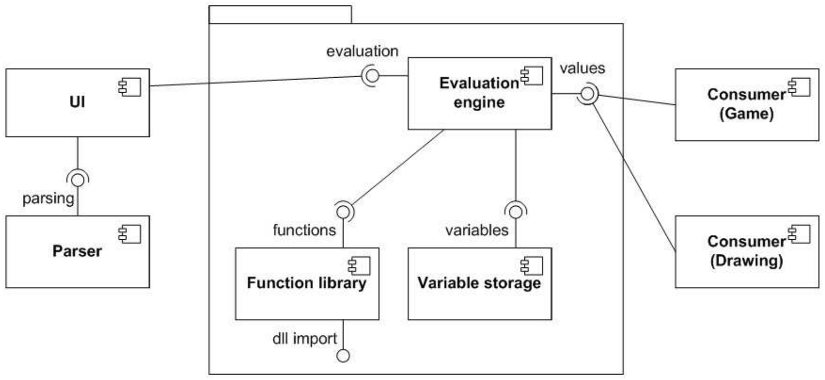

Figure 1: Components of *Geometrijica*

2 Representation of geometrical drawings

Geometrical shapes are entered in textual form, as functions applied to arguments. Each shape is represented by a corresponding function, e.g., a point can be declared as `A = Point(2,3)`. For often used primitive shapes, such as a point, there is also a short notation which omits the function name, e.g., a point can also be declared as `B = (0,-1)`. Properties of geometrical shapes are accessed using the dot notation, e.g., the length of a segment can be accessed with `Segment(A,B).Length`. The usual C# expression syntax is also supported, which allows expressions such as `D = 2 + Sqrt(3)`, or `E = (A.X, B.Y/2)`. The basic idea is to represent everything as functions applied to arguments, as it is done in some functional programming languages, for example Mathematica. In order to implement member access (i.e. accessing properties of objects) and reading of variable values as functions, we introduced the `ValueOf` and `MemberOf` functions.

Thus the expressions `D+1` and `A.X` are transformed to `ValueOf("D") + 1` and `MemberOf(ValueOf("A"), "X")` respectively. Furthermore, since addition is represented by the `Plus` function, the first expression now becomes `Plus(ValueOf("D"), 1)`.

Each function is implemented as a separate class deriving from the base class `Expression`, see Figure 2. These classes can have properties that are accessible using the dot notation. For example, the class `EPoint` has the properties `X` and `Y`, representing coordinates of a point in plane, and the class `ELine` has the properties `P1` and `P2`, which represent two points in plane that define the line.

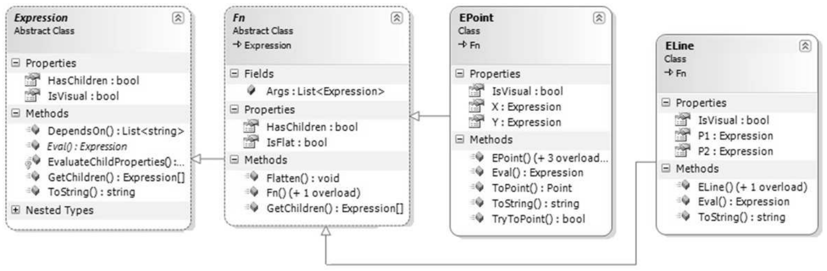

Figure 2: Excerpt of the expression class hierarchy in *Geometrijica*

The parser builds an expression tree from the text input, which is then passed to the evaluation engine, which evaluates expression trees in bottom-up order. An expression tree can be evaluated or assigned to a variable. If the expression is assigned to a variable it can be used in further expressions. To make sure that the expressions remain computable, the evaluation engine prohibits circular references, e.g., `A = f(A)` and broken dependencies, e.g., `A = f(B)` where `B` is undefined. Variables containing expressions that reference other variables are recalculated every time a referenced variable changes.

If the evaluation of a function fails, e.g., because of a division by zero, an `Error` object is returned. In compound expressions, such as `f(g(x))`, where `g(x)` yields error, the `Error` result bubbles up the expression tree.

```
public override Expression Eval() {
   Expression first = _x.Eval();
   Expression second = _y.Eval();

   if ((first is Number))!
       Error("Argument '{0}' is not a number.", _x.ToString());
   if ((second is Number))!
       Error("Argument '{0}' is not a number.", _y.ToString());

   return new Number((Number)first + (Number)second);
```

Listing 1. An example of the full format of authority record for the author

The `Eval()` method, implemented in each expression class, is called by the evaluation engine when it evaluates the expression. The example in Listing 1 shows the `Eval()` method of the expression class Plus. The two arguments of the function are evaluated, the result types are checked, and then the sum is calculated.

3 Extending the parser

The Input language for *Geometrijica* recognizes expressions and value assignments as statements. Expressions are built from literals and variables, using the arithmetic operations (`+`, `-`, `*` and `/`), function calls and member access. All expressions are written in prefix notation as functions applied to lists of arguments. For simplicity, however, arithmetic operations can also be written in infix notation, and member access using the dot notation. The parser for this language is generated in C# from an attributed grammar using the compiler generator Coco/R.

As our goal is to make both the parser and the evaluation engine easily extensible with new functions, the parser creates function objects with the factory class `FunctionFactory`. If the function names were hard-coded in the grammar and the factory class, then each added function would result in several modifications of the source code:

- Add the function name to the grammar,
- Add the function implementation to the C# project,
- Add the function name to the function factory class and implement object creation,
- Recompile the entire project.

In order to simplify this process, we use reflection to obtain all relevant information from the function implementation and then propagate the information to the parser and the `FunctionFactory`. The parser recognizes new function names and generates appropriate objects during parsing.

The grammar rules for function calls are:

```
<Function> := <Identifier> <ArgumentList>
<ArgumentList> := '(' { <Expression> { ',' <Expression> } } ')'
```

The `FunctionFactory` maintains a mapping between known function names and corresponding classes and creates appropriate objects during parsing. If an unknown function name is encountered, an exception is raised.

The class `Fn` serves as a base class for all functions. This class provides common code, used by all functions in *Geometrijica*. Thereby, for a new function only the implementation of the method `Eval()` must be given in the deriving class. Further this class must be annotated with the attribute `Function`, to specify the function name, which will be used during input.

The example in Listing 2 demonstrates the implementation of the mathematical function square root. It can be observed that there is no error handling code. Instead, error handling is performed by the evaluation engine. Thus writing the function classes comes down to implementing the method `Eval()`.

```
[Function("Sqrt")]
public class Sqrt : Fn {
    public override Expression Eval() {
        if (Args.Count = 1) !
            Error("Invalid number of arguments.");
        } else {
            Number ex = (Number)Args[0].Eval(depth+1);
            return new Number(Math.Sqrt(ex.Value));
        }
    }
}
```

Listing 2. The implementation of the square root function

Custom functions can be added to the parser and the evaluation engine by putting the compiled dll file in the installation directory of *Geometrijica*. *Geometrijica* searches for classes that implement functions in the dll files in this directory, found classes are added to the function factory and thus become usable.

4 Conclusion

The parser and expression evaluator in *Geometrijica* can be easily extended by implementing a new function derived from the base class `Fn`, annotating it with the `Function` attribute, and implementing the method `Eval()`. This approach enables us to have a single point for adding new functions, without the need for changes in *Geometrijica*.

References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers – Principles, Techniques and Tools, Addison-Wesley publishing company, 1986

[2] Hanspeter Mössenböck - Coco/R - A Generator for Fast Compiler Front-Ends, Report 127, Feb. 1990
ftp://ftp.ssw.uni-linz.ac.at/pub/Reports/Coco.Report.ps

[3] Hanspeter Mössenböck, The Compiler Generator Coco/R – User Manual, Johannes Kepler University Linz, Institute of System Software
http://ssw.jku.at/Coco/

[4] P. D. Terry, Compiler and compiler generators – an introduction with C++, International Thompson Computer Press, 1997
http://www.scifac.ru.ac.za/coco/

[5] Coco R-plugin for VS.NET
http://www.ssw.uni-linz.ac.at/Teaching/Projects/CocoPlugin/

[6] Hanspeter Mössenböck, Data Structures in Coco/R, Johannes Kepler University Linz, Institute of System Software, April 2005
http://www.ssw.uni-linz.ac.at/Coco/Doc/DataStructures.pdf

[7] Hanspeter Mössenböck, Tutorial on the Compiler Generator Coco/R, Johannes Kepler University Linz, Institute of System Software
http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/Tutorial/

[8] Albrecht Wöß, Markus Löberbauer, Hanspeter Mössenböck, LL(1) Conflict Resolution in a Recursive Descent Compiler Generator, Johannes Kepler University Linz, Institute of System Software
http://www.ssw.uni-linz.ac.at/Coco/Doc/ConflictResolvers.pdf