PEG-Based Language Workbench

Yuriy Korenkov, Ivan Loginov, Arthur Lazdin
Saint Petersburg National Research University of Information Technologies, Mechanics and Optics
Saint Petersburg, Russia
{ ged.yuko, ivan.p.loginov }@gmail.com, lazdin@yandex.ru

Abstract—**In this article we present a new tool for language-oriented programming which provides to user convenient means to describe the domain specific languages in the form of language based on parsing expression grammars and helpful tools for grammar debugging. Also we consider the sample of using this toolkit as a part of an integrated development environment.**

I. INTRODUCTION

Computers are widely used in different areas therefore mass creation of corresponding software for them represents a very important problem. For example economic problems are characterized by a large amount of calculations and data that is used, including presence of complex logical structures, use of various input and output result tables such as applications, statements, reports, etc. Manual programming of such tasks potentially is a very slow process associated with a large number of bugs that can be detected only during the debugging and testing process.

Automatic programming which purpose is a generation of a program using computers themselves can eliminate these difficulties. To make this possible we need a formal language that can be used to describe the solution of the problem in terms of a particular domain. Such languages are called Domain Specific Languages (DSL) [1].

There are many examples where such languages are used in a variety of areas. Some of them are directly related to software development while others are very specific.

As an example of such language the Perl language [2] can be mentioned. It was designed for work with texts, allowing extraction of information from them and generation of reports that are based on this information. To perform these tasks Perl needs implementation of mechanisms that can do processing of text file contents in a convenient way. One of such mechanisms is the support of regular expressions.

A good example of a domain specific language is SQL. This language was designed specifically to work with relational databases. Additionally, some languages that were created for data manipulation are based on SQL. For example LINQ (Language Integrated Query) [3], the built-in C# for querying collections of data. Also there is a large number of languages that are specialized for different areas of business and production.

For example languages for hardware description (VHDL, Verilog), languages for symbolic computations (Mathematica, Maple, Maxima, etc.).

At present many domain specific languages are created for many different subject areas. On the other hand continuous attempts to develop a new solution for language-oriented programming indicate that further work in this area is needed. Such work includes improving usability (convenience) of the created DSL by integration with existing development environments or creation of a new tool. Such tool can be oriented on the specific tasks (within project for concrete DSL) or can represent general solution (like MPS [4]). In this article convenience is understood as presence of program features such as syntax highlighting, auto-completion of a text, etc.

II. STATE OF ART

Quite a lot of tools for language-oriented programming have been created already: various template engines, visual editors of data models and processes, markup tools, integrated with the program code, and so on.

Two categories of existing solutions can be distinguished:

1) Highly specialized in the private subject area of a specific project. Usually they represent an implementation of the most nowhere used DSL.

2) Tools with opportunities for language-oriented programming [5], whereby new DSL can be created.

Highly specialized solutions are implemented through prototyping tool (in this case they inherit appropriate opportunities) or created "from scratch".

Some of latest existing solutions can be categorized as language workbench. This definition describes a new set of tools that are used for creation of domain specific languages. Such tools allow to define the abstract syntax that can be used to acquire corresponding language. This language will be then accessible in the integrated development environment (IDE). Language workbench can be defined as one of language-oriented programming means.

One of the most famous examples of tools for language-oriented programming is JetBrains Meta Programming System (MPS). During the process of language development this system provides all the features that are typical for modern IDE including auto-complete, syntax highlighting "on the fly", error checking and others. MPS is shipped with many samples for extending the Java language. It should be mentioned that MPS is independent from the programming languages.

Definition of a language with MPS consists of three stages:

- The description of the abstract syntax (in the terminology of MPS concepts);
- description of a code generator;
- description of an editor for the language.

The dnq language can be mentioned as one of the examples of using MPS in practice. This language provides support for databases. For example it is used in the issue tracker called YouTrack [6] that is based on MPS (implements language-oriented approach) and which syntax reminds LINQ.

Another example of language workbenches is Nitra [7]. Nitra is a set of tools that aims to facilitate the creation of programming languages and DSLs in particular.

At present Nitra supports development of parser generators for the .NET Framework. However the system is designed in the way that it also allows creation of parsers for the other platforms. Potentially it can be used to generate code of parsers in pure C or code for a virtual machine like LLVM or Java.

The development of programming language with Nitra starts with the creation of syntax module.

Below (see Listing 1) is an example of Nitra-grammar [8], which describes the language of arithmetic expressions.

This article introduces a solution that allows creation of domain-specific languages that can be used in a wide range of general-purpose programming languages (over 50 languages, such as C#, C++/CLI, VB.NET, F#, etc.). The proposed solution does not use any additional dependencies, unlike other tools. For example Nitra is built on the Nemerle and because of that infrastructure of this language is needed for the usage of resulting DSL by the end user.

III. Configurable parser

To be able to create a DSL by flexible way mechanism that allows parsing the source code of various types (describing various grammars) is required. Therefore the parser should have possibility to be configured in different ways.

It should be mentioned that for the effective usage of DSLs, this configuration must be made as flexible as possible. The classical parser building tools don't provide that because they are based on generative grammars by Chomsky classification, which leads to the need of complex transformations of a formalized language for parser creation.

Possible solution of the above problem is to use analytical grammar class PEG (Parsing Expression Grammar) [9]. PEG essentially consists of a set of rules similar to the set of nonterminals from grammars of Chomsky's hierarchy that are got from the analyzed text. In this case, a grammar parser can be easily constructed from a stack-machine whose configuration can be set dynamically. In addition, PEG does not require a separate phase of parsing, because the rules for it can be determined together with other rules for non-terminals.

Listing 1 Nitra simple grammar example

```
syntax module Calc
{
 using Whitespaces;

 [StartRule, ExplicitSpaces]
 syntax Start = s Expr !Any;

 syntax Expr
 {
  | Number
    {
     regex Digitd = ['0'..'9'];
     regex Number = Digitd+ ('.' Digitd+)?;
    }

  | Number;
  | Parentheses = '(' Expr ')';
  | Add       = Expr '+' Expr precedence 10;
  | Sub       = Expr '-' Expr precedence 10;
  | Mul       = Expr '*' Expr precedence 20;
  | Div       = Expr '/' Expr precedence 20;
  | Pow       = Expr '^' Expr precedence 30
                                right-associative;
  | Neg       = '-' Expr     precedence 100;
 }
}
```

IV. Our approach to implementation

In this research the PEG-parser was implemented. This parser processes PEG, presented as a set of named rules that describe expression of parsing. It contains rich features for describing rules of grammars of varying complexity.

Rules can be parameterized by the expressions-arguments passed into them. That makes it possible to describe a kind of macro-rules, forming the final expression of this rule already in the time of use.

Also there is a possibility to set pattern of skipping globally for the entire grammar. Such pattern defines the ignorable parts of the text. This allows distinguishing different parts of the text in terms of their need for the user and from the viewpoint of the analysis model, which represents parsing result. One of the examples is ordinary and documenting comments that should be distinguished from each other to allow syntax highlighting in the editor. This pattern represents an expression that is similar to the root expression of grammar. Root expression is an expression from which the parsing of text with given grammar starts.

Table I lists main types of expressions used for the grammar of the implemented parser.

TABLE I. GRAMMAR EXPRESSION TYPES

Expression type	Purpose	Description
Symbols	Parsing	Parsing of fixed sequence of characters.
Regex	Parsing	Parsing the sequence of characters defined by the regular expression.
RuleUsage	Parsing	Nested call another rule by name.
SpecialUsage	Call the internal function parser.	Used to perform actions not directly related to the process of parsing - transmission of information about an error in the text, logging of parsing process.
And	Control, Parsing	Executes a sequence of calls of child expressions.
Or	Control, Parsing	Executes a sequence of calls to the first successful call of the child expression, confirming only his change of position in the parsed text.
MatchNumber	Control, Parsing	Executes a sequence of calls of a one child expression specified number of times.
Not	Control, introspection	Checks that the further text could not be parsed by specified subexpression. Does not move the position of the text-pointer.
Check	Control, introspection	Checks that the further text could be parsed by specified subexpression. Does not move the position of the text-pointer.

The result of analysis of the text by this parser is a tree of StringTreeNode elements, describing corresponding fragments of parsed text to specified grammar rules. After analysis of this tree it is possible to translate it automatically into a tree of any other objects for example to abstract syntax tree of the compiler that consists of defined structures and classes.

To use the parser firstly we need to specify its grammar. For this purpose we use the tool that allows describing the language in the form of grammar similar to PEG.

Listing 2 Example of the simple arithmetic grammar

```
[OmitPattern("[\s]*")]
[RootRule(expr)]
SimpleArithmetics {

        productOp: '*' | '/';
        sumOp: '+' | '-';

        /* arithmetic expression */
        [RewriteRecursion]
        #expr: {
                |sum: expr sumOp expr;
                |product: expr productOp expr;
                |[right]power: expr '^' expr;
                |#braces: '(' expr ')';
                |num: "[0-9]+";
        };
}
```

The proposed language for specifying of grammar suggests writing of grammar in an intuitive manner. Regular expressions and attributes for rules (written in square brackets, for example, [right], as shown in Listing 2) are supported. See currently supported attributes in Table II.

Parsing alternatives can be described in two ways:

- as or-expression, which looks like a|b|c
- as extensible rule, which consists of a complex rule definition, where each nested rule marked with '|', like rule *expr* from the Listing 2.

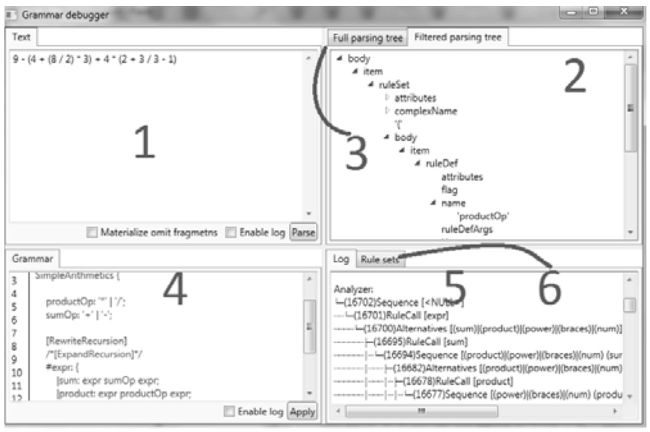
Extensible rules provide a way to extend once described grammar without need to change its definition.

TABLE II. GRAMMAR ATTRIBUTES

Attribute	Purpose
left	Marks an expression as left associative.
right	Marks an expression as right associative.
OmitPattern	Defines an expression for omitting.
RootRule	Defines the root rule.
RewriteRecursion	Marks a rule that contains alternatives to parse to make automatically rewrite of recursion.

V. Grammar debugger

A special debugger has been created to simplify the process of creating DSL grammars. It allows detailed observing how the analysis of a given text works with a given grammar, and what parsing trees are generated as a result. Fig. 2 demonstrates debugger’s window.



```
9 - (4 + (8 / 2) * 3) + 4 * (2 + 3 / 3 - 1)
```

```
body
  item
    ruleSet
      attributes
      complexName
      '{'
      body
        item
          ruleDef
            attributes
            flag
            name
              'productOp'
            ruleDefArgs
```

```
SimpleArithmetics {

    productOp: '*' | '/';
    sumOp: '+' | '-';

    [RewriteRecursion]
    /*[ExpandRecursion]*/
    #expr: {
        |sum: expr sumOp expr;
        |product: expr productOp expr;
```

```
Analyzer:
└(16702)Sequence [<NULL>]
---└(16701)RuleCall [expr]
------└(16700)Alternatives [(sum)|(product)|(power)|(braces)|(num)]
---------├(16695)RuleCall [sum]
---------|--└(16694)Sequence [(product)|(power)|(braces)|(num) (sur
---------|-----├(16682)Alternatives [(product)|(power)|(braces)|(num)
---------|-----|--├(16678)RuleCall [product]
---------|-----|--|--└(16677)Sequence [(power)|(braces)|(num) (produ
```

Fig. 1. Grammar debugger’s window

Application window contains following areas:

1) Text-field to input text to parse.

2) Filtered parsing tree.

3) Full parsing tree.

4) Text-field for grammar definition.

5) Parsing log.

6) A set of rules used in the analysis as a tree.

The program allows controlling the granularity of the logging process of analysis (grammar and parse text), and - the ability to produce materialization of skipped source code fragments.

Grammar rules tree helps to find structural errors in expressions of rules, and detailed log of the parsed text - logical errors. Full parsing log is very helpful to analyze the parser behavior. For example we can look at full log with materialization to get a lot of helpful information at the Fig 2.

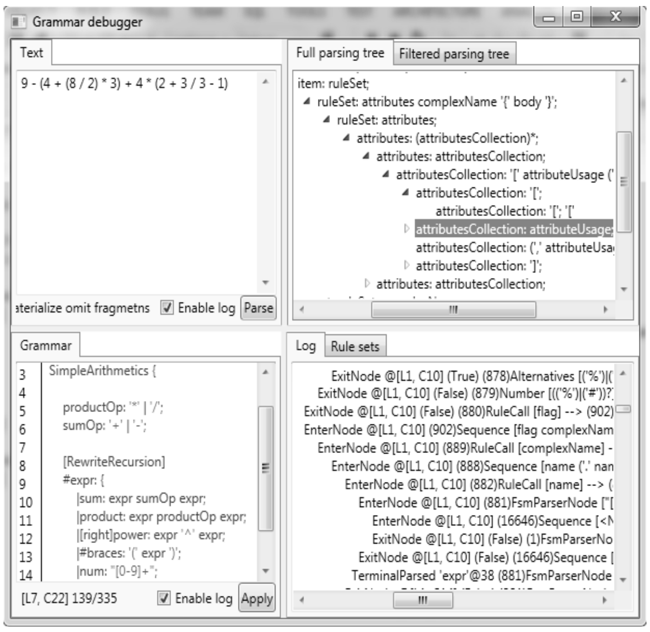
VI. Code generation

One of the most important aspects of the usage of DSL is a generation of code, which is able to interact with code written manually.

To do this it is necessary to get rid of the intermediate code generation, and DSL integrate into the general-purpose programming language. This can be done using various techniques that are specific to the programming language.

For example:

- In C there is a possibility of using the built-in text macros [10], with which the subject area can be described so that when a program is compiled this description becomes a correct description of data structures and functions of C language, which oriented to solve particular problems.



```
9 - (4 + (8 / 2) * 3) + 4 * (2 + 3 / 3 - 1)
```

```
item: ruleSet;
  ruleSet: attributes complexName '{' body '}';
    ruleSet: attributes;
      attributes: (attributesCollection)*;
        attributes: attributesCollection;
          attributesCollection: '[' attributeUsage ('
            attributesCollection: '[';
              attributesCollection: '['; '['
            attributesCollection: attributeUsage;
            attributesCollection: (',' attributeUsa
            attributesCollection: ']';
        attributes: attributesCollection;
```

```
SimpleArithmetics {

    productOp: '*' | '/';
    sumOp: '+' | '-';

    [RewriteRecursion]
    #expr: {
        |sum: expr sumOp expr;
        |product: expr productOp expr;
        |[right]power: expr '^' expr;
        |#braces: '(' expr ')';
        |num: "[0-9]+";
```

```
[L7, C22] 139/335
```

```
      ExitNode @[L1, C10] (True) (878)Alternatives [('%')|(
     ExitNode @[L1, C10] (False) (879)Number [(('%')|('#'))?
    ExitNode @[L1, C10] (False) (880)RuleCall [flag] --> (902)
   EnterNode @[L1, C10] (902)Sequence [flag complexNam
    EnterNode @[L1, C10] (889)RuleCall [complexName] -
      EnterNode @[L1, C10] (888)Sequence [name ('.' nan
       EnterNode @[L1, C10] (882)RuleCall [name] --> (
         EnterNode @[L1, C10] (881)FsmParserNode ["[
          EnterNode @[L1, C10] (16646)Sequence [<N
           ExitNode @[L1, C10] (False) (1)FsmParserNo
         ExitNode @[L1, C10] (False) (16646)Sequence [
        TerminalParsed 'expr'@38 (881)FsmParserNode
```

Fig. 2. Grammar debugger with enabled logging

- In C# there are various elements of the syntactic sugar, which allows to hide behind them the formation of the object model that describes the partial subject area in the context of object-oriented programming. These features include the collection initializers and extension-methods.

It is also possible to use programming language features with the potential in terms of the introduction of built-in DSL.

Code generation is an important issue while integrating solution with IDE because we need to obtain a generator from the parsing results.

Now have a possibility to generate a text model described in C# source code for custom grammar definitions. This source can be compiled with C# compiler for further usage with any other CLI-compatible tools, languages, etc.

Such statically generated text models consist of a set of classes where each class corresponds to rule from grammar definition. In addition, another one class is generated to implement an automatic mapping of a parsing tree (in terms of StringTreeNode) to text model (in terms of generated classes), where MappingContext<TranslationContext> contains mapping result and a set of information about mapping process.

Listing 3 demonstrates part of generated classes for arithmetic grammar from Listing 2.

Listing 3 Example text model classes

```
...
public class exprBracesType
{
    public exprType expr;
    public string[] @string;

    public exprBracesType(exprType expr,
        string[] @string) { ... }

}

...
public class exprNumType
{
    public string @string;

    public exprNumType(string @string) { ... }
}

...
public class exprType
{
    public exprSumType sum;
    public exprProductType product;
    public exprPowerType power;
    public exprBracesType braces;
    public exprNumType num;

    public exprType(exprSumType sum,
        exprProductType product,
        exprPowerType power,
        exprBracesType braces,
        exprNumType num) { ... }
}
...
public class SimpleArithmeticsTypesMapping
{
    public SimpleArithmeticsTypesMapping(
        RuleSet ruleSet) { ... }

    public MappingContext<TranslationContext> Map(
        IParsingTreeNode tree,
        ISourceTextReader reader) { ... }
}
```

VII. IDE SUPPORT

Different ways are possible for integration with IDE. The most frequently used way involves automatically or manually created resources and source code, that is used to store domain information and access to that information from general-purpose programming language of target development project respectively. Here as we talk about some tools for operations with domain specific information, representation of this information may utilize special forms of visualization for editing and observing, including graphical. However, such tools have to use efficiently proccessible form to store this information (kind of XML documents are frequently used). Depending on particularly tool, generated code may be of any complexity, but the first requirement for it is to be seamlessly integratable into main development process, to be accessible for general-purpose code completion services, etc.

Thus, there are a few bottlenecks: intermediate representation handling inside IDE and code generation process itself.

The first lies in the fact that the intermediate representation needs a number of operations to deal with it: read information to editor, store it back after changes, read it again to generate code, and then analyze that code with general-purpose language tools. Each of these steps is a potential source of errors and misunderstandings between various instruments. For example, when some error occurred during code generation, IDE needs to translate this error from domain of code generation process to domain of source DSL scope.

Frequently usage scenarios is complicated because of the necessity of calling some tool or a whole project build to completely validate domain information modifications and/or to update the general purpose language's services knowledge about generated code.

The second bottleneck lies in the fact that developers used variety of tools for project build and development operations automatization, and because of this it is necessary to be able to perform code generation outside of the IDE. For example, during build process on a build server.

In these terms, instrumentation for custom text DSLs creation and usage have to propose following compatibilities:

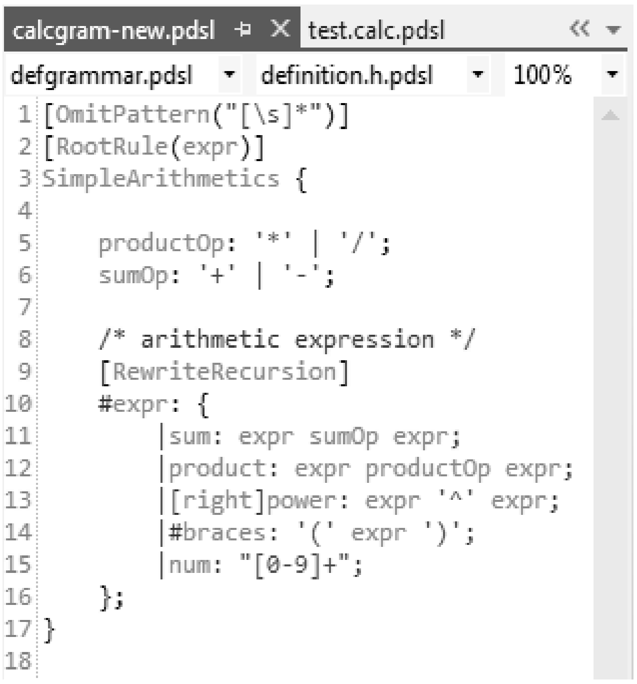
- Text editor for DSL creation and domain-specific text editing with possibility to write language definition and use this language instantly with all convenient services.
- Method to rapidly integrate created DSL into target development project.

The proposed solution – a plugin for the IDE Microsoft Visual Studio. It provides opportunities for syntax highlighting of grammar that describes a specific language as well as for the language itself, based on the same parser implementation, which is used to provide DSL integration for target project, as well as it does not require explicit intermediate representation and code generation.

Text model generation is optional and may be used with IDE command only if the DSL definition change affects parsing tree structure. Text model generation also can be performed with separate executable tool.

In the editor it is possible to set a custom highlighting scheme - the style of visual design for text - the different rules, terminals, and switch between the schemes by using drop-down lists above the window text editor.

On Fig. 3 we present screenshot of text-editor, which supports developed language workbench.



```
[OmitPattern("[\s]*")]
[RootRule(expr)]
SimpleArithmetics {

    productOp: '*' | '/';
    sumOp: '+' | '-';

    /* arithmetic expression */
    [RewriteRecursion]
    #expr: {
        |sum: expr sumOp expr;
        |product: expr productOp expr;
        |[right]power: expr '^' expr;
        |#braces: '(' expr ')';
        |num: "[0-9]+";
    };
}

```

Fig. 3. Visual Studio integration sample

To be clear with coloring schemas, see an example for simple arithmetic language on Listing 4.

VIII. Conclusion

Developed language workbench provides a convenient syntax to describe the grammar, with support it in IDE and in a special tool that allows debugging of developed Domain Specific Language.

Listing 4 Coloring schema for simple arithmetic DSL.

```
!default {
   color: #000000;
   background: #ffffff;
}
num {
   color: #0000ff;
}
sumOp, productOp {
   color: #008800;
}
/braces, braces/braces {
   color: #888888;
}
braces {
   background: #00ffff;
}
```

TABLE III. Language workbenches comparison

	MPS	**Xtext**	**Nitra**	**Presented solution**
Editor capabilities.				
Environment integration	No	Yes	Yes	Yes
Autocompletion	Yes	Yes	Yes	Planned
Syntax-highliting	Yes	Yes	Yes	Yes
High-level analysis	Yes	Yes	Planned	Planned
Static AST code generation	Yes	Yes	Yes	Yes
Lack of unnecessary code generation	No	No	No	Yes
On-the-fly grammar update	No	No	No	Yes
Text parsing capabilities				
Standalone usage	No	Yes	Yes	Yes
Usage without AST model	No	No	No	Yes
Grammar definition loading	No	No	No	Yes
Fluent grammar definition	No	No	No	Yes
Environment components independency	No	No	No	Yes

Table III presents a comparison of the developed language workbench with existing tools. It is divided into two sections. The first part describes various aspects of DSL text editing. The second part describes sides of usage mentioned tool as a component while creating some final application.

Here is brief description of some comparison criteria:

- Environment integration is about kind of editor implementation: MPS is implemented as a full-featured IDE and requires corresponding development process, while other mentioned workbenches are implemented as extensions for existing general-purpose IDE so they can be easily integrated into existing development process.

- High-level analysis is about checking of constraints between parts of an AST, type systems, multiple sources handling etc.
- Unnecessary code generation means generation of some sources, packages or other external representations to pass them between independent parts of workbench during DSL grammar editing and testing.
- On-the-fly grammar update is a possibility to change DSL grammar and see updated parser behavior immediately, without building, compiling or generating something explicitly. For example, Xtext requires another instance of an Eclipse editor to be started for DSL definition testing.
- Usage without AST model gives a possibility to use parser without definition of explicit types for AST or with manually defined AST construction.
- Grammar definition loading makes possible to load DSL definition and change parser behavior during final application execution.
- Fluent grammar definition is a feature for creation of a particular DSL definition from source code in general-purpose language.
- Environment components independency is a lack of necessity in other components or packages except parser itself. For example, Nitra parser requires Nemerle language runtime libraries.

In the nearest future, we plan to implement features:

- Autocompletion of text with respect to its grammar – one of the most important features, because people want to write code on their own DSLs in modern text editors with functions that make process faster and easier.
- Careful failure recovery. It is necessary to clearly and accurately tell the user about any errors, and ensure fast recovery after an error to continue to search for other errors. In particular, this mechanism should not slow down text editor.
- Own regular expressions engine to enable parallel parsing of alternative branches for ambiguity resolution (like in GLR-parsers). It also can be used to partially parallelize parsing algorithm.
- Recursion rethinking. As presented on Listing 2, now we use attribute [RewriteRecursion] to specify, that it is necessary to rewrite recursive calls of *expr* rule inside its alternative branches as it follows from PEG principles of left recursion handling. Firstly, it is possible to detect such things automatically. Secondly, with parallel alternative branches parsing we can try to reinterpret left recursion calls in a way when there is no need to rewrite recursion at all.
- Extend coloring schema definitions support to provide more rich and flexible ways to describe highlighting rules, especially for extendable languages.
- Extendable grammars support in Visual Studio integration – a way to use a set of grammars in one instance of an editor simultaneously.

REFERENCES

[1] Martin Fowler website, DomainSpecificLanguage, Web: http://martinfowler.com/bliki/DomainSpecificLanguage.html.
[2] M.P.Ward, “Language Oriented Programming” Computer Science Department, Science Labs, South Rd Durham, DH1 3LE, October 1994, pp. 15-16.
[3] Pro LINQ: Language Integrated Query in C# 2010. — M.: «Williams», 2011. p. 656.
[4] Martin Fowler website, A Language Workbench in Action - MPS, Web: http://martinfowler.com/articles/mpsAgree.html.
[5] Martin Fowler website, A Language Workbench in Action - MPS, Web: http://martinfowler.com/bliki/LanguageWorkbench.html
[6] Official JetBrains website, web: https://www.jetbrains.com/youtrack/
[7] An introduction to Nitra, web: http://blog.jetbrains.com/blog/2013/11/12/an-introduction-to-nitra/.
[8] RSDN Magazine: “Описание языка описания расширяемых парсеров «Nitra»”, web: https://rsdn.ru/article/nitra/Nitra-doc5.xml //Description of a language for description extensible parsers «Nitra».
[9] Bryan Ford, “Parsing Expression Grammars: A Recognition-Based Syntactic Foundation”, Massachusetts Institute of Technology, Cambridge, MA.
[10] International Standard ISO/IEC 9899:201x “Programming languages — C”, p. 166.