

# String-Embedded Language Support in Integrated Development Environment\*

Semen Grigorev  
St. Petersburg State University  
198504, Universitetsky  
prospekt 28  
Peterhof, St. Petersburg,  
Russia.  
rsdpisuy@gmail.com

Ekaterina Verbitskaia  
St. Petersburg State University  
198504, Universitetsky  
prospekt 28  
Peterhof, St. Petersburg,  
Russia.  
kajigor@gmail.com

Andrei Ivanov  
St. Petersburg State University  
198504, Universitetsky  
prospekt 28  
Peterhof, St. Petersburg,  
Russia.  
ivanovandrew2004@gmail.com

Marina Polubelova  
St. Petersburg State University  
198504, Universitetsky  
prospekt 28  
Peterhof, St. Petersburg,  
Russia.  
polubelovam@gmail.com

Ekaterina Mavchun  
St. Petersburg State University  
198504, Universitetsky  
prospekt 28  
Peterhof, St. Petersburg,  
Russia.  
emavchun@gmail.com

## ABSTRACT

Most general-purpose programming languages allow to use string literals as source code in other languages (they are named string-embedded languages). Such strings can be executed or interpreted by dedicated runtime component. This way host program can communicate with DBMS or web browser. The most common example of string-embedded language is Dynamic SQL or SQL embedded into C#, C++, Java or other general-purpose programming languages. Standard Integrated Development Environment functionality such as syntax highlighting or static error checking in embedded languages can help developers who use such technique, but it is necessary to process string literals as a code to provide these features. We present a platform allowing to create tools for string-embedded languages processing easily, and compare it with other similar tools like IntelliLang. We also demonstrate a plug-in for ReSharper created by using the platform. The plug-in provides code highlighting and static error checking for string-embedded T-SQL in C#.

## Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*parsing*;  
D.2.7 [Software Engineering]: Distribution, Maintenance,  
and Enhancement—*Restructuring, reverse engineering, and reengineering*

---

\*(Does NOT produce the permission block, copyright information nor page numbering). For use with ACM.PROC.ARTICLE-SP.CLS. Supported by ACM.

## General Terms

Algorithms, Parsing

## Keywords

String-embedded language, abstract parsing, parser generator, lexer generator, integrated development environment, IDE, Dynamic SQL.

## 1. INTRODUCTION

Multiple languages are often used in large software system development. In this case there are one *host* language and one or more *string-embedded* or just *embedded* languages. String expressions of host language form programs in another language and then they are interpreted in runtime by special component such as database management system or web-browser. The majority of general-purpose programming languages can play role of both the host and the embedded programming language. Examples of embedded languages are presented below.

- Dynamic SQL:

```
CREATE PROCEDURE [dbo].[MyProc]
    @TBLRes VarChar(30)
AS
EXECUTE ('INSERT INTO ' + @TBLRes + ' (f1)'
+ ' SELECT ''Additional condition: '' + f2'
+ ' from #tt where sAction = ''100''')
GO
```

- Multiple embedded into PHP languages (MySQL, HTML):

```
<?php
$query = 'SELECT * FROM '. $my_table;
$result = mysql_query($query);
echo "<table>\n";
while ($line =
    mysql_fetch_array($result, MYSQL_ASSOC)){
```

```

echo "\t<tr>\n";
foreach ($line as $col_value) {
    echo "\t\t<td>$col_value</td>\n";
}
echo "\t</tr>\n";
}
echo "</table>\n";?>

```

String-embedded languages may help to compensate the lack of expressivity of general-purpose programming language in domain-specific settings. However, it is rather hard to develop, support or reengineer string-embedded code using this technique. Dynamically generated expressions are often constructed of string primitives of the host language by concatenations in loops, conditional expressions or recursive procedures. Dynamically generated expressions are simple strings in the point of view of the host language analyser and even syntactic analysis is undecidable in general case. Impossibility of static check for dynamic expression results in high possibility of getting errors in runtime.

Common practice in software system development is Integrated Development Environments using that provides such features as code highlighting, autocompletion, error handling, different kinds of refactoring. All these significantly simplifies development and debugging process. Thus the tools being able to perform *abstract analysis* — static analysis of dynamically-generated expression value set — may be really helpful.

Grammarware research and development project YaccConstructor [5] is now aimed to create an infrastructure for development of string-embedded language processing tools. In this paper we provide overview of the tools for embedded languages processing, describe the infrastructure under development and its components, and pay attention to multi-language support problems. We also describe error reporting and semantic calculation for embedded languages. Finally, we illustrate infrastructure features with developed plug-in to ReSharper<sup>1</sup>.

## 2. RELATED WORK

There are several approaches for string-embedded languages processing. The first is based on comparison of the specification of the language obtained by regular or context-free approximation of dynamically generated expression with some language reference grammar [6, 8]. This approach answers the question of the dynamic expression correctness, but cannot provide meaningful error report for an incorrect expression. The most languages being used as embedded are at least context free and it is rather hard to find context free grammar for them. This circumstance leads to the second drawback: low analysis precision due to the type of approximation being used.

Second approach — *abstract analysis* [2] — is a static analysis of some representation of dynamic expression value set. This representation can be data-flow equation, regular expression, or finite automaton — as it is in our platform. The

<sup>1</sup>ReSharper is plug-in to Microsoft Visual Studio, extending standard IDE functionality. Site (accessed: 07.07.2014): <http://www.jetbrains.com/resharper/>

tools implementing this approach can be separated into two categories: *analysis-and-parsing* and *analysis-then-parsing*. The first one stands for performing of analysis and parsing at the same time on the fly. And the second one, which is also named *step-by-step analysis*, means that first the analysis of source code — constant propagation, approximation construction — is performed and after that the parsing itself is executed. We use step-by-step analysis in our framework.

## 2.1 Existing Tools

There are several tools for processing of concrete string-embedded languages. They differ in the approaches used and the ease of new language extension support. We provide a brief overview of these tools below.

### 2.1.1 PhpStorm

PhpStorm<sup>2</sup> — integrated development environment for PHP which implements code highlighting and autocompletion of string-embedded HTML, CSS, JavaScript or SQL code. However, only if the tool deals with string primitive (no string operation is used for expression construction), it is able to provide this support. You can see PhpStorm screenshot that illustrates this feature in figure 1. "." is concatenation operator and ".=" is concatenating assignment operator. As you can see, PhpStorm recognises `$hello` value as HTML expression and provides code highlighting, but no highlighting is performed for `$string` values. PhpStorm provides a separate code editor for every string-embedded language. The drawback is that the tool does not provide error reporting for incorrect expressions (see `$error` in figure 1).



```

1  <?php
2      $usualString = 'simple string';
3      $hello ='<html><body>Hello, world</body></html>';
4      $string = '<';
5      if (cond)
6          $string .= 'html';
7      else
8          $string .= 'body';
9      $string .= '>';
10
11     $error = 'SELECT * FROM table1 FROM table1';
12  ?>

```

Figure 1: HTML code embedded in PHP code in PhpStorm

### 2.1.2 IntelliJLang

IntelliLang<sup>3</sup> — plug-in to PhpStorm IDE and IntelliJ IDEA<sup>4</sup> performing code highlighting and error reporting for string-embedded languages. It provides a separate code editor for embedded language processing by analogy with PhpStorm. The screenshots of IntelliJLang are presented in figures 2 and 3.

<sup>2</sup>IDE for PHP programming language. Site (accessed: 07.07.2014): <http://www.jetbrains.com/phpstorm/>

<sup>3</sup>IntelliLang is a plug-in offering a number of features related to the processing of embedded languages. Site (accessed: 07.07.2014): <http://www.jetbrains.com/idea/webhelp/intellilang.html>

<sup>4</sup>IntelliJ IDEA is an IDE for JVM-based development. Site (accessed: 07.07.2014): <http://www.jetbrains.com/idea/>

```

3 public String getJava(){
4     return "class A extends ";
5 }

```




Figure 2: Embedded Java code fragment in IntelliJ IDEA

The drawback of the plug-in is that it is necessary to specify manually the language of every string expression to be analysed. Figure 3 illustrates the drawback: if substring "<html>" is marked as HTML expression, IntelliLang highlights this tag and the one matching it, but it does not highlight variable *s* value despite the fact it is used in HTML code construction.

```

7 public String getHTML(){
8     String body = "<body>" + "Hello!" + "</body>";
9     String html = "<html>" + body + "</html>";
10    return html;
11 }

```

Figure 3: "<html>" is marked as HTML language and "<body>" is not marked

### 2.1.3 Alvor

Alvor<sup>5</sup> [1] — plug-in to Eclipse IDE — intended to statically validate SQL expressions embedded into Java code. It does not require specification of dynamic expression language. The tool structure is presented in figure 4.

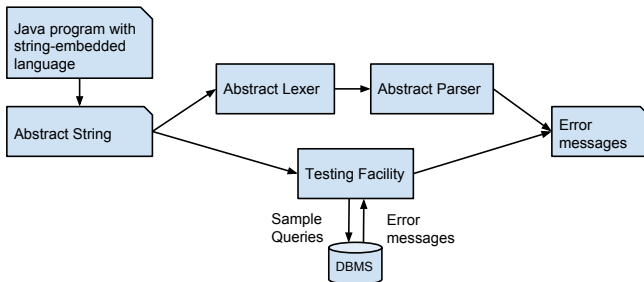


Figure 4: Alvor structure

The value set of dynamic expression is represented with abstract string — in the context of Alvor it is regular expression. After the representation is created, abstract analysis based on GLR-analysis is performed. The analysis reports lexical and syntactic errors. If there are more than one error in the expression, only the first of them is reported (see figure 5).

Alvor statically validates SQL-queries constructed of string primitives using concatenations and conditional statements in interactive mode, and performs interprocedural analysis. Alvor supports several SQL dialects (Oracle PL/SQL,

<sup>5</sup>Alvor project site (accessed: 07.07.2014):<https://code.google.com/p/alvor/>

```

11 public static void executeSQL(Connection connection)
12     throws SQLException{
13     String sql = "select id, first_name from person where ";
14     int a = 2;
15     if(a > 3){
16         sql += " b => 1 ";
17     }else{
18         sql += " c => 1 ";
19     }
20     sql += " order by first_name";
21     PreparedStatement preparedStatement =
22         connection.prepareStatement(sql);
23     ResultSet result = preparedStatement.executeQuery();
24 }

```

Figure 5: Error reporting in Eclipse IDE using Alvor

MySQL, PostgreSQL), but adding of the new languages support requires source code modification.

### 2.1.4 Java String Analyzer

Java String Analyzer<sup>6</sup> [8] — tool answering the question of syntactic correctness of dynamically generated expressions embedded in Java. The structure of JSA is presented in the figure 6.

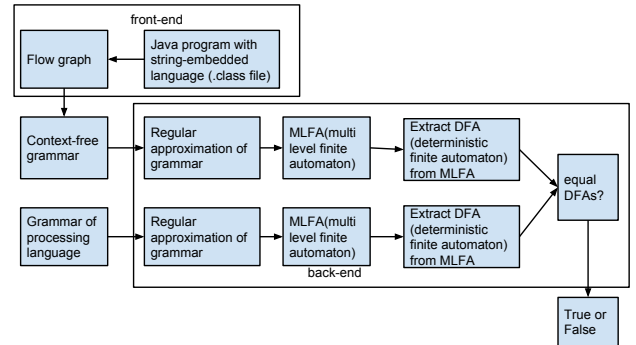


Figure 6: JSA structure

JSA allows to process several embedded languages by modifying only front-end of the tool which creates flow-graph representation of the input data. Back-end of the tool represents a flow-graph as a context-free grammar which is approximated with regular grammar, and then a finite automaton is constructed by regular grammar. This automaton approximates value set of dynamic expression. The regular approximation and finite automaton are constructed similarly for the reference grammar of the language being analysed. After that, two automata are compared to each other, and if they are equal, then the analysed expression is assumed to be syntactically correct.

### 2.1.5 PHP String Analyzer

PHP String Analyzer<sup>7</sup> [6] — tool for static validation of dynamic expressions, constructed by PHP programs. HTML

<sup>6</sup>Java String Analyzer project site (accessed: 07.07.2014):<http://www.brics.dk/JSA/>

<sup>7</sup>PHP String Analyzer project site (accessed: 07.07.2014):<http://www.score.cs.tsukuba.ac.jp/~minamide/phpsa/>

and XML expressions can be analysed as embedded. The tool is based on the ideas of JSA algorithm, but the authors proposed to use context-free approximation instead of regular.

## 2.2 String-Embedded Languages Processing

We perform step-by-step static abstract analysis of value set of dynamically generated expression approximated with finite automaton in our framework. Finite automaton can be represented as a graph and this representation is very intuitive, so we say hereinafter that it is a graph which approximates input. The description of analysis steps is provided below.

The first step is *approximation* — creation of compact representation of dynamic expression possible values. We use graph representation of finite automaton: edges labeled by strings and nodes corresponded with concatenations used to build dynamic expression. Note that this step is frontend-specific, so approximation function should be implemented for each tool created with our platform. Another important restriction: the result of approximation must be a **direct acyclic graph** (DAG) because core functions such as abstract parsing can process DAGs only in current implementation [14].

The next step is *abstract lexing* or *tokenization* of approximation result. This step transforms an input graph with string edge labels to a graph with token labels. Token part positions in source code data — references to string literals which contain parts of token and position in this literals — are preserved during tokenization. Abstract lexer can be generated from lexical specification of the language to be processed. Generator is based on FsLex<sup>8</sup> — lexer generator for F# [13].

Further, *abstract parsing* is applied to tokenized graph. Abstract parsing algorithm is based on GLR parsing algorithm which can process ambiguous context-free grammars [14]. The result of abstract parsing step consists of a parse forest for all correct values and a set of errors occurred in incorrect values. We use classical GLR **graph structured stack** (GSS) which allows to fork and merge stack according to forks and merges in input graph. We use standard generator for parsing tables building from grammar specified in Yard language [10] which allows to use attributed grammars making semantic specification possible.

As a result of abstract parsing, one gets a forest containing trees for each correct value produced by finite automaton approximating input. In the context of embedded language analysis the size of such forest is usually huge. A big number of nodes are common for different dynamic expression values and may be reused so we use **shared packed parse forest** (SPPF) [4] introduced by Rekers to compress parse forest. One often needs to calculate semantic for parsing results and it means manipulation with separate trees, not SPPF. It is enough to enumerate not all the trees from SPPF but some subset for some tasks. But sometimes enumeration of all the trees from parse forest is required, and it can lead to memory

<sup>8</sup>FsLex is lexer generator for F#. Documentation (accessed: 07.07.2014): <https://fsharpowerpack.codeplex.com/wikipage?title=FsLex%20Documentation>

and performance issues. We propose lazy enumeration of trees to solve them. One of possible function of forest subset extraction and lazy enumeration was implemented and will be described below.

## 3. THE PLATFORM

The majority of IDEs can support more than one programming language and provide the possibility of extension to support new languages. The similar functionality is necessary to extend existing IDEs and code editors with string-embedded languages support. We propose to use classical modular mechanism implying that new language support should be provided as single package named *language extension package* or *language package*. This way is suitable for end users and for language package developers because it allows to develop and use different language extensions independently.

Existent tools for string-embedded language processing are targeted to one or several concrete languages or require modification of existing code to add new language extension package. This approach limits the following development of the instrument, so it is necessary to provide developer toolkit (SDK) for language package developers and a simple package installation mechanism for users. Moreover, we should provide an ability to specify language for string variables in user code. It is necessary to provide project-depended data on mapping between strings and embedded languages used in it.

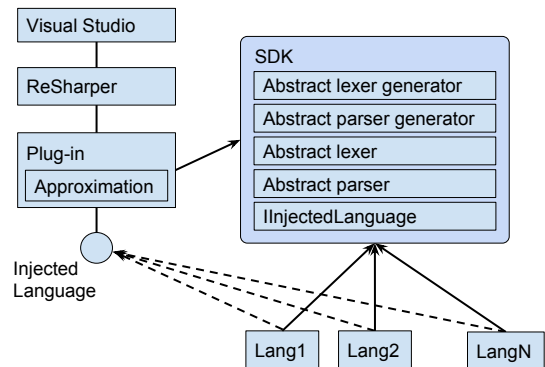


Figure 7: High-level platform structure

In figure 7 you can see a general structure of our platform. IDE integration part is based on ReSharper — Microsoft Visual Studio plug-in which provides additional functionality for code refactoring, code navigation, etc. We integrate string-embedded language support as plug-in for ReSharper. Our plug-in can load language modules dynamically (modules *Lang1*, *Lang2* ... in figure 7). This way a user can extend IDE with the support of necessary languages.

Each language module should implement common interface allowing to add new language support to IDE universally. To create new language extension, developer should describe grammar and lexical specification of the language to be supported. Grammar and lexical specification are needed for lexer and parser generation. After lexer and parser gener-

ation you can use functions from SDK to implement standard. As a result, developer gets module which adds new embedded language support.

SDK provides a set of tools and predefined functions (variety of generators and syntactic analysis functions) that simplify development of new language extension. It also contains interface for language extension packages named *InjectedLanguageModule*, and attribute *InjectedLanguage* that provides an ability to point out an embedded language which is used to construct expression. Attribute *InjectedLanguage* contains a *languageName* property of string type. The search of language extension is performed by *languageName* value which is set by the user. If there is no language extension for the pointed out language, then a user will receive an error message.

This attribute is applied to method definition. If default method is used, then it should be defined as a hotspot in separate configuration file. For example, if custom method for T-SQL query execution is implemented, then it should be marked with *InjectedLanguage* attribute as shown in the following example.

```
[InjectedLanguage ("T-SQL")]
public static void ExecuteImmediate(string query){
    Logger.log(query);
    DB.execute(query);
}
```

Assume that `DB.execute` is a third-library method and its definition cannot be marked, then it should be added into a configuration file of the tool manually.

Dynamic loading of language extensions is essential to simplify the developed tool usage. It is carried out by means of Mono.Addins — framework for extensible application creation.

### 3.1 Abstract Lexer Generator

Abstract lexer generator is a tool for creation of string-embedded language tokenizer by lexical specification of the language to be processed. We use lexer generator for F# — Fslex — as a base for our generator. Abstract lexical analysis is based on **finite-state transducer**, FST [17]. FST is a finite state automaton which can produce finite symbol sequence for each input symbol. In our tool FST is used to transform input graph to graph with edges tagged with tokens that are produced according to tables generated by specification.

Unlike the classical lexing, the abstract lexing may face the situation when single token is constructed from parts placed on different edges of input graph (and in different literals in source code). We name such tokens *multipart tokens*. You can see an example of multipart token in the code presented below. Dynamic expression in T-SQL is embedded in C# code. Multipart token is table name which constructed with ternary branching operator.

```
private void Go(bool cond)
```

```
{
    string field = cond ? "fld1 " : "fld2 ";
    string name_table = cond ? "x" : "y";
    Program.ExecuteImmediate("select " + field +
        "from table_" + name_table);
}
```

The result of dynamic query value set approximation is an input graph for abstract lexer (see figure 8).

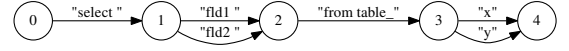


Figure 8: Approximation for code presented in example 1

The result of tokenization is presented in figure 9. As far as we should preserve references to original code for tokens, we also demonstrate backreferences in this figure. **Back-reference** (br) is data on position of token in source code: corresponded string literals and position in literal. In the figure we present only corresponded literal data. Note that in simple case we can preserve only reference to source string literal, and for multipart literal we should preserve similar data for each part produced by each independent literal. Spaces and comments are skipped as in classical lexing.

### 3.2 Abstract Parser Generator

Abstract parser generator based on RNGLR [7] parsing algorithm was implemented as a part of YaccConstructor project. RNGLR is a modification of Tomita GLR algorithm [12] which allows to process arbitrary context-free grammars. We use Yard language [10] as grammar specification language. Yard is a powerful language supporting Extended Backus-Naur Form, template rules and modularity. Also Yard supports attributed grammar specification, therefore it is possible to specify custom user semantic.

Original GLR parsing algorithm can process *Shift/Reduce* and *Reduce/Reduce* conflicts — situations when available data are not enough to choose the correct way of parsing continuation. GLR analyses every possible way in case of conflict. This allows to produce several derivations for single input sequence. One may notice that processing of the linear subgraph does not differ from the processing of sequential input. In vertices with more than one outgoing edges the situation is similar to the classical LR conflict: there are several possibilities to Shift next token. This situation is not a conflict: all variants should be processed and there is no need to choose one of them, but we call this situation *Shift/Shift* conflict by the analogy with classical conflicts. *Shift/Shift* conflict processing produces new branches in stack as any classical conflict, so forks in input graph correspond to forks in GSS. Detailed description of

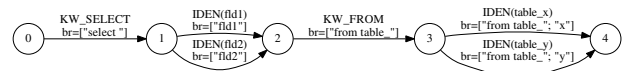


Figure 9: Result of graph presented in figure 8 tokenisation. Back references are shown.

modifications required to implement abstract parsing based on GLR parsing algorithm can be found in the paper [14].

We use classical graph structured stack and branch it when the fork appears in input graph. Unlike Alvor, we merge produced branches with equal states on the top: it improves performance and space requirements of algorithm. As far as we should process branches sequentially, we perform only single *push* for vertex with more than one outgoing edge at a time. *Push* corresponded with edge with minimal tail number in topological order should be performed immediately and other pushes should be postponed. Also we use shared packed parse forest (SPPF) [4] — a compact representation of derivation forest which allows to reuse common nodes of derivation trees. This information is important for further discussion of error detection in abstract LR parsing.

### 3.2.1 Error Detection and Error Recovery

The result of classical parsing of sequential input in a case of no error recovery performed is either correct forest or error data. However, in abstract parsing we analyze the set of values: one value per each path in input graph. Therefore abstract parsing should return as an output the set of results: success or fail for each generated expression. The result of abstract syntactic analysis in our case is the list of detected errors for incorrect paths **and** parse forest for all correct paths compressed into a single SPPF. Note that either error list or parse forest can be empty.

GLR parsing algorithm is targeted to ambiguous grammar processing and operates with graph structured stack which can be branched in case of conflict situation. The set of top vertices for every branch are *active vertices* — these vertices can be used for parsing continuation. In classical GLR if no action can be performed for some active vertex, then the vertex is removed from active vertices. Error is detected when the set of active vertices becomes empty. In abstract parsing it is a wrong approach. As far as stack branch can be produced not only by classical LR conflict but by the fork in input graph, impossibility of parsing continuation from the vertex may denote that the path being processed contains an error and corresponded data should be added to the result error set. Moreover, after an error has been detected in some path, parsing of other paths should be continued, but the processing of the erroneous paths should be skipped to prevent false error notifications. Current implementation of abstract parsing uses input graph vertex sequence sorted in the topological order as an input [14] and this fact makes erroneous path eliminating rather nontrivial. When error is detected we should compute the set of vertices to skip. The next vertex to process can be selected either from postponed pushes or from active vertices. In the first case, the vertex with minimal tail number of postponed pushes should be chosen, and in the second — the vertex which can produce *push* to the vertex with minimal number. This way we skip only the subset of input graph edges which cannot be used in any correct paths. Figure 10 illustrates input edges skipping. The token `)` is erroneous, so edges  $4 \rightarrow 6$  and  $4 \rightarrow 5 \rightarrow 7$  should be skipped.

Note that error reporting and error recovery in Generalized LR parsing are nontrivial problems. A considerable amount

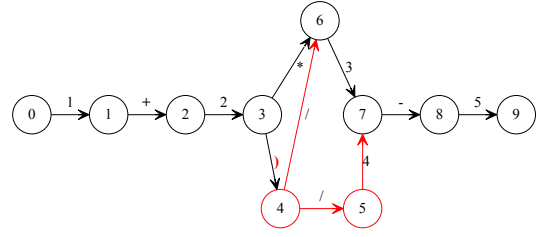


Figure 10: Example of paths skipping

of research has been done on improving error reporting for LR parsers, although for GLR parsers [11] relatively little work has been done. During the experiments we figured out that error processing in GLR-based abstract parsing is far more complex than in classical GLR parsing algorithm. It is caused by increased algorithm complexity (addition of a new type of conflict — *Shift/Shift*). So we should separate errors corresponded with wrong way of parsing continuation in classical LR conflicts and errors in the branches produced by *Shift/Shift* conflict. Conflicts mixing — when input grammar is ambiguous and input graph contains a lot of forks — is a situation in which the separation of actual errors and false errors produced by grammars ambiguity is nontrivial problem.

In spite of error recovery for RNGLR parsing algorithm has been implemented in YaccConstructor [5], the necessity of its using in abstract parsing requires detailed research. Error recovery can be useful but it potentially produce a huge number of false errors for complex dynamic expressions.

## 3.3 Semantic Calculation in Abstract Analysis

Abstract parsing is based on GLR parsing algorithm and uses well-known structure to store parsing result (SPPF) which allows to calculate semantic for extracted trees. So it is possible to support user semantic calculation in abstract analysis in case when the attributed grammar is used for the language specification. But efficient tree extraction is a problem. Unlike classical GLR, SPPF in abstract parsing contains forest for multiple input values (all correct values of dynamically generated expression). When the number of possible values for expression is very big, a huge number of trees is extracted. Full parse forest extraction may require an exponential or even infinite memory. In the next sections we propose approaches aimed to decrease semantic calculation resource requirements, which we use in our tool.

### 3.3.1 SPPF Processing in Abstract Parsing

Abstract parser returns parse forest compressed to SPPF which allows to share the nodes that are common for several parsing trees. In classical parsing nodes can be reused if they correspond to the same nonterminal and produce the same substring. Moreover, nonterminal may have more than one variant of derivation. Intermediate nodes represent such situations in SPPF: they allow to separate situations when nonterminal has multiple children (figure 11a) and when nonterminal has several possible derivations (figure 11b). In our implementation intermediate nodes store



information about the production which has caused the addition of the node to SPPF.

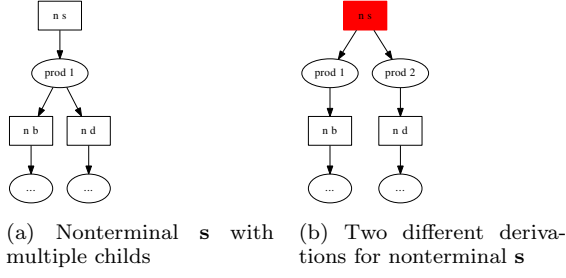


Figure 11: Node for rule  $s \rightarrow bd$  (11a) and node for two different derivation for nonterminal  $s$ :  $s \rightarrow b$  or  $s \rightarrow d$  (11b).

Trees in abstract parsing often have big number of common nodes because different values of single dynamic expression usually have a big number of common parts. This makes SPPF a good option for parse forest of single dynamic expression compression. It is important fact that for every tree from SPPF there is correct expression generated by finite automaton, which approximates input.

Suppose it is not true: it means SPPF contains parse tree of expression which is not contained in approximation. Consider the following grammar.

$s$ :  $a \mid b$   
 $a$ :  $A \ c$   
 $b$ :  $A \ D \ c$   
 $c$ :  $B \mid E \ F$

Let analyse a graph presented in figure 12. Resulting SPPF should contain two trees: the derivation tree for sequence "AB" and the other for sequence "ADEF". These trees are presented in figures 13a and 13b. These two trees contain the node for nonterminal  $c$  and it seems that the trees can be merged as presented in figure 13c. In this case four trees can be extracted from resulted SPPF. They are derivation trees for input string "AB", "AEF", "ADB", "ADEF". But only two of them ("AB" and "ADEF") can be generated by the input finite automaton.

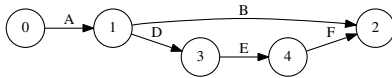


Figure 12: Input graph which produces SPPF which contains multiple trees

But it is not possible to get the SPPF presented in figure 13c because rules of merging are violated. Nodes can be reused only if derived substring are equal for both of them. But in our case one node for  $c$  is a derivation for string "B" and the other node is a derivation for string "EF". We can not reuse this node for this reason. Only the node for terminal "A" can be reused in our case. Corresponded SPPF is presented in figure 13d. So if SPPF construction is correct — only nodes

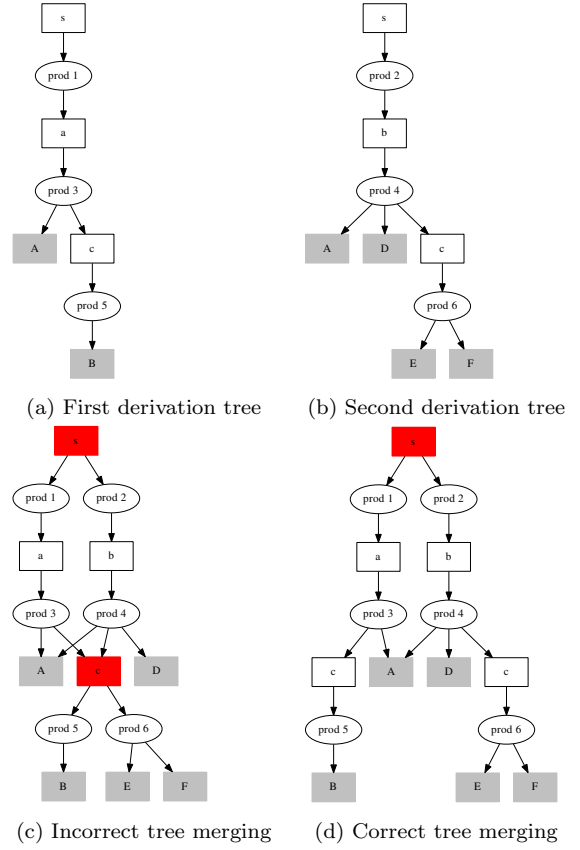


Figure 13: Tree merging

for actually common parts of expressions are reused — then SPPF does not contain incorrect trees.

### 3.3.2 Syntax Highlighting

It is not necessary to extract all the trees from SPPF for some tasks. For syntax highlighting, for example, it is enough to extract only a minimal subset of syntactically correct trees which contain all tokens in leaves. In other words, for every token we should extract at least one tree containing this token, and each tree usually contain more than one token which reduces the number of trees to be analysed. Textual representation of token in the source code can be colored in the single color, so even if there are two different trees, containing identical set of tokens, we use only one of them for code highlighting. Only one restriction is important: this tree should be syntactically correct. And this condition is always true because, as we discussed above, SPPF contains only correct trees.

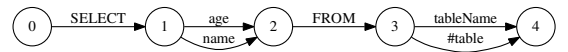


Figure 14: Example of input for sql syntax highlighting

A result SPPF for the input graph presented in figure 14 contains four trees. But two trees are enough for syntax highlighting. Remind that it should be trees containing all

tokens from the graph. For example, the trees built for the the next strings: "SELECT age FROM tableName" and "SELECT name FROM #table".

But it is still necessary to list all trees in the worst case: for instance, for the input graph presented in figure 15. Moreover, some algorithms may require full forest enumeration even for such graphs as one presented in figure 14.

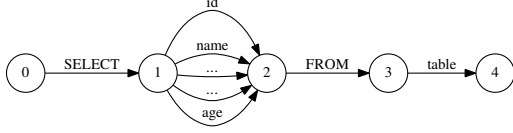


Figure 15: An example of input graph which require enumerate all parsing trees

We propose lazy enumeration of trees to reduce time and memory consumption for semantics calculation. Lazy generation allows to start tree processing before the full forest is generated. Storing full forrest as a list of trees naturally leads to memory blowup, and lazy generation allows to avoid it.

An algorithm of lazy tree generation may be implemented in different ways: it depends on the semantic calculation algorithms to be applied to parse trees. The tree enumeration algorithm which is usable for code highlighting is described below. The algorithm uses SPPF and context information — the set of tokens which are not contained in the list of already generated trees — to generate the next tree.

To generate the next tree we traverse SPPF from the root. If vertex under consideration has more than one outgoing branch, we choose only one of possible derivation variants (one of subtree). Suppose we have two vertices in SPPF presented in figure 16 where we should choose a subtree to include it in a result tree: vertex with label **field** and vertex with label **tableName**. Previously visited tokens are used to decide what tree should be chosen. This way, a subtree with biggest number of unvisited tokens are chosen. Consider the tree presented in figure 17a as a result of first traversal. When the next tree is extracted, there are two subtrees to choose from — containing token **FIELD (age)** and containing token **FIELD (name)** — in the vertex with label **field**. But the token **FIELD (age)** has been visited in the previous traversal and is contained in the first tree. Therefore the tree containing the token **FIELD(name)** is chosen 17b.

In the case of multiple trees with equal number of unvisited tokens, any of the trees is chosen. Consider the tree presented in figure 17a as the first extracted tree. And suppose we try to process the vertex with label **tableName** during the second tree generation. There are two possible subtrees and each of them contains only one token: **TABLE(table2)** or **TABLE(table3)**. At the current step we can choose any of these trees. There will be one more unvisited token after the step is completed, so one more traversal will be needed. The algorithm returns the next tree and the new set of unvisited tokens as a result. If the set of unvisited tokens is empty,

then all required trees has been generated and empty tree is returned.

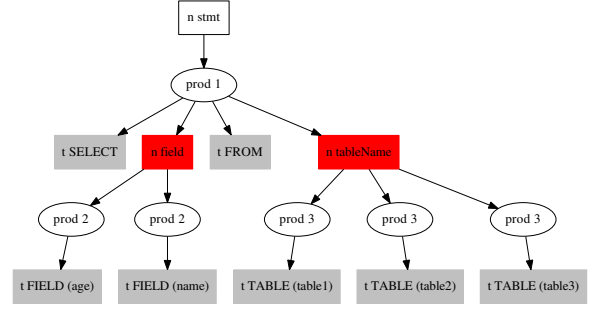


Figure 16: Example of SPPF

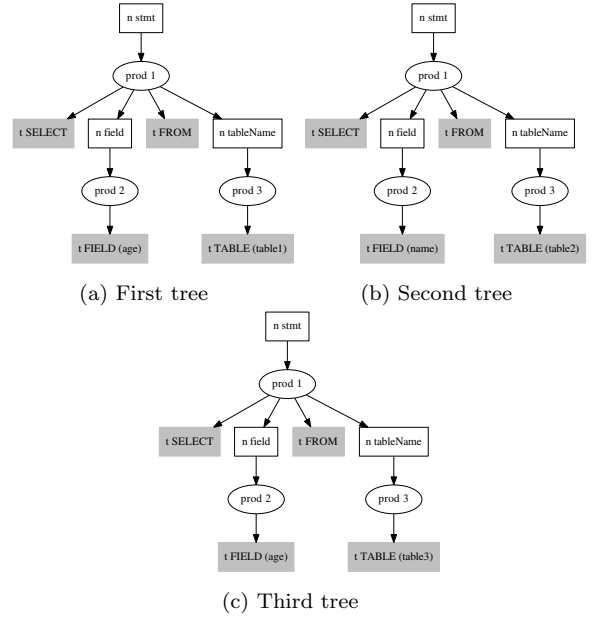


Figure 17: Trees extracted from SPPF presented in figure 16

### 3.3.3 Matching Delimiters Highlighting

IDEs and code editors provide not only static code highlighting, but perform additional dynamic highlighting based on the context information. For instance, when the caret is placed before an opening parenthesis, the matching closing parenthesis is additionally highlighted. The similar feature is useful for string-embedded languages.

Consider the input graph presented in figure 18. Here we use **LBR** and **RBR** tokens to present left and right parentheses. Let the caret is placed just before the symbol which has been processed as **LBR(1)**. As you can see, two different **RBR** tokens match the selected **LBR(1)**: **RBR(1)** and **RBR(2)**, so three symbols should be highlighted.

Suppose, we have an algorithm for highlighting of matching delimiters for singular tree. In this case it is sufficient to enumerate the trees using the algorithm described above and apply existing matching delimiters highlighting algorithm.



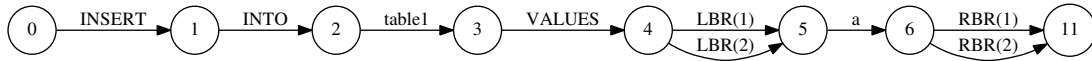


Figure 18: An example of input graph for delimiters highlighting

There is no need to enumerate all the trees for the matching delimiter highlighting — we can filter them by the existence of the matching token.

## 4. EVALUATION

The platform described above is implemented in F# [13] programming language and named **AbstractYaccConstructor** — AYC. AYC is a part of YaccConstructor — grammarware research and development project. Plug-in to Microsoft Visual Studio IDE based on ReSharper is also implemented. This plug-in is created as a demonstration of AYC and at the same time is a tool for supporting string-embedded languages in Microsoft Visual Studio IDE.

We have implemented plug-ins for two languages: arithmetic expression language (named **Calc**) and the subset of T-SQL language. We have developed lexer and parser specifications for these languages. Lexer specification was developed in a subset of FsLex language which is supported by the abstract lexer generator. Grammar (parser specification) was written in Yard language. The set of specifications can be found in the project repository<sup>9</sup>. These specifications have been used for abstract lexer and abstract parser generation using generators described above. XML-files for syntax highlighting specification and the set of classes for integration with ReSharper have been also produced during parser generation. Finally, we have built a package for each language using generated stuff.

Built packages have been loaded into the plug-in to demonstrate features provided by them. In the current version you should specify **hotspot methods** — methods which can process string-embedded languages — and corresponded languages in special configuration file to get an ability to define mapping from string literals to languages.

We have created test project and have defined hotspot methods for our string-embedded languages: *ExecuteImmediate* for T-SQL and *Eval* for Calc language. We present examples of functionality of created language extensions below.

**Syntax highlighting.** Example of string-embedded T-SQL syntax highlighting is presented in the figure 19. Only keywords are highlighted in the current configuration. One can easily configure colors for different token types using configuration XML-file generated automatically for highlighting customization. You can see that multipart tokens are supported correctly: the insert keyword is built from two string literals, and it is still detected as keyword and highlighted properly. You can also see that in the figure 20 in the line 14 "where" substring is not a keyword, although it may seem so if the first literal is analysed separately. But if the

whole statement is analysed, we detect that it is an identifier "wherev", so it should not be highlighted as a keyword.

```
32 public void Insert()
33 {
34     Program.ExecuteImmediate(
35         "ins" + "ert into x(a, b, c)" + "values(@a, @b, @c)";
36 }
```

Figure 19: T-SQL syntax highlighting

**Multiple languages support.** An example of multiple string-embedded languages support is presented in the figure 20. You can see that highlighting can be configured independently: numbers in T-SQL and in Calc expressions have different color.

```
11 public void Languages()
12 {
13     Program.ExecuteImmediate("insert into y(x,v) values (1++,2)");
14     Program.ExecuteImmediate("select x from y where" + "v > 1");
15     Program.Eval("123 +" + " 23 * 3 + 14");
16 }
```

Figure 20: Multiple language support

**Static errors detection and notification.** Our plug-in provides static error detection for string-embedded languages. You can see an example of this feature in the figure 20 in line 13: an error in values to insert is detected. An important feature of our algorithm is an ability to detect multiple errors: one error for each possible value of dynamic expression can be found. For example, the string to be executed can construct two different values by means of *if*-statement and every branch contains an error (figure 21). All these errors are found in such case by our tool unlike Alvor which detects only one error in the similar test described above (Figure 5).

```
18 public void Select(bool cond)
19 {
20     string query = "select first_name from person where";
21     if (cond)
22     {
23         query = query + " b > 1 ++ 2";
24     }
25     else
26     {
27         query = query + " c < 1 =* 3";
28     }
29     query = query + " and x > y";
30     Program.ExecuteImmediate(query);
31 }
```

Figure 21: Static error detection

**Matching parenthesis highlighting.** The feature which significantly improves readability and understanding of the code is matching parenthesis highlighting: if the caret is placed near one of the parenthesis, then the tool should

<sup>9</sup>YaccConstructor source code (accessed: 07.07.2014):<https://code.google.com/p/recursive-ascent/>

highlight the matching parenthesis. This feature is also useful for string-embedded languages but its implementation is more complex than in classical languages analysis. The construction of an expression using string-embedded language is more flexible, so several matching elements may exist for the selected one. You can see such situation in the figure 22a: there are two closing parentheses (in line 12 and 14) for the selected opening one (line 10), so all three symbols are highlighted. On the other hand, there are only one left parenthesis for each right one. So, if we select right parenthesis in line 14 then only one left parenthesis is highlighted (Figure 22b). There are special section in highlighting configuration XML-file where the matching delimiters can be specified.

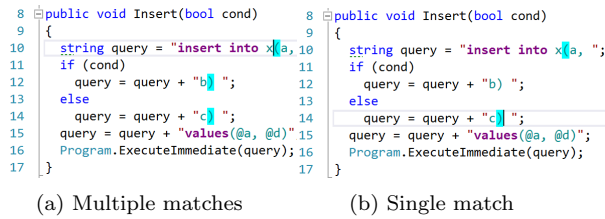


Figure 22: Paranthesis highlighting

We have demonstrated a possibility of separated implementation of support for different string-embedded languages based on SDK and the plug-in described above. Provided functionality are presented. We can conclude that even though at the current moment functionality completeness is lower than in similar tools, our tool can process more complex cases: branches and concatenations are supported, multiple errors can be found, highlighting is configurable.

## 5. FUTURE WORK

Improvements in both the platform and the plug-in are necessary. At the platform level, it is necessary to implement mechanisms required for string-embedded language. Embedded language transformation may be useful for system migration from one DBMS to another [9] or for migration to new techniques such as LINQ. This task corresponds with two big problems: possibility of nontrivial transformations (some advanced techniques [15, 16] may be useful for this problem) and transformation correctness proof. An approximation also requires improvement. First of all, we should support cycles in input graph for abstract lexing and abstract parsing. The composition of two FST is also FST [17]. So string operations which can be described with FST (replace, trim, substring, remove, etc.) can be supported naturally at abstract lexer step.

The next big task is error detection and notification improvement. The main direction of research here is a migration to abstract parsing based on GLL parsing algorithm. Motivation is that the quality of error processing is higher in LL-parsers than in LR and implementation is simpler. If this characteristics are inherited in abstract GLL [3], then the error processing in abstract parsing will be improved. We are also going to implement type checking for string-embedded languages. For SQL it may be type checking inside the dynamically constructed queries and the consistency of returned and expected types: we should check whether

the type of query result equals to the type of the variable to which the result is assigned.

## 6. REFERENCES

- [1] Annamaa A., Breslav A., Kabanov J. e.a. An Interactive Tool for Analyzing Embedded SQL Queries. Programming Languages and Systems. LNCS, vol. 6461. Springer: Berlin; Heidelberg, p. 131–138, 2010.
- [2] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt. Abstract LR-parsing, In Formal modeling, Gul Agha, José Meseguer, and Olivier Danvy (Eds.). Springer-Verlag, Berlin, Heidelberg, p. 90–109, 2011.
- [3] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing, Electron. Notes Theor. Comput. Sci. 253, 7 (September 2010), p. 177–189.
- [4] Rekers J. G. 1992. Parser generation for interactive environments. Ph.D. thesis, University of Amsterdam.
- [5] Iakov Kirilenko, Semen Grigorev, and Dmitriy Avdiukhin. Syntax analyzers development in automated reengineering of informational system. St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems, 174(3), June 2013.
- [6] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In Proceedings of the 14th International Conference on World Wide Web, WWW '05, p. 432–441, New York, NY, USA, 2005. ACM.
- [7] Elizabeth Scott and Adrian Johnstone. Right nulled glr parsers. ACM Trans. Program. Lang. Syst., 28(4):p. 577 – 618, July 2006.
- [8] Aske Simon Christensen, Møller A., Michael I. Schwartzbach. Precise analysis of string expressions, Proc. 10th International Static Analysis Symposium (SAS), Vol. 2694 of LNCS. Springer-Verlag: Berlin; Heidelberg, June, p. 1 – 18, 2003.
- [9] Semen Grigorev. Automated transformation of dynamic sql queries in information system reengineering. Master's thesis, Saint-Petersburg State University, 2012.
- [10] Dmitry Avdyukhin. Translation definition language for informational system reengineering tools. Graduation thesis, Saint-Petersburg State University, 2013.
- [11] Giorgios Robert Economopoulos. Generalised LR parsing algorithms. 2006.
- [12] Tomita, Masaru. LR parsers for natural languages, In 10th International Conference on Computational Linguistics, p. 354–357. ACL, 1984.
- [13] Syme D., Granicz A., and Cisternino A.: Expert F#, Apress (2007).
- [14] Semen Grigorev and Iakov Kirilenko. 2013. GLR-based abstract parsing. In Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '13). ACM, New York, NY, USA, Article 5, 9 pages.
- [15] Andrey Terekhov. 2013. Good technology makes the difficult task easy. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). ACM, New York, NY, USA, P. 683–686.
- [16] D. Yu. Boulychev, D. V. Koznov, and Andrey A. Terekhov. 2002. On Project-Specific Languages and

Their Application in Reengineering. In Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR '02). IEEE Computer Society, Washington, DC, USA, p. 177–185.

- [17] Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Comput. Linguist.* 23, 2 (June 1997), p. 269–311.