

GLR-based Abstract Parsing*

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

Iakov Kirilenko
St. Petersburg State University
198504, Universitetsky prospekt 28
Peterhof, St. Petersburg, Russia.
jake@math.spbu.ru

ABSTRACT

Abstract parsing is an important step of the processing of dynamically constructed statements or string-embedded languages (such as embedded or dynamic SQL). Existing LALR-based algorithms have performance issues. To increase performance we propose to use a GLR-algorithm as a base for abstract parsing and to reuse graph-structured stack and shared packed parse forest. RNGLR-algorithm modification for abstract parsing is presented.

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*

General Terms

Algorithms, Parsing

Keywords

Dynamic SQL, GLR, RNGLR, abstract parsing, embedded languages, string-embedded languages, two-staged languages, injected languages, abstract translation

1. INTRODUCTION

*Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

CEE-SECR '13, October 23 - 25 2013, Moscow, Russian Federation Copyright 2013 ACM 978-1-4503-2641-4/13/10\$15.00.

Abstract parsing [7] is an important step of solutions for dynamically generated statements processing such as static errors checking, translation, and string-embedded languages support in IDE. Practice experience of abstract analysis implementation as part of dynamic SQL [1] translation shows that exponential resources are required [9] if semantic calculation is necessary. This fact often makes it impossible to process the code of real-world systems. In order to improve analysis performance we have suggested the solution based on dynamic filtration of the parsing results¹ However, this solution has some critical problems with semantic calculation correctness, besides, performance improvements is not enough for interactive processing in IDE.

Most abstract parsing implementation – such as PHP String Analyzer [12] or Alvor² [4, 2] – are based on LR or GLR parsing algorithm [10] and require additional data structures to store state data which lead to a big memory consumption in case of supporting semantics. It is possible to improve performance and to reduce memory consumption by using GLR-algorithm, extended with support of branching in the input graph, as the base for algorithm of analysis. We can also achieve this aim by means of graph-structured stack (GSS) and the intermediate representation of parsing results reusing.

In this work we present abstract parsing algorithm based on modified RNGLR-algorithm [13] and the generator of the corresponding parsers. We also describe the main points of the algorithm and the basic aspects of the implementation. Described algorithm is implemented as a part of YaccConstructor³ [11] project – an open source framework for research and development of parser generators, compiler-compilers and other grammarware for .NET. In addition to this, we present some examples which demonstrate dynamically generated statements processing using suggested algorithm.

2. RELATED WORK

Most practical tasks require not just parsing, which check whether input is in specified language, but also the ability of various parsing results transformations. There are abstract parsing and abstract lexing for string-embedded languages processing [7]. But these algorithms are not applicable for

¹Solution was presented at the CEE-SECR 2012.

²Alvor project site: <http://code.google.com/p/alvor/>

³YaccConstructor project site:
<http://recursive-ascent.googlecode.com>

transformation as they skip information about the initial source code positions of the tokens produced from embedded statements, which makes it impossible to modify the original code. Moreover parsing does not include semantic calculation which is required for translation and number of other transformations. However this kind of parsing can be useful for the errors checking or improvements of the code analysis of the whole system.

Dynamically constructed queries in the real-world systems have a high complexity so it is very important to improve analysis performance. Implementation of query constructing may contains more than dozens of assignments and conditional branchings [9]. Generally, the algorithm directly based on abstract parsing require an exponential resources. Algorithm for dynamic filtration of syntax analysis results was proposed in work [9]. The approach described in it can significantly reduce forest size for complex queries. But suggested algorithm has a number of problems with semantic calculation correctness. Moreover not all of the performance issues were solved.

2.1 Abstract Parsing Algorithm

Abstract parsing is one of the algorithms for syntax analysis of such generalized representation of strings set as data-flow equation or regular expression [7]. For practical use we often suppose that the compact representation of the set being processed is a graph. Each edge of this graph contains a token and each vertex corresponds to the concatenation of statement parts. The following example illustrates it. Here you can see the code sample in which some dynamic query is constructed:

```
(1) IF @X = @Y
(2) SET @TABLE = '#tbl1'
(3) ELSE
(4) SET @TABLE = 'tbl2'
(5) SET @S = 'SELECT x FROM ' + @TABLE
(6) EXECUTE (@S)
```

Variable @S contains dynamically generated query and has 2 possible values at the point of query execution. During approximation we can get a graph which presents the set of the possible values of the variable @S at the line 6. This graph is presented in figure 1.

Abstract parsing is based on the idea of reusing of the control structures utilized in the classical parsing with the implementation of special mechanism of their interpretation. Control tables for the analyser (action, goto) may be generated by the language specification by using some standard tool i.e. Yacc. LR automaton [10] should be modified so that it is able to compute all possible parser states for each vertex of the graph [7]. So, the basic idea of the abstract parsing is a graph processing with fix-point calculation [4].

Suppose that the grammar of language is the following:

```
s -> Ae
e -> BD
e -> CD
```

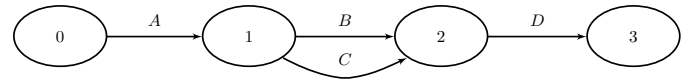


Figure 2: Input graph for abstract parsing.

Input graph for analyzer, which was generated by the specified grammar, is presented in the figure 2.

During syntax analysis the set of the parser states should be calculated for each vertex of the graph. States calculation result is shown in figure 3.

2.2 Tools for Static Analysis of Dynamic Statements

Many tools use dynamically generated queries for communicating with databases. This leads to the fact that safety assurance and quality of work for real-world systems is very difficult. So static processing of dynamically generated strings is an important problem. One of the most popular area is string-embedded SQL processing: static errors detection, SQL injections vulnerabilities detection [8]. Tools for Web-pages (HTML) generation also exist.

The following tools can be used for dynamically generated SQL queries processing:

- Java String Analyzer (JSA) [5] is a tool for analyzing the flow of strings and string operations in Java programs. For each string expression, it computes a finite-state automaton that provides an upper approximation of the values that may appear at runtime.
- Alvor [2] is an Eclipse plug-in, which statically validates SQL sentences embedded in Java code. It can be used either as one-shot full-program analyzer or as incremental as-you-type error guard. SQL strings found in the code can be checked against built-in SQL grammar or against actual test database.
- PHP String Analyzer (Phasa)⁴ [12] is a static program analyzer that approximates the string output of a PHP program with a context-free grammar. The analyzer can be used to check properties of a PHP program. For example, it can be used to validate dynamically generated Web pages by a PHP program.
- SAFELI [8] is a static analysis framework for identifying SQL injection attack vulnerabilities at compile time. SAFELI statically inspects MSIL bytecode of an ASP.NET Web application.
- A Static Analysis Framework for Database Applications [6] is a framework which can analyze database application binaries that use ADO.NET data access APIs. It can be used for a variety of analysis tasks such as SQL injection detection, workload extraction, identifying performance problems, and verifying data integrity constraints in the application.

⁴PHP String Analyzer project site:
<http://www.score.is.tsukuba.ac.jp/~minamide/phpsa/>

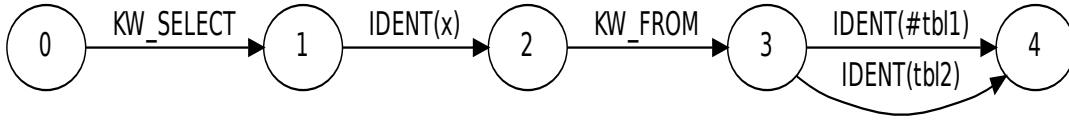


Figure 1: Graph for dynamic query.

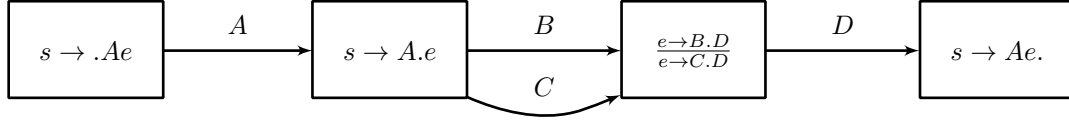


Figure 3: Parser states for graph in figure 2.

Most of tools use LR-based abstract parsing as a core of the algorithm. Alvor uses some ideas from GLR-algorithm for stack organization. However, some additional data structures are required during parsing. Also note that none of these tools does not solve the task of dynamically generated statements translation.

3. MODIFICATIONS OF GLR-ALGORITHM

The generator of the parsers based on RNGLR-algorithm has been implemented as a part of YaccConstructor [3]. RNGLR-algorithm is a modification of the classical GLR-algorithm which allow to process arbitrary context-free grammars. This implementation has an important feature of parsing performed in two steps: the first is to construct the lightweight internal graph-structured results representation (shared packed parse forest, SPPF [13]) and the second is to calculate user-defined semantics. This allows skip the calculations which may be unnecessary. For example, calculations in the erroneous branch of stack.

The implementation allows to reduce required memory and to increase performance of abstract parsing by making some calculations directly from the parser result graph during the tools development. If only the data about statement correctness are required then trees construction is unnecessary. In case of specific tools development when performance is critical or reusing of the algorithms, which were developed for tree (not graph) processing, is not required we can calculate all necessary information directly by SPPF.

If translation or other transformations are required then the main goal is to calculate new values for all the variables which construct a dynamic statement so that all the statements in the target system are also correct. For most tasks it is enough to build only a minimal subset of correct trees containing all the variables, for which new values should be calculated, instead of full forest construction. By implementing lazy tree generation from the graph, we can receive correct results as long as it is necessary without the calculation of the full forest.

Classical RNGLR-algorithm operates with two types of conflicts: Shift/Reduce and Reduce/Reduce which occur when there is some ambiguity of the next action to do. By analogy with them we suggest adding new type of “conflict” – Shift/Shift – corresponding to the branchings in the input graph. Shift/Shift is not an actual conflict, but should be

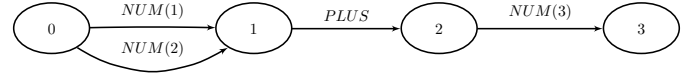


Figure 4: Graph with branching.

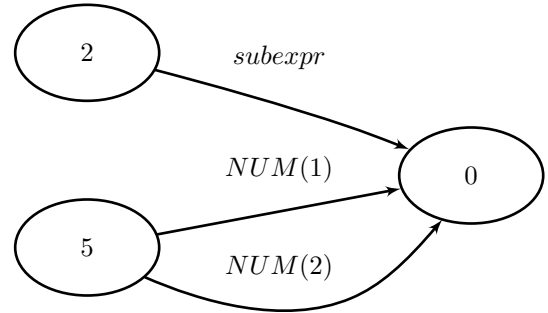


Figure 5: GSS after reduction by the rule $subexpr \rightarrow NUM$.

processed likewise one of them.

Let us clarify the point by the following example. The grammar under consideration is presented below. The input graph to be processed is in figure 4. There is branching in this graph and the Shift/Shift “conflict” corresponds to it. The GSS state for the input graph after reduction by the second rule of the grammar is shown in figure 5.

```
[<Start>]
expr: subexpr PLUS NUM
subexpr: NUM
```

It is important that we suppose that input data structure for parser is DAG with one source and one sink vertices. Our experience of dynamic SQL translation for some real information system shows that DAG is a good approximation of the dynamically constructed expressions set for practical use. We should replace all cycles with single repetition during approximation to get such graph. This way we can process all vertices in the topological order⁵. On each step

⁵Topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge $u \rightarrow v$ from vertex u to vertex v , u comes before v in the ordering.

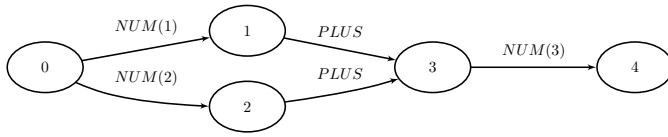


Figure 6: Input graph with branching.

we should process all outgoing edges for current vertex that leads to Shift/Shift “conflicts” appearance. As a result we have additional branches in GSS.

We now describe steps of parsing in more details. The grammar, which we use as an example, is shown below.

s: NUM PLUS e
e: NUM

Input graph is presented in figure 6.

There are two paths from the start vertex to the final in the input graph: $(0 \rightarrow 1 \rightarrow 3 \rightarrow 4)$ and $(0 \rightarrow 2 \rightarrow 3 \rightarrow 4)$. This paths contain a common subpath $(3 \rightarrow 4)$ which can be reduced to nonterminal **X**. So we should get SPPF where nodes for nonterminal **X** are common for two trees.

In this part we describe steps of syntax analysis. Vertices of the input graph are processed in the order of topological sort. In one step of the algorithm all outgoing edges of each vertex should be processed. However we only move to stack the edges which are required for the next in order vertex processing. So in case of multiple outgoing edges the algorithm processes only the edge with the tail to be processed the next. The processing of other edges should be postponed. The fact, that each edge will be processed just before processing of the vertex which is the target for this edge, allows to avoid incorrect merging of identical states. To clarify this we describe stack transformations in greater details.

During processing of the first vertex with number 0 (figure 6 and table 1.A) we have branching in the input graph and Shift/Shift “conflict”. Vertex with number 1 is the next vertex in topological order and only token assigned to the edge $(0 \rightarrow 1)$ is pushed to stack (table 1.B). So, at first the edges from subpath $(0 \rightarrow 1 \rightarrow 3)$ is processed, and only after that the algorithm processes the edge $(0 \rightarrow 2)$. Because of impossibility of reduction by any rule, the tokens from the path $(0 \rightarrow 1 \rightarrow 3)$ are just pushed into the stack (table 1.C).

After that all edges from path $(0 \rightarrow 2 \rightarrow 3)$ are processed. In this step a second branch in the stack is arisen. RNGLR-algorithm operates with some property of the vertex called level which determines the order of processing. We redefine the level of the vertex as its number in the topological order. The analyzer merges only the branches tails which have equivalent states and which are on the same level. The last condition guarantees that we avoid false merging of the vertices having equal states but being placed in different paths of an input graph. In our example we should merge the tails of the branches and must not merge the vertices 1 and 2

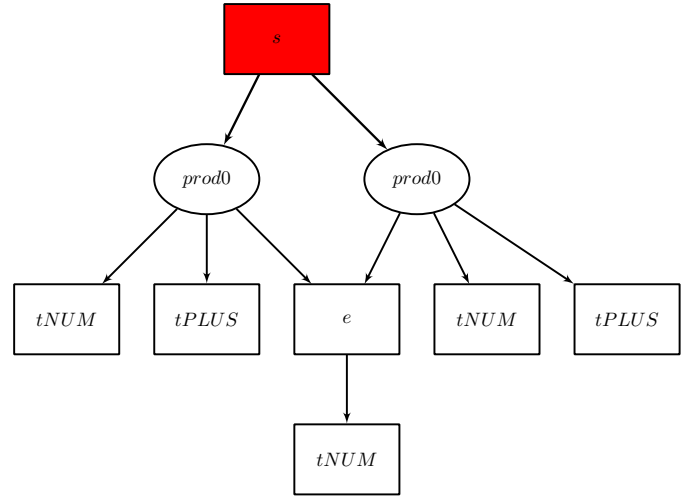


Figure 7: Parsing result.

(table 1.D and table 1.E). After that, the rest of the graph is processed (table 1.F – table 1.H).

The reductions to the nonterminals *e* and *s* are the last steps in the parsing. SPPF construction process is the same as in classical RNGLR.

The resulting SPPF contains two trees corresponding with two paths of the input graph and the subgraph for the non-terminal **X** is reused (figure 7).

4. EVALUATION

Proposed algorithm of abstract parsing and corresponded parsers generator is implemented as part of YaccConstructor project. Generator allows to create abstract parser by grammar specification. It is implemented as one of modules of YaccConstructor and based on RNGLR parsers generator which has been implemented previously. A big number of core modules and data structures is reused. So it is possible to generate syntax analyzers for string-embedded languages by the language specification written in one of supported languages such as Yard or FsYacc.


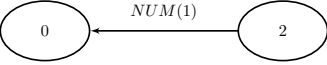
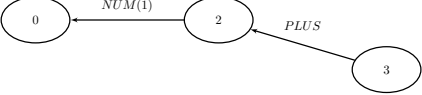
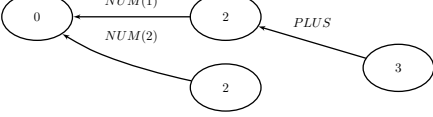
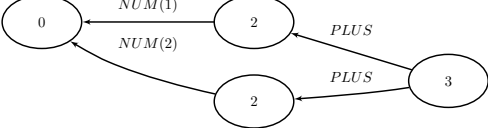
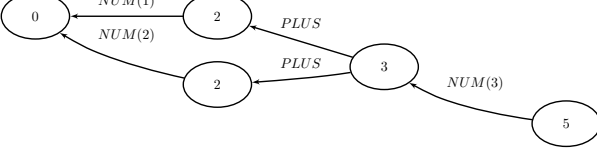
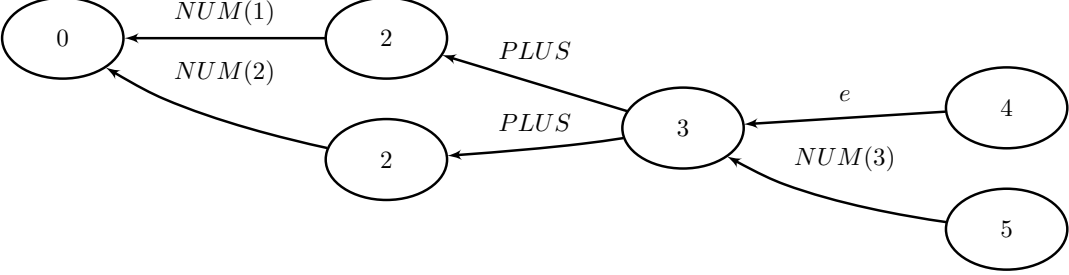
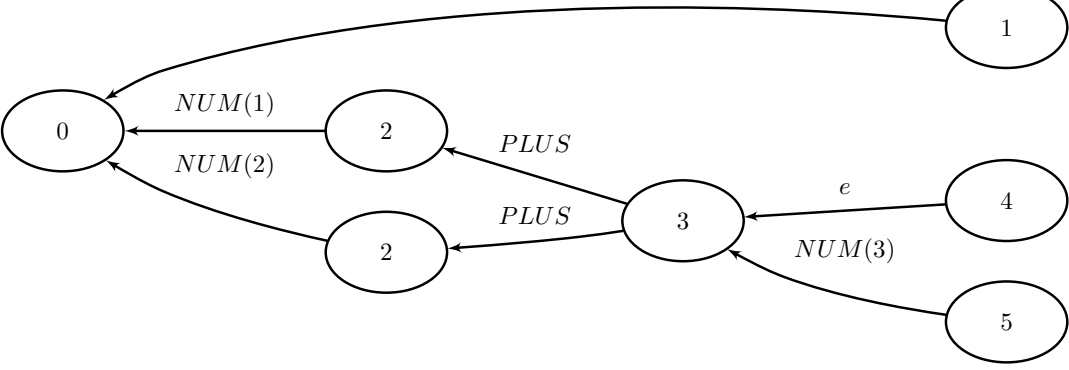
In order to demonstrate the implemented tool let us consider the following grammar:

```
binExpr<operand binOp>:operand (binOp operand)*
```

```
[<Start>]
expr: binExpr<term termOp>
termOp: PLUS | MINUS
term: res=binExpr<factor factorOp>
factorOp: MULT | DIV
factor: binExpr<powExpr powOp>
powOp: POW
powExpr: n=NUMBER | LBRACE expr RBRACE
```

The graph shown in figure 8 is used as an input for the syntax analysis.

Table 1: States of Parsing Statck

 <p>A. Start state.</p>	 <p>B. First shift.</p>
 <p>C. First branch processing is finished.</p>	 <p>D. Second branch processing.</p>
 <p>E. Second branch processing is finished.</p>	 <p>F. Common "tail" processing.</p>
 <p>G. Reduction to nonterminal e.</p>	
 <p>H. Reduction to nonterminal s.</p>	

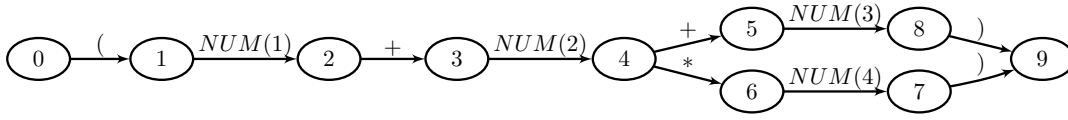


Figure 8: Input graph.

The graph shown in figure 9 is a result of the syntax analysis. As you can see, the subtrees corresponded to the common subexpression are re-used. This approach allows to reduce memory consumption in case of large input graphs.

We also have tested the tool as a parser of the T-SQL language subset. The results of the set of performance tests are provided below.

All tests were performed on a PC with following characteristics:

- OS Name: Microsoft Windows 7 Professional
- Version: 6.1.7601 Service Pack 1 Build 7601
- System Type: x64-based PC
- CPU: Intel(R) Core(TM) i5-2400S CPU @ 2.50GHz, 3201 Mhz, 4 Core(s), 4 Logical Processor(s)
- RAM: 16 GB

Input data for tests were generated to demonstrate the main cases previously extracted from real-world information system:

1. Fragments with parallel branches (produced by case-statements or if-statements) sequentially concatenated to a single statement. The most popular example is a select-statement where the name of each field may be calculated within if-statement or case-statement.
2. Nested branches being used for query construction.

An input data for the first case were based on select statement with common prefix and suffix, and fields list which was combined of sequentially concatenated basic blocks with parallel branches. Let n is a number of parallel paths in a basic block and m is a number of blocks. So the number of path in graph is n^m . We performed the set of tests for $n \in \{1; 2; 5; 10\}$. For each n the set of graphs with $m \in \{1..52\}$ was generated. The common structure of a graph for $n = N$ and $m = M$ is presented in figure 11. The example of graph for $n = 2$ and $m = 2$ is presented in figure 10.

So we have a five families of graphs parameterized by n . For each family we measure the parsing speed as a function of the number of sequential blocks m . Results of tests show that parsing require polynomial time as demonstrated in figure 13 and figure 12).

For the second case (which illustrated conditional branching in a query construction) tests were constructed from recursively nested blocks. All tests had the common prefix

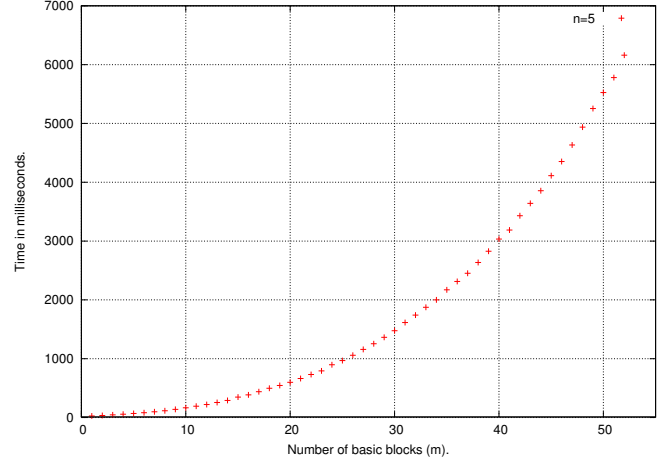


Figure 12: Plotted results of performance tests for case 1, $n=5$.

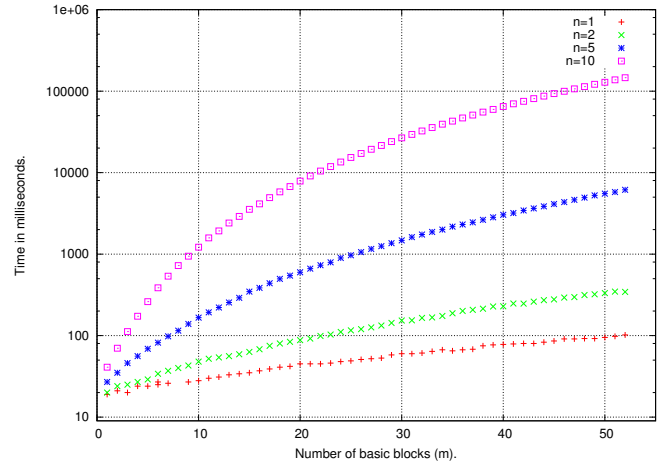


Figure 13: Plotted results of performance tests for all series in case 1.

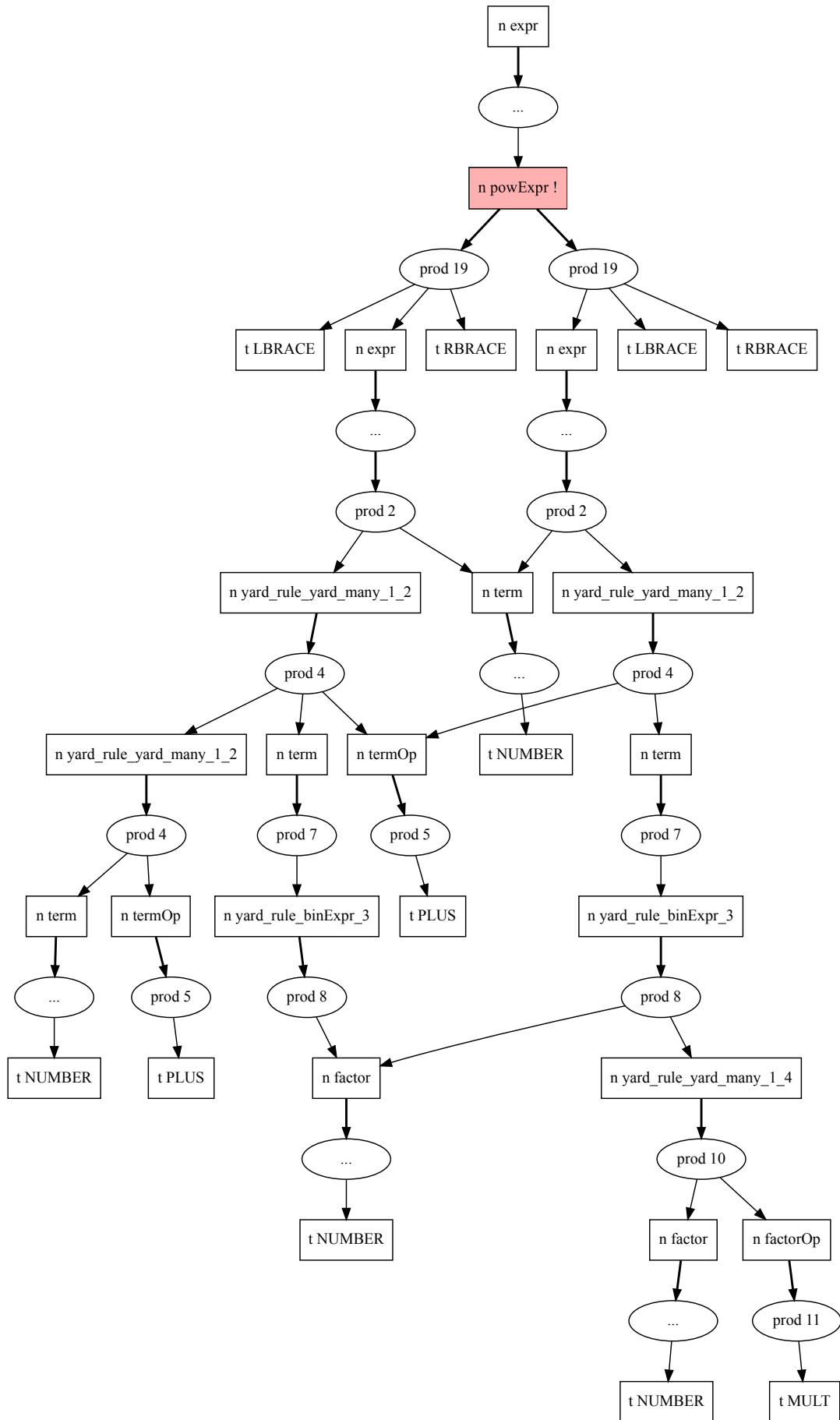


Figure 9: Result of the processing of the graph which presented in figure 8.

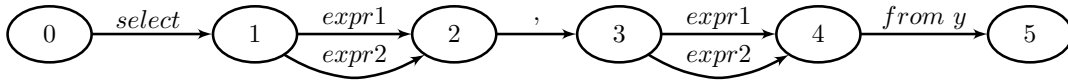


Figure 10: Example of graph with $n = 2$ and $m = 2$

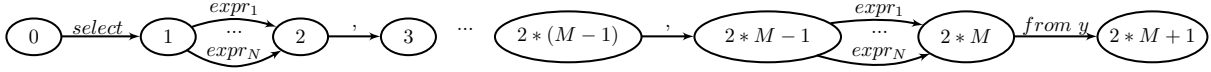


Figure 11: Common structure of graph with $n = N$ and $m = M$

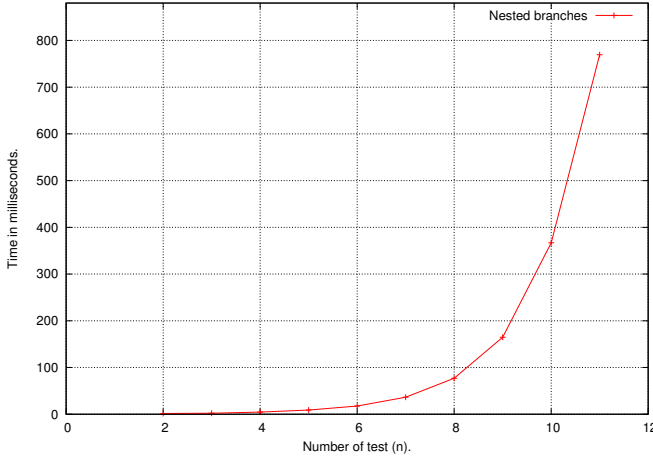


Figure 15: Plotted results of performance tests for case 2.

SELECT, the common suffix FORM Y, and the list of fields for select which were constructed recursively from basic blocks which contained two vertex and two parallel paths. Let n is a number of test and it specifies the depth of recursion. The list of fields can be generated by the specified number with the function described below.

```
let rec nestedBranches n =
  if n = 1
  then basicBlock
  else
    cond ( expr1 + comma
          + nestedBranches (n-1) + expr2)
          ( expr3 + nestedBranches (n-1)
            + comma + expr4 )
```

The example of an input graph with $n = 2$ is presented in figure 14.

As a result of the experiments we get that parsing require exponential time. Plotted results of measurements are presented in figure 15.

5. CONCLUSIONS AND FUTURE WORK

We created GLR-based algorithm for abstract parsing which reuses graph-structured stack and shared packed parse forest and implemented abstract parsers generator based on

this algorithm. We also carried out a series of experiments showing the practical applicability of the generator and the correctness of the implemented modified RNGLR-analyzer.

Plugin for ReSharper⁶ based on described algorithm is in active development. This plugin is designed for support of string-embedded languages in Microsoft Visual Studio IDE. We are also working on static errors checking in dynamically constructed statements at the current time. The next step is the implementation of the other features of IDE for string-embedded languages such as syntax highlighting, autocompletion and refactorings. Plugin development poses many new questions associated with such problems as incremental regular approximation construction, semantic calculation directly by graph and results caching. Unified modular infrastructure for fast and easy support of new languages is in active development.

In the future, errors handling algorithm should be enhanced and necessity of errors recovery should be investigated. The fact that input graph may contains a huge number of “incorrect-by-design” paths makes these questions a bit difficult to answer. It is caused by the fear that errors recovery may produce a big number of inconsistent results.

We also should implement efficient generation of parsing trees from SPPF and user-defined semantic calculation for them. Parallel processing may be used for these tasks but the real usefulness of this approach is a question to be answered by running on the real-world examples. The ambiguity here is caused by the complexity of the data structures for processing.

One more question is whether it is possible to generalize classical parsing to abstract parsing. Experiments have shown that a generalization is possible but the current implementation causes big performance issues. Generalized parsing performance is almost 10 times lower than the performance of the classical parsing based on the same algorithm. This may be related to the ineffective implementation of the generalized algorithm or the theoretical limitations. This problem requires an additional investigation.

6. REFERENCES

- [1] ISO. ISO/IEC 9075:1992. Information technology — database languages — sql, 1992.

⁶ReSharper – developer productivity tool for Microsoft Visual Studio. It’s provide powerful code inspections, automated code refactorings, blazing fast navigation, and coding assistance. <http://www.jetbrains.com/resharper/>

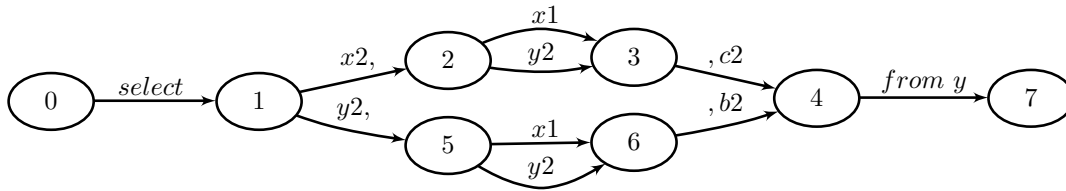


Figure 14: Example of graph for nested branches with $n = 2$.

- [2] Aivar Annamaa, Andrey Breslav, Jevgeni Kabanov, and Varmo Vene. An interactive tool for analyzing embedded sql queries. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, APLAS'10, pages 131–138, Berlin, Heidelberg, 2010. Springer-Verlag.
- [3] Dmitriy Avdiukhin. Implementantion of glr parsers generator for .net. Course work, 2012.
- [4] Andrey Breslav, Aivar Annamaa, and Varmo Vene. Using abstract lexical analysis and parsing to detect errors in string-embedded dsl statements. In *Proceedings of the 22nd Nordic Workshop on Programming Theory*, TUCS General Publication, pages 20–22. Turku Centre for Computer Science, 2010.
- [5] AskeSimon Christensen, Anders MÅyler, and MichaelI. Schwartzbach. Precise analysis of string expressions. In Radhia Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2003.
- [6] Arjun Dasgupta, Vivek Narasayya, and Manoj Syamala. A static analysis framework for database applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1403–1414, Washington, DC, USA, 2009. IEEE Computer Society.
- [7] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt. Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 256–272, Berlin, Heidelberg, 2009. Springer-Verlag.
- [8] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 87–96, Washington, DC, USA, 2007. IEEE Computer Society.
- [9] Semen Grigorev. Automated transformation of dynamic sql queries in information system reengineering. Master's thesis, Saint-Petersburg State University, 2012.
- [10] Dick Grune. *Parsing Techniques: A Practical Guide*. Springer Publishing Company, Incorporated, 2nd edition, 2010.
- [11] 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.
- [12] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 432–441, New York, NY, USA, 2005. ACM.
- [13] Elizabeth Scott and Adrian Johnstone. Right nulled glr parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, July 2006.