Adaptation of Generalized LR Parsing Algorithm for String-embedded Languages Processing

-

-

-

Abstract

Direct execution of dynamically generated strings can be used in order to compose general purpose programming languages and domain specific languages. This approach is still in common use despite the popularity of generative programming techniques. There is an abstract parsing algorithm proposed by Kyung-Goo Doh, Hyunha Kim, David A. Schmidt to process such string-embedded languages. This algorithm based on LR(k) parsing algorithm and allows to analyze data-flow equations which approximate the set of possible values of string expressions generated by program.

We propose to use RNGLR algorithm as a base of abstract parsing. RNGLR was described by Elizabeth Scott and Adrian Johnstone. This way we can handle ambiguous context free grammars in abstract parsing. Moreover RNGLR-based abstract parsing allows to use classical GLR data structures, such as Graph Structured Stack (GSS) and Shared Packed Parsing Forest (SPPF) to handle multiple stacks and parsing trees in optimal way. We describe changes in GSS manipulation and SPPF construction in order to adapt classic GLR algorithm for abstract parsing purposes. Also we provide results of evaluation of implemented algorithm.

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 GLR, RNGLR, abstract parsing, string-embedded language, injected language, dynamic SQL, static analysis, string analysis

1. Introduction

Object Relational Mapping technologies, generative programming techniques, metaprogramming are being actively developed and becoming in greater use. Despite that, direct execution of dynamically generated strings for communicating with databases or generating web-pages is still in common use. This approach allows to use expressivity of domain specific languages in general-purpose language code.

POPL '15, Month d–d, 2015, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Expression constructed from strings in runtime of some program is named **dynamically generated expression** or **dynamic expression**; the program constructing it is named **host**; the language of this expression is named **injected** or **string-embedded language**. The following example demonstrates SQL injected into C# code.

```
1  class EmbeddedSQL
2  {
3    static void ConstructAndExecuteQuery(bool cond)
4    {
5      var query = "SELECT";
6      if (cond)
7        query = query + " *";
8      else
9        query = query + " column1, column3";
10     query = query + " FROM table1";
11     Program.Execute(query);
12   }
13 }
```

Actually, injected language expressions are ordinary string variables in the host language and perform the code in the other programming language. So they can contain typical errors. For example, some errors can be made during development, and can be detected only in runtime. It is also necessary to support and modify systems which use dynamically generated expressions. In this case standard features of an Integrated Development Environment like code highlighting, renaming and other types of refactoring can be really helpful. Translation tasks can also arise such as migration of a system which uses injected languages for communicating with database from one database management system to another. But the majority of standard instruments considers dynamically generated expressions as simple string expressions and provides none of the listed functionality.

All the mentioned features cannot be provided without static syntactic analysis of dynamically generated expression or **static abstract syntactic analysis** [3]. This term is stand for performing an analysis of the set of expressions which could be generated in runtime from string expressions of the host program without actual execution of the program. A set of dynamically generated expressions values can be very large if numerous conditional statements were used in the process of its construction. It even can become infinite in case of using the loops for dynamic expression construction. These circumstances make static abstract parsing far more complex in comparison with classical static analysis and make classical syntactic algorithms inapplicable to string-embedded languages processing.

There are several approaches for string-embedded languages processing. They are designed for different purposes and provide different analysis precision. One of the most powerful is compo-

sition of data-flow analysis and LR parsing introduced by Kyung-Goo Doh, Hyunha Kim, David A. Schmidt [4]. In this paper we propose the improvement of parsing part of this approach and describe parsing algorithm for string-embedded languages processing based on RNGLR [11] parsing algorithm. Parser generator based on described algorithm is presented. We also provide results of evaluation on subset of Transact SQL language, describe drawbacks and revealed issues of our algorithm, and discuss possible ways to solve them.

2. String-embedded Language Analysis

There are several approaches to handling injected languages [1, 4, 10, 12, 13, 15, 16]. One of them is based on the following idea: if there is specification G_1 for some language $L(G_1)$, and there is specification G_2 describing the language $L(G_2)$ of dynamically generated expressions, and $L(G_2) \subseteq L(G_1)$ then the dynamic expression values are correct sentences in the language $L(G_1)$. An algorithm implementing this idea takes as an input a reference grammar for injected language and a code fragment to be analysed. The language of dynamically generated expression is approximated with context free or regular grammar. Then it is checked whether the approximation is a specification for some subset of language described with the reference grammar. This idea is implemented in such tools as JSA[1] [12] and PHPSA[2] [10]. The approach is intended to be used for syntactical error detection. No parse trees are constructed during the analysis, thus it cannot be used for semantic calculation or transformation. Moreover, an error detection precision is rather low. The reason of this is the method can only determine whether the approximation is a subset of the referenced language and in case of negative result it is rather complicated to report an error in terms which are clear to user. One more disadvantage is the loss of precision in case of regular approximation as the most of practically applicable languages are at least context free.

The second approach is named abstract parsing [4]. It is a syntactic analysis (i. e. based on LR-algorithm) of some compact representation of dynamically generated expression in a hotspot. There could be any type of compact representation: data-flow equation, regular expression or finite automaton generating possible values. This method uses standard *action* and *goto* tables, which are constructed by a specification of the language being analysed, and a modified interpreter (LR-automaton), which can process the compact representation of expression. The following describes the tools which use this approach.

Kyung-Goo Doh et al. [3–5] were first who introduced the term abstract analysis. The tool they developed analyses dynamically generated expressions during analysis of the host program. The problem of potentially infinite derivation is solved by stack reduction to remove repeating segments. Kyung-Goo Doh et al. use abstract interpretation [20] to do this. This approach allows to calculate semantics [5].

The last idea is separate the analysis into steps: dynamically generated expression value set approximation, lexical analysis, syntactic analysis, and further steps like semantic calculation or transformation. This approach is implemented in Alvor[3] [1, 2] — plugin for Eclipse IDE analysing SQL injected in Java. In this tool maximum stack depth is limited to prevent infinite derivation. The stack is branched likewise in GLR-analysis. However, stack branches are not merged back and this causes performance issues in case of big number of conditional expressions used to construct dynamic expression.

In our work we use step-by-step analysis to process dynamic expressions. We shortly describe steps of analysis with example. Suppose that approximation is finite automaton which can be represented as a graph with string labels on edges. Vertices are corresponded to concatenations used to construct an expression. For example, let we try to process the code below.

```
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 values of dynamically generated expression. There are two different values of `@S` in the line 6. *Approximation* of `@S` value set in line 6 can be represented as graph presented in the figure 1.

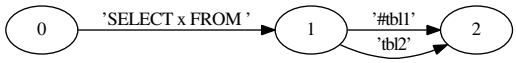


Figure 1: Approximation for code presented in example above

Next step — *abstract lexical analysis* — is a procedure which transforms input graph with string labels on edges into graph with token labels on edges. The result of input graph tokenization (or abstract lexing) is presented in the figure 2.

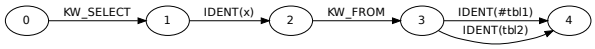


Figure 2: Result of tokenezation of graph in figure 1

Then we should perform *abstract parsing*. We use the second approach — reusing of classical LR parsing mechanism. Let we introduce the next example to explain basic ideas of abstract parsing. We will use the following grammar:

```
[<Start>]
s -> Ae
e -> BD | CD
```

Input for analyzer builded by given grammar is a graph presented in the figure 3.

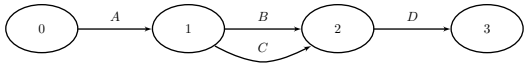


Figure 3: Input graph to processing

The set of parser states are computed for each graph vertice during parsing. All states that may be computed during parsing of all paths from start vertice to current should be computed for each vertice (figure 4). For example, there are two different states for vertice 2 in the figure 4: $\{e \to B.D; e \to C.D\}$. It is a result of sequences AB and AC parsing.

Since the base of abstract parsing is classical LR algorithm, it is possible to process attributed grammars and, as a result, calculate user semantic actions.

[1] JSA is a tool for Java string analysis. Project site (accessed: 01.07.2014):`http://www.brics.dk/JSA/`

[2] PHPSA is a tool for PHP string analysis. Project site (accessed: 01.07.2014): `http://www.score.cs.tsukuba.ac.jp/~minamide/phpsa/`

[3] Alvor is a tool for string-ambedded SQL in Java analysis. Project site (accessed: 01.07.2014):`https://code.google.com/p/alvor/`

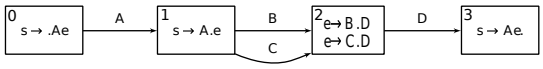


Figure 4: Parser states for graph presented in figure 3

3. Generalized LR parsing

Not all grammars are unambiguous but it is still necessary to process them, so there is a class of Generalized LR-analysis algorithms for ambiguous grammars handling. Classical GLR was introduced by Tomita [19]. It is based on classical LR analysis [8] and has the number of significant differences making it possible to process ambiguous grammars. The algorithm uses parse tables, that are similar to parse tables for classical LR-algorithm, but can contain more than one possible action in each cell. This situation is named conflict and one of the two possible types can occur: *Shift/Reduce* — it is possible either to shift one more token from input stream or to reduce the stack — and *Reduce/Reduce* — there are more than one possibility to reduce the stack. In addition, more complex data structure — **Graph Structured Stack, GSS** [19] — is used instead of common stack. Manipulation with GSS is based on breadth-first traversal, so there are levels corresponding to the token position in the input stream.

Since it is possible to get more than one result of parsing in case of processing of ambiguous grammar then a special structure that allows to reuse nodes in parsing forest is used. This structure is introduced by Reker [7] and named **Shared Packed Parsing Forest** or **SPPF**. SPPF is an inner data structure and it stores the forest data very compact. User defined semantic possibly requiring a lot of memory can be calculated after parsing finished only for necessary trees extracted from SPPF. This way we can reduce resource requirements. In our algorithm we also use classical GSS and SPPF as base data structures.

However, classical Tomita GLR constraints grammar. Elizabeth Scott et al. [11] proposed development of Tomita's idea — RNGLR-algorithm of syntactic analysis, which we use as a base for our algorithm. RNGLR-algorithm can handle arbitrary ambiguous grammar, like classical GLR-algorithm, and uses GSS and SPPF. The authors also introduced *push operation* instead of classical *shift* and *goto* which better correspond to adding new edges to GSS [11]. So **Push(k)** is an action noting that symbol k should be placed to stack. We will use this terminology for the rest part of the paper.

4. Abstract Generalized LR Parsing

4.1 Input Data Format

We approximate dynamic expression value set by constant propagation. Resulting data structure is a graph, which contains strings, used for expression construction, on its edges and its vertices correspond to concatenations. One might say we build finite automaton describing the language which consists of dynamically generated expression values.

If there is loops used in the construction of expression, value set may become infinite. We replace cycles in the graph with *single repetition* of its body for now. So the input data structure for our algorithm is **Direct Acyclic Graph (DAG)** with only one source and sink vertices. This guarantee the finiteness of value set, but adversely affect on the reliability of the results since this replacement can lead to both loss of some possible expression and generation of new expressions. Let consider the following example. Suppose we need to process the code below.

```
1 query = "select * from ";
2 for(int i = 0; i < tables.size(); i++)
3 {
4    if(i != 0) query += ", ";
5    query += tables.get(i);
6 }
7 query += ";";
```

Regular expression $select * from \cdot ((, |\varepsilon)\{tables.get(i)\}|\varepsilon)^\star \cdot ;$ is built as a regular approximation of set of possible values of variable `query`. Corresponding finite automaton is presented in the figure 5. Notice, that we do not provide the value of expression $\{tables.get(i)\}$ as it depends on constant propagation algorithm possibilities. In order to get rid of cycle we replace it with single repetition of its body. Graph presented in the figure 6 is the result of the replacement for our example.

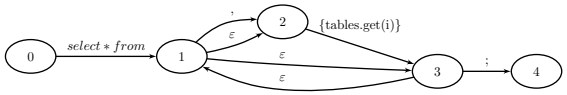


Figure 5: Correct finite automaton (graph representation)

Figure 6: Result of cycles approximation for graph in figure 5

As one can see, approximation graph does not contain paths corresponded to many values i.e. `select * from tbl1, tbl2;` and `select * from;`. This leads to impossibility of static check of theirs correctness. Only two possible values of initial infinite set can be generated from the approximation graph. Thus we cannot check all possible values in case of the loops that are used to construct queries. By the way, this approach provides enough data for such problems as code highlighting because all tokens which form expression will be processed.

4.2 Graph Structured Stack

As soon as input data structure for our algorithm is DAG we can process all vertices in topological order instead of least-fixed point calculation. We define function $T(v)$ as a function which returns the number in topological order for vertex v. If there are more than one possible order, we choose one of them and fix it.

Stack building in classical GLR is based on breadth-first traversal: firstly all actions with current level of stack should be performed, then if there are no available operation, pushes to the next level are performed. Operations in GLR stack are performed level-by-level. In RNGLR-algorithm levels correspond to input token number. We define level as a vertice number in topological order: this allows to avoid incorrect merging of stack branches. To process graph presented in the figure 7 we should process vertices sequentially in the next order: $0, 1, 2, 3, 4, 5, 6, 7$. In the figure 8 you can see the graph to process (in figure 7) the layout of which is changed according to level data.

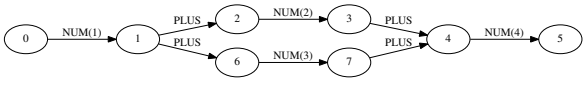


Figure 7: Input graph for abstract parser

Original GLR parsing algorithm can process *Shift/Reduce* and *Reduce/Reduce* conflicts — situations when available data are not

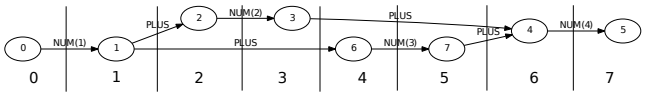


Figure 8: Input graph in figure 7 with topological ordering applied

enough to choose 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 name 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. We improve original parsing algorithm with *Shift/Shift* conflict processing mechanism and it will be described below in details.

GSS branch merging mechanism can be fully reused from GLR-algorithm. Vertices from graph are processed in the topological order. This ensures that states merging is possible only in top vertices corresponded to branches which are produced by classical LR conflicts or in vertices with more than one incoming edges. In both situations context and position in input data structure are the same for processed vertices. If parsing states for any two vertices are equal, they can be merged by the analogy with GLR algorithm. Note that not all vertices with more than one incoming edges should initiate stack branches merging.

The next difference from classical GSS manipulation is *push* operation introduced as a replacement of *goto* and *shift* operations. *Push* operation performs actual attachment of new edges to top vertices in GSS. In case of *Shift/Shift* conflict we should process every outgoing edge for the vertice being processed. But only one of the edges should be really added to GSS at a time: the edge e with the minimal end number in the topological order *T(e.ToVertex)*. Remaining pushes should be postponed and applied only when *T(e.ToVertex)* — where e is corresponded edge from input graph — is minimal for all vertices that are available to be processed during current step. This way we can avoid an early attachment of vertices, breaking topological order, to GSS. So, we do not perform all possible pushes like RNGLR and attach new edges only when it is necessary.

4.3 Example of GSS Construction

We introduce the next example to demonstrate GSS manipulation in abstract parsing. We consider the following grammar of arithmetic expressions. Only two binary operators are specified: $+$ and $*$ The list of tokens is presented below.

```
PLUS = +
MULT = *
LBR = (
RBR = )
NUM = [1..9]
```

And the final grammar is:

```
s -> s PLUS e | e
e -> e MULT t | t
t -> LBR s RBR
t -> NUM
```

Input graph to be processed is presented in the figure 9. This graph is ready for parsing: vertices are numbered in the topological order and edge labels are tokens. This graph is a representation of the following set of arithmetic expressions:

$$
\begin{array}{l}
\{ \\
1 + 4 * (7 * 8) \\
\quad [0 \to 1 \to 2 \to 3 \to 8 \to 9 \to 10 \to 11 \to 12 \to 13] \\
1 * 5 + (7 * 8) \\
\quad [0 \to 1 \to 4 \to 5 \to 8 \to 9 \to 10 \to 11 \to 12 \to 13] \\
1 + 6 + (7 * 8) \\
\quad [0 \to 1 \to 6 \to 7 \to 8 \to 9 \to 10 \to 11 \to 12 \to 13] \\
\}
\end{array}
$$

Now we can demonstrate stack transformations performed during analysis of the graph in the figure 9. At the beginning stack contains one start vertice with number 0. At the first step token `NUM` at the edge $0 \to 1$ is pushed. After that the sequence of reductions is performed. The reduction to nonterminal `s` is final. Note that all intermediate outcome should be preserved in stack because pushes to next level may be available for every produced state. In our example token MULT at the edge $1 \to 4$ can be pushed for state corresponded to vertex with $id = 1$ and token `PLUS` at the edges $1 \to 2$ and $1 \to 6$ can be pushed for state corresponded to vertex with $id = 18$. On the other hand, there are no pushes for vertices with $id = 21$ and $id = 17$, so this vertices and outgoing edges will be removed at the next step. Stack after described operations is presented in the figure 10.

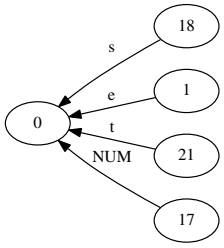


Figure 10: GSS after first reduction to nonterminal *s* is finished

At the next step we should process every outgoing edge for vertex with $id = 1$ in the input graph. This vertex has three outgoing edges, so pushes for all of them should be constructed. But only one push should be available for further processing: corresponded to the vertex with minimal number. Postponed pushes will be processed in order of its start vertex topological number. In our example each branch will be processed sequentially until vertex with $id = 8$ is processed. Then the next branch will be restored from postponed pushes and then processed. Stacks corresponded to processing of the paths $1 \to 2 \to 3 \to 8$ and $1 \to 4 \to 5 \to 8$ are presented in the figures 11–13. Note that in the figures 12–16 postponed pushes are not drawn.

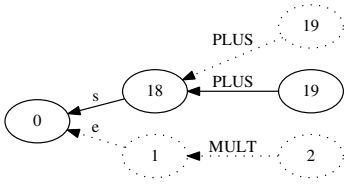


Figure 11: Stack with postponed pushes

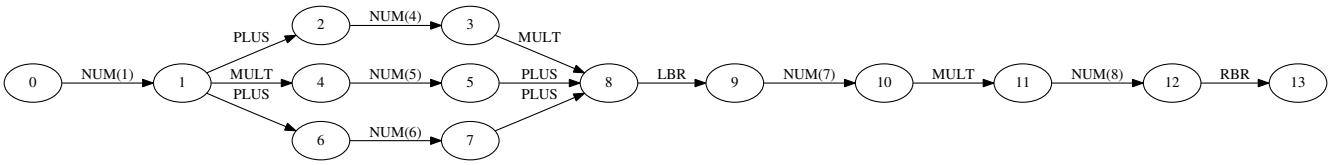


Figure 9: Input graph GSS construction example

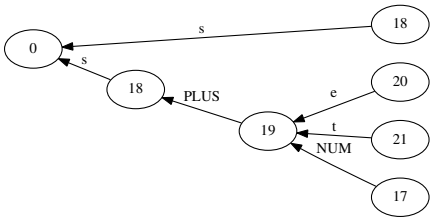


Figure 12: Stack corresponded to processing of the path $1 \to 2 \to 3$

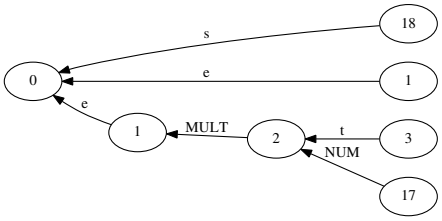


Figure 13: Stack corresponded to processing of the path $1 \to 4 \to 5$

The stack presented in the figure 14 is got when all edges incoming to vertex with $id = 8$ are processed. There are three incoming edges to vertex with $id = 8$ but only two top vertices in stack are merged. It is happened because only two of them correspond to identical parsing states. After that, when open brace form edge $8 \to 9$ in input graph is processed, all vertices corresponded to identical states and placed at the same level will be merged. Result of merge is presented in the figure 15.

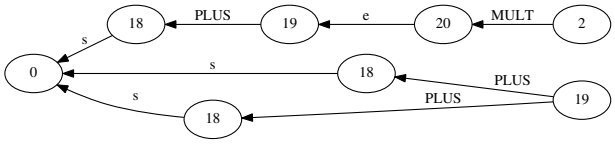


Figure 14: GSS after all incoming edges to vertex with $id = 8$ from the input graph are processed. Only two top vertices can be merged

Further changes in the stack are performed in the order that has been described above. Final state of GSS is presented in the figure 16.

We have demonstrated main steps of GSS construction: branching, pushes processing, branches merging. One can notice that it is similar to classical GLR GSS construction but differs in push processing.

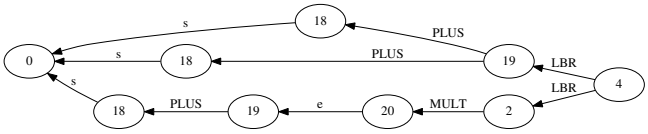


Figure 15: GSS after all incoming edges to vertex with $id = 9$ from the input graph are processed. All states are merged

4.4 Shared Packed Parsing Forest

Lightweight inner representation of parsing results — Shared Packed Parse Forest, SPPF — was proposed in a paper [7] in order to minimize the size of parsing forest. After parsing completion SPPF can be used for semantic calculation or transformations. User can pick single parsing tree and calculate semantics only for it that can help to get rid of redundant calculation. Part of the calculation can also become redundant if it took place in the erroneous stack branch. These features are very valuable for abstract analysis as the number of common nodes in parse trees is huge and such nodes can be reused among the trees that are corresponded to different paths in input graph. The number of mistakes can also be enormous and there is no need to calculate semantics for the whole parse forest. Use of SPPF can significantly speed the abstract analysis up and reduce memory consumption.

SPPF building in the process of abstract analysis does not differ from the one in RNGLR-algorithm. Intermediate nodes are introduced for correct representation of multiple possible parse trees for nonterminal that is processed. In our algorithm if there are several ways to construct node for nonterminal s, then additional vertices are created as children of this node and each of them is a root of parse tree for one of the possibilities. We paint nodes for such nonterminals in red in our figures. You can see root node s which has three different derivation (and thus three different parse trees merged into SPPF) in the figure 17. Child nodes of the node s with label $prod0$ are additional nodes. Their subtrees are corresponded to possible variants of s derivation.

As the process of SPPF construction is the same as in classic RNGLR-algorithm, we just provide the result for the example which has been described above.

As it is said before, sometimes there is no need to calculate the semantics for the whole parse forest, or the forest is too large. It is also possible that there are trees which are correct syntactically but not semantically. In these cases lazy tree generation helps to avoid unnecessary semantic calculation. This way one can retrieve trees from SPPF and calculate semantics for them one-by-one.

4.5 Parsing Tables

Abstract parsing uses original LR parsing tables. Kyung-Goo Doh, Hyunha Kim and David A. Schmid [3] use LALR(1) tables generated with ocamlyacc[4] in their work. We use RNGLR parsing algo-

[4] Parser generator for OCaml. Project site (accessed: 01.07.2014): `http://caml.inria.fr/pub/docs/manual-ocaml-400/manual026.html`

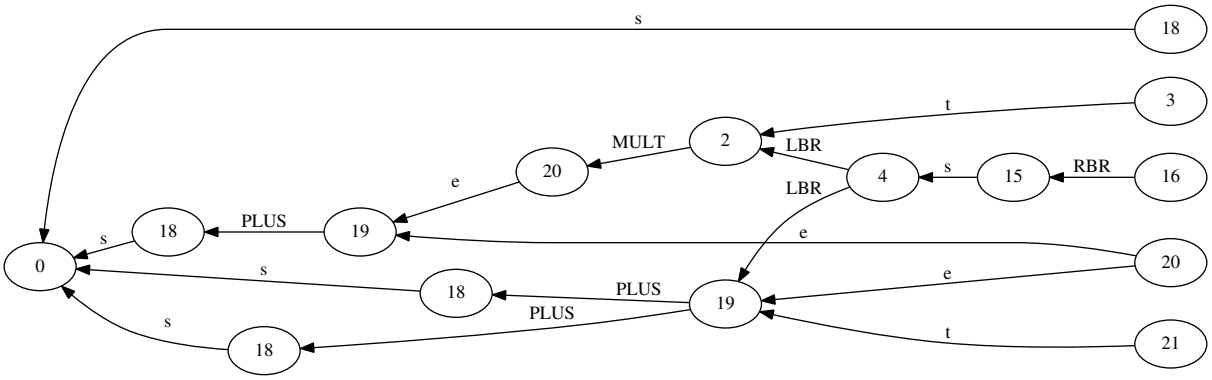


Figure 16: Final GSS state for input graph in figure 9

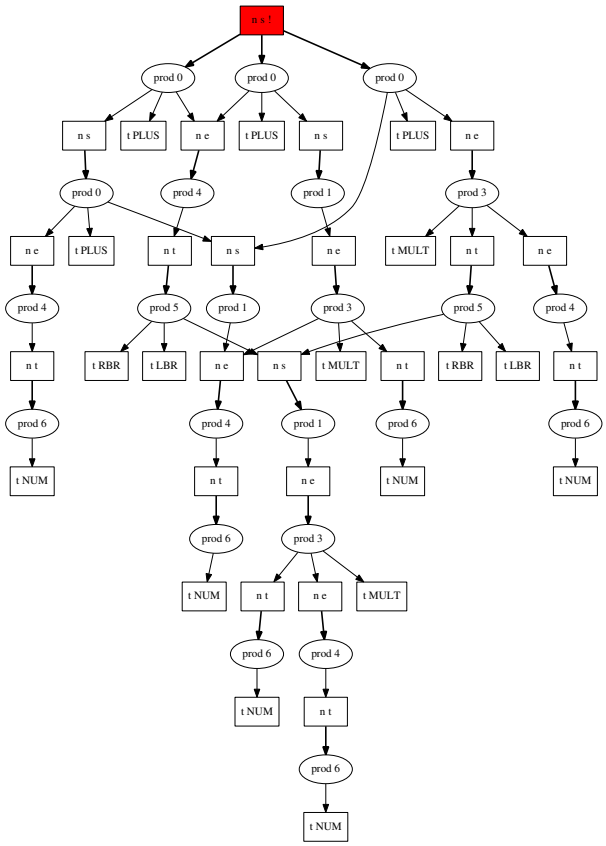


Figure 17: SPPF for input presented in figure 9.

rithm in our tool. RNGLR can process arbitrary context free grammars and has been proposed by Elizabeth Scott and Adrian Johnstone [11]. This algorithm has been implemented as a part of research project YaccConstructor[5] [9]. Different dialects of SQL with procedure extensions, JavaScript or other languages can play the role of embedded languages. Available specifications of this languages are often not LALR(1) and contain ambiguity. This makes it important to be able to process arbitrary context-free grammars in order to simplify development and support language processing tools. Therefore usage of GLR parsing algorithm instead of LALR(1) is better and our abstract parsing algorithm uses original RNGLR parsing tables.

5. Evaluation

Generator of abstract parsers based on described parsing algorithm was implemented in F# [21] programming language as a part of YaccConstructor. RNGLR parser generator used as a base of our algorithm was implemented before as a part of YaccConstructor. As we discussed before, the generator itself is reused without any changes. Table interpreter was modified, although main data structures and base logic were reused. One can use an implemented framework for generation of a tool designed for abstract parsing of the language that is specified in an input grammar. Thus the framework is similar to the classical parser generators. Abstract lexer generator and approximation builder were also implemented. General high-level structure of our tool is presented in the figure 18.

We has compared generated parser with similar tools to evaluate its performance. Only Alvor[6] is available for comparison at this moment. Tool developed by Kyung-Goo Doh et al. and described in their papers is also very important for comparison but neither sources nor binaries of the tool are available.

Grammar of the subset of Transact SQL language was developed for testing purposes. Grammar contains specification for procedure extension and Data Manipulation Language. The grammar developed is available in a project repository[7]. We aimed to test parser only so approximation step was omitted. An input data is a set of previously generated and serialized representations of dynamic expression approximations. We use our experience in production information system migration from MS-SQL Server 2005 to Oracle 11gR2 to generate relevant input data [17]. The source system consists of 850 stored procedures and contains more than 3000 dynamic queries. Total size of migrated system is 2.7 million lines of code. The number of operations used to construct values for more than half of queries is between 7 and 212, average is 40. We performed an information system source code analysis and figured out that the most frequent template for dynamic query building is sequential concatenation of blocks constructed with branching op-

[5] YaccConstructor is a platform for grammarware research and development. Project cite (accessed: 01.07.2014): `https://code.google.com/p/recursive-ascent/`

[6] Repository of Alvor (accessed: 01.07.2014) `https://code.google.com/p/alvor/`

[7] Repository of YaccConstructor (accessed: 01.07.2014) `https://code.google.com/p/recursive-ascent/`

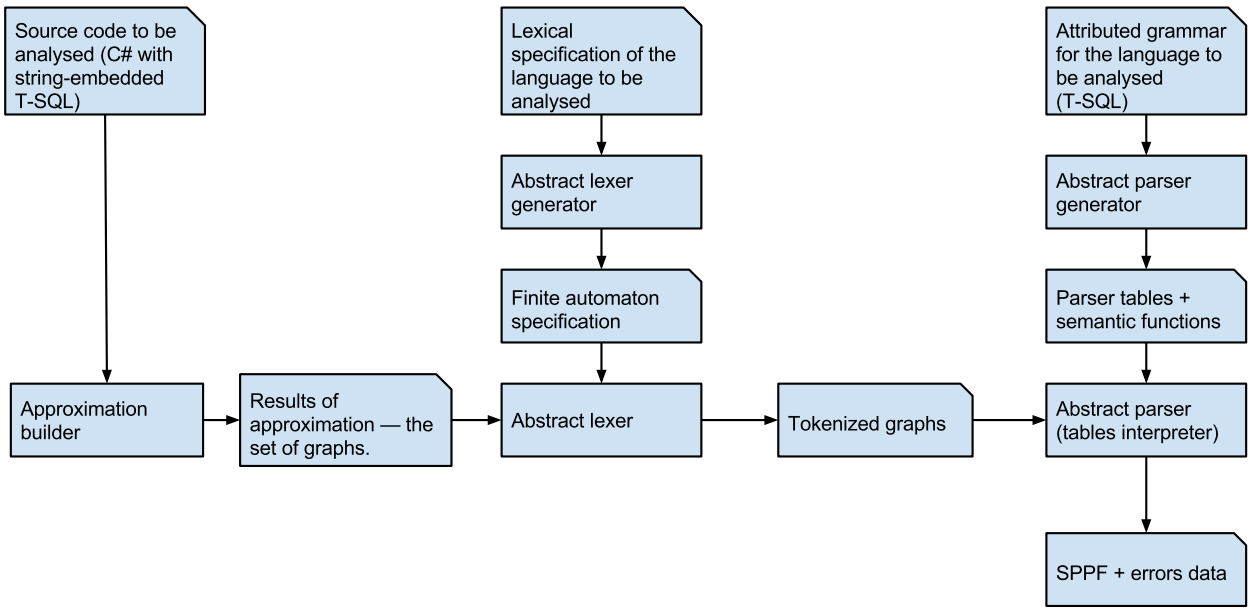


Figure 18: Abstract analysis tool structure

erators: *if* statement or *case* statement. Input data for tests were generated according to this result. Each test is described by two parameters: the number of parallel edges in each block m, and the number of sequentially concatenated blocks n. All blocks in one sequence have equal number of parallel edges. Example of graph with $m = 3$ and $n = 2$ is presented in the figure 19.

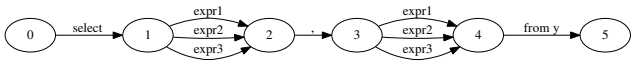


Figure 19: Example of input graph for $n = 2$ and $m = 3$

Structure of input data was identical for all tested tools but representation of data is different and we should apply it in test generator. YaccConstructor uses graph as an input data structure and loads data from DOT[8] file. On the other hand, Alvor uses regular expression corresponded to finite automaton which was represented as a graph for YaccConstructor and loads it from text file. The following constructions are available:

- ”a” ”b” — concatenation
- {”a”, ”b”, ”c”} — variants

We created a test set generator which takes patterns as an input and generates tests using this pattern for both YaccConstructor and Alvor. Input regular expression for Alvor and the equivalent input graph for YaccConstructor are presented in the figure 20.

We used PC with the next configuration for test purposes.

- Operation system: Windows 8.1 Pro
- Operation system type: 64-bit OS, x64-based CPU
- CPU: Intel Core i7-2620M CPU 2.70GHz

[8] DOT is a text language for graph description. Language specification (accessed: 01.07.2014): `http://www.graphviz.org/doc/info/lang.html`

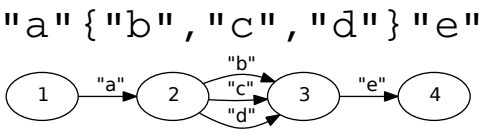


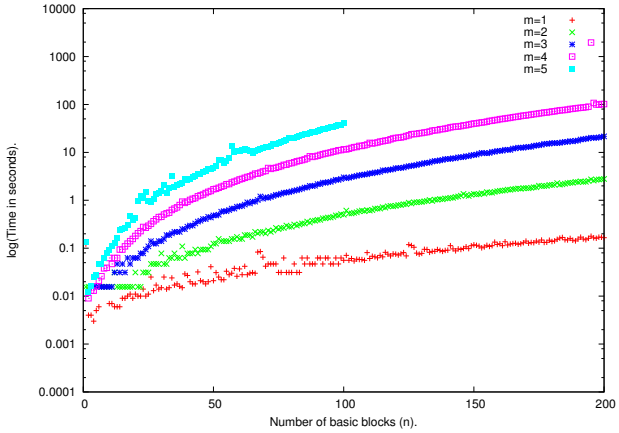
Figure 20: Eqivalent input for YaccConstructor and Alvor

- RAM: 12Gb
- Tools were implemented for different platforms (Alvor for Java platform, YaccConstructor for .Net framework) so we used corresponding environments:
  - Java:
    - Java(TM) SE Runtime Environment (build 1.7.0_45-b18)
    - Java HotSpot(TM) 64-Bit Server VM (build 24.45-b08, mixed mode)
  - .Net:
    - Microsoft.NET Framrwork v4.0.30319

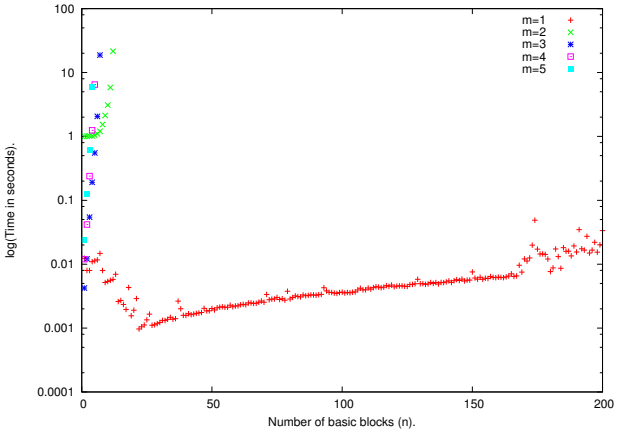
You can find the set of diagrams which illustrates the result of performance measurement for YaccConstructor and Alvor (figures 21, 22, 23, and 24). Tools were tested for input data generated with template described above. Values of m is between 1 and 5, and n is up to 200.

Note that we stopped the measurements for Alvor after very small value of n because it required more than 4Gb of RAM and took much execution time. It seems that Alvor requires exponential resources to process an input with big number of branches. It can be caused by the fact that Alvor can only branch stack and does not perform the merging.

From the data gathered it can be concluded that the performance of implemented tool is better than performance of Alvor in a wide



(a) Performance of YaccConstructor for $m \in \{1; 2; 3; 4; 5\}$



(b) Performance of Alvor for $m \in \{1; 2; 3; 4; 5\}$

Figure 24: Performance comparison of YaccComctructor and Alvor

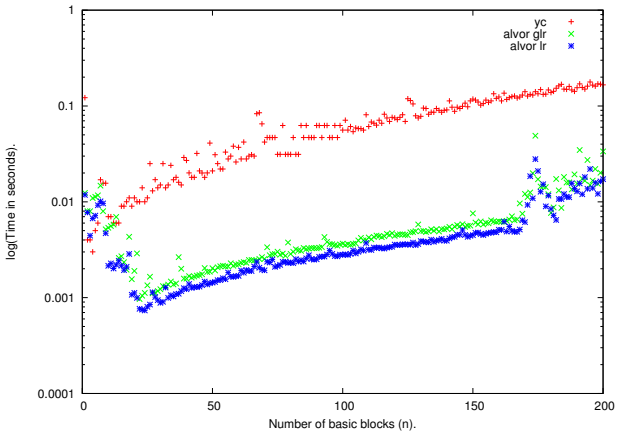


Figure 21: Performance comparison of YaccConstructor and Alvor for $m = 1$

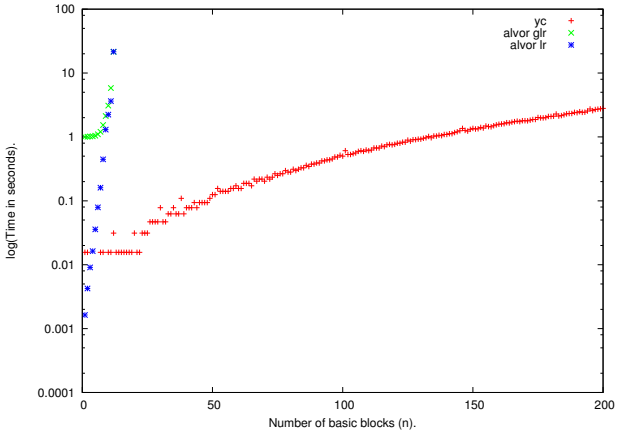


Figure 22: Performance comparison of YaccConstructor and Alvor for $m = 2$

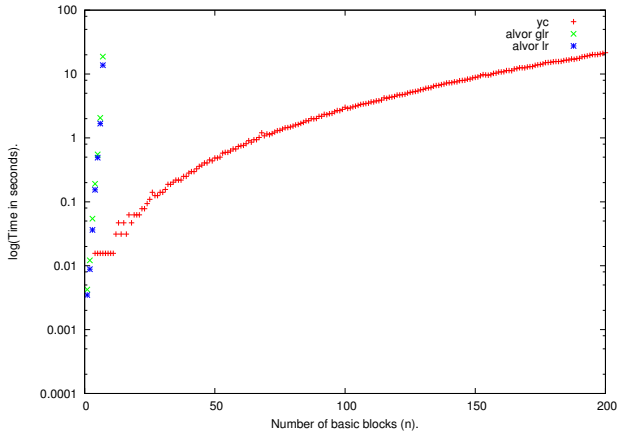


Figure 23: Performance comparison of YaccConstructor and Alvor for $m = 3$

rage of practical cases: 2–3 parallel edges in block and the number of blocks up to 200. But in case of linear input our tool performance is worse than Alvor performance.

6. Future Work

One of the main application of developed tool is error detecting, so it is important to improve corresponded algorithms. Error reporting and error recovery in generalized LR parsing are nontrivial problems. A considerable amount of research has been done on improving error reporting for LR parsers, although relatively little work has been done for GLR parsers [18]. 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. On the other hand, error processing in LL parsing algorithm is easier than in LR-algorithm, so we are going to investigate GLL parsing algorithm [6] and research whether it can be used as a base for abstract parsing algorithm. We expect that GLL-based abstract parsing will be based on the same ideas as GLR-based algorithm because it also manipulates with GSS and SPPF data structures. We also expect error processing in GLL-based abstract parsing to be more qualitative.

The situations in which there are huge amount of code which contains a few errors and should be processed in short time often occur in software reengineering. In this case GLR-based abstract parsing algorithm may be useful in spite of error processing issues. It is important that in such case we may notify developers of every suspicious situation in code. Migration to generalized parsing algorithms with better performance than RNGLR — BRNGLR [22] or RIGLR [23] — may help to increase performance of abstract parsing.

It is also important to improve quality of analysis so the next big problem is improvement and adjustment of approximation. Firstly, it is necessary to support cycles in input graph, but this may provoke parsing complexity increase and performance problems. As a possible solution to this problems, improved algorithm should only be applied when it is necessary (input graph contains cycles).

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] Annamaa A., Breslav A., Vene V. Using Abstract Lexical Analysis and Parsing to Detect Errors in String-embedded DSL Statements, Proceedings of the 22nd Nordic Workshop on Programming Theory. Marina Walden and Luigia Petre, editors, p. 20–22. 2010.

[3] Kyung-Goo Doh, Hyunha Kim, David A. Schmidt. Abstract Parsing: Static Analysis of Dynamically Generated Sring Output Using LR-parsing Technology, Proceedings of the 16th International Symposium on Static Analysis, SAS09. Springer-Verlag: Berlin; Heidelberg, p. 256–272, 2009.

[4] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt. Abstract LR-parsing, In Formal modeling, Gul Agha, Jos Meseguer, and Olivier Danvy (Eds.). Springer-Verlag, Berlin, Heidelberg, p. 90–109, 2011.

[5] H. Kim, K. Doh, and D.A. Schmidt, Static Validation of Dynamically Generated HTML Documents Based on Abstract Parsing and Semantic Processing. ;In Proceedings of SAS, 194 – 214, 2013.

[6] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing, Electron. Notes Theor. Comput. Sci. 253, 7 (September 2010), p. 177–189.

[7] Rekers J. G. 1992. Parser generation for interactive environments. Ph.D. thesis, Universty of Amsterdam.

[8] Grune D., Ceriel J. H. Jacobs. Parsing techniques: a practical guide. Ellis Horwood, Upper Saddle River, NJ, USA, 1990. p. 322.

[9] 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.

[10] 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.

[11] Elizabeth Scott and Adrian Johnstone. Right nulled glr parsers. ACM Trans. Program. Lang. Syst., 28(4):p. 577 – 618, July 2006.

[12] 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.

[13] Costantini G., Ferrara P., Cortesi F. Static analysis of string values, Proceedings of the 13th international conference on Formal methods and software engineering, ICFEM11. Springer-Verlag: Berlin; Heidelberg, p. 505-521, 2011.

[14] ISO. ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL. 1992. p. 668.

[15] Xiang Fu, Xin Lu, Peltsverger B. e.a. A static analysis framework for detecting SQL injection vulnerabilities, Proceedings of the 31st Annual International Computer Software and Applications Conference. Vol. 01, COMPSAC07, Washington, DC, USA, IEEE Computer Society, p. 87–96, 2007.

[16] 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, p. 1403–1414, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Semen Grigorev. Automated transformation of dynamic sql queries in information system reengineering. Master's thesis, Saint-Petersburg State University, 2012.

[18] Giorgios Robert Economopoulos. Generalised LR parsing algorithms. 2006.

[19] Tomita, Masaru. LR parsers for natural languages, In 10th International Conference on Computational Linguistics, p. 354–357. ACL, 1984.

[20] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '77). ACM, New York, NY, USA, p. 238–252.

[21] Syme D., Granicz A., and Cisternino A.: Expert F#, Apress (2007).

[22] Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos. 2007. BRNGLR: a cubic Tomita-style GLR parsing algorithm. Acta Inf. 44, 6 (September 2007), p. 427-461.

[23] Elizabeth Scott and Adrian Johnstone. Table based parsers with reduced stack activity. Technical Report CSD-TR-02-08, Royal Holloway, University of London, Department of Computer Science, Egham, Surrey TW20 0EX, England, May 2003.