# Relaxed Parsing of Regular Approximations of String-Embedded Languages

Ekaterina Verbitskaia[1], Semyon Grigorev[2], and Dmitry Avdyukhin[3]

[1] Saint Petersburg State University
kajigor@gmail.com,
[2] rsdpisuy@gmail.com
[3] dimonbv@gmail.com

**Abstract.** We present a technique for syntax analysis of a regular set of input strings. This problem is relevant for the analysis of string-embedded languages when a host program generates clauses of embedded language at run time. Our technique is based on a generalization of RNGLR algorithm, which, inherently, allows us to construct a finite representation of parse forest for regularly approximated set of input strings. This representation can be further utilized for semantic analysis and transformations in the context of reengineering, code maintenance, program understanding etc. The approach in question implements *relaxed parsing*: non-recognized strings in approximation set are ignored with no error detection.

**Keywords:** string-embedded languages, string analysis, parsing, parser generator, RNGLR.

## Introduction

There is a broad class of applications which utilize the idea of *string embedding* of one language into another. In this approach a host program generates string representation of clauses in some external language which are then passed to a dedicated runtime component for analysis and execution. One significant example of string embedded language is embedded SQL [1]; among others, some frameworks such as JSP [2] and PHP mySQL interface[4] can be mentioned.

Despite providing a high level of expressiveness and flexibility, string embedding makes the behavior of the system less predictable and harder to reason about since a whole class of verification procedures is postponed until run time, which complicates development, testing and maintenance. To overcome this deficiency, it is desirable to perform syntax analysis of well-formedness of all generated clauses prior to execution. However, since the host language, as a rule, is Turing-complete, the precise analysis is undecidable; the common approach is to analyse an over-approximating set of strings represented in some constructive form. It's worth to mention that, similarly to regular syntax analysis, the analysis of string-embedded languages often follows two-level scheme: first a set of

[4] http://php.net/manual/en/mysqli.query.php

strings is tokenized to provide an approximation for tokenized stream set, then this set is parsed by a syntax analyzer.

This paper contributes a generalization of RNGLR (Right-Nulled Generalized LR [3]) algorithm, which, instead of linear stream of tokens, analyzes a regular set of streams. Our algorithm can be considered as proper generalization since it provides exactly the same result as the original one on a trivial one-element set; moreover, our implementation reuses original RNGLR parse tables. The distinctive feature of our approach in comparison with other techniques for analysis of string-embedded languages is that it provides the set of parsing trees, encoded in the form of Shared Packed Parse Forest (SPPF) [4]. The choice of RNGLR looks quite natural in this regard since in its original form it already incorporates the technique to deal with multiple ways of parsing. On the other hand, our approach can be categorized as *relaxed* parsing since it silently ignores non-recognized part of the input; we do not consider this property as an essential drawback because it only means that, to provide best results, it has to be combined with some existing recognition-centric approaches.

## 1 Related Works

Our approach for syntax analysis of string-embedded languages borrows some common principles from existing techniques in this area. In addition, we reuse RNGLR syntax analysis algorithm and some accompanying constructs. In this section we provide a review and recollect some important notions which will be referred to later on.

### 1.1 String-Embedded Languages Analysis Techniques

The analysis of string-embedded languages, as a rule, requires a set of *hotspots* to be indicated in the host application source code. Hotspot is considered as some "point of interest", where the analysis of the set of possible string values is desirable. This task can be performed either in a user-assisted manner or automatically using some pragmatic considerations or knowledge of the framework being analyzed. The following logical steps include static analysis to construct an approximation for the set of all possible string values, lexical, syntax, and, perhaps, some kind of semantic analysis. These steps are not necessarily performed separately; some of them may be omitted.

A rather natural idea of *regular approximation* is to approximate the set of all possible strings by a regular expression. In recognition-centric formulation, this approach boils down to the problem of inclusion of approximating regular language into context-free reference language, which is decidable for a number of practically significant cases [5]. Many approaches follow this route. In [6], forward reachability analysis is used to compute regular approximation for all string values in the program. Further analysis is based on patterns detection in approximation set or generation of some finite subset of strings for analysis by

standalone tools. Regular approximation in [7] is acquired by widening context-free approximation, initially built as a result of program analysis. Our approach is partially inspired by Alvor [8, 9] which utilizes GLR-based technique for syntax analysis of regular approximation; this framework implements abstract lexical analysis to convert a regular language over characters into regular language over tokens, which simplifies syntax analysis.

Kyung-Goo Doh et al. in a series of papers [10–12] introduced an approach, based on implicit representation of the set of potential strings as a system of data-flow equations. Conventional LALR(1) is chosen for the basis of parsing algorithm; original parse tables are reused. Syntax analysis is performed as the system of dataflow equations is being solved iteratively in the space of abstract stacks. The problem of infinite stack growth, which appears in general case, is handled using abstract interpretation [13]. This approach later evolved to a certain kind of semantic processing in terms of attribute grammars which made it possible to analyze a wider class of languages, than LALR(1).

### 1.2 Right-Nulled Generalized LR Parsing Algorithm

RNGLR (Right-Nulled Generalized LR) is a modification of Generalized LR (GLR) algorithm, which was developed by Masaru Tomita [14] in the context of natural language processing. GLR was designed to handle ambiguous context-free grammars. Ambiguities in the grammar produce shift/reduce and reduce/reduce conflicts, speaking in terms of LR approach. The algorithm uses parse tables, similar to those for classical LR, each cell of which can contain multiple actions. The general approach is to carry out all possible actions during parsing using graph-based data structures to efficiently represent the set of stacks and derivation trees. Originally, Tomita's algorithm was unable to recognize all context-free languages. Elizabeth Scott and Adrian Johnstone presented RNGLR [3], which extends GLR with a certain way of handling *right nullable* rules (i.e. rules of the form $A \to \alpha\beta$, where $\beta$ reduces to an empty string).

To efficiently represent the set of all stacks produced during parsing, RNGLR uses Graph Structured Stack (GSS). GSS is a directed graph, whose vertices correspond to the elements of individual stacks and edges link successive stack elements. Each vertex can have multiple incoming and outgoing edges to merge multiple stacks together; thus stack element sharing is implemented. Each vertex is a pair $(s, l)$, where $s$ is a parser state and $l$ is a *level* (position in the input string). Vertices in GSS are unique and there are no multi-edges.

According to RNGLR, an input is read left-to-right, one token at a time, and the levels of GSS are constructed sequentially for each input position: first, all possible reductions are applied, then the next input terminal is shifted and pushed to the GSS. When a reduction or pushing is performed, the algorithm modifies GSS in the following manner. Suppose an edge $(v_t, v_h)$ has to be added to the GSS. By construction, the head vertex $v_h$ is always already in the GSS. If the tail vertex is also in the GSS, then a new edge $(v_t, v_h)$ is added (provided it is not yet there); otherwise both new tail vertex and new edge are created and added to the GSS. Every time a new vertex $v = (s, l)$ is created, the algorithm

calculates the new parser state $s'$ from $s$ and the next terminal of the input. The pair $(v, s')$, called *push*, is added to the global collection $\mathcal{Q}$. The set of $\epsilon$-reductions (i.e. reductions with length $l = 0$) is also calculated, when a new vertex is added to the GSS, and reductions from this set are added to the global queue $\mathcal{R}$. Reductions with length $l > 0$ are calculated and added to $\mathcal{R}$ each time a new (non-$\epsilon$) edge is created.

An input string can have several derivation trees and, as a rule, they can have numerous identical subtrees. Shared Packed Parse Forest (SPPF) [4] is a directed graph designed for a compact representation of all possible derivation trees. SPPF has the following structure: the *root* (i.e. vertex with no incoming edges) corresponds to the starting nonterminal of the grammar; vertices with no outgoing edges correspond to terminals or derivation of $\epsilon$-string; the rest of the vertices is divided into two classes: *nonterminal* and *production*. Each nonterminal vertex keeps a collection of production nodes, each of which represents one possible derivation of that nonterminal. Production vertices represent a right-hand side of the production and keep an ordered list of terminal or nonterminal nodes. The length of this list lies in the range $[l - k..l]$, where $l$ is the length of production right-hand side, and $k$ is the number of rightmost symbols which derive $\epsilon$ (nullable symbols are ignored to reduce memory consumption).

SPPF is constructed simultaneously with GSS. Each edge of the GSS is associated with either a terminal or nonterminal node. When a GSS edge is added with a push, a new terminal node is created and associated with the edge. Nonterminal nodes are associated with edges which were added, when reductions were performed: if the edge has already been in GSS, a production node is added to the family of nonterminal nodes, associated with the edge. All subgraphs from the edges of the reduction path are added as children to the production node. After the input is read to the end, all vertices with accepting states are searched and nodes associated with outgoing edges of such vertices are merged to form the resulting SPPF. All unreachable vertices are deleted from the SPPF graph, which leaves only the actual derivation trees for the input.

The detailed algorithm description in the form of pseudocode can be found in Appendix B.

## 2 Relaxed Parsing of Regular Sets

The input of our algorithm (see Algorithm 1) is a reference grammar $G$ with alphabet of terminal symbols $T$ and a finite non-deterministic automaton $(Q, \Sigma, \delta, q_0, q_f)$ with a single start state $q_0$, single final state $q_f$ and no $\epsilon$-transitions, where $\Sigma \subseteq T$ — alphabet of input symbols, $Q$ — alphabet of states, $\delta$ — transition relation. RNGLR parse tables and some accessory information (called $parserSource$ in pseudocode) are generated for the grammar $G$.

The general idea of the algorithm is to traverse the automaton graph and sequentially construct GSS, similarly as in RNGLR. However, as we deal with a graph instead of a linear stream, the next symbol turns into the *set of terminals* on all outgoing edges of current vertex. This results in a different semantics of

pushing and reducing (see line 5, Algorithm 2, and lines 9 and 21, Algorithm 3). We use queue $\mathcal{Q}$ to control the order of automaton graph vertices processing. Every time a new GSS vertex is added, all zero-reductions have to be performed and then new tokens have to be shifted, so a corresponding graph vertex has to be enqueueed for further processing. Addition of new GSS edge can produce reductions to handle, so the graph vertex at the tail of the added edge has also to be enqueueed (see Algorithm 3). Reductions are applied along the paths in GSS, and if we add a new edge to some tail vertex, which was already presented in GSS, we also have to recalculate all *passing* reductions (see *applyPassingReductions* function in Algorithm 2).

Like RNGLR, we associate GSS vertices with positions in the input, and, in our case, a position coincides with some state of input automaton. We construct some inner data structure (referred to as *inner graph*) by copying input automaton graph and extending each of its vertices with the following collections:

- *processed*: GSS vertices, for which all the pushes were processed. This set aggregates all GSS vertices, associated with inner graph vertex.
- *unprocessed*: GSS vertices, for which all the pushes are to be processed. This set is analogous to $\mathcal{Q}$ of original RNGLR.
- *reductions*: a queue, which is analogous to $\mathcal{R}$ of original RNGLR: all reductions to be processed.
- *passingReductionsToHandle*: pairs of GSS vertex and GSS edge to apply passing reductions along them.

Besides parser *state* and *level* (which is equal to the input automaton state), a collection of *passing reductions* is stored in a GSS vertex. Passing reduction is a triplet $(startV, N, l)$, representing reductions, whose path contains given GSS vertex. This triplet is similar to one describing reduction, where $l$ is a remaining length of the path. Passing reductions are stored for every vertex of the path (except for the first and the last) during path search in *makeReductions* function (see Algorithm 2).

We inherit SPPF construction from the original RNGLR; in our case, derivation trees for strings, accumulated along the paths of the input automaton graph, are merged.

### 2.1 Correctness of the Algorithm

We conclude this section by justification of termination and correctness of our algorithm.

Theorem 1. *Algorithm terminates for any input.*

Proof. Each vertex of inner graph contains, at most, $N$ GSS vertices, where $N$ is the number of parser states. So, the total number of GSS vertices is, at most, $N \times n$, where $n$ is the number of vertices in the inner graph. Since GSS has no multi-edges, the number of its edges is $O((N \times n)^2)$. The algorithm dequeues some vertex to process from $\mathcal{Q}$ in each iteration of the main loop. Vertices are enqueued to $\mathcal{Q}$ only when a new edge is added to GSS. Since the number of GSS edges is finite, the algorithm always terminates. □

**Algorithm 1** Parsing algorithm

```
1: function PARSE(grammar, automaton)
2:     inputGraph ← construct inner graph representation of automaton
3:     parserSource ← generate RNGLR parse tables for grammar
4:     if inputGraph contains no edges then
5:         if parserSource accepts empty input then report success
6:         else report failure
7:     else
8:         ADDVERTEX(inputGraph.startVertex, startState)
9:         Q.Enqueue(inputGraph.startVertex)
10:        while Q is not empty do
11:            v ← Q.Dequeue()
12:            MAKEREDUCTIONS(v)
13:            PUSH(v)
14:            APPLYPASSINGREDUCTIONS(v)
15:        if ∃v_f : v_f.level = q_f and v_f.state is accepting then report success
16:        else report failure
```

**Algorithm 2** Single vertex processing

```
1: function PUSH(innerGraphV)
2:     U ← copy innerGraphV.unprocessed
3:     clear innerGraphV.unprocessed
4:     for all v_h in U do
5:         for all e in outgoing edges of innerGraphV do
6:             push ← calculate next state by v_h.state and the token on e
7:             ADDEDGE(v_h, e.Head, push, false)
8:             add v_h in innerGraphV.processed
9: function MAKEREDUCTIONS(innerGraphV)
10:    while innerGraphV.reductions is not empty do
11:        (startV, N, l) ← innerGraphV.reductions.Dequeue()
12:        find the set of vertices X reachable from startV
13:         along the path of length (l − 1), or 0 if l = 0;
14:        add (startV, N, l − i) in v.passingReductions,
15:         where v is an i-th vertex of the path
16:        for all v_h in X do
17:            state_t ← calculate new state by v_h.state and nonterminal N
18:            ADDEDGE(v_h, startV, state_t, (l = 0))
19: function APPLYPASSINGREDUCTIONS(innerGraphV)
20:    for all (v, edge) in innerGraphV.passingReductionsToHandle do
21:        for all (startV, N, l) ← v.passingReductions.Dequeue() do
22:            find the set of vertices X,
23:             reachable from edge along the path of length (l − 1)
24:            for all v_h in X do
25:                state_t ← calculate new state by v_h.state and nonterminal N
26:                ADDEDGE(v_h, startV, state_t, false)
```

**Algorithm 3** GSS construction

```
 1: function addVertex(innerGraphV, state)
 2:     v ← find a vertex with state = state in
 3:      innerGraphV.processed ∪ innerGraphV.unprocessed
 4:     if v is not null then                    ▷ The vertex have been found in GSS
 5:         return (v, false)
 6:     else
 7:         v ← create new vertex for innerGraphV with state state
 8:         add v in innerGraphV.unprocessed
 9:         for all e in outgoing edges of innerGraphV do
10:             calculate the set of zero-reductions by v
11:              and the token on e and add them in innerGraphV.reductions
12:         return (v, true)
13: function addEdge(v_h, innerGraphV, state_t, isZeroReduction)
14:     (v_t, isNew) ← addVertex(innerGraphV, state_t)
15:     if GSS does not contain edge from v_t to v_h then
16:         edge ← create new edge from v_t to v_h
17:         Q.Enqueue(innerGraphV)
18:         if not isNew and v_t.passingReductions.Count > 0 then
19:             add (v_t, edge) in innerGraphV.passingReductionsToHandle
20:         if not isZeroReduction then
21:             for all e in outgoing edges of innerGraphV do
22:                 calculate the set of reductions by v
23:                  and the token on e and add them in innerGraphV.reductions
```

To prove correctness, we first introduce the following definition:

Definition. *Correct tree* is an ordered tree with the following properties:

1. The root is the start nonterminal of the grammar $G$.
2. The leaf nodes are terminals of $G$. The sequence of the leaf nodes corresponds to some path in the inner graph.
3. The interior nodes are nonterminals of $G$. All children of nonterminal $N$ correspond to the symbols of the right-hand side of some production for $N$ in $G$.

Informally, a correct tree is a derivation tree (w.r.t. reference grammar) for some word in regular approximation. Now we have to prove that, first, SPPF contains only correct trees, and second, that for any recognized by the reference grammar string there is some correct tree in the SPPF.

Lemma. For every GSS edge $(v_t, v_h)$, $v_t \in V_t.processed$, $v_h \in V_h.processed$, the terminals of the associated subtree correspond to some path in the inner graph $p$ from $V_h$ to $V_t$.

Proof. The proof is by induction on the height of derivation tree. The base case is either some $\epsilon$-tree or a tree with a single leaf. An $\epsilon$-tree corresponds to a path of zero length; the tail and the head of the edge associated with $\epsilon$-tree are identical, thus the statement is true. A tree with the single leaf corresponds to a

single terminal read from an edge $(V_h, V_t)$ of the inner graph, thus the statement is true.

A tree of height $k$ has a nonterminal $N$ as its root. By third statement of correct tree definition, there is a production $N \to A_0, A_1, \ldots, A_n$ for children $A_0, A_1, \ldots, A_n$ of the root node. A subtree $A_i$ is associated with GSS edge $(v_t^i, v_h^i)$ and, as its height is $k-1$, by inductive hypothesis, there is a path in the inner graph from $V_h^i$ to $V_t^i$. $V_t^i = V_h^{i+1}$, since $v_t^i = v_h^{i+1}$, thus there is a path in the inner graph from $V_h^0$ to $V_t^n$, corresponding to the tree under consideration. □

Theorem 2. *Every tree, generated from SPPF, is correct.*

Proof. Consider arbitrary tree, generated from SPPF, and prove that it is correct. The first and the third statements of correctness definition immediately follow from SPPF definition. The second statement of the definition follows from Lemma 1 by considering all edges from GSS vertices on the last level, labeled by accepting state, to the vertices on level 0. □

Theorem 3. *For every path p in the inner graph, recognized w.r.t. reference grammar, a correct tree corresponding to p can be generated from SPPF.*

Proof. Consider arbitrary correct tree and show it can be generated from SPPF. The proof follows the proof of correctness for RNGLR algorithm, except for the following moment. RNGLR constructs GSS layer-by-layer: it is guaranteed, that $\forall j \in [0..i-1]$ $j$-th level of the GSS would be fixed by the time, when $i$-th level is processed. In our case, this property does not hold, which leads to a possible generation of some paths for already applied reductions. The only possible way to actually add a new path is to add an edge $(v_t, v_h)$, where $v_t$ is already in the GSS and it has some incoming edges. Since the algorithm stores which reductions have passed through each vertex, to overcome this problem it is sufficient to continue passing reductions, stored in $v_t$, and this is exactly what *applyPassingReductions* function does. □

## 3 Conclusion

We presented and proved the correctness of generalized RNGLR algorithm, designed for syntactic analysis of regular sets of tokens. The algorithm constructs a set of derivation trees for every recognized string of the input set in the form of SPPF, whereas non-recognized part of the input is ignored. The distinctive feature of our approach is that, unlike others, it delivers a set of all parse trees, encoded in the form of SPPF, for recognized part. We implemented our algorithm in F# as a part of YaccConstructor project[5]; host-language specific features were implemented using JetBrains ReSharper SDK[6], which potentially makes it possible to analyse multiple host languages (our experiments involved C# and Javascript). An example of regular set parsing and SPPF construction is shown in the Appendix A.

We can indicate some directions for future research. First, the complexity estimation of our algorithm is still unclear; existing literature say very little on

[5] https://github.com/YaccConstructor/YaccConstructor
[6] https://www.jetbrains.com/resharper

this subject; in addition the contribution of SPPF construction has to be taken into account. Another direction concerns the utilization of SPPF for semantic analysis. While it is clear, that availability of SPPF is beneficial in general sense, the concrete ways of its utilization can be cumbersome since SPPF represents potentially infinite set of parse trees.

*Acknowledgments.* We thank Dmitri Boulytchev for the scientific guidance and the feedback on this work.

## References

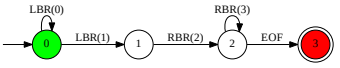
1. ISO. ISO/IEC 9075:1992. Information Technology — Database Languages — SQL, 1992.
2. Damon Houglan, Aaron Tavistock. Core JSP // Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000, 416 p.
3. Elizabeth Scott, Adrian Johnstone. Right Nulled GLR Parsers // ACM Trans. Program. Lang. Syst., Vol. 28, № 4, 2006, P. 577–618.
4. Jan Rekers. Parser Generation for Interactive Environments. PhD Thesis. University of Amsterdam, 1992, 174 p.
5. Peter R. J. Asveld, Anton Nijholt. The Inclusion Problem for Some Subclasses of Context-free Languages // Theoretical Computer Science, Vol. 230, № 1-2, 1999, P. 247–256.
6. Fang Yu, Muath Alkhalaf, Tevfik Bultan, Oscar H. Ibarra. Automata-based Symbolic String Analysis for Vulnerability Detection // Formal Methods in System Design, Vol. 44, № 1, 2014, P. 44–70.
7. Aske Simon Christensen, Anders Møller, Michael I. Schwartzbach. Precise Analysis of String Expressions // Proceedings of the 10th International Conference on Static Analysis, 2003, P. 1–18.
8. Aivar Annamaa, Andrey Breslav, Jevgeni Kabanov, Varmo Vene. An Interactive Tool for Analyzing Embedded SQL Queries // Proceedings of the 8th Asian Conference on Programming Languages and Systems, 2010, P. 131–138.
9. Aivar Annamaa, Andrey Breslav, Varmo Vene. Using Abstract Lexical Analysis and Parsing to Detect Errors in String-Embedded DSL Statements // Proceedings of the 22nd Nordic Workshop on Programming Theory, 2010, P. 20–22.
10. Kyung-Goo Doh, Hyunha Kim, David A. Schmidt. Abstract Parsing: Static Analysis of Dynamically Generated String Output Using LR-parsing Technology // Proceedings of the 16th International Symposium on Static Analysis, 2009, P. 256–272.
11. Kyung-Goo Doh, Hyunha Kim, David A. Schmidt. Abstract LR-parsing // Formal Modeling, 2011, P. 90–109.
12. Kyung-Goo Doh, Hyunha Kim, David A. Schmidt. Static Validation of Dynamically Generated HTML Documents Based on Abstract Parsing and Semantic Processing // Proceedings of the 20th International Symposium on Static Analysis, 2013, P. 194–214.
13. Patrick Cousot, Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints // Proceedings of the 4th Symposium on Principles of Programming Languages, 1977, P. 238–252.
14. Masaru Tomita. An Efficient All-paths Parsing Algorithm for Natural Languages // Carnegie-Mellon University, Dept. of Computer Science, 1984.

# A Appendix: Example of parsing and SPPF construction

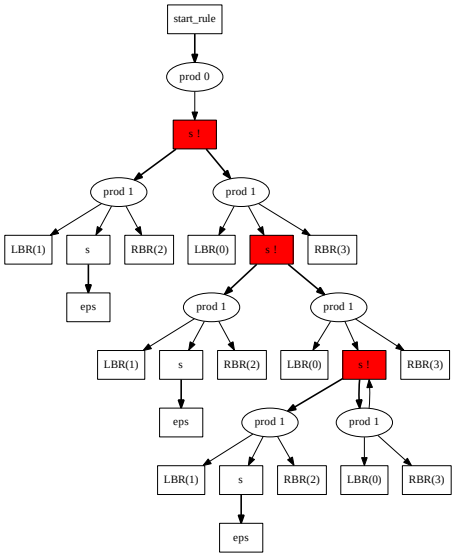
We demonstrate the application of our algorithm by the following example. The reference grammar is shown below:

$$
\begin{array}{ll}
(0)\ \mathit{start\_rule} ::= s \\
(1) \qquad s ::= \texttt{LBR}\ s\ \texttt{RBR}\ s \\
(2) \qquad s ::= \epsilon
\end{array}
$$

The automaton for regular approximation after tokenization is shown on the Fig. 1a; the SPPF, provided by our algorithm, is shown on the Fig. 1b.




(a) Regular approximation for string-embedded code after tokenization




(b) SPPF

Fig. 1: Regular approximation and SPPF

As it can be seen, some of the words from regular approximation do not belong to the reference language (for example, `LBR LBR RBR`). The algorithm ignores such strings and constructs SPPF, which contains derivation trees for all recognized strings w.r.t. reference grammar.

# B Appendix: RNGLR pseudocode

**Algorithm 4** RNGLR algorithm

```
1: function PARSE(grammar, input)
2:     R ← ∅      ▷ Queue of tuples of GSS vertex, nonterminal, and reduction length
3:     Q ← ∅                          ▷ Collection of pairs of GSS vertex and parser state
4:     if input = ϵ then
5:         if grammar accepts empty input then report success
6:         else report failure
7:     else
8:         ADDVERTEX(0, 0, startState)
9:         for all i in 0..input.Length − 1 do
10:            REDUCE(i)
11:            PUSH(i)
12:        if i = input.Length − 1 and there is a vertex in the last level of GSS which
   state is accepting then
13:            report success
14:        else report failure
15: function REDUCE(i)
16:    while R is not empty do
17:        (v, N, l) ← R.Dequeue()
18:        find the set X of vertices reachable from v along the path of length (l − 1)
19:        or length 0 if l = 0
20:        for all v_h = (level_h, state_h) in X do
21:            state_t ← calculate new state by state_h and nonterminal N
22:            ADDEDGE(i, v_h, v.level, state_tail, (l = 0))
23: function PUSH(i)
24:    Q' ← copy Q
25:    while Q' is not empty do
26:        (v, state) ← Q.Dequeue()
27:        ADDEDGE(i, v, v.level + 1, state, false)
```

**Algorithm 5** GSS construction

```
1: function addVertex(i, level, state)
2:     if GSS does not contain vertex v = (level, state) then
3:         add new vertex v = (level, state) to GSS
4:         calculate the set of shifts by v and the input[i + 1] and add them to Q
5:         calculate the set of zero-reductions by v and the input[i + 1] and
6:         add them to R
7:     return v
8: function addEdge(i, v_h, level_t, state_t, isZeroReduction)
9:     v_t ← addVertex(i, level_t, state_t)
10:    if GSS does not contain edge from v_t to v_h then
11:        add new edge from v_t to v_h to GSS
12:        if not isZeroReduction then
13:            calculate the set of reductions by v and the input[i + 1] and
14:            add them to R
```