Relaxed Parsing of Regular Approximations of String-Embedded Languages

Author: Ekaterina Verbitskaia

Saint Petersburg State University
JetBrains Programming Languages and Tools Lab

26/08/2015

String-embedded code

- Embedded SQL

```
SqlCommand myCommand = new SqlCommand(
    "SELECT * FROM table WHERE Column = @Param2",
    myConnection);
myCommand.Parameters.Add(myParam2);
```

- Dynamic SQL

```
IF @X = @Y
    SET @TBL = ' #table1 '
ELSE
    SET @TBL = ' table2 '
SET @S = 'SELECT x FROM' + @TBL + 'WHERE ISNULL(n,0) > 1'
EXECUTE (@S)
```

Problems

- String-embedded code are expressions in some programming language
  - It may be necessary to support them in IDE: code highlighting, autocomplete, refactorings
  - It may be necessary to transform them: migration of legacy software to new platforms
  - It may be necessary to detect vulnerabilities in such code
  - Any other problems of programming languages can occur

Static analysis of string-embedded code

- Performed without programm execution
- Checks that the set of properties holds for each possible expression value

- Undecidable for string-embedded code in the general case
- The set of possible expression values is over approximated and then the approximation is analysed

Existing tools

- PHP String Analyzer, Java String Analyzer, Alvor
  - Static analyzers for PHP, Java, and SQL embedded into Java respectively
- Kyung-Goo Doh et al.
  - Checks syntactical correctness of embedded code
- PHPStorm
  - IDE for PHP with support of HTML, CSS, JavaScript
- IntelliLang
  - PHPStorm and IDEA plugin, supports various languages
- STRANGER
  - Vulnerability detection of PHP

- Flaws
  - Limited functionality
  - Hard to extend them with new features or support new languages
  - Do not create structural representation of code

Static analysis of string-embedded code: the scheme

- Identification of hotspots: points of interest, where the analysis is desirable
- Approximation construction
- Lexical analysis
- **Syntactic analysis**
- Semantic analysis

Code: hotspot is marked

```
string res = "";
for(i = 0; i < l; i++)
    res = "()" + res;
use(res);
```

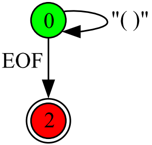
Possible values

`{"", "()", "()()", ..., "()"^l}`

Regular approximation

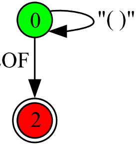
`("()")*`

Approximation

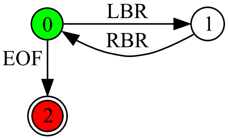


Static analysis of string-embedded code: the scheme

Approximation



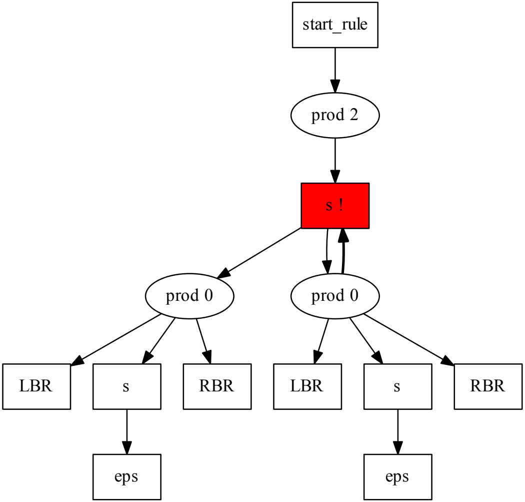
After lexing



Grammar

$$
\begin{array}{rcl}
start & ::= & s \\
s & ::= & \mathtt{LBR}\ s\ \mathtt{RBR}\ s \\
s & ::= & \epsilon
\end{array}
$$

Parse forest



Problem statement

The aim is to develop the algorithm suitable for syntactic analysis of string-embedded code

Tasks:

- Develop an algorithm for parsing of regular approximation of embedded code which produce a finite parse forest
- Parse forest should contain a parse tree for every correct (w.r.t. reference grammar) string accepted by the input automaton
- Incorrect strings should be omitted: no error detection
- An algorithm should not depend on the language of the host program and the language of embedded code

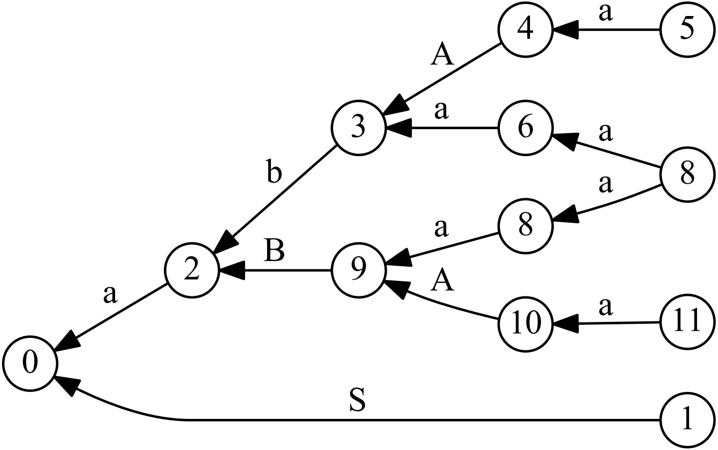
Algorithm

- **Input**: reference DCF grammar G and DFA graph with no ϵ-transitions over the alphabeth of terminals of G
- **Output**: finite representation of the trees corresponding to all correct string accepted by input automaton

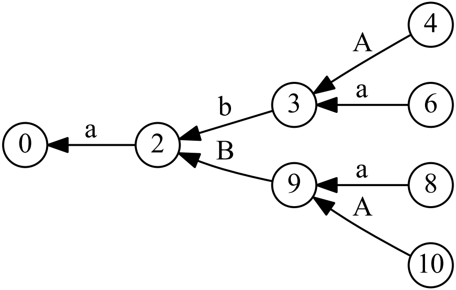
Right-Nulled Generalized LR algorithm

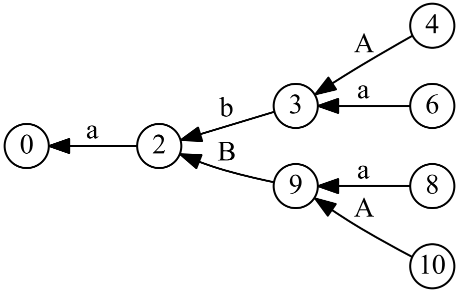
- RNGLR processes context free grammars
- In case when LR conflicts occur, parses in each possible way
  - Shift/Reduce conflict
  - Reduce/Reduce conflict
- Uses data structures which reduce memory consumption and guarantee appropriate time of analysis

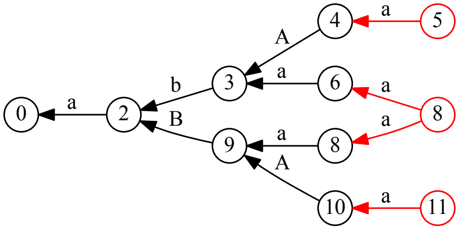
RNGLR data structures: Graph-Structured Stack



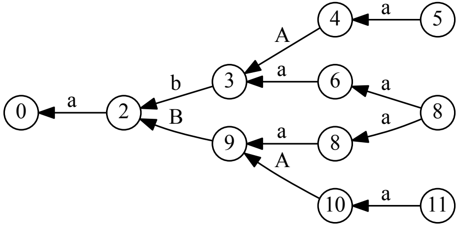
RNGLR operations: shift



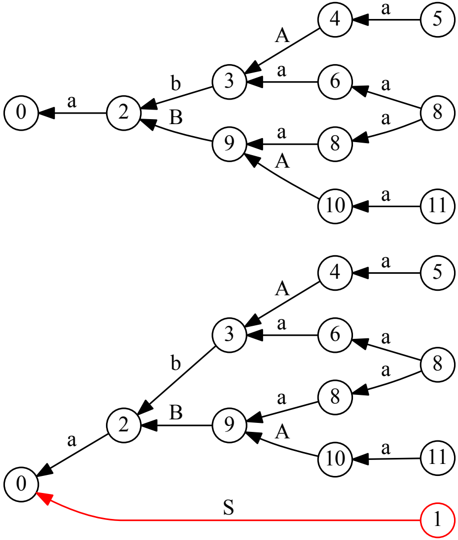

0
a
2
b
3
A
4
a
6
B
9
a
8
A
10


0
a
2
b
3
A
4
a
5
a
6
a
8
B
9
a
8
a
A
10
a
11

RNGLR operations: reduce



RNGLR operations: reduce



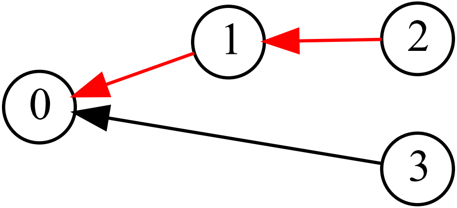
RNGLR

- Read the input sequentially
- Process all reductions
- Shift the next token
- Each time new vertex is added to the GSS, shift is calculated
- Each time new edge is created in the GSS, reductions are calculated

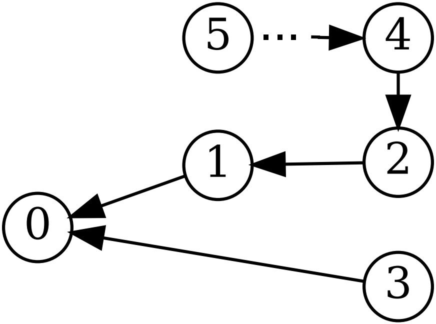
Algorithm

- Traverse the automaton graph and sequentially construct GSS, similarly as in RNGLR
- New type of “conflict”: Shift/Shift
- The set of LR-states is associated with each vertex of input graph
- The order in which the vertices of input graph are traversed is controled with a queue. Whenever new edge is added to GSS, its tail vertex is enqueued
- The algorithm implements relaxed parsing: errors are not detected, erroneous strings are ignored

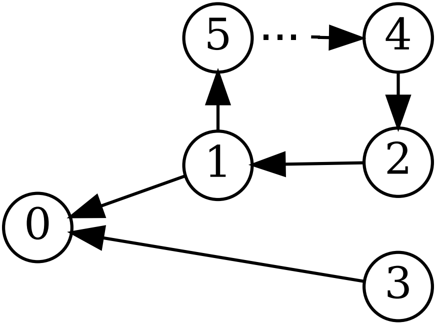
Cycles processing: initial stack



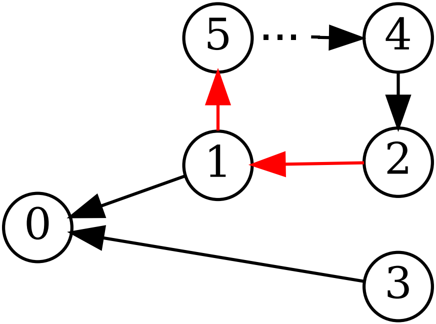
Cycles processing: some edges were added



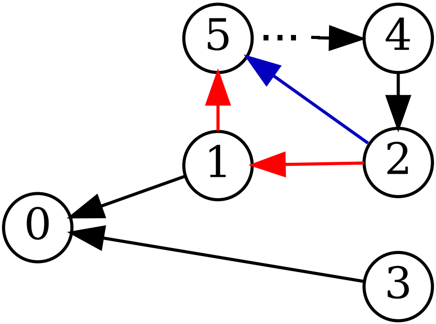
Cycles processing: cycle closed



Cycles processing: new reduction



Cycles processing: new reduction added

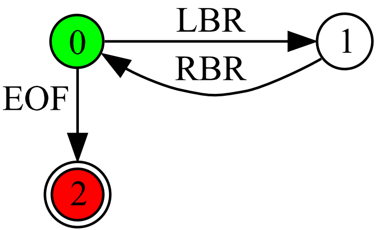


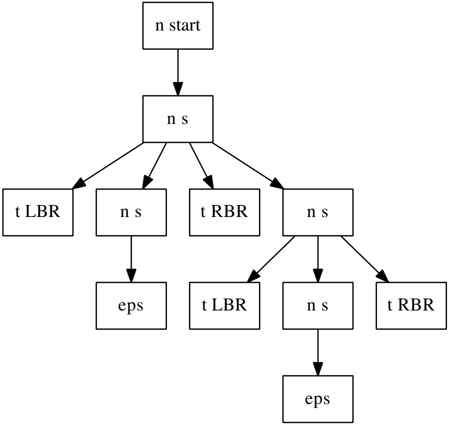
Parse forest construction

- Shared Packed Parse Forest — graph, in which all derivation trees are merged
- Constructed in the same manner as in the RNGLR-algorithm
  - Fragments of derivation trees are associated with GSS edges
  - Each time shift is processed, new tree of one terminal is created
  - Each time reduction is processed, new tree with children accumulated along the paths is created
    - Children are not copied, they are reused
  - The root of resulting tree is associated with GSS edge, corresponding to reduction to the starting nonterminal
    - Unreachable vertices are deleted from resulting graph

Algorithm: correctness

Correct tree — derivation tree of some string accumulated along the path in the input graph





Algorithm: correctness

Theorem (Termination)

Algorithm terminates for any input

Theorem (Correctness)

Every tree, generated from SPPF, is correct

Theorem (Correctness)

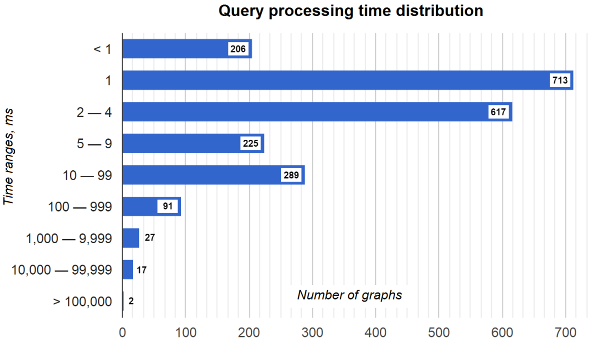
For every path p in the inner graph, recognized w.r.t. reference grammar, a correct tree corresponding to p can be generated from SPPF

Implementation

- The algorithm is implemented as a part of YaccConstructor project using F# programming language
- The generator of RNGLR parse tables and data structures for GSS and SPPF are reused

Evaluation

- The data is taken from the project of migration from MS-SQL to Oracle Server
- 2,7 lines of code, 2430 queries, 2188 successfully processed
- The number of queries which previously could not be processed because of timeout is decreased from 45 to 1



Conclusion

- The algorithm for parsing of regular approximation of dynamically generated string which constructs the finite representation of parse forest is developped
- Its termination and correctness are proved
- The algorithm is implemented as a part of YaccConstructor project
- The evaluation demonstrated it could be used for complex tasks