GLL parsing for embeded languages

Ragozina Anastasiya
Saint Petersburg State University

String-embedded languages are used to generate SQL-queries, HTML-pages, and etc., but ORM technologies are becoming more common and replacing dynamically generated character strings. In spite of this, embedded languages previously were very popular. Now, over the years, it may be necessary to re-engineer such systems. Also this approach is still used in the applications in which performance is crucial. Thus such software needs to be supported in IDEs (syntax highlighting, autocompletion, error detection) to simplify development process. It is necessary to parse embedded expressions to solve the problems described above. Parsing or syntactic analysis of string-embedded languages allows to check these expressions for errors statically, translate, support in IDEs.

Unlike host languages, dynamically generated code often cannot be represented as a linear stream, because such code is usually formed by string operations in loops or conditional statements. To represent nonlinear input we construct regular over-approximation of possible values for string expressions [1]. It means that instead of linear stream of tokens, a regular set of strings (which can be infinite) is to be analyzed. Such input can be represented with a finite state automaton which represents all possible values for string expression. The scheme of the approach is shown in Figure 1. Arithmetic expressions embedded to C# are considered as an example. Firstly, it is necessary to get the structural representation of source code written in C# (see Figure 1a). This step is carried out by a third-party tool. After that, we construct the automaton using the structural representation mentioned above. The finite state automaton is represented by a graph with edges labeled by strings (see Figure 1b). Its graph is an input for the lexer which builds an automaton over the alphabet of tokens (see Figure 1c). After determinization, the automaton is processed by the parser based on Generalised LL algorithm (GLL) [2]. *Generalised* stands for the property of the algorithm to parse all context free grammars (including left recursion) and builds parse forest. The parser represent the complete set of derivation trees with a shared packed parse forest (SPPF) [3] in Figure 1d. In SPPF, common subtrees are shared and nodes which correspond to different derivations of the same substring from the same nonterminal are combined. It allows to reduce the memory required for the parse forest. The red-colored nodes indicate different subtrees under this nonterminal node. The reasons of this situation are ambiguities in grammar or input. It is important that this representation is finite even for infinite input (automaton with cycles represent potentially infinite input).

GLL algorithm was chosen because it allows to process ambiguous grammars and reduces the memory required for parsing by means of using special data structures for stack (GSS) and parse forest (SPPF). Moreover, descriptors which are at the heart of the GLL technique allow to process nonlinear input with almost no changes in the parsing process. Descriptors contain the necessary information to re-starts parsing process from the point recorded in the descriptor. Rather than process single token from the input stream, it covers all outgoing edges from the current node in the input automaton. Situation when input graph contains cycles is covered too. This is possible due to the fact that the descriptors are added only once and using of SPPF allows to reuse subtrees and create cycles in them.

Described approach was implemented as a part of the tool YaccConstructor on .NET platform. Figure 1 illustrates the scheme and results of the parsing algorithm proposed. Current work is the algorithm implementation in Iguana project. Iguana is a Java implementation of the GLL parsing algorithm with more effective stack [4]. Earlier, as a part of YaccConstructor project, we proposed and implemented RNGLR-based algorithm for parsing of regular sets. Completeness and correctness were proved for this algorithm [5]. Current goal is to compare and unify these approaches.

1. REFERENCES

[1] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. 2014. Automata-based symbolic string analysis for vulnerability detection. Form. Methods Syst. Des. 44, 1 (February 2014), 44-70.

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

[3] Jan Rekers. 1992. Parser generation for interactive environments. Ph.D. thesis, University of Amsterdam.

[4] Ali Afroozeh and Anastasia Izmaylova. 2015. Faster, Practical GLL Parsing. Lecture Notes in Computer Science Volume 9031, 89-108

[5] Ekaterina Verbitskaia, Semyon Grigorev, and Dmitry Avdyukhin. 2015. Relaxed Parsing of Regular Approximations of String-Embedded Languages. Unpublished.

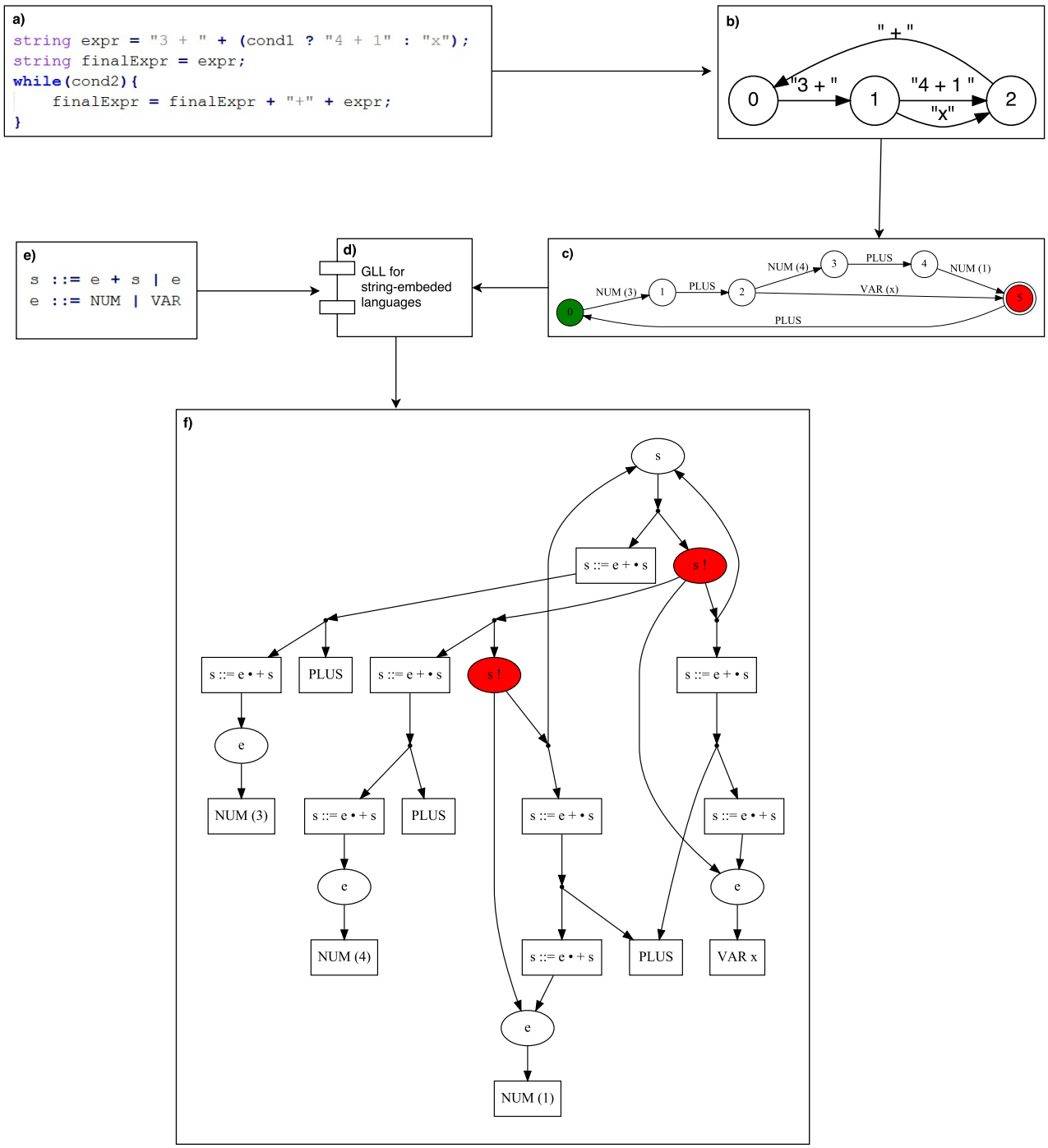

Figure 1: Scheme of embedded parsing with GLL-based algorithm