Context-Free Path Querying with Structural Representation of Result

Semyon Grigorev
and Anastasiya Ragozina

Saint Petersburg State University, 7/9 Universitetskaya nab.
St. Petersburg, 199034 Russia
semen.grigorev@jetbrains.com, ragozina.anastasiya@gmail.com

Abstract. Graph data model and graph databases are popular in such areas as bioinformatics, semantic web, and social networks. One specific problem in the area is a path querying with constraints formulated in terms of formal grammars. The query in this approach is written as a grammar and paths querying is graph parsing with respect to the grammar. There are several solutions to it, but they are based mostly on CYK or Earley algorithms which have some restrictions in comparison with other parsing techniques, and usage of advanced parsing techniques for graph parsing is still an open problem. In this paper we propose a graph parsing technique which is based on generalized top-down parsing algorithm (GLL) and allows one to build finite structural query result representation with respect to the given grammar in polynomial time and space for arbitrary context-free grammar and graph.

Keywords: Graph database, path query, graph parsing, context-free grammar, top-down parsing, GLL, LL

1 Introduction

Graph data model and graph data bases are very popular in such areas as bioinformatics, semantic web, and social networks. *(I don't like the next sentence)* Hence, different graph structured data analysis problems are stated and appropriate methods to solve these problems are required. One specific problem—path querying with constraints—is usually formulated in terms of formal grammars and is called formal language constrained path problem [4].

Classical parsing techniques can be used to solve formal language constrained path problem and thus the more common problem—graph parsing. Graph parsing may be required in graph data base querying, formal verification, string-embedded language processing, and any other areas where graph structured data is used.

Existing solutions in context-free graph querying field usually employ such parsing algorithms as CYK or Earley (for example [8,20,17]). These algorithms are simple, but impose restrictions. For example, CYK algorithm demands the input grammar to be transformed into Chomsky normal form (CNF) causing

performance issues, since parsing time depends on grammar size which significantly increases during the transformation. Such algorithms as GLR and GLL process arbitrary context-free grammars, thus performance may be improved by employing them for graph parsing problem. Other properties of parsing algorithms also affect performance: for example, in [9] author expects that it is possible to improve evaluation of queries for a given pair of nodes by using top-down directed parsing algorithm. Both CYK and Earley parsing algorithms are bottom-up, CYK is undirected, and Earley-based implementation [17] is known to have issues with cycles processing. *(Is it not able to parse them whatsoever?).* In this paper we show this assumption is true. The applicability of advanced parsing techniques [7] for path querying is stated as an open question in [9] and we provide positive answer to it. *Next sentence was here before: Thus, we argue that it is required to research the applicability of advanced parsing techniques [7] for graph processing, which is also stated in [9] as open question and we provide answer to it.*

Even though a set of path querying solutions has been developed [17,8,3,12], query result exploration is still a challenge [10], as also there is a need for simplification of complex query debugging, especially for context-free queries. *(The link between this and the next sentence is unclear)* In [9], annotated grammars are proposed as a possible solution: this representation is finite for any input data and contains information necessary for detailed result exploration. At the same time, classical parsing techniques provide another representation—derivation tree—which contains exhaustive information about parsed sentence structure in terms of specified grammar, and is more native for grammar based analysis. In the paper we show that finite derivation forest may be constructed for arbitrary graph and context-free grammar.

We make the following contributions in this paper.

1. We propose a graph parsing algorithm based on the generalized top-down parsing algorithm—GLL [13]—and provide its time and space complexity estimations. For graph $M = (V, E, L)$, space complexity is $O(|V|^3 + |E|)$ and time complexity is $O\left(|V|^3 * \max_{v \in V}(deg^+(v))\right)$.
2. We answer some questions on advanced parsing techniques applicability for graph processing stated in [9].
3. Proposed graph parsing algorithm constructs finite representation of parse forest containing derivation trees for all matched paths in graph. We show how this representation can be used for realistic problems solving.
4. We have implemented the proposed algorithm and our evaluation shows that advanced parsing techniques increase performance (up to 1000 times in some cases) as compared to CYK-based implementation, proposed in [20].

2 Preliminaries

In this work we are focused on the parsing algorithm, and not on the data representation, and we assume that whole input graph can be optimally located in RAM memory.

We start by introduction of necessary definitions.

- Context-free grammar is a quadruple $G = (N, \Sigma, P, S)$, where N is a set of nonterminal symbols, Σ is a set of terminal symbols, $S \in N$ is a start nonterminal, and P is a set of productions.
- $\mathcal{L}(G)$ denotes a language specified by grammar G, and is a set of terminal strings derived from start nonterminal of G: $L(G) = \{\omega | S \Rightarrow^*_G \omega\}$.
- Directed graph is a triple $M = (V, E, L)$, where V is a set of vertices, $L \subseteq \Sigma$ is a set of labels, and a set of edges $E \subseteq V \times L \times V$. We assume that there are no parallel edges with equal labels: for every $e_1 = (v_1, l_1, v_2) \in E, e_2 = (u_1, l_2, u_2) \in E$ if $v_1 = u_1$ and $v_2 = u_2$ then $l_1 \neq l_2$.
- $tag : E \to L$ is a helper function which returns a tag of a given edge.

$$tag(e = (v_1, l, v_2), e \in E) = l$$

- $\oplus : L^+ \times L^+ \to L^+$ denotes tag concatenation operation.
- Ω is a helper function which constructs a string produced by the given path. For every p path in M

$$\Omega(p = e_0, e_1, \ldots, e_{n-1}) = \\ tag(e_0) \oplus \cdots \oplus tag(e_{n-1}).$$

Using these definitions, we define context-free language constrained path querying as, given a query in form of grammar G, to construct the set of paths

$$Q(M, G) = \{p | p \text{ is path in } M, \Omega(p) \in \mathcal{L}(G)\}.$$

Note that $Q(M, G)$ can be an infinite set, hence it cannot be represented explicitly. In order to solve this problem, we construct compact data structure representation which stores all elements of $Q(M, G)$ in finite space and from which one can extract any of them.

2.1 Generalized LL Parsing Algorithm

One of widely used classes of parsing algorithms is LL(k) [7]—top-down algorithms which read input from left to right, build leftmost derivation, and use k symbols for lookahead. LL(k) parser may be implemented as deterministic pushdown automaton (DPDA). On the other hand, LL(k) parser may be implemented in recursive-descent manner: each rule transforms to function which can call functions for other rules in order specified by right hand side of corresponded rule. In this case, stack of DPDA is replaced with functions call stack. *(Functions call stack functions as a stack of DPDA (haha))*

Classical LL algorithm operates with a pointer to the input (position i) and a grammar slot—pointer to the grammar of form $N \to \alpha \cdot x\beta$. Parsing may be described as a transition of these pointers from the initial position ($i = 0$, $S \to \cdot\beta$, where S is start nonterminal) to the final ($i = input.Length$, $s \to \beta\cdot$). *(Strange terminology. Can we say it can be described as a transition system for initial state to the final? Maybe even list the following options in a usual for transition systems manner?)* At each step, there are four possible cases in processing of these pointers.

1. $N \to \alpha\cdot x\beta$, when x is a terminal and $x = input[i]$. In this case both pointers should be moved to the right ($i \leftarrow i + 1$, $N \to \alpha x \cdot \beta$).
2. $N \to \alpha \cdot X\beta$, when X is nonterminal. In this case we push return address $N \to \alpha X \cdot \beta$ to the stack and move pointer in the grammar to position $X \to \cdot\gamma$.
3. $N \to \alpha\cdot$. This case means that processing of nonterminal N is finished. We should pop return address from stack and use it as a new slot.
4. $S \to \alpha\cdot$, where S is a start nonterminal of grammar. In this case we should report success if $i = input.Length - 1$ or failure otherwise.

There can be several slots $X \to \cdot\gamma$ in the second case since the algorithm is nondeterministic, so some strategy to choose one of them for furhter parsing is needed. Lookahead is used to avoid nondeterminism in LL(k) algorithms, but this strategy is still not perfect because for some context-free languages deterministic choice is impossible even with infinite lookahead [5]. On the contrary to LL(k), generalized LL does not choose at all, but instead handles all possible variants by means of descriptors mechanism. Descriptor is a quadruple (L, s, j, a) where L is a grammar slot, s is a stack node, j is a position in the input string, and a is a node of derivation tree being constructed. Each descriptor fully describes parser state, thus instead of immediate processing of all variants, GLL stores all possible branches and process them *sequentially late (what?).*

The stack in parsing process is used to store return information for the parser—a state to return to after current state processing is finished. As mentioned before, generalized parsers process all possible derivation branches and parser must store its own stack for every branch. Being done naively, it leads to an infinite stack growth. Tomita-style graph structured stack (GSS) [19] combines stacks resolving this problem. Each GSS node contains a pair of a position in input and a grammar slot in GLL.

In order to provide termination and correctness, we should avoid duplication of descriptors, and be able to process GSS nodes in arbitrary order. The following additional sets are used for these purposees.

- R—working set which contains descriptors to be processed. Algorithm terminates as soon as R is empty.
- U—all created descriptors. New descriptor is added to R, only if it is not in U. This way each descriptor is processed only once which guarantees termination of the algorithm.

– P—popped nodes. This set allows to process descriptors (and GSS nodes) in arbitrary order. *(allow smb to do smth — other way this expression cannot be used. Rephrase)*

There can exist several derivation trees for a string with respect to an ambiguous grammar. Generalized LL builds all such trees and compacts them in a special data structure Shared Packed Parse Forest [11] *(And when you wrote about GSS, you used small letters. Make this consistent throughout the paper)*, which is described in the following section.

2.2 Shared Packed Parse Forest

Binarized Shared Packed Parse Forest (SPPF) [16] compresses derivation trees optimally reusing common nodes and subtrees. Version of GLL which utilizes this structure for parsing forest representation achieves worst-case cubic space complexity [14].

Binarized SPPF can be represented as a graph in which each node has one of four types described below. We denote the start and the end positions of substring as i and j respectively, and we call tuple (i, j) an *extension* of a node.

– **Terminal node** with label (i, T, j).
– **Nonterminal node** with label (i, N, j). This node denotes that there is at least one derivation for substring $\alpha = \omega[i..j-1]$ such that $N \Rightarrow^*_G \alpha, \alpha = \omega[i..j-1]$. All derivation trees for the given substring and nonterminal can be extracted from SPPF by left-to-right top-down graph traversal started from respective node.
– **Intermediate node**: a special kind of node used for binarization of SPPF. These nodes are labeled with (i, t, j), where t is a grammar slot.
– **Packed node** with label $(N \to \alpha, k)$. Subgraph with "root" in such node is one variant of derivation from nonterminal N in case when the parent is a nonterminal node labeled with $(\lhd\, (i, N, j))$.

An example of SPPF is presented in figure 2a. We remove redundant intermediate and packed nodes from the SPPF to simplify it and to decrease the size of the structure.

3 GLL-based Graph Parsing Algorithm

In this section we present such modification of GLL algorithm, that for input graph M, set of start vertices $V_s \subseteq V$, set of final vertices $V_f \subseteq V$, and grammar G_1, it returns SPPF which contains all derivation trees for all paths p in M, such that $\Omega(p) \in L(G_1)$, and $p.start \in V_s,\ p.end \in V_f$. In other words, we propose GLL-based algorithm which can solve language constrained path problem. *(I whould probably swap these to sentences)*

First of all, note that an input string for a classical parser can be represented as a linear graph, and positions in the input are vertices of this graph. This

observation can be generalized to arbitrary graph with remark that for evey position there is a set of labels of all outgoing edges for given vertex instead of just one next symbol. Thus, in order to use GLL for graph parsing we need to use graph vertices as positions in the input and modify step *(I'm not sure it was called a step)* 1 from section 2.1 to process multiple "next symbols". Small modification is also required for initialization of R set: the set of descriptors for all vertices in V_s should be added to R, not only one initial descriptor. Also, in the step 4: we should check that $i \in V_f$, not that $i = input.Length - 1$. All other steps (and correspondent functions) are reused from the original algorithm without any changes.

Our solution handles arbitrary numbers of start and final vertices, which allows one to solve different kinds of problems arising in the field, namely querying of all paths in graph, all paths from specified vertex, all paths between specified vertices. As a result, we can use proposed algorithm for querying paths for two specified vertices, and thus we provide positive answer for a question stated in [9]. Also SPPF represents a structure of paths in terms of grammar which provides exhaustive information about result. *(The last sentence is out of context)*

The algorithm has the following properties. Detailed discussion of them can be found in our report [6].

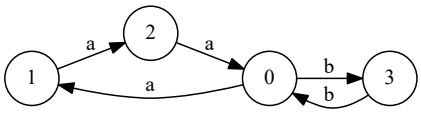
- Proposed algorithm terminates for arbitrary input data.
- The worst-case space complexity of proposed algorithm for graph $M = (V, E, L)$ is $O(|V|^3 + |E|)$.
- The worst-case runtime complexity of proposed algorithm for graph $M = (V, E, L)$ is $O\left(|V|^3 * \max_{v \in V}\left(deg^+(v)\right)\right)$.
- Result SPPF size is $O(|V'|^3 + |E'|)$ where $M' = (V', E', L')$ is a subgraph of input graph M which contains only matched paths.

4 An Example

In this section we demonstrate our algorithm on a small problem. The query used in our example may seem too primitive, but keep in mind that it is a classical context-free *same-generation queries* [1], which is base for other context-free queries [17].

Suppose that you are a student in School of Magic. It is your first day at School, so navigation in the building is a problem for you. Fortunately, you have a map of the building (fig. 1a) and additional knowledge about building construction:

- there are towers in the school (depicted as nodes of the graph in your map);
- towers can be connected by one-way galleries (represented as edges in your map);
- galleries have a "magic" property: you can start from any floor, but by following each gallery you either end up one floor above (edge label is 'a' *(consider renaming into 'up' and 'down')*), or one floor below (edge label is 'b').



(a) The map of School (input graph M)

$0 : S \to a\ S\ b$
$1 : S \to Middle$
$2 : Middle \to a\ b$

(b) Grammar G_1 for language $L = \{a^n b^n; n \geq 1\}$ with additional marker for the middle of a path

Fig. 1: An example: input graph and grammar

You want to find a path from your current position to the same floor in another tower. If only you have a map with all such paths. But orienteering is not your forte, so you prefer the structure of the paths be as simple as possible and all paths to have additional checkpoints to control your route.

It is evident that the simplest structure of required paths is $\{ab, aabb, aaabbb, \dots\}$. In terms of our definitions, it is necessary to find all paths p such that $\Omega(p) \in \{a^n b^n, n \geq 1\}$ in the graph $M = (\{0; 1; 2; 3\}, E, \{a; b\})$ (figure 1a).

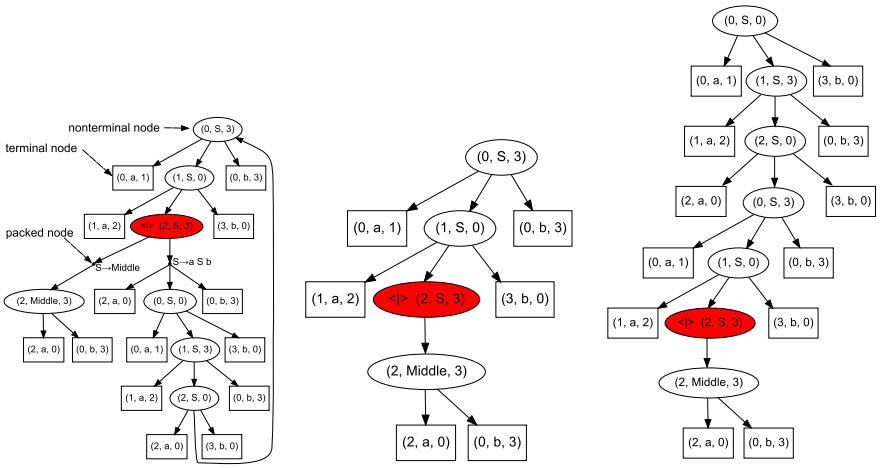
Unfortunately, language $\mathcal{L} = \{a^n b^n; n \geq 1\}$ is not regular which restricts the set of tools you can use. Another problem is the infinite size of solution. Being incapable to comprehend infinite set of paths, you want to obtain a finite map. Moreover, you want to know the structure of paths in terms of checkpoints.

We are not aware of any existing tools which can solve this problem, except for the one presented in the paper. Let us show how to get a map which helps to navigate in this strange School. *(don't forget that you moved this section after the algorithm, so you need to illustrate the algorithm, not show what the algorithm is supposed to do. This sentence should be rephrased.)*

Fortunately, the language $\mathcal{L} = \{a^n b^n; n \geq 1\}$ is context-free, thus it can be specified with context-free grammar. The fact that one language can be described with multiple grammars allows to add checkpoints: additional nonterminals can mark required subpaths *(or substrings — I am not sure which one suites better)*. In our case, desired checkpoint can be in the middle of the path. As a result, required language can be specified by the grammar G_1 presented in figure 1b, where $N = \{s; Middle\}$, $\Sigma = \{a; b\}$, and S is a start nonterminal.

Now, let us show that SPPF can be a desired map. SPPF for the example is presented in figure 2a. Each terminal node corresponds to the edge in the input graph: for each node with label (v_0, T, v_1) there is $e \in E : e = (v_0, T, v_1)$. Extensions stored in SPPF nodes allow us to check whether path from u to v exists and to extract it by SPPF traverse.

To illustrate how to use derivation structure, we can find a middle of any path. It is sufficient to find correspondent nonterminal *Middle* in SPPF for this purpose. So we can conclude from examining the result that there is only one (common) middle for all results, and it is a vertex with $id = 0$.



(a) Result SPPF (b) Derivation tree for p_0 (c) Derivation tree for p_1

Fig. 2: SPPF and examples of trees for specific paths for data from example (fig 1). Redundant nodes are eliminated; terminal nodes are duplicated only for figure simplification. We use filled shape and label of form $(\lhd\rhd\ (i, N, j))$ for nonterminal node to denote that there are multiple derivations from nonterminal N for substring $\omega[i..j-1]$.

Lets find paths p_i such that $S \underset{G_1}{\Longrightarrow}^* \Omega(p_i)$ and p_i starts from the vertex 0. To do this, we should find vertices with label $(0, S, \_)$ in SPPF. (There are two such vertices: $(0, S, 0)$ and $(0, S, 3)$.) Then let us to extract corresponded paths from SPPF. There is a cycle in SPPF in our example, so there are **at least** two different paths: $p_0 = \{(0,a,1);(1,a,2);(2,a,0);(0,b,3);(3,b,0);(0,b,3)\}$ and $p_1 = \{(0,a,1);(1,a,2);(2,a,0);(0,a,1);(1,a,2);(2,a,0);(0,b,3);(3,b,0);(0,b,3);(3,b,0);(0,b,3);(3,b,0)\}$. Trees for these paths are presented in figures 2b and 2c respectively.

We demonstrate that SPPF which was constructed by described algorithm can be useful for query result investigation. Also, when explicit representation of matched subgraph is preferable, required subgraph may be extracted from SPPF trivially by its traversal.

5 Evaluation

One of classical graph querying problems is a navigation queries for ontologies, and we apply our algorithm to this problem in order to estimate its practical value. We used dataset from paper [20]. Our algorithm is aimed to process graphs, so RDF files were converted to edge-labeled directed graph. For each triple (o, p, s) from RDF we added two edges: (o, p, s) and (s, p^{-1}, o).

We perform two classical *same-generation queries* [1], which, for example, is an important part of similarity query to biomedical databases [17].

Query 1 is based on the grammar for retrieving concepts on the same layer (presented in figure 3a). For this query our algorithm demonstrates up to 1000 times better performance and provides identical results as compared to the presented in [20] for Q_1.

Query 2 is based on the grammar for retrieving concepts on the adjacent layers (presented in figure 3b). Note that this query differs from the original query Q_2 from article [20] in the following details. First of all, we count only triples for nonterminal S because only paths derived from it correspond to paths between concepts on adjacent layers. Algorithm which is presented in [20] returns triples for all nonterminals. Moreover, grammar $\mathcal{G}_2$, which is presented in [20], describes paths not only between concepts on adjacent layers. For example, path "$subClassOf\ subClassOf^{-1}$" can be derived in $\mathcal{G}_2$, but it is a path between concepts on the same layer, not adjacent. We changed the grammar to fit a query to a description provided in paper [20]. Thus results of our query is different from results for Q_2 which provided in paper [20].

All tests were run on a x64-based PC with Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz and 32 GB RAM. Results of both queries are presented in table 1, where #triples is a number of (o, p, s) triples in RDF file, and #results is a number of triples of form (S, v_1, v_2). In our approach result triples can be founded by filtering out all SPPF nonterminal nodes labeled by (v_1, S, v_2).

$0 : S \to subClassOf^{-1}\ S\ subClassOf$
$1 : S \to type^{-1}\ S\ type$
$2 : S \to subClassOf^{-1}\ subClassOf$
$3 : S \to type^{-1}\ type$

(a) Grammar for query 1

$0 : S \to B\ subClassOf$
$1 : B \to subClassOf^{-1}\ B\ subClassOf$
$2 : B \to subClassOf^{-1}\ subClassOf$

(b) Grammar for query 2

Fig. 3: Grammars for evaluation

As a result, we conclude that our algorithm is fast enough to be applicable to some real-world problems.

6 Conclusion and Future Work

We propose GLL-based algorithm for context-free path querying which constructs finite structural representation of all paths satisfying given constraint. Provided data structure can be useful for result investigation and processing, and for query debugging. Presented algorithm has been implemented in F# programming language [18] and is available on GitHub:`https://github.com/YaccConstructor/YaccConstructor`.

Table 1: Evaluation results for Query 1 and Query 2

Ontology	#triples	Query 1		Query 2	
		time(ms)	#results	time(ms)	#results
skos	252	10	810	1	1
generations	273	19	2164	1	0
travel	277	24	2499	1	63
univ-bench	293	25	2540	11	81
foaf	631	39	4118	2	10
people-pets	640	89	9472	3	37
funding	1086	212	17634	23	1158
atom-primitive	425	255	15454	66	122
biomedical-measure-primitive	459	261	15156	45	2871
pizza	1980	697	56195	29	1262
wine	1839	819	66572	8	133

We are working on performance improvement by implementation of recently proposed modifications in original GLL algorithm [15,2]. One direction of our research is generalization of grammar factorization proposed in [15] which may be useful for the processing of regular queries which are common in real world application.

Acknowledgments

We are grateful to the Jelle Hellings and Ekaterina Verbitskaia for their careful reading, pointing out some mistakes, and invaluable suggestions. This work is supported by grant from JetBrains Research.

References

1. S. Abiteboul, R. Hull, and V. Vianu. Foundations of databases, 1995.
2. A. Afroozeh and A. Izmaylova. Faster, practical gll parsing. In *International Conference on Compiler Construction*, pages 89–108. Springer, 2015.
3. P. Barceló, G. Fontaine, and A. W. Lin. Expressive path queries on graphs with data. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 71–85. Springer, 2013.
4. C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.
5. J. C. Beatty. Two iteration theorems for the ll (k) languages. *Theoretical Computer Science*, 12(2):193–228, 1980.
6. S. Grigorev and A. Ragozina. Context-free path querying with structural representation of result. *CoRR*, abs/1612.08872, 2016.
7. D. Grune and C. J. H. Jacobs. *Parsing Techniques (Monographs in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
8. J. Hellings. Conjunctive context-free path queries. 2014.

9. J. Hellings. Path results for context-free grammar queries on graphs. *CoRR*, abs/1502.02242, 2015.
10. P. Hofman and W. Martens. Separability by short subsequences and subwords. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 31. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
11. J. G. Rekers. *Parser generation for interactive environments*. PhD thesis, Citeseer, 1992.
12. J. L. Reutter, M. Romero, and M. Y. Vardi. Regular queries on graph databases. *Theory of Computing Systems*, pages 1–53, 2015.
13. E. Scott and A. Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.
14. E. Scott and A. Johnstone. Gll parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.
15. E. Scott and A. Johnstone. Structuring the gll parsing algorithm for performance. *Science of Computer Programming*, 125:1–22, 2016.
16. E. Scott, A. Johnstone, and R. Economopoulos. Brnglr: a cubic tomita-style glr parsing algorithm. *Acta informatica*, 44(6):427–461, 2007.
17. P. Sevon and L. Eronen. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics*, 5(2):100, 2008.
18. D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. Springer, 2012.
19. M. Tomita. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'85, pages 756–764, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.
20. X. Zhang, Z. Feng, X. Wang, G. Rao, and W. Wu. Context-free path queries on rdf graphs. In *International Semantic Web Conference*, pages 632–648. Springer, 2016.