

# Parser-Combinators for Context-Free Path Querying

Smolina  
Saint Petersburg State University  
St. Petersburg, Russia  
trovato@corporation.com

Ilya Kirillov  
Saint Petersburg State University  
St. Petersburg, Russia  
larst@affiliation.org

Ekaterina Verbitskaia  
Saint Petersburg State University  
St. Petersburg, Russia  
webmaster@marysville-ohio.com

Semyon Grigorev  
Saint Petersburg State University  
St. Petersburg, Russia  
semen.grigorev@jetbrains.com

## ABSTRACT

Aaaaabstract! Abstract, abstract, abstract, abstract, abstract, abstract, ab-  
stract, abstract, abstract, abstract, abstract, Abstract, abstract, ab-  
stract, abstract, abstract, abstract, abstract, abstract, abstract, abstract,  
abstract, Abstract, abstract, abstract, abstract, abstract, abstract, abstract, ab-  
stract, abstract, abstract, abstract, Abstract, abstract, abstract, abstract, ab-  
stract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, Ab-  
stract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, ab-  
stract, abstract, abstract, Abstract, abstract, abstract, abstract, abstract, ab-  
stract, abstract, abstract, abstract, abstract, abstract, abstract, Abstract, ab-  
stract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, ab-  
stract, abstract, abstract, abstract, abstract, abstract, Abstract, abstract, ab-  
stract, abstract, abstract, abstract, abstract, abstract, abstract, abstract, ab-  
stract, abstract, abstract, abstract, Abstract, abstract, abstract, abstract, ab-  
stract, abstract, abstract, abstract, abstract, abstract, abstract, abstract,

## CCS CONCEPTS

- **Information systems** → Graph-based database models; Query languages for non-relational engines;
- **Software and its engineering** → Functional languages;
- **Theory of computation** → Grammars and context-free languages;

## KEYWORDS

Graph data bases, Language-constrained path problem, Context-Free path querying, Parser Combinators, Domain Specific Language, Generalized LL, GLL, Neo4J, Scala

### ACM Reference Format:

Smolina, Ekaterina Verbitskaia, Ilya Kirillov, and Semyon Grigorev. 2018. Parser-Combinators for Context-Free Path Querying. In *Proceedings of Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) 2018 (GRADES-NDA'18)*. ACM, New York, NY, USA, Article 4, 4 pages. <https://doi.org/10.475/1234>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

*GRADES-NDA'18, June 2018, Houston, Texas USA*

© 2018 Copyright held by the owner/author(s).

ACM ISBN 123-4567-24-567/08/06...\$15.00

[https://doi.org/10.475/123\\_4](https://doi.org/10.475/123_4)

## 1 INTRODUCTION

Creation of data-centric applications which use graph structured data (graph data bases, social graph, code analysis applications) requires developed in general-purpose languages special languages for graph traversing/querying (SPARQL, cypher, gremlin, etc) , data access, languages integration for graph-structured data (or graph DB) access.

One of type of avigation queries is a language-constrained path querying [?]. Many of languages allow users to specify regular constraints, but some tasca requerd more powerfull – context-free constarints. For example, classical same generation query is not a regular, also CFL reactivity can be used for static code analysis. Not only reachability information, but strucure of results: for debugging and futher processing.

Some different formulation of the similar problems: Language-constrained path querying and language reachability. Language-constrained path querying, Yannakakis [19]. Hellings [6, 7], RDF [20], etc [2, 3, 12, 15, 18]

Integration with general purpose programming languages is a classical problem. Language integration problem: special DSLs for SQL, ORM, Linq. Transparent integration into gp programming language: static correctness, typing, composability. Similar to problem with SQL: correctness, type safety, etc. Special DSL vs Combinators (LINQ [4, 11], etc) cf Sparql — separated language. Special graph query languages. SPARQL, cypher<sup>1</sup>, gremlin [16]. String-embedded DSLs.

Language may be specified with grammar. In this area parser combinator technique is a classical alternative for specialized DSLs for grammar specification. An idea to use combinators for graph traversing proposed in [9], but has some problems: cycles, left-recursive grammars. Parser-combinators is one of classical approach for parsing!!!. Scala combinators for graph [9] — one of attempt to adopt combinators technique for graph processing. Only an idea of combinators using, but language class and restrictions are not discussed. Problems with cycles in graph. Ad-hoc solution. We propose a general solution. Problems with left-recursive grammars(???Should be checked)

Classical combinators are based on LL(k) and has restrictions: left-recursive grammars. GLL [17] can handle arbitrary context-free grammars, SPPF [13]

Structural representation of result: not only reachability, but paths; debuggibg; understanding.

<sup>1</sup><https://neo4j.com/developer/cypher-query-language/>

Combinators for arbitrary grammars on GLL and GLL for graphs. Parser combinators library based on GLL – Meerkat<sup>2</sup> [8]. Can handle arbitrary context-free grammars Written in Scala

We show how to compose this ideas and get general solution for arbitrary CF grammars combinators, structural representation. We make the following contributions in this paper.

- (1) Combinators for CF path querying with structural representation of result. Transparent integration of query language into general-purpose programming language. Compositionality (subquerying mechanism)
- (2) Implementation in Scala. Generalization of linear parsing. Integration with Neo4J graph data base. Available on gitHub:<https://github.com/YaccConstructor/Meerkat>
- (3) Evaluation on realistic data, which shows that it is applicable. Comparison with other tools for CF path querying.

## 2 PARSER-COMBINATORS FOR PATH QUERYING

In this section we present our implementation of and describe some details.

Our implementation is based on Meerkat library. We need only some steps for generalization.

As far as linear input is a one of case of graph, it is possible to provide input abstraction which make possible to generalize combinators.

SPPF may be an arbitrary graph in opposite of linear input parsing.

Let we introduce an example. Graph. Grammar. In terms of combinators.

- Interface for Neo4J data base
- Extensible solution
- An architecture of the solution.

## 3 EVALUATION

In this section we present evaluation of Meerkat graph querying library. We show its performance on a classical ontology graphs for in memory graph and for Neo4j database, show application on may-alias static code analysis problem, and compare with Trails [9] library for graph traversals.

All tests are performed on a machine running Fedora 27 with quad-core Intel Core i7 2.5 GHz CPU with 8 GB of memory.

### 3.1 Ontology querying

One of well-known graph querying problems is a queries for ontologies [1]. We use Meerkat to evaluate it on some popular ontologies presented as RDF files from paper [20]. We convert RDF files to a labeled directed graph like the following: for every RDF triple  $(subject, predicate, object)$  we create two edges  $(subject, predicate, object)$  and  $(object, predicate^{-1}, subject)$ . On those graphs we apply two queries from the paper [5] which grammars are in Fig. 1, and Fig. 2

Meerkat representation of those queries in terms of parser combinators is similar to its EBNF form and presented in Fig. 3 and Fig. 4

$$\begin{aligned} S &\rightarrow subClassOf^{-1} S subClassOf \\ S &\rightarrow type^{-1} S type \\ S &\rightarrow subClassOf^{-1} subClassOf \\ S &\rightarrow type^{-1} type \end{aligned}$$

Figure 1: Query 1 grammar

$$\begin{aligned} S &\rightarrow B subClassOf \\ B &\rightarrow subClassOf^{-1} B subClassOf \\ B &\rightarrow subClassOf^{-1} subClassOf \end{aligned}$$

Figure 2: Query 2 grammar

```
val S: Nonterminal = syn(
  "subclassof-1" ~~ S ~~ "subclassof" |
  "type-1" ~~ S ~~ "type" |
  "subclassof-1" ~~ "subclassof" |
  "type-1" ~~ "type")
```

Figure 3: Meerkat representation of Query 1

```
val S: Nonterminal = syn(
  "subclassof-1" ~~ S ~~ "subclassof" |
  "subclassof")
```

Figure 4: Meerkat representation of Query 2

The queries applied in two following ways.

- Convert RDF files to a graph input for meerkat and then directly parse on query 1 and query 2
- Convert RDF files to a Neo4j database and then parse this database on given queries

Table 1 shows experimental results of those two approaches over the testing RDF files where *number of results* is a number of pairs of nodes  $(v_1, v_2)$  such that exists S-path from  $v_1$  to  $v_2$ .

The performance is about 2 times slower than in [5] and shows the same results. If compare the performance of in memory graph querying and database querying, the second one is slower in about 2 – 4 times.

### 3.2 Static code analysis

Alias analysis is one of the fundamental static analysis problems [10]. Alias analysis checks may-alias relations between code expressions and can be formulated as a Context-Free language (CFL) reachability problem [14]. In that case program represented as Program Expression Graph (PEG) [21]. Vertices in PEG are program expressions and edges are relations between them. In a case of analysing C source code there is two kind of edges **D**-edge and **A**-edge.

<sup>2</sup><https://github.com/meerkat-parser/Meerkat>

Ontology	#triples	Query 1				Query 2			
		#results	In memory graph (ms)	DB query time (ms)	Trails (ms)	#results	In memory graph (ms)	DB query time (ms)	Trails (ms)
atom-primitive	425	15454	174	236	2849	122	49	56	453
biomedical-mesure-primitive	459	15156	328	398	3715	2871	36	52	60
foaf	631	4118	23	42	432	10	1	2	1
funding	1086	17634	151	175	367	1158	18	23	76
generations	273	2164	9	27	9	0	0	0	0
people_pets	640	9472	68	87	75	37	2	3	2
pizza	1980	56195	711	792	7764	1262	44	56	905
skos	252	810	4	29	6	1	0	1	0
travel	277	2499	23	93	34	63	2	2	1
univ-bench	293	2540	19	74	31	81	2	3	2
wine	1839	66572	578	736	3156	133	5	7	4

Table 1: Evaluation results for In Memory Graph and Graph DB

$$M \rightarrow \bar{D} V D$$

$$V \rightarrow (M? A)^* M? (A M?)^*$$

Figure 5: Context-Free grammar for the may-alias problem in syntax

Program	Code Size (KLOC)	Count of aliases		Time (ms)
		M aliases	V aliases	
wc-5.0	0.5K	0	174	350
pr-5.0	1.7K	13	1131	532
ls-5.0	2.8K	52	5682	436
bzip2-1.0.6	5.8K	9	813	834
gzip-1.8	31K	120	4567	1585

Table 2: Running may-alias queries on Meerkat on some C open-source projects

- Pointer dereference edge (**D**-edge). For each pointer deference  $*e$  there is a directed D-edge from  $e$  to  $*e$ .
- Pointer assignment edge (**A**-edge). For each assignment  $*e_1 = e_2$  there is a directed A-edge from  $e_2$  to  $*e_1$ .

Also, for the sake of simplicity, there are edges labeled by  $\bar{D}$  and  $\bar{A}$  which corresponds to reversed D-edge and A-edge, respectively.

The grammar for may-alias problem from [21] presented in Fig. 5. It consists of two nonterminals **M** and **V**. It allows us to make two kind of queries for each of nonterminals **M** and **V**.

- **M** production shows that two l-value expression are memory aliases i.e. may stands for the same memory location.
- **V** shows that two expression are value aliases i.e. may evaluate to the same pointer value.

We made **M** and **V** queries on the code some open-source C projects. The results are presented on the Table 2  
It may be usefull for tools development.

### 3.3 Comparison with Trails

Trails [9] is a Scala graph combinator library. It provides traversers for describing paths in graphs in terms of parser combinators and allows to get results as a stream (maybe infinite) of all possible paths described by composition of basic traversals. Trails as well as Meerkat support parsing in memory graphs, so we compare performance of Trails and Meerkat on the queries from subsection 3.1. Query 1 and Query 2 in terms of Trails are in Fig. 6 and Fig. 7.

Here queries made in same way as in Meerkat. S traversal returns pairs (*begin node*, *start node*) of S-path. Combine operators  $\sim$  and  $\rightarrow$  in queries made in the way to get only last node from path.

```

val B = (out("type-1") ~> out("type")) |
  (out("subclassof-1") ~> B ~> out("subclassof")) |
  (out("type-1") ~> B ~> out("type")) |
  (out("subclassof-1") ~> out("subclassof"))
val S = V ~ B

```

Figure 6: Trails representation of Query 1

```

val B = out("subclassof") |
  (out("subclassof-1") ~> B ~> out("subclassof"))
val S = V ~ B

```

Figure 7: Trails representation of Query 2

The result of comparison are in table 1. Trails gives the same results as Meerkat (column *results* in table 1) but slower than Meerkat.

## 4 CONCLUSION

We propose a native way to integrate language for language-constrained path querying into general purpose programming language. We implement it and show that our implementation can be applied for real problems. Arbitrary context-free grammars for querying.

Code is available on GitHub:  
 Future work is  
 SPPF utilization for debugging and results processing  
 Attributed grammars processing to provide mechanism for semantics calculation

## REFERENCES

- [1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. Foundations of Databases. (1995).
- [2] Pablo Barceló, Gaelle Fontaine, and Anthony Widjaja Lin. 2013. Expressive Path Queries on Graphs with Data. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 71–85.
- [3] Chris Barrett, Riko Jacob, and Madhav Marathe. 2000. Formal-language-constrained path problems. *SIAM J. Comput.* 30, 3 (2000), 809–837.
- [4] James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-integrated Query. *SIGPLAN Not.* 48, 9 (Sept. 2013), 403–416. <https://doi.org/10.1145/2544174.2500586>
- [5] Semyon Grigorev and Anastasiya Ragozina. 2016. Context-Free Path Querying with Structural Representation of Result. *CoRR* abs/1612.08872 (2016). <http://arxiv.org/abs/1612.08872>
- [6] Jelle Hellings. 2014. Conjunctive context-free path queries. (2014).
- [7] Jelle Hellings. 2015. Path Results for Context-free Grammar Queries on Graphs. *CoRR* abs/1502.02242 (2015). <http://arxiv.org/abs/1502.02242>
- [8] Anastasia Izmaylova, Ali Afrozeh, and Tijs van der Storm. 2016. Practical, General Parser Combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. ACM, New York, NY, USA, 1–12. <https://doi.org/10.1145/2847538.2847539>
- [9] Daniel Kröni and Raphael Schweizer. 2013. Parsing Graphs: Applying Parser Combinators to Graph Traversals. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA, Article 7, 4 pages. <https://doi.org/10.1145/2489837.2489844>
- [10] Thomas J. Marlowe, William G. Landi, Barbara G. Ryder, Jong-Deok Choi, Michael G. Burke, and Paul Carini. 1993. Pointer-induced Aliasing: A Clarification. *SIGPLAN Not.* 28, 9 (Sept. 1993), 67–70. <https://doi.org/10.1145/165364.165387>
- [11] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 706–706. <https://doi.org/10.1145/1142473.1142552>
- [12] A. Mendelzon and P. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Computing* 24, 6 (1995), 1235–1258.
- [13] Joan Gerard Rekers. 1992. *Parser generation for interactive environments*. Ph.D. Dissertation. Universiteit van Amsterdam.
- [14] Thomas Reps. 1997. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming (ILPS '97)*. MIT Press, Cambridge, MA, USA, 5–19. <http://dl.acm.org/citation.cfm?id=271338.271343>
- [15] Juan L Reutter, Miguel Romero, and Moshe Y Vardi. 2015. Regular queries on graph databases. *Theory of Computing Systems* (2015), 1–53.
- [16] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. ACM, 1–10.
- [17] Elizabeth Scott and Adrian Johnstone. 2010. GLL parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177–189.
- [18] Petteri Sevon and Lauri Eronen. 2008. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics* 5, 2 (2008), 100.
- [19] Mihalis Yannakakis. 1990. Graph-theoretic methods in database theory. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 230–242.
- [20] Xiaowang Zhang, Zhiyong Feng, Xin Wang, Guozheng Rao, and Wenrui Wu. 2016. Context-free path queries on RDF graphs. In *International Semantic Web Conference*. Springer, 632–648.
- [21] Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 197–208. <https://doi.org/10.1145/1328438.1328464>