From Logic Programming to Programming in Logica: A First-Course in Declarative Data Science & Engineering

Evgeny Skvortsov[1], Yilin Xia[2], Shawn Bowers[3] and Bertram Ludäscher[2]

[1]Google LLC, WA, USA
[2]University of Illinois Urbana-Champaign, School of Information Sciences, IL, USA
[3]Gonzaga University, Department of Computer Science, Spokane, WA, USA

Abstract

While imperative programming is prevalent in software engineering and education, the declarative nature of logic programming can play a vital role in helping students further develop problem-solving and conceptual-modeling skills. Logica, an open-source logic programming language, extends Datalog by incorporating support for numerical computations, including aggregation. It serves as a comprehensive execution environment, compiling programs into iterative SQL queries that can be executed locally via DuckDB or SQLite, or in the cloud through PostgreSQL and Google BigQuery. Logica can also be accessed from within Python and Jupyter Notebooks, allowing it to be used seamlessly within typical data-science tool chains. These intuitive features make Logica an accessible and practical tool for students to learn logic programming. In this paper, we propose a new course, complete with lecture materials centered around Logica, aimed at teaching students declarative data science and engineering while covering topics such as knowledge representation and reasoning, database queries, and constraint-based programming.

1. Introduction

Relational databases and data warehouses are at the center of modern software engineering and data science. These systems are omnipresent, from SQLite databases running on devices, to single remote server systems like PostgreSQL, to serverless data warehouses like Google BigQuery for distributed and parallel query evaluation over thousands of servers. However, it is often not immediately obvious when learning how to access these systems via SQL, SQL-like languages, or programmatic libraries (such as LINQ [1] and Pandas [2]), that there exists a deep connection and dependence on logic and logic programming. In Codd's seminal paper on the relational model [3], he wrote:

> The adoption of a relational model of data, as described above, permits the development of a universal data sublanguage based on an **applied predicate calculus**. [...] Such a language would provide a yardstick of linguistic power for all other proposed data languages, and would **itself be a strong candidate** [...]

Today, SQL is used as the de-facto language for accessing the vast majority of relational data, either explicitly through writing SQL by hand, or indirectly through object-relational models and programmatic library calls. However, the need to develop and deploy more complex data analysis tasks has grown considerably over the last several years with the popularity and wide-scale adoption of machine learning and data-science approaches. For many of these analytics problems, it can be difficult to use SQL, e.g., when performing graph analytics, large scale statistical analyses, and the development and tuning of custom machine-learning models. The reason SQL is not ideal in these cases is that they often require complex recursive (iterative) solutions involving numerical computations, including aggregation. Logic programming, which has many similarities to SQL's underlying predicate calculus model, however, can provide a more natural syntax and framework for implementing these complex data analysis tasks.

Logica[1] [4] is an open-source logic programming language that compiles to (iterated) SQL and allows its users to exploit the power of modern relational databases from the comfort of a full-featured logic

PEG 2024: 2nd Workshop on Prolog Education, October, 2024, Dallas, USA.

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]See https://github.com/EvgSkv/logica

programming syntax. Logica can be viewed as a (bottom-up) Datalog-like language, but with support for numerical computation, including aggregation through recursion. In our experience as educators, students with at least some training in logic programming (either via Prolog, Datalog, or Answer-Set Programming) are able to quickly learn Logica and have a significant productivity boost from using the logic-based programming model in Logica when compared to SQL. We have also observed the opposite: experienced engineers without training in logic programming have a much more difficult time quickly switching from SQL syntax to the syntax of Logica (and similar languages). To help address this discrepancy, we propose a set of eleven modules [5], including examples and exercises, that provide basic training in logic programming through Logica. Taken together, the modules form a fully-developed course that can be taught at the graduate or advanced undergraduate level. Some or all of the modules can also be used as components within other courses as well, e.g., within a block on logic programming within a typical undergraduate programming language concepts course [6], within a course on logic programming (e.g., where a portion of the course can focus on data science applications), within a course on data science and machine learning, or within a traditional algorithms course (where Logica can serve as the implementation language).

While a significant amount of work has been done in the area of logic programming education, e.g., see [7, 8], we believe that the modules proposed here can provide (at least incremental) benefit to this work. In particular, because Logica is integrated with a number of widely-used database systems, can be used within Python and Jupyter notebooks, and focuses on supporting complex data analytics tasks, it provides a practical logic-based language for the data-to-day tasks of a data engineer and data scientist. This emphasis can further attract students who may not have considered or see the benefit of logic programming (and declarative languages more broadly) otherwise.

The rest of this paper is organized as follows. Section 2 gives an overview of each of the eleven course modules. Section 3 describes the potential benefits of learning Logica and logic-programming for data science and similar fields. We conclude with future work in Section 4.

2. Course Overview and Organization

The current iteration of the course is made up of eleven distinct modules, each of which can be divided into one or more lectures (e.g., depending on the pace of the course and if it is offered at a graduate or undergraduate level). At the end of each module (except for the first and last), homework exercises are given. Additionally, labs are also assumed to follow each module, e.g., allowing students to solve problems within groups and to ask questions concerning the homework.

Figure 1 shows the general themes and sequencing of the modules. As shown, the first four modules form a basic introduction to Logica, the next two modules focus on Logica semantics and its relationship to SQL, the next two modules provide extended examples as well as some of the software development features (e.g., unit testing) supported by Logica, and the final three modules focus on advanced aspects of programming in Logica. In addition, each module emphasizes either specific features of Logica (colored in yellow), programming techniques in Logica (colored in green), or how Logica works "under the hood" (colored in blue).

The course described here is targeted at graduate students and advanced undergraduate students studying computer science or a related major such as Information or Data Science. Students are expected to have proficiency in at least one imperative programming language, have completed a college-level discrete math course (that covers formal logic, set theory, etc.), and possess an understanding of basic data science concepts. Ideally, students should have also taken an introductory course in database management. However, it is possible for students without knowledge of SQL to complete the modules. In such cases, a basic introduction to SQL should be provided before starting module 6, or module 6 can be skipped.

Module 1: Introduction. This module provides a general overview and introduction to logic programming and Logica, highlighting that Logica runs on modern SQL engines, which enable it to easily scale to large volumes of data. Simple examples of logic programs and running Logica are also given. In

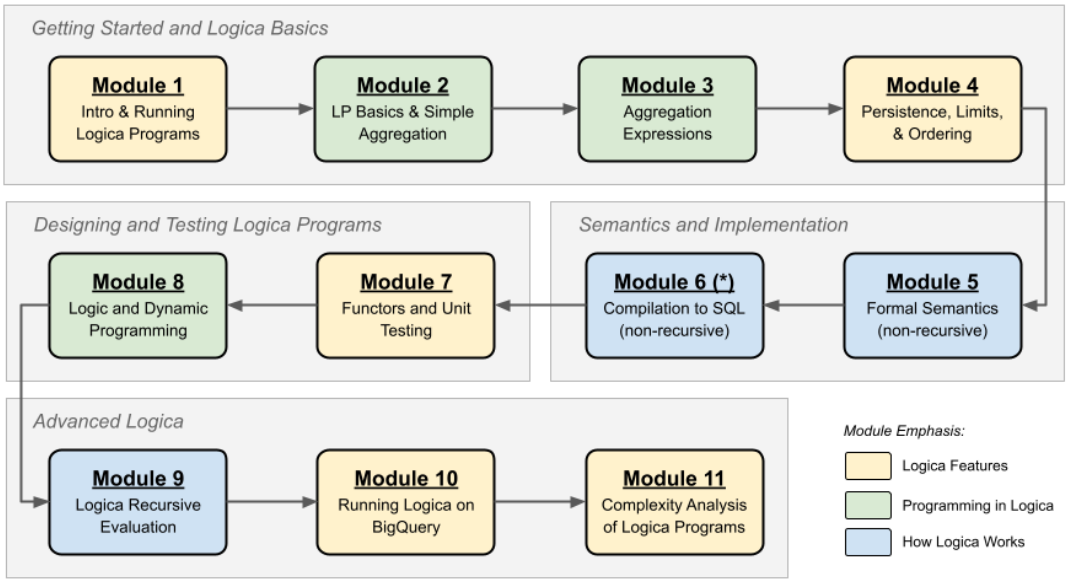

Figure 1: Summary and emphasis of proposed Logica course modules.

particular, it is shown that Logica programs can be run as scripts or from within Jupyter notebooks.

Module 2: Basics – Rules, Data, and Aggregation. This module begins by introducing the concepts of facts and rules. This introduction includes learning about conjunction and disjunction as well as atomic and composite datatypes (lists and records). This module also covers predicate-level aggregation. The material covered allows students to write a large number of simple, but practical programs (e.g., expressed as queries) in Logica.

Module 3: The Secret Sauce – Aggregating Expressions and Functions. This module covers the more advanced Logica features of aggregating expressions and functions. Aggregating expressions provides a mechanism for performing a large number of practical operations over composite data types. Functions are "syntactic sugar" for making logic programming more ergonomic and easier to understand for students with knowledge of conventional programming languages, where functions play a central role. This modules also introduces notions of *injectable* and *concrete* predicates. Logica use a hybrid combination of top-down and bottom-up evaluation strategies. Predicates that correspond to concrete tables are evaluated bottom-up and are called *concrete predicates*. Predicates that are like function evaluations are essentially computed top-down. Technically, this is done by *injecting* them, i.e., replacing their call with their definition. Hence the name *injectable predicates*.

Module 4: Imperatives and Database Connections. In natural language, linguists differentiate between indicative and imperative moods [9]. Sentences in indicative mood state how things are whereas sentences in imperative mood instruct the listener to do something. We use this differentiation in Logica where common predicates that contain the data are in an indicative mood, while predicates in an imperative mood tell the compiler directly to do something. Most importantly, this module details the imperative predicate `@Ground(P)`, which associates the predicate `P` to a physical table in the database. This predicate provides a means for writing Logica program results back into the database where it can be accessed from other systems. The module also describes imperative predicates that allow artificially limiting the number of rows in a table as well as syntax for sorting answers (rows) via Logica. We note that the material in this module is simpler than in modules 2 and 3. While the module covers how predicates are connected to the real world (through database), it also gives them some "breathing room" to internalize logic programming that they were thrown into in the previous two modules.

Module 5: Formal Semantics. Because many logic programs consist of a simple (high-level) syntax

forming a set of relatively simple (if-then style) rules, we expect that students will be able to solve a large number of basic problems in Logica without fully understanding its underlying formal semantics. However, exposing students to the formal semantics of Logica is helpful in that they can confidently reason about complex problems. In this module, we describe multi-sets and how predicate expressions construct new multi-sets from existing ones. Note that, however, at this point in the course, we are not yet considering recursive programs (which introduce additional complexity to the underlying semantics, and described in later modules).

Module 6: Logica Compilation to SQL. As mentioned above, the course assumes that students are familiar with database concepts and SQL. In this module, students are taught how non-recursive rules are compiled to SQL by Logica. Compilation of a single non-aggregating conjunctive rule is straightforward, resulting in a single (trivial) SQL statement. Logica programs involving disjunctive rules and aggregations result in (slightly) more complicated SQL. The translation of Logica to SQL for these classes of rules is simple enough to explain how it is performed by Logica within a single lecture. For courses where the majority of students do not have experience in SQL, additional material can be added (e.g., as a separate lecture prior to starting the module) and/or additional lab time can be spent helping students to learn the basics of SQL. Alternatively, module 6 can be safely skipped without impacting the rest of the course. Note that one purpose of this module is to *connect the dots* and show to students the strong connection between logic programming and relational databases, which is often only done in graduate-level database theory courses.

Module 7: Functors and Unit Testing. While functors and unit testing are distinct topics, they are both relatively small and the unit testing features in Logica rely on functors. Thus, it is convenient to combine them into a single module. Functors in Logica are second-order functions that allow for the reuse of large chunks of logic among tasks. A functor is similar to the notion of first-class functions supported today in most modern programming languages in which functions can be passed as parameter values to other functions (and then be called within the receiving function). While first-class functions support a greater degree of abstraction and code reuse, they can make debugging and design more complicated. Logica takes a middle ground: In a *functor call*, predicates can be used as values, however, there are limitations imposed on functor calls prohibiting their use with recursion. Thus, while functors provide less expressive power than typical first-class functions, they are still able to cover many practical situations for logic reuse. Like design and reuse, unit testing is also an important software engineering practice that is critical for establishing basic correctness and maintaining (e.g., through regression testing) large reliable code bases. Logica supports unit testing natively using additional syntax. This module first describes the notion of functors, their support in Logica, and then unit testing with examples.

Module 8: Recursive Programming. This module begins by describing the use, benefits, and examples of recursion in logic programming. The module starts by explaining how to write recursive programs in Logica, and then takes a hands-on approach by walking students through a number of more involved problems and example solutions leveraging recursive programs. Discussion of how Logica performs evaluation is provided in the next module.

Module 9: Recursion in Depth. This module describes how Logica evaluates recursive predicates. Recursive predicates in Logica are required to be concrete and thus they are always evaluated bottom up. The bottom-up evaluation is carried out in Logica through a simple iteration of rule application. The explanation of recursive evaluation in Logica is expected to take approximately two thirds of the lecture(s) devoted to the module. This includes going over additional examples of recursive logic programs. While less commonly used for basic data analytics tasks, this deep dive into logical recursion is useful as it helps build up problem solving skills and helps students become more efficient in solving easier query-like problems. The techniques in Modules 8 and 9 are also beneficial for encoding more advanced analytical algorithms and solutions that are becoming increasingly more common in modern day data science and engineering.

Module 10: Running Logica on BigQuery. In this module, students are shown how to get started

using Google Cloud Platform (GCP), the basics of BigQuery, and how to run Logica programs using BigQuery via the GCP free tier. With little work students are able to run their programs over a large number of machines, taking advantage of the serverless features of BigQuery. Like Module 4, this module is not as involved technically as the previous modules, providing students with additional "breathing room".

Module 11: Computational Complexity. This module describes how to estimate the computational complexity of the programs written in Logica. A possible source of frustration when using the declarative programming paradigm can be the difficulty in estimating the time-complexity of more-involved programs as well as making the desired program run in a reasonable amount of time. This module helps students to determine the space and time complexity of their programs and how to avoid common programming patterns that can cause inefficiencies.

3. Potential Benefits of Learning Logic Programming via Logica

In this section, we further discuss how teaching declarative languages, logic programming, and Logica in particular, can benefit students interesting in pursuing careers in computing, data science, and software engineering.

Preparing Students for Future Applications. Over the last several years there has been a resurgence of interest in declarative data-oriented languages including Datalog, Prolog, SQL-like data access languages for NoSQL systems, and graph-based languages such as SPARQL [10] and Neo4J's Cypher [11]. Researchers have also explored new and exciting areas of logic programming for a variety of applications, such as data engineering and migration [12], causal reasoning [13], graph queries [14], and recursive computations [15, 16] As the role of data analytics and data science increases within the computer-science industry, and as the problems tackled continue to evolve and increase in complexity, by learning data-oriented declarative languages, students will more easily be able to familiarize themselves and use these (and future) languages throughout their career.

Inclusive Programming Learning Environment. Students in the broader field of information science, who originate from multiple disciplines such as psychology, social science, and traditional library & information science, typically have varying levels of programming expertise, which makes it difficult for instructors to choose an appropriate programming language for their courses. Many imperative programming languages, which seek to provide sufficient power for arbitrary programming tasks, make data-specific programming unnecessarily difficult (or at least act as an additional "hurdle") for students without formal training. In contrast, declarative languages are more inclusive because they have typically been simplified to avoid syntactic elements, making them more similar to natural languages, and designed to prevent users from becoming overwhelmed by syntax errors [17]. Much like SQL for basic data management and analytical tasks, learning Logica can empower students to more quickly solve complex data science tasks when compared to having to learn the details of imperative languages first.

Natural Method for Learning Knowledge Representation and Reasoning. Knowledge representation and reasoning enables artificial intelligence systems to display a wide variety of knowledge structures and relationships, making them crucial to information science and data science. As illustrated by Levesque and Brachman [18], knowledge bases can typically be represented in first order logic, allowing declarative languages such as Datalog to represent complex relationships, rules, and constraints in a more natural manner. Despite the application contexts being in chemistry and physics, the cases presented by Zhang et al. [19] demonstrated that declarative programming languages make it easy and natural for students to transform domain-specific knowledge into logic programming rules and answer questions based on the rules. This suggests that declarative languages have the potential to effectively teach knowledge representation and reasoning skills for information and data science students.

Improve Problem Solving Skills. To implement solutions in declarative languages, students will need

to identify the problem's fundamental objects, relationships, and constraints. Regarding the food chain example provided by Zhang et al. [19], students were given four species including eagle, snake, rabbit and carrot, along with two types of relationship—*feedsOn(X,Y)* and *extinct(X)*. The objective was to identify all extinct species. Students must deduce the potential relationship between *feedsOn(X,Y)* and *extinct(X)* from the descriptions provided to solve the problem. As the study later shows, declarative languages indeed help students better understand science topics, which shows possibility that they could assist students in the data sciences as well.

Better Understanding of Logic and Formal Methods. Declarative languages are based on formal logic principles. By studying these languages, students are exposed to concepts such as conjunction, disjunction, negation, etc., which are necessary for comprehending formal methods. In addition, declarative programming languages, such as SQL, Datalog, and Answer Set Programming, require students to concentrate on defining the desired outcome rather than the process of accomplishing it [20]. This strategy promotes critical thinking and enables students to gain a deeper understanding of problems and their solutions (without getting bogged down in syntax or how to implement the solution using imperative constructs), which is consistent with the fundamental principles of formal methods. On top of that, declarative languages can help students develop a systematic and rigorous approach to problem solving, which is essential for working with formal methods in computer science and prepares students in the data sciences for success within their careers.

Ease of Maintenance and Sharing. Declarative languages are typically more concise and simpler to maintain than imperative languages, as they describe the desired outcome as opposed to the specific steps required to achieve it. This attribute can result in code that is more maintainable, which is advantageous in a professional setting [17]. Therefore, students studying the data sciences can use these languages to more effectively communicate their problem-solving methods and collaborate with individuals of varying levels of expertise.

4. Conclusion and Future Work

We propose a set of open-source educational modules for teaching logic programming using Logica, a full-featured declarative (logic programming) language designed to leverage modern SQL engines and to support data analytics and data science applications. The modules can be used within a self-contained course on logic programming for (declarative) data science, and have the potential to be reused within a number of other types of courses. An overview of the modules is provided and a brief discussion of the potential benefits of using Logica to teach logic programming within the context of information and data sciences are presented. As future work, we plan to leverage the modules in our existing courses, collect data on their effectiveness, and continue to update and improve them based on our experience and results.

Another possible venue of the future work is using Logica in advanced courses about Logic Programming and non-monotonic reasoning. In particular Logica can be viewed as a general non-monotonic reasoning engine. We believe that theoretical connection of the extension of predicate caclulus with aggregation and the non-monotonic reasoning could can be further improved by using Logica as a formalism.

References

[1] E. Meijer, B. Beckman, G. Bierman, Linq: Reconciling object, relations and xml in the .net framework, in: Proceedings of the ACM SIGMOD/PODS International Conference on the Management of Data, 2006, p. 706.

[2] Wes McKinney, Data Structures for Statistical Computing in Python, in: Proceedings of the Python in Science Conference, 2010, pp. 56–61.

[3] E. F. Codd, A relational model of data for large shared data banks, Communications of the ACM 13 (1970) 377–387.
[4] E. Skvortsov, Y. Xia, B. Ludäscher, Logica: Declarative data science for mere mortals, in: Proceedings of the International Conference on Extending Database Technology (EDBT), 2024, pp. 842–845.
[5] E. Skvortsov, Y. Xia, S. Bowers, B. Ludäscher, Logica Education Course, 2024. URL: https://tinyurl.com/LogicaCourse2024.
[6] A. N. Kumar, R. K. Raj, S. G. Aly, M. D. Anderson, B. A. Becker, R. L. Blumenthal, E. Eaton, S. L. Epstein, M. Goldweber, P. Jalote, D. Lea, M. Oudshoorn, M. Pias, S. Reiser, C. Servin, R. Simha, T. Winters, Q. Xiang, Computer Science Curricula 2023, ACM, 2024.
[7] V. Dahl, D. Cukierman, G. Bel-Enguix, M. D. Jiménez-López, Logic programming: Teaching strategies for students with no programming background, Proceedings of the 15th Western Canadian Conference on Computing Education (WCCCE) (2010).
[8] S. Yang, M. Joy, Approaches for learning prolog programming, Innovation in Teaching and Learning in Information and Computer Sciences 6 (2007) 88–107.
[9] C. Han, The Structure and Interpretation of Imperatives: Mood and Force in Universal Grammar, Outstanding dissertations in linguistics, Garland Science, 2000.
[10] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of sparql, ACM Transactions on Database Systems (TODS) 34 (2009) 1–45.
[11] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor, Cypher: An evolving query language for property graphs, in: Proceedings of the ACM SIGMOD/PODS International Conference on the Management of Data, 2018, p. 1433–1445.
[12] Y. Wang, R. Shah, A. Criswell, R. Pan, I. Dillig, Data migration using datalog program synthesis, Proceedings of the VLDB Endowment 13 (2020) 1006–1019.
[13] B. Salimi, H. Parikh, M. Kayali, L. Getoor, S. Roy, D. Suciu, Causal relational learning, in: Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data, 2020, pp. 241–256.
[14] L. Jachiet, P. Genevès, N. Gesbert, N. Layaïda, On the optimization of recursive relational queries: Application to graph queries, in: Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data, 2020, pp. 681–697.
[15] M. A. Khamis, H. Q. Ngo, R. Pichler, D. Suciu, Y. Remy Wang, Datalog in wonderland, ACM SIGMOD Record 51 (2022) 6–17.
[16] Y. R. Wang, M. Abo Khamis, H. Q. Ngo, R. Pichler, D. Suciu, Optimizing recursive queries with progam synthesis, in: Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data, 2022, p. 79–93.
[17] C. Kelleher, R. Pausch, Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers, ACM Computing Surveys (CSUR) 37 (2005) 83–137.
[18] H. J. Levesque, R. J. Brachman, Expressiveness and tractability in knowledge representation and reasoning 1, Computational intelligence 3 (1987) 78–93.
[19] Y. Zhang, J. Wang, F. Bolduc, W. G. Murray, W. Staffen, A preliminary report of integrating science and computing teaching using logic programming, in: Proceedings of the AAAI Conference on Artificial Intelligence, 2019, pp. 9737–9744.
[20] M. Hanus, Multi-paradigm declarative languages, in: Proceedings of the International Conference on Logic Programming (ICLP), 2007, pp. 45–75.