

From Formalization to Execution: EIDOS as an Active Learning Environment for Context-Free Grammars

Belmondo Rodrigues Aragão Junior¹

¹University of Fortaleza (UNIFOR)
Fortaleza – CE – Brazil

belmondo@unifor.br

Abstract. *The teaching of Context-Free Grammars in undergraduate Computer Science courses remains predominantly theoretical, creating a gap between formal specification and the concrete experience of execution: students write grammars on paper but rarely experience them as the specification of an executable language. This paper presents EIDOS, a VS Code extension that lets students describe a CFG in BNF or EBNF, map each production rule to professor-predefined semantic primitives, and execute programs in the language they created, with real-time feedback on output, variable state, and derivation tree. The artifact was validated through 205 automated test cases; the empirical evaluation with students is identified as the necessary next step.*

1. Introduction

Context-Free Grammars (CFGs) hold a central position in the undergraduate Computer Science curriculum. Formalized by Chomsky within the hierarchy of formal languages [Chomsky 1956], CFGs constitute the theoretical framework upon which rest the syntactic definition of programming languages, the construction of syntactic analyzers, and the development of compilers and interpreters [Sipser 2013, Hopcroft et al. 2006]. Courses such as Formal Languages and Automata and Theory of Computation are responsible for introducing this formalism, typically through activities involving BNF (Backus-Naur Form) or EBNF (Extended BNF) specification, derivations, and syntactic tree analysis.

The ability to design and interpret grammars is therefore a fundamental competency for the formation of Computing professionals, yet CFG teaching remains predominantly theoretical and disconnected from execution experience. Students specify grammars on paper, construct derivation trees, but they rarely experience their grammar functioning as what it formally defines: the syntactic specification of an executable programming language. This one-sided abstraction produces a gap between symbolic formalization and the intuition that a grammar specifies something computationally real.

The Computer Science Education literature documents that courses with strong formal content, such as automata theory, logic, and formal languages, present failure and disengagement rates systematically higher than other courses, partly because students cannot assign immediate practical meaning to the formalisms studied [Gomes and Mendes 2007, Robins et al. 2003]. Tools such as JFLAP [Rodger and Finley 2006] have advanced toward interactive visualization of derivations and automata, but remain on the plane of syntactic recognition, without offering students the experience of executing programs written in the language they themselves created. Parser-generation environments such as ANTLR [Parr 2013] and Yacc/Bison allow the

construction of complete compilers, but require technical knowledge that exceeds the scope of theoretical courses, shifting focus from the grammar to the implementation infrastructure. There exists, therefore, a gap: no environment has been identified, to the best of our knowledge, that allows students to start from BNF/EBNF notation, exactly as it appears in area textbooks, and execute programs written in the language they themselves formalized, while remaining at the level of abstraction compatible with the theoretical course.

This paper presents EIDOS (Execution Interface for Designing and Operating Syntax), a Visual Studio Code extension enabling a new pedagogical approach to CFG teaching: the student describes a grammar in BNF or EBNF, executes programs written in the language they created, and observes in real time the output, variable state, and generated derivation tree. Its semantic primitives are parameterizable at two levels: an expression level (arithmetic, relational, and logical operations) and an imperative level (assignment, sequencing, conditionals, loops, and I/O). The main contributions are: (i) a pedagogical approach based on creating and executing custom languages for teaching CFGs in undergraduate theoretical courses; (ii) a declarative semantic mapping model connecting BNF/EBNF rules to operational primitives at the theoretical level of the course; and (iii) the EIDOS artifact itself. Methodologically, this is an emerging-results / work-in-progress contribution following an artifact-centered (design-oriented) stance, reporting the design, implementation, and technical validation of the tool; empirical evaluation of learning outcomes with students remains future work. Methodologically, this paper is positioned as an emerging-results / work-in-progress contribution: it reports the design, implementation, and technical validation of an educational artifact, following an artifact-centered (design-oriented) research stance. The empirical evaluation of learning outcomes with students is a future work.

The remainder of the paper is organized as follows: Section 2 presents the theoretical foundation and details the pedagogical problem; Section 3 discusses related work; Section 4 describes the approach and the tool; Section 5 presents the emerging results and evaluation protocol; and Section 6 offers final considerations.

2. Background

2.1. Context-Free Grammars in the Computing Curriculum

Context-Free Grammars (CFGs) integrate the second level of the Chomsky Hierarchy [Chomsky 1956] and constitute the generative formalism of context-free languages, recognized by pushdown automata. From a curricular standpoint, CFGs occupy a strategic position: they are simultaneously objects of study in automata theory, where their closure properties, undecidability, and complexity are of interest, and practical instruments in software engineering, where they define the syntax of programming languages through notations such as BNF [Backus 1959] and EBNF [Wirth 1977]. This dual nature, theoretical and applied, is precisely what makes the teaching of CFGs simultaneously relevant and challenging: it requires students to transition between the formal rigor of theory and practice.

The ACM/IEEE curriculum guidelines for Computing courses [ACM/IEEE-CS Joint Task Force on Computing Curricula 2013] position topics on formal languages, grammars, and automata as mandatory components of a Computer Science

education, emphasizing that their comprehension is a prerequisite for compiler courses, natural language processing, and formal program verification. In Brazil, the National Curriculum Guidelines for Computing courses [Brasil. Ministério da Educação 2016] reiterate this centrality by including formal languages and theory of computation among the essential content of Computer Science and Computer Engineering programs. Despite this relevance, the Computer Science Education literature documents that courses with strong formal content present lower performance and engagement indicators than other courses, configuring a pedagogical problem [Gomes and Mendes 2007, Robins et al. 2003].

2.2. The Gap Between Formalization and Execution

The dominant practice in CFG teaching has students write production rules, perform derivations, build syntactic trees, and check ambiguity, yet the final product is a symbolic object, a sequence of rules, a drawn tree, a step-by-step derivation. The grammar is never experienced as what it formally defines: a sentence generator of a language. This gap between the symbol and its computational interpretation has documented consequences. [Gomes and Mendes 2007] identify students' inability to connect abstract representations to their concrete effects in computational systems as a central difficulty, and [Robins et al. 2003] concludes that learning formalisms is substantially facilitated when students directly observe the consequences of their formal choices, that is, when the formalism produces an artifact with observable behavior. For CFGs, this implies that students should be able to write a grammar and immediately verify that it accepts valid sentences, rejects invalid ones, and produces the expected derivation structure.

Directly visualizing grammar behavior, recognition, tree generation, and program execution, would reduce this extraneous load. The meta-analysis of 225 studies by [Freeman et al. 2014] shows that immediate feedback on student production yields statistically significant performance gains over expository instruction, and [Prince 2004] links the effectiveness of active methodologies to short feedback cycles: the faster the response about correctness, the more effective the learning.

3. Related Work

JFLAP (Java Formal Languages and Automata Package) [Rodger and Finley 2006] is the most widely adopted educational tool for teaching formal languages, with registered use in hundreds of higher education institutions worldwide [Rodger et al. 2009]. JFLAP offers visual support for the construction and simulation of deterministic and non-deterministic finite automata, pushdown automata, Turing machines, and grammars, allowing students to visualize derivations step by step and verify string recognition. Its pedagogical contribution is significant and well-documented: the ability to observe the execution of an automaton or the derivation process of a grammar [Chakraborty et al. 2014]. However, JFLAP operates exclusively on the plane of syntactic recognition. Students can verify whether a string belongs to the language generated by their grammar, but cannot write programs in that language and execute them.

In the domain of Domain-Specific Languages (DSLs), tools such as Xtext [Bettini 2013] and MPS [Völter et al. 2013] allow developers to define grammars and automatically obtain editors, validators, and interpreters for the resulting language. These tools are conceptually closer to a pedagogical approach, however, their target audience is

experienced software engineers, not undergraduate students in early stages: the learning curve is substantial and the notation used diverges from the BNF/EBNF of textbooks.

4. EIDOS

4.1. Overview of the Approach

EIDOS structures the student's experience in a three-phase cycle that integrates formalization, semantic mapping, and execution in a single development environment. Figure 1 illustrates this cycle.

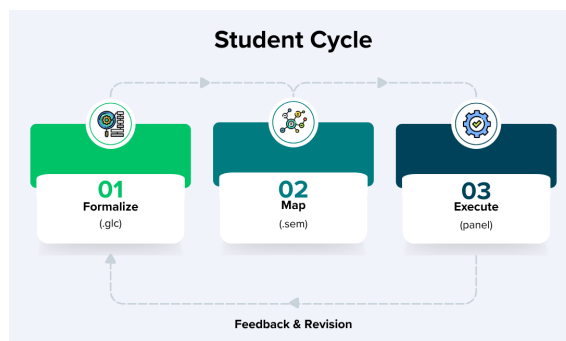


Figure 1. EIDOS usage cycle

In the first phase, the student formalizes their CFG in a .glc file using BNF or EBNF notation directly in the VSCode editor. In the second phase, it is necessary to create a .sem file with the same name, in which they map each production rule to a semantic primitive. This mapping is the moment at which the grammar acquires computational behavior: each production alternative becomes associated with an operation, such as a sum, an assignment, a loop. Finally, in the third phase, the student opens the EIDOS execution panel (Figure 2), writes a valid program in the language they created, and executes it, observing in real time the produced output, variable state, and the derivation tree generated by the parser.

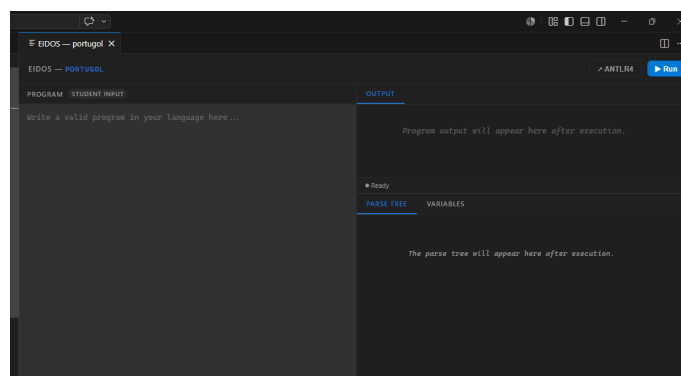


Figure 2. EIDOS execution panel

The Professor's role in this cycle is twofold. Before the activity, they may configure or provide the .sem file, delimiting the scope of the language students will create. During and after the activity, they use the .glc and .sem files as evaluation instruments to verify the formal correctness of the grammar, the coherence of the semantic mapping, and the expressiveness of the created language.

4.2. Notation and Semantic Mapping Model

EIDOS adopts a notation convention that unifies BNF and EBNF into a single format, respecting the representations already familiar to students from area textbooks [Sipser 2013, Hopcroft et al. 2006]. Table 1 presents the notation elements and their visual conventions.

Table 1. Notation conventions for the .glc file.

Element	Notation	Example
Non-terminal (BNF)	< >	<expr>
Non-terminal (EBNF)	lowercase	expr
Lexical token	UPPERCASE	INTEGER, ID
Keyword	double quotes	"if", ";"
Alternative		<a>
Zero-or-more	* / {}	<cmd>*
One-or-more	+	<cmd>+
Optional	? / []	["else" <cmd>]
Grouping	()	(<a>)+
Comment	//	// description

The distinction between pre-defined lexical tokens (written in UPPERCASE - Figure 3a), and non-terminals (written in lowercase or between < > - Figure 3b), eliminates the ambiguity that would arise if both used the same typographic style. The available pre-defined lexical tokens are: INTEGER (digit sequence), REAL (decimal number), ID (alphanumeric identifier), TEXT (double-quoted string), and BOOLEAN (the literals true and false). Every .glc file must declare two mandatory directives in the header: *@nivel*, which defines the set of available semantic primitives (e.g., Imperative), and *@inicio*, which indicates the grammar's start symbol (Figure 3c).

<pre><fator> ::= INTEIRO REAL ID "(" <expressao> ")"</pre>	<pre><expressao> ::= <expressao> "=" <soma> <expressao> "!=" <soma> <expressao> ">" <soma> <expressao> "<" <soma> <expressao> ">=" <soma> <expressao> "<=" <soma> <soma></pre>
(a) Uppercase Example	(b) non-terminals example
<pre>@nivel: imperativo @inicio: <programa> // — Estrutura principal — <programa> ::= "inicio" <bloco> "fim" <bloco> ::= <comando>+ // — Comandos disponíveis — <comando> ::= <atribuicao> <impressao> <leitura> <condicional> <repeticao></pre>	
(c) header configuration example	

Figure 3. Configuration of File .sem example

4.3. Semantic Mapping Model

The .sem file associates each production rule from the .glc file with a semantic primitive through mapping blocks introduced by the keyword map. Each rule alternative is mapped to a primitive on a separate line, using the $\rightarrow$ operator. The primitive receives as arguments the names of the relevant non-terminals or lexical tokens in the alternative, in the

order they should be evaluated. Figure 4 presents an example of an arithmetic expression grammar and its corresponding semantic mapping.

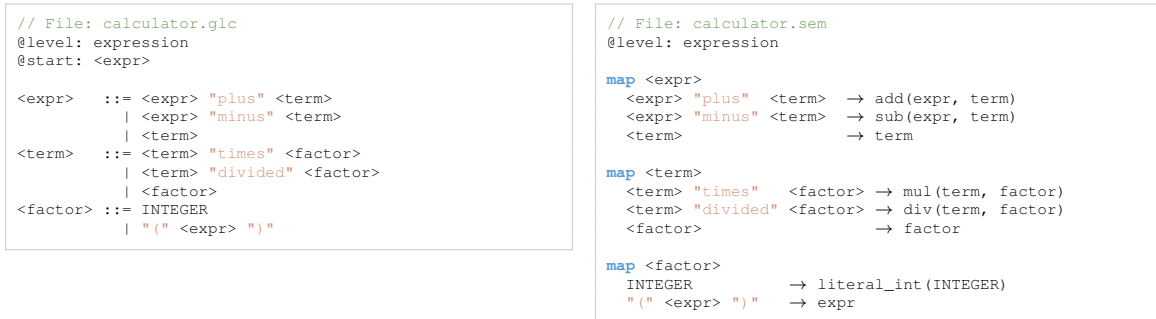


Figure 4. Example of an arithmetic expression grammar (left) and its corresponding semantic mapping (right).

With these two files defined, the student can open the execution panel and submit expressions written in the syntax they designed. The expression 3 plus 4 times 2, for example, is recognized by the dynamic parser according to the student's grammar, which establishes the precedence of times over plus through the hierarchy between `<expr>` and `<term>`. The execution engine traverses the resulting derivation tree, applies the mapped primitives at each node (`mul(4, 2)` before `add(3, 8)`) and produces the result. The student observes not only the numerical result, but the complete derivation tree that makes explicit how the grammar they wrote determined the evaluation order of the expression.

The available semantic primitives are organized into two cumulative levels. The expression level offers arithmetic operations (add, sub, mul, div, mod, pow, neg), relational (eq, neq, lt, gt, leq, geq), logical (and, or, not), string (concat), and literals (literal_int, literal_real, literal_bool, literal_str). The imperative level adds the primitives var (variable read), assignment (assignment), sequence (command sequencing), if_then, if_then_else (conditional), while_do, for_do (iteration), print, print_line (output), read (input), and block (scoped grouping). This division into levels allows the professor to calibrate activity complexity to the stage of the course: early-stage classes can work exclusively with the expression level, while more advanced classes can explore the full imperative level.

4.4. Execution Pipeline

Figure 5 illustrates the execution pipeline of EIDOS.

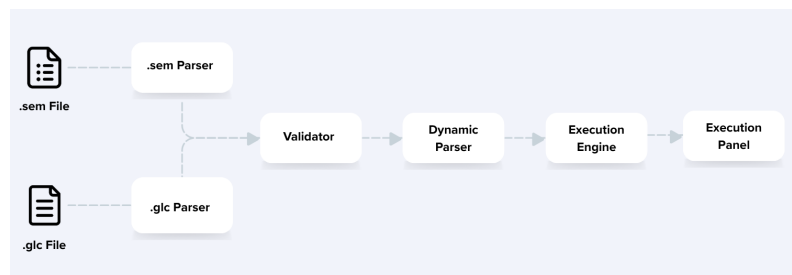


Figure 5. EIDOS execution pipeline

The .glc Parser reads the student's grammar file and produces a typed internal object representing the grammar(rules, alternatives, symbols, and directives). The Validator cross-references the grammar object with the corresponding .sem file, verifying structural consistency, non-terminal reachability, and primitive compatibility with the declared level, and publishes the resulting diagnostics directly in the VSCode editor. The Dynamic Parser consumes the grammar object and constructs a recursive-descent syntactic analyzer specific to the student's language. The Execution Engine traverses the derivation tree produced by the dynamic parser and applies the semantic primitives defined in the .sem file, maintaining an execution state that records variables, output, and step count. The Execution Panel, implemented as a VSCode WebView, presents to the student the program input area, the produced output, the colored derivation tree, and variable state at execution completion (Figure 6).

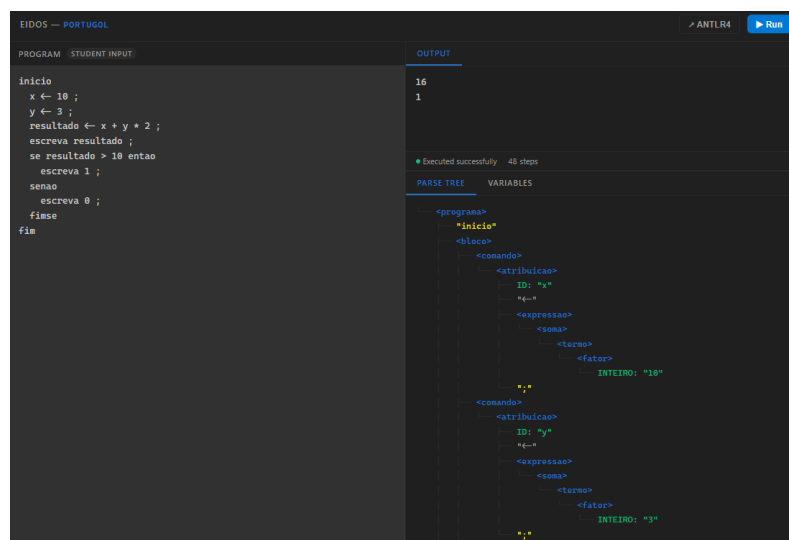


Figure 6. EIDOS Running Example

5. Emerging Results and Validation of the Artifact

EIDOS was developed following a test-driven approach, with each pipeline component verified incrementally before integration into the next. The tool comprises 205 automated test cases organized into five suites (Table 2): the GLC Parser suite (26 cases), covering recognition of BNF and EBNF grammars; the Syntax Highlighting suite (81 cases), covering the regular-expression patterns; the Validator suite (27 cases), covering cross-validation between .glc and .sem files; the Transpiler and Dynamic Parser suite (42 cases), covering grammar generation, dynamic lexer tokenization, and left-recursion elimination; and the Execution Engine suite (29 cases), covering the complete .glc/.sem-to-output pipeline across four language categories: arithmetic expressions with operator precedence, logical and relational expressions, imperative languages with assignment and output, and imperative languages with conditionals and loops. All 205 cases pass, demonstrating the technical consistency of the artifact.

5.1. Threats to Validity and Current Limitations

The evaluation reported in this section is strictly technical: it establishes the correctness and internal consistency of the artifact, not its pedagogical effectiveness. The automated test cases provide evidence that EIDOS reliably parses BNF/EBNF grammars,

Table 2. Distribution of test cases by suite and component.

Suite	Component	Cases	Verified Categories
GLC Parser	Parser + structural validator	26	BNF, EBNF, directives, diagnostics
Syntax Highlighting	.glc and .sem tmLanguage	81	Colorization patterns, absence of false positives
Validator	Cross-validation .glc/.sem	27	Synchronization, primitives, cache
Transpiler + Dynamic Parser	Parsing engine	42	ANTLR4, lexer, parser, left recursion
Execution Engine	Complete pipeline	29	Expressions, imperative, execution errors
Total		205	

validates the grammar–semantics mapping, and executes the resulting languages, but they do not constitute evidence of learning gains, engagement, usability, or classroom adoption. Consequently, the characterization of EIDOS as an active learning environment is grounded in the design rationale connecting immediate feedback to reduced extraneous load [Prince 2004, Freeman et al. 2014], rather than in observed student outcomes. This work is explicitly framed as an emerging result: the artifact is functionally complete and technically validated, while its educational evaluation remains as future work. A controlled classroom study is required before any claim of pedagogical efficacy can be sustained.

6. Conclusion and Future Works

This work presents EIDOS, a VSCode extension that allows students to formalize their CFG in BNF or EBNF, map each production rule to professor-predefined semantic primitives, and execute codes written in the language they created, with real-time feedback on output, variable state, and derivation tree. The technical viability of the approach was demonstrated through 205 automated test cases organized in five suites covering all pipeline components, with a 100% pass rate.

The contributions of this work are: i) a pedagogical approach was proposed in which students create and execute their own programming languages as the primary activity for learning CFGs in undergraduate theoretical courses, shifting the student's role from passive analyst of grammars to active designer of languages; ii) a declarative semantic mapping model was designed and implemented, connecting BNF/EBNF production rules to operational primitives in a manner that remains accessible at the theoretical level of the course, without requiring students to implement an interpreter; and Finally, iii) these contributions were materialized in EIDOS, a VSCode extension that is technically validated.

The primary limitation of this work is the absence of an empirical evaluation with students: the validation was strictly technical, and the pedagogical effectiveness of EIDOS as an active learning environment remains unobserved. This defines our immediate research agenda. We plan a classroom study contrasting EIDOS-based grammar-design activities with traditional paper-based exercises, combining pre-/post-tests of grammar comprehension, perception, and qualitative observation of the difficulties students face when moving from formalization to execution. In parallel, we intend to gather professors' feedback on the effort of authoring .sem scopes and to publish the extension on the VSCode Marketplace, enabling broader adoption and independent evaluation.

Artifact Availability

The source code of the EIDOS extension, and the example grammars used are maintained in a public repository under an open-source license (MIT), accessible at: <https://github.com/Belmondo/EIDOS>.

Declaration on the Use of Artificial Intelligence

During the writing of this article, a Generative Artificial Intelligence tool was used to support textual revision and manuscript organization. All technical, conceptual, and methodological content presented is entirely authored by and remains the sole responsibility of the authors, who critically reviewed the outcome of each interaction. The tool was not used to generate data, results, or references cited in this article.

References

- ACM/IEEE-CS Joint Task Force on Computing Curricula (2013). Computer science curricula 2013. Technical report, ACM Press and IEEE Computer Society Press.
- Backus, J. W. (1959). The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132, Paris, France. UNESCO.
- Bettini, L. (2013). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, Birmingham, UK.
- Brasil. Ministério da Educação (2016). Diretrizes curriculares nacionais para os cursos de graduação em computação. Resolução CNE/CES nº 5, Conselho Nacional de Educação, Brasília, DF.
- Chakraborty, S., Kulkarni, A., and Bhattacharyya, P. (2014). Interactive simulation of formal language theory: A pedagogical tool. In *Proceedings of the International Conference on Technology for Education (T4E)*, pages 66–69. IEEE.
- Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.
- Freeman, S., Eddy, S. L., McDonough, M., Smith, M. K., Okoroafor, N., Jordt, H., and Wenderoth, M. P. (2014). Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences*, 111(23):8410–8415.
- Gomes, A. and Mendes, A. J. (2007). Learning to program — difficulties and solutions. In *Proceedings of the International Conference on Engineering Education (ICEE)*, Coimbra, Portugal.
- Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison-Wesley, Boston, MA, 3 edition.
- Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Dallas, TX.
- Prince, M. (2004). Does active learning work? A review of the research. *Journal of Engineering Education*, 93(3):223–231.
- Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172.

- Rodger, S. H. and Finley, T. W. (2006). *JFLAP: An Interactive Formal Languages and Automata Package*. Jones & Bartlett Publishers, Sudbury, MA.
- Rodger, S. H., Wiebe, E., Lee, K. M., Morgan, C., Omar, K., and Tan, J. (2009). Increasing engagement in automata theory with JFLAP. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 403–407, Chattanooga, TN, USA. ACM.
- Sipser, M. (2013). *Introduction to the Theory of Computation*. Cengage Learning, Boston, MA, 3 edition.
- Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., Visser, E., and Wachsmuth, G. (2013). *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.
- Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823.