Lit Review: Domain Specific Language Teaching Tool

Ahmed Ghoor
Department of Computer Science,
University of Cape Town,
Private Bag X3, Rondebosch, 7701,
South Africa
ghrahm004@myuct.ac.za

ABSTRACT

A proposed research project is looking to create and assess a simple interactive tool for developing a Domain Specific Language as part of the process of learning Compiler Theory. This review investigates similar research on different approaches to incorporating the practical component in the teaching of compiler theory and analyses them based on the latest relevant concepts in Computer Science Education to identify gaps in current solutions.

Keywords

Computer Science Education, Interactive Tools, Domain Specific Languages, Compiler Theory

1. INTRODUCTION

1.1 What is the project

A proposed research project is looking to create and assess a simple interactive tool for developing a Domain Specific Language as part of the process of learning Compiler Theory. The development of Domain Specific Languages is not a new approach to teaching compiler theory and several tools for automating much of the process are already available [1]. The use of interactive tools to assist in the learning process is also not new in the field of Computer Science Education or even Compiler Theory Education [2]. By drawing on established educational theory, and the current research completed in these fields, this project will look to find the gap and address it, potentially building on an existing implementation.

1.2 Why is it important

Compiler Theory is arguably a very important topic in Computer Science Education. It teaches students how high-level source code gets translated, through a pipeline of processes, to low-level machine code that computers can understand and execute.

1.2.1 Challenges

Despite the importance of the subject, there are some challenges that come with teaching compiler theory.

Dealing with low-level translation process concepts can be complicated, and some of the theory is quite abstract [3]. Simplified practical problems that help students actively engage with the material need to be created to assist students' understanding of these concepts. As we will show later in the paper, coming up with this practical problem is the subject of much debate.

Secondly, students are not always interested in the subject. The theory does not seem to have immediate or obvious market relevance. In other words, the likelihood that they would be using this skill set in a job seems low to students. [1]

1.3 Where does this research fit

The development of a Domain Specific Language (DSL) teaching tool aims to address these challenges.

By providing an interactive educational tool, we can potentially facilitate an engaging visual learning experience that simplifies the process of understanding and applying Compiler Theory concepts [3].

And by taking the approach of teaching Compiler theory using DSLs, we can potentially highlight examples of the practical relevance of Compiler Theory in the industry, thereby increasing students' interest in the subject [1].

We will expand upon both of the above arguments later in the paper.

1.4 Where does this review fit

This review looks to investigate similar research on different approaches to incorporating the practical component in the teaching compiler theory and analysing them based on the latest relevant concepts in Computer Science Education. This is necessary to identify gaps in current solutions. The review will also explore methods of evaluating educational tools so that they can be applied to the final DSL Tool. Doing this before the tool is built may help mitigate bias.

1.5 Different subsections of this review

Each sub-section plays an important role in laying the groundwork for the development of an effective DSL teaching tool for Compiler Theory.

1.5.1 Computer Science Education

This review will start by investigating the relevant pedagogical theories and best practices in Computer Science Education, with a particular focus on the use of interactive techniques to enhance students' understanding of complex concepts. This is essential for designing a DSL teaching tool that aligns with established pedagogy and effectively helps achieve the required learning outcomes.

1.5.2 Compiler Theory

This next sub-section will look to explore an overview of the key concepts taught in Compiler Theory courses, with a focus on Domain Specific Languages and the tools used to build them. Understanding the course content is necessary for ensuring that the DSL Teaching Tool is able to bridge the gap between the complex abstract theory and the practical component that attempts to help facilitate the understanding of that theory.

1.5.3 Teaching Compiler Theory

The review will then examine the different approaches that have been taken to incorporate a practical component in the teaching of

the Compiler Theory above. This will include a special focus on the past use of domain-specific languages and interactive teaching tools. By analyzing the different approaches, we can identify the most effective and relevant strategies that could be incorporated into the DSL teaching tool.

1.5.4 Creating and Assessing a DSL Teaching Tool

This section draws on the above literature to highlight potential key considerations and success factors for the DSL Tool. It also explores methods for evaluating teaching tools that can be used to assess the proposed DSL Teaching Tool once it is completed.

2. Computer Science Education (CSE)

In the field of Computer Science Education (CSE), there is a lot of literature that could potentially be relevant and applicable to teaching Compiler Theory. Additionally, numerous general educational theories and approaches may also have implications for CSE. However, given that the primary focus of this paper is not to provide an exhaustive literature review on the philosophies of CSE, we have limited our investigation to concepts that may be particularly pertinent to the teaching of Compiler Theory. Some of these concepts will be referenced later in the paper, which will explicitly illustrate their relevance.

2.1 Key Theories in CSE

2.1.1 Constructivism

A learning theory that emphasizes the active role of the learner in constructing their own understanding of the subject matter. Students don't just passively receive information. Rather, they tend to build their own understanding of new knowledge upon pre-existing knowledge by constructing mental models.[4]

2.1.2 Cognitive Load Theory

This theory focuses on managing the cognitive demands placed on students during the learning process. Based on its tenet that "human working memory is limited"[5], this theory justifies breaking down complex concepts into smaller, more manageable parts, in a way that reduces extraneous cognitive load and allows students to gradually build the complete skill.[6]

2.1.3 Problem-Based Learning

A teaching approach that emphasizes active learning through solving real-world problems. This might involve presenting students with practical design challenges and encouraging them to work together to develop solutions. This can help students to develop critical thinking skills, in addition to a deeper understanding of the concepts.[7]

2.1.4 Zone of Proximal Development

This concept, introduced by Lev Vygotsky, refers to the difference between what a learner can do without help and what they can achieve with guidance and support [8]. For a teaching tool, this might involve providing students with interactive suggestions, and hints, to help them bridge the gap between their current understanding and the problem that they're trying to solve as part of the learning process.

2.1.5 Active Learning

Active learning is an approach that emphasizes the importance of engaging students in the learning process, rather than passively transmitting the information. Active learning can involve group discussion, tracing algorithms, coding assignments and what-if scenarios [9]. Interactive tools that provide immediate feedback and allow students to explore the impact of changes to the system have been shown to assist with active learning [10].

2.2 Interactive Techniques in CSE

Using interaction can be powerful for enhancing students' understanding of complex concepts in computer science. Below is a review of some effective interaction techniques.

2.2.1 Immediate feedback

Interactive tools can provide immediate feedback on students' work, allowing them to quickly identify and correct errors, play with different scenarios, and develop a clearer understanding of the consequences of their actions [11].

2.2.2 Gamification

Integrating game-like elements into interactive tools can make learning more engaging and enjoyable for students [12]. This can help to motivate students and encourage them to spend more time actively working with the material [1].

2.2.3 Visualizations

Interactive visualization techniques can play a big role in facilitating students' understanding of complex concepts in computer science. It can help bridge the gap between practical coding assignments and the theory taught in class [3]. By providing interactive visual representations of the code's data structures, and algorithms, visualization tools can help students to build mental models and make connections between different theoretical concepts [4].

3. Compiler Theory

The subject of Compiler Theory explains how high-level source code gets translated, through a pipeline of processes, to low-level machine code that computers can understand and execute. There are various stages in this pipeline, namely: Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generator, Optimization and Code Generation [13].

This project will narrow its focus on creating a DSL Teaching Tool for the front end of the pipeline; the Lexical, Syntax and Semantic Analysis.

3.1 The Compilation Pipeline

3.1.1 Lexical Analysis

Lexical analysis is carried out by a lexical analyzer. The analyser takes in the stream of characters in a source program and then breaks it up into a stream of tokens. Based on the string pattern, usually defined by Regular Expressions, each token is assigned a type eg Identifier, Integer, Operator etc. [14]

Example

Character Input Stream: num = 10

Output of Lexical Analyzer: <id, "num">, <op, "=">, <int, "10">

, where each <...> represents an individual token.

3.1.2 Syntax Analysis

Also known as parsing, the Syntax Analyser ensures that the program conforms to a set of grammar rules by taking in a series of tokens and outputting a parse and abstract syntax tree as in Figure 1.[14]

Example

Input Stream: <int, "4">,<op,"+">,<int,"3">,<op,"+">, <int,"2">

Abstract Syntact Tree:

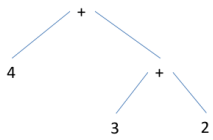


Figure 1

A program could pass the Lexical Analyser by having valid tokens, but could still not be a valid program. The Syntax Analyser, or Parser, ensures that the series of tokens in the program is valid[13].

3.1.3 Semantic Analysis

Before the Abstract Syntax Tree (AST) can be sent to the next phase, the Semantic Analyser ensures that there are no type or context errors using a Symbol Table. For example, ensuring that variables are defined before they are used, that expressions are type-consistent, or that an array reference is in bounds, amongst other checks.

Using the Symbol Table to store the meaning of Symbols/Identifiers, the abstract syntax tree is traversed to perform these checks. [13]

The AST is then sent to the intermediate code generator which converts it for the backend of the Compiler. That aspect of Compiler Theory, however, is not in the scope of this project. Note that some textbooks don't describe Semantic Analysis as a separate step, but rather include it in the previous step or consider it as a check that happens in parallel [14].

3.2 Domain Specific Languages

3.2.1 What are Domain Specific Languages?

Domain Specific Languages (DSLs) are programming languages that are tailored to a specific application domain or problem area, providing special constructs, abstractions, and syntax to express solutions concisely and efficiently [15].

In contrast, General Purpose Languages (GPLs) are designed to be versatile and applicable to a wide range of application domains and problem types. However, while GPLs offer this broad flexibility, they may lack the features and optimizations that can make DSLs more user-friendly and efficient for specific tasks [15].

Examples of Domain-Specific Languages include SQL for database management [16], HTML for web design [17] and MATLAB for mathematical computing[18]. These languages are tailored to their respective domains, offering specialized features and abstractions that make them well-suited to solving problems within those areas.

3.2.2 Tools for creating Domain Specific Languages

Creating DSLs can be a challenging task, but there are several tools and frameworks available to help developers in this process. These tools vary in terms of features, functionality, and target languages.

3.2.2.1 Traditional Tools:

Lex and Yacc: Lex and Yacc are well-known tools for compiler construction published in 1975 [19][20] in the C programming language. Lex creates a lexical analyzer, while Yacc creates a parser. Their popular equivalents are the open-sourced Flex and GNU Bison [21], respectively. Working together, these tools can help build compilers by generating code for the lexical analysis and syntax analysis phases, respectively.

ANTLR: ANTLR (Another Tool for Language Recognition) is a flexible parser generator that supports various languages, including Java, C, Python and Go [22]. ANTLR can generate both lexical analyzers and parsers which can automatically generate parse trees. The tool also has a useful graphical interface for visualizing parse trees and debugging grammars, shown in Figure 2 below.

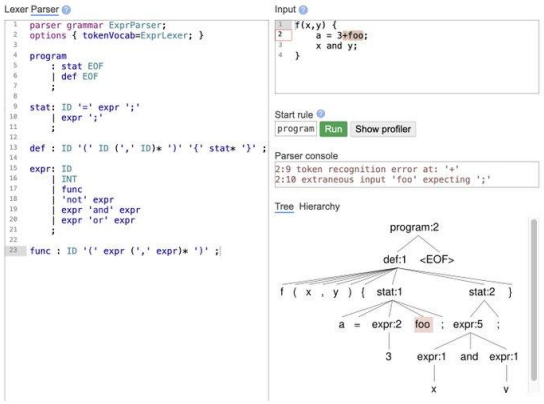


Figure 2

3.2.2.2 Modern Tools:

JetBrains MPS (Meta-Programming System): JetBrains MPS is an open-source language workbench that lets developers develop new Domain Specific or General Purpose Languages [23]. It's built on a projectional/structural editor which means that it does not limit developers to text editing but makes it possible to visualize and edit a representation of the Abstract Syntax Tree [24]. It also assists in developing non-textual domain-specific notation for your language, including math notations and diagrams. This arguably simplifies the DSL design, although its unconventionality might make it difficult for developers to get used to.

Xtext: Xtext is an open-source parser generator developed under the Eclipse Modelling Project, and can integrate very well with the Eclipse Modelling ecosystem [25]. A helpful feature of Xtext is that it can generate a serializer and smart editor along with the parser, without any additional code if customization is not necessary.

Ply for Python: Ply (Python Lex-Yacc) is a lexer and parser generator for Python. It is essentially a python implementation of Lex and Yacc that simplifies the code generation step [13]. Ply provides a familiar and simple interface for Python developers to create compilers and interpreters for custom languages. Ply also has helpful features, like error reporting, precedence rules, and support for LR parsings.

4. Teaching Compiler Theory

All approaches to teaching Compiler Theory strive to find the right balance between teaching theoretical concepts and assisting that understanding through practical assignments. Theoretical concepts are essential for understanding the underlying principles

of compiler design and implementation, and are largely the same across course designs [2].

The method with which the practical element is incorporated, however, is the subject of much debate. Practical assignments help solidify these concepts by providing students an opportunity to actively learn by applying the abstract theory on a problem, as well as help students see its practical relevance of the theory [1].

4.1 Different Approaches

4.1.1 Case-Based Problem

Kundra and Sureka argued for using a case-based teaching approach, presenting students with specific examples or case studies of real-world compiler design problems [26]. This approach encourages students to think critically about the challenges and trade-offs involved in designing and implementing compilers in the real world and arguably helps make learning easier and more interesting for students. An example presented in the paper was the use of lexical analysis to develop a program that can detect spam emails.

4.1.2 A Research Activity

Another approach is to assign aspects of the theory as a practical research activity. Moreno-Seco and Forcada drew on the Constructivist theory of education to formulate this method [27]. It essentially looks at students as novice researchers, that need to rapidly learn what has been done in a field in order to find the gaps and solve a problem. This also applies the problem-based educational approach by encouraging the learning of new concepts and theories only after the problem that motivated them is understood.

4.1.3 Mini versions of existing GPLs

A common approach to the practical part of Compiler Theory courses is to have students implement a compiler for a simplified or limited version of an existing General Purpose Language (GPL), such as C or Pascall. Terry [28] developed CLANG, a simplified subset of Pascal, and Rakic et al [29] developed a mini C language, in a similar fashion. These mini-language projects allow students to apply theoretical concepts to a mini version of a general language that still contains the most important characteristics of the programming language.

4.1.4 Using Domain Specific Languages

An approach that this project will draw on is the use of Domain Specific Languages. Henry[2] presents a strong argument for using domain-specific languages (DSLs) for the practical component of Compiler Theory Courses. This approach involves having students design and implement compilers for languages tailored to specific domains, such as scientific computing or embedded systems.

This can help students see the relevance of learning compiler theory [2] as they are learning how to produce working versions of domain-specific programming languages that can address gaps, as opposed to reinventing the wheel by developing smaller versions of existing languages.

It could also help students develop a deeper understanding of the unique challenges and requirements that could be present in different application domains and learn how to adapt compiler design principles to meet those needs. Shatalin et al. [23] pointed out that it is rare that developers have both the knowledge of Compiler Theory and domain knowledge to know when to use Domain Specific Languages. This could help address that gap.

4.2 Use of Interactive Tools

In order to address the challenge of helping students understand abstract concepts and complex algorithms in Compiler Theory, several attempts have been made to develop interactive simulators to visualise aspects of Compiler Theory. Stamenkvic et al. [2] surveyed several different simulators and evaluated them based on their characteristics and features, as well as the amount of Compiler Theory topics covered.

We will review the best interactive simulators described in that paper according to the mentioned criteria, including a solution developed by the same authors the following year.

4.2.1 LISA

LISA, developed by Mernik and Zumer [10], is an advanced interactive multipurpose graphical simulator built using Java for Desktops.

It covered the highest percentage of Compiler Theory Topics, at 46.2%. The closest was JFLAP at 38.8%, and most other simulators covered less than 30% of the topics. What is further notable for the purposes of our project, is that the only topics that LISA doesn't cover at least partially, are the topics not in the scope of this paper.

LISA illustrates Lexical Analysis animating the deterministic finite-state automata, Syntax Analysis with a Syntax Tree and the check for Semantic rules with a Semantic tres that highlights dependencies. Examples of his are shown in Figure 3.

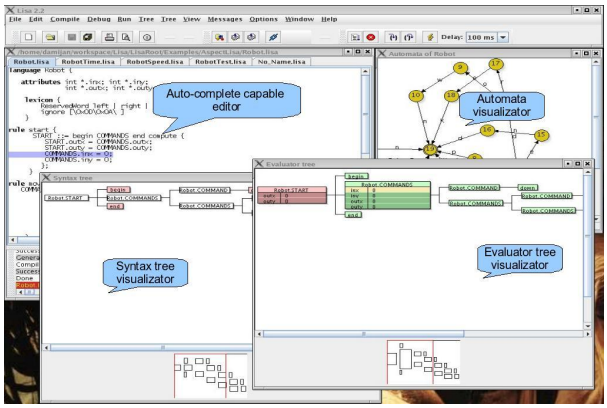


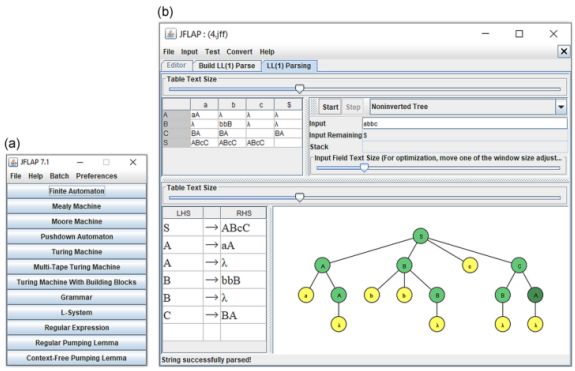
Figure 3

4.2.2 JFLAP

JFLAP is also an advanced interactive multipurpose graphical simulator built using Java for Desktops [2].

It covers 38% of the topics, with a focus on defining automata and grammar [2]. Notable distinct features include a graphic editor for drawing many types of automata and, unlike LISA, JFLAP has the ability to convert nondeterministic finite automata into deterministic automata and transform automata into appropriate regular grammar [30].

Figure 4 shows some of the options offered by JFLAP



(a) Options offered by Java Formal Languages and Automata Package (JFLAP) at startup and (b) the parsing process

Figure 4 [57]

4.2.3 Stamenković and Jovanović

Based on their survey and evaluation above, Stamenkovic and Jovanovic published a paper on two alternative interactive graphical simulation tools to visualise the theory underpinning lexical and syntax analysis [3]. One is a web-based tool, specifically for lexical analysis, and the other is coded using Java, for desktops.

The solution does not seem to require coding. Rather the visualisation provides textboxes or objects, with a lot of guidance, to input information. It then outputs the information, either using easy-to-understand text or by visualizing the relevant data structures.

The programs allow users to:

- Easily input regular definitions and see the implications of those definitions on an input sentence/string.
- Define automata with a drawing or an automatic transition table, and see the implications on an input sentence/string.
- Transform a regular expression into a DFA or NFA, simulated in steps or using Thompson's algorithm.
- Simulate the construction of the syntax tree for a particular regular expression.

5. Discussion of Findings

The need for an interactive Domain Specific Teaching Tool can be justified both by the arguments that the concepts are perceived to be complex and irrelevant by students, and by the amount of research that has been done, and is still being done, in Universities all over the world.

There have been several implementations of an interactive graphical compiler simulator. However, looking at similar systems, we can see some potential gaps.

None of the multipurpose implementations, that cover a substantial amount of theory, are built for multiple platforms. This makes them difficult to use across all the platforms that students use. A web app or an app for multiple platforms using tools like Flutter, with as many features as LISA, could be a potential solution.

Secondly, while the implementations of the simulators illustrate the underlying data structures quite clearly, there is rarely any actual coding required. This level of abstraction might make the gap between the interactive simulators and using a real-world parser too great. A middle ground between compilers like ANTLR, which allow users to code and visualise some data structures, and the educational and simulation features of the current interactive teaching tool might be a solution to this.

The implementations apply two of the reviewed educational interactive techniques very well, Immediate Feedback and Visualizations. However, none of the implementations applied Gamification. This might be a helpful inclusion, in the form of points for completing quiz problems in the system or for logging in daily.

In the Computer Science Education Theory reviewed, collaboration, group work and discussions seemed to be a recurring theme. The popular coding education app, SoloLearn, achieves this by allowing users to comment and upvote comments on every short lesson and assessment [12]. This allows for common misunderstandings and mistakes to be addressed at every step of the learning process and allows users to interact with other students. Comment upvotes can also be integrated into the gamified aspect of the program to encourage students to provide helpful comments that other users can benefit from.

6. Project Evaluation Criteria

The method of evaluation could be more robust to that of Stamenković et al. above since they were constrained by the task of having to evaluate several tools and not having as many groups of students. They were, hence, forced to evaluate the tools based on a reasonable assumption of what desirable factors would be [2]. This project could include that as part of the evaluation but, since we will only be evaluating our tool, can also attempt to look for a causal relationship between the teaching tool and the learning experience of students.

To do this, we could assess the counterfactual impact that it has on the learning experience. This would require a control group, as they do in Randomized Control Trials [31]. A good control group would have the same lecturer, theory, and assessment, but would not have access to the DSL Tool. Data for the two groups can be collected in the following ways:

6.1 Pre- and post-test assessments

Conduct assessments before and after using the DSL teaching tool to measure students' knowledge and understanding of Compiler Theory concepts, comparing the results to determine the impact of the tool on learning outcomes.

6.2 Student feedback and surveys

Collect feedback from students through surveys or focus groups to gather insights into their experiences and perceptions of the learning experience. This will also identify areas for improvement and potential enhancements of the DSL Tool.

6.3 Observations and classroom assessments

Observe students' interactions during class sessions, assessing their engagement, problem-solving abilities, and collaboration skills, as well as their ability to apply Compiler Theory concepts in practice.

6.4 Longitudinal studies

Track students' progress and performance over time, analyzing the long-term effects of using the DSL teaching tool on their understanding and application of Compiler Theory concepts [32]. This could be done by observing differences in students' understanding of a second compiler course if the University offers one.

7. Conclusion

The literature review revealed and expanded upon the importance of incorporating interactive techniques into education tools. Reviewing the compiler theory covered in courses and past approaches to teaching it, the teaching of Compiler Theory seems to be no exception to this recommendation.

The review also highlighted the benefits of past uses of Domain Specific Languages and interactive tools to teach compiler Theory. Although there have been many past implementations, the review also revealed some gaps that the project could be able to fill.

Limitations to this review include potentially missing novel well-implemented approaches to creating an interactive teaching tool for compiler theory, or educational theory that could be incorporated.

8. REFERENCES

[1] Henry, T.R. (2005) "Teaching compiler construction using a domain specific language," *Proceedings of the 36th SIGCSE technical symposium on Computer science education* [Preprint].

[2] Stamenković, S., Jovanović, N. and Chakraborty, P. (2020) "Evaluation of simulation systems suitable for teaching compiler construction courses," Computer Applications in Engineering Education, 28(3), pp. 606–625. 3[1] Stamenkovia, S. and Jovanovia, N. (2021) "Improving participation and learning of compiler theory using educational simulators," *2021 25th International Conference on Information Technology (IT)* [Preprint].

[4] Ben-Ari, M. (1998) "Constructivism in computer science education," ACM SIGCSE Bulletin, 30(1), pp. 257–261.

[5] Wilson, B.G. and Cole, P. (1996) "Cognitive teaching models," In D. H. Jonassen (Ed.), Handbook of research in instructional technology , pp. 601–621.

[6] Shaffer, D., Doubé, W. and Tuovinen, J. (2003) "Applying Cognitive load theory to computer science education," Annual Workshop of the Psychology of Programming Interest Group, pp. 333–346. 7[30] Hmelo-Silver, C.E. (2004) "Problem-based learning: What and how do students learn?," *Educational Psychology Review*, 16(3), pp. 235–266. 8[33] Anderson, N. and Gegg-Harrison, T. (2013) "Learning computer science in the 'comfort zone of proximal development,'" *Proceeding of the 44th ACM technical symposium on Computer science education* [Preprint]. Available at: https://doi.org/10.1145/2445196.2445344.

[9] McConnell, J.J. (1996) "Active learning and its use in computer science," *Proceedings of the 1st conference on Integrating technology into computer science education - ITiCSE '96* [Preprint]. 10[37] Mernik, M. and Zumer, V. (2003) "An educational tool for teaching compiler construction," *IEEE Transactions on Education*, 46(1), pp. 61–68.

[11] Corbett, A.T. and Anderson, J.R., 2001, March. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In Proceedings of the SIGCHI conference on Human factors in computing systems (pp. 245-252)

[12] Putra, G.N.Y.A., Junus, K. and Santoso, H.B., 2022, October. Gamification-Based Online Collaborative Learning Feature Design on SoloLearn Application with Mechanics-Dynamics-Aesthetics Framework and User-Centered Design Method. In 2022 International Conference on Advanced Computer Science and Information Systems (ICACSIS) (pp. 65-74). IEEE.

[13] Amiguet, M., 2010. Teaching compilers with python.

[14]Mogensen, T.Æ., 2009. Basics of compiler design. Torben Ægidius Mogensen.

[15] Mernik, M., Heering, J. and Sloane, A.M., 2005. When and how to develop domain-specific languages. ACM computing surveys (CSUR), 37(4), pp.316-344.

[16]Melton, J. and Simon, A.R., 1993. Understanding the new SQL: a complete guide. Morgan Kaufmann.

[17] Raggett, D., Le Hors, A. and Jacobs, I., 1999. HTML 4.01 Specification. W3C recommendation, 24.

[18] Matlab, S., 2012. Matlab. The MathWorks, Natick, MA.

19[45]Lesk, M.E. and Schmidt, E. (1990) "Lex − A Lexical Analyzer Generator."

[20] Johnson, S.C., 1975. Yacc: Yet another compiler-compiler(Vol. 32). Murray Hill, NJ: Bell Laboratories.

[21] Levine, J., 2009. Flex & Bison: Text Processing Tools. " O'Reilly Media, Inc.".

[22]Parr, T., 2013. The definitive ANTLR 4 reference. The Definitive ANTLR 4 Reference, pp.1-326.

[23] Pech, V., Shatalin, A. and Voelter, M., 2013, September. JetBrains MPS as a tool for extending Java. In Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (pp. 165-168)

[24] Campagne, F., 2014. The MPS language workbench: volume I (Vol. 1). Fabien Campagne.

[25] Eysholdt, M. and Behrens, H., 2010, October. Xtext: implement your language faster than the quick and dirty way. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (pp. 307-309).

[26] Kundra, D. and Sureka, A. (2016) "An experience report on teaching compiler design concepts using case-based and project-based learning approaches," *2016 IEEE Eighth International Conference on Technology for Education (T4E)* [Preprint].

[27] Moreno‐Seco, F. and Forcada, M.L. (1996) "Learning compiler design as a research activity," Computer Science Education, 7(1), pp. 73–98.

[28] Anderson, N. and Gegg-Harrison, T. (2013) "Learning computer science in the 'comfort zone of proximal development,'" *Proceeding of the 44th ACM technical symposium on Computer science education* [Preprint].

[29] Rakic, Z.S., Rakic, P. and Petric, T. (2014) "miniC Project for Teaching Compilers Course," *ICIST 2014*, 2.

[30] Shaffer, D., Doubé, W. and Tuovinen, J. (2003) "Applying Cognitive load theory to computer science education," *Annual Workshop of the Psychology of Programming Interest Group*, pp. 333–346.

[31] Stanley, K., 2007. Design of randomized controlled trials. Circulation, 115(9), pp.1164-1169.

[32] Caruana, E.J., Roman, M., Hernández-Sánchez, J. and Solli, P., 2015. Longitudinal studies. Journal of thoracic disease, 7(11), p.E537