DSLDoodle: An Interactive Teaching Tool for Designing and Visualising DSLs

Ahmed Ghoor
University of Cape Town
South Africa
ghrahm004@myuct.ac.za

Abstract

Compiler Theory is an essential and foundational topic in Computer Science Education. It teaches students how high-level source code gets translated to low-level machine code that computers can understand and execute. Despite its significance, research shows that students often struggle to see the relevance of the subject and have difficulty understanding the concepts. To address these challenges, this work presents DSLDoodle, a simple interactive tool that allows students to design their own Domain-Specific Languages (DSLs) while simultaneously visualising the underlying data structures. The tool focuses on Domain-Specific Languages, which are becoming increasingly prevalent in various industry sectors, thereby highlighting the real-world relevance of Compiler Theory. Furthermore, the visualisation feature of DSLDoodle assists in demystifying abstract concepts, allowing students to grasp the intricacies of how compilers operate. Preliminary evaluations suggest that the integration of tools like DSLDoodle into the curriculum can boost student engagement and foster a deeper understanding of Compiler Theory. This work underscores the potential of visual interactive tools in Computer Science Education in general.

CCS Concepts: • **Programming Languages**; • **Compilers**; • **Computer Science Education**; • **Visualisation**;

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

1 Introduction and Motivation

1.1 What is the Project

The software engineering project aims to develop and evaluate an interactive visual tool for creating Domain-Specific Languages as a means to enhance the learning of Compiler Theory.

1.2 Importance of the Project

Compiler Theory is an essential 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. Despite this, the teaching of Compiler Theory is accompanied by certain challenges that DSLDoodle attempts to address:

Conceptual Complexity The translation process involving low-level concepts can be intricate and abstract. Students often struggle to grasp these concepts [45]. Developing simplified practical problems to engage students and facilitate their understanding of such abstract concepts is a pressing requirement, as detailed later in this paper.

Lack of Interest Students sometimes exhibit disinterest in Compiler Theory due to its perceived lack of immediate practical relevance. The subject's application in real-world scenarios might not be evident to students, leading to reduced motivation [19].

DSLDoodle aims to address these challenges. An interactive visual tool can potentially facilitate a more engaging learning experience that enhances students' understanding of Compiler Theory concepts [45]. In addition, using Domain Specific Languages can potentially highlight examples of the practical relevance of Compiler Theory in industry, thereby increasing students' interest in the subject [19].

2 Theoretical Background and Related Work

2.1 Computer Science pedagogy

While there is extensive literature in Computer Science Education (CSE) relevant to teaching Compiler Theory, this paper narrows its focus to concepts specifically relevant to the topic. Some of these concepts will be referenced later in the paper, which will explicitly highlight their relevance.

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.[5]

2.1.2 Constructivism. Constructivism points out that students don't just absorb information. They actively build on pre-existing knowledge to form their own understanding by constructing mental models.[5]

UCT, September 11–06, 2023, Cape Town, South Africa

2.1.3 Cognitive Load Theory.

Given our limited working memory [55], this theory suggests breaking down complex concepts into smaller parts. This reduces extraneous cognitive load and allows students to gradually build the complete skill. [42]

2.1.4 Problem-Based Learning.

This active learning approach pushes students to collaboratively solve real-world problems, helping them think critically and get a deep understanding of the concepts.[21]

2.1.5 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 [4].

2.1.6 Active Learning.

Active learning emphasizes the importance of engaging students, rather than passively transmitting information. It can involve group discussion, tracing algorithms, coding assignments and what-if scenarios [29]. Interactive tools, providing immediate feedback and allowing students to explore the impact of changes, have been shown to assist with active learning [32]. The next subsection explores this more.

2.1.7 Interactive Tools.

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

- **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 [8].
- **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 [45]. 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 [5].

2.2 Compiler Theory

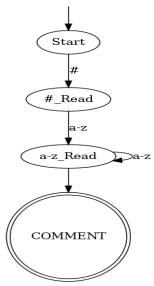
The subject of Compiler Theory explains how high-level source code gets translated, through a pipeline of processes, to low-level machine code that computers can understand and execute. There are various stages in this pipeline, namely: Lexical Analysis, Syntax Analysis, Semantic Analysis, Code Optimization and Code Generation [3]. 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.

2.2.1 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 lexical rules, usually defined by Regular Expressions, each token is assigned a type, e.g. Identifier, Integer, Operator etc. [33] For example:

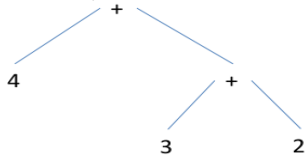
- Character Input Stream: num = 10
- Output of Lexical Analyzer: <id, "num">, <op, "=">, <int, "10"> , where each <...> represents an individual token.

The lexical rules can be represented using Automaton diagrams. For example, if the rule for a coding comment was defined as r'[#][a-z]+', the resultant automata would be:



2.2.2 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 an abstract syntax tree (AST). [33] Example Input Stream:
<int, "4">,<op,"+">,<int,"3">,<op,"+">, <int,"2">
Abstract Syntax Tree:



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 [3].

2.2.3 Semantic Analysis.

Before the Abstract Syntax Tree 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. [3]

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[33].

2.2.4 DSLs vs GPLs. Domain Specific Languages (DSLs) are tailored for specific problem areas or application domains, providing special constructs and syntax to efficiently solve domain-specific problems, like SQL for databases[30], HTML for web design[39] or MATLAB for mathematical computing[49]. On the other hand, General Purpose Languages (GPLs) are broader, and versatile for numerous tasks, but may not have the niche optimizations that make DSLs efficient for specific tasks.[31]

2.3 Approaches to teaching Compiler Theory

Most approaches to teaching Compiler Theory aim for a balance between teaching theoretical concepts and practical assignments. Theoretical concepts are essential for understanding the underlying principles and are largely the same across most courses [46]. The manner of integrating practical work, however, is debated. Such assignments help solidify the theory by providing students an opportunity to actively learn by applying the abstract theory to a problem, as well as help students see the practical relevance of the theory[19].

2.3.1 Case-Based Problems. Kundra and Sureka suggest case-based teaching using real-world compiler design issues[24]. An example presented is using lexical analysis for spam email detection. This makes students weigh real-world trade-offs critically, arguably making 'learning easier and more interesting.

2.3.2 A Research Activity. In this method, students act as beginner researchers, diving deep into existing knowledge to find and fill gaps. Rooted in Constructivist theory and problem-based learning, this approach pushes students to comprehend theories post recognizing the related problem[34].

2.3.3 Mini versions of existing GPLs. A common approach is to have students develop a compiler for a simplified version of an existing General Purpose Language (GPL).

Terry [48] developed CLANG, a simplified subset of Pascal, and Rakic et al [40] developed "mini-C" language, in a similar fashion. These projects allow students to apply theoretical concepts to a mini version of a general language that still contains most of it's important characteristics.

2.3.4 Using Domain Specific Languages. An approach that this project drew on is the use of Domain Specific Languages (DSLs). Henry[19] presents a strong argument for using DSLs for the practical component of Compiler Theory courses. Rather than recreating existing languages, students develop compilers that can address domain-specific software gaps, helping them appreciate the relevance and value of the theory. It also enhances their understanding of domain-specific challenges and could bridge the gap identified by Shatalin et al., where few developers excel in both Compiler Theory and knowing when to employ DSLs[37].

2.4 Tools for creating DSLs

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.

2.4.1 Traditional Tools.

- **Lex and Yacc:** These classic compiler tools, Lex for lexical analysis and Yacc for parsing, have been available since 1975[25][23]. Open-source versions, Flex and GNU Bison, are common alternatives. Together, they assist in building compilers.
- **ANTLR:** ANTLR (Another Tool for Language Recognition) is a flexible parser generator that supports various languages, including Java, C, Python and Go [36]. 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.

2.4.2 Modern Tools.

- **JetBrains MPS (Meta-Programming System):** JetBrains MPS lets developers develop compilers. [37]. It's built on a projectional editor which allows it to edit a representation of the Abstract Syntax Tree [7]. It also assists in developing non-textual notation for languages.
- **Xtext:** Part of the Eclipse Modelling Project and ecosystem [13], Xtext can generate a serializer and smart editor along with the parser, no additional code 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 [3]. 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.

2.5 Interactive Compiler Visualisations

2.5.1 In Education Simulators. Many isual simulators attempt to help clarify complex compiler concepts. Stamenkvic and others reviewed various simulators, ranking them by features and topics covered[46].

This paper reviews the best interactive simulators surveyed, according to the mentioned criteria, including a solution developed by Stamenkvic et al. the following year.

- **LISA:** LISA, developed by Mernik and Zumer [10][32], is an interactive graphical simulator built using Java

for Desktops. It covered the highest percentage of Compiler topics, at 46.2%, including all topics in the scope of this paper. The closest was JFLAP at 38.8%, and most other simulators covered less than 30% of the topics. 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 tree that highlights dependencies.

- **JFLAP:** JFLAP is also a graphical simulator built using Java for Desktops [38]. It covers 38% of the topics, with a focus on defining automata and grammar [46]. 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 [38].
- **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 [45]. 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.

2.5.2 In Compiler Creators. As shown in section 2.5, there are compiler construction tools that are not designed for educational purposes, but still provide interactive visualisations to aid the language-building process. ANTLR's graphical interface is useful for visualizing parse trees and debugging grammars and Jetbrains' MPS's projectional editor makes it possible to visualise and edit a representation of the Abstract Syntax Tree [7].

2.6 Review Summary

The theoretical review revealed the educational benefit 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 applicability of such a tool is supported by both the research showing the perceived complexity and irrelevance of compiler concepts by students [10][45], and research that has been done on building interactive tools and teaching with Domain-Specific Language assignments to address these respective challenges [19][46].

There are many past implementations of an interactive graphical compiler simulator [46]. That said, 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.

Secondly, while the simulators illustrate the underlying data structures quite clearly, there is often not much coding required. This is understandable, as unconstrained code input can make it difficult to consistently extract the relevant information to produce visualisations. This level of abstraction, however, might make the gap between the interactive simulators and using a real-world parser too great. A solution might be a middle ground between Ply, which draws on traditional compiler principles in a familiar language and the simulation features of the current interactive teaching tools might be a solution to this.

3 Methods

This paper presents a potential solution to some of the challenges highlighted in the theoretical review, namely an interactive teaching tool for designing and visualising domain-specific languages.

3.1 Project Aims

Based on the above review, this project has the following aims:

1. Build and evaluate if a cross-platform compiler creation app, with increased visualisations of underlying data structures, increases students' understanding of complex compiler theory concepts.
2. Evaluate if the implementation of the cross-platform app with interactive visualisations and domain-specific language crafting capabilities in a familiar language like Python can increase student engagement and interest in compiler theory.

3.2 Requirements

The requirements of the system are based on the theoretical review and the aims of this project. They include:

1. **Cross-Platform Compatibility:** The application should be accessible and functional on all major operating systems.
2. **Visual Representation of Data Structures:** The system should provide clear visualizations of underlying compiler data structures.
3. **Domain-Specific Language Crafting:** The tool should allow users to design and test domain-specific languages within the context of a familiar language.
4. **Interactive Interface:** The user interface should be intuitive and interactive, supporting real-time feedback.
5. **Reliability and Robustness:** The system should be able to handle a wide variety of inputs and use cases without crashing or producing incorrect outputs.

3.3 System Design

The system design process is the blueprint for the implementation, emphasising user experience and structural integrity. It balances planning with adaptability and ensures the tool's reliability and functionality.

3.3.1 User-interface Design. To design the User-Interface, Ben Shneiderman's Golden Rules of Design were incorporated wherever possible [43]. His rules include striving for consistency, catering to universal usability, offering informative feedback, preventing errors, permitting easy reversal of actions and reducing short-term memory load.

To maintain consistency, the layout, graphics and fonts were designed with the aim of mimicking a user-friendly Android application, which dominates the mobile application and operating system market share, especially in emerging markets [1][16]. This also helps ensure universal usability across technical backgrounds.

The app is also designed to be interactive, offering feedback whenever the user takes an action. When any of the buttons are clicked, a visualization is created or a menu drops down, when any other text boxes are clicked to input text, they light up and when buttons are clicked, the GUI informs the user if the action has been a success. This facilitates letting users feel in control of the app.

Measures to avoid errors include labelling and adding tooltips to text boxes, and ordering GUI elements in the sequence of the action order.

If a language is badly specified and this is only discovered later in the process, the program allows users to go back and rebuild the language and visualisations, thus allowing reversal of actions.

Lastly, in order to reduce short-term memory load, the project draws inspiration from ANTLR's web interface [36] which compacts several sections of the program into one screen (the user-defined DSL rules, generated visualisations and the user's DSL code) so that the user does not have to remember information from one display to the other.

3.3.2 Software Design Pattern. The application followed the Model-View-Controller (MVC) design pattern [9]. In this framework, the "Model" encompasses the core data and logic, serving as the application's backbone. The "View" is the user interface, presenting the model's current state in a user-friendly manner. Meanwhile, the "Controller" bridges the Model and View, processing user input and updating the view accordingly.

3.3.3 Engineering Methodology. The engineering phase largely followed the waterfall methodology for most of the project with the incorporation of agile components towards the end[11]. West et al. [54] coined the term "Water-Scrum-Fall" and hypothesized that hybrid development methods would become the standard, which certainly held true for this project. The limitation of using a fully Agile approach from the beginning, which would have been ideal, was that the project had to wait for ethical clearance before it could have access to students to provide feedback on the tool. This methodology involved gathering requirements and specifications to develop the application through a literature review and engagements with the project proposer and supervisor. While building the application, Alpha system tests were constantly run to test the robustness of the application. Once ethical clearance was received, Beta testing could commence. This is where Agile software development methodologies were incorporated. 10 users, split into 4 groups, individually tested and provided feedback on the application. After all users in a group had completed testing of the application, the feedback was incorporated into the app during four separate sprints [41].

3.3.4 System Testing Methodology. System Testing consists of Alpha and Beta testing. Alpha testing in this project consisted of a series of white box tests designed by the researcher. Beta testing entailed giving the application to users outside of the development process to play with.[22]

To do Alpha testing, the project separately tested functions and modules in the code to ensure that each individual segment of the code was working as it should [35]. The Unit tests can be seen in the *dsldoodle_test.py* file. Each test has its own folder containing a text file with the test input and PNG files of the correct output.

As mentioned above, Beta testing involved splitting 10 users into 4 groups and having each user test and provide feedback individually. At each iteration, the users were completely different. This allowed users to test the improvements without having the data skewed by the users that suggested the improvement.

Details of the User Testing methodology are explained in the next section.

3.4 Evaluation Methodology

Given the time frame in which this project needs to be completed, it was not feasible to design the experiment exactly as it was laid out in the initial Literature Review [17], by doing a randomised control trial on the experience and effect on course marks through using this tool in a Compiler Theory course. The project will also not be able to do a longitudinal study by observing the effect on a second compiler course. The project will evaluate the tool by providing it to students who have already studied a university-level compilers course to complete a few tasks with and without the tool, before getting them to answer questions about it through surveys and interviews.

3.4.1 User Testing Methodology. User testing informed the Beta System Testing and Evaluation process. To do user testing, the study aimed to have 10 participants from the University of Cape Town who completed the Computer Science Honours course on Compilers in the last two years. The

student participants were Honours students from the University of Cape Town, who have previously taken a compiler course.

3.4.2 User Testing Procedure.

1. **Building a DSL without the tool**
   Before interacting with the tool, students were given working language rules written in Ply and tasked with printing the lexical tokens, abstract syntax tree, and then tweaking the rules to observe changes in said tokens and tree. During this phase, students were given a verbal recap of the relevant Compiler Theory, were allowed to access Ply docs and the internet, and could ask any questions about Ply or Compiler Theory while they were completing the task. The objective of this step was to refresh students' memory of Compiler Theory and what coding in Ply entails to help them set a mental benchmark to use when evaluating the tool.
2. **Cognitive Walk-Through** [26]
   Students were then given the tool and prompted to explore the application without any external guidance. Throughout this exploration, students were encouraged to verbally share their thoughts, shedding light on their firsthand interactions and impressions. The objective of this process was to gather insights into how intuitive and user-friendly the tool is.
3. **Self-assessment of Tool Experience (Quantitative)**
   Post familiarisation, but before external guidance was given, students were presented with a set of statements on an anonymous form to gauge to what extent their experience aligned with the statements, measured on a Likert scale with 5 levels from Strongly Disagree (1) to Strongly Agree (5) [2]. The statements were:
   - "I feel confident in my ability to utilize this tool."
   - "The design and functionality of the tool are clear and intuitive."
4. **Building a DSL with the Tool**
   Students were then given instructions on how to use the tool and the same Ply base code provided in Step 1. They were then tasked with tweaking the language rules while observing changes in the lexical and syntax visualizations, and tweaking the DSL code while observing changes in all three visualizations.
5. **Post-task survey (Qualitative)**
   Upon completion of the tasks, students filled in an anonymous survey to offer insight based on their experiences. The survey contained questions related to both the implementation of the tool, as well as the conceptual approach behind the implementation. This is so that the project could assess both the usefulness of the particular implementation, as well as the concepts behind the implementation that could provide insights for similar tools. The latter evaluation is based on the Technology Acceptance Model (TAM), an extensively used theoretical framework designed to evaluate individuals' acceptance of an information system [27]. The Questionnaire can be found in Appendix B.

3.4.3 Evaluation Method Reliability.

This section will discuss the reliability of the evaluation method chosen.

Iterative development and evaluation combined: It may seem unusual to continue to develop the system while it is in the process of being evaluated, so it is worth discussing if this could affect the reliability of the results.

The reason for combining the two was that the ethics approval for accessing students for user testing arrived very close to the submission deadline. Thus the choice was between doing no iterative development from user feedback and doing it while evaluating the tool. The former was preferred for the benefits it offers in developing a more robust and user-friendly system [11].

With regards to reliability, it is important to first note that the improvements were not be in any fundamental conceptual approaches to the implementation. As illustrated in a following section, it affected aspects such as layout and tooltips to make the tool more intuitive, as well as system reliability. Additionally, it could be argued that any effect of continuous development would likely not affect the reliability of the results in a manner that would be favourable to the tool or the concept being tested. Since, with this method, a portion of the results would more likely be based on negative feedback that the final tool no longer deserves, as the feedback would often be taken into consideration to improve the tool in the next development sprint. It could thus be argued that this decision would make the results a conservative estimate of how effective it is in achieving its aims, which would potentially increase the reliability of any positive conclusions.

Neutrality of Participants: The participants were all honours students at the University of Cape Town and hence would know, or know of, the researcher who is also a fellow Honours student. This could cause the participants to be biased in their evaluations. To mitigate this, all surveys submitted were anonymous and participants were reminded that feedback given would be used to iteratively develop the system, thus any constructive feedback would be useful to the project and should not be avoided.

Level of Participants: The students testing the system have already completed a Compiler Theory course, even though this tool is designed to be used while the course content is still being learnt. This, however, would likely not affect the reliability of results in a major way. The tool is not designed to replace any compiler theory content but to be a supplement to it. Hence, any content that could potentially be taken for granted while using the tool would not be "missed" by the tool. The tool is an addition to the course. It could

also cause students to **underestimate** how effective the tool is in helping them understand Compiler Theory concepts as they have already learnt many of them. This is better than an overestimate as, as mentioned above, a conservative estimate of how effective the tool is in achieving its aims would be favourable from a reliability perspective.

3.4.4 Ethical Issues. In this study involving human subjects, addressing ethical concerns is very important [20]. Before engaging in the study, participants carefully reviewed and signed a consent form. Ethical approval was obtained from both the computer science department and the university. To ensure confidentiality, no identifiable information about participants was disclosed in the published results. Data collected is stored on a password-protected computer, with access limited to the researcher, supervisor and second reader of this project. All software employed in this study will have been released under open-source licenses, and any software created during the research process will be made available as open-source under the GNU General Public License.

4 Development and Implementation

4.1 System Implementation

4.1.1 Language and Library Choices. The language and library chosen for users to develop their DSLs are Python and Ply respectively. Since Ply is a Python implementation of Lex and Yacc [3], it allows students to gain exposure to traditional compiler-building principles in a language that they are already comfortable with. This choice allows students to immediately focus on learning to build compilers without first crossing a language barrier. This is based Lev Vygotsky's theory of the Zone of Proximal Development that was explained earlier in the paper [4].

Following on from this, it made sense to build the entire back-end in Python. Python has a large online community and, potentially due to Python's recent rise in popularity due to its use in Statistics and Machine Learning, there are also several libraries for building graphs and visualisations in the language [18][44]Ȧfter some trial and error with open-source libraries, this project decided to use the libraries Graphviz[12] and Automata-lib[6]. Graphviz is an open-source graph visualisation software that is quite flexible. Automata-lib is built on top of Graphviz and assists with visualizing Automaton. These libraries, although not without their limitations, proved quite useful. One of the more tricky aspects of this project on the engineering front was automatically translating the various ways that the user could define their DSLs into the specific format required by these libraries.

The Python library chosen for the User-Interface (UI) was Flet [14]. The project initially explored using the Dart language, with its accompanying Flutter framework, in the front end [47]. This would call a Python script to process the user input and build visualisations which could then be sent back to the User-Interface. One benefit of Flutter is that allows a single code base to be compiled for multiple platforms which would go a long way in ensuring the portability of the program, a key consideration of the project. The default UI is also relatively easy to use and looks like a modern Android app as it is created and maintained by Google. From a user-experience perspective, this helps to achieve Ben Shneiderman's golden rule of consistency [43]. The reason for the deviation to Flet is that Flet is a new Python library that allows developers to build Flutter-like apps in Python for multiple platforms without the need for any Dart code. Therefore, since the rest of the application was being built in Python, it made sense to keep it all in Python if the key benefits of Flutter could be retained.

One limitation of the Flet library built on the Flutter framework [15] that was observed at the time of development was the inability to have the "tab" keyboard action create an indentation in the text field when writing Python code. The action, instead, switches to the next GUI element. However, Python PEP8 guidelines suggest using spaces instead of tabs [52] and an educational tool could justifiably encourage good coding practices from students. It thus did not seem worth the project's time or sacrifice of other Flet library features to rewrite much of the code base in another language to include this feature.

4.1.2 Programming Techniques. To facilitate understanding and reusability, the code is modularised [50]. The code is organised into several functions, each with a specific purpose. For instance, "get_patterns_from_module()" returns a dictionary of tokens with their respective regular expressions which is then sent to generate_nfa(). For increased modularisation, the Lexers and Parsers are also built using separate modules.

The project also tried to keep the code clean and readable by using meaningful function and variable names like traverse_ast_for_semantics and generate_nfa, which clearly indicate their purpose, and generally follow PEP 8 guidelines [52]. In addition to this, the code is extensively documented with comments and function descriptions.

4.1.3 Maintainability and Portability. As demonstrated above, the project makes a concerted effort to ensure that the code is readable, well-documented, modular, and follows the common PEP8 standards. All of which assist with maintainability [28]. Additionally, dependencies between components are reduced by building some visualisations, such as the automaton diagram, by calling a symbol dictionary that is separate from the main program's dictionary.

Portability is ensured by building the app using Flet, a Python library that allows a single codebase to be compiled for multiple platforms [28]. Furthermore, most of the libraries chosen are relatively commonly chosen options for their

tasks. This mitigates issues when installing external dependencies.

4.2 Educational Visualizations

4.2.1 Lexical Analysis. The lexical analysis breaks down code into a sequence of tokens and is visualised with an NFA automaton diagram and a lexical analysis table. Based on the regular expression token definitions, the automaton diagram produced shows all ways of getting to a final token definition state from a start state. Furthermore, contrary to most educational automaton diagrams, the states are named with explanatory names to better facilitate an understanding of the underlying process.

The diagram is built using the automata-lib[6] library, a python library built on Graphviz[12]. Automata-lib requires the NFA to be defined with a particular syntax. This syntax was produced by first extracting the token definitions from the Python module and building a dictionary. The dictionary is then sent to the generate_nfa function which builds a series of dictionaries to capture and process all the individual transitions and input them into a Python code string to be executed.

The lexical table is produced when the DSL code is compiled. Based on the regular expression token definitions, the code is analysed and split up to produce a table of values from the code with their respective token type. Ply provides the function to split the code up into token values with their respective token types [3]. This was transferred to a Flet data table which could be visually displayed.

4.2.2 Syntax Analysis. The Syntax Analysis uses the lexical and the syntax rules, along with the sample DSL code, to visualise an abstract syntax tree (AST). Ply outputs an AST as a tuple of nested tuples [3]. This is then converted to the syntax required by Graphviz which is then executed to build a tree and output it to a PNG which can be displayed [12].

4.2.3 Semantic Analysis. The Semantic Analysis traverses the AST to perform semantic checks. Ply does not provide dedicated functionality for this. To illustrate what semantic analysis can do, the code provides users with the ability to check if a variable has been used before being defined and if it is being defined more than once non-mutability is a criterion).

The algorithm takes in the Ply-defined Rules, in addition to the name of the tokens used for variable assignment and symbols provided by the User in separate text fields. It then builds and queries a symbol table as it traverses the tree to do the semantic checks. Errors and the variable names causing the error are appended to an array. Both the symbol table and the errors are visualised with Flet Data tables.

5 Results and Discussion

5.1 Results of System Tests

The primary focus of the system testing was to ensure that DSLDoodle was functionally sound, providing accurate visualizations of underlying data structures, and could support the crafting of domain-specific languages in Python. System testing was divided into Alpha and Beta phases [22].

5.1.1 Alpha Testing. The core functionalities and aspects of the DSLDoodle tool were tested during this phase:

- **Lexer and Parser Creation:** All the functions associated with creating lexers and parsers were tested. The system was able to correctly generate tokenizers for the provided test cases and could parse sample source codes without any errors.
- **Visualization Mechanism:** The functionality that transformed the underlying data structures into visual graphics was tested with varied DSL definitions. The generated AST, automaton diagrams and tables matched the expected outputs, as confirmed by comparison with the stored PNG and text files.
- **Performance:** The application was able to handle larger DSL definitions and source codes without significant delays or crashes.

The dsldoodle_test.py file contains detailed unit tests for each segment.

5.1.2 Beta Testing. During the Beta testing phase, users identified several bugs and limitations in the system:

- **Program breaking in the back-end:** Users in the first user testing group identified actions that broke the system.
- **Semantic Analysis Difficulties:** Users highlighted that they struggled to understand how to use the semantic analysis functionality before it was explained. While the tool provided capabilities to analyze semantics, it wasn't always straightforward to apply.
- **Diagram size:** As the lexical rules for a language increased, it became harder to see details in the automaton diagram.

After each iteration of the Beta test, feedback was collected, bugs were addressed, and improvements were made in the next development sprint.

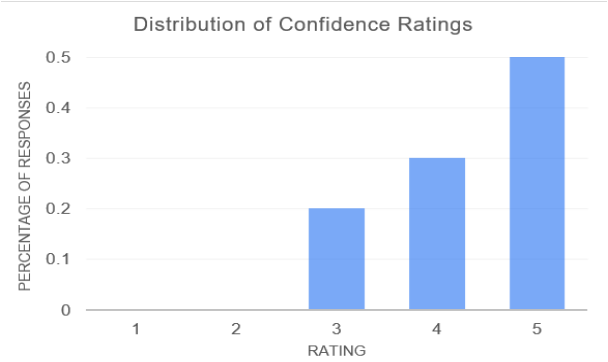
5.1.3 Final System Performance. Post-iterative development and evaluation:

- The program was made more decoupled and modularised to ensure that actions out of the norm did not break the entire system [50].
- Semantic labels we rewritten multiple times and tooltips were added to make semantic analysis tasks more intuitive. The position of the semantic analysis check was also moved to make the flow of tasks more intuitive which seemed to help as well.

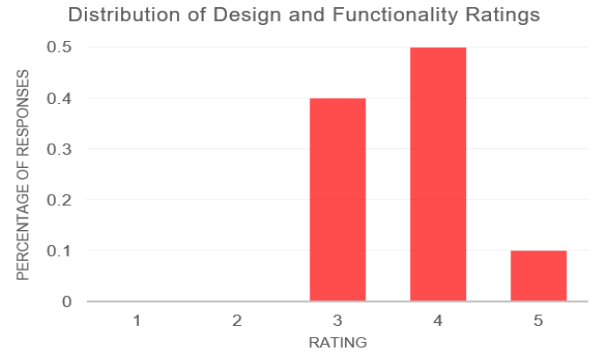
- The automaton diagram was made significantly larger to make it easier to see.

5.2 Results of User Tests

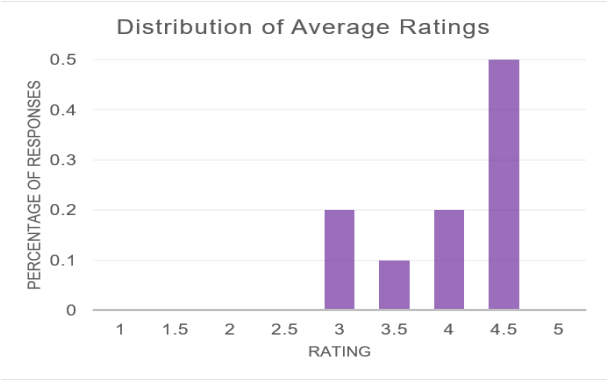
5.2.1 Self-assessment of tool experience. This short survey was given to users **before** instructions and a demonstration on how to use the tool was given. The objective was to gather insights into how intuitive and user-friendly the tool is. As explained above, students rated a set of statements to gauge the extent to which their experience aligned with the statements, measured on a Likert scale with 5 levels from Strongly Disagree (1) to Strongly Agree (5). The results were:



Confidence (Blue): Most respondents feel quite confident in their ability to utilize the tool, with the highest number of responses for rating 5 (the maximum rating). The distribution is generally skewed towards higher confidence levels, with no respondents giving lower confidence ratings (1 and 2).



Design & Functionality (Red): The distribution of ratings regarding the design and functionality of the tool is more evenly spread out. Ratings 3 and 4 have notable percentages, indicating that while many users find the tool design adequate, there's room for improvement.



Average (Purple): The average ratings are primarily concentrated around the scores of 3 - 4.5. The highest percentage of respondents has an average rating of 4.5. The mean of the average ratings is 4.0. This suggests that the overall sentiment towards the tool, when considering both confidence in their ability to use it and the design & functionality, is positive. However, there's still room for improvement, as the very highest average rating (5.0) isn't on the distribution.

It's important to note that the results above represent students' experience before any instructions were given. Once instructions were given, the students were able to use the application confidently. It also includes ratings from user experiences before software improvements as the software was being developed based on the feedback.

5.2.2 Post-Task Survey. The post-task survey looked to discern if the tool proved advantageous, clarified any compiler theory concepts, had any beneficial features or had any conceptual shortcomings or strengths. It also attempted to assess if a tool with this conceptual approach would add additional value to a Compiler Course, specifically with regards to enhancing students' comprehension of compiler theory concepts and increasing interest and engagement in the course. Lastly, it assessed if the tool's cross-platform capability would be useful.

Advantageous Aspects of the Tool: The majority of respondents indicated that they found the tool advantageous for their tasks. This question did not specifically ask for a a reason, but common reasons still given were the automated visualisations allowing them to; visualise their code, understand how lexical analysis and parsing work and prevent getting confused when needing to make changes. Other reasons given were the logical separation of defining the rules of the compiler and "implementing actual parsing" and explaining errors they made.

Clarification of Compiler Theory Concepts: Most respondents stated that the tool helped clarify their understanding of some compiler theory concepts. 30% of respondents stated that the tool didn't clarify anything beyond what they already knew but of that 30%, all selected "Yes" to the question asking if they thought a tool like this would have helped clarify concepts **during** their compiler theory course and

67% mentioned that it helped with managing or debugging their code.

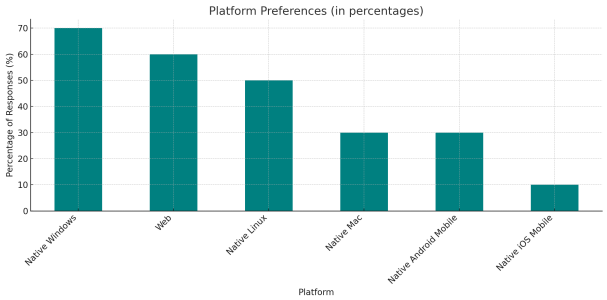
Beneficial Features of the Tool: For this question, several respondents reiterated the benefit of having automated visualisations in making it easier and faster to debug their code. Respondents also mentioned the connection between the code and the visual representation of the automaton diagram and the AST as being beneficial. Other beneficial features mentioned were the "implied sequence in which things have to be done in the tool ... [which] helps with understanding how the order of these things is important", refreshing their memory of semantic analysis and making it less confusing than having to run the code without the tool.

Conceptual Shortcomings and Improvements: Users in the first user testing group identified a few bugs in the system, including actions that broke the system and created an automaton diagram that was too small to see. After the first update, users in the second, third and fourth testing groups did not experience this. A commonly identified limitation of the tool throughout most of the testing was the ability to identify how to use the tool to correctly perform a semantic analysis before an explanation. Each sprint attempted to improve on this and users in the final group didn't have as much of a problem figuring it out before an explanation. Suggestions for improvements included having more prompts and explanations throughout the system, modularising the section for inputting language rules and having further semantic options.

Conceptual Strengths: The primary strength of the tool, as identified by respondents, lies in its automated visualisation of "very abstract concepts", especially the diagrams it produces. Respondents explained that it helps to understand the "logic behind the code/rules", how the inputs are related and "flow through the compiler" and what the errors in your code are directly. One respondent argued that the main conceptual strength is that "the tool encourages one to play around with a DSL. It is nice to have a sandbox to play around with and be able to visualize a DSL."

Value in Compiler Course: 90% of participants stated that a tool with this conceptual approach would have been a valuable asset in enhancing their comprehension of the theory during their Compiler course, with 10% uncertain. The responses to the question about whether a tool with this conceptual approach would have increased their engagement and interest in compiler theory during their Compiler Course also had 90% "Yes" responses with "10%" uncertain. No respondents selected "No" for either.

Platform Preferences:



Most respondents (70%) would use a tool like this as a native Windows application and 60% as a Web Application. The second most preferred desktop platform for a tool like this was Linux with half the respondents, while 30% of the respondents would use a Mac version of the application. Notably, since none of the respondents selected both Android and iOS simultaneously, at least 40% of the respondents would use a tool like this as a native mobile application.

6 Discussion

6.1 System Test Discussion

6.1.1 Relevance of Alpha Testing. Alpha testing primarily focused on the technical capabilities of DSLDoodle. Achieving substantial code coverage was instrumental in ensuring that the tool's outputs were reliable. The success of the lexer, parser, and visualization tests indicated that the tool's core mechanism was sound, which served as a good foundation for the functionalities built on top of it.

6.1.2 Feedback-Driven Iterative Development. The feedback received during the Beta phase was invaluable. While Alpha testing ensured that the output was technically sound, Beta testing highlighted areas of usability and reliability that were not immediately apparent in the development phase. An example was the semantic analysis that users had difficulty with. It's possible that users had some difficulty figuring out without the additional explanation because, unlike with lexing and parsing, they had never used a program that automated semantic analysis. In their Compiler Theory course, they used Ply without any visualisation tool, and Ply primarily has dedicated functionality for lexing and parsing [3]. By embracing a feedback-driven approach, DSLDoodle has been refined into a tool that is both more user-friendly and technically robust.

6.2 Tool and Concept Evaluation Discussion

6.2.1 Intuitiveness and User Experience. The initial self-assessment results indicated that the majority of the participants found the DSLDoodle tool to be intuitive and user-friendly, even before receiving any explicit instructions on its usage. This is an encouraging sign, as it suggests that users can easily get started with the tool, potentially making it more likely to be adopted in educational settings [51].

However, there is still room for improvement, as indicated by the spread in the "Design & Functionality" ratings.

6.2.2 Educational Potential. The post-task survey results reinforced the educational potential of DSLDoodle. Many respondents acknowledged the tool's ability to clarify certain compiler theory concepts which speaks to its effectiveness as a teaching aid. Furthermore, respondents repeatedly cited the automated visualizations as being particularly beneficial, showcasing that a visual, interactive approach can bridge the gap between abstract theoretical concepts and tangible understanding.

6.2.3 Practical Utility. Beyond educational insights, the tool's ability to aid in debugging and managing code was also highlighted by respondents. This underscores the tool's practical utility, implying that it could be useful even beyond a purely educational context.

6.2.4 Cross-Platform Benefits. The results from the post-task survey question on platform preferences show that students would use a tool like this on multiple different platforms. The results highlight the benefit of having a cross-platform application and hence support the decision to develop that application with Flet, which allowed the project to compile one code base for multiple platforms.

A notable result is that 40% of respondents included a native mobile application in their selection. None of the respondents attempted to use DSLDoodle on mobile, so the veracity of their estimation could be challenged from that angle. The result, however, could also potentially be an underestimate of the number of students that would use a tool like this on mobile. A web application, which many respondents selected, can also be used on mobile. Furthermore, all students in the study have access to a desktop computer or laptop, which is not the case for many students interested in higher learning in South Africa due to *socio-technical marginalization* [53]. Therefore a mobile compiler teaching tool like DSLDoodle can potentially also assist with accessibility and inclusivity.

6.2.5 Limitations and Areas of Improvement. While the iterative development and evaluation approach has helped address some of the initial challenges, there are still areas of improvement that emerged from the feedback. The issues surrounding the semantic analysis segment highlight an important design challenge.

6.2.6 Conceptual Strengths as an Educational Tool. DSLDoodle's conceptual strengths, as acknowledged by the respondents, largely revolve around its visual and interactive nature. Visualizing abstract concepts has long been regarded as an effective pedagogical strategy, and the feedback confirms this. The sandbox nature of the tool, which allows students to play and experiment with DSLs, is another significant strength, promoting an active learning approach [29].

6.2.7 Implications for Compiler Theory Education. The feedback suggests that tools like DSLDoodle can serve as a valuable supplementary tool for the practical component of a Compiler Theory course. Given the positive feedback on the tool's potential to enhance comprehension and engagement, educators could consider integrating similar tools into their curriculum.

6.2.8 Benefit of building DSLs. Based on the literature review, the project theorised that it may be beneficial to have students build a Domain Specific Language (DSL) to increase student engagement and interest, as opposed to some of the other previously mentioned approaches to having a practical component in a Compiler Course [19]. Interestingly, none of the respondents explicitly highlighted the use of DSLs as a reason for the tool increasing their engagement or interest in the subject, despite 90% of participants stating that a tool like this in their Compiler Course would have done so. This does not necessarily mean that it wasn't a factor in the experience, but simply that it wasn't raised in the responses. One potential reason for this is that the participants were already given DSLs to build for the practical component of their Compiler Theory course. Therefore, unlike the visualisations, it was not something different from the evaluation benchmark in their mind, based on previous experience.

6.3 Key Takeaways

The key takeaways from the above discussion are:

1. **Visualization Aids Comprehension and Engagement:** One of the overarching findings from the user feedback was the immense value derived from the visualization capabilities of DSLDoodle. It supports the pedagogical viewpoint that interactive visualization, especially in subjects with a high degree of abstraction, can significantly improve understanding.
2. **Cross-Platform capabilities are useful:** The results show that students would use a teaching tool like DSLDoodle on multiple different platforms. Creating a tool that can be used natively on multiple platforms is clearly useful, and potentially makes the tool more accessible and inclusive.
3. **Practical Utility Beyond Education:** Although DSLDoodle was developed with an educational focus, the feedback highlighting its potential for debugging and managing code opens doors for its application beyond educational contexts.
4. **DSLDoodle as a Supplementary Tool:** Feedback consistently underscored the potential of an application like DSLDoodle as a complementary resource in Compiler Theory courses. Its integration could effectively bridge the gap between theoretical lectures and practical understanding, as well as potentially increase engagement and interest in Compiler Theory courses.

7 Future Work

Given the findings and the identified areas of improvement, the following paths are suggested for future endeavours:

- **Explanatory Features:** To address issues the challenges surrounding the semantic analysis segment, additional modules or tutorials can be integrated into the tool to offer clearer explanations and demonstrations.
- **Replication with a different participant pool:** To further test the hypothesis regarding the effectiveness of DSLs in bolstering interest, the evaluation could be carried out amongst students who were not previously given assignments on building Domain Specific Languages while being taught compiler theory.
- **Advanced Debugging Capabilities:** Building upon its practical potential, the tool can integrate advanced debugging features, making it a more comprehensive solution for developers working with DSLs.
- **Back-end of the Compiler:** Very little previous work has been done on visualising compilation stages at the back-end of the compiler (code optimisation and generation) [46]. Adding this was part of the initial plan before the second researcher on the project had to leave. Adding these capabilities to the tool in future work could be a novel contribution to Compiler Education.

8 Conclusions

The development and subsequent testing of DSLDoodle emerged from a pressing need to address the inherent challenges faced in teaching Compiler Theory, namely the conceptual complexity and lack of interest. As highlighted throughout the discussion, this tool not only showcased its potential to make abstract concepts more comprehensible and increase interest and engagement in the subject, but it also demonstrated a degree of utility in practical contexts beyond education.

DSLDoodle is a potentially very effective tool in enhancing Compiler Theory education. Its design ethos, grounded in interactive visualisation and domain-specific languages, has proven its worth. As education continues to evolve in the digital age, tools with the conceptual approach of DSLDoodle are poised to play a pivotal role in shaping how compiler theory is taught and understood.

Future work could explore incorporating additional explanatory features, replicating the evaluation with different participant pools, incorporating more debugging features to build the application into a developer tool and adding visualisations of processes at the back end of the compilation process.

References

[1] [n. d.]. *Operating System Market Share Worldwide.* https://gs.statcounter.com/os-market-share Accessed: 01 September 2023.

[2] I Elaine Allen and Christopher A Seaman. 2007. Likert scales and data analyses. *Quality progress* 40, 7 (2007), 64–65.

[3] M. Amiguet. 2010. Teaching compilers with python.

[4] N. Anderson and T. Gegg-Harrison. 2013. Learning computer science in the 'comfort zone of proximal development'. In *Proceeding of the 44th ACM technical symposium on Computer science education.* https://doi.org/10.1145/2445196.2445344 Preprint.

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

[6] caleb531. 2023. *Automata: A Python Library for Simulating Finite Automata, Pushdown Automata, and Turing Machines.* https://github.com/caleb531/automata Accessed: 2023-05-08.

[7] F. Campagne. 2014. *The MPS language workbench: volume I.* Vol. 1.

[8] A.T. Corbett and J.R. Anderson. 2001. 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.* 245–252.

[9] John Deacon. 2009. Model-view-controller (mvc) architecture. *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf* 28 (2009).

[10] Saumya Debray. 2002. Making Compiler Design Relevant for Students Who Will (Most Likely) Never Design a Compiler. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education.* 220–223.

[11] Alina Mihaela Dima and Maria Alexandra Maassen. 2018. From Waterfall to Agile software: Development models in the IT sector, 2006 to 2018. Impacts on company management. *Journal of International Studies (2071-8330)* 11, 2 (2018).

[12] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. 2002. Graphviz—open source graph drawing tools. In *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers 9.* Springer, 483–484.

[13] M. Eysholdt and H. Behrens. 2010. 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.* 307–309.

[14] flet dev. 2022. flet. https://github.com/flet-dev/flet. GitHub repository.

[15] Flutter. 2021. Issue 79775. https://github.com/flutter/flutter/issues/79775 GitHub repository issue.

[16] Diptiban Ghillani and Diptiben H Gillani. 2022. A perspective study on Malware detection and protection, A review. *Authorea Preprints* (2022).

[17] A. Ghoor. 2023. Literature Review: Domain Specific Language Teaching Tool. (2023). University of Cape Town - Computer Science Department - CSC4019Z.

[18] Alec Helbling, Duen Horng, et al. 2023. ManimML: Communicating Machine Learning Architectures with Animation. *arXiv preprint arXiv:2306.17108* (2023).

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

[20] Daryl H Hepting. 2006. Ethics and usability testing in computer science education. *ACM SIGCSE Bulletin* 38, 2 (2006), 76–80.

[21] C.E. Hmelo-Silver. 2004. Problem-based learning: What and how do students learn? *Educational Psychology Review* 16, 3 (2004), 235–266.

[22] Ajay Jangra, Gurbaj Singh, Jasbir Singh, and Rajesh Verma. 2011. Exploring testing strategies. *International Journal of Information Technology and Knowledge Management* 4 (2011), 297–299.

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

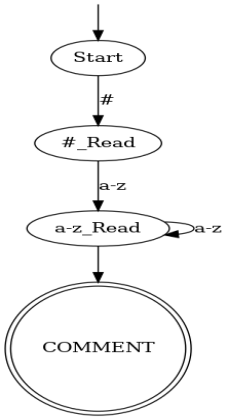
[24] D. Kundra and A. Sureka. 2016. An experience report on teaching compiler design concepts using case-based and project-based learning approaches. In *2016 IEEE Eighth International Conference on Technology for Education (T4E).* Preprint.

[25] M.E. Lesk and E. Schmidt. 1990. *Lex A Lexical Analyzer Generator.* Technical Report.
[26] Thomas Mahatody, Mouldi Sagar, and Christophe Kolski. 2010. State of the art on the cognitive walkthrough method, its variants and evolutions. *Intl. Journal of Human–Computer Interaction* 26, 8 (2010), 741–785.
[27] Nikola Marangunić and Andrina Granić. 2015. Technology acceptance model: a literature review from 1986 to 2013. *Universal access in the information society* 14 (2015), 81–95.
[28] Mari Matinlassi. 2004. Evaluating the portability and maintainability of software product family architecture: Terminal software case study. In *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004).* IEEE, 295–298.
[29] J.J. McConnell. 1996. Active learning and its use in computer science. In *Proceedings of the 1st conference on Integrating technology into computer science education - ITiCSE '96.* Preprint.
[30] J. Melton and A.R. Simon. 1993. *Understanding the new SQL: a complete guide.* Morgan Kaufmann.
[31] M. Mernik, J. Heering, and A.M. Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.
[32] M. Mernik and V. Zumer. 2003. An educational tool for teaching compiler construction. *IEEE Transactions on Education* 46, 1 (2003), 61–68.
[33] T.Æ. Mogensen. 2009. *Basics of compiler design.* Torben Ægidius Mogensen.
[34] F. Moreno-Seco and M.L. Forcada. 1996. Learning compiler design as a research activity. *Computer Science Education* 7, 1 (1996), 73–98.
[35] Kshirasagar Naik and Priyadarshi Tripathy. 2011. *Software testing and quality assurance: theory and practice.* John Wiley & Sons.
[36] T. Parr. 2013. *The definitive ANTLR 4 reference.* 1–326 pages.
[37] V. Pech, A. Shatalin, and M. Voelter. [n. d.]. 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.* 165–168.
[38] M. Procopiuc, O. Procopiuc, and S. H. Rodger. 1996. Visualization and Interaction in the Computer Science Formal Languages Course with JFLAP. In *Technology-Based Re-Engineering Engineering Education Proceedings of Frontiers in Education FIE'96 26th Annual Conference.* 121–125.
[39] D. Raggett, A. Le Hors, and I. Jacobs. 1999. *HTML 4.01 Specification.* Technical Report recommendation, 24. W3C.
[40] Zoran S. Rakic, Predrag Rakic, and Tomaž Petric. 2014. miniC Project for Teaching Compilers Course. In *ICIST 2014.* 2.
[41] Ken Schwaber. 1997. Scrum development process. In *Business Object Design and Implementation: OOPSLA'95 Workshop Proceedings 16 October 1995, Austin, Texas.* Springer, 117–134.
[42] D. Shaffer, W. Doubé, and J. Tuovinen. 2003. Applying Cognitive load theory to computer science education. In *Annual Workshop of the Psychology of Programming Interest Group.* 333–346.
[43] Ben Shneiderman and Catherine Plaisant. 2010. *Designing the user interface: Strategies for effective human-computer interaction.* Pearson Education India.
[44] Ali Hassan Sial, Syed Yahya Shah Rashdi, and Abdul Hafeez Khan. 2021. Comparative analysis of data visualization libraries Matplotlib and Seaborn in Python. *International Journal* 10, 1 (2021).
[45] S. Stamenkovia and N. Jovanovia. 2021. Improving participation and learning of compiler theory using educational simulators. In *2021 25th International Conference on Information Technology (IT).* Preprint.
[46] S. Stamenković, N. Jovanović, and P. Chakraborty. 2020. Evaluation of simulation systems suitable for teaching compiler construction courses. *Computer Applications in Engineering Education* 28, 3 (2020), 606–625.
[47] A. Tashildar, N. Shah, R. Gala, T. Giri, and P. Chavhan. 2020. Application development using flutter. *International Research Journal of Modernization in Engineering Technology and Science* 2, 8 (2020), 1262–1266.
[48] P. D. Terry. 1985. CLANG - a Simple Teaching Language. *SIGPLAN Notices* 20, 12 (dec 1985), 54–63. https://doi.org/10.1145/382086.382627
[49] The MathWorks, Natick, MA 2012. *Matlab.* The MathWorks, Natick, MA.
[50] André van der Hoek and Nicolas Lopez. 2011. A Design Perspective on Modularity. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development* (Porto de Galinhas, Brazil) *(AOSD '11).* Association for Computing Machinery, New York, NY, USA, 265–280. https://doi.org/10.1145/1960275.1960307
[51] Sonya E Van Nuland, Roy Eagleson, and Kem A Rogers. 2017. Educational software usability: Artifact or Design? *Anatomical sciences education* 10, 2 (2017), 190–199.
[52] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. 2001. PEP 8–style guide for python code. *Python. org* 1565 (2001), 28.
[53] Marielle Venturino and Yu-Chang Hsu. 2022. Using whatsapp to enhance international distance education at the University of South Africa. *TechTrends* 66, 3 (2022), 401–404.
[54] Dave West, Mike Gilpin, Tom Grant, and Alissa Anderson. 2011. Water-scrum-fall is the reality of agile for most organizations today. *Forrester Research* 26, 2011 (2011), 1–17.
[55] B.G. Wilson and P. Cole. 1996. Cognitive teaching models. In *Handbook of research in instructional technology*, D. H. Jonassen (Ed.). 601–621.

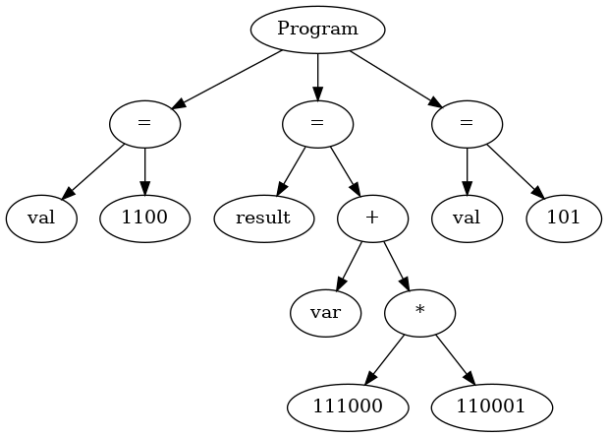
A Appendix

A.1 Appendix A: Automaton and AST Diagrams

A.1.1 An Automata Diagram. Diagram generated from r"[#][a-z]+"



A.1.2 An AST Diagram. Diagram generated from parser rules that define a calculator for binary literals.



A.1.3 A Semantic Analyis. Tables generated from a semantic check of the following code:
val = 1100
result = var + 111000 * 110001
val = 101

Variable	Error
var	Variable used before being defined
val	Variable has already been defined

Symbol Table (Variables)
val
result

A.2 Appendix B: Post-Task Survey

- Did you find the tool advantageous for the tasks you performed?
  - **Short Answer**
- Did the tool clarify your understanding of any compiler theory concepts?
  - **Short Answer**
- In what ways, if any, did the tool prove beneficial for you?
  - **Short Answer**
- Do you think a tool with this conceptual approach would have been a valuable asset in enhancing your comprehension of the theory during your Compiler course?
  - **Yes / No / Maybe**
- Do you think a tool with this conceptual approach would have been a valuable asset in increasing your engagement and interest in compiler theory during your Compiler Course?
  - **Yes / No / Maybe**
- Were there conceptual shortcomings or limitations in the tool that you noticed, and how do you think they can be improved?
  - **Short Answer**
- What would you consider as the main conceptual strengths, if any, of the tool?
  - **Short Answer**
- On what platforms would you use a tool with this conceptual approach? (Select all that apply):
  - **Native Windows Desktop Application**
  - **Native Linux Desktop Application**
  - **Native Mac Desktop Application**
  - **Web Application**
  - **Native Android Mobile Application**
  - **Native iOS Mobile Application**