Project Proposal: Visual DSL Teaching Tool

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

Julian Janisch
Computer Science
University of Cape Town
South Africa
jnsjul010@myuct.ac.za

CCS CONCEPTS

• Programming Languages • Compilers • Computer Science Education

KEYWORDS

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

1 Project Description

1.1 What is the problem

The proposed research project will be to create and assess an interactive visual tool for developing a Domain Specific Language as part of the process of learning Compiler Theory. The project will have to draw on established educational theory, and the current research completed in these fields, to address oversights in current implementations.

1.2 Why is it important

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. Computer Scientists should therefore have an understanding and appreciation of the theory, as it is a foundational topic of the science. Ensuring it is taught well is then, by extension, very important.

1.3 What are the issues

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

1.2.1 Concepts. Dealing with low-level translation process concepts can be complicated, and some of the theory is quite abstract [1]. 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.

1.2.2 Interest. 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. [2]

1.4 Related Work

1.4.1 Use of Interactive Visualisation.

1.4.1.1 Simulators. 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. [3] surveyed several different simulators and evaluated them based on their characteristics and features, as well as the amount of Compiler Theory topics covered.

For example, LISA, developed by Mernik and Zumer [4], covered the most amount of Compiler Theory topics (46.2%) by 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.

Another example is JFlap which covered 38% of Compiler Theory content and with a focus on defining automata and grammar [4]. 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

1.4.1.2 Compiler Creators. There are some Compiler creating tools that are not designed for educational purposes, but still provide interactive visualisations to aid the language-building process. Jetbrains' MPS, an open-source language workbench that lets developers develop new Domain Specific or General Purpose Languages, uses a projectional editor, which means that it

does not limit developers to text editing but makes it possible to visualise and edit a representation of the Abstract Syntax Tree [5]. And ANTLR, a lexical and parser generator that supports multiple languages, has a useful graphical interface for interactively visualizing parse trees and debugging grammars as you build your compiler, shown in Figure 2 below [6].

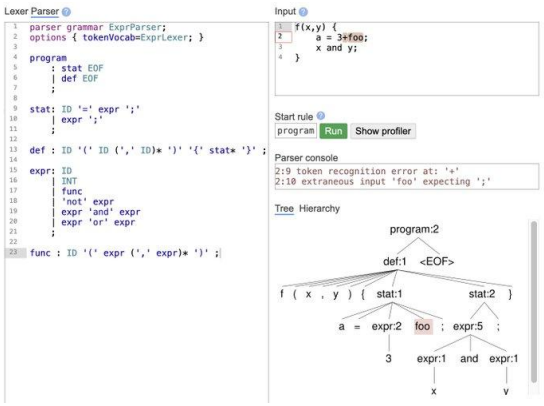


Figure 2

1.4.2 Use of 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. [7] 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.

1.5 Oversights in previous work

1.5.1 Not Cross-Platform. None of the interactive visualisation simulators, 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 visualisations as LISA and JFlap, could be a potential solution.

1.5.2 Minimal Coding. While the implementations of the simulators illustrate the underlying data structures quite clearly, there is often not much coding required. This level of abstraction might make the gap between the interactive simulators and using a real-world parser too great.

1.5.3 No Compiler Backend Visualisations. Almost all visualisations we found are focused on the front end of the compiler [3]. Significantly less work has been done on backend visualisation.

1.5.4 No Tracking of Variables Across Compilation Stages. Since the only visualisation tools we found do either frontend or backend visualisation, there are none that can be used to track individual variables as they travel through the compiler, from source to generated code.

2 Problem Statement

The need for an Interactive Visual 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.

2.1 Project Problem

2.2.1 Front-End. There are a lot of current attempts to visualise the Front-End of the Compiler. However, as mentioned above, none of the implementations currently work across multiple platforms and the simulators with the best visualisations often don't require much coding.

To address these gaps, this paper proposes building an app that allows students to fully code up a domain-specific language, while simultaneously being able to view multiple underlying data structures, and that works across multiple platforms.

The proposal is for it to also draw on the positive lessons of using Domain-Specific Language [2] in a simple, familiar language like Python. Ply, a lexer and parser generator for creating DSLs, is a user-friendly Python implementation of Lex and Yacc that can be used [8].

Essentially, this aspect of the project will look at the educational value of a cross-platform app that wraps Python's PLY and ANTLR's interface, with more visualisations of the underlying data structures.

The educational value being the app's ability to solve the two current observed challenges, namely to aid in the understanding of complex compiler topics and make the subject more interesting.

2.2.2 Back-End. There are very few visualisation tools for code optimisation and visualisation. Additionally, there don't seem to be any that visualises the entire compilation process. Since papers on visualisation tools commonly conclude that visualisations help students to understand compilers better, it makes sense to explore how further visualisations can improve the process. Finally, back-end visualisations tend to be implemented for a specific language. This leaves an opening to create a system that could generate back-end visualisations for any user-created DSL.

2.2 Project Aims

2.2.1 Ahmed Ghoor - Front-End.

1) Asses how useful do students find a cross-platform compiler creation app, with increased visualisations of underlying data structures, in understanding complex compiler theory concepts.
2) Assess to what extent does the implementation of a cross-platform app with interactive visualisations and domain-specific language coding capabilities in a familiar language like Python increase student engagement and interest in compiler subjects.

2.2.2 Julian Janisch - Back-End.

1) Design visualisations for code optimisation and code generation, and assess their usefulness to students learning compiler theory.
2) Assess how useful tracking and visualising the transformations that individual variables undergo during code optimisation and code generation is when teaching compiler theory.

3 Procedures and Methods

3.1 Program Architecture

The tool can be split up into a GUI and a compiler. The compiler can then further be divided into a front-end and a back-end.

3.1.1 GUI. The GUI will be written using Flutter, a declarative Dart-based application framework [9]. The GUI will provide the following key features:

- An interface to define tokens and a grammar (i.e. the DSL)
- An interface to enter source code in the defined language.
- Visualisations of the compilation process
- Highlighting to track individual pieces of code as they travel through the compiler

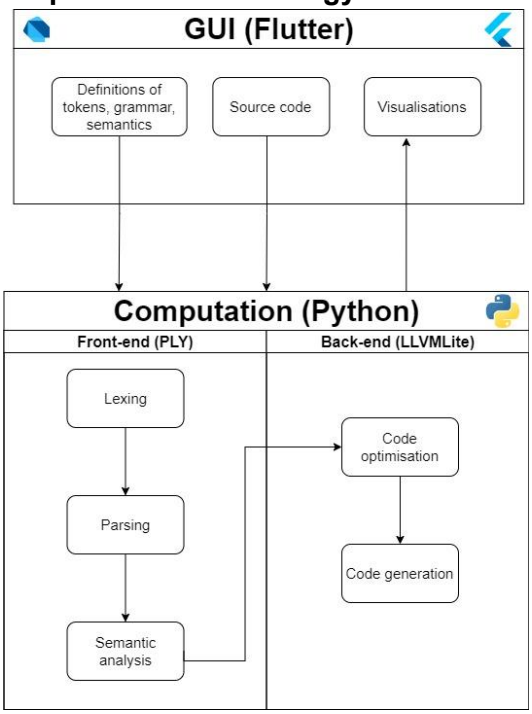
The Python code will be called from Dart, which is how the GUI and the compiler will communicate.

3.1.2 Compiler. The compiler will be written in Python. It will be built off of existing libraries and will provide the functionality required by the GUI. Each stage will need to output data in a specific format that provides the information required for visualising that stage.

3.1.2.1 Front-end. The front-end will be in charge of lexing, parsing and semantic analysis. The first two tasks will be performed using the PLY library. The last one will need to be implemented by creating an additional module in python.

3.1.2.2 Back-end. The back-end will handle code optimisation and code generation. This will be done using LLVMLite, a Python library providing a wrapper around a subset of LLVM's features. LLVM is a project providing many libraries and tools for compilation [10]. What we are most interested in are the code optimisation and code generation tools that it provides.

3.2 Implementation Strategy



To start, we will need to work together to set up the development environment. This will include creating a git repository, downloading required libraries and setting up a minimal GUI and compiler. These will not need to be functional, but will just serve as a common base from which we will start work on our individual components. The

common GUI will be separated into 3 sections (On one screen for desktop and separate screens for mobile), drawing on ANTLR's interface:

- Section one: A text box to add compiler creation code (e.g. token rules, grammars etc)
- Section two: A text box to add a piece of input code in the newly created language, for processing.
- Section three: A section that outputs interactive visualisations of the underlying data structures.

3.2.1 Front-end.

1. Take the lexer, parser and semantic analysis Python code written in section 1 of the GUI Flutter interface, send it to a Python script and build it using PLY.
2. Take the high-level code written using the newly created language's rules from Section 2 and process it through the new compiler.
3. Send the outputs back to the Flutter GUI to output visualisations of the underlying data structures in Section 3.

3.2.2 Back-end. The back-end will be implemented by completing the following tasks:

- Take the front-end's output and transform it into the LLVM intermediate representation (IR)
- Run an LLVMLite optimisation pass on the IR code
- Add information to optimised code for visualisation
- Display code optimisation visualisation in Section 3 of the GUI
- Generate output code (most likely assembly)
- Add any additional information required for visualisation
- Display code generation visualisation in Section 3 of the GUI.

We will use an agile approach, as that will encourage us to build and test our software in a modular way. It will also help us to always have a working piece of software as we progress.

3.3 Evaluation

3.1.1 Evaluation Design. Given the time-frame in which this project needs to be completed, it would not be feasible to design the experiment exactly as it was laid out in the initial Literature Review [11], 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:

1) Providing it to students who have already studied a university-level compilers course to complete various tasks, before getting them to answer questions about it through surveys and focus groups.
2) Providing it to teaching staff to evaluate whether the tool accurately captures the Compiler Theory concepts taught in class.

3.1.2 Participants. The student participants will be Honours students from the University of Cape Town, who have previously taken a compiler course. The teaching staff participants will be teachers that have previously taught a compiler theory course. There would need to be a minimum of just one teaching staff to provide an expert evaluation on the content.

3.1.3 Measures. The project will gather qualitative and quantitative data on the students' experience with the tool. Specifically, what they did and didn't like about the tool, how it compares to other tools they've encountered, and how the tool affected the participants' understanding of Compiler theory, confidence in creating a compiler, interest in the skill and the learning experience as a whole.

3.1.4 Procedure. The evaluation will occur in two distinct phases: preliminary testing and ultimate evaluation. A pilot study will first be conducted on the system to gather insights on apparent concerns. This stage will involve focus groups where participants examine the tool's functionalities and provide verbal feedback. The subsequent phase entails participants engaging with the tool and completing a questionnaire in the absence of researchers.

4 Ethical, Professional and Legal Issues

In this study involving human subjects, addressing ethical concerns is very important. Before engaging in the study, participants must carefully review and sign a consent form. Ethical approval will be obtained from both the computer science department and the university. To ensure confidentiality, no identifiable information about participants will be disclosed in the published results. Data collected will be stored on a password-protected computer, with access limited to the pair of researchers, supervisor and second-reader in this project. All software that will be 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.

5 Anticipated Outcomes

5.1 Anticipated DSLTool

We expect to create a tool which can be used to define a domain-specific language, and which can compile source code in that language into assembly. The tool will provide visualisations of the code and underlying data structures at every step as it is processed and transformed by the compiler.

5.2 Anticipated Outcomes From Use

We expect that our tool will aid students in understanding how compilers work, and do so more effectively than existing tools. It will also make the subject matter and building of compilers seem more interesting and relevant. Success in this regard will be measured in our interviews with and surveys completed by students, where we will take the feedback gathered, analyse it, and draw our conclusions.

6 Project Plan

6.1 Risks

6.1.1 Complexity. Compilers are complex pieces of software. We may run into challenges when implementing ours, especially since we are aiming to support any DSL that can be defined within the limits of the tool. To mitigate this, we will be making use of existing libraries that will handle some of the more challenging tasks for us.

6.1.2 Time Constraints. We only have a limited amount of time in which to complete the project before we need to perform user testing. There are a lot of features that we could add to the tool, so we will need to ensure we only implement a small subset of those. Additionally, we will make the program highly modular, so that if we don't finish a module, we still have a functional tool. Compilers lend themselves to a modular architecture, so this should be achievable.

6.1.3 Bugs Surface During User Testing. Bugs are an inevitable part of software engineering. However, we need to ensure that they do not affect the user testing process in any significant way, as the feedback gathered is what we need to answer our research questions. We will minimise this risk in two ways. Firstly, we will limit the scope of the project. This reduces the number of places in which bugs could arise, and gives us more time to write good code. Secondly, we will test our program thoroughly. We will do integration testing to ensure the separate stages and libraries all work together, and end-to-end testing to ensure a smooth user experience.

6.2 Timeline

Please see the GANNT Chart under Appendix A.

6.3 Required Resources

We already have access to the following required resources:

- computers for software development
- shared storage for a git repository
- various software libraries (all publicly available)

The only resources we do not yet have access to are a group of students and at least one, but preferably more, teaching staff willing to test our tool. We will focus on finding these participants later in the project.

6.4 Deliverables

The following deliverables are still to be submitted:

- Final project proposal
- Completed DSLTool software
- Final paper draft
- Final paper
- Website
- Poster

6.5 Milestones

The milestones for the project are as follows:

- Project Proposal Presentation: 25 April
- Final Project Proposal Due: 28 April
- Ethics Applications Preferred Deadline: 5 May
- Initial Software Feasibility Demonstration: 22 May
- Ethics Application Final Deadline: 26 May
- Complete Draft of Final Paper Due: 28 August
- Project Paper Final Submission: 11 September
- Project Code Final Submission: 15 September
- Final Project Demonstration: 26 September

6.6 Work Allocation

6.6.1 Ahmed Ghoor - Front End. Ahmed will be working on the Front-end of the Compiler (not the front-end as in the user interface): Lexing, Parsing and Semantic Analysis. He will create the GUI visualizations in Flutter, as well as the code to do the processing in Python under the hood.

6.6.2. Julian Janisch - Back End. Julian will be working on the backend of the compiler. He will transform the data that the frontend outputs into something that the backend will understand, i.e. into data that can be passed through the optimisation and code generation phases. He will also create visualisations of these phases of compilation.

REFERENCES

[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].

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

[3] 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.

[4] Mernik, M. and Zumer, V. (2003) “An educational tool for teaching compiler construction,” IEEE Transactions on Education, 46(1), pp. 61–68

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

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

[7] 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)

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

[9] Tashildar, A., Shah, N., Gala, R., Giri, T. and Chavhan, P., 2020. Application development using flutter. International Research Journal of Modernization in Engineering Technology and Science, 2(8), pp.1262-1266.

[10] The LLVM Compiler Infrastructure Project. Retrieved April 23, 2023 from https://llvm.org/.

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

Appendix A: Gannt Chart