Logo Compiler

Project Report
ICPL Seminar Spring Semester 2025

University of Basel
Faculty of Science
Department of Mathematics and Computer Science

Alexander Marggraf
Max Reinert
Pascal von Fellenberg

June 11, 2025

Contents

1 Abstract

2 Methodology

Our initial goal was to implement a compiler into the existing XLogoOnline implementation, but we quickly gave up on this as it proved to be too challenging. Instead, we decided to build our own standalone web application that allows the user to write, compile, render, and benchmark code.

To compile our Logo code, we started off with the raw code itself. Our implementation requires the code to be present in a .lgo file as used by XLogoOnline, or it can be written directly in the application. Files ending in .lgo are essentially .txt files with a different extension. This was convenient as we could easily parse the contents of the files into a string. This string was then parsed into an AST using the grammar, lexer, and parser of XLogoOnline. With the AST in hand, we used our own compiler to generate a JavaScript AST, which we later used to generate executable code. The compiler iterates over the AST and uses the action set, which holds all the built-in functions of the renderer, to convert the Logo AST nodes into JavaScript AST nodes. However, sometimes additional steps were necessary to ensure correct execution, which will be specified in the Implementation section. Furthermore, the compiler has different modes that determine how the code is generated. The default mode is direct access. This means that each function call of the action set is done via dot notation. In contrast, the array access mode uses bracket notation to access the functions of the action set, and the direct mode uses no function calls to the action set and instead implements the functions directly inside the code. With the JavaScript AST, we used Escodegen to generate the JavaScript code, which can then be used to render onto the screen.

To render the code, the application uses a prefix that contains some mathematical functions, constants, and helper functions, as well as the code itself, to generate the final executable script, which is then run asynchronously. For rendering, we used the HTML canvas element, which provides all the necessary functionality for a basic turtle graphics renderer.

The implemented benchmark functionality is quite straightforward. We generated a predetermined routine of running scripts ten times and calculating the average execution time and runtime, as well as the standard deviation. These results are then displayed directly within the application. The benchmarks themselves are programs that test various aspects of the Logo language, such as loops and recursion.

3 Polymorphic Inline Caches - an Overview

4 Implementation

5 Performance of Different Approaches

6 Comparison With XLogoOnline

7 Conclusion

[?]