Logo Compiler

Alexander Marggraf
Max Reinert
Pascal von Fellenberg

June 11, 2025

**University
of Basel**

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

Our compiler tests various approaches as to how exactly the compiled code should look like. We compare the performance of those results. To know why different approaches might behave differently from a performance point of view, we present one possible optimization modern JavaScript-engines employ. What follows is an introductoin to polymorphic inline caches.

3.1 Terminology

- A method-call on an object *o* is also called a *message send* with *receiver o*.

- A programming language is AOT-compiled if the code is compiled before the program is run (i.e. at compile time).

- A programming language is JIT-compiled if the code is compiled at runtime.

- A place in the code where a message send happens is called a *call site*.

3.2 Overview

From this point on, we only consider object-oriented languages which are compiled to machine code (either JIT or AOT). Otherwise the overhead from interpretation of the language would dominate and it would not make sense to optimize the way it is done with inline caches.

Message sends in object-oriented languages are very expensive if no caching is implemented in the runtime. A reason for this is the inheritance rules of those languages that have to be ensured. The consequence of this is that before each message send, an expensive lookup procedure must be called which determines, based on the type of the object, which piece of machine code is run.

The first optimization one can make is to store the results of this lookup procedure in a simple lookup cache (lookup table) which maps the type of the receiver together with the name of the message to the address of the corresponding machine code. So at the call site, instead of calling the lookup procedure, the lookup cache is consulted and the expensive lookup procedure is only performed on a cache miss. This optimization improves performance considerably. (source??????????????????????????????)

However, this lookup cache still implies at least one indirection per message send. Inline caching addresses this problem by storing the address of the procedure corresponding to the last message send directly at the call site, together with a check that ensures the type of the object is still the same (note that the message name is assumed to be constant per call site so it doesn't have to be checked). When an inline cache miss happens, the correct procedure has to be found - either through the lookup table or the lookup procedure - and it's address inserted at the call site. Additionally, the check has to be updated. Inline caches work under the assumption that the receiver type of a call site doesn't change often.

Now what happens if this assumption doesn't work, i.e. the receiver type of a call site changes all the time? Then we have more overhead compared to the simple lookup table: instead of just consulting the lookup table, we also have to update the call site in each call. To circumvent this, polymorphic inline caches do not only store one address in a call site, but several. So at each call site, there is conceptually switch statement on the type of the receiver which has call instructions with the procedure corresponding to that type. In case of a cache miss (the receiver's type is not in the polymorphic inline cache of this call site), this call site is extended with that type and the corresponding procedure.

What happens in modern JavaScript engines is that depending on the behaviour of a call site, different methods of caching are employed. If the receiver type of a call site doesn't change (a monomorphic call site), it uses an inline cache. If the receiver type changes (a polymorphic call site), a polymorphic inline cache is used. In call sites where the type changes a lot (megamorphic call sites), more dynamic approaches are needed which are not covered in this overview.

3.3 Relevance to our Project

We implement a compiler which has different modes, varying the way methods are called. The benchmark shows how the different modes impact performance, i.e. how well the JavaScript engine is able to optimize the compiled code.

4 Implementation

5 Performance of Different Approaches

6 Comparison With XLogoOnline

7 Conclusion

[1]

References

[1] Mr. Inline Cache, "Cool Reference." [Online]. Available: https://www.youtube.com/watch?v=dQw4w9WgXcQ. Accessed 8 Jun. 2025.