Logo Compiler

Alexander Marggraf
Max Reinert
Pascal von Fellenberg

June 11, 2025

**University
of Basel**

Contents

1   Abstract

This report documents the development of a Logo compiling and rendering web application and compares different approaches in compiling regarding compile time and run time. Furthermore, the web application is also compared to the commercial web app XLogoOnline, which allows users to write and run Logo code, to understand how compiling versus interpreting code changes the performance of the applications. To achieve this, we built three compilers which used different approaches for method calling—namely member expression, computed member expression, and hard-coding—and our own renderer, making it accessible through a user interface. To test the implementation, we implemented a benchmarking functionality and wrote our own benchmarks, designed to test different aspects of the programming language. We successfully managed to build the aforementioned application and found that at runtime the three different compilers were roughly equally performant, but at compile time the hard-coded approach takes considerably longer. We also found that the interpreted approach has performance trade-offs the more complex the program is. These findings suggest that our tested compiling strategies have small effects on the actual performance of the compiled code, and that interpreting might considerably slow down execution time for larger programs.

2   Introduction

Logo is an educational programming language that was developed in 1967 and is commonly used for its simple and intuitive turtle graphics. [1] The PrimaLogo project of the University of Basel and ETH Zürich uses Logo to teach children in the 5th and 6th grades the basics of programming. To teach, they used the XLogoOnline website, developed in collaboration with the University of Trier, which allowes users to write and execute Logo code. The concrete usage of the language is outlined in the Section 4.

The main problem with the XLogoOnline implementation is that it is interpreted, which can lead to poor performance. This motivated us to implement a compiler for the Logo language to improve performance and to understand how different compilation techniques affected performance in both compile time and run time.

Our initial goal was to integrate a compiler into the existing XLogoOnline implementation, but we abandoned this idea because it proved too challenging. Instead, we decided to build a standalone web application that allowes users to write, compile, render, and benchmark Logo code.

To compile Logo code, we started 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 text files with a different extension. This allows the program to easily parse the contents into a string, which is then converted into an AST using the grammar, lexer, and parser of XLogoOnline.

With the AST prepared, we use our compiler to generate a JavaScript AST, which is later transformed into executable code. The compiler iterates over the AST and uses the action set, which contains all built-in functions of the renderer, to convert Logo AST nodes into JavaScript AST nodes. In some cases, additional steps are necessary to ensure correct execution; these are described in the Implementation section.

The compiler supports different modes for code generation. The default mode is direct ac-

cess, where each function call of the action set uses member expression. The array access mode uses computed member expression to access the functions of the action set. The direct mode avoids function calls entirely and instead implements the functions directly in the code. After generating the JavaScript AST, we use Escodegen [2] to produce the final JavaScript code, which is then used for rendering.

To render the code, the application uses a prefix containing mathematical functions, constants, and helper functions, along with the compiled code, to produce the final executable script. This script is run asynchronously on the HTML canvas element, which provides all necessary functionality for a basic turtle graphics renderer.

Benchmarking our compiler was straightforward. We executed a predetermined routine of running scripts ten times, calculated the average execution time and runtime, and computed the standard deviation. These results were displayed directly in the application. The benchmarks themselves tested various aspects of the Logo language, such as loops and recursion.

3 Polymorphic Inline Caches

Our compiler implements various approaches as to how the compiled code looks. 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 introduction 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 that 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) that 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. ([3])

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 its 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 a 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 behavior 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.

4 Logo Overview

To know what we had to implement, it is good to have an overview of what constructs the turtle graphics language Logo supports. Here are the relevant components. This is not the full language, just the part relevant for this report.

4.1 Basic Commands

Movement of the turtle	
`fd` k, `bk` k	go k steps[1] forward or backward respectively
`rt` θ, `lt` θ	turn θ degrees to the right or left respectively
`setx` x, `sety` y, `setxy` x y	set the position of the turtle or just one component of it
`home`	go to the origin (same as `setxy 0 0`)
`setheading` θ	set the direction in which the turtle points, in degrees
State of the turtle	
`setpc` c	set the pen color to c, where c can be a color by name or a list of 3 numbers (RGB values from 0 to 255)
`setsc` c	set the background color to c
`setpw` w	set the width of the pen to w
`pu`, `pd`	disable or enable the pen, respectively (pen up, pen down)
Clearing the screen	
`wash`	clear the screen but keep the state of the turtle (pen state, position and direction)
`cs`	clear the screen and the state of the turtle
Miscellaneous	
`print` v	print v to the console
`wait` h	wait h hundredths of a second

4.2 Variables

`make "`n` `v	declare variable with name n and initialize it with the value v
`make :`n` `v	assign the value v to the variable with name n

When referencing a variable, one must precede the name with a colon.

4.3 Control Structures

`repeat `k` [`s`]`	repeat the statements s k times
`if `c` [`i`] [`o`]`	if condition c is fulfilled, execute statements i otherwise the statements o
`while [`c`] [...]`	repeat the statements inside the square brackets until the condition c is false
`to `n` `p s `end`	declare a procedure with name n which executes the statements s. p is the list of parameters accepted by that procedure which can be used by the statements s. Procedures cannot return anything. The entry point of the source code in our implementation is the procedure with the name `main`.
`stop`	If used inside a procedure this is an early return, inside a loop (`repeat` or `while`) it serves as a break statement, exiting just the loop

4.4 Example

```
1  to main
2    setpc red
3    square 100
4  end
5  to square :l
6    repeat 4 [
7      fd :l
8      rt 90
9    ]
10 end
```

Listing 1: Logo code that draws a red square

5 Implementation

Our implementation consists of a compiler that compiles from Logo to JavaScript. The compiled code uses an object that we call an action set. This action set defines what happens for each basic command in the Logo code. This means that for each basic command, a method with the same name exists on the action set, which defines its behavior. In a visual application, such an action set renders the commands in a canvas, but this can be adapted to other circumstances (e.g., for testing). Calls on such an action set `act` can be done in multiple ways in JavaScript; here are the three we decided to test:

1. as a member expression: `act.fd(100)` (in this report called *direct access*)

2. as a computed member expression: `act['fd'](100)` (in this report called *array access*)

3. pasting the source code of the method to be called into the call site (in this report called *hard coded*)

5.1 Compiler

We use the existing parser from the xLogoOnline project to get an abstract representation (abstract syntax tree or AST) of the Logo code. This AST is then transformed into a JavaScript AST, which is converted back to a string using the library escodegen[2]. The resulting string can then be run in various JavaScript environments.

Note that all the constructs in Logo have matching constructs in JavaScript[2]. This means that our compiler is rather simple in that we mostly just map Logo AST nodes to the corresponding JavaScript AST nodes.

5.2 Action Set Calls

The basic commands (see Section 4.1) are mapped to action set calls in the compiler. How those are generated depends on the strategy the compiler is configured with. The cases *direct access* and *array access* are straight forward:

Logo code	*direct access*	*array access*
`setxy 42 0`	`act.setxy(42, 0)`	`act['setxy'](42, 0)`

However, we had to do a bit more to implement the strategy *hard coded*. The goal of this strategy is to - instead of calling a method on the action set - paste the source code of said action set call to the call site. To get the source code of a method, we use the `toString` method defined on the methods of an object. This string is then parsed with esprima, and the resulting AST is transformed in the following manner:

- The parameters of a method become local variables that get declared and initialized at the beginning of the resulting call site.

- Member expressions on `this` (which represents the action set) get modified to have the `__act_` prefix instead of `this.`.

- All of this code gets wrapped in a scope to not redefine the same variable if a method gets called twice[3].

To see this in action, we take a look at what happens when transforming the call `rt 100` to a *hard coded* call.

```
1  rt(angle) {
2    this.turtleAngle = (this.turtleAngle + angle) % 360;
3  }
```
Listing 2: This is what `act.rt.toString()` returns.

becomes

```
1  {
2    let angle = 100;
3    __act_turtleAngle = (__act_turtleAngle + angle) % 360;
4  }
```
Listing 3: The effective code at the call site.

If inside a method there is a call to a member of `this`, the same rule is applied. We opted not to recursively hard code these calls as well, since recursion would pose a problem. Another thing to note is that identifiers in logo get a prefix in the resulting JavaScript code to avoid naming collisions with the source code of the action set. In the following example we transform the call `fd :l`.

---

[2] `repeat` gets transformed to a `for` loop, `stop` to a `return` or `break` based on the context it appears in and procedures to functions

[3] This could be circumvented by renaming the local variables, but due to time constraints we weren't able to do that

```
1  fd(steps) {
2    const newX = this.turtleX + steps * Math.cos(this.toRadians(this.turtleAngle))
         ;
3    const newY = this.turtleY + steps * Math.sin(this.toRadians(this.turtleAngle))
         ;
4    if (this.penDown) {
5      this.ctx.beginPath();
6      this.ctx.moveTo(this.turtleX, this.turtleY);
7      this.ctx.lineTo(newX, newY);
8      this.ctx.stroke();
9    }
10   this.turtleX = newX;
11   this.turtleY = newY;
12 }
```

Listing 4: This is what `act.fd.toString()` returns.

becomes

```
1  {
2    let steps = logovar_l;
3    const newX = __act_turtleX + steps * Math.cos(__act_toRadians(
         __act_turtleAngle));
4    const newY = __act_turtleY + steps * Math.sin(__act_toRadians(
         __act_turtleAngle));
5    if (__act_penDown) {
6      __act_ctx.beginPath();
7      __act_ctx.moveTo(__act_turtleX, __act_turtleY);
8      __act_ctx.lineTo(newX, newY);
9      __act_ctx.stroke();
10   }
11   __act_turtleX = newX;
12   __act_turtleY = newY;
13 }
```

Listing 5: The effective code at the call site.

Now one might wonder where the objects and functions with the prefix __act_ are defined, which should correspond to fields and methods in the action set, respectively. Our approach is to compile the action set so that fields become global variables and methods become functions, which all have the required prefix. We then preface the rest of the code with this version of the action set code, such that all the necessary variables and functions are defined.

To do this, we use that the method `act.constructor.toString` returns the source code of the action set class. First, we compile the constructor of this class by transforming the assignments of members to global variable definitions. Then we iterate over all methods and convert them to functions.

5.3 Executing the Compiled Code

In JavaScript, one can use the function `eval(code)` to execute JavaScript code, given as a string. `eval` inherits the scope of the caller, which means that we can "pass" the action set by just defining a variable with the same name as is used in the compiled code and then calling `eval` (see first part of Listing 6). However, `eval` does not work well with bundlers[4] and is quite error-prone because of the implicit inclusion of all these objects.

So we opted to use the `Function` class. Using the `Function` class, one can construct a callable object that executes arbitrary code, but it behaves like a normal function. This means that one

---

[4]we use esbuild as a bundler for our project

can define parameters that have to be supplied to the callable explicitly (see the second part of Listing 6). In our case, the action set and the canvas context have to be supplied to this callable object.

```
1  // using eval
2  let param1 = 2, param2 = 4;
3  eval('console.log(param1 + param2)'); // outputs 6 to the console
4  // using Function
5  let callable = new Function('param1', 'param2', 'console.log(param1 + param2)');
6  callable(2, 4) // outputs 6 to the console
```

Listing 6: Code snippet showing different ways to run arbitrary code in JavaScript

5.4 On Asynchronicity

One implementation detail that wasn't mentioned yet is that the `wait` command introduces asynchronicity into the running code. This means that Logo code procedures using the `wait` command must correspond to async functions in JavaScript. Since we only had limited time to spend on the implementation, we just assume all of the functions to be async and call them using the await expression. To justify this decision, one can mention that await expressions on function calls behave semantically the same as normal function calls if the function being called never uses actual asynchronous code [4].

A benefit from wait calls that we get is that we can interrupt the code whenever such a wait call is made. This is done by having what we call a stopper object, which is passed to the running code. After each wait call, it is checked whether the value of the stopper object has changed, and if so, the program is terminated. Such a stopper object can then be used to stop running code in an application.

6 Performance of Different Approaches

There are now three different ways to compile the code, the direct approach, the array approach and the hard-coded approach. One might now wonder which one of these approaches is the fastest one. To compare these three approaches, multiple benchmarks were introduced in order to test the performance of the approached for different functions. The first benchmarks were to test the basic functionalities to draw on the canvas and to write simple code. Here benchmarks for movement, pen manipulations but also repeat blocks, condition blocks and while blocks were implemented. Since these benchmarks only test the basic functionality, three additional benchmarks to test more applicable cases were added. So a benchmark for recursions, one for a big picture and a benchmark for an animation were also introduced. For all the benchmarks, the compile time as well as the runtime was measured. In order to get a proper measurement, the average over ten executions was calculated since one execution might vary in time. The result of these benchmarks are shown in the following graphicFigure 1.
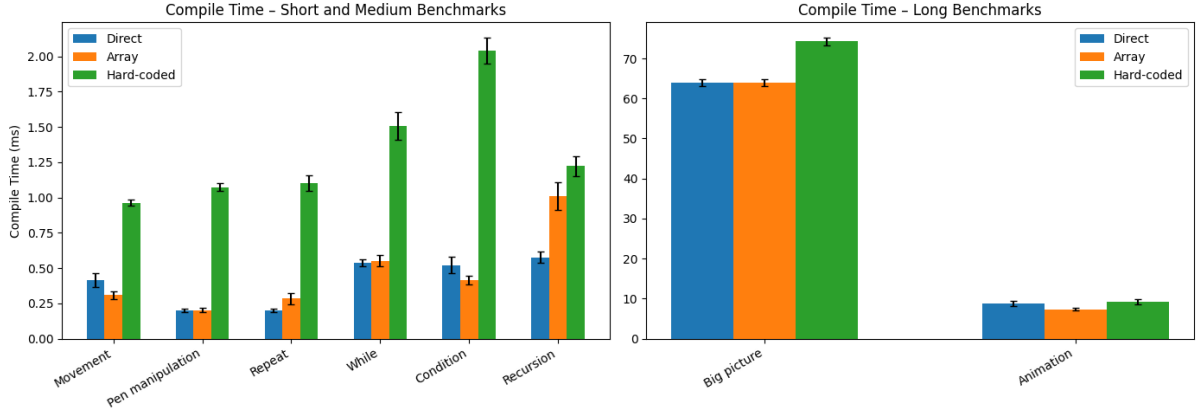
Figure 1: The measurement of the compile time.

InFigure 1 the average of the measured compile time for eah benchmark and compiler approach is shown. It can be seen that for all benchmarks, the hard-coded approach is the slowest one and the direct as well as the array approach gave pretty similar results. The benchmark for recursion on the other hand also shows a difference in the direct and array approach, here the direct approach is being faster than the array approach. For the more complex benchmarks like the big picture benchmark and the animation benchmark, we can see that the differences observed before are not as visible anymore. Although the hard-coded approach is slower for the big picture benchmark and the array approach is faster for the animation benchmark.

After looking at the compile time, the runtime was also measured since the compiled code is different using the three approaches and this might also lead to significant differences.
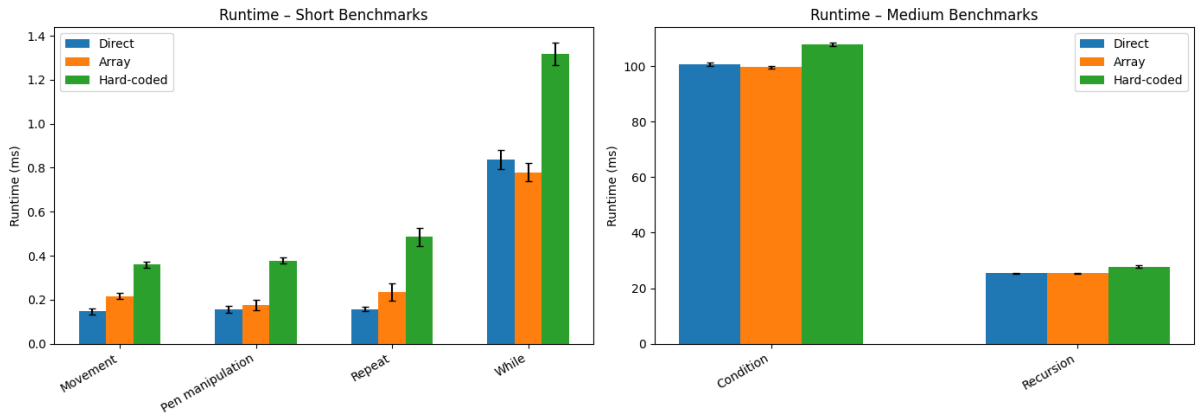

Figure 2: The measurement of the run time.

In this graphFigure 2 the difference in runtime of the code obtained by using the different compiler approaches can be observed. Here the animation benchmark is missing since the runtime of this benchmark is not as significant as the others and this result is also shown in the comparison to xLogoOnline. For the actual runtime of the code, it is also obtained that the hard-coded approach leads to an increased runtime. The direct approach is also almost always faster than the array approach with two exceptions. The array approach is faster for the while benchmarks and the condition benchmark. Although only for the while benchmark, the difference in runtime compared to the direct approach is significant.

So it can be said that the hard-coded approach is the slowest approach of these three if the

compile time and the runtime are compared. In general for the benchmarks the direct approach leads to a shorter runtime but longer compile time compared to the array approach.

Whether the direct approach or the array approach is faster, cannot be said since it is a trade-off between compile time and runtime. If the code has to be compiled multiple times compared to be run, the array approach might be faster but if the code is compiled once and run very often, the direct approach might lead to better results.

7   Comparison With XLogoOnline

Up to this point, only the three compilers were compared. It is also interesting to compare the results to xLogoOnline, which does not use a compiler but only an interpreter. Although this makes a comparison between the compiler for Logo and xLogoOnline more complex since we do not have a compiled time and runtime for xLogo. In order to make a propper comparison, the time it takes for xLogo to go from the input code to the final picture on the canvas is measured. For the three compiler, the compile time and the runtime were added since this is also the time it takes form the actual code to the picture on the canvas. Although this is not a real comparison between the interpreter and compilers, it is still possible to make a fair comparison even though the renderers for our implementation nad xLogo are also taken into account. In order to keep the results pretty visible, we split the benchmarks up into the simpler and faster benchmarks for the basic functionalities and the longer benchmarks which represent larger programms. The benchmarks are the same as for the comparison between the compilers.
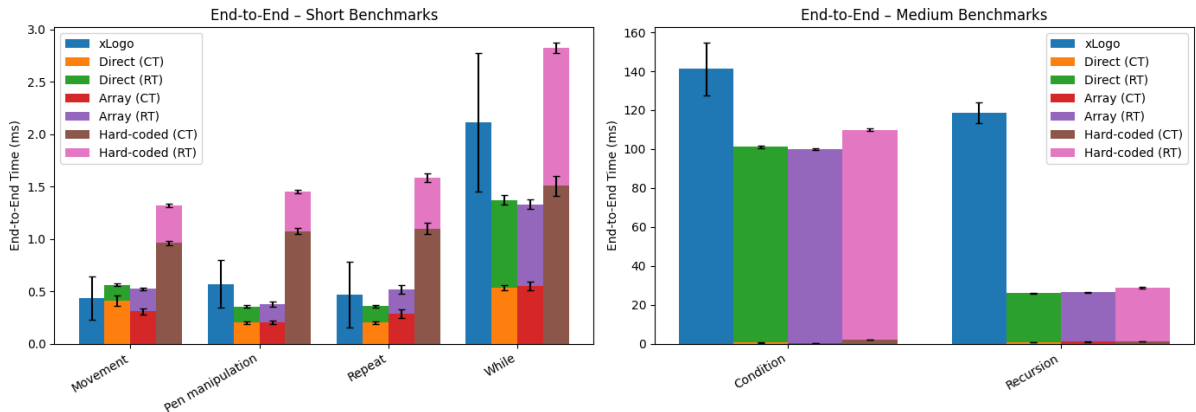

Figure 3: The measurement of the benchmarks for simple programs.

Here in 3 it can be obtained that the results for xLogo vary in the measured time for each benchmark and the margin of error is big. But for simpler tasks, it is obtained that the hard-coded approach is still the slowest of them all with xLogo being the second slowest. Where an actual difference between the direct approach, the array approach and xLogo cna be seen is for the benchmarks for condition blocks and for recursions. For condition blocks, the xLogo is not much slower than the compiler approached but for the recursion, xLogo is significantly slower than all the compiler approaches. For recursions xLogo is almost five times slower than the complier approaches.

In the following graph4, the comparison between the three compiler approaches and xLogo for the big picture benchmark and the animation benchmark can be seen.
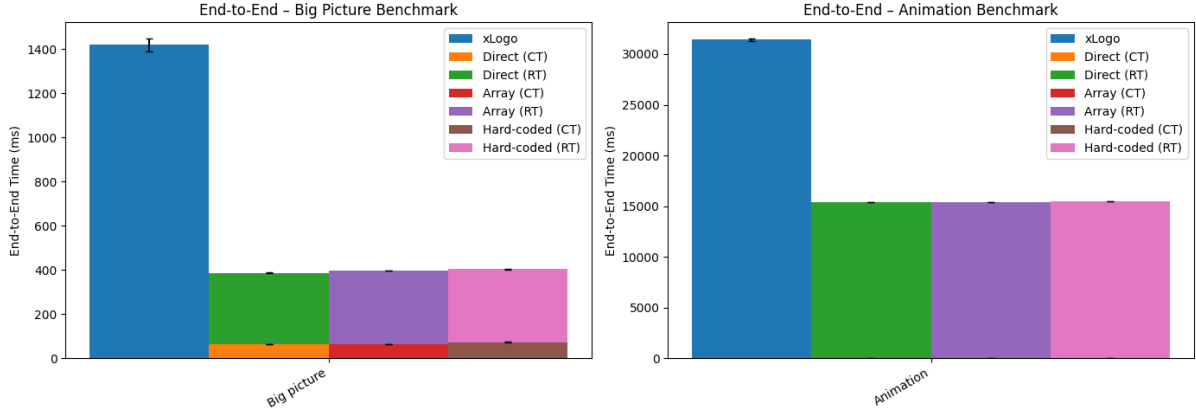
9

Figure 4: The measurement of the compile time for more complex programs.

For both benchmarks, it can be seen that the three compilers perform better than xLogo for complex programs, as already seen for the recursion benchmark. For big pictures, xLogo is about three to four times slower than the compiled versions. There is also a difference in the time, it takes xLogo to finish the animations compared to the compilers. It is important to state tha the time it takes for the animation to finish is heavily dependent on the time it takes to draw each frame. Since xLogo takes twice as long as the compilers to finish the animation, it can be said that xLogo takes significantly longer to draw each frame of the animation.

So we can say that for simple and small programs, containing almost only basics commands, xLogo is as fast as the direct approach and the array approach. But if the programs become larger and more complex like adding while loops, condition blocks or adding animations or recursions, the performance of the compilers becomes better than xLogo. For the simpler and smaller programs, xLogo is faster than the hard-coded approach but if the programs become more complex, the hard-coded approach also becomes faster than xLogo. So for realistic scenarios and complex programs with many condition blocks, loops, animations and recursions, the direct and array approach become significantly better and superior compared to xLogo.

8   Conclusion

Our goal was to write our own web application to allow users to write, compile, render, and benchmark code, as well as to test compiling strategies and compare our implementation against the XLogo implementation. We successfully managed to develop the aforementioned web application, which allows users to select .lgo files or write their own code, compile the code using direct access, array access, and hard-coded approaches, run the code, and see the result rendered onto the screen, as well as benchmark the code and get the results displayed.

Furthermore, we found that the different compiling strategies we implemented performed roughly equally, but the hard-coded approach led to slower compile times. Additionally, we also found that the XLogoOnline implementation performs roughly equal to our implementation for smaller programs but struggles to keep up as the programs become more complex and longer.

These results suggest that on a large scale it does not matter too much whether the code is accessed using member expressions or computed member expressions. It also seems that our approach of hard-coding the functions instead of accessing them tends to lead to slightly slower runtimes. Compiling is essentially equally performant for the array access and direct access modes, but is roughly twice as slow when selecting the hard-coded approach. This is probably

10

due to the high amount of boilerplate code that needs to be generated, as code has to be reparsed using Esprima and functions have to be explicitly reimplemented before being able to use them.

Our findings show that the XLogoOnline implementation struggles to keep up with our implementation as the complexity of the programs increases. This indicates that compiling may lead to better performance when it comes to code execution. However, it must be said that a fair comparison between XLogoOnline and our implementation is not possible. Our application is much less powerful, as XLogoOnline provides four different versions of their service and includes many features that we do not. Furthermore, the renderers being used are also vastly different and will greatly impact performance. We also used a rather small amount of basic benchmarking programs, which may impact the generalizability of our evaluation.

9   Individual Contributions

1. TODO

References

[1] Wikipedia contributors, "Logo (programming language)." https://en.wikipedia.org/wiki/Logo_(programming_language), accessed January 18, 2026, 2025.   Online; accessed 18-Jan-2026.

[2] estools, "GitHub Repository of The escodegen project." [Online]. Available: https://github.com/estools/escodegen. Accessed 17 Jan. 2026.

[3] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *ECOOP '91 proceedings, Springer Verlag Lecture Notes in Computer Science 512*, (Stanford, California), pp. 2–4, 1991.

[4] MDN, "async function - Description." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#Description. Accessed 17 Jan. 2026.