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

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 Logo Overview

To know what we had to implement, it is good to have an overview over 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

4.4 Examples

The entry point of the source code is the procedure which is called `main`.

```
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 which compiles from Logo to JavaScript. The compiled code uses an object which we call 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 it's behaviour. 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 our code, this is called *direct access*)
2. as a computed member expression: `act['fd'](100)` (in our code, this is called *array access*)
3. or pasting the source code of the method to be called into the call site (in our code, this is 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 (source?????????????). The resulting string can then be run in various JavaScript environments.

Note that all of the constructs in Logo have matching constructs in javascript. 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 in. 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: All the parameters of a method become local variables which get declared and initialized at the beginning of the resulting call site. All member expressions on `this` (which represents the actionset) get modified to have the `__act_` prefix instead of `this.`. And all of this code gets wrapped in a scope so as to not redefine the same variable if a method gets called twice[2].

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

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

```
rt(angle) {
  this.turtleAngle = (this.turtleAngle + angle) % 360;
}
```

Listing 2: This is what `act.rt.toString()` returns.

becomes

```
{
  let angle = 100;
  __act_turtleAngle = (__act_turtleAngle + angle) % 360;
}
```

Listing 3: The effective code at the call site.

If inside a method, a method call to `this` happens, the same rule is applied, we opted not to recursively hard code these calls as well. 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 shown in we transform the call `fd :l`.

```
fd(steps) {
  const newX = this.turtleX + steps * Math.cos(this.toRadians(this.turtleAngle))
      ;
  const newY = this.turtleY + steps * Math.sin(this.toRadians(this.turtleAngle))
      ;
  if (this.penDown) {
    this.ctx.beginPath();
    this.ctx.moveTo(this.turtleX, this.turtleY);
    this.ctx.lineTo(newX, newY);
    this.ctx.stroke();
  }
  this.turtleX = newX;
  this.turtleY = newY;
}
```

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

becomes

```
{
  let steps = logovar_l;
  const newX = __act_turtleX + steps * Math.cos(__act_toRadians(
      __act_turtleAngle));
  const newY = __act_turtleY + steps * Math.sin(__act_toRadians(
      __act_turtleAngle));
  if (__act_penDown) {
    __act_ctx.beginPath();
    __act_ctx.moveTo(__act_turtleX, __act_turtleY);
    __act_ctx.lineTo(newX, newY);
    __act_ctx.stroke();
  }
  __act_turtleX = newX;
  __act_turtleY = newY;
}
```

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 functions which all have the required prefix.

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. This order is essential, because the resulting functions might reference the global variables.

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`. However, `eval` does not work well with bundlers[3] 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 which executes arbitrary code but it behaves like a normal function. This means that one can define parameters which have to be supplied to the callable explicitly (see Listing 6). In our case, the action set and the 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 [1].

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. Whenever a wait call happened, it is checked whether the value of the stopper object has changed and if so, the code is terminated. Such a stopper object can then be used to stop running code in a application.

6 Performance of Different Approaches

7 Comparison With XLogoOnline

8 Conclusion

[2]

[3] we use esbuild

References

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

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