Writing a Compiler in Node.js

Jarrod Connolly

May 31, 2024

Abstract

This article asks whether Node.js is a reasonable place to write a compiler, using a toy language as the vehicle. I walk through the lexer, the parser, and the AST, and I explain why I started by targeting Babel instead of emitting machine code. The goal was a side project that would force me to understand Node.js streams, not a production language. What follows is the architecture, the trade-offs, and what I learned building the first version of Complect.

Introduction

Ever wondered if creating a compiler in Node.js is a good idea? I found myself pondering this very question while on the lookout for a side project to explore the depths of Node.js Streams. I'd like you to join me as I walk you through the decisions I made and the setups I tinkered with to bring a toy compiler to life in Node.js.

This post will explain why I used the Streams module from Node.js for this project. I will discuss converting lines of code into something executable by a computer, including the basics of the Lexer, the Parser, and constructing an Abstract Syntax Tree (AST). Also, I will give you an overview of how Node.js and the Stream module performed in the overall context of the project.

I initially spoke about the first incarnation of this project at the OpenJS World 2022 conference. You can find the lecture slides and recording of my talk here. Slides Video

Prepare for a journey where code evolves from text on a screen to an executable program, all thanks to the power of Node.js Streams.

Build a compiler in Node.js?

I wanted to find a project with some depth that would allow me to explore Node.js and the Streams module. I looked for something other than another To-Do app. I have nothing against them, but I do not need another one.

The idea of writing a compiler came up as I was searching for a project, though I was still determining if this was something I could build. I did not take compilers in computer science, and much of the literature was way over my head.

Although most compiler textbooks were overly math-heavy, I saw enough toy compiler projects to give me the confidence to tackle this.

I aimed to maximize my use of Node.js, so I consciously decided to manually write as much code as possible rather than relying on lexer and parser generators.

Why write a compiler

Compilers and related tools are everywhere in the Node.js world, even if we only sometimes recognize them. Tools like Babel, ESLint and Typescript all function in a very compiler-like fashion.

This would also be a project that could keep on giving. As I learned more about compilers, I can expand and grow the project in many ways.

What are compilers

The classic definition is something like: > A compiler is a computer program that translates code in one programming language into code in another language.

This usually refers to code in a high-level language like C++ compiling into code in a low-level language like machine code that could execute directly on some hardware architecture.

Other related tools exist. An interpreter can directly execute high-level code, while a transpiler can compile one high-level language into another.

The compiler world has become blurry, with many popular languages compiling to intermediate languages executing on cross-platform virtual machines like C# and Java. The V8 engine in Chrome and Node.js uses both Interpreters and multiple compilers to optimize and execute JavaScript code.

We have a computer program that translates code from one form to another. This process involves multiple layers that we will examine in more detail.

Compiler Layers

This is not an exhaustive list of compiler layers, nor is it how all compilers work. These are the layers that I would have to implement to make my toy language work.

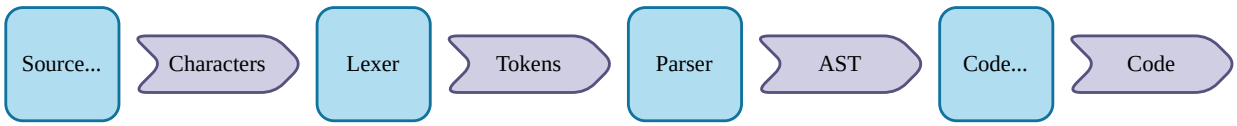


Figure 1: Compiler Layers

Lexical Analysis

During lexical analysis, the first layer called the lexer, translates the source code text into a series of tokens such as identifiers, keywords, operators and others depending on your language. It reads in the source code, character by character, trying to assign meaning to each piece of code.

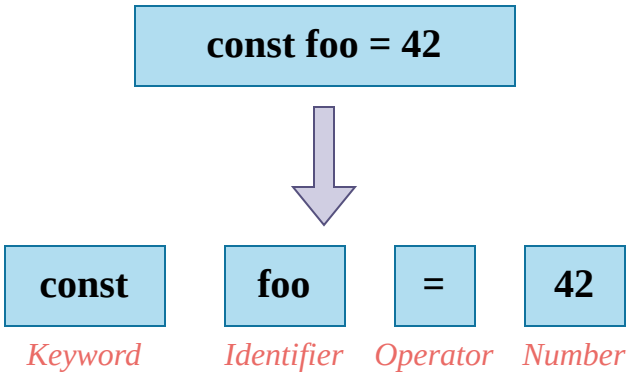


Figure 2: Lexical Analysis

We have identified tokens but still need to understand their intent and how they interact. These tokens are then passed to the next layer for further processing and to build some of this context.

Syntax Analysis

At this layer, the tokens from the lexer are analyzed and formed into a tree structure called the abstract syntax tree (AST). At this layer, our software program is in a tree form that represents it but is abstracted from our language.

This parsing step is relatively complex. Before translating this into a tree structure, we must understand the language tokens and how our language works.

The AST can be used to analyze the program for correctness or optimizations. Here, we could write an interpreter that walks our AST and executes the code node-by-node in the tree.

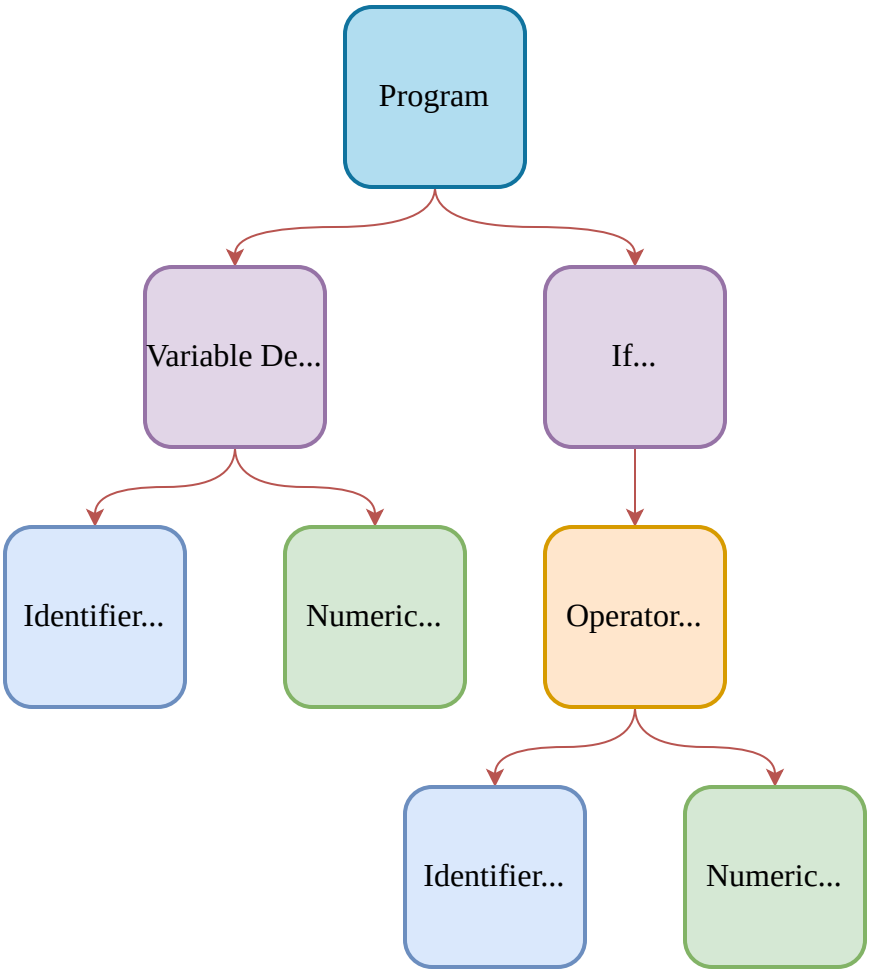


Figure 3: Syntax Analysis

Code Generation

At this point, we use the AST tree structure to generate code in our target language. I knew that at least to start with, everything up to this point would already be a massive undertaking for me. I decided to have my parser build the AST using Babel, allowing me to skip code generation for now.

Node.js Streams

Streams are vital to Node and are present in almost every application you develop. Streams enable us to handle large data volumes without storing them in memory.

You may recognize this pattern in many other systems. Unix's core philosophy is to have small tools that do one thing well, and data gets piped between these tools to perform more complex tasks.

I will write a more detailed post specific to streams at some point, but in short, they provide exactly what I need to stitch my compiler layers together and allow for future changes.

I used Readable streams to read source code from files or stdin on the command line for input.

The compiler layers are written as transform streams, which allows me to separate them and pass data between them. It also allows me to later insert layers between other layers to perform tasks like optimizations or validation.

At the output, I have a Writable stream to output our generated code either to a file or the screen at stdout.

Streams in Node.js are a delight and should be perfect for this task. I was now ready to create a language.

Language

At some point, I stumbled upon a name for my project. I called my compiler Complect. > Complect - interwoven or interconnected

This would be a great name because I was weaving many things together.

Grammar

I needed a grammar for this language that would work well for Lexing and Parsing. A simplified grammar would make building the compiler much less challenging. I wanted to create a grammar I could parse with a straightforward state machine.

Variables

```
make foo 0
assign foo 42
```

Expressions

```
foo = foo + 1
```

Conditional

```
if foo > 0
endif
```

Iteration

```
as foo > 0
repeat
```

Debug

```
print "Hello"
print foo
```

The grammar design choices give our language a charming appearance. This design lets us determine the entire statement from the first token we encounter, so we know immediately how many tokens to read ahead, simplifying the process.

Lexer

I’ve divided the Lexer into two separate transform streams to keep each stage simple. The first Preprocessor stage has a few small tasks. It converts the incoming characters into a set of simple tokens, ready for further processing by the next phase of the lexer.

The second part of the lexer performs more complex tokenization. It processes the preprocessor’s tokens and converts them into tokens our parser can use. We have identified and separated the keywords specific to our language and the reserved words that form part of our language. Additionally, we have converted numbers from their string representation to their numeric values.

Parser

I had the most to learn about the parser. I decided to use the Babel library for the AST representation and focus on understanding the parser states. This would be a significant intermediate step, allowing me to first build a source-to-source compiler or transpiler as a stepping stone to making Complect a complete compiler later.

The parser has to understand the language you are building and the AST structure you are generating. Proper parsing is very complex, even with a straightforward programming language. There are many states to maintain and keep on a stack as you build the more complex tree structures for conditionals and iteration.

Using this knowledge, I built a relatively readable top-down parser that could construct a valid AST with Babel.

Abstract Syntax Tree (AST)

I used a fantastic tool AST Explorer to visualize and reverse engineer how JavaScript is represented in the Babel AST.

Using the Babel APIs allowed me to take the AST generated by my parser and output it as JavaScript.

Now, when looking at this, you might ask: “Wouldn’t it be easier to use some replacements to convert one language to another?” That may be true, but now we have this lovely stream-based compiler framework. We can add new features and eventually swap out the AST for one of our own.

This process taught me more about AST construction from a parser and code generation. I am much more confident about my move to the Binaryen AST to generate WebAssembly code. I can also see one day building a simple AST and generating executable code for at least one hardware architecture like Linux x86_64.

Examples

We now have a toy language that can generate executable JavaScript code. We can direct the JavaScript output from the AST to Node.js and write some usable programs in Complect.

FizzBuzz

One classic test program that I wanted my language to be able to run was FizzBuzz. This program helped flush out the details of getting conditional statements parsed correctly.

```
make i 1
make f 0
make b 0
make output ''

as i <= 16
  f = i % 3
  b = i % 5
  assign output ''
  if f == 0
    output = output + 'Fizz'
  endif
  if b == 0
    output = output + 'Buzz'
  endif
  if output == ''
```

```
    output = output + i
  endif
  print output
  i = i + 1
repeat
```

Fibonacci

Another classic. Fibonacci was a great test case that helped me get loops parsed correctly.

```
make a 0
make b 1
make t 0
make n 12
as n > 0
  n = n - 1
  assign t a
  assign a b
  b = b + t
  print a
repeat
```

Both of these test cases were immensely helpful in getting the basics of Complect working.

Today I Learned

The Node.js core Streams module was perfect for this project. I handled the passing of characters and tokens between the layers, which allowed me to focus on the logic. Streams will be a huge benefit as I extend Complect with more abilities.

I want to add more language features and clean up some code duplication in the parser.

I plan to add another backend using the AST in the Binaryen project. This will allow me to compile directly to Web Assembly.

While writing Complect, I learned a great deal about compilers and realized I have only scratched the surface. I plan to continue working on this project and expand my understanding of this incredible area of computer science.

Since writing this, Complect has grown: an LLVM backend for native binaries, user-defined functions, manual memory management, and SDL graphics. Read the follow-up in Compiler - A Backend.