Let’s Create a Tiny Programming Language

Md Shuvo on May 10, 2022 (Updated on May 11, 2022)

By now, you are probably familiar with one or more programming languages. **But have you ever wondered how you could create your own programming language?** And by that, I mean:

> *A programming language is any set of rules that convert strings to various kinds of machine code output.*

In short, a programming language is just a set of predefined rules. And to make them useful, you need something that understands those rules. And those things are *compilers*, *interpreters*, etc. So we can simply define some rules, then, to make it work, we can use any existing programming language to make a program that can understand those rules, which will be our interpreter.

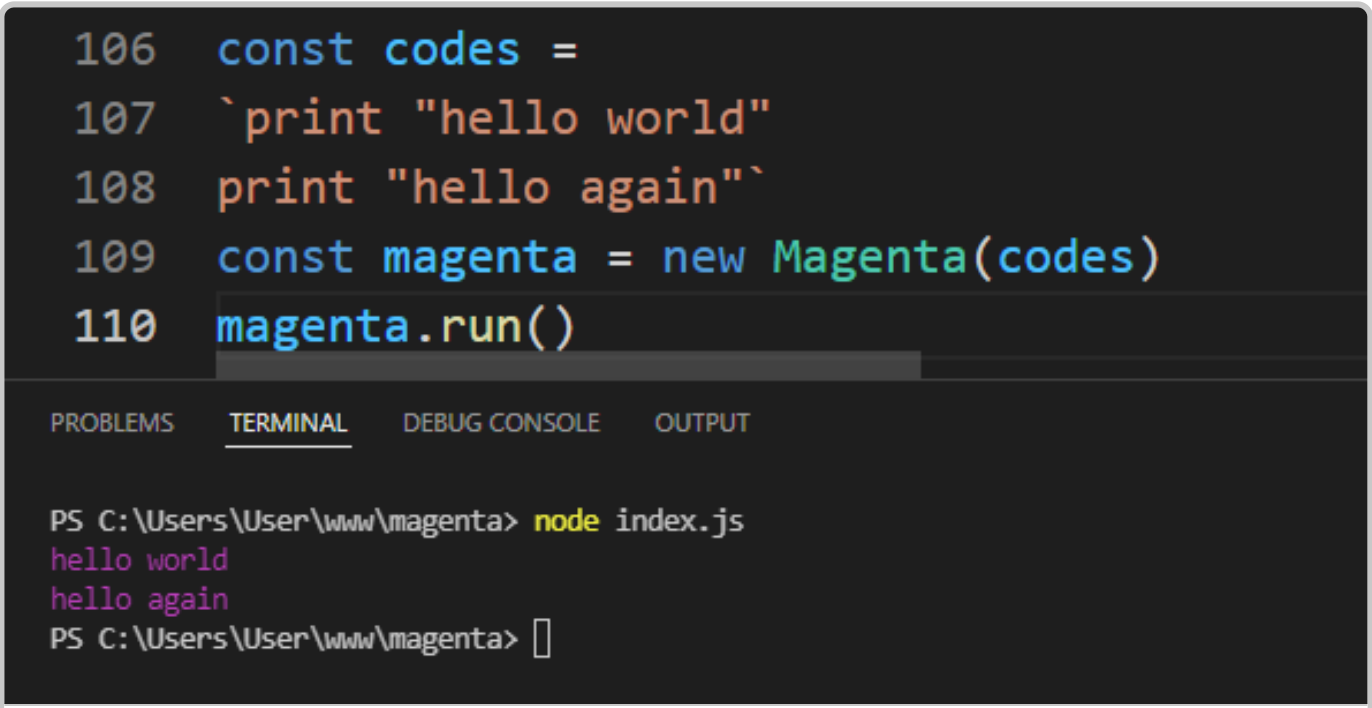
(#aa-compiler) Compiler

A compiler converts codes into machine code that the processor can execute (e.g. C++ compiler).

(#aa-interpreter) Interpreter

An interpreter goes through the program line by line and executes each command.

Want to give it a try? Let’s create a super simple programming language together that outputs magenta-colored output in the console. We’ll call it **Magenta**.



Our simple programming language creates a codes variable that contains text that gets printed to the console... in magenta, of course.

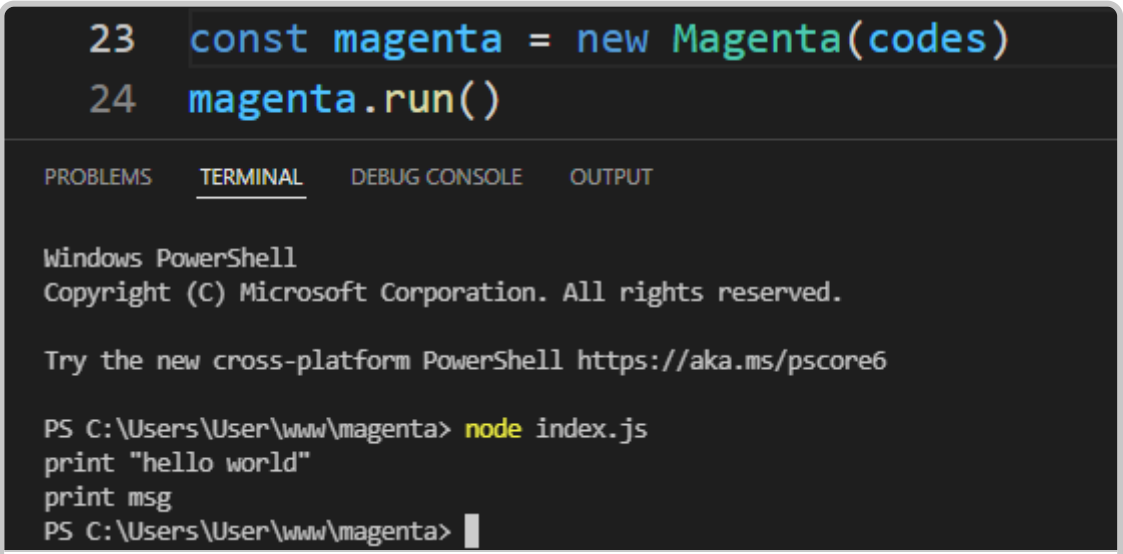
(#aa-setting-up-our-programming-language) Setting up our programming language

I am going to use Node.js but you can use any language to follow along, the concept will remain the same. Let me start by creating an `index.js` file and set things up.

```JavaScript
class Magenta {
  constructor(codes) {
    this.codes = codes
  }
  run() {
    console.log(this.codes)
  }
}

// For now, we are storing codes in a string variable called `codes`
// Later, we will read codes from a file
const codes =
`print "hello world"
print "hello again"`
const magenta = new Magenta(codes)
magenta.run()
```

What we’re doing here is declaring a class called `Magenta`. That class defines and initiates an object that is responsible for logging text to the console with whatever text we provide it via a `codes` variable. And, for the time being, we’ve defined that `codes` variable directly in the file with a couple of “hello” messages.



If we were to run this code we would get the text stored in codes logged in the console.

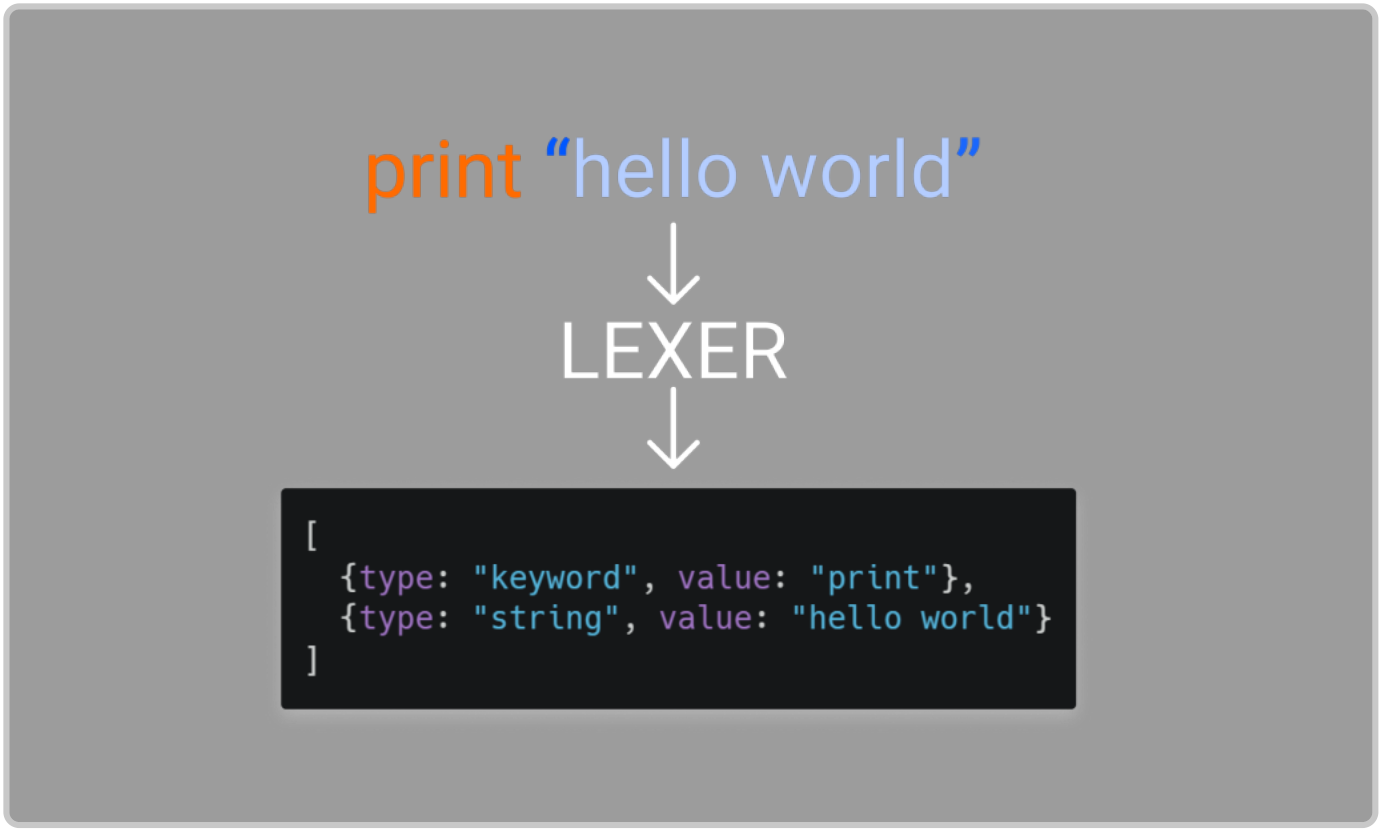
OK, now we need to create a what’s called a Lexer.

(#aa-what-is-a-lexer) What is a Lexer?

OK, let’s talks about the English language for a second. Take the following phrase:

> *How are you?*

Here, “How” is an adverb, “are” is a verb, and “you” is a pronoun. We also have a question mark (“?”) at the end. We can divide any sentence or phrase like this into many grammatical components in JavaScript. Another way we can distinguish these parts is to divide them into small tokens. The program that divides the text into tokens is our **Lexer**.



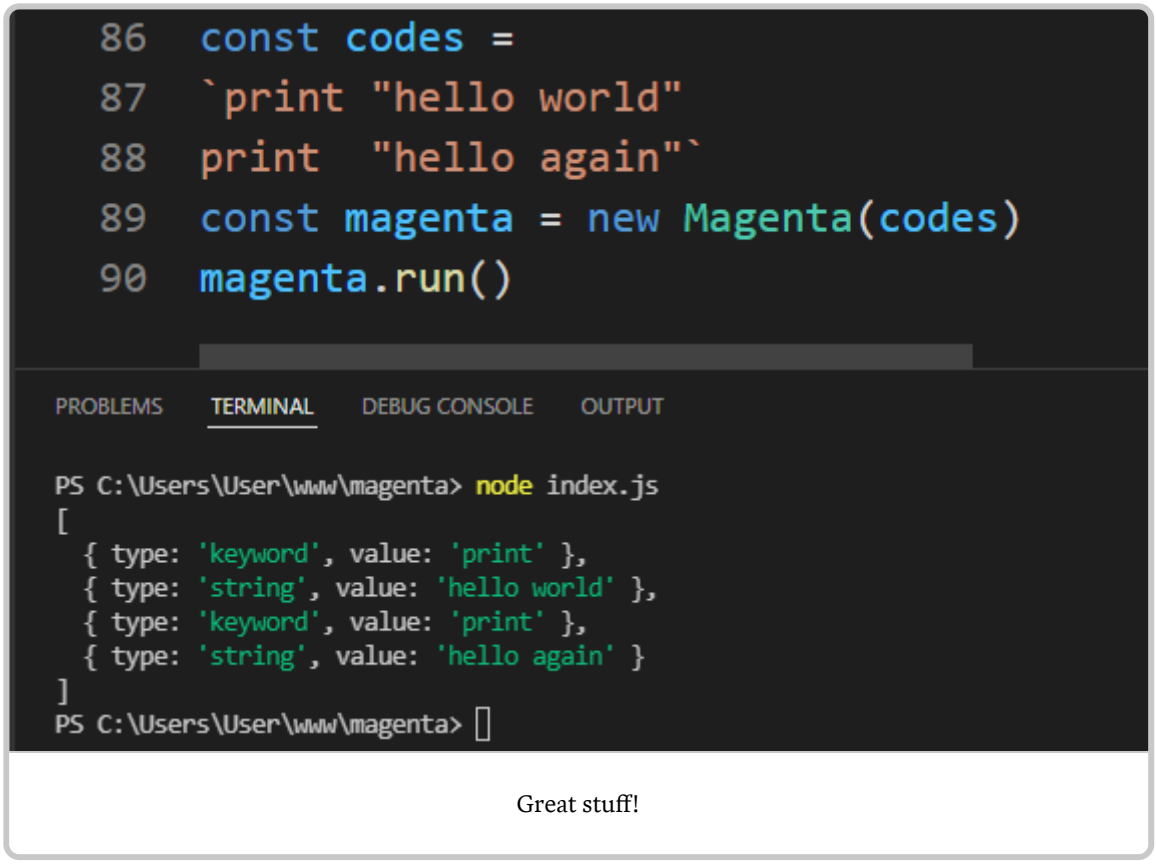
Since our language is very tiny, it only has two types of tokens, each with a value:

1. `keyword`
2. `string`

We could’ve used a regular expression to extract tokes from the `codes` string but the performance will be very slow. A better approach is to loop through each character of the `code` string and grab tokens. So, let’s create a `tokenize` method in our `Magenta` class — which will be our Lexer.

▶ Full code

If we run this in a terminal with `node index.js`, we should see a list of tokens printed in the console.



Great stuff!

(#aa-defining-rules-and-syntaxes) Defining rules and syntaxes

We want to see if the order of our codes matches some sort of rule or syntax. But first we need to define what those rules and syntaxes are. Since our language is so tiny, it only has one simple syntax which is a `print` keyword followed by a string.

Magenta

```
keyword:print string
```

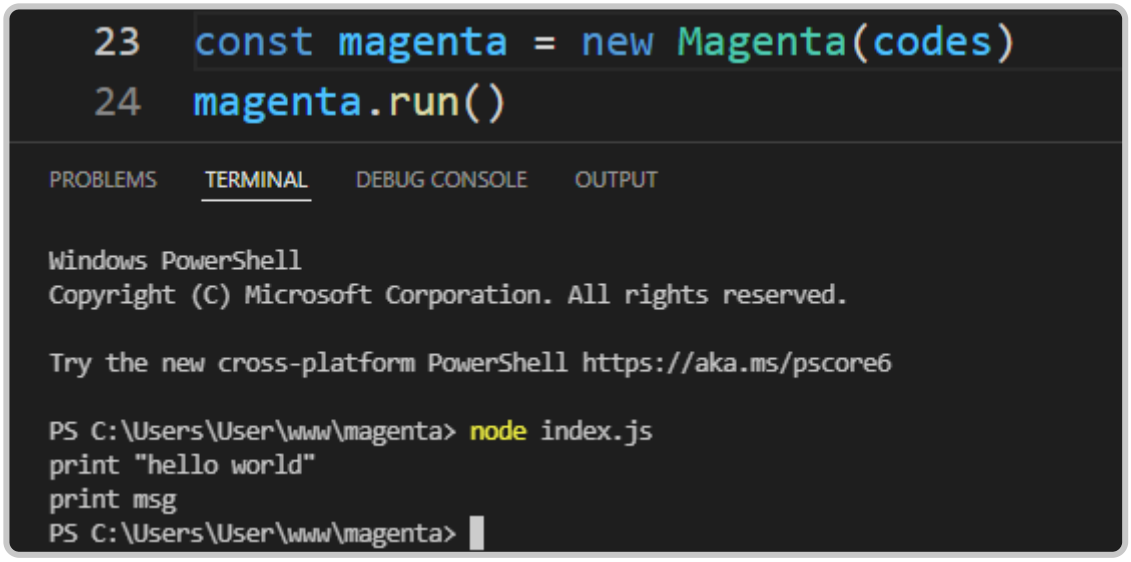
So let’s create a `parse` method that loops through our tokens and see if we have a valid syntax formed. If so, it will take necessary actions.

JavaScript

```
class Magenta {
  constructor(codes) {
    this.codes = codes
  }
  tokenize(){
    /* previous codes for tokenizer */
  }
```

```
    parse(tokens){
      const len = tokens.length
      let pos = 0
      while(pos < len) {
        const token = tokens[pos]
        // if token is a print keyword
        if(token.type === "keyword" && token.value === "print") {
          // if the next token doesn't exist
          if(!tokens[pos + 1]) {
            return console.log("Unexpected end of line, expected string")
          }
          // check if the next token is a string
          let isString = tokens[pos + 1].type === "string"
          // if the next token is not a string
          if(!isString) {
            return console.log(`Unexpected token ${tokens[pos + 1].type}, expected string`)
          }
          // if we reach this point, we have valid syntax
          // so we can print the string
          console.log('\x1b[35m%s\x1b[0m', tokens[pos + 1].value)
          // we add 2 because we also check the token after print keyword
          pos += 2
        } else{ // if we didn't match any rules
          return console.log(`Unexpected token ${token.type}`)
        }
      }
    }
    run(){
      const {tokens, error} = this.tokenize()
      if(error){
        console.log(error)
        return
      }
      this.parse(tokens)
    }
  }
```

And would you look at that — we already have a working language!



Okay but having codes in a string variable is not that fun. So lets put our **Magenta** codes in a file called `code.m`. That way we can keep our magenta codes separate from the compiler logic. We are using `.m` as file extension to indicate that this file contains code for our language.

Let’s read the code from that file:

```JavaScript
// importing file system module
const fs = require('fs')
//importing path module for convenient path joining
const path = require('path')
class Magenta{
  constructor(codes){
    this.codes = codes
  }
  tokenize(){
    /* previous codes for tokenizer */
 }
  parse(tokens){
    /* previous codes for parse method */
 }
  run(){
    /* previous codes for run method */
  }
}

// Reading code.m file
// Some text editors use \r\n for new line instead of \n, so we are removing \r
const codes = fs.readFileSync(path.join(__dirname, 'code.m'), 'utf8').toString().replace(/\r/g, &amp;quot;
const magenta = new Magenta(codes)
magenta.run()
```

(#aa-go-create-a-programming-language) Go create a programming language!

And with that, we have successfully created a tiny Programming Language from scratch. See, a programming language can be as simple as something that accomplishes one specific thing. Sure, it’s unlikely that a language like Magenta here will ever be useful enough to be part of a popular framework or anything, but now you see what it takes to make one.

The sky is really the limit. If you want dive in a little deeper, try following along with this video I made going over a more advanced example. This is video I have also shown hoe you can add variables to your language also.