Writing a Logo Interpreter

Alabhya Jindal

Contents

Welcome

Hi! Welcome!

We are going to write a interpreter in this book.

We'll implement a language called Logo. Logo is a visual language and can be used to draw shapes on a canvas.

To do this, we'll work with 4 languages - JavaScript, HTML, CSS and SVG. No external libraries. No regular expressions.

Let's start!

Designing the Playground

Create a folder on your computer. Call it `logo`. Now open this folder in your preferred text editor.

Create a `index.html` file:

```
<!-- index.html -->

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>The Logo Programming Language</title>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <div class="app">
      <aside>
        <h1>The Logo Programming Langauge</h1>
        <p class="description">
          Logo is a visual programming language. Type commands into the input
          and experiment. Draw shapes, have fun.
        </p>
        <section class="commands">
          <div>
            <p>
              <strong>fd &lt;steps&gt;</strong><span>,</span>
              <strong>bk &lt;steps&gt; </strong>
            </p>
            <p>forward, backward</p>
            <p><em>fd 10</em></p>
          </div>
          <div>
            <p>
              <strong>rt &lt;degrees&gt;</strong><span>,</span>
              <strong>lt &lt;degrees&gt; </strong>
            </p>
            <p>right turn, left turn</p>
            <p><em>rt 90</em></p>
          </div>
          <div>
            <strong>repeat &lt;count&gt; [commands]</strong>
            <p>repeat commands</p>
            <p><em>repeat 36 [lt 10 pu fd 1 pd repeat 120 [fd 4 rt 3]]</em></p>
          </div>
          <div>
```

```
            <p>
              <strong>pu</strong><span>,</span>
              <strong>pd</strong>
            </p>
            <p>pen up, pen down</p>
          </div>
          <div>
            <p>
              <strong>hd</strong><span>,</span>
              <strong>st</strong>
            </p>
            <p>hide turtle, show turtle</p>
          </div>
          <div>
            <strong>ct</strong>
            <p>center turtle</p>
          </div>
          <div>
            <strong>cs</strong>
            <p>clear screen</p>
          </div>
        </section>
      </aside>
      <main>
        <svg width="400" height="400"></svg>
        <form>
          <input
            type="text"
            placeholder="Enter here"
            autofocus
            autocomplete="off"
          />
          <button>Go</button>
        </form>
      </main>
    </div>
  </body>
</html>
```

Create a `style.css` file.

```
/*style.css*/

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  background-color: #e5e7eb;
  font-family: Inter, Roboto, 'Helvetica Neue', 'Arial Nova', 'Nimbus Sans',
    Arial, sans-serif;
}

.app {
  margin: 2rem auto;
```

```
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  gap: 3rem;
  padding: 1rem;
}

@media (min-width: 768px) {
  .app {
    flex-direction: row;
    align-items: flex-start;
  }
}

aside {
  max-width: 400px;
}

@media (min-width: 768px) {
  aside {
    width: 20rem;
  }
}

h1 {
  font-size: 1.125rem;
  font-weight: bold;
  text-transform: uppercase;
}

.description {
  margin-top: 1rem;
  font-size: 0.875rem;
}

.commands {
  margin-top: 1.5rem;
  display: flex;
  flex-direction: column;
  gap: 1.15rem;
  font-size: 0.875rem;
}

main {
  width: 400px;
  display: flex;
  flex-direction: column;
}

svg {
  border: 2px solid #6b7280;
  background-color: white;
  border-radius: 2px;
}
```

```css
form {
  margin-top: 1.5rem;
  display: flex;
  gap: 0.5rem;
}

input {
  flex-grow: 1;
  border: 1px solid #6b7280;
  border-radius: 2px;
  padding: 0.4rem 0.6rem;
  height: 2rem;
  font-family: 'Fira Mono', monospace;
  font-weight: bold;
  outline: none;
}

input:focus {
  border-color: #374151;
}

button {
  outline: none;
  border: none;
  background-color: #3b82f6;
  font-weight: 700;
  color: white;
  border-radius: 2px;
  padding: 0.4rem 1.6rem;
  font-family: monospace;
  cursor: pointer;
}

button:hover {
  background-color: #2563eb;
}
```

Open the HTML file in your web browser.

You should see a beautiful looking website. There are some instructions on the left and a white empty square on the right with an input and a button below it - we'll call this the playground.

Now the bulk of our work is going to be spent writing the parser for the Logo language. And we could do that - we could start by writing the parser. But that's boring. Because we won't see anything on the screen or be able to play until later.

Instead, we'll start by drawing a cursor to the screen. And then write some functions that can move that cursor based on the user input. And come to the parser later.

I didn't want to write the parser for Logo. But turns you can't expect nested `repeat` commands to work by just splitting the text by a space.

Drawing the turtle

Create a `turtle.js` file. Link to it to your HTML file.

```
<!-- index.html -->

<!-- Add this before the body tag closes -->
<script src="turtle.js"></script>
```

```
// parser.js

class Turtle {
  constructor(x, y, direction) {
    this.center = { x, y }
    this.x = x
    this.y = y
    this.direction = direction
    this.pen = true
    this.color = '#000000'
    this.element = null
  }
}
```

We will use a class for the cursor. We're calling it Turtle. Here we have defined some values that we'll use later. Let's start by actually getting the cursor drawn on the screen.

```
// parser.js

// Add this to the top of the file
const container = document.querySelector('svg')

// Add this inside the Turtle class
init() {
  container.innerHTML = ''
  const turtleElement = document.createElementNS(
    'http://www.w3.org/2000/svg',
    'g'
  )
  turtleElement.innerHTML = `
    <path d="M 0 0 l 10 10 l -10 -25 l -10 25 z" fill="red" stroke="black"></path>`
  turtleElement.setAttribute(
    'transform',
    `translate(${this.x}, ${this.y}) rotate(${this.direction})`
  )
  container.appendChild(turtleElement)
  this.element = turtleElement
}
```

First we get a reference to the container. Which is a SVG element.

We then create the `init` method inside the Turtle class. Let's understand what this is doing.

It empties the contents of the container. It does this because we will call the `init` function later to clear the screen.

Next up, it creates a `turtleElement` which is a SVG path. The path is drawn by a series of commands in the `d` attribute. The path starts at the origin and then moves relative to it's position to draw a cursor. We then add the `transform` property which takes two commands. `translate` determines the turtle position and `rotate` determines the direction it's facing.

Finally, we append this newly created element to the container.

Create a `main.js` file. Add it to the HTML file.

```
<!-- index.html -->

<!-- Add this after the turtle script tag -->
<script src="main.js"></script>
```

```
// main.js

let turtle = new Turtle(200, 200, 0)
turtle.init()
```

We initialize a turtle with x and y position of 200 - which means the center of the container since our container is of height 400. We set the direction of the turtle to 0 which means the cursor points north on initialization.

Open the browser. You should see a red cursor. Well done!

Accepting input

```
// main.js

// Add the following after initializing turtle
const input = document.querySelector('input')
const form = document.querySelector('form')

form.addEventListener('submit', (e) => {
  e.preventDefault()
  main(input.value)
  input.value = ''
})

function main(text) {
  const [name, arg] = text.split(' ')
  console.log(name)
  console.log(arg)
}
```

We get a reference to the form and the input element. Attaching a submit listener on the form means we can enter commands by pressing the `Enter` key or clicking the 'Go' button.

`e.preventDefault()` prevents the page from reloading. We call the `main` function with the current input value. Finally we clear the input.

The `main` function splits the text it receives by a space character. This gives us an array of length 2. We destructure this output and assign it to the variables name and arg.

Open the browser. And the developer tools. Type in `fd 50`. Press Enter. You should see two statements in the console - `fd` and `50`. This is great!

Moving the turtle

Now we have a way to access the command and the argument specified by the user, let's write code that will execute the user's inputs.

```
// turtle.js

// Add this inside the Turtle class
fd(steps) {
  if (!steps) return

  const radians = (this.direction / 180) * Math.PI
  const x = steps * 1 * Math.sin(radians)
  const y = steps * -1 * Math.cos(radians)
  this.x += x
  this.y += y
  this.element.setAttribute(
    'transform',
    `translate(${this.x}, ${this.y}) rotate(${this.direction})`
  )
  if (this.pen) {
    const path = document.createElementNS(
      'http://www.w3.org/2000/svg',
      'path'
    )
    path.setAttribute('d', `M ${this.x - x} ${this.y - y} l ${x} ${y}`)
    path.setAttribute('stroke', this.color)
    path.setAttribute('fill', 'none')
    container.appendChild(path)
  }
}
```

Let's unpack this.

We return early if the parameter is incorrect. We do some maths to figure out the new position of the turtle. We then set the element's attributes to the newly calculated x and y. Direction remains the same as it was before.

You might think that we should only set the x and y, and not direction since that's not changing. But that will remove the direction property and we'll lose the direction in which the turtle was originally facing. Remember that every time we make modification to the `transform` value - we are overwriting it with the new values.

If the pen property is true, then we draw a line from the old turtle position to the new one. We set the color of the path to the pen color.

```
// main.js

// Modify the main function to the following
```

```
function main(text) {
  const [name, arg] = text.split(' ')
  if (name === 'fd') {
    turtle.fd(arg)
  }
}
```

Type `fd 100` in the input field. Press Enter. You should see the cursor move!

Rotating the turtle

We'll now rotate the cursor.

```
// turtle.js

// Add this inside the Turtle class
rt(degrees) {
  if (!degrees) return

  this.direction += degrees
  this.element.setAttribute(
    'transform',
    `translate(${this.x}, ${this.y}) rotate(${this.direction})`
  )
}
```

This is simpler than the `fd` method. We add the given direction to the existing one. And we use it to update the `transform` property.

We have now written two methods for the Turtle class. We already have everything we need to get these 4 commands working.

1. fd - move forward
2. bk - move backward
3. rt - right turn
4. lt - left turn

Move backward is `fd` called with negative steps. Left turn is `rt` called with negative degrees.

We could add more if statements to map the user input to these commands. But we're smarter than that. No, I'm not talking about a switch statement.

```
// main.js

// Modify the main function to the following
function main(text) {
  const [name, arg] = text.split(' ')
  commandsMap[name](parseInt(arg))
}

// Add this to the end of the file
const commandsMap = {
  fd: (steps) => turtle.fd(steps),
  bk: (steps) => turtle.fd(-steps),
  rt: (degree) => turtle.rt(degree),
  lt: (degree) => turtle.rt(-degree),
  pd: () => (turtle.pen = true),
  pu: () => (turtle.pen = false),
  ct: () => turtle.ct(),
```

```
  cs: () => turtle.init(),
  ht: () => turtle.ht(),
  st: () => turtle.st(),
}
```

We can use this object whose values are functions we want to call. It has all the functions we'll implement.

We can then call our functions as shown in `main`. We convert the argument string to a number using `parseInt` before calling the function.

`fd`, `bk`, `rt`, `lt`, `pd`, `pu` and `cs` should all work as expected. Try it out! Try drawing a square. Can you draw two parallel lines with a space in between and nothing else?

Implementing hide, show and center turtle

Let's complete the Turtle class. We have 3 methods left to implement. `ct` to center turtle, `ht` to hide turtle and `st` to show turtle.

```
// turtle.js

// Add the following inside the Turtle class
ct() {
  this.x = this.center.x
  this.y = this.center.y
  this.element.setAttribute(
    'transform',
    `translate(${this.x}, ${this.y}) rotate(${this.direction})`
  )
}

ht() {
  this.element.style.visibility = 'hidden'
}

st() {
  this.element.style.visibility = 'visible'
}
```

`ct` works be setting the x and y position to center.x and center.y where were initialized to 200 when we initialized the turtle.

`ht` and `st` work by toggling CSS styles which hide and show the element.

We have now completed the Turtle class. All the Logo commands should now work! Amazing!

Overview of Repeat

Let's now turn our attention to the parser. Writing a parser will allow us to use `repeat` commands in Logo. This means using repeat to repeat multiple commands but also having nested repeat commands.

Before we start writing code, let's talk about what a parser is. And more importantly what our parser will do.

A parser is a program that takes in a string representation of a program and then returns a data structure that represents the program in a different manner. This data structure can be anything. The important thing is that the returned structure should make it easy to run the program that was given as a string.

We will use the input below to implement the parser together.

```
repeat 36 [lt 10 pu fd 1 pd repeat 120 [fd 4 rt 3]]
```

This is a nested repeat command. Our goal when we write this parser is to convert the above string to the following data structure.

```
[
  {
    "name": "repeat",
    "arg": 36,
    "commands": [
      {
        "name": "lt",
        "arg": 10
      },
      {
        "name": "pu"
      },
      {
        "name": "fd",
        "arg": 1
      },
      {
        "name": "pd"
      },
      {
        "name": "repeat",
        "arg": 120,
        "commands": [
          {
            "name": "fd",
            "arg": 4
          },
          {
            "name": "rt",
```

```
            "arg": 3
          }
        ]
      }
    ]
  }
]
```

Take a moment to read through the JSON structure. It is meant to represent the string in a different manner. The conveyed information is the same, but this structure will allow us to execute the program.

The structure is an array of objects. Each object has:

1. A name property
2. An optional arg property
3. A commands property if name is repeat

Notice the objects in the commands array have the same type as the ones in the top level.

Words from string

Create a `parser.js` file. Add it to your HTML file.

```
<!-- index.html -->

<!-- Add this before the main script tag -->
<script src="parser.js"></script>
```

```
// parser.js

// Add to the top of the file
function getWords(text) {
  const temp = text.split(' ')
  let words = []

  for (let t of temp) {
    if (t.includes('[')) {
      const count = t.split('[').length - 1
      for (let i = 0; i < count; i++) {
        words.push('[')
      }
      words.push(t.replaceAll('[', ''))
    } else if (t.includes(']')) {
      words.push(t.replaceAll(']', ''))
      const count = t.split(']').length - 1
      for (let i = 0; i < count; i++) {
        words.push(']')
      }
    } else words.push(t)
  }

  return words
}
```

That looks cryptic as hell. Let's understand what's happening.

First we split the text by spaces. This gives us a great start already as a lot of the program is neatly segregated into words. But it doesn't handle square brackets. Which are still attached to the words.

To fix this, we iterate over the newly split array. We check if a word includes an opening or a closing brackets. If it does:

1. We check how many brackets
2. We insert the the bracket that many times in a new array
3. We insert the word itself

The order of 2 and 3 switches depending on whether it's a closing bracket or an opening bracket. If the word doesn't have any brackets we just add the word itself.

The return value of `getWords` is the argument of the upcoming `parse` function.

If we call this function with our repeat command, we'll get the following:

```
[
  "repeat",
  "36",
  "[",
  "lt",
  "10",
  "pu",
  "fd",
  "1",
  "pd",
  "repeat",
  "120",
  "[",
  "fd",
  "4",
  "rt",
  "3",
  "]",
  "]"
]
```

It's an array of strings. Where each string is a non-space character from the original.

Parsing non repeats

```
// parser.js

// Add the following after the getWords function
function parse(words) {
  const argCommands = ['fd', 'bk', 'rt', 'lt']
  const nonArgCommands = ['pu', 'pd', 'ct', 'cs', 'ht', 'st']
  let index = -1

  return parseExpression(words)

  function remainingTokens() {
    return index < words.length
  }

  function nextToken() {
    if (remainingTokens()) return words[++index]
  }

  function parseExpression() {
    let commands = []

    while (remainingTokens()) {
      let token = nextToken()

      if (argCommands.includes(token)) {
        let cmd = {
          name: token,
          arg: parseInt(nextToken()),
        }
        commands.push(cmd)
      } else if (nonArgCommands.includes(token)) {
        let cmd = {
          name: token,
        }
        commands.push(cmd)
      }
    }
    return commands
  }
}
```

Let's understand what this is doing.

There are two arrays which store a list of commands. One array for commands that require an argument and one which doesn't. This comes in useful further down. We also declare an index, initialized to -1.

This is used to track the character that we're currently looking at.

We return a nested `parseExpression` function. This will come in handy when we implement repeat.

The `remainingTokens` function returns `true` if there are more than 0 characters left. This is used later for a while loop.

`nextToken` returns the next token. Which is as easy as returning the word at the index position + 1. `++index` increments index and returns the new value, which is then used to return a word.

Finally, `parseExpression` does the following:

1. Get the next token as long as there are tokens available
2. Push an object with name and an optional arg depending on the token type
3. Returns the commands

That's cool but are app is still in limbo. Because we can't run anything! Let's fix that.

```
// main.js

// Replace the main function to the following
function main(text) {
  const words = getWords(text)
  const tokens = parse(words)
  execute(tokens)
}

// Add after main function
function execute(tokens) {
  for (let token of tokens) {
    let { name, arg, commands } = token
    arg = parseInt(arg)
    commandsMap[name](arg)
  }
}
```

Now we are back where we were a few pages ago. All the Logo commands should work except repeat.

The `execute` function is very similar to what we were doing earlier in the `main` function. The difference is that it iterates over the tokens and executes each one.

Parsing repeat

Let's work on implementing the `repeat` command. The general idea is that the commands inside the square brackets are the same as the outside ones. That means we need to parse them the same way.

So we need to get the words within square brackets and then call parse on it again. This will involve recursion.

```
// parser.js

// Add this inside the parse function, after remainingTokens
function getRepeat() {
  nextToken()
  nextToken()

  let bracketsFound = 1
  const nest = []

  while (bracketsFound > 0) {
    if (words[index] == '[') bracketsFound++
    if (words[index] == ']') bracketsFound--

    nest.push(words[index])
    nextToken()
  }
  return nest
}

// Add this inside parseExpression after else if
else if (token == 'repeat') {
  let cmd = {
    name: token,
    arg: parseInt(nextToken()),
  }
  let toRepeat = getRepeat()
  cmd.commands = parse(toRepeat)
  commands.push(cmd)
}
```

`getRepeat` is responsible for getting the words inside the brackets for a repeat command. It does this by adding the next word to an array until it matches all opening and closing brackets. The while loop checking for the bracket count is important. We can't exit the while loop on the first occurrence of the closing square bracket character because a repeat command can have multiple repeats.

Further down, in `parseExpression` we get the repeat words, parse it and push it o the current token's commands property.

Executing

Our parser is now complete. The only thing left is to execute them!

```
// main.js

// Modify the execute function to the following
function execute(tokens) {
  for (let token of tokens) {
    let { name, arg, commands } = token
    arg = parseInt(arg)

    if (name == 'repeat') {
      for (let i = 0; i < arg; i++) {
        execute(commands)
      }
    } else {
      commandsMap[name](arg)
    }
  }
}
```

The change here is that `execute` recursively calls itself if it finds a repeat command. Else we execute the token.

Congratulations

We are done. Congratulations!

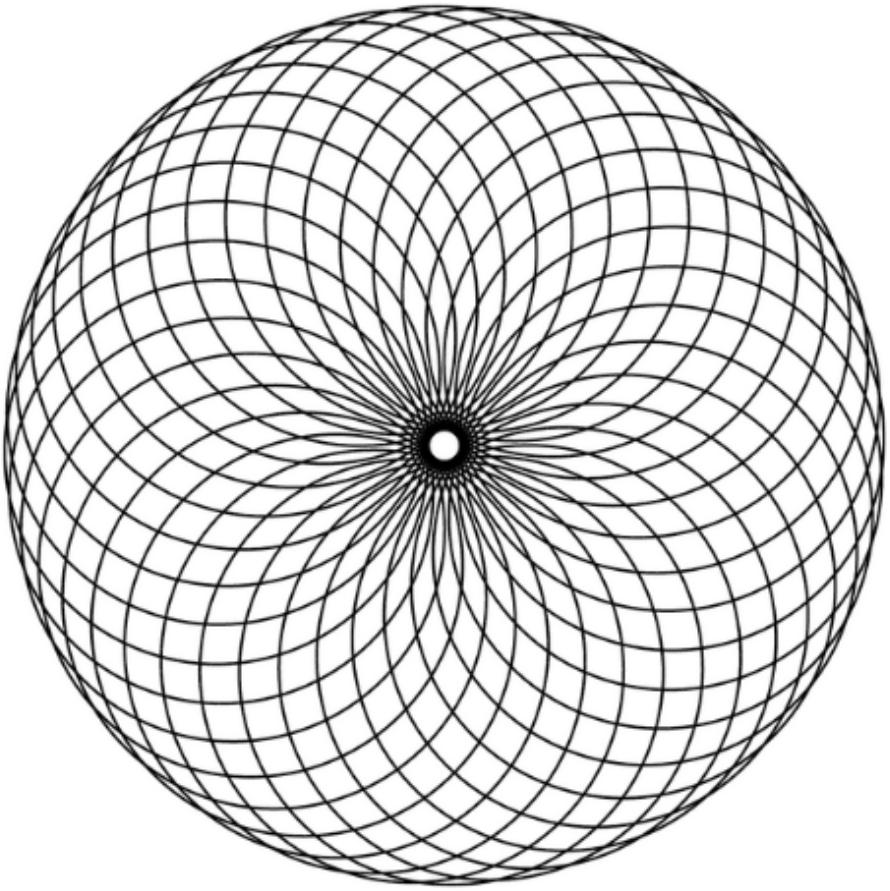

Figure 1: `repeat 36 [lt 10 pu fd 1 pd repeat 120 [fd 4 rt 3]]`