Title

So you're writing a website, right? And you're just thinking to yourself "Wow, javascript really sucks, I wish I could be using anything else right now." Well guess what, you can, and it's called webassembly, and I'm going to teach you all about it in tonights session.

What am I talking about?

So what will I be talking about?

- What is webassembly?
- Rust
  - Mandelbrot set
- Raw
  - for crazy people or you want to learn fundamentals
  - AoC day 4 part 1

What am I not talking about?

First, what I won't be talking about are...

- Batteries included frameworks
  - Compiles your code into webassembly
  - Abstract away webassembly into a technical detail
- I'm teaching webassembly, not a framework
- Super useful if you want to make a whole website in webassembly

How do websites work?

So first, how do websites work?

1. Put an address into your browser
2. Browser makes HTTP request
3. Returns an HTML file
   - Content of the webpage
   - Example on the right
   - Instructs the browser to download additional files
- CSS
  - How your website looks
- Javascript
  - Programming language
  - What the website should do
  - Ex: Making scrolling do something other than actually scrolling

What problems does Webassembly solve?

So where does webassembly fit into this? Before that I need to explain what problems webassembly attempts to solve.

- Javascript is cursed
  - You can compare strings and numbers and it might return true
  - Added a triple equals operator when people figured out that made no sense
  - No type checking
- Scripting language; Slow as heck
  - No rendering fractals

- Running non-web stuff on the web
  - FFMPEG
  - Your browser doesn't understand assembly

So for these reasons, developers invented Webassembly.

What is Webassembly?

So now, what is webassembly?

- Standardized machine code
  - Compilation target for other languages
  - You're not meant to code in it directly, but you totally can and we will
  - You can write code for the web in any language you like
  - Compiling desktop programs for the web
- Can only interface with javascript
  - Importing and exporting functions from/to javascript
- Can't send or receive complex data structures, only numbers
  - You need javascript to encode and decoded data directly in your webassembly module's memory
  - "Glue code"
    - We'll use Rust for our first demo and we'll see that it takes care of this for us.
- No direct access to DOM or standard library
  - You need to import from Javascript
  - Also taken care of by Rust

Demo 1!

So now we can start with the first demo!

- Mandelbrot set
  - I'm assuming that if you wanted to follow along you set everything up in accordance with the README on the github page.

First, we need to set up the HTML for our page.

HTML

index.html

```
<!doctype html>

<html>
  <head>
    <title>Mandelbrot</title>
  </head>

  <body>
    <canvas id="canvas" width="800" height="600"></canvas>
    <script type="module" src="sketch.js"></script>
  </body>
</html>
```

JS

Now lets write our Javascript code

I'm implementing this in a roundabout way

- JS asks WASM to render a frame
- WASM renders frame
- WASM asks JS to display the rendered frame
- JS asks WASM through a callback to copy the frame data into the canvas buffer
- More straightforward to have WASM return the rendered frame and have JS copy it into the buffer but I want to demo the cool JS interop

sketch.js

```
import init, { render_data } from "./pkg/mandelbrot.js";

(async () => {
  await init()

  showThingy(1);
})()
```

—

```
function showThingy(scale) {
  render_data(scale)

  requestAnimationFrame(() => showThingy(scale * 0.99))
}
```

—

```
export function set_canvas_data(callback) {
  let canvas = document.getElementById("canvas");
  let ctx = canvas.getContext("2d");

  let imageData = ctx.getImageData(0, 0, 800, 600);

  callback(imageData.data)

  ctx.putImageData(imageData, 0, 0);
}
```

Rust

Now we can write our Rust code that will actually generate the mandelbrot set.

Cargo.toml

```
[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = "0.2.74"
js-sys = "0.3.68"
```

—

lib.rs

```
fn mandelbrot_shade_at_pos(c: (f64, f64), count: usize) -> f64 {
    let mut z = (0., 0.);

    for i in 0..count {
        if z.0 * z.0 + z.1 * z.1 > 4. {
            return i as f64 / count as f64;
        }
```

```
        z = (z.0 * z.0 - z.1 * z.1 + c.0, z.0 * z.1 * 2. + c.1);
    }

    return 1.;
}
```

—

```
#[wasm_bindgen]
pub fn render_data(scale: f64) {
    let mut data = vec![0; 800 * 600 * 4];

    let count = (10. * (1. / scale).ln()) as usize + 30;

    for x in 0..800 {
        for y in 0..600 {
            let shade = mandelbrot_shade_at_pos(
                (
                    (x as f64 - 400.) / 200. * scale - 1.5,
                    (y as f64 - 300.) / 200. * scale,
                ),
                count,
            );

            data[4 * (x + y * 800) + 0] = (shade * 256.) as u8;
            data[4 * (x + y * 800) + 1] = (shade * 256.) as u8;
            data[4 * (x + y * 800) + 2] = (shade * 256.) as u8;
            data[4 * (x + y * 800) + 3] = 255;
        }
    }

    set_canvas_data(&Closure::once(move |array: Uint8ClampedArray| {
        array.copy_from(&data)
    }));
}
```

—

```
#[wasm_bindgen(raw_module = "/sketch.js")]
extern "C" {
    fn set_canvas_data(s: &Closure<dyn FnMut(Uint8ClampedArray) -> ()>);
}
```

Compiling and running

Now we can actually run it!

```
wasm-pack build --target web
python3 -m http.server
```

WAT

Now observe this meme, it's very important...

Now I know what you're thinking...

WAT?

And you'd be right!

How does it work?

So how does it work?

Sections

- Global scope stuff
  - Defining memory
  - Defining globals
  - Importing functions from JS
  - Defining functions
  - etc.
- The example on the right will be in the demo and I'll explain everything.

Instructions

- Small instruction set
- Mix of stack and register based
  - It has a stack
  - But you can define locals at the top of the function
  - The size of the stack at any given point is known at parse time
    - Can be implemented with a fixed array or with registers
- Control flow constructs are high level
  - If/else
  - Loops
  - Blocks
  - No goto
- Will make sense when we get into coding

Demo

So now for the demo.

- Might get boring or hard to follow
- Feel free to leave or yell at me if you have a question

For this, we'll be doing advent of code, day four, part one.

Basically, we have this table, and we want to find how many of the numbers on the left side are on the right side

- Do the first row

Then each row is worth zero points for zero matches, one for one match, and doubles for each match after that

- That row has four matches so it'll have a score of eight
  - One for the first match, then doubled three times for the three next ones.

Module setup

```
(module
    (memory (import "js" "mem") 1)
    (global $end_of_input (import "js" "endOfInput") i32)

    (import "console" "log" (func $log (param i32) (result i32)))
    (import "js" "parseNum" (func $parse_num (param i32) (result i32)))

    (func (export "solve") (result i32)
    )
```

```
    (func $solve_line (param $start_of_line i32) (param $start_of_scratch i32) (result
i64)
    )

    (func $check_num (param $num i32) (param $winning_numbers_buf_addr i32) (param
$buf_end_addr i32) (result i32)
    )
)
```

`check_num`

First let's implement `check_num`, which will check if a number is a winning number.

We have...

- Winning numbers in a block of memory
- A number we want to check if it's a winning number
- `$num` is the number we want to check
- `$winning_numbers_buf_addr` is the address of the start of the list of winning numbers
- `$buf_end_addr` is the address after the end of the winning numbers buffer
- Return `1` if it's a winning number, `0` if it isn't.
  - In wasm, $1 = \text{true}$, $0 = \text{false}$

```
(loop $check_loop
    ;; Load the number in memory and compare it with the number given
    local.get $winning_numbers_buf_addr
    i32.load
    local.get $num
    i32.ne

    (if (then
        ;; Go to the next number
        local.get $winning_numbers_buf_addr
        i32.const 4
        i32.add
        local.tee $winning_numbers_buf_addr

        ;; Continue if we haven't hit the end of the buffer yet
        local.get $buf_end_addr
        i32.ne
        br_if $check_loop
    ) (else
        ;; Return if they are equal
        i32.const 1
        return
    ))
)

i32.const 0 ;; Return zero if we went through the whole loop and didn't find a match
```

`solve_line`

- Will involve a lot of parsing, we'll go character by character

- `$start_of_line` will be an address to the start of the line, but we'll use it as the current character position
  - When we write the javascript, we'll have it write the string into our module's memory
- `$start_of_scratch` will be an address to random scratch data
- We'll return an i64 where the first four bytes are the card's score and the last four bytes are the position of the next line

```
;; I'm using the scratch space to store the winning numbers
(local $winning_number_pos i32)
(local $winning_number_count i32)

;; Skip to the first colon
(loop $skip_to_colon
    ;; Compare the character
    local.get $start_of_line
    i32.load8_u
    i32.const 58 ;; :
    i32.ne

    ;; Go to next char
    local.get $start_of_line
    i32.const 1
    i32.add
    local.set $start_of_line

    br_if $skip_to_colon
)

;; Skip the space
local.get $start_of_line
i32.const 1
i32.add
local.set $start_of_line

;; Load the winning numbers
(loop $load_nums_loop
    ;; Find the address to write the number
    local.get $start_of_scratch
    local.get $winning_number_pos
    i32.add

    ;; Parse the winning number
    local.get $start_of_line
    call $parse_num

    ;; Store it
    i32.store

    ;; Set the memory address for the next winning number
    local.get $winning_number_pos
    i32.const 4
    i32.add
    local.set $winning_number_pos

    ;; Skip to the next number
    local.get $start_of_line
```

```
    i32.const 3
    i32.add
    local.set $start_of_line

    ;; Compare the character
    local.get $start_of_line
    i32.load8_u
    i32.const 124 ;; Pipe
    i32.ne

    br_if $load_nums_loop
)

;; Skip the pipe (land on the space so as we keep adding three we'll land on the
newline instead of skipping over it to the next card)
local.get $start_of_line
i32.const 1
i32.add
local.set $start_of_line

;; Check the card numbers
(loop $check_nums_loop
    ;; Parse the number
    local.get $start_of_line
    call $parse_num

    ;; Check the number and maybe increment $winning_number_count
    local.get $start_of_scratch

    local.get $start_of_scratch
    local.get $winning_number_pos
    i32.add ;; Add to find the end of the buffer

    ;; The stack looks like [parsed num, start of winning numbers, end of winning
numbers]
    call $check_num

    ;; Increment the winning number count if it is
    local.get $winning_number_count
    i32.add
    local.set $winning_number_count

    ;; Skip to the next number
    local.get $start_of_line
    i32.const 3
    i32.add
    local.set $start_of_line

    ;; Compare the character
    local.get $start_of_line
    i32.load8_u
    i32.const 10 ;; \n
    i32.ne

    br_if $check_nums_loop
)
```

```
;; Return the result + funny exponent thing
i32.const 1

local.get $winning_number_count
i32.const 1
i32.sub

;; Shift 1 by one minus the amount of winning numbers (if there are zero, it'll shift
-1, right shifting out the one)
i32.shl

i64.extend_i32_u

;; Put the new line start in there
local.get $start_of_line
i64.extend_i32_u
i64.const 32
i64.shl

i64.or
```

`index.html`

Now we can finish it off by writing the JS code to interface with it.

```
// Create the memory (one page)
const memory = new WebAssembly.Memory({ initial: 1, maximum: 1 });

// Get the memory buffer
const array = new Uint8Array(memory.buffer);

// Write the input into memory
for (let i = 0; i < input.length; i++) {
  array[i] = input.charCodeAt(i);
}

// Load the module
let module = await WebAssembly.instantiateStreaming(
  fetch("solution.wasm"),
  {
    js: {
      mem: memory,
      endOfInput: input.length,
      parseNum: (pos) => parseInt(input.slice(pos, pos + 3)),
    },
    console: { log: (n) => (console.log(n), n) },
  },
);

document.getElementById("answer").innerText =
  module.instance.exports.solve();
```

Test it

```
wat2wasm solution.wat
python3 -m http.server
```