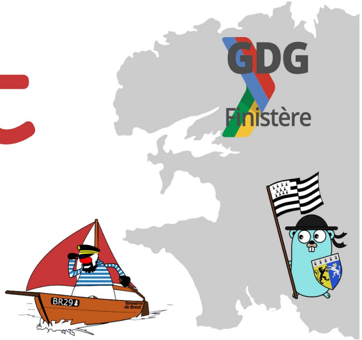
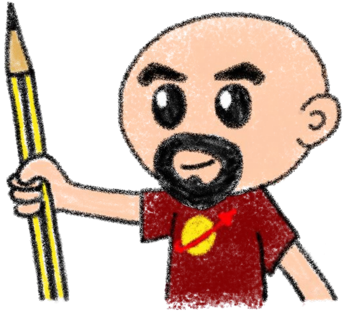
Finist Devs



WebAssembly for Developers (web... or not)

Horacio Gonzalez

@LostInBrittany

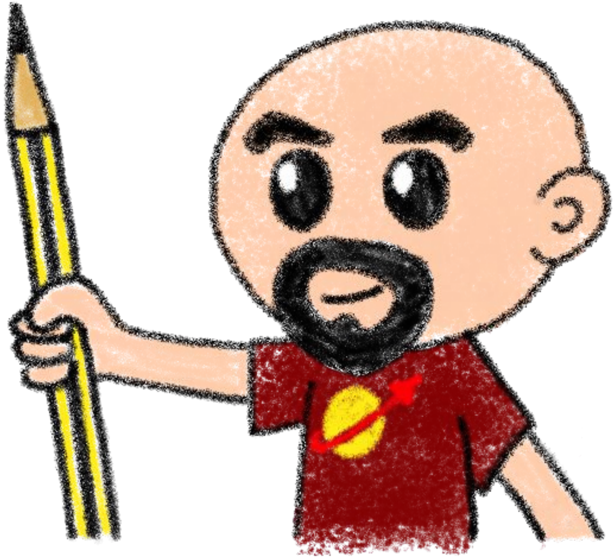


Horacio Gonzalez

@LostInBrittany

OVH
Team DevRel

Spaniard lost in Brittany, developer, dreamer and all-around geek



DevFest du Bout du Monde

Finist Devs

GDG Finistère

Google Developers Experts 2019

Web Technologies GDE

Did I say WebAssembly?

WASM for the friends...

WebAssembly, what's that?

Can I code webapps in Rust?

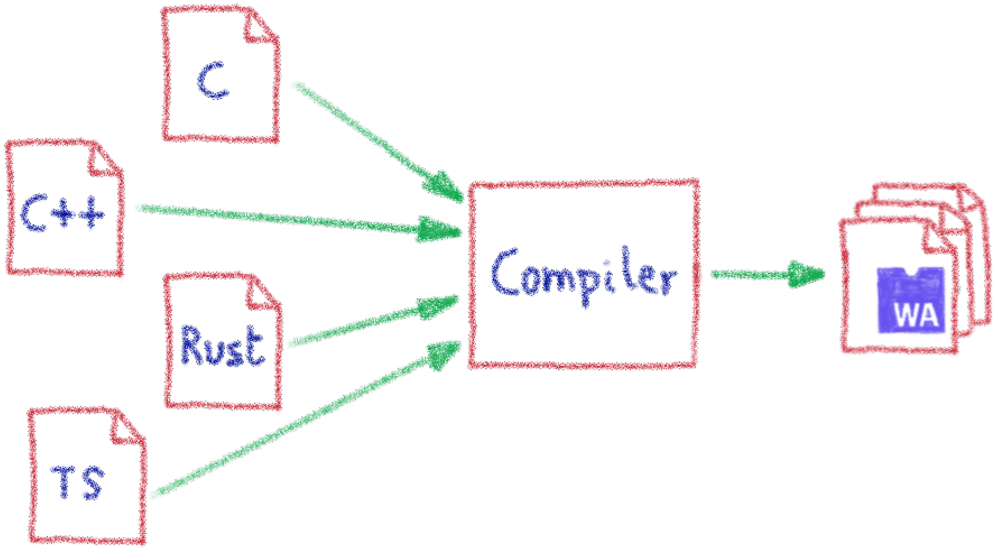
What's WASM ?

Does it replace JS ?

Is HTML/CSS/JS stack obsolete?

Let's try to answer those (and other) questions...

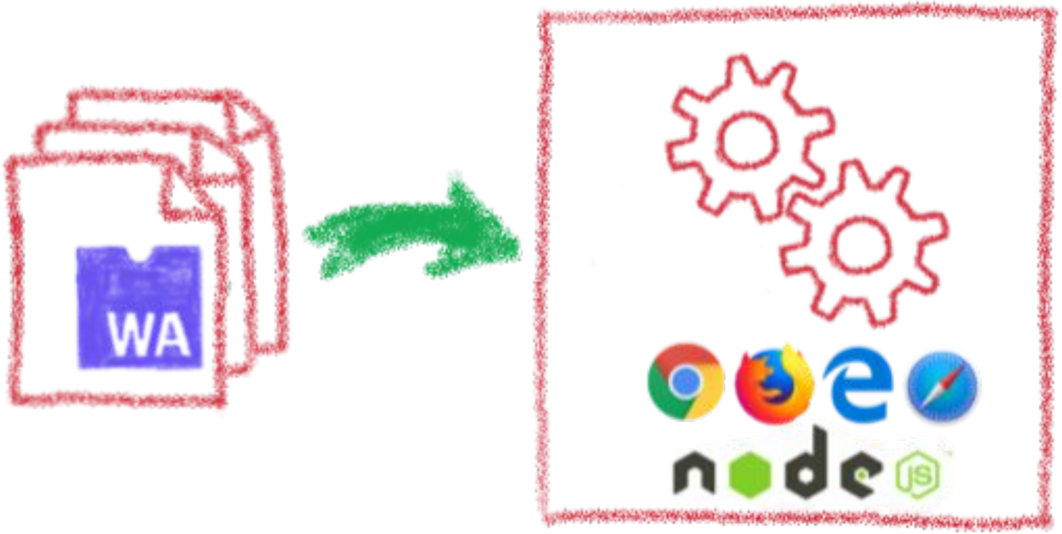
A low-level binary format for the web



Not a programming language

A compilation target

That runs on a stack-based virtual machine



A portable binary format that runs on all modern browsers...

but also on NodeJS!

With several key advantages

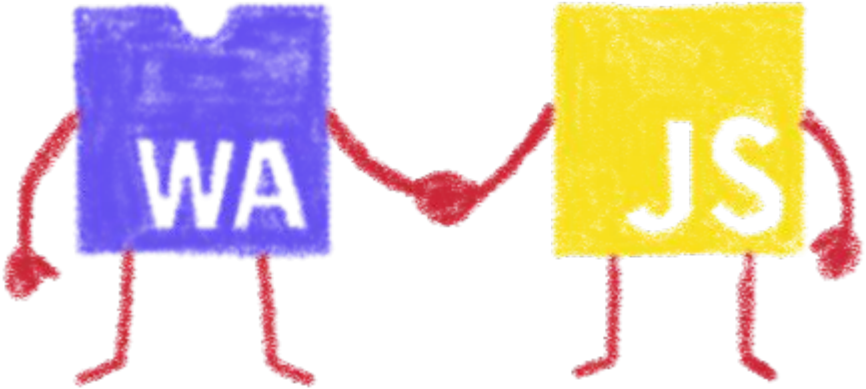
Fast & Efficient ⚡

🔒 Memory-safe & Sandboxed

Open & Deboggable 📄

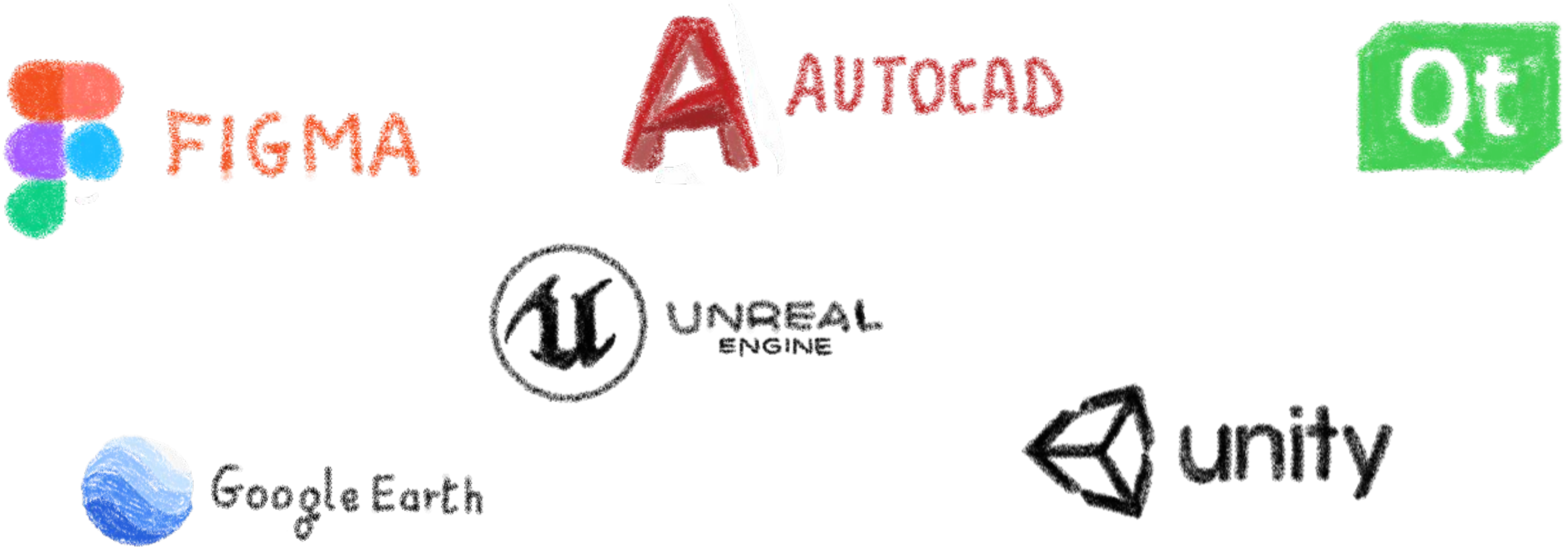
www Part of the Web Platform

But above all...



WebAssembly is not meant to replace JavaScript

Who is using WebAssembly today?



And many more others...

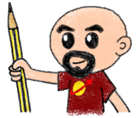
A bit of history

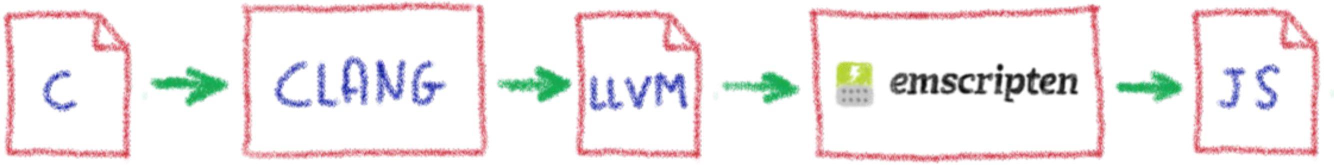
Remembering the past
to better understand the present

Executing other languages in the browser

A long story, with many failures...

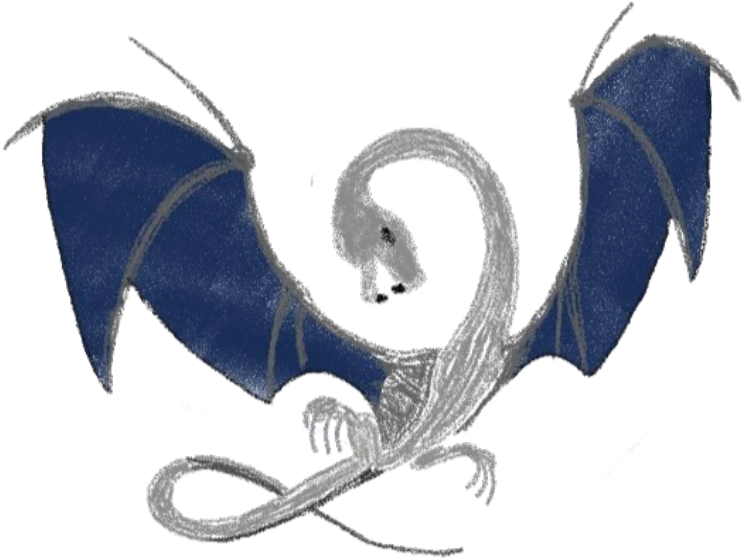
2012 - From C to JS: enter emscripten





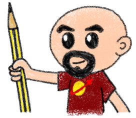
Passing by LLVM pivot

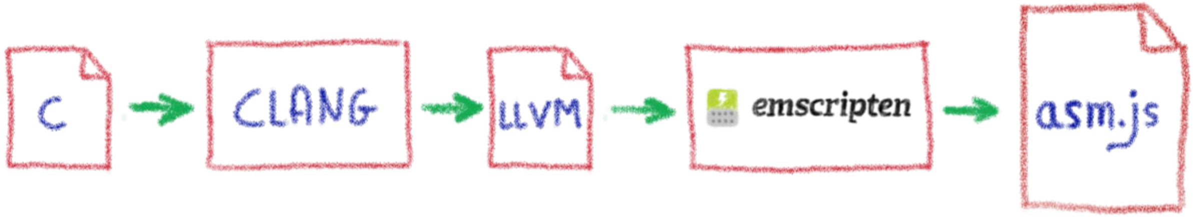
Wait, dude! What's LLVM?



A set of compiler and toolchain technologies

2013 - Generated JS is slow...





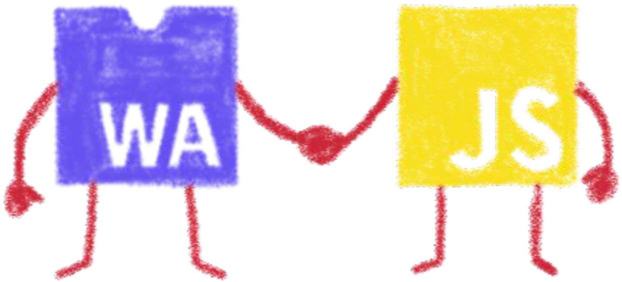
Let's use only a strict subset of JS: asm.js

Only features adapted to AOT optimization

WebAssembly project

moz://a
Google
Microsoft
W3C

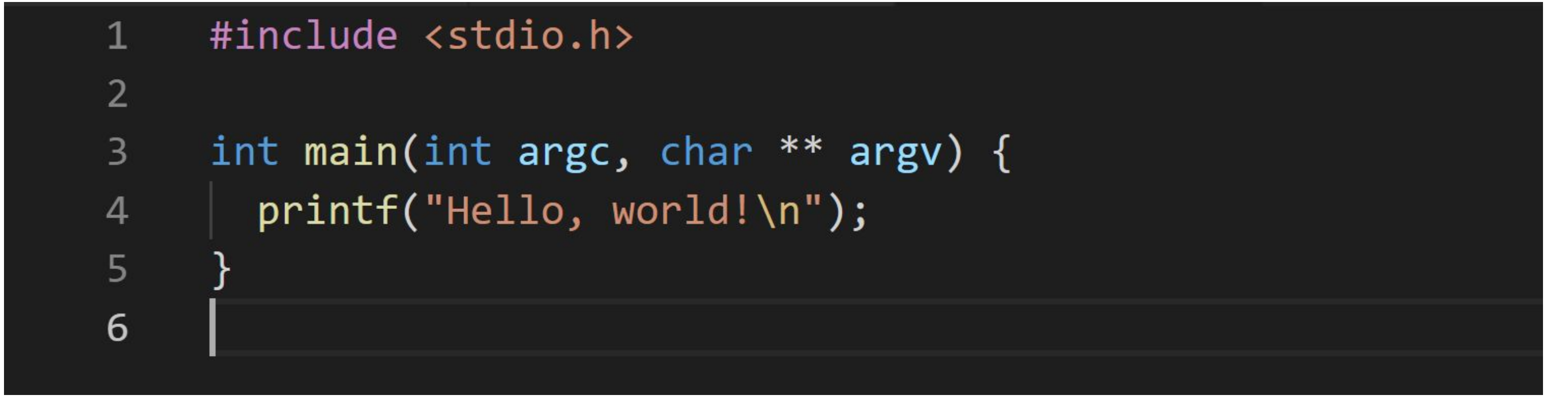
Joint effort



Hello W(ASM)orld

My first WebAssembly program

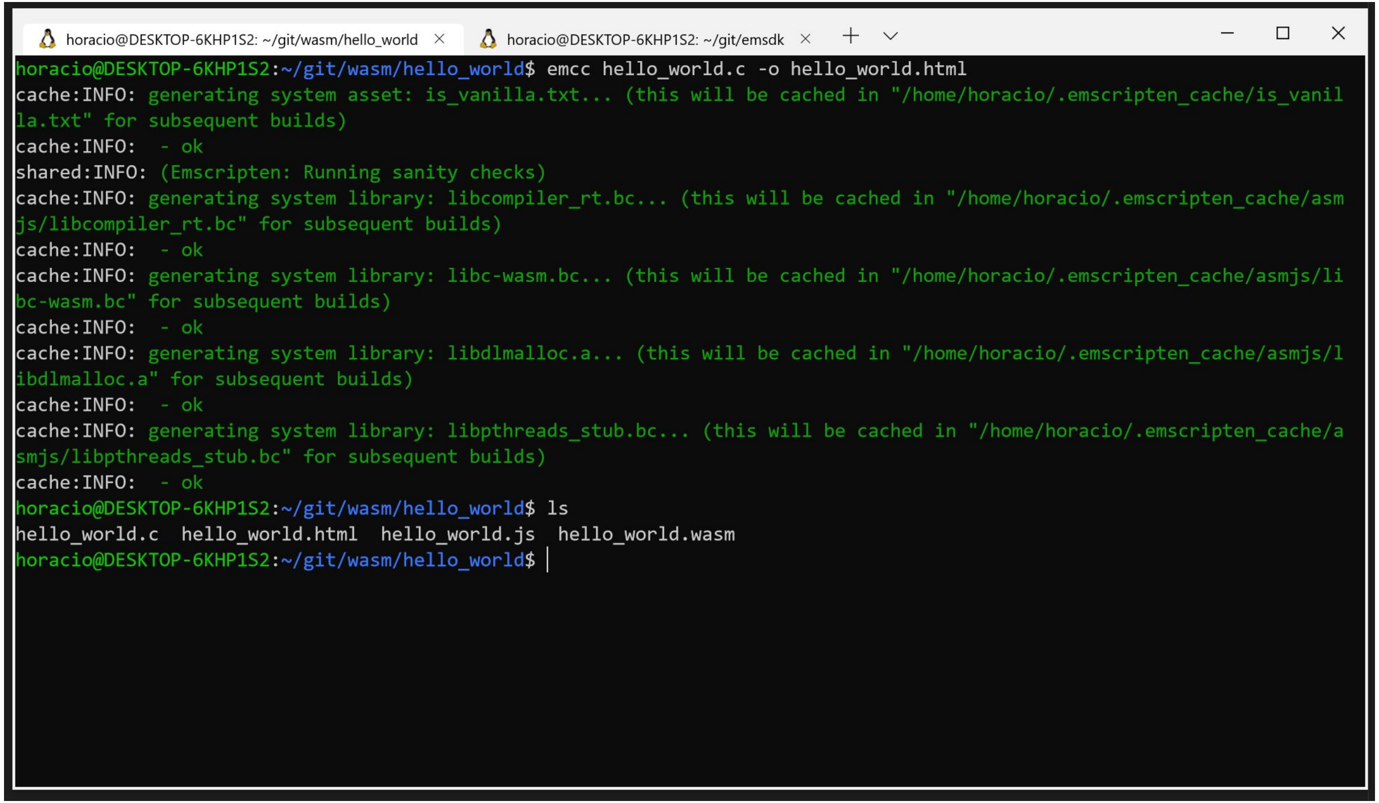
Do you remember your 101 C course?



```c
#include <stdio.h>

int main(int argc, char ** argv) {
  printf("Hello, world!\n");
}

```

A simple *HelloWorld* in C

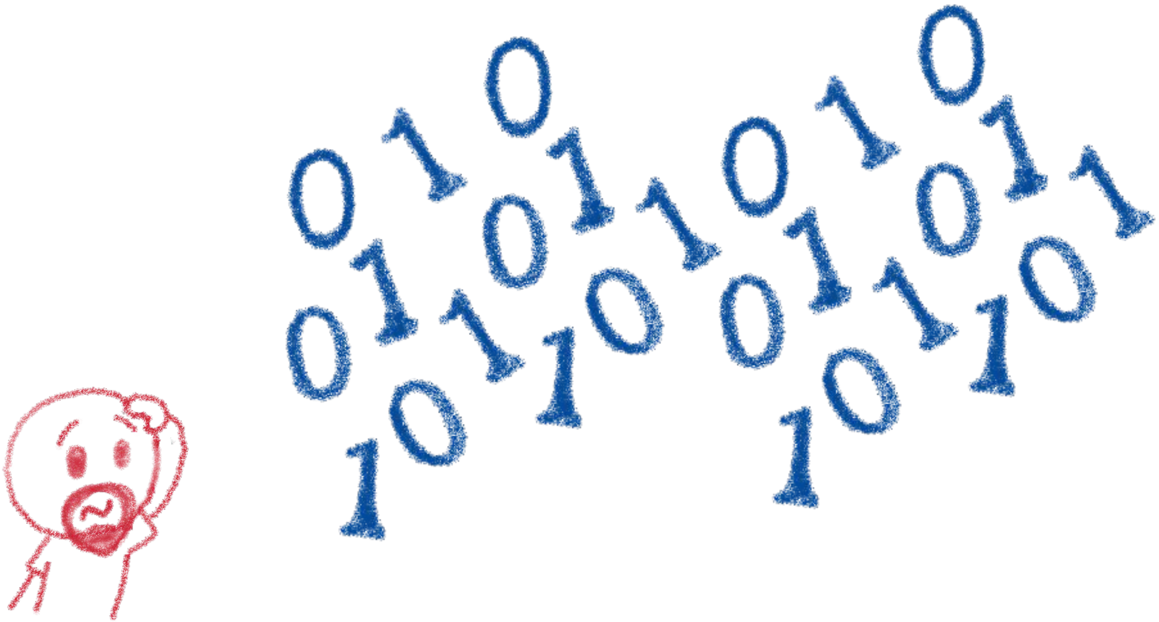
We compile it with emscripten



horacio@DESKTOP-6KHP1S2: ~/git/wasm/hello_world
horacio@DESKTOP-6KHP1S2: ~/git/emsdk

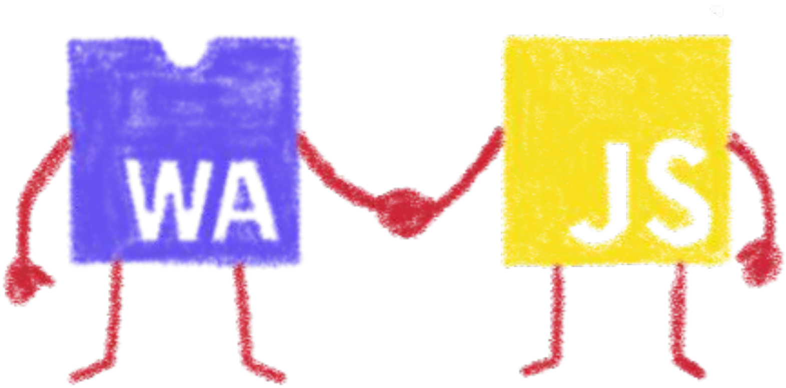
```
horacio@DESKTOP-6KHP1S2:~/git/wasm/hello_world$ emcc hello_world.c -o hello_world.html
cache:INFO: generating system asset: is_vanilla.txt... (this will be cached in "/home/horacio/.emscripten_cache/is_vanil
la.txt" for subsequent builds)
cache:INFO:  - ok
shared:INFO: (Emscripten: Running sanity checks)
cache:INFO: generating system library: libcompiler_rt.bc... (this will be cached in "/home/horacio/.emscripten_cache/asm
js/libcompiler_rt.bc" for subsequent builds)
cache:INFO:  - ok
cache:INFO: generating system library: libc-wasm.bc... (this will be cached in "/home/horacio/.emscripten_cache/asmjs/li
bc-wasm.bc" for subsequent builds)
cache:INFO:  - ok
cache:INFO: generating system library: libdlmalloc.a... (this will be cached in "/home/horacio/.emscripten_cache/asmjs/l
ibdlmalloc.a" for subsequent builds)
cache:INFO:  - ok
cache:INFO: generating system library: libpthreads_stub.bc... (this will be cached in "/home/horacio/.emscripten_cache/a
smjs/libpthreads_stub.bc" for subsequent builds)
cache:INFO:  - ok
horacio@DESKTOP-6KHP1S2:~/git/wasm/hello_world$ ls
hello_world.c  hello_world.html  hello_world.js  hello_world.wasm
horacio@DESKTOP-6KHP1S2:~/git/wasm/hello_world$ 
```

We get a .wasm file...



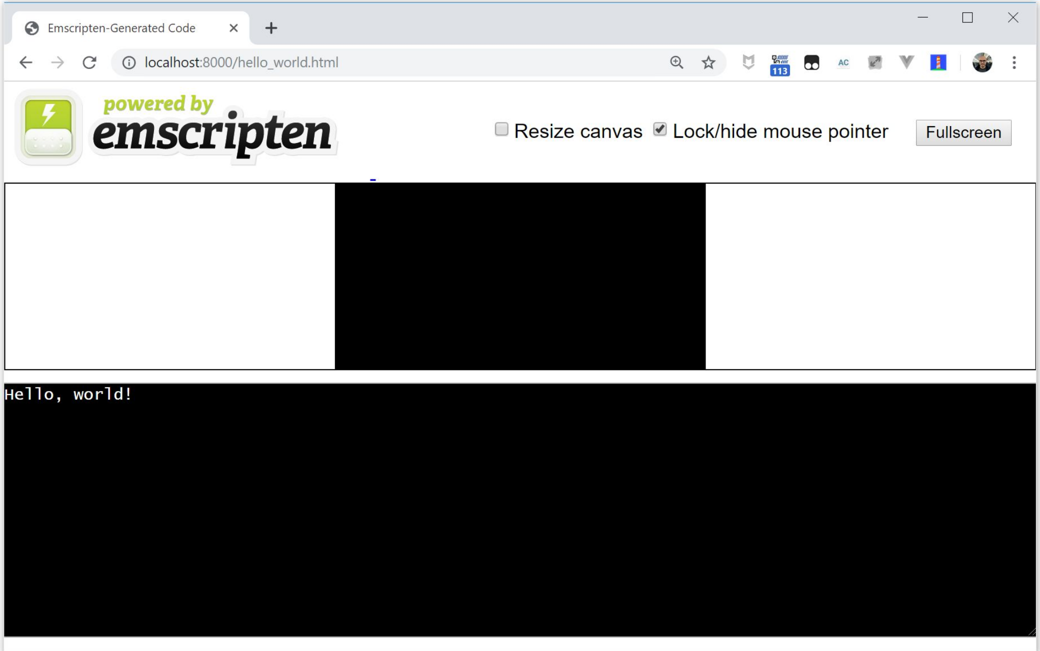
Binary file, in the binary WASM format

We also get a `.js` file...



Wrapping the WASM

And a .html file



Emscripten-Generated Code
localhost:8000/hello_world.html
powered by
emscripten
□ Resize canvas
☑ Lock/hide mouse pointer
Fullscreen

```
Hello, world!
```

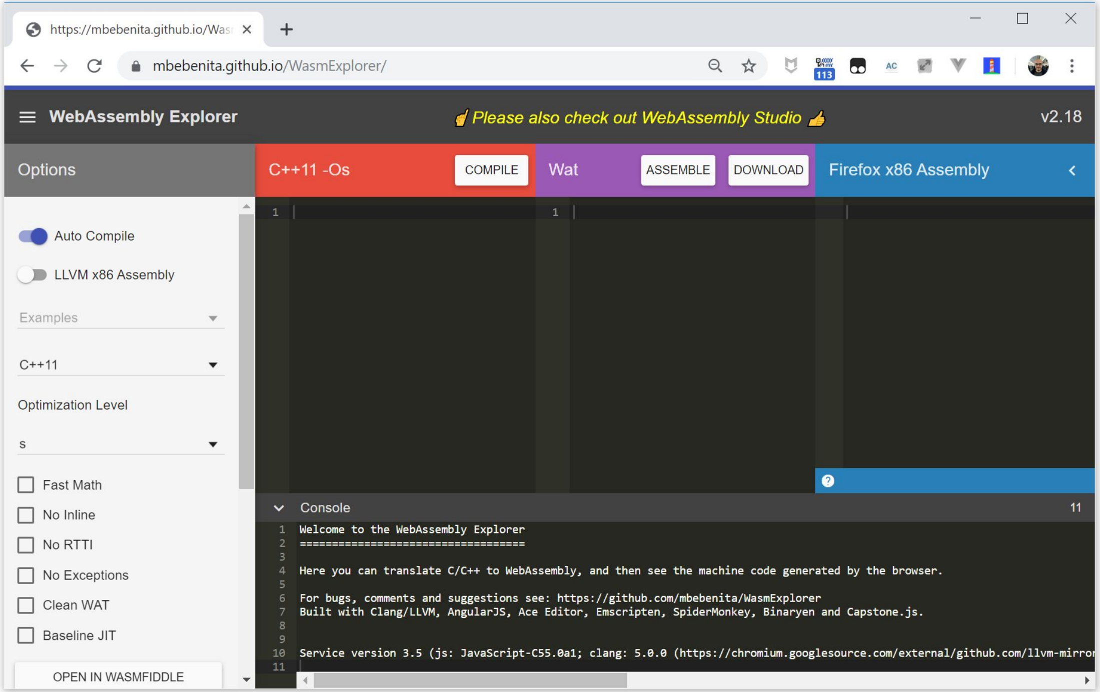
To quickly execute in the browser our WASM

And in a more Real World^{TM} case?

A simple process:

- Write or use existing code
  - In C, C++, Rust, Go, AssemblyScript...
- Compile
  - Get a binary `.wasm` file
- Include
  - The `.wasm` file into a project
- Instantiate
  - Async JavaScript compiling and instantiating the `.wasm` binary

I think I need a real example now



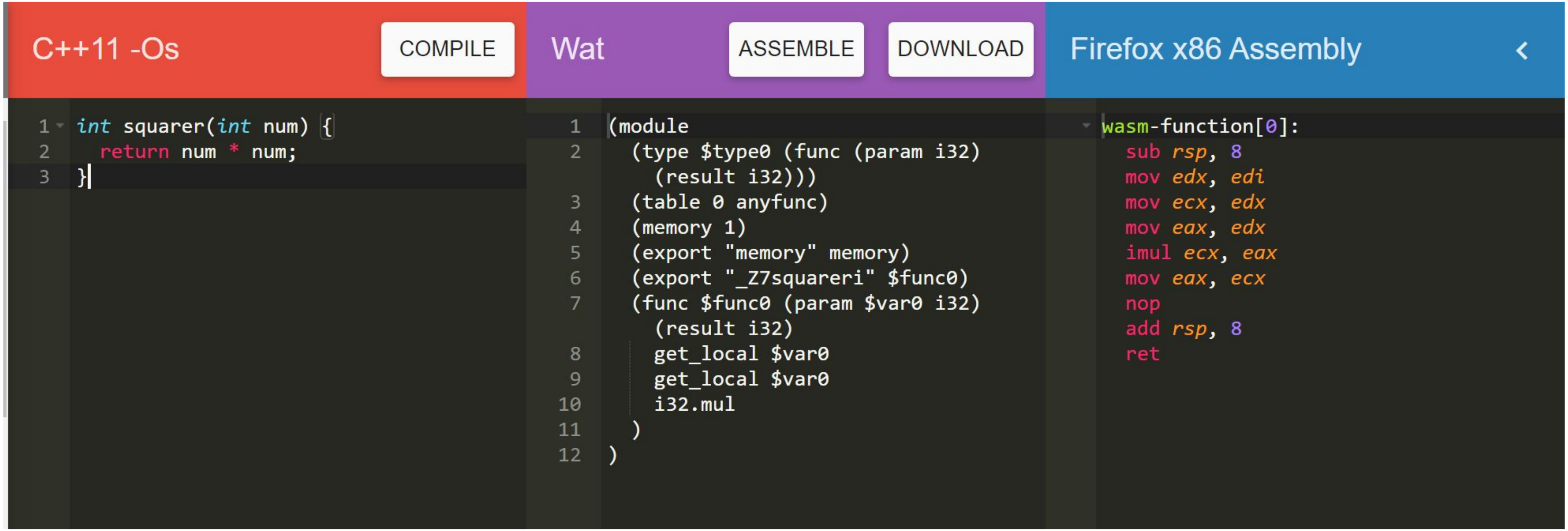
- ● Auto Compile
- ○ LLVM x86 Assembly
- C++11
- Optimization Level: s
- □ Fast Math
- □ No Inline
- □ No RTTI
- □ No Exceptions
- □ Clean WAT
- □ Baseline JIT

```
Welcome to the WebAssembly Explorer
==================================

Here you can translate C/C++ to WebAssembly, and then see the machine code generated by the browser.

For bugs, comments and suggestions see: https://github.com/mbebenita/WasmExplorer
Built with Clang/LLVM, AngularJS, Ace Editor, Emscripten, SpiderMonkey, Binaryen and Capstone.js.


Service version 3.5 (js: JavaScript-C55.0a1; clang: 5.0.0 (https://chromium.googlesource.com/external/github.com/llvm-mirror
```

Let's use WASM Explorer

https://mbebenita.github.io/WasmExplorer/

Let's begin with the a simple function



```cpp
int squarer(int num) {
  return num * num;
}
```

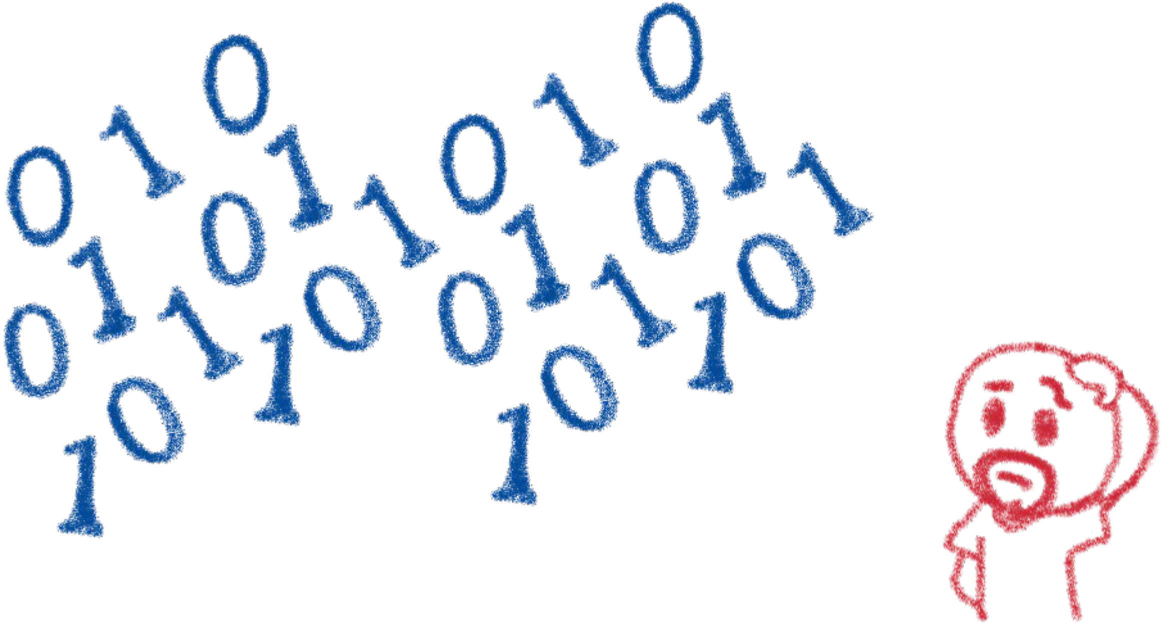
```wat
(module
  (type $type0 (func (param i32)
    (result i32)))
  (table 0 anyfunc)
  (memory 1)
  (export "memory" memory)
  (export "_Z7squareri" $func0)
  (func $func0 (param $var0 i32)
    (result i32)
    get_local $var0
    get_local $var0
    i32.mul
  )
)
```

```asm
wasm-function[0]:
  sub rsp, 8
  mov edx, edi
  mov ecx, edx
  mov eax, edx
  imul ecx, eax
  mov eax, ecx
  nop
  add rsp, 8
  ret
```

WAT: WebAssembly Text Format

Human readable version of the .wasm binary

Download the binary .wasm file

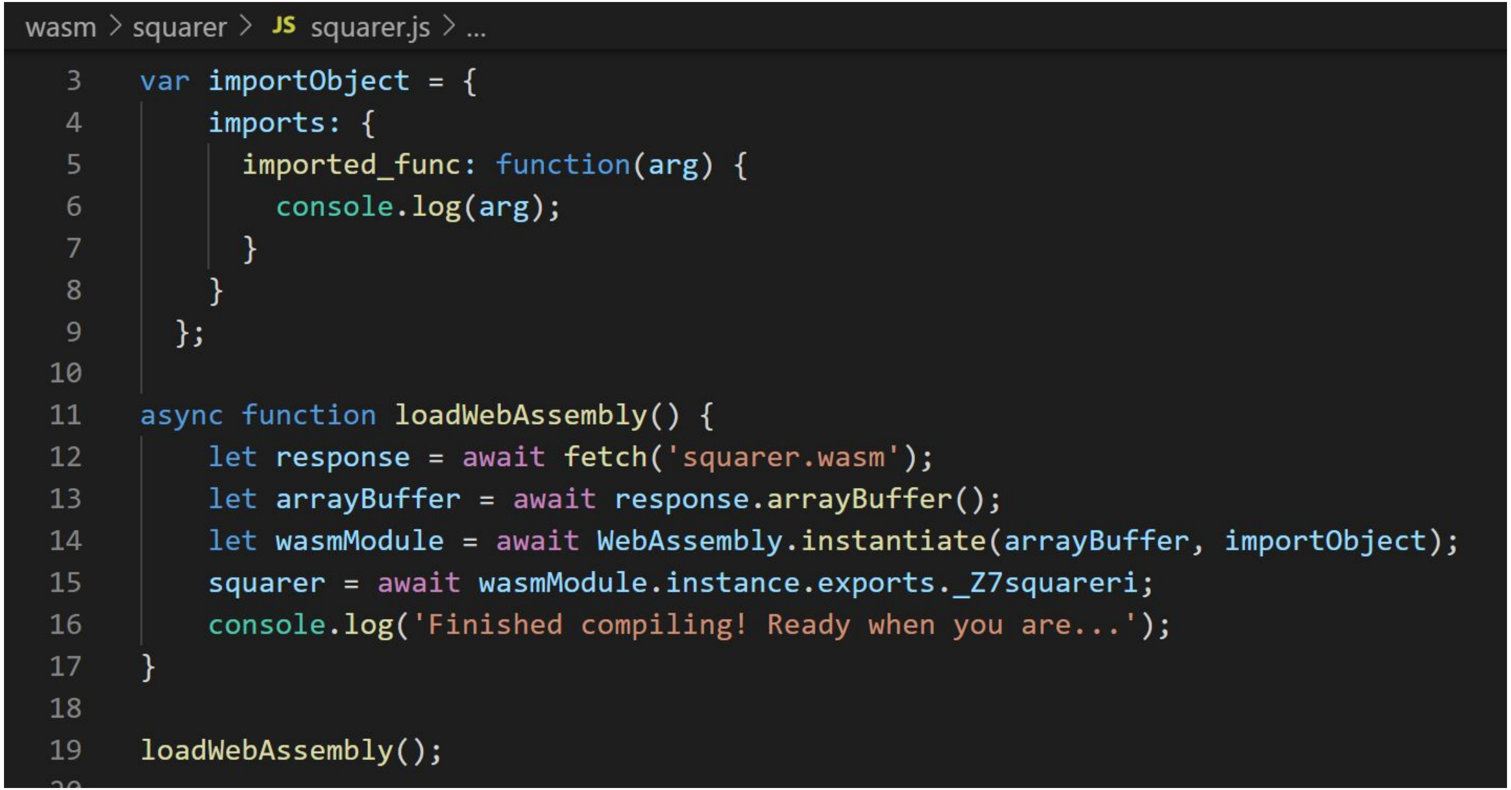


Now we need to call it from JS...

Instantiating the WASM

1. Get the `.wasm` binary file into an array buffer
2. Compile the bytes into a WebAssembly module
3. Instantiate the WebAssembly module

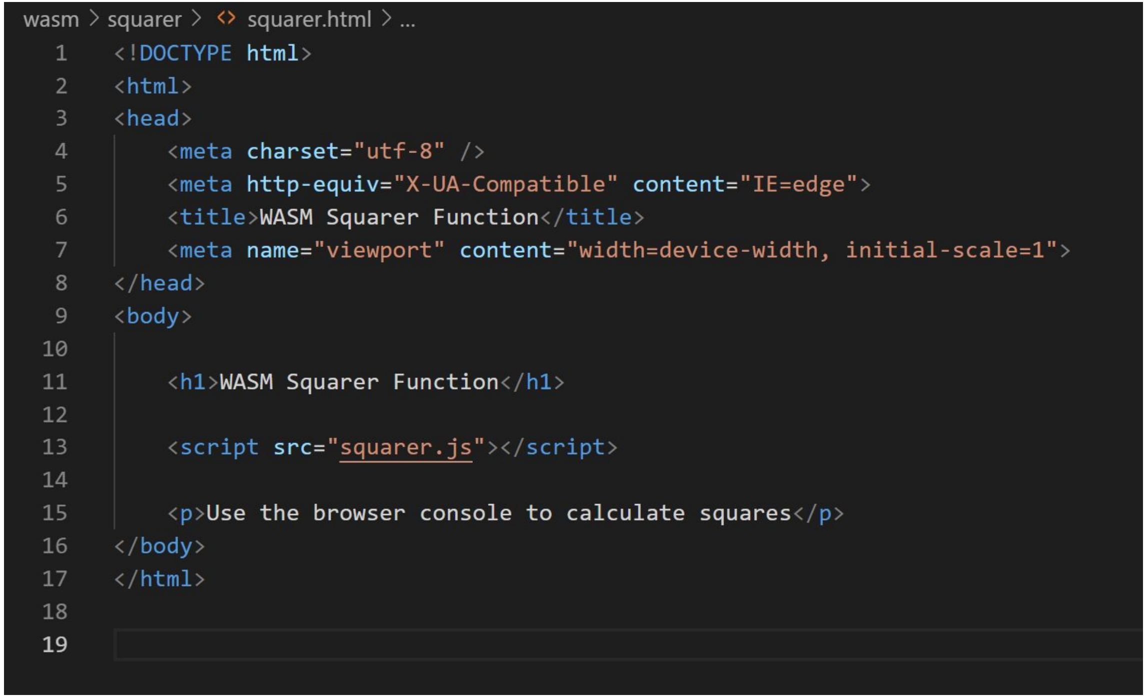
Instantiating the WASM



wasm > squarer > JS squarer.js > ...

```
var importObject = {
    imports: {
      imported_func: function(arg) {
        console.log(arg);
      }
    }
  };

async function loadWebAssembly() {
    let response = await fetch('squarer.wasm');
    let arrayBuffer = await response.arrayBuffer();
    let wasmModule = await WebAssembly.instantiate(arrayBuffer, importObject);
    squarer = await wasmModule.instance.exports._Z7squareri;
    console.log('Finished compiling! Ready when you are...');
}

loadWebAssembly();
```

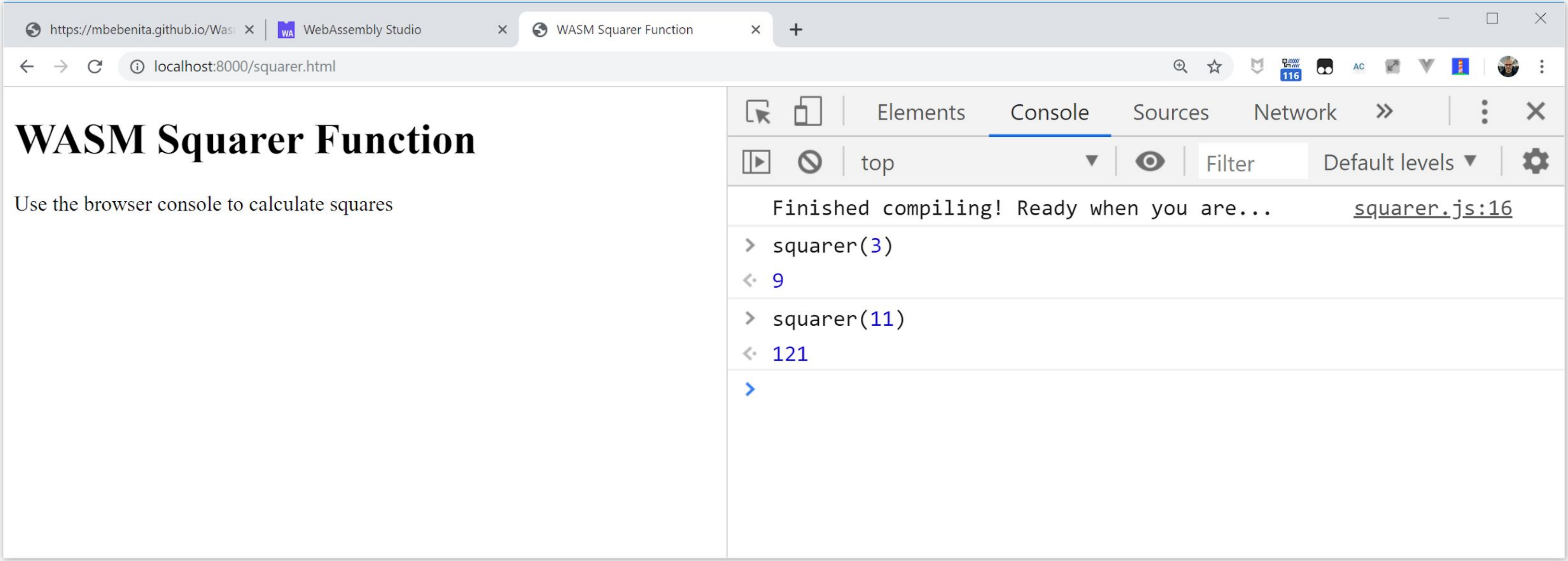
Loading the `squarer` function



```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>WASM Squarer Function</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>

    <h1>WASM Squarer Function</h1>

    <script src="squarer.js"></script>

    <p>Use the browser console to calculate squares</p>
</body>
</html>


```

We instantiate the WASM by loading the wrapping JS

Using it!



WASM Squarer Function

Use the browser console to calculate squares

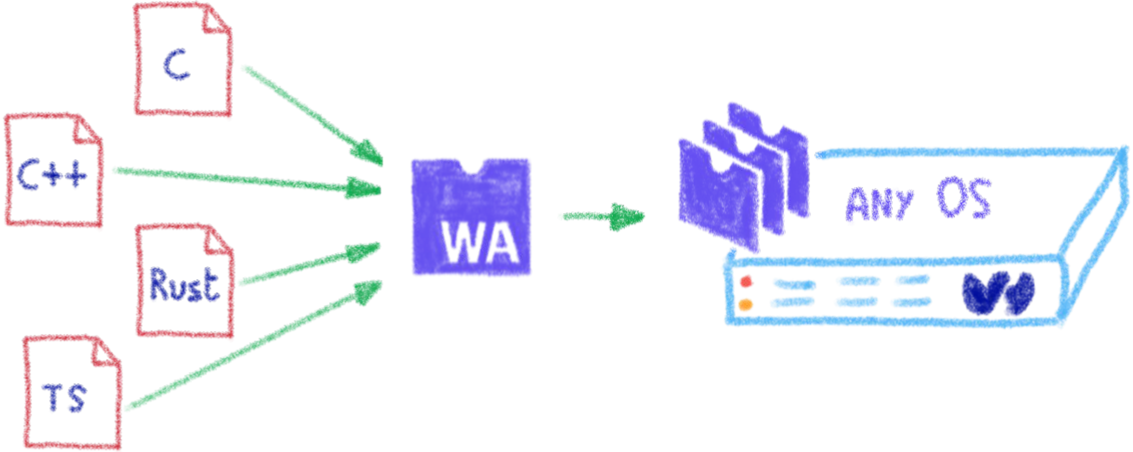
```
Finished compiling! Ready when you are...     squarer.js:16
> squarer(3)
<· 9
> squarer(11)
<· 121
>
```

Directly from the browser console (it's a simple demo...)

WASM outside the browser

Not only for web developers

Run any code on any client... almost



Languages compiling to WASM

Includes WAPM

```
wapm install optipng
```

Oh, like npm for WASM!

The WebAssembly Package Manager

Some use cases

What can I do with it?

Tapping into other languages ecosystems

OptiPNG (c)

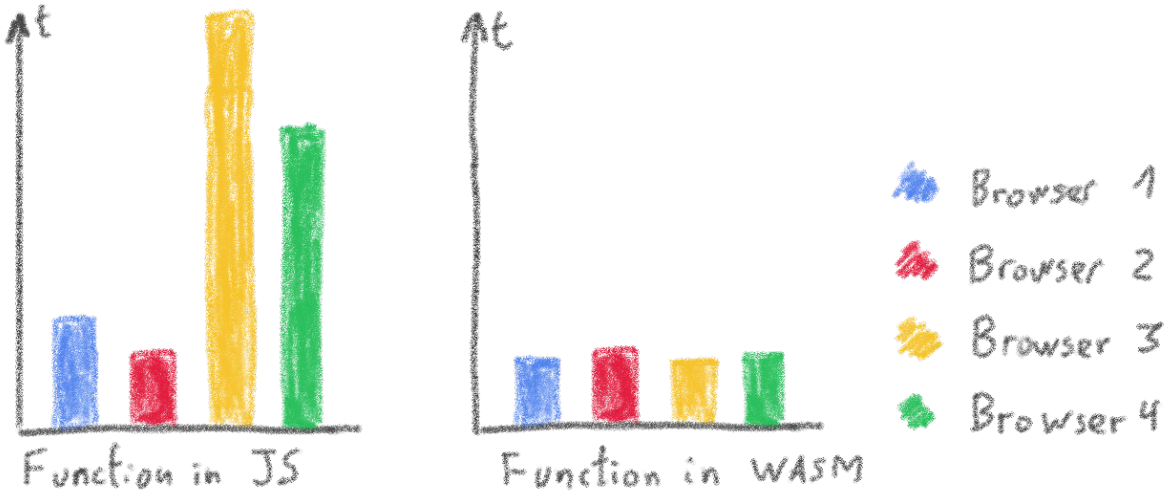
Resize (Rust)

MozJPEG (C++)

webp (c)

Don't rewrite libs anymore

Replacing problematic JS bits



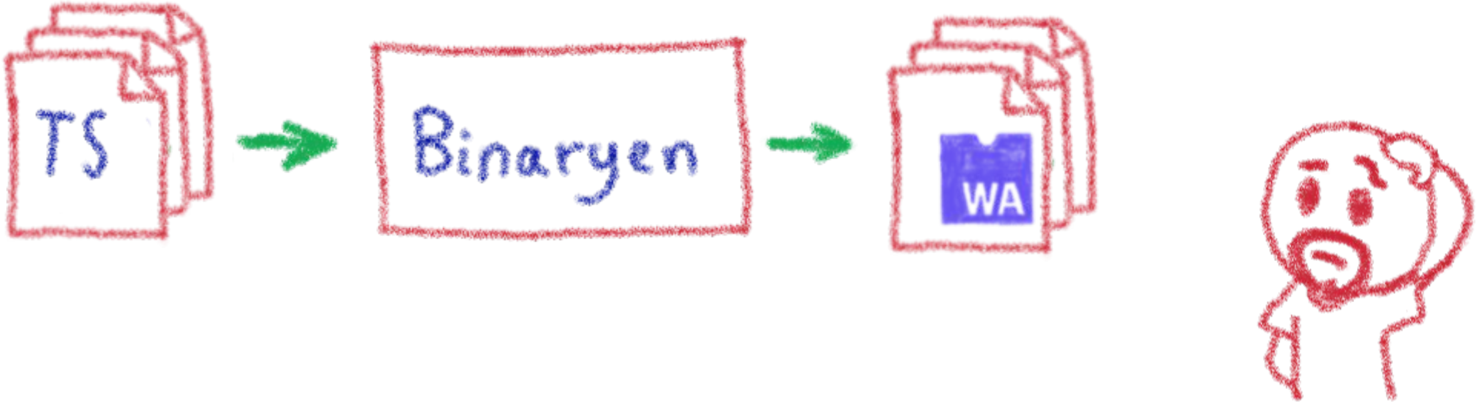
Predictable performance

Same peak performance, but less variation

AssemblyScript

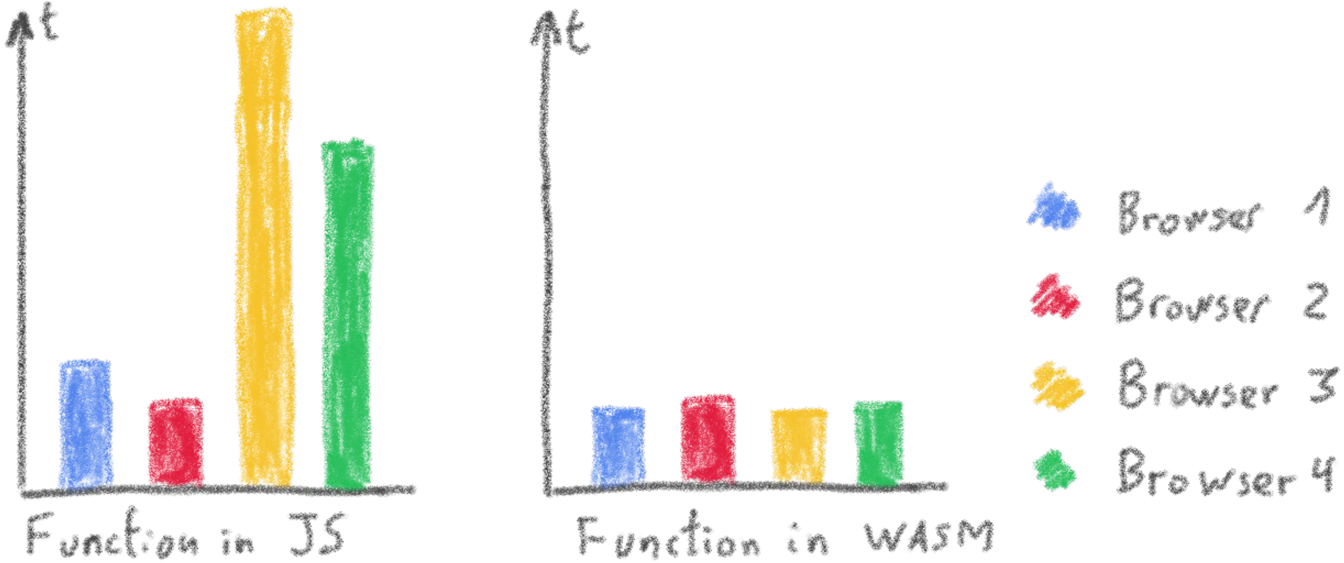
Writing WASM without learning a new language

TypeScript subset compiled to WASM



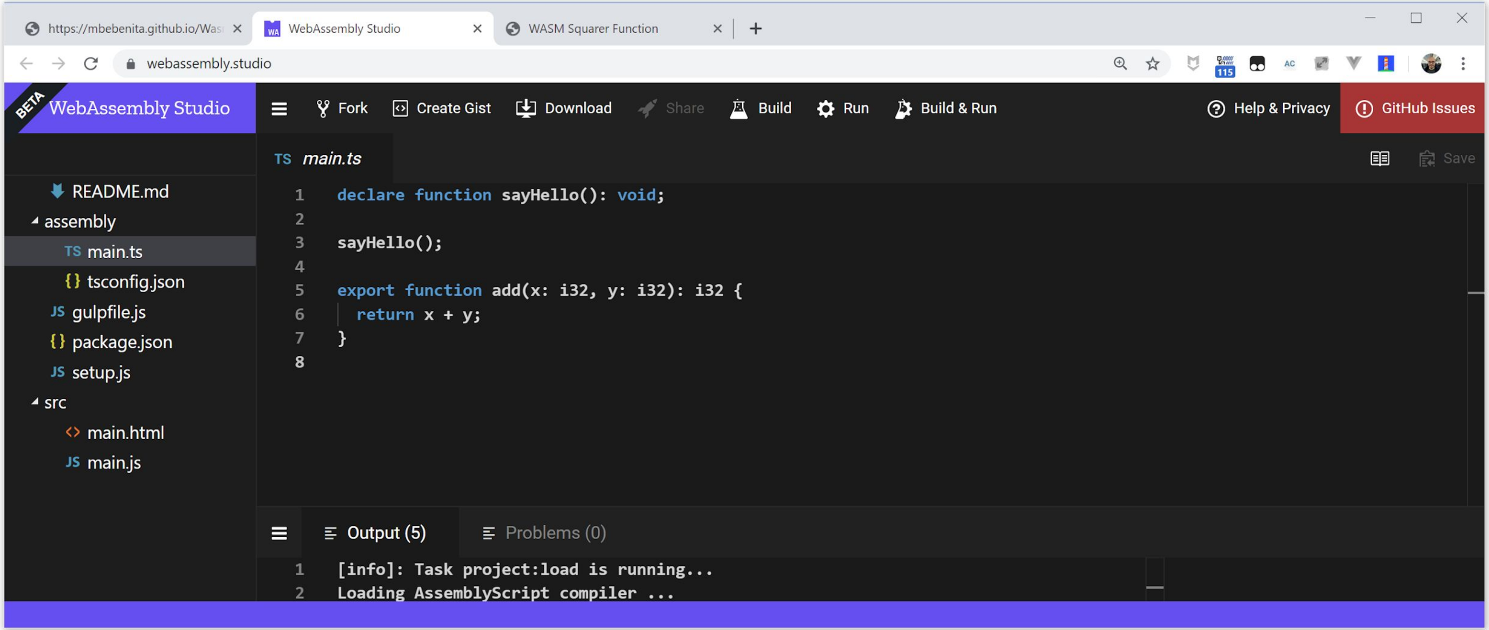
Why would I want to compile TypeScript to WASM?

Ahead of Time compiled TypeScript



More predictable performance

Avoiding the dynamicness of JavaScrip

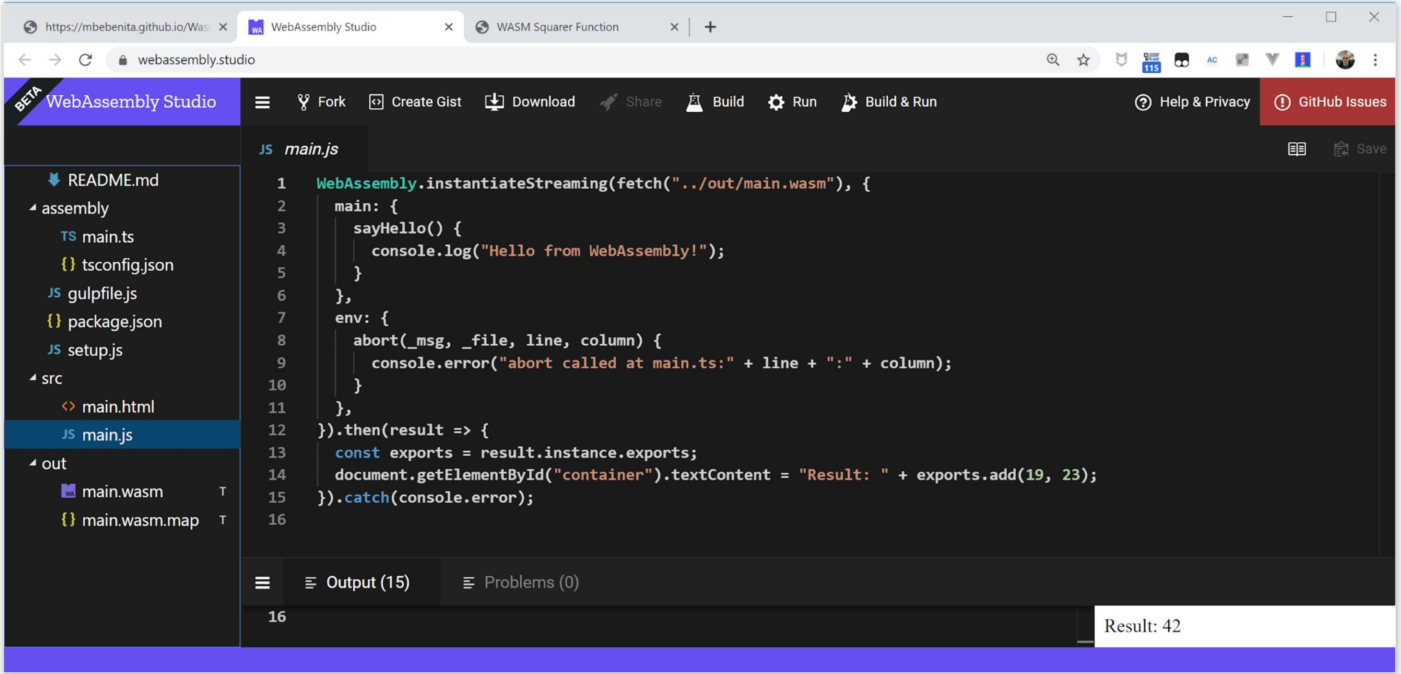


```
declare function sayHello(): void;

sayHello();

export function add(x: i32, y: i32): i32 {
  return x + y;
}
```

```
[info]: Task project:load is running...
Loading AssemblyScript compiler ...
```

More specific integer and floating point types

Objects cannot flow in and out of WASM yet



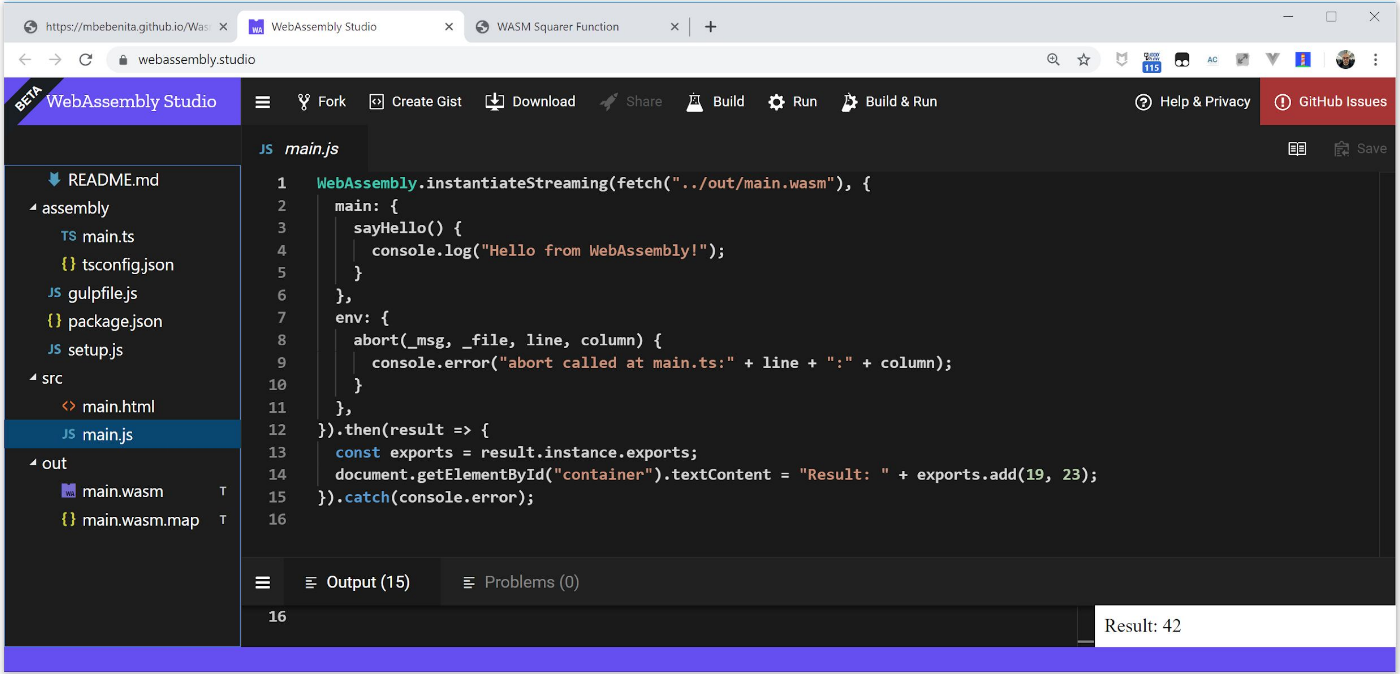
```
WebAssembly.instantiateStreaming(fetch("../out/main.wasm"), {
  main: {
    sayHello() {
      console.log("Hello from WebAssembly!");
    }
  },
  env: {
    abort(_msg, _file, line, column) {
      console.error("abort called at main.ts:" + line + ":" + column);
    }
  },
}).then(result => {
  const exports = result.instance.exports;
  document.getElementById("container").textContent = "Result: " + exports.add(19, 23);
}).catch(console.error);
```

```
16
```

```
Result: 42
```

Using a loader to write/read them to/from memory

No direct access to DOM

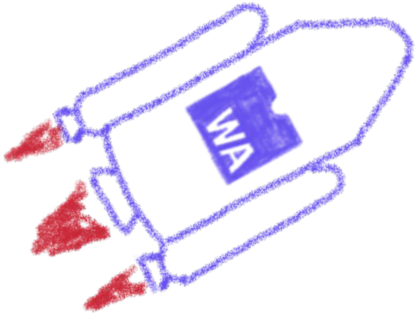


```
WebAssembly.instantiateStreaming(fetch("../out/main.wasm"), {
  main: {
    sayHello() {
      console.log("Hello from WebAssembly!");
    }
  },
  env: {
    abort(_msg, _file, line, column) {
      console.error("abort called at main.ts:" + line + ":" + column);
    }
  },
}).then(result => {
  const exports = result.instance.exports;
  document.getElementById("container").textContent = "Result: " + exports.add(19, 23);
}).catch(console.error);
```

```
16
```

Result: 42

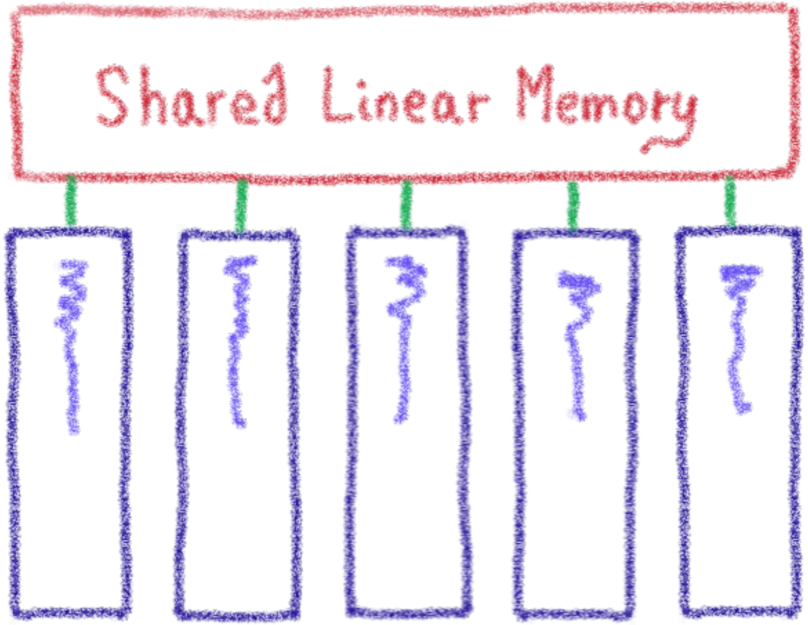
Glue code using exports/imports to/from JavaScript



Future

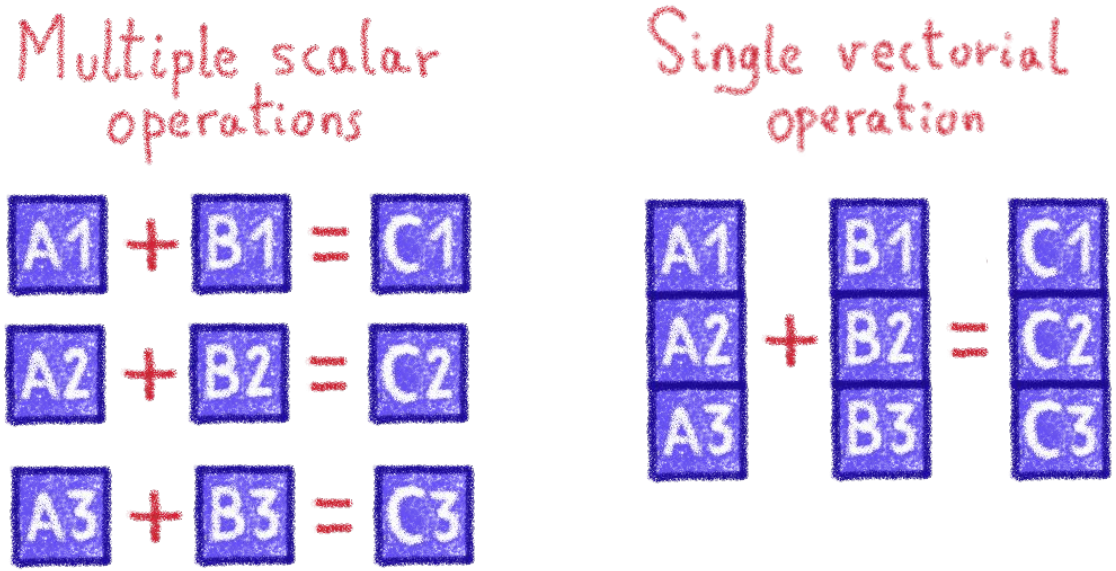
To the infinity and beyond!

WebAssembly Threads



Threads on Web Workers with shared linear memory

SIMD



Already available in Wasmer

Single Instruction, Multiple Data

Garbage collector

And exception handling