The

WebAssembly REVOLUTION

has begun

Jay Phelps | @_jayphelps

WebAssembly will change the way we think of **"web apps"**

Jay Phelps

Senior Software Engineer | NETFLIX

@_jayphelps

So...**what is WebAssembly?** aka **wasm**

Efficient, low-level bytecode for the Web

Efficient, **low-level bytecode** for the Web

Fast to **load** and **execute**

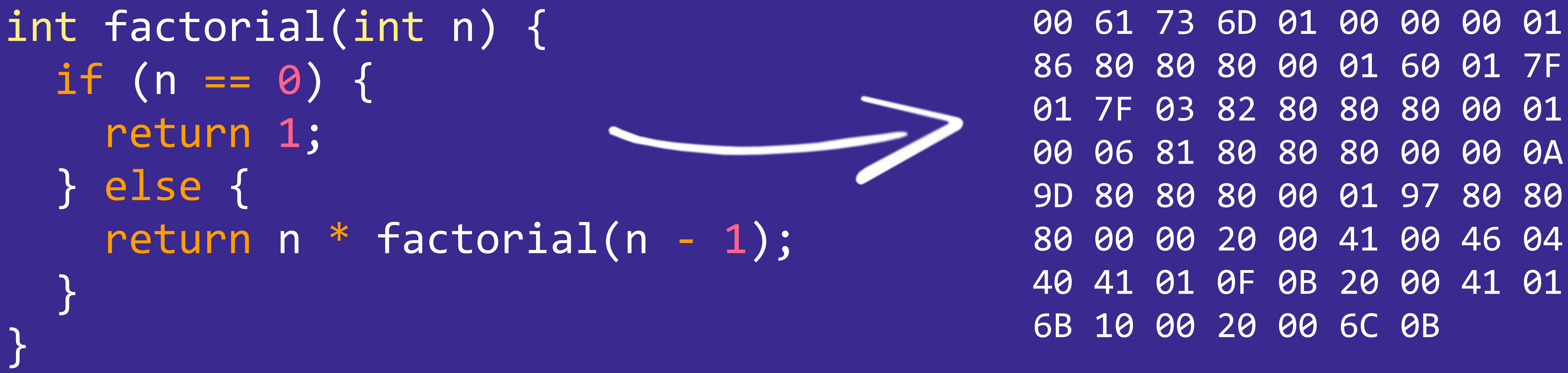
Efficient, **low-level bytecode** for the Web

0x6a

01101010

Intended as a **compilation target**

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```



```c
int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

```
00 61 73 6D 01 00 00 00 01
86 80 80 80 00 01 60 01 7F
01 7F 03 82 80 80 80 00 01
00 06 81 80 80 80 00 00 0A
9D 80 80 80 00 01 97 80 80
80 00 00 20 00 41 00 46 04
40 41 01 0F 0B 20 00 41 01
6B 10 00 20 00 6C 0B
```

Safe and **portable**

just like JavaScript is

Is it going to **kill JavaScript**?

Nope!*

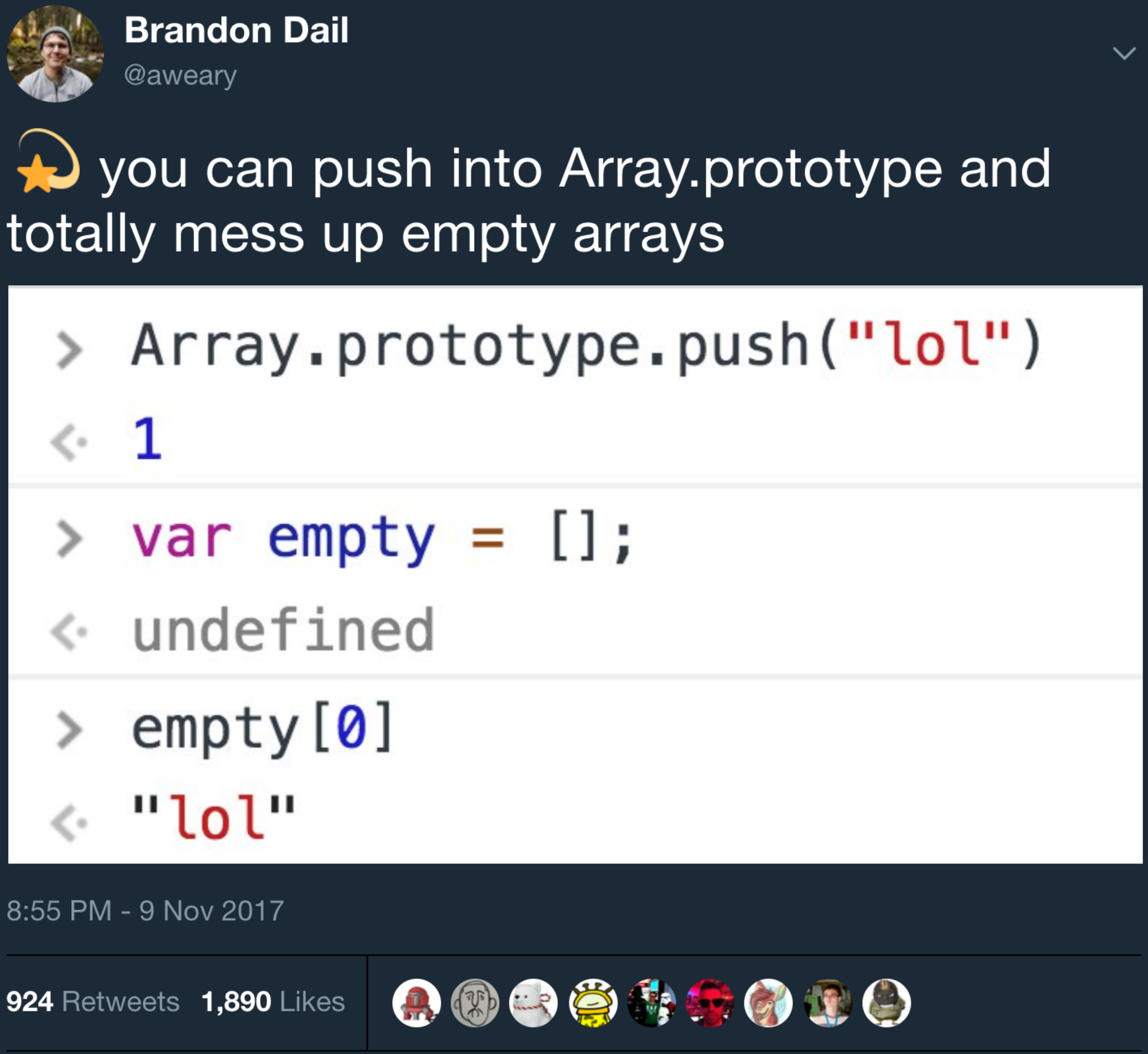
says browser vendors

*well...***maybe***...some day...a long time from now

(my own opinion)

Will we compile **JavaScript** to **WebAssembly**?

JavaScript is an *extremely* dynamic language



Brandon Dail
@aweary

💫 you can push into Array.prototype and totally mess up empty arrays

```
> Array.prototype.push("lol")
<· 1
> var empty = [];
<· undefined
> empty[0]
<· "lol"
```

8:55 PM - 9 Nov 2017

924 Retweets 1,890 Likes

Compiling ***JavaScript* to WebAssembly** would run **slower**

What about a something JavaScript-like?

AssemblyScript, **TurboScript**, **ThinScript**, etc

```
class Coordinates {
  x: i64;
  y: i64;
  z: i64;

  constructor(x: i64, y: i64, z: i64) {
    this.x = x;
    this.y = y;
    this.z = z;
  }
}

export function example() {
  let position = new Coordinates(10, 20, 30);
  // later
  delete position;
}
```

WebAssembly is still missing things

for broad adoption

v1 MVP is best suited for languages like **C/C++** and **Rust**

But other languages soon!

Things like **Java**, **OCaml**, and **even brand new ones**

REASON

```
type schoolPerson = Teacher | Director | Student(string);

let greeting = (stranger) =>
  switch stranger {
    | Teacher => "Hey professor!"
    | Director => "Hello director."
    | Student("Richard") => "Still here Ricky?"
    | Student(anyOtherName) => "Hey, " ++ anyOtherName ++ "."
  };
```

When should I target WebAssembly **right now**?

Heavily **CPU-bound number computations**

Games

Physics Simulation

Encryption

Compression

Video Decoding

Audio Mixing

Language Detection

Games

Physics Simulation

Encryption

Compression

Video Decoding

Audio Mixing

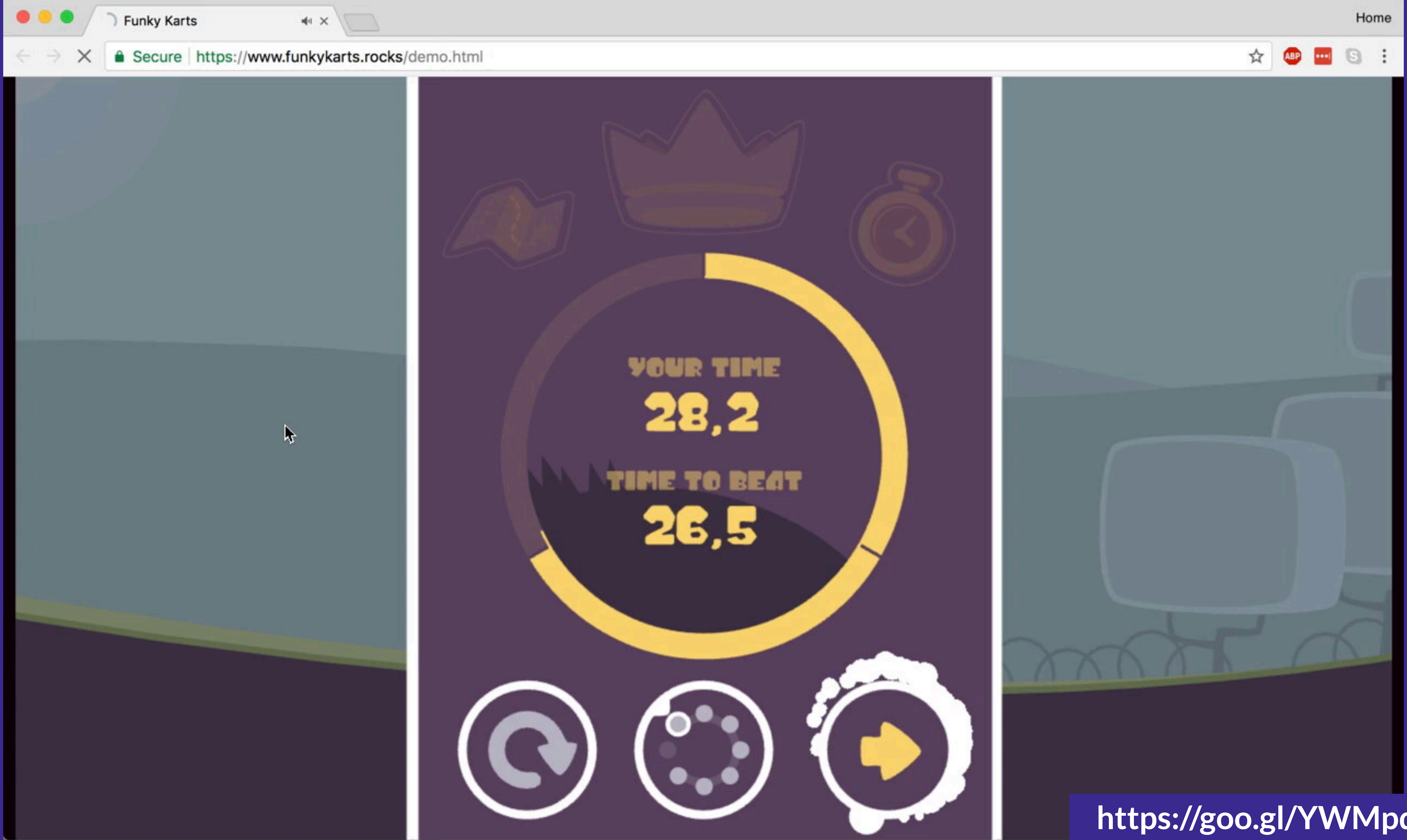
Language Detection

unity

UNREAL
ENGINE

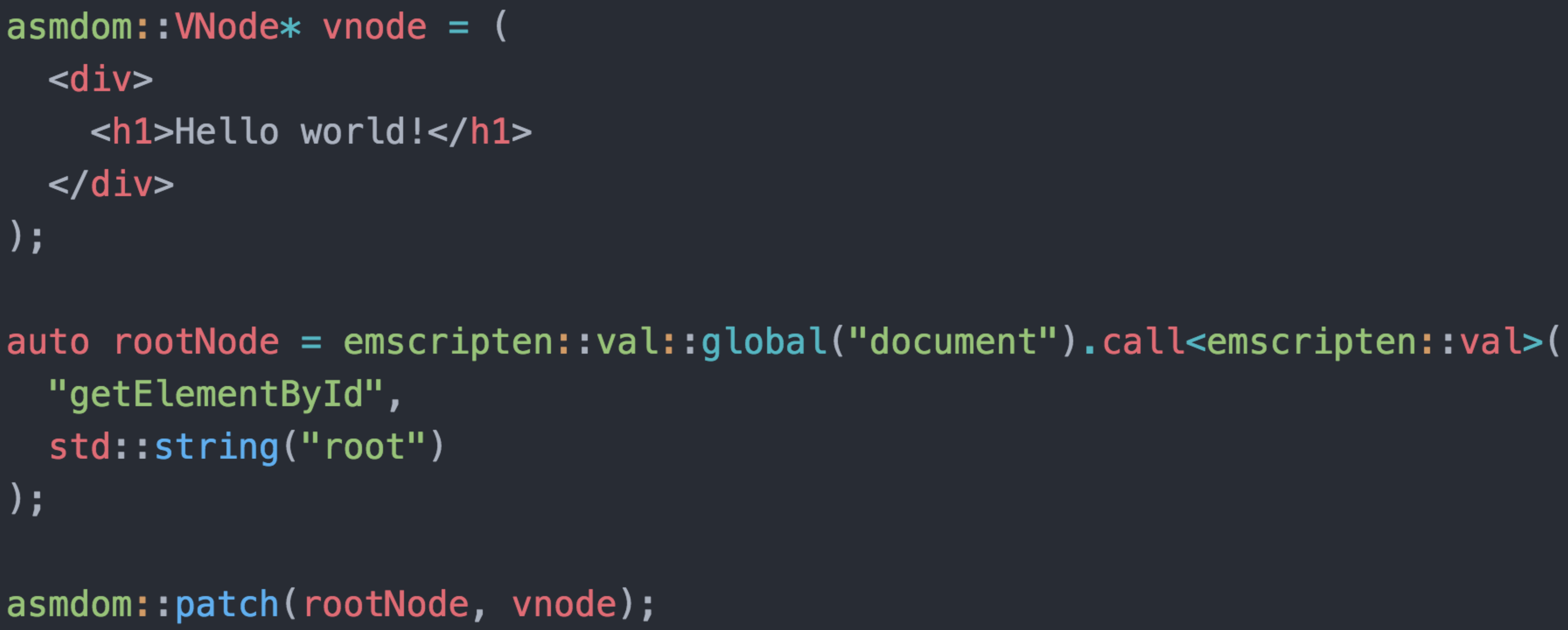
EpicZenGarden
Search or enter address
Search
Clear cache (205 MB)
FullScreen

https://goo.gl/yvV92X



https://goo.gl/YWMpdp

asm-dom



```
asmdom::VNode* vnode = (
  <div>
    <h1>Hello world!</h1>
  </div>
);

auto rootNode = emscripten::val::global("document").call<emscripten::val>(
  "getElementById",
  std::string("root")
);

asmdom::patch(rootNode, vnode);
```

https://goo.gl/XWBeme

Other use cases just around the corner

You'll likely **consume compiled** WebAssembly **libraries** even sooner

What was that binary stuff?

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

→

```
00 61 73 6D 01 00 00 00 01
86 80 80 80 00 01 60 01 7F
01 7F 03 82 80 80 80 00 01
00 06 81 80 80 80 00 00 0A
9D 80 80 80 00 01 97 80 80
80 00 00 20 00 41 00 46 04
40 41 01 0F 0B 20 00 41 01
6B 10 00 20 00 6C 0B
```

```
00 61 73 6D 01 00 00 00 01
86 80 80 80 00 01 60 01 7F
01 7F 03 82 80 80 80 00 01
00 06 81 80 80 80 00 00 0A
9D 80 80 80 00 01 97 80 80
80 00 00 20 00 41 00 46 04
40 41 01 0F 0B 20 00 41 01
6B 10 00 20 00 6C 0B
```

```
00 61 73 6D 01 00 00 00 01
86 80 80 80 00 01 60 01 7F
01 7F 03 82 80 80 80 00 01
00 06 81 80 80 80 00 00 0A
9D 80 80 80 00 01 97 80 80
80 00 00 20 00 41 00 46 04
40 41 01 0F 0B 20 00 41 01
6B 10 00 20 00 6C 0B
```

```
00 61 73 6D 01 00 00 00 01
86 80 80 80 00 01 60 01 7F
01 7F 03 82 80 80 80 00 01
00 06 81 80 80 80 00 00 0A
9D 80 80 80 00 01 97 80 80
80 00 00 20 00 41 00 46 04
40 41 01 0F 0B 20 00 41 01
6B 10 00 20 00 6C 0B
```

```
03 82 80 80 80
81 80 80 80 00
80 80 00 01 97
00 20 00 41 00
```

```
03  82  80  80  80
81  80      80  00
80  80      01  97
00  20  00  41  00
```

Binary can be *a little* intimidating

Protip: don't worry about it

(unless of course, you want to)

Textual representation to the rescue!

```
(func $factorial (param $n i32) (result i32)
  get_local $n
  i32.const 0
  i32.eq
  if $if0
  i32.const 1
  return
  end $if0
  get_local $n
  i32.const 1
  i32.sub
  call $factorial
  get_local $n
  i32.mul
)
```

```
(func $factorial (param $n i32) (result i32)
  get_local $n
  i32.const 0
  i32.eq
  if $if0
  i32.const
  return
  end $if0
  get_local
  i32.const
  i32.sub
  call $factor
  get_local $n
  i32.mul
)
```

WebAssembly is a **stack machine** language stack machine: ***instructions*** **on a stack**

1 + 2

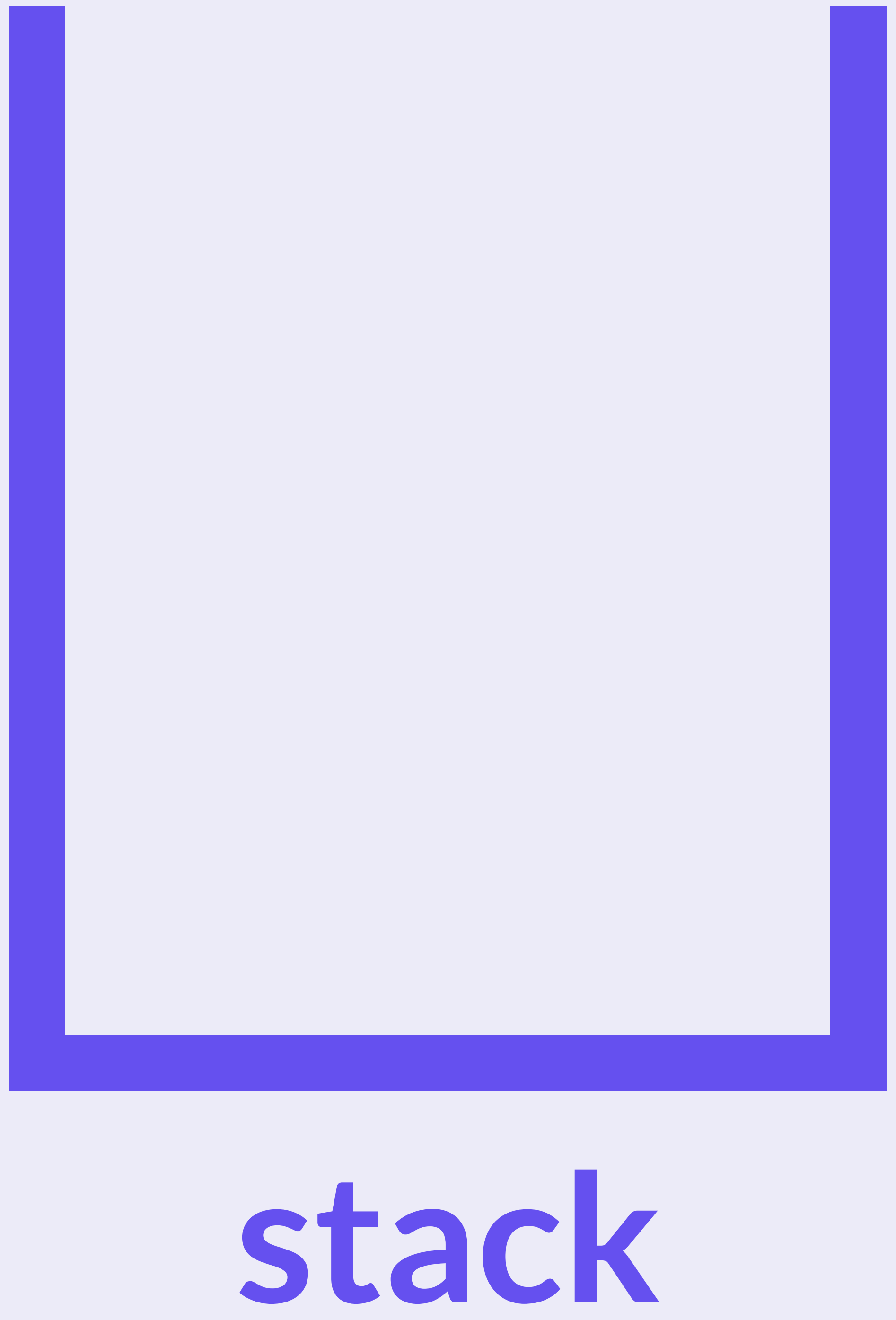
mnemonic

opcode

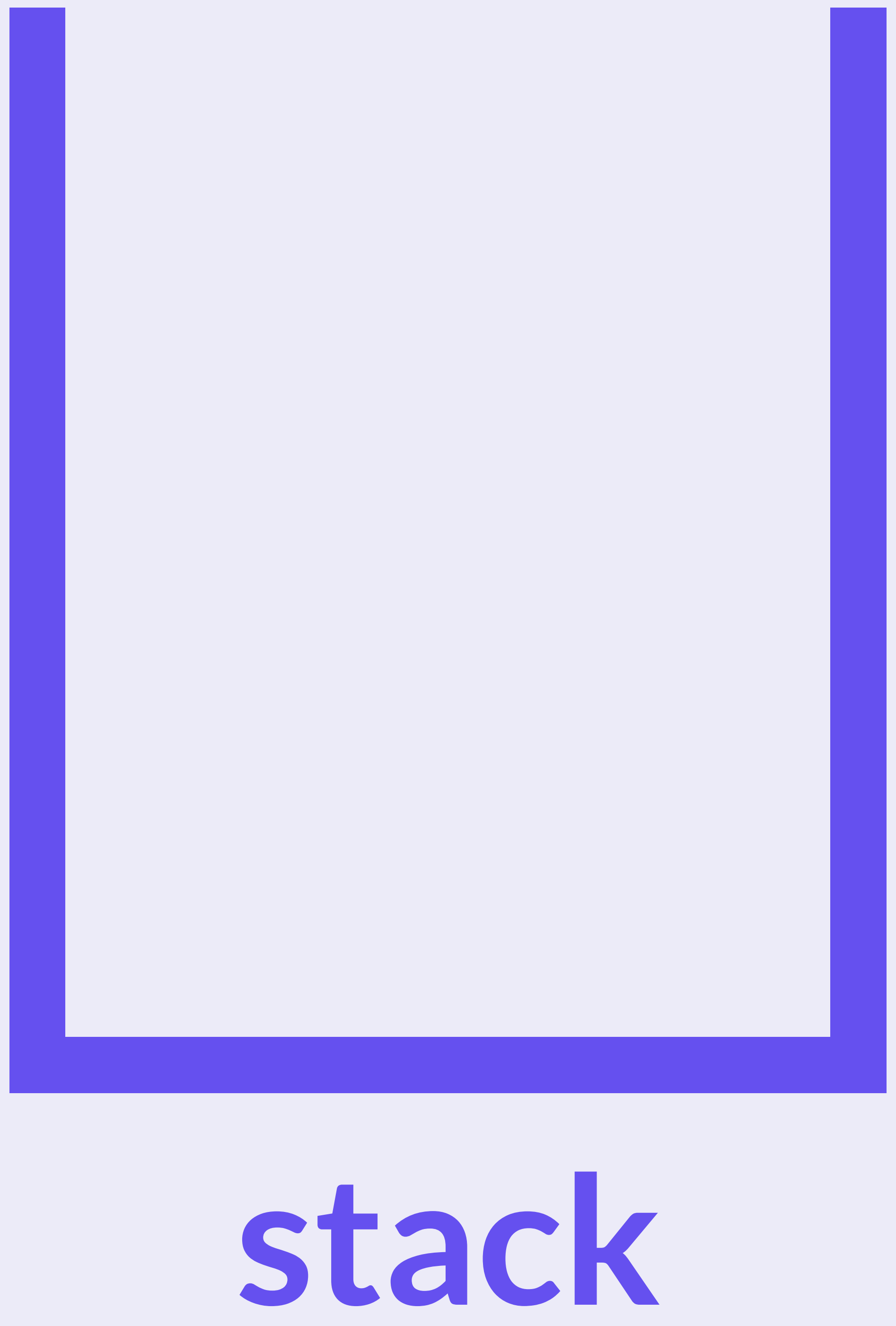
`i32.add` → `0x6a`

01101010

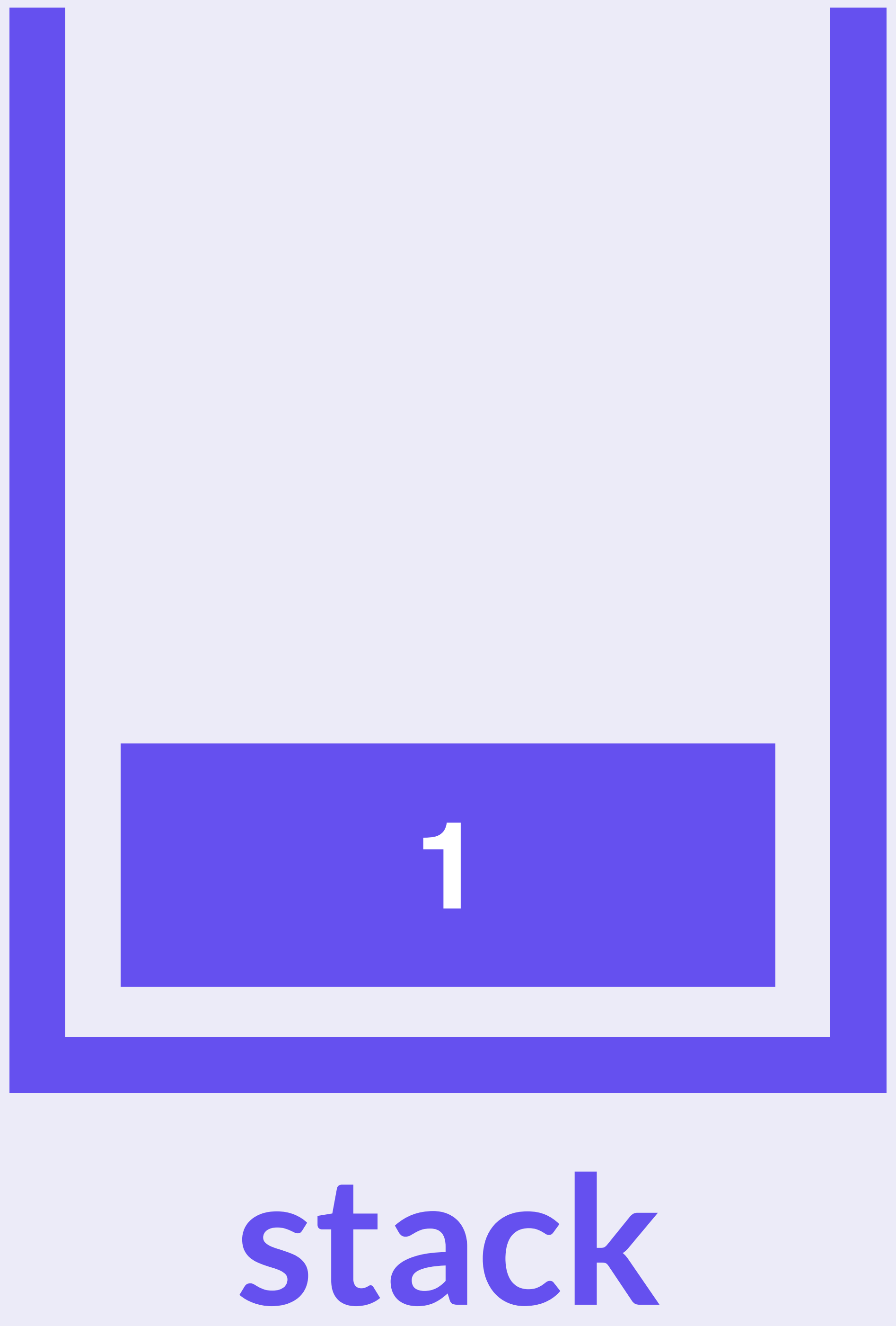
```
i32.const 1
i32.const 2
i32.add
```



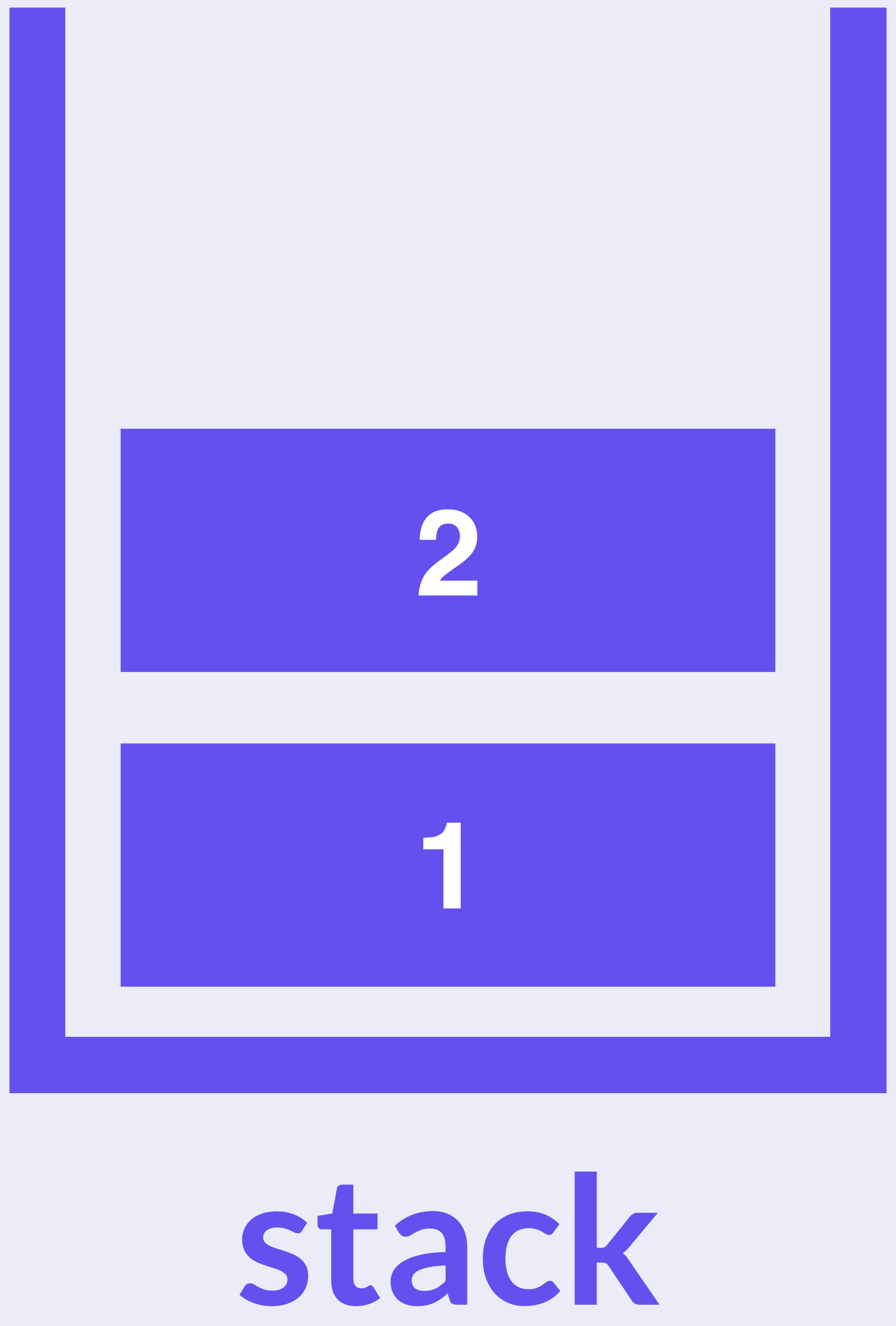
```
i32.const 1
i32.const 2
i32.add
```



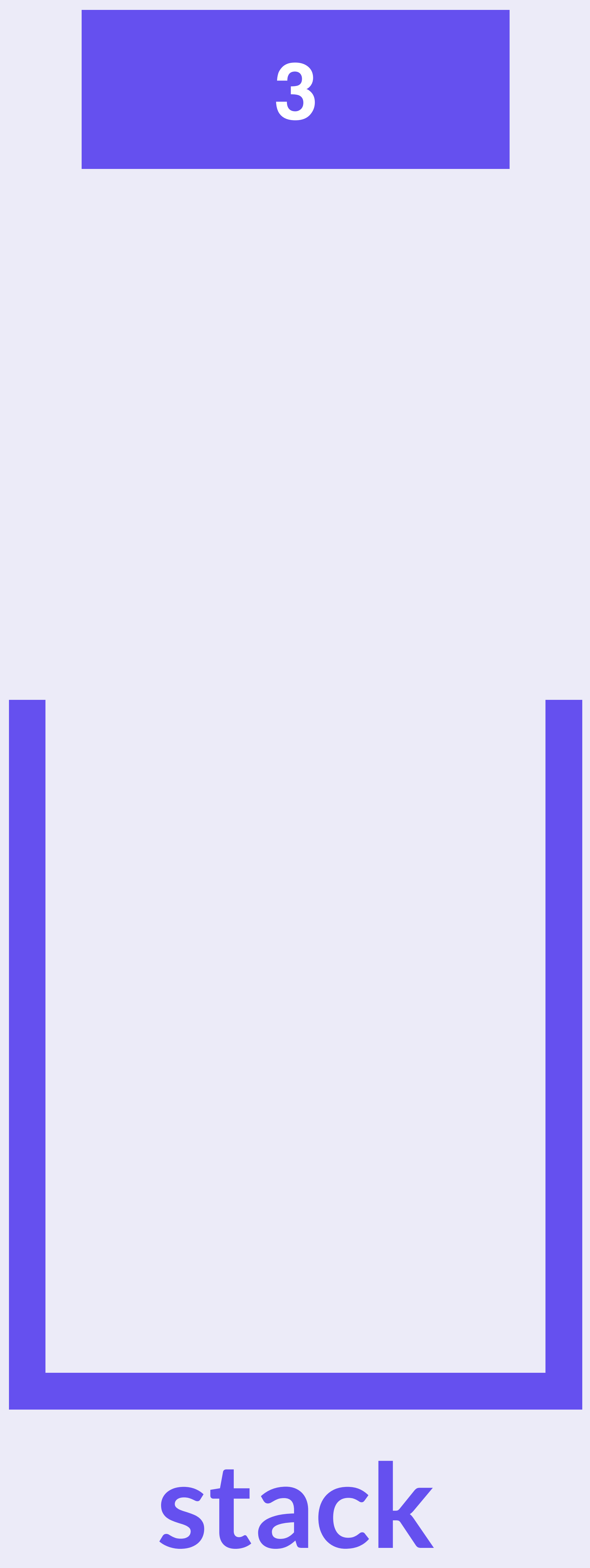
```
i32.const 1
i32.const 2
i32.add
```



```
i32.const 1
i32.const 2
i32.add
```



```
i32.const 1
i32.const 2
i32.add
```



3

stack

```
i32.const 1
i32.const 2
i32.add
call $log
```

Compilers usually apply optimizations

real-world output is **often less understandable** to humans

```
i32.const 1
i32.const 2
i32.add
call $log
```

```
i32.const 3
call $log
```

Most tooling supports an **Abstract Syntax Tree** (AST)

still compiled and evaluated as a stack machine

```
i32.const 1
i32.const 2
i32.add
call $log
```

```
(call $log
  (i32.add
    (i32.const 1)
    (i32.const 2)
  )
)
```

s-expressions

Yep, looks like Lisp

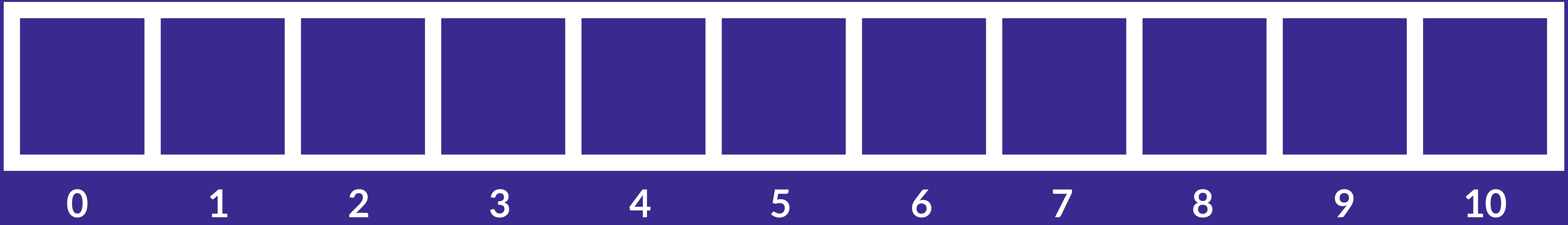
```
(call $log
  (i32.add
    (i32.const 1)
    (i32.const 2)
  )
)
```

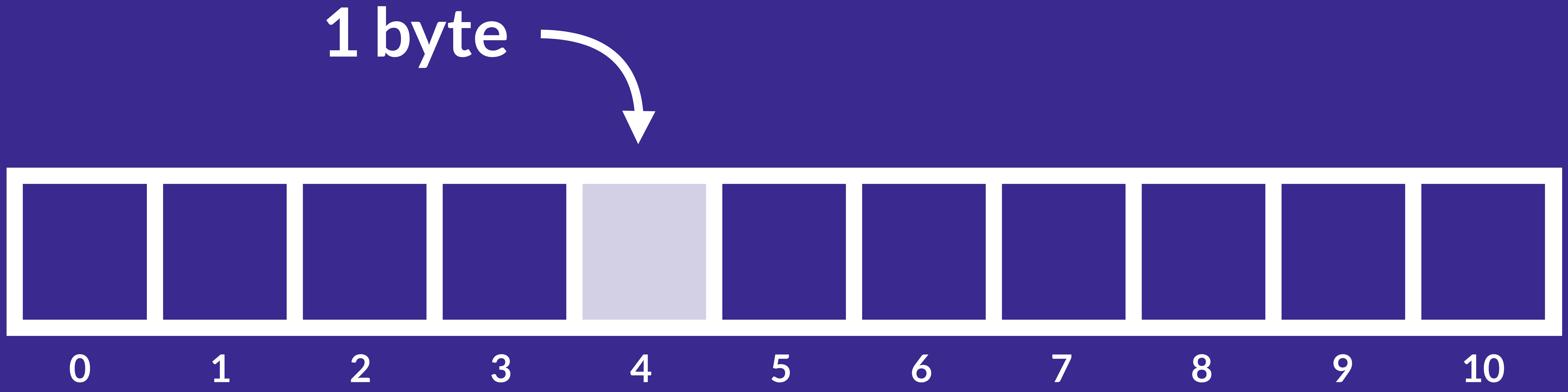
Source map support is coming

What about memory on the heap?

A **linear memory** is a contiguous, byte-addressable range of memory

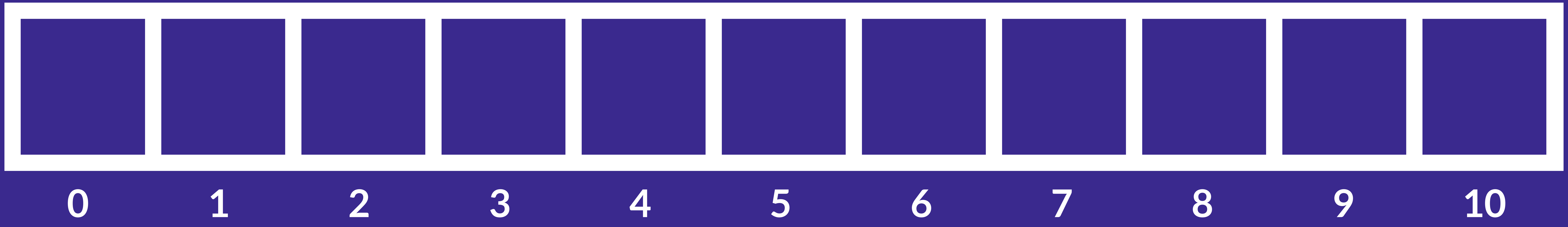
Accessed with instructions like
i32.load and **i32.store**


0
1
2
3
4
5
6
7
8
9
10

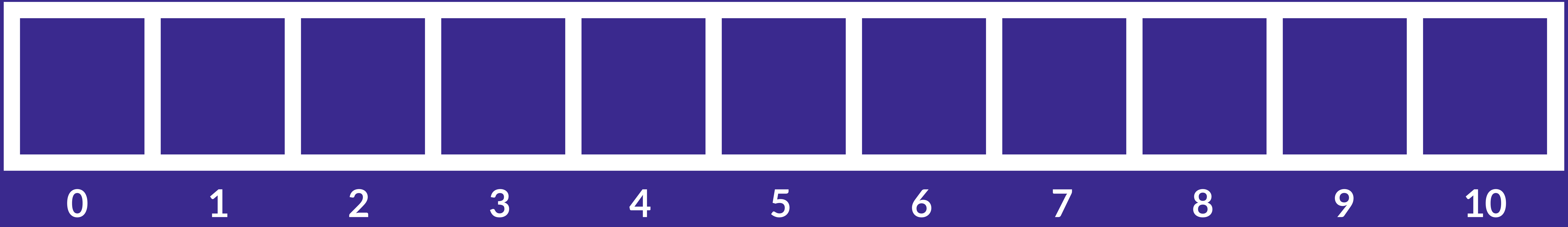


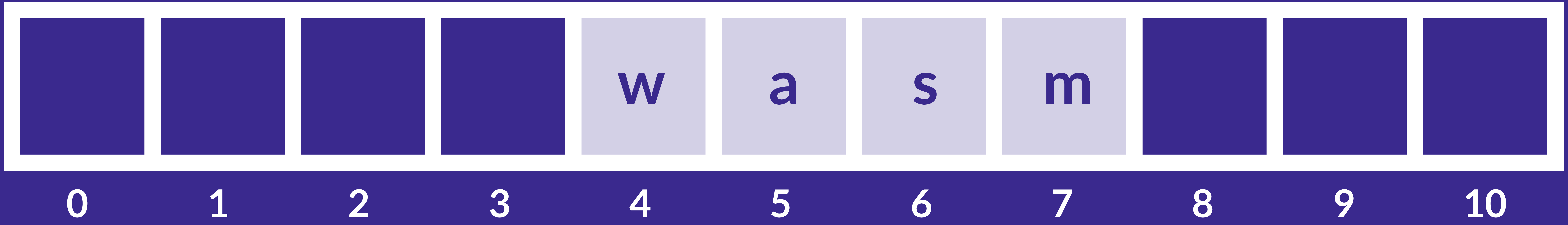
1 byte

0 1 2 3 4 5 6 7 8 9 10

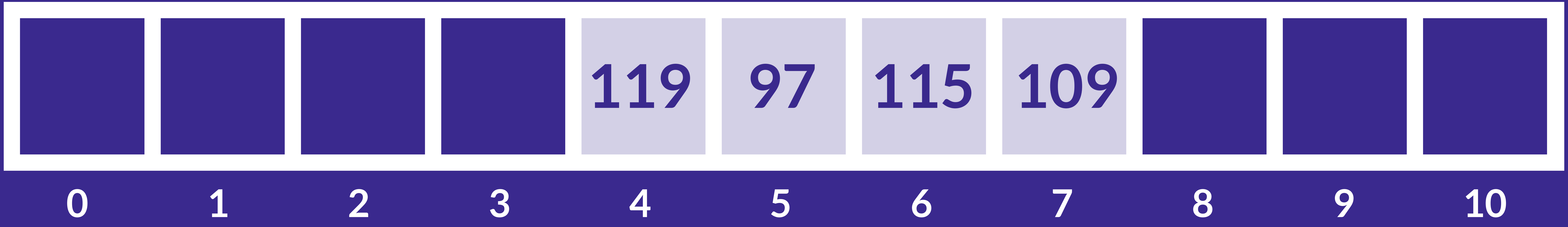

0
1
2
3
4
5
6
7
8
9
10

w a s m





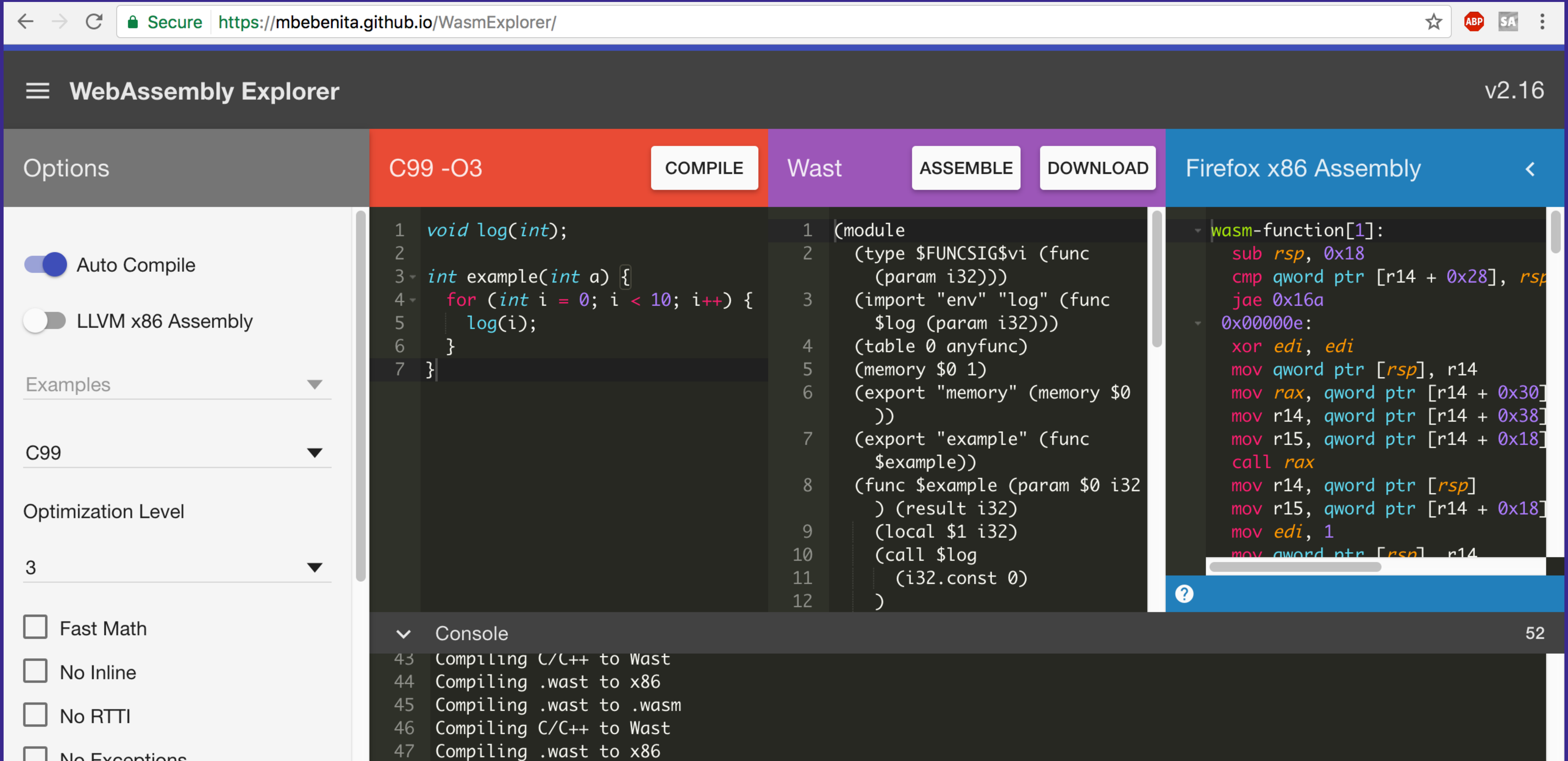
				w	a	s	m			
0	1	2	3	4	5	6	7	8	9	10



0	1	2	3	4	5	6	7	8	9	10
				119	97	115	109			

How do I **get started**?

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



● Auto Compile

○ LLVM x86 Assembly

Examples ▼

C99 ▼

Optimization Level

3 ▼

□ Fast Math

□ No Inline

□ No RTTI

□ No Exceptions

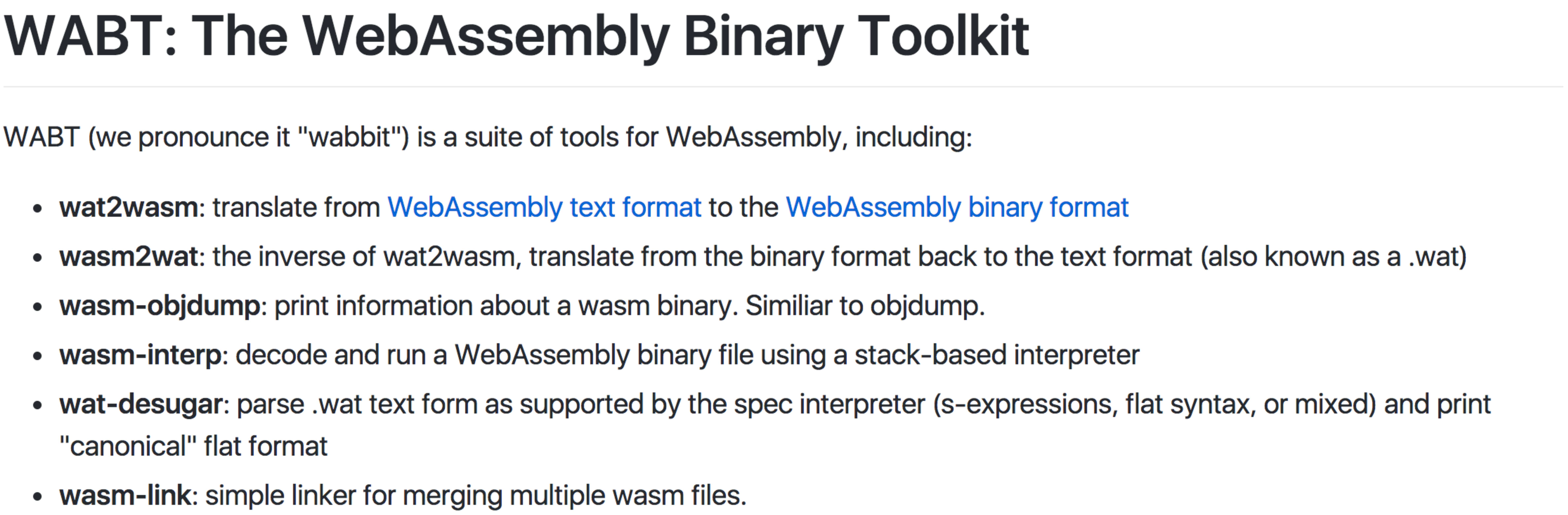
```
void log(int);

int example(int a) {
  for (int i = 0; i < 10; i++) {
    log(i);
  }
}
```

```
(module
  (type $FUNCSIG$vi (func
    (param i32)))
  (import "env" "log" (func
    $log (param i32)))
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0
    ))
  (export "example" (func
    $example))
  (func $example (param $0 i32
    ) (result i32)
    (local $1 i32)
    (call $log
      (i32.const 0)
    )
```

```
wasm-function[1]:
  sub rsp, 0x18
  cmp qword ptr [r14 + 0x28], rsp
  jae 0x16a
 0x00000e:
  xor edi, edi
  mov qword ptr [rsp], r14
  mov rax, qword ptr [r14 + 0x30]
  mov r14, qword ptr [r14 + 0x38]
  mov r15, qword ptr [r14 + 0x18]
  call rax
  mov r14, qword ptr [rsp]
  mov r15, qword ptr [r14 + 0x18]
  mov edi, 1
  mov qword ptr [rsp], r14
```

```
Compiling C/C++ to Wast
Compiling .wast to x86
Compiling .wast to .wasm
Compiling C/C++ to Wast
Compiling .wast to x86
```

https://github.com/WebAssembly/wabt

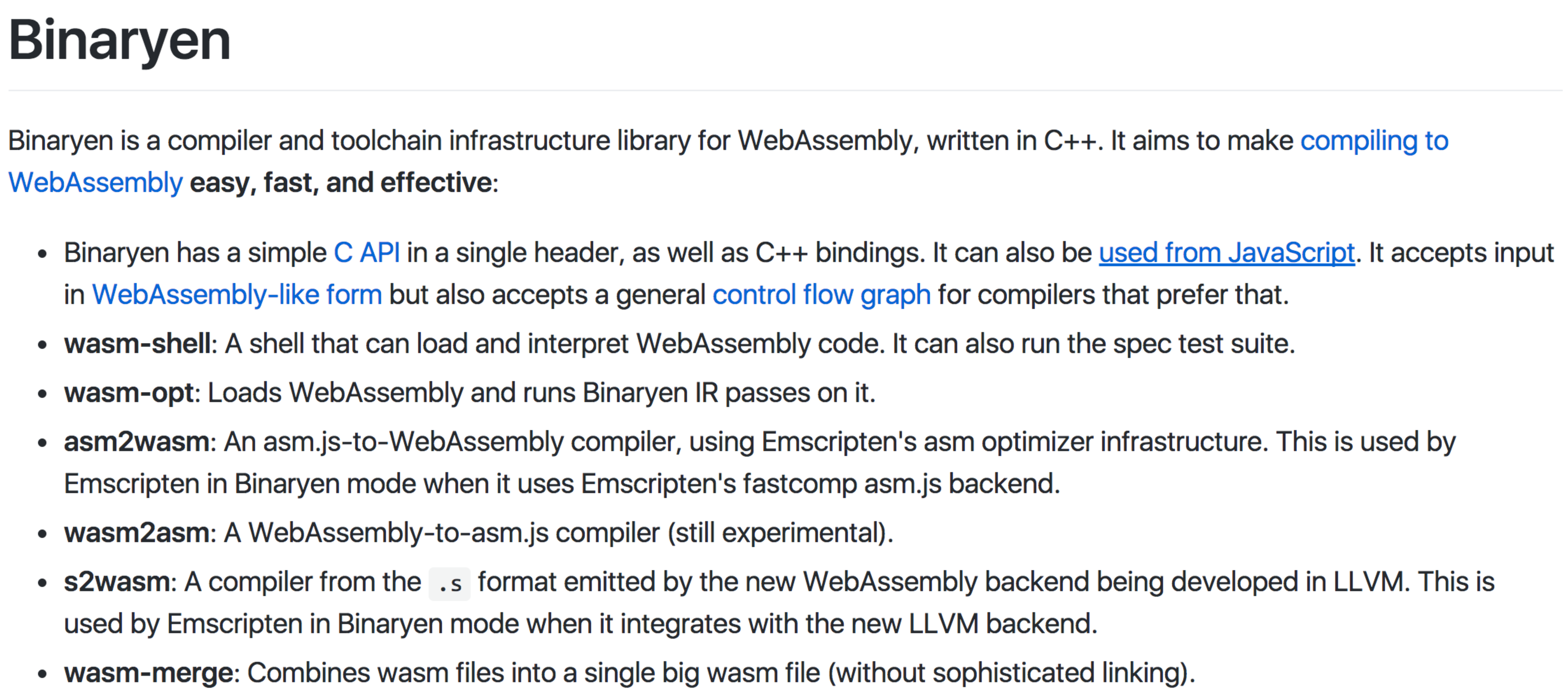


WABT: The WebAssembly Binary Toolkit

WABT (we pronounce it "wabbit") is a suite of tools for WebAssembly, including:

- **wat2wasm**: translate from WebAssembly text format to the WebAssembly binary format
- **wasm2wat**: the inverse of wat2wasm, translate from the binary format back to the text format (also known as a .wat)
- **wasm-objdump**: print information about a wasm binary. Similiar to objdump.
- **wasm-interp**: decode and run a WebAssembly binary file using a stack-based interpreter
- **wat-desugar**: parse .wat text form as supported by the spec interpreter (s-expressions, flat syntax, or mixed) and print "canonical" flat format
- **wasm-link**: simple linker for merging multiple wasm files.

https://github.com/WebAssembly/binaryen



Binaryen

Binaryen is a compiler and toolchain infrastructure library for WebAssembly, written in C++. It aims to make compiling to WebAssembly **easy, fast, and effective**:

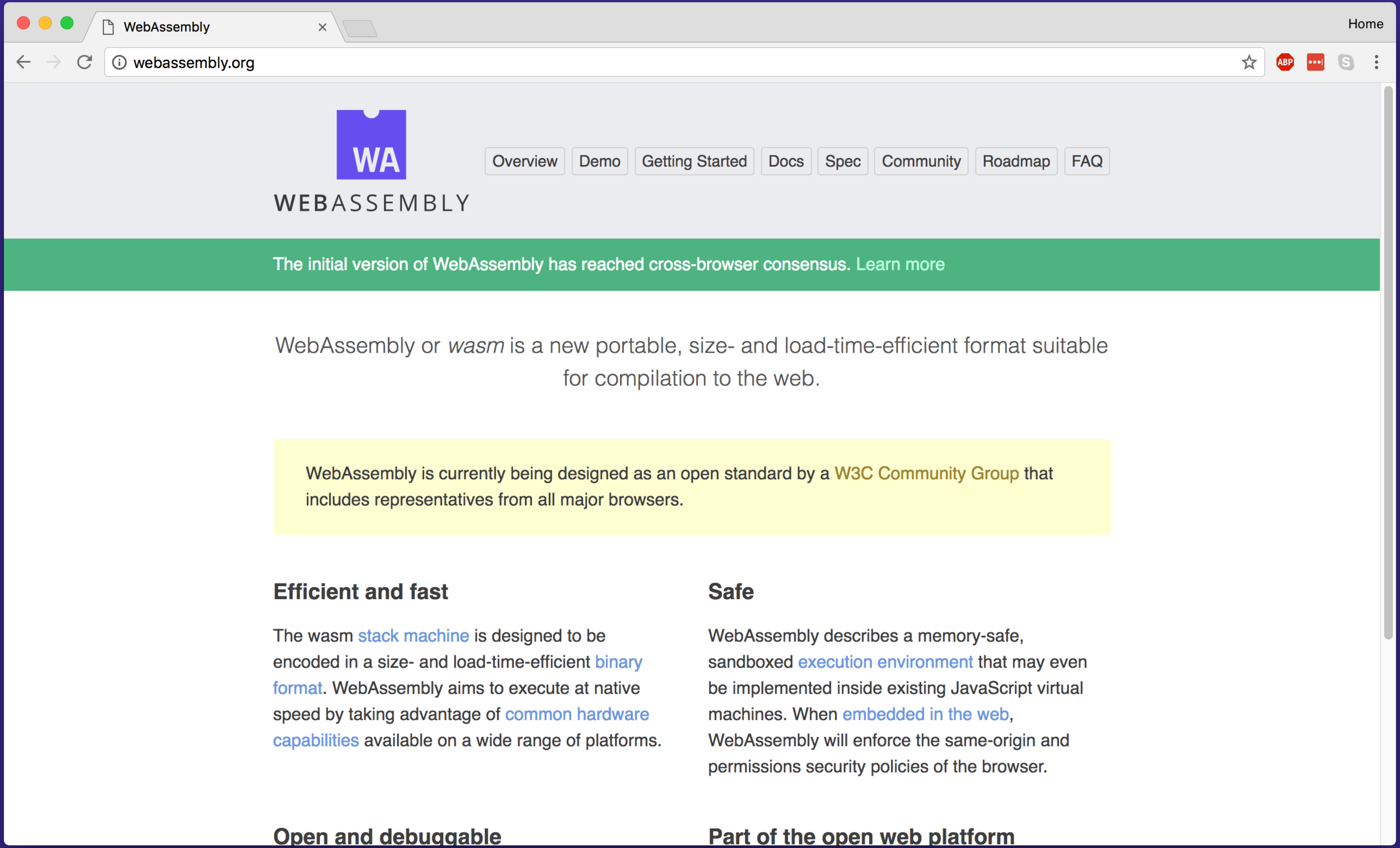
- Binaryen has a simple C API in a single header, as well as C++ bindings. It can also be used from JavaScript. It accepts input in WebAssembly-like form but also accepts a general control flow graph for compilers that prefer that.
- **wasm-shell**: A shell that can load and interpret WebAssembly code. It can also run the spec test suite.
- **wasm-opt**: Loads WebAssembly and runs Binaryen IR passes on it.
- **asm2wasm**: An asm.js-to-WebAssembly compiler, using Emscripten's asm optimizer infrastructure. This is used by Emscripten in Binaryen mode when it uses Emscripten's fastcomp asm.js backend.
- **wasm2asm**: A WebAssembly-to-asm.js compiler (still experimental).
- **s2wasm**: A compiler from the `.s` format emitted by the new WebAssembly backend being developed in LLVM. This is used by Emscripten in Binaryen mode when it integrates with the new LLVM backend.
- **wasm-merge**: Combines wasm files into a single big wasm file (without sophisticated linking).

emscripten

emscripten

```
$ emcc main.c -s WASM=1 -o app.html
```

webassembly.org



The initial version of WebAssembly has reached cross-browser consensus. Learn more

WebAssembly or *wasm* is a new portable, size- and load-time-efficient format suitable for compilation to the web.

WebAssembly is currently being designed as an open standard by a W3C Community Group that includes representatives from all major browsers.

Efficient and fast

The wasm stack machine is designed to be encoded in a size- and load-time-efficient binary format. WebAssembly aims to execute at native speed by taking advantage of common hardware capabilities available on a wide range of platforms.

Safe

WebAssembly describes a memory-safe, sandboxed execution environment that may even be implemented inside existing JavaScript virtual machines. When embedded in the web, WebAssembly will enforce the same-origin and permissions security policies of the browser.

Open and debuggable

Part of the open web platform

Webpack is adding support (!!!)

They received a $125,000 grant from MOSS

Imagine a **cpp-loader** / **rust-loader**

What's missing?

Direct access to Web APIs

You have call into JavaScript, right now

Garbage collection

also necessary for better interop with JavaScript and WebIDL

Multi-threading

Browser support?

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			49						
		4 52	60			10.2			
	3 15	55	61	10.1		10.3		4.4	
11	16	56	62	11	48	11	all	56	61
	17	57	63	TP	49				
		58	64		50				
		59	65						

The revolution is **just beginning**

ethereum

BLOCKCHAIN APP PLATFORM

Efficient, low-level bytecode **for the Web**

Efficient, low-level bytecode ~~for the Web~~

H
HD
HISTORY.COM

Questions?

@_jayphelps

Thanks!

🐦 @_jayphelps

```c
void log(char *);

void example() {
  log("wasm");
}
```

```
(module
  (import "env" "log" (func $log (param i32)))
  (memory $0 1)
  (data (i32.const 0) "wasm\00")
  (func $example
    (call $log
      (i32.const 0)
    )
  )
)
```

```
(module
  (import "env" "log" (func $log (param i32)))
  (memory $0 1)
  (data (i32.const 0) "wasm\00")
  (func $example
    (call $log
      (i32.const 0)
    )
  )
)
```

```
(module
  (import "env" "log" (func $log (param i32)))
  (memory $0 1)
  (data (i32.const 0) "wasm\00")
  (func $example
    (call $log
      (i32.const 0)
    )
  )
)
```