WebAssembly

history, internals, security and more

Mingshen Sun, April 16, 2020

Background

History all starts from the Web

- Text
- HTML
- Flash, ActionScript, Adobe Air, Silverlight, Java, ActiveX, JavaScript

Background

History all starts from the Web

- Text
- HTML
- Flash, ActionScript, Adobe Air, Silverlight, Java, ActiveX, **JavaScript**

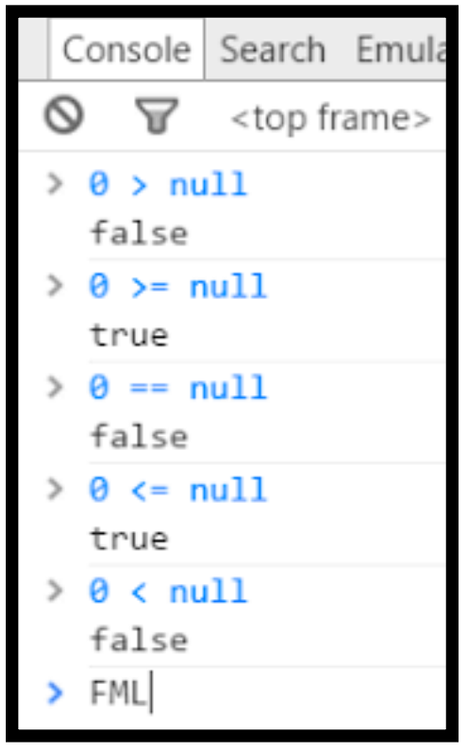
Background

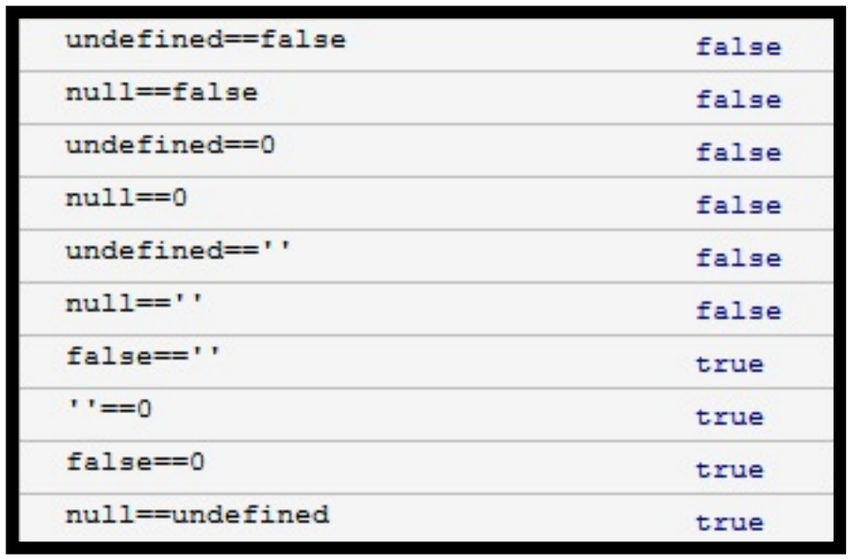
JavaScript engine

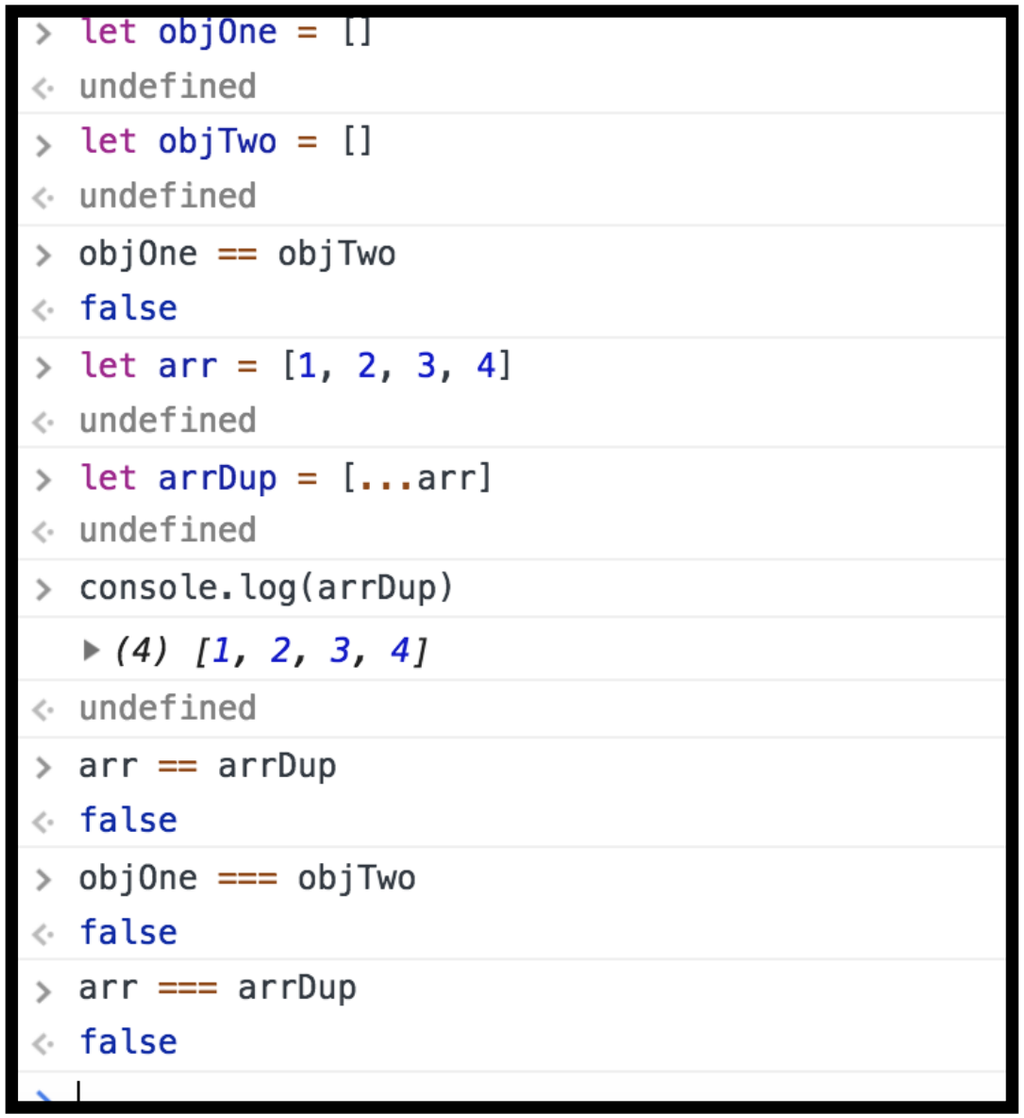
- SpiderMonkey
- V8
- WebKit/JavaScriptCore
- ~~Chakra~~

Background

JavaScript is ...

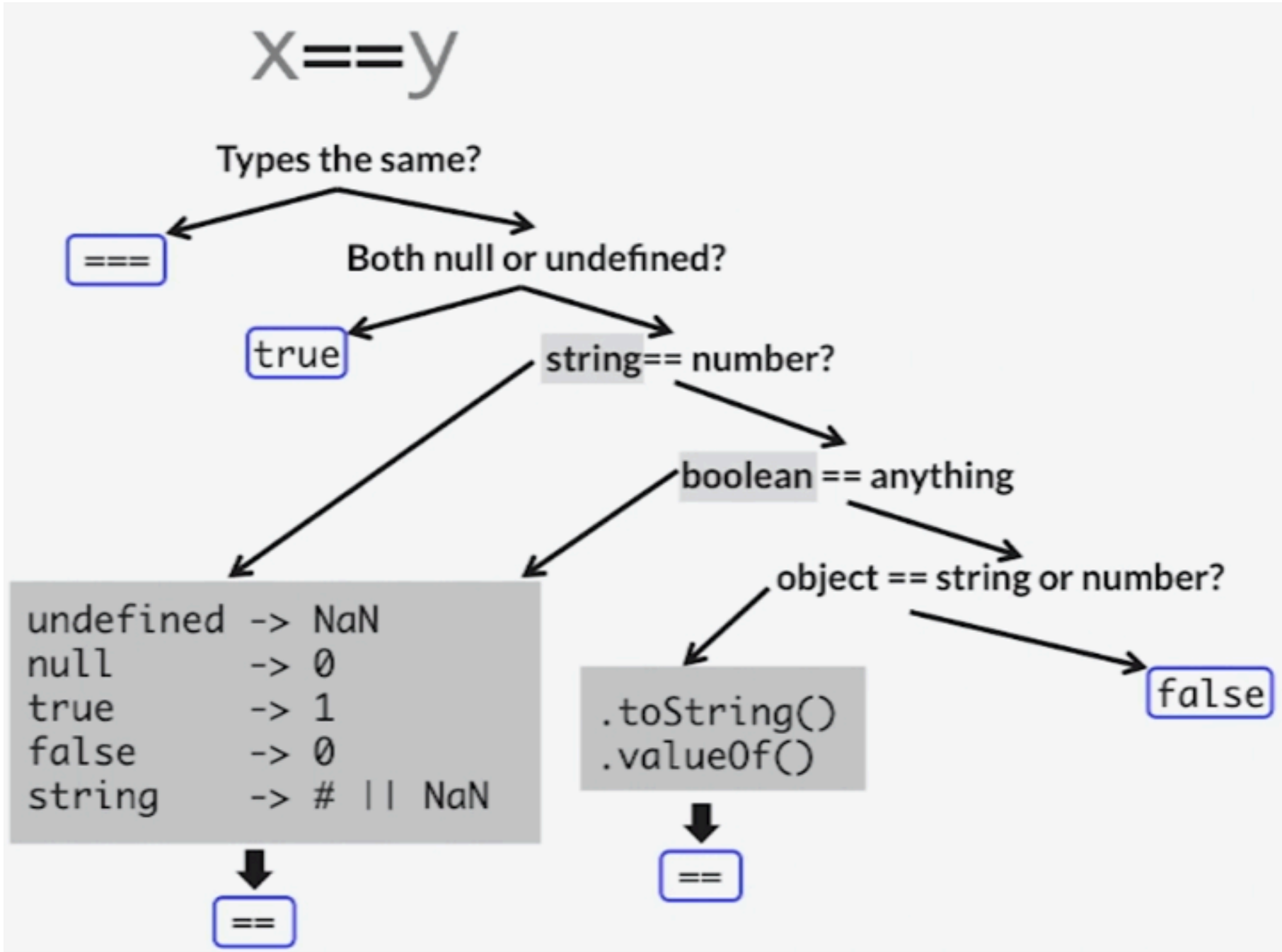




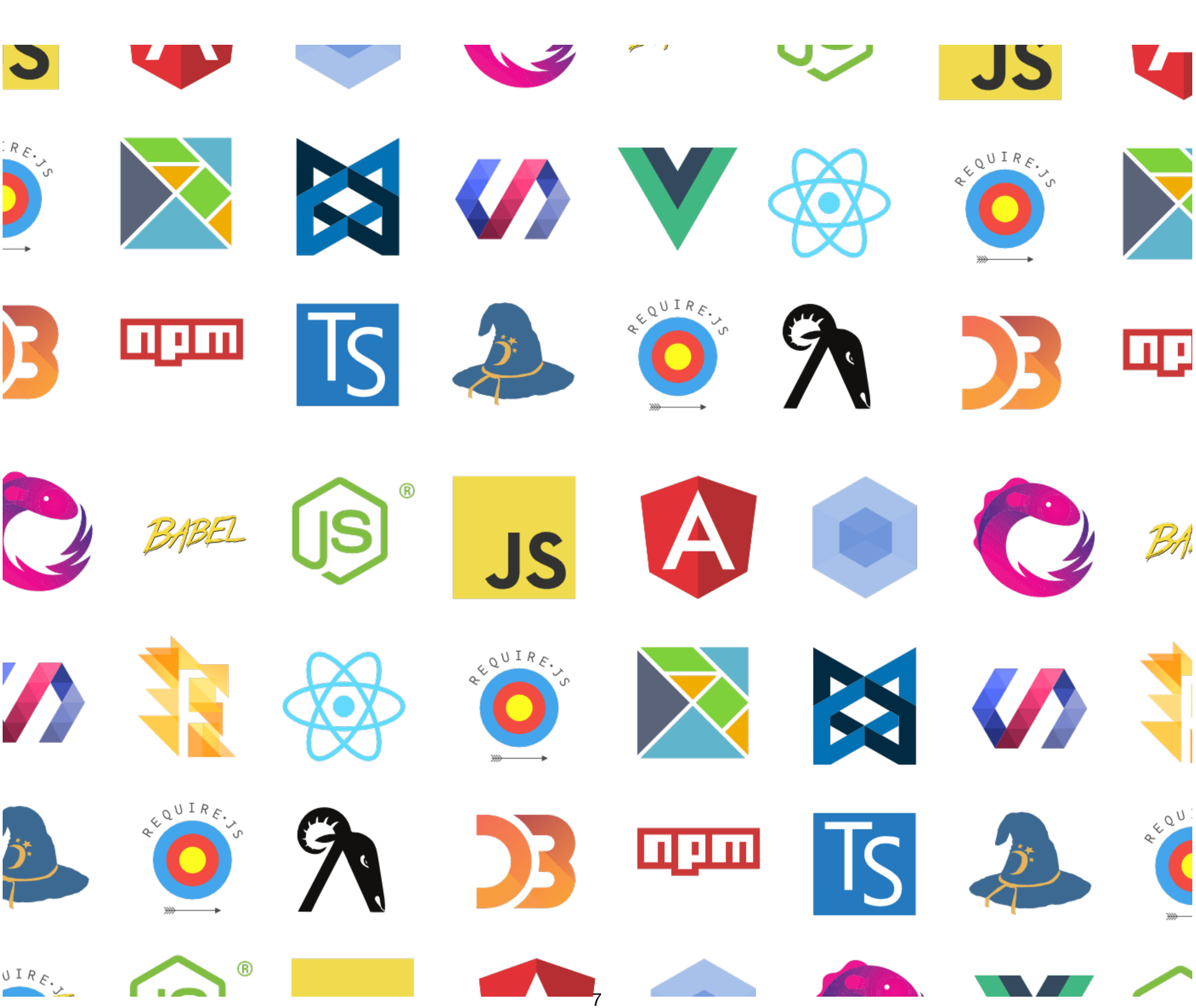


Background

JavaScript is confusing.



https://stackoverflow.com/questions/359494/which-equals-operator-vs-should-be-used-in-javascript-comparisons

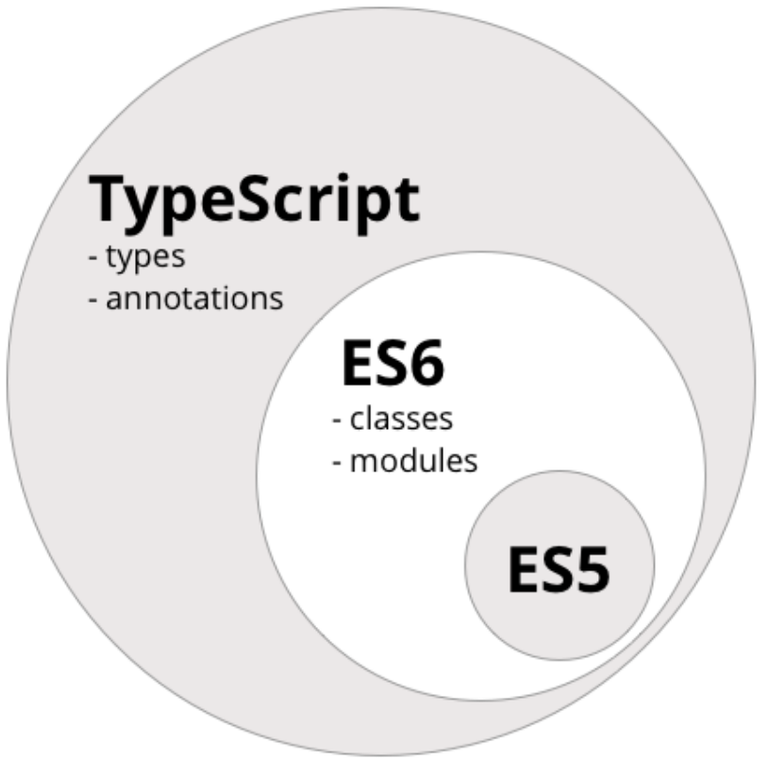

JS
REQUIRE·JS
npm
TS
BABEL
JS
REQUIRE·JS
npm
TS
REQUIRE·JS
JS

Background

TypeScript

JavaScript that scales.

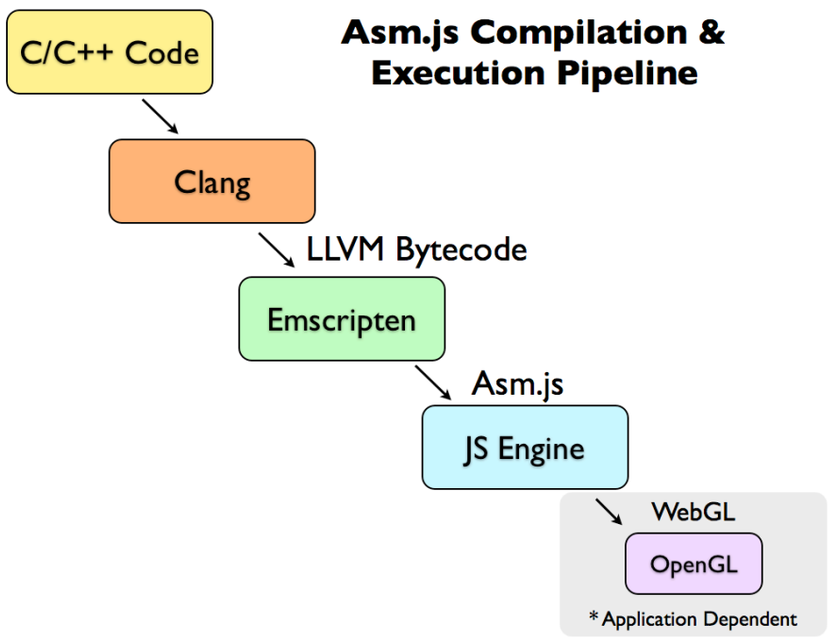
TypeScript is a **typed superset of JavaScript** that compiles to plain JavaScript.



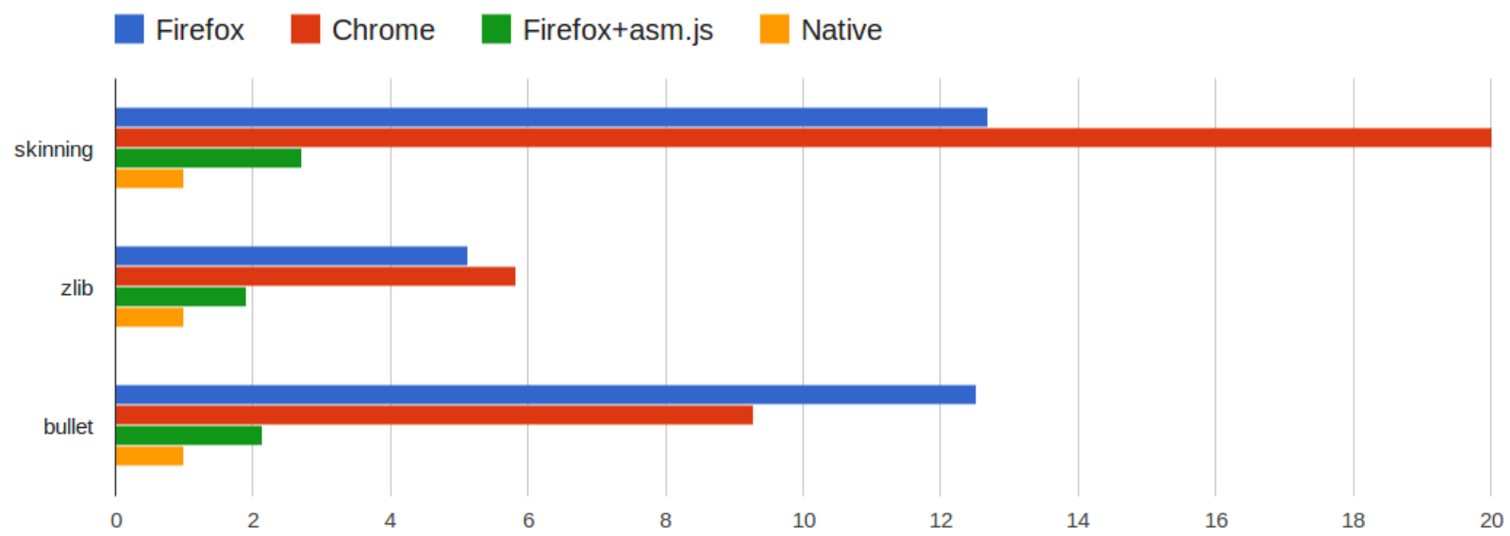
Background

asm.js

- asm.js is a **subset** of JavaScript
- allow other languages such as C to be run as web applications while maintaining performance characteristics considerably better than standard JavaScript,



https://johnresig.com/blog/asmjs-javascript-compile-target/



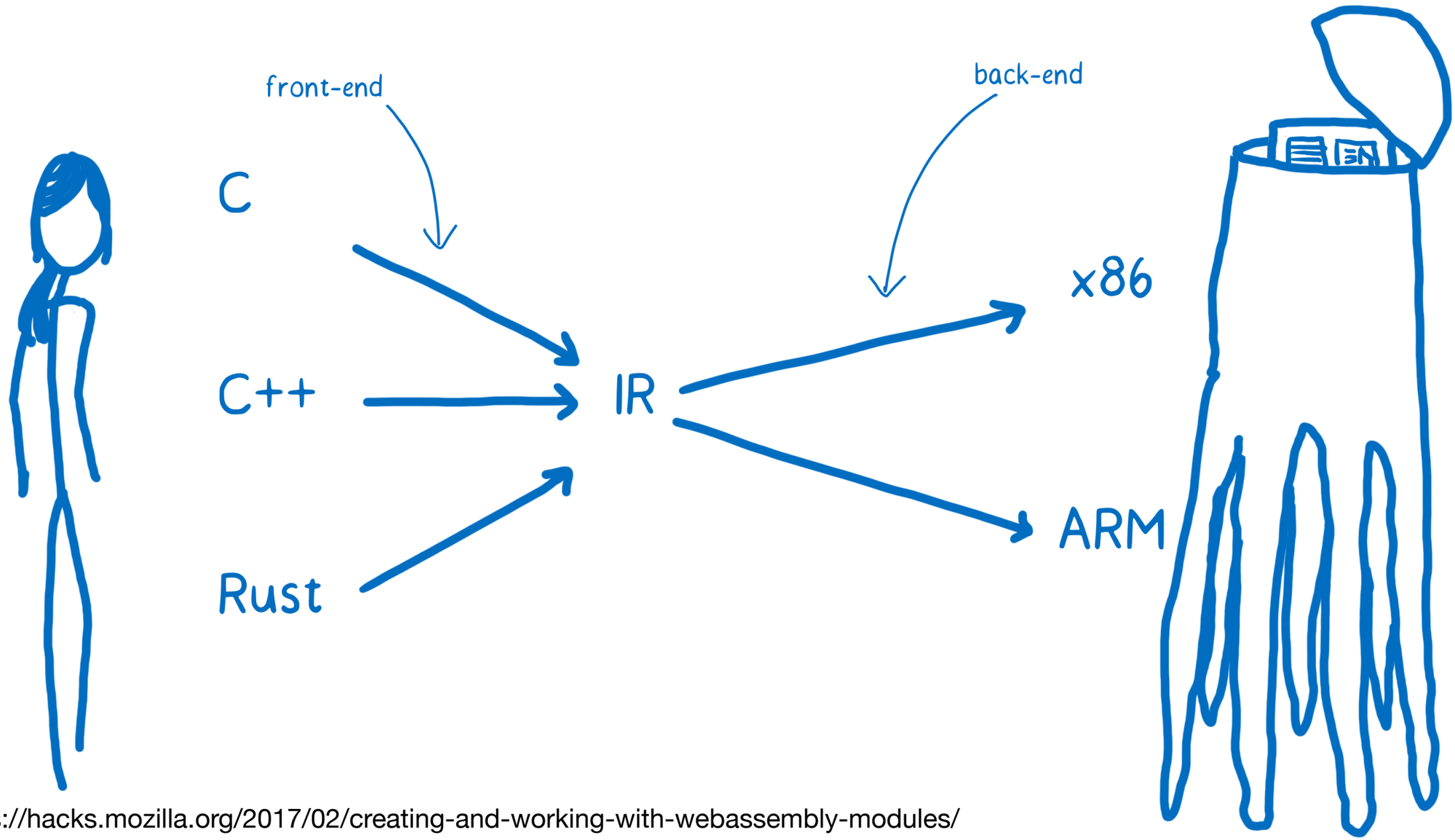
Introduction

WebAssembly to save the world?

- WebAssembly (abbreviated Wasm) is a safe, portable, low-level code format designed for efficient execution and compact representation.

Introduction

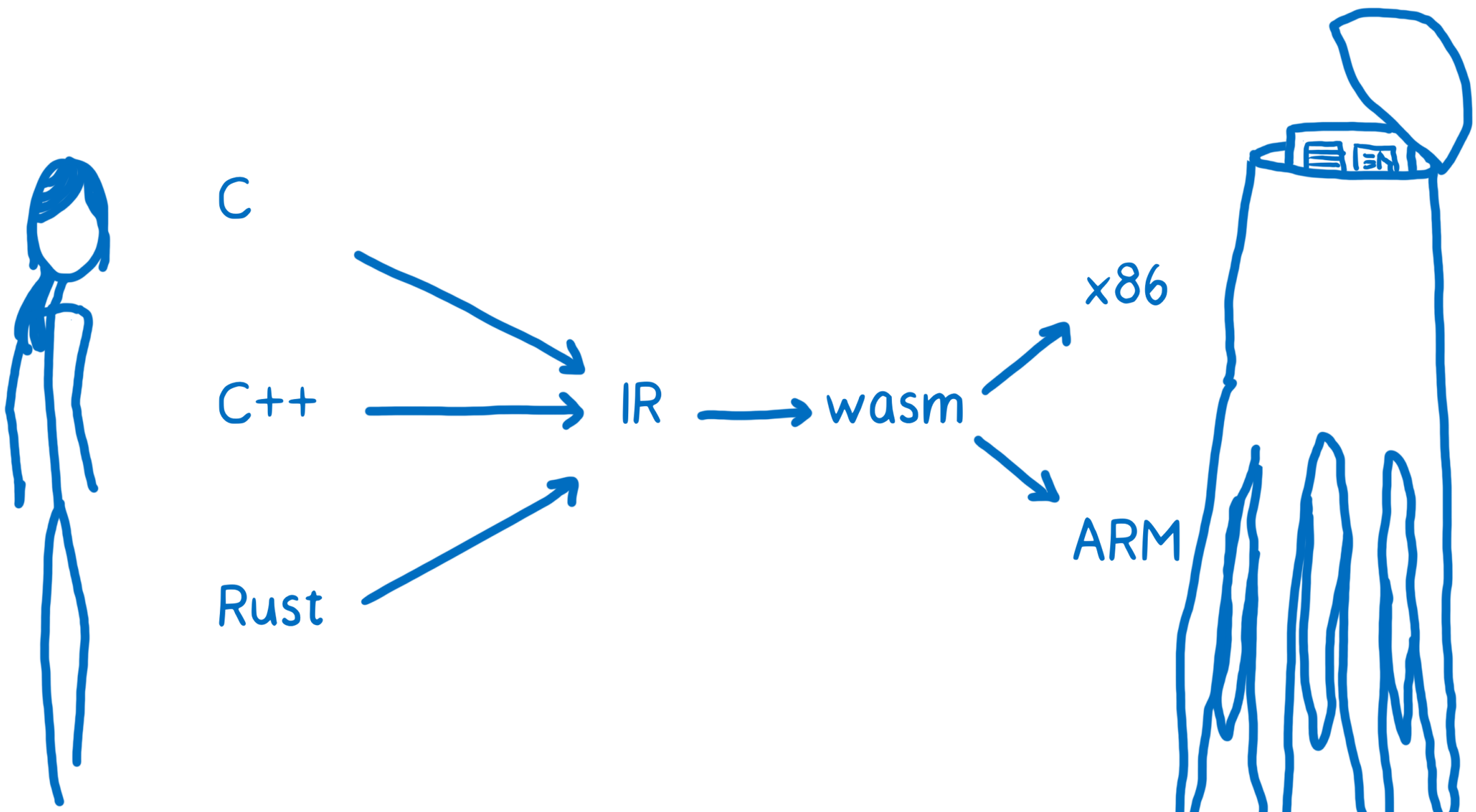
Where does WebAssembly Fit?



https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/

Introduction

Where does WebAssembly Fit?



https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/

wat/wasm

Binary representation (.wasm)

wat2wasm demo

WebAssembly has a text format and a binary format. This demo converts from the text format to the binary format.

Enter WebAssembly text in the textarea on the left. The right side will either show an error, or will show a log with a description of the generated binary file.

WAT example: simple Download

```
(module
  (func $addTwo (param i32 i32) (result i32)
    get_local 0
    get_local 1
    i32.add)
  (export "addTwo" (func $addTwo)))

```

BUILD LOG

```
0000000: 0061 736d                                 ; WASM_BINARY_MAGIC
0000004: 0100 0000                                 ; WASM_BINARY_VERSION
; section "Type" (1)
0000008: 01                                        ; section code
0000009: 00                                        ; section size (guess)
000000a: 01                                        ; num types
; type 0
000000b: 60                                        ; func
000000c: 02                                        ; num params
000000d: 7f                                        ; i32
000000e: 7f                                        ; i32
000000f: 01                                        ; num results
0000010: 7f                                        ; i32
0000009: 07                                        ; FIXUP section size
; section "Function" (3)
0000011: 03                                        ; section code
0000012: 00                                        ; section size (guess)
0000013: 01                                        ; num functions
0000014: 00                                        ; function 0 signature index
0000012: 02                                        ; FIXUP section size
; section "Export" (7)
0000015: 07                                        ; section code
0000016: 00                                        ; section size (guess)
```

Textual representation (.wat)

JS

```
const wasmInstance =
      new WebAssembly.Instance(wasmModule, {});
const { addTwo } = wasmInstance.exports;
for (let i = 0; i < 10; i++) {
  console.log(addTwo(i, i));
}

```

JS LOG

```
0
2
4
6
8
10
```

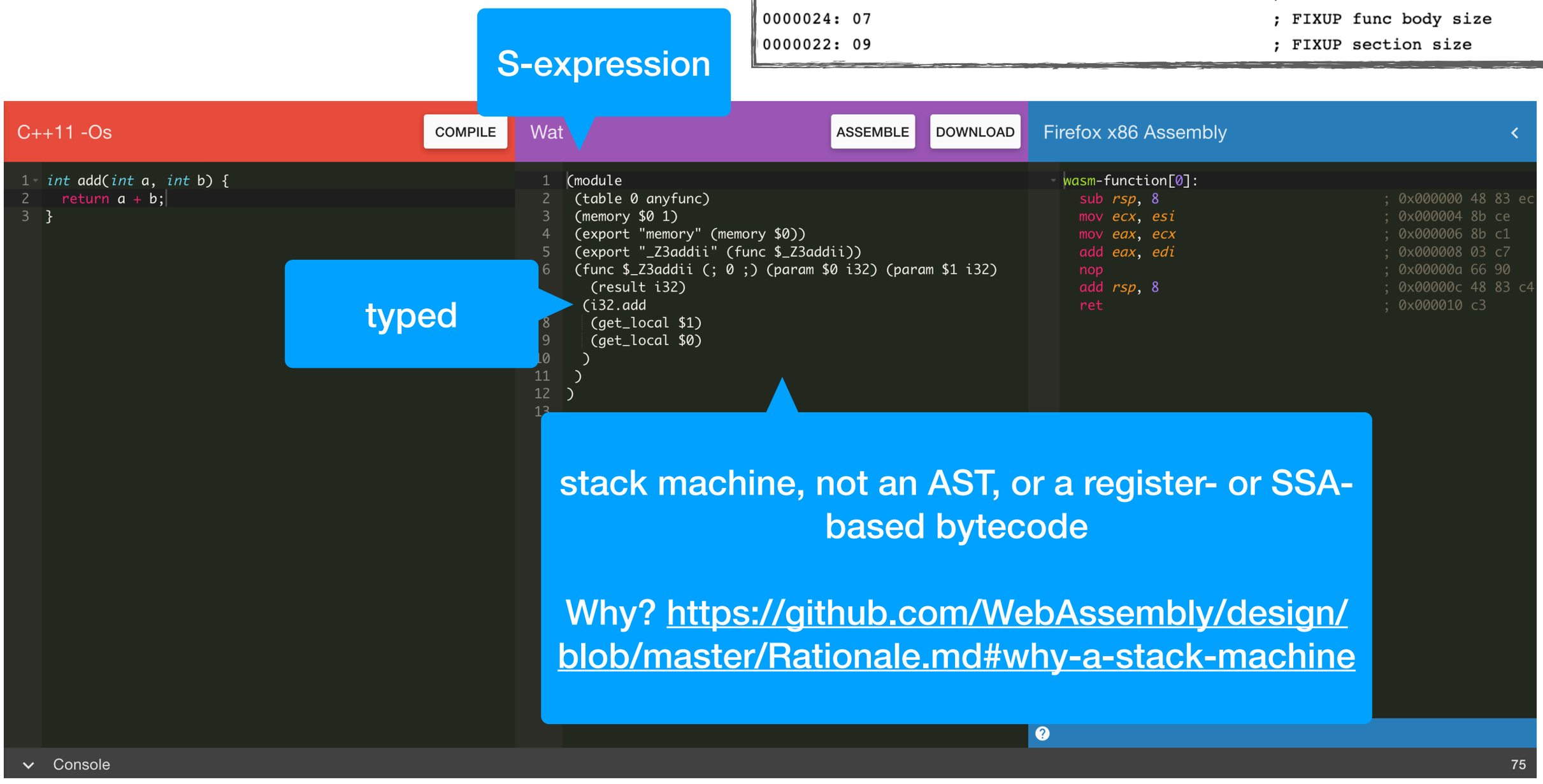
JS APIs

JavaScript APIs

```
WebAssembly.compile(new Uint8Array(
  `00 61 73 6d  0d 00 00 00  01 09 02 60  00 00 60 01
7f 01 7f 03  03 02 00 01  05 04 01 00  80 01 07 07
01 03 66 6f  6f 00 01 08  01 00 0a 3a  02 02 00 0b
35 01 01 7f  20 00 41 04  6c 21 01 03  40 01 01 01
0b 03 7f 41  01 0b 20 01  41 e4 00 6c  41 cd 02 20
01 1b 21 01  41 00 20 01  36 02 00 41  00 21 01 41
00 28 02 00  0f 0b 0b 0e  01 00 41 00  0b 08 00 00
00 00 2c 00  00 00`.split(/[\s\r\n]+/g).map(v => parseInt(v, 16))
)).then(mod => {
  let m = new WebAssembly.Instance(mod)
  console.log('foo(1) =>', m.exports.foo(1))
  console.log('foo(2) =>', m.exports.foo(2))
  console.log('foo(3) =>', m.exports.foo(3))
})
```

wat/wasm

```
; function body 0
0000024: 00                                        ; func body size (guess)
0000025: 00                                        ; local decl count
0000026: 20                                        ; local.get
0000027: 00                                        ; local index
0000028: 20                                        ; local.get
0000029: 01                                        ; local index
000002a: 6a                                        ; i32.add
000002b: 0b                                        ; end
0000024: 07                                        ; FIXUP func body size
0000022: 09                                        ; FIXUP section size
```



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

Wasm

Anatomy of a WebAssembly program

module “example1”

version	1

type section

#0	(i32, i32) → (i32)
#1	() → ()

import section

#0	“half” from “example2” of type 0

function section

#1	type 1
#2	type 0

export section

“double” of type 0

start section

function #0

code section

#0	i32.const 2 i32.load 2 0 0 call 1
#1	load_local 0 load_local 1 i32.mul

data section

0x5 0x0 0x0 0x0

•••

Wasm

Anatomy of a WebAssembly program

module “example1”

version	1

type section

#0	
#1	

import section

#0	“half” f

function section

#1	
#2	

export section

“double” of type 0

start section

#0

data section

0x5 0x0 0x0 0x0

Types

WebAssembly has the following *value types*:

- `i32` : 32-bit integer
- `i64` : 64-bit integer
- `f32` : 32-bit floating point
- `f64` : 64-bit floating point

Wasm

Anatomy o

module “example1”

version	1

type section

#0	(i3
#1	

import section

#0	“half” fro

function section

#1	
#2	

export section

“double

A module contains the following sections:

- import
- export
- start
- global
- memory
- data
- table
- elements
- function and code

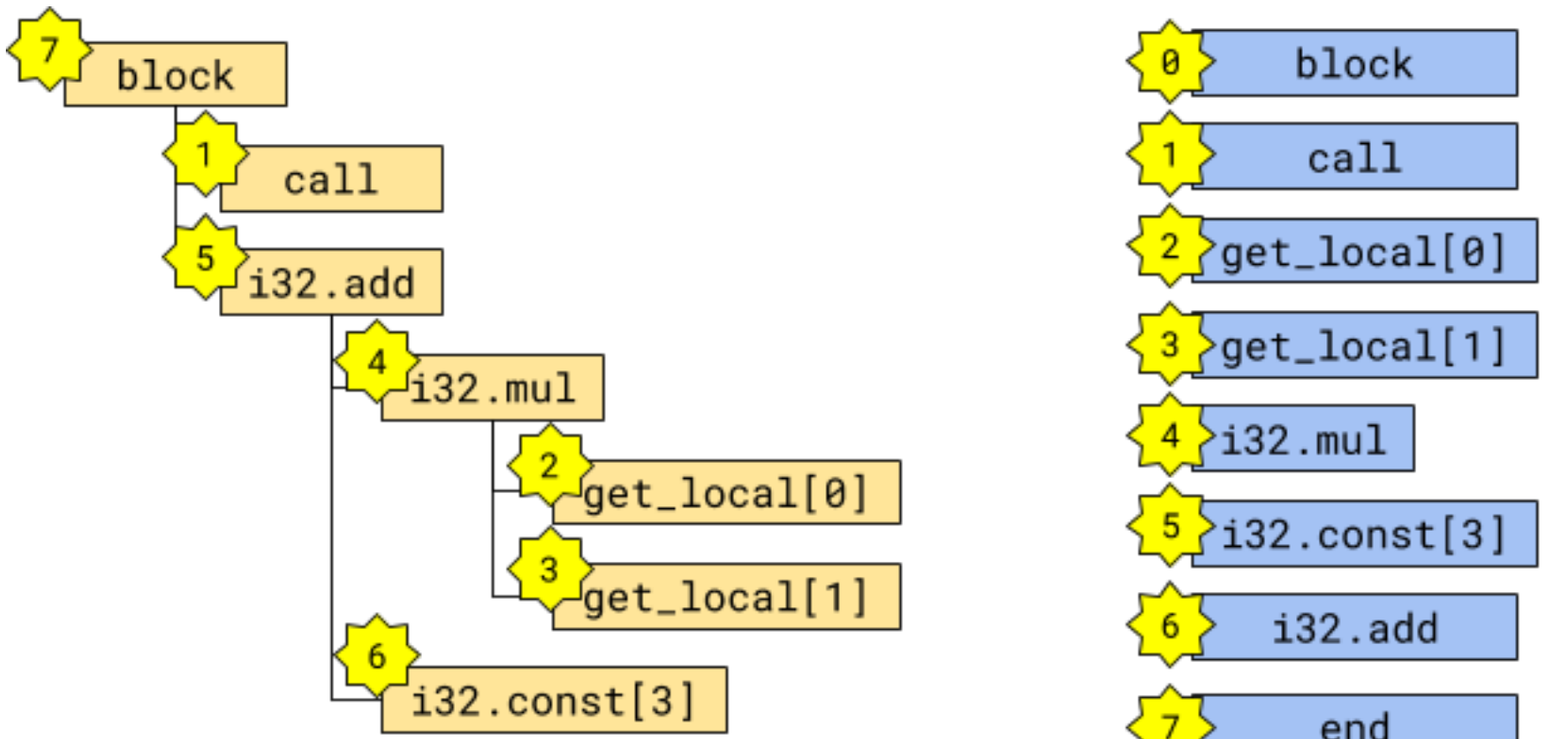
A module also defines several *index spaces* which ar
section fields in the module:

- the function index space
- the global index space
- the linear memory index space
- the table index space

x0

...

Wasm Stack Machine



Abstract Syntax Tree

Postorder Encoding

Wasm Memory Model

Linear memory mordel

- Memory is linear
- memory-page size: 64KiB

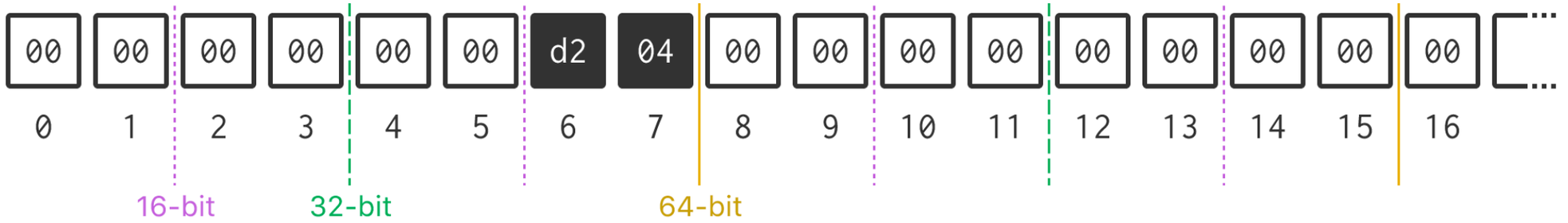
Memory being “linear” means that there’s no random allocator operators available — all memory addresses used in a module’s code are expressed in terms of byte offsets from the beginning of a memory segment.

WASM being a low-level format, this makes a lot of sense. It’s up to the higher-level language that targets WASM to provide memory management on top of this linear memory space, if needed.

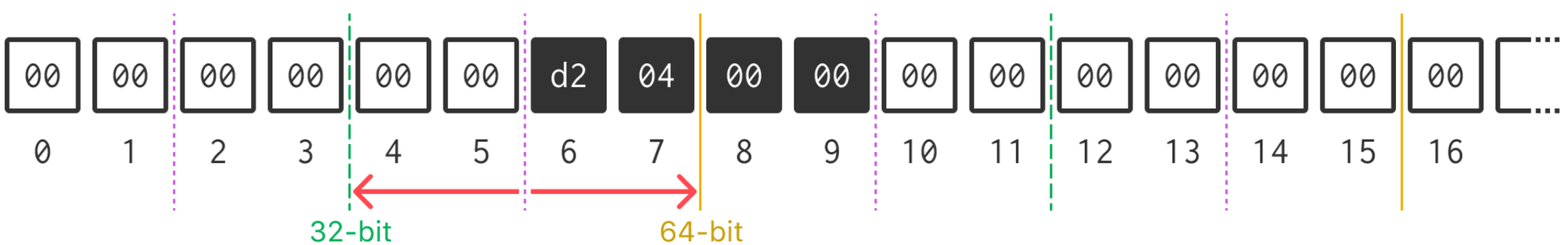
Wasm Memory Model

Load/Store

```
i32.const 3       // address-operand = 3
i64.const 1234    // value
i64.store16 1 3   // alignment = 1 = 16-bit, offset-immediate = 3
                  // effective-address = 3 + 3 = 6
```

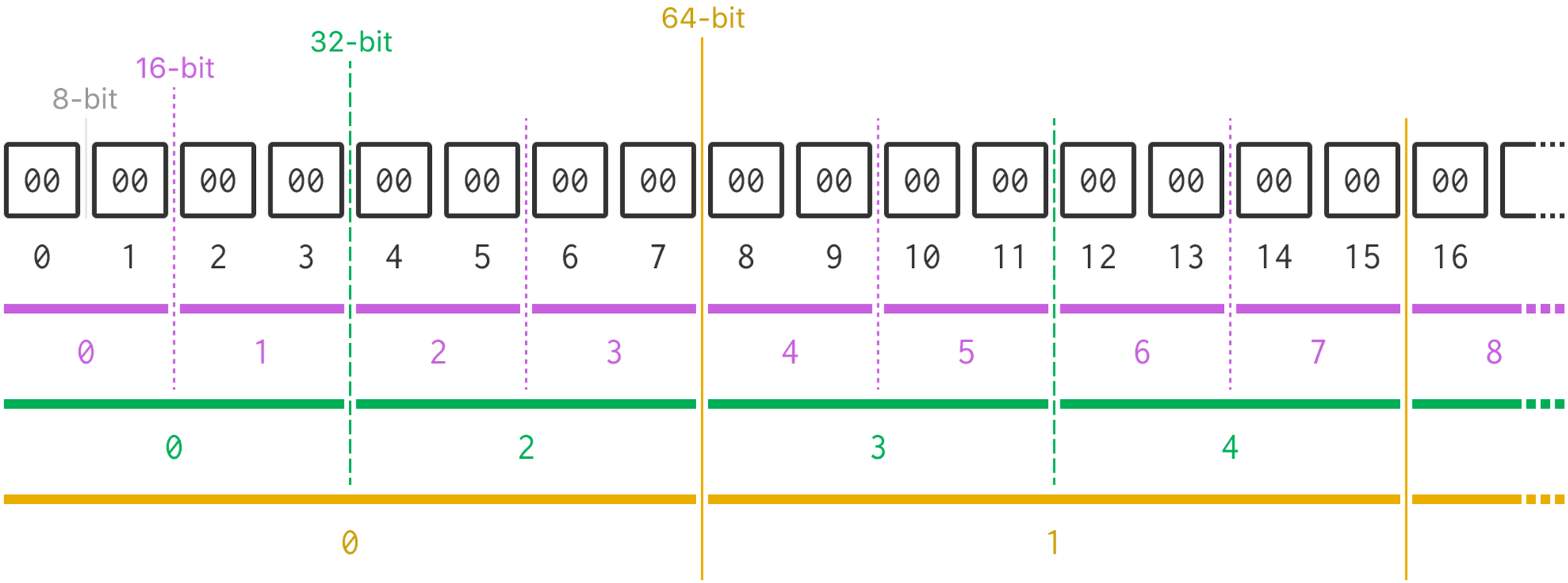



```
i64.store32 2 3  // align = 2 = 32-bit, offset = 3  =>  addr = 6
```



Wasm Memory Model

Load/Store



The **effective address** is the offset in bytes measured from the beginning of the memory. The effective address is the sum of the address operand and the offset immediate.

```
effective-address = address-operand + offset-immediate
```

https://rsms.me/wasm-intro

WASI

a system interface for WebAssembly to run outside the web

- POSIX (The Portable Operating System Interface)
  - Glibc, musl libc
  - MSVC
- CloudABI
- Android

WASI

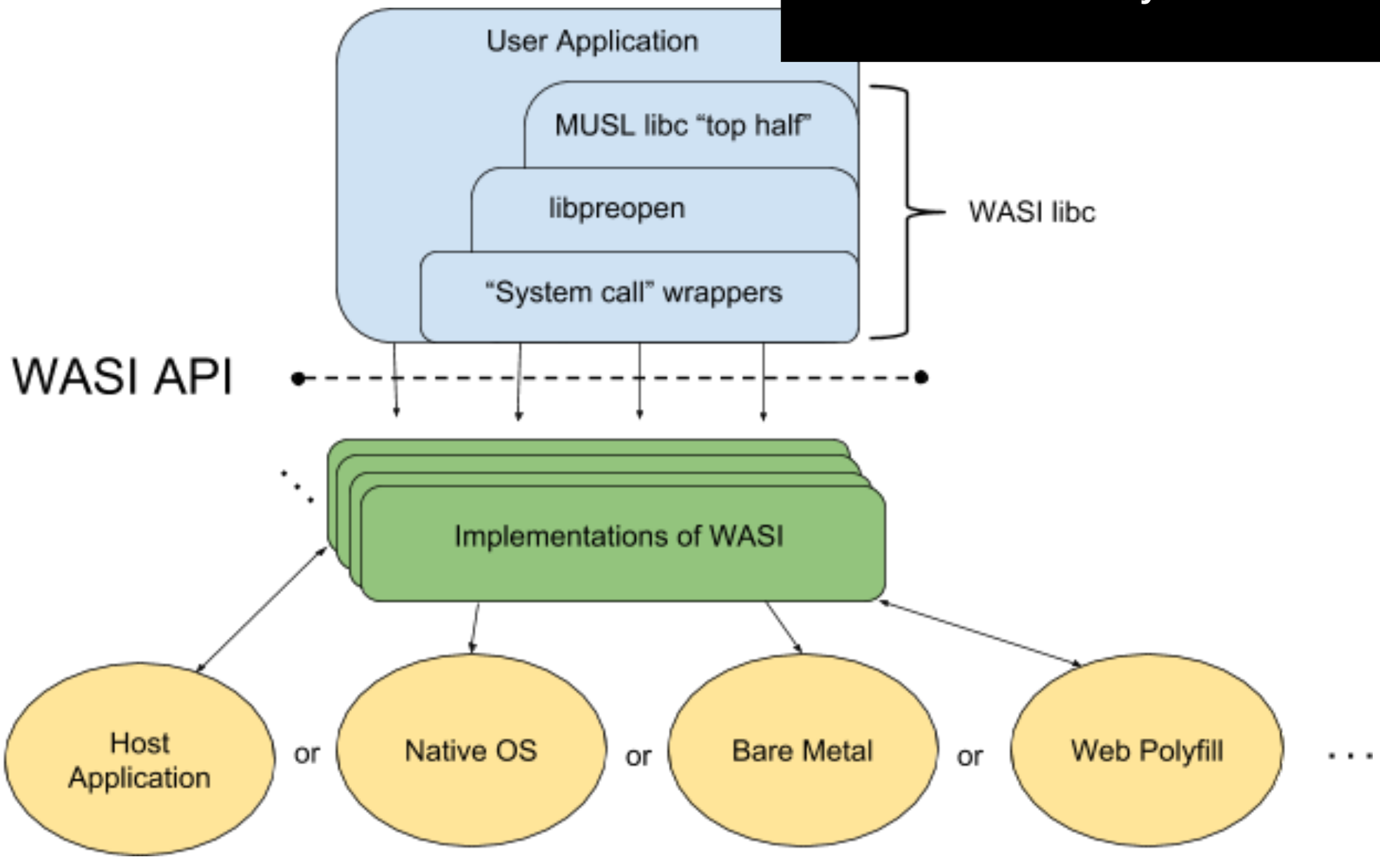
a system interface for WebAssembly to run outside the web

- POSIX (The Portable Operating System Interface)
- Capability-based security means that processes can only perform actions that have no global impact. Processes cannot open files by their absolute path, cannot open network connections, and cannot observe global system state such as the process table.

CloudABI is what you get if you take POSIX, add capability-based security, and remove everything that's incompatible with that. The result is a minimal ABI consisting of only 49 syscalls.

WASI API

For example, instead of a typical open system call, WASI provides an openat-like system call, requiring the calling process to have a file descriptor for a directory that contains the file, representing the capability to open files within that directory.



Example

Hello, World!

```rust
fn main() {
    println!("Hello, world!");
}
```

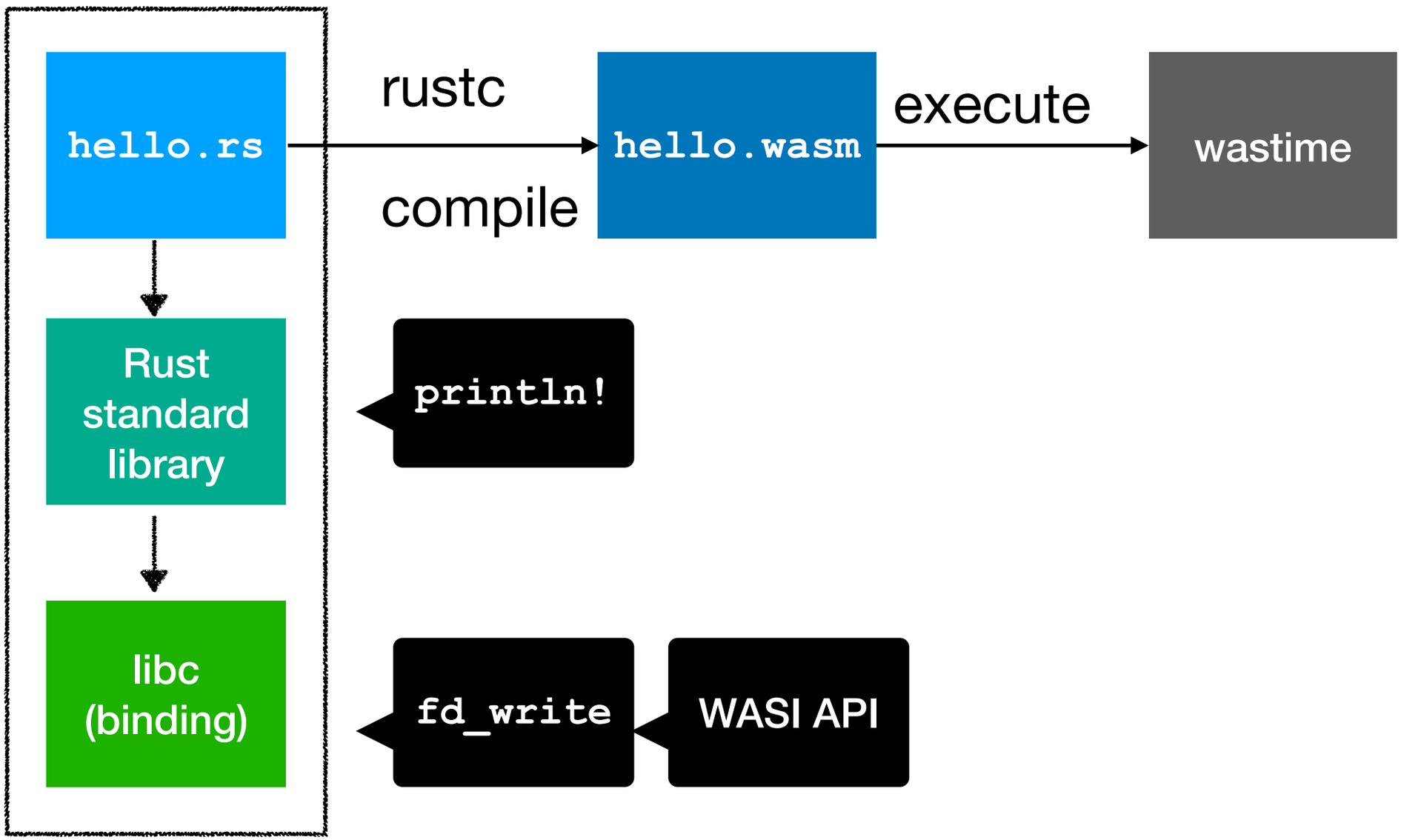
```
$ rustup target add wasm32-wasi
$ rustc hello.rs --target wasm32-wasi
$ wasmtime hello.wasm
Hello, world!
```

Example

Hello, World

Example

Hello, World



Example

Hello, World!

`fd_write(fd: fd, iovs: ciovec_array) -> (errno, size)`

Write to a file descriptor. Note: This is similar to `writev` in POSIX.

Params

- `fd`: `fd`
- `iovs`: `ciovec_array` List of scatter/gather vectors from which to retrieve data.

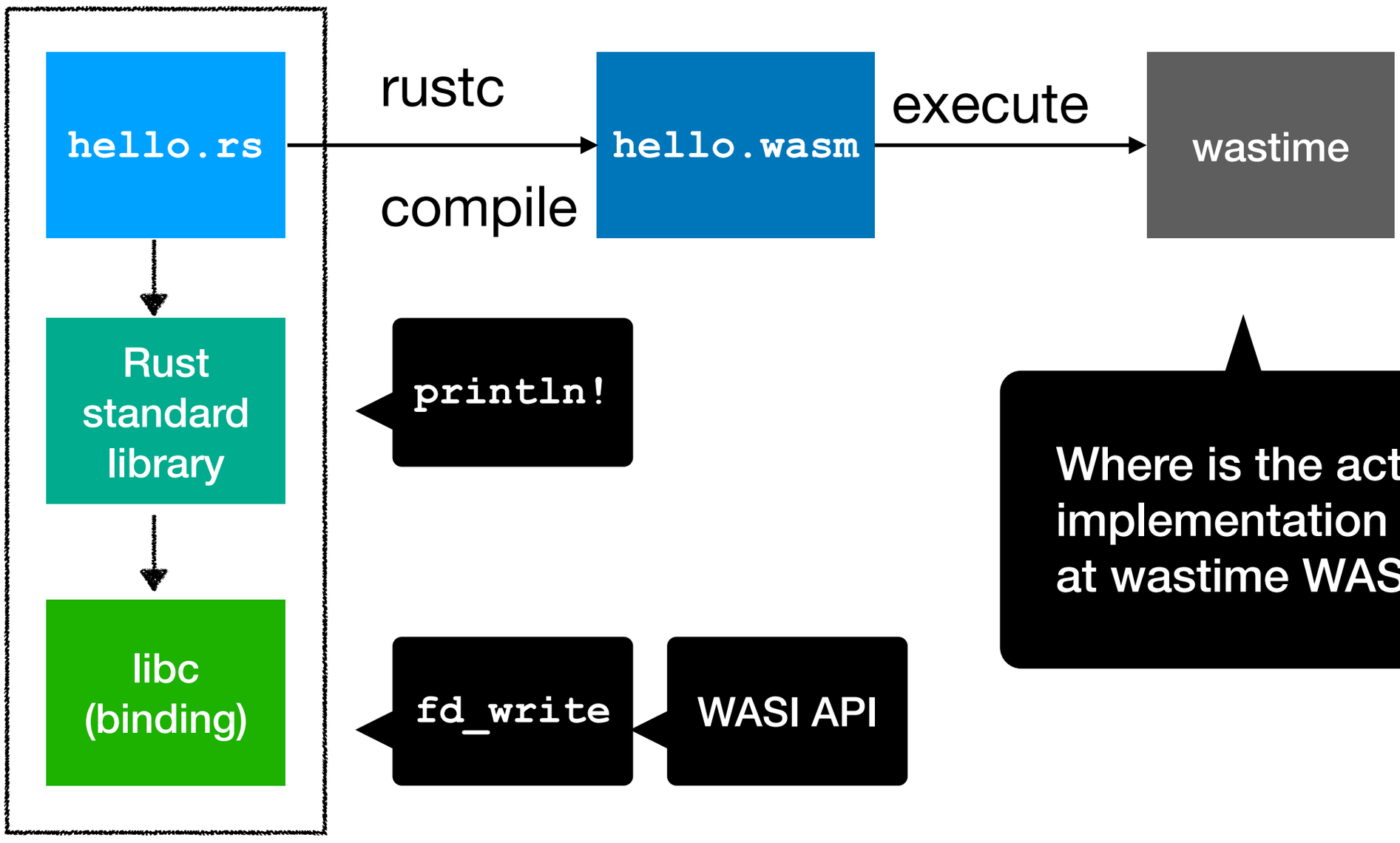
Results

- `error`: `errno`
- `nwritten`: `size` The number of bytes written.

```
(module
  (type $t0 (func))
  (type $t1 (func (param i32)))
  (type $t2 (func (param i32) (result i64)))
  (type $t3 (func (param i32 i32) (result i32)))
  (type $t4 (func (param i32 i32 i32 i32)))
  (type $t5 (func (param i32 i32)))
  (type $t6 (func (param i32 i32 i32)))
  (type $t7 (func (param i32) (result i32)))
  (type $t8 (func (param i32 i32 i32) (result i32)))
  (type $t9 (func (param i32 i32 i32 i32) (result i32)))
  (type $t10 (func (result i32)))
  (type $t11 (func (param i32 i32 i32 i32 i32)))
  (type $t12 (func (param i32 i32 i32 i32 i32) (result i32)))
  (type $t13 (func (param i32 i32 i32 i32 i32 i32) (result i32)))
  (type $t14 (func (param i64 i32 i32) (result i32)))
  (import "wasi_snapshot_preview1" "proc_exit" (func $__wasi_proc_exit (type $t1)))
  (import "wasi_snapshot_preview1" "fd_write" (func $_ZN4wasi13lib_generated22wasi_snapshot_preview18fd_write17h647699597ede499dE (type $t9)))
  (import "wasi_snapshot_preview1" "fd_prestat_get" (func $__wasi_fd_prestat_get (type $t3)))
  (import "wasi_snapshot_preview1" "fd_prestat_dir_name" (func $__wasi_fd_prestat_dir_name (type $t8)))
  (import "wasi_snapshot_preview1" "environ_sizes_get" (func $__wasi_environ_sizes_get (type $t3)))
  (import "wasi_snapshot_preview1" "environ_get" (func $__wasi_environ_get (type $t3)))
  (func $__wasm_call_ctors (type $t0)
    (call $__wasilibc_populate_environ)
    (call $__wasilibc_populate_libpreopen))
```

Example

Hello, World



Example

Hello, World

```
424     fn fd_write(&self, fd: types::Fd, ciovs: &types::CiovecArray<'_>) -> Result<types::Size> {
425         let mut bc = GuestBorrows::new();
426         let mut slices = Vec::new();
427         bc.borrow_slice(&ciovs)?;
428         for ciov_ptr in ciovs.iter() {
429             let ciov_ptr = ciov_ptr?;
430             let ciov: types::Ciovec = ciov_ptr.read()?;
431             let slice = unsafe {
432                 let buf = ciov.buf.as_array(ciov.buf_len);
433                 let raw = buf.as_raw(&mut bc)?;
434                 &*raw
435             };
436             slices.push(io::IoSlice::new(slice));
437         }
438         let required_rights = EntryRights::from_base(types::Rights::FD_WRITE);
439         let entry = self.get_entry(fd)?;
440         let isatty = entry.isatty();
441         let host_nwritten = entry
442             .as_handle(&required_rights)?
443             .write_vectored(&slices, isatty)?
444             .try_into()?;
445         Ok(host_nwritten)
446     }
```

OsHandle

io::stdout();

/crates/wasi-common/src/snapshots/wasi_snapshot_preview1.rs

XXX to/in Wasm

- Rust: rustc
- C/C++
  - Emscripten - an LLVM-to-JavaScript/Webassembly compiler. It takes LLVM bitcode - which can be generated from C/C++, using llvm-gcc (DragonEgg) or clang, or any other language that can be converted into LLVM - and compiles that into JavaScript or wasm.
  - Cheerp - an open-source, commercial C/C++ compiler for Web applications. It can compile virtually any C/C++ code (up to C++14) to WebAssembly, JavaScript, asm.js or a combination thereof.
- Python
  - Pyodide - a port of Python to WebAssembly that includes the core packages of the scientific Python stack (Numpy, Pandas, matplotlib). Objects transparently convert and share between Python and Javascript.
  - MicroPython - a lean and efficient Python implementation for microcontrollers and constrained systems.
  - Batavia - Batavia is an implementation of the Python virtual machine, written in JavaScript. With Batavia, you can run Python bytecode in your browser.

Wasm Runtime

- V8: https://chromium.googlesource.com/v8/v8/+/refs/heads/master/src/wasm/
- Wastime: https://github.com/bytecodealliance/wasmtime
- Lucet: https://github.com/bytecodealliance/lucet
- wasm-micro-runtime: https://github.com/bytecodealliance/wasm-micro-runtime
- Wasmer: https://github.com/wasmerio/wasmer
- Wasmi: https://github.com/paritytech/wasmi

Security

- WebAssembly design doc: https://webassembly.org/docs/security/
- Users
  - Protect *users* from buggy or malicious modules.
- Developers
  - Provide *developers* with useful primitives and mitigations for developing safe applications, within the constraints of above one.

Wasm Security

Users

- **Sandbox** (with fault isolation)
- **Deterministic execution** (with limited exceptions)
- Each module is subject to the security policies of its embedding: same-origin policy or POSIX

Wasm Security

Developers

- **Modules**
  - explicit function declaration allow CFI enforcement
  - code is immutable and not observable at runtime

- Function calls must specify the index of a target that corresponds to a valid entry in the function index space or table index space.
- Indirect function calls are subject to a type signature check at runtime; the type signature of the selected indirect function must match the type signature specified at the call site.
- A protected call stack that is invulnerable to buffer overflows in the module heap ensures safe function returns.
- Branches must point to valid destinations within the enclosing function.

Wasm Security

Developers

- **Varibles**
  - **local variables**: zero by default, protected call stack
  - **global variables**: global index space
  - **unclear static scope** (address-of op): separate user-addressable stack linear memory, zero by default
    - References to this memory are computed with infinite precision to avoid wrapping and simplify bounds checking

Wasm Security

Developers

- **Traps**: terminate execution and signal abnormal behavior to the execution environment

- specifying an invalid index in any index space,
- performing an indirect function call with a mismatched signature,
- exceeding the maximum size of the protected call stack,
- accessing out-of-bounds addresses in linear memory,
- executing an illegal arithmetic operations (e.g. division or remainder by zero, signed division overflow, etc).

Wasm Security

Developers

- Memory safety (overflows)
  - Buffer overflows **cannot affect local or global variables** stored in index space, they are fixed-size and addressed by index.
  - Data stored in linear memory **can overwrite adjacent objects**, since bounds checking is performed at linear memory region granularity and is not context-sensitive.
  - However, the presence of control-flow integrity and **protected call stacks prevents direct code injection attacks**. Thus, common mitigations such as data execution prevention (DEP) and stack smashing protection (SSP) are not needed.

Wasm Security

Developers

- Memory safety (use-after-free)
  - the semantics of **pointers have been eliminated** for function calls and variables with fixed static scope, allowing references to invalid indexes in any index space to trigger a validation error at load time, or at worst a trap at runtime.
  - Accesses to **linear memory are bounds-checked** at the region level, potentially resulting in a **trap** at runtime. These memory region(s) are isolated from the internal memory of the runtime, and are set to zero by default unless otherwise initialized.

Wasm Security

Developers

- **Memory safety (others)**
  - possible to hijack the control flow of a module using code reuse attacks against **indirect calls**
  - **ROP attacks are not possible, because control-flow integrity ensures that call targets are valid functions declared at load time.**
  - **race conditions**, such as time of check to time of use (TOCTOU) vulnerabilities, are possible, since no execution or scheduling guarantees are provided beyond in-order execution and atomic memory primitives
  - **side channel attacks**
  - In the future, additional protections may be provided by runtimes or the toolchain, such as code diversification or memory randomization (ASLR), or bounded pointers (“fat” pointers).

Wasm Security

Developers

- **Control-flow integrity**
  - Runtime instrumentation: During compilation, the compiler generates an expected **control flow graph** of program execution, and **inserts runtime instrumentation** at each call site to verify that the transition is safe.
  - Sets of **expected call targets** are constructed from the set of all possible call targets in the program, unique identifiers are assigned to each set, and the instrumentation checks whether the current call target is a member of the expected call target set.
  - If this check succeeds, then the original call is allowed to proceed, otherwise a failure handler is executed, which typically terminates the program.

Wasm Security

Developers

1. Direct function calls,
2. Indirect function calls,
3. Returns.

- **Control-flow integrity**
  - In WebAssembly, the execution semantics implicitly guarantee the safety of (1) through usage of **explicit function section indexes**, and (3) through a **protected call stack**. Additionally, the **type signature** of indirect function calls is already checked at runtime, effectively implementing coarse-grained type-based control-flow integrity for (2). All of this is achieved without **explicit runtime instrumentation** in the module.

  - However, as discussed previously, this protection **does not prevent code reuse attacks** with function-level granularity against indirect calls.

Wasm Security

Developers

1. Direct function calls,
2. Indirect function calls,
3. Returns.

- **Control-flow integrity**
  - Clang/LLVM **`-fsanitize=cfgi`**
  - better defend against code reuse attacks that leverage indirect function calls (2)
  - enhances the **built-in function signature checks** by operating at the C/C++ type level, which is semantically richer that the WebAssembly type level, which consists of only four value types.
  - Currently, enabling this feature has a **small performance cost** for each indirect call, because an integer range check is used to verify that the target index is trusted, but this will be eliminated in the future by leveraging built-in support for multiple indirect tables with homogeneous type in WebAssembly.

Reference

- Frontend revolution: https://github.com/ManzDev/frontend-evolution
- State of JS: https://2019.stateofjs.com/overview/
- Creating and working with WebAssembly modules: https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/
- https://github.com/mbasso/awesome-wasm
- Bringing the Web up to Speed with WebAssembly: https://people.mpi-sws.org/~rossberg/papers/Haas,%20Rossberg,%20Schuff,%20Titzer,%20Gohman,%20Wagner,%20Zakai,%20Bastien,%20Holman%20-%20Bringing%20the%20Web%20up%20to%20Speed%20with%20WebAssembly.pdf
- Awesome WebAssembly Languages: https://github.com/appcypher/awesome-wasm-langs
- Introduction to WebAssembly: https://rsms.me/wasm-intro
- WebAssembly Out of Bounds Trap Handling: https://docs.google.com/document/d/17y4kxuHFrVxAiuCP_FFtFA2HP5sNPsCD10KEx17Hz6M/edit#