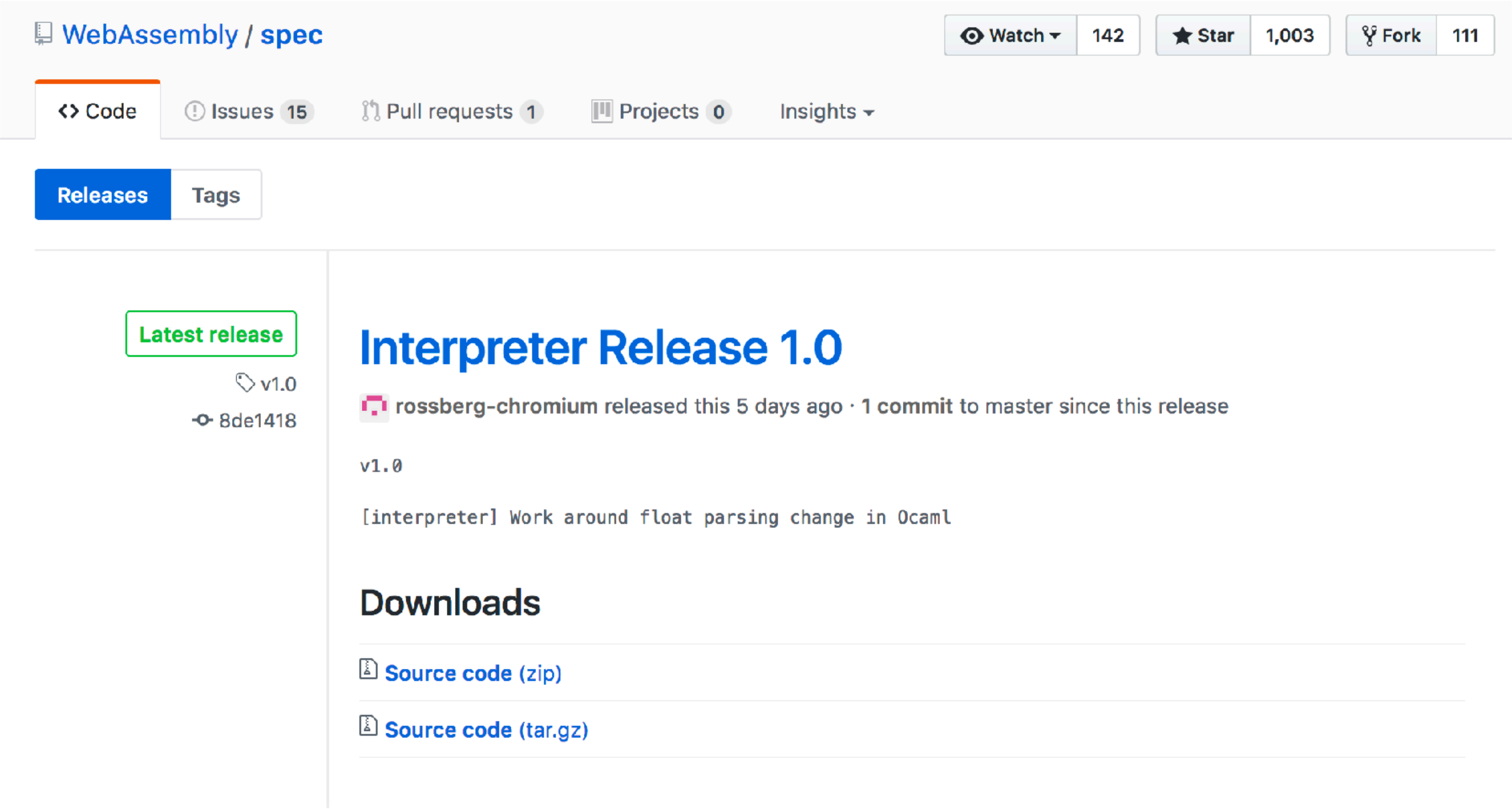
WASM1.0リリース記念（?） 最近の状況アップデート

@chikoski



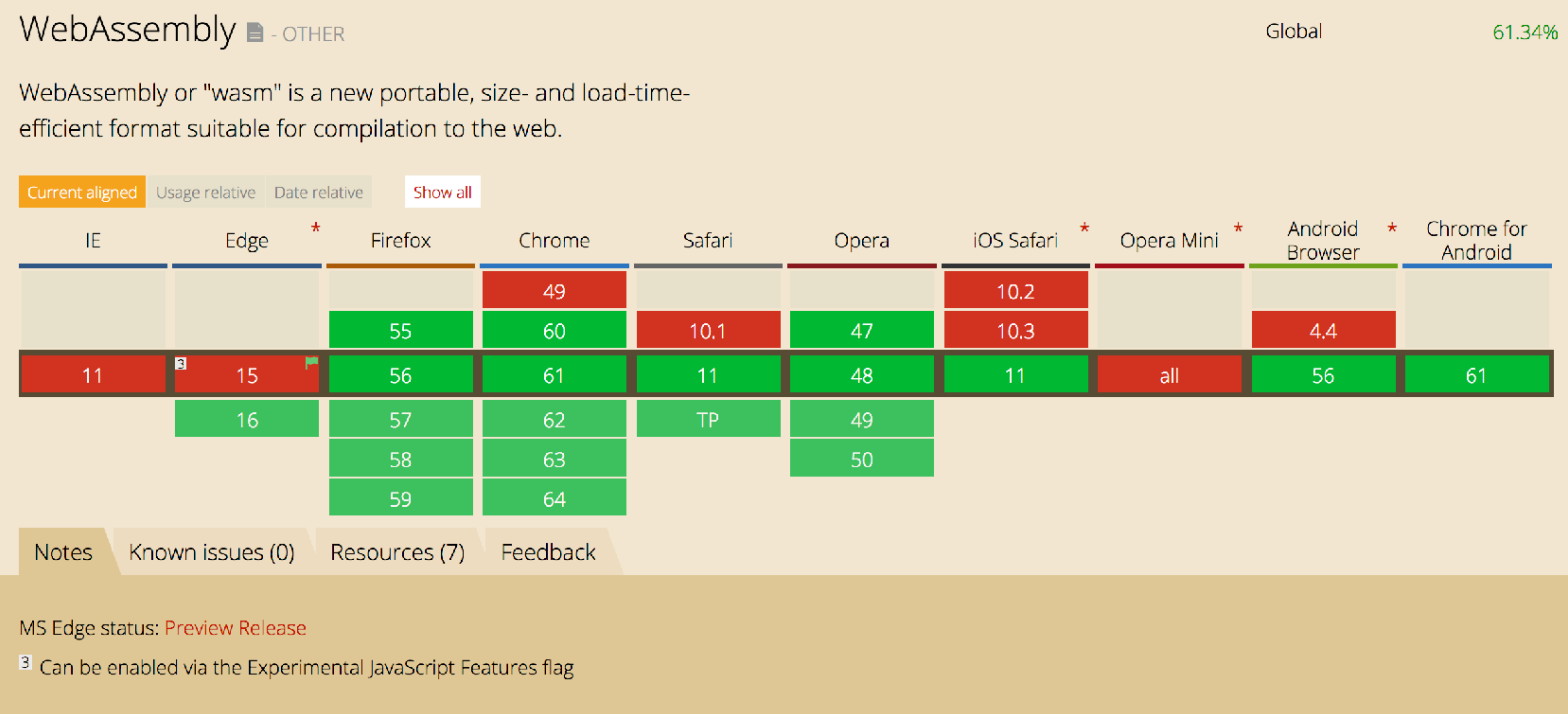
Interpreter Release 1.0

rossberg-chromium released this 5 days ago · **1 commit** to master since this release

```
v1.0

[interpreter] Work around float parsing change in Ocaml
```

Downloads

- **Source code** (zip)
- **Source code** (tar.gz)



WebAssembly - OTHER

Global 61.34%

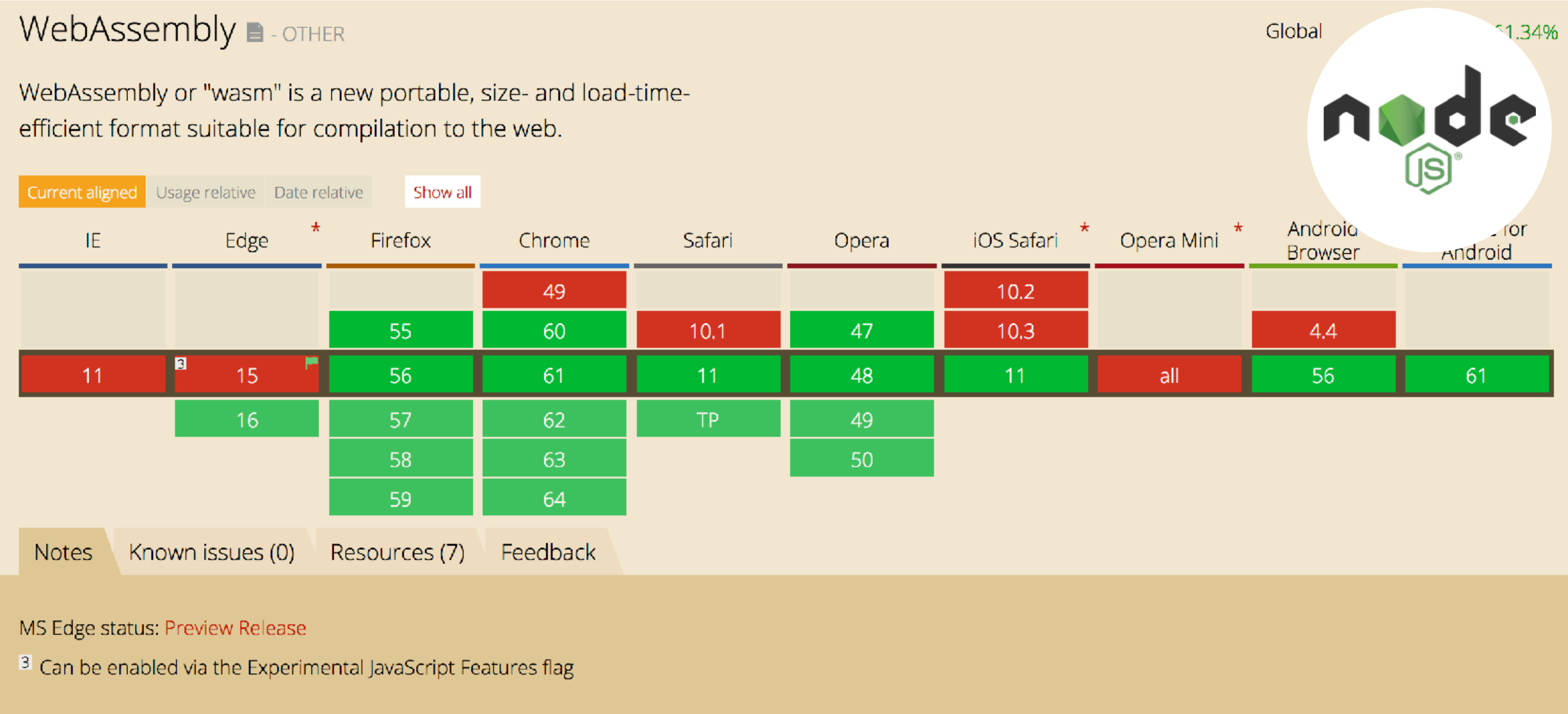
WebAssembly or "wasm" is a new portable, size- and load-time-efficient format suitable for compilation to the web.

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			49			10.2			
		55	60	10.1	47	10.3		4.4	
11	[3] 15	56	61	11	48	11	all	56	61
	16	57	62	TP	49				
		58	63		50				
		59	64						

MS Edge status: Preview Release

[3] Can be enabled via the Experimental JavaScript Features flag

https://caniuse.com/#feat=wasm



WebAssembly - OTHER

Global 1.34%

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

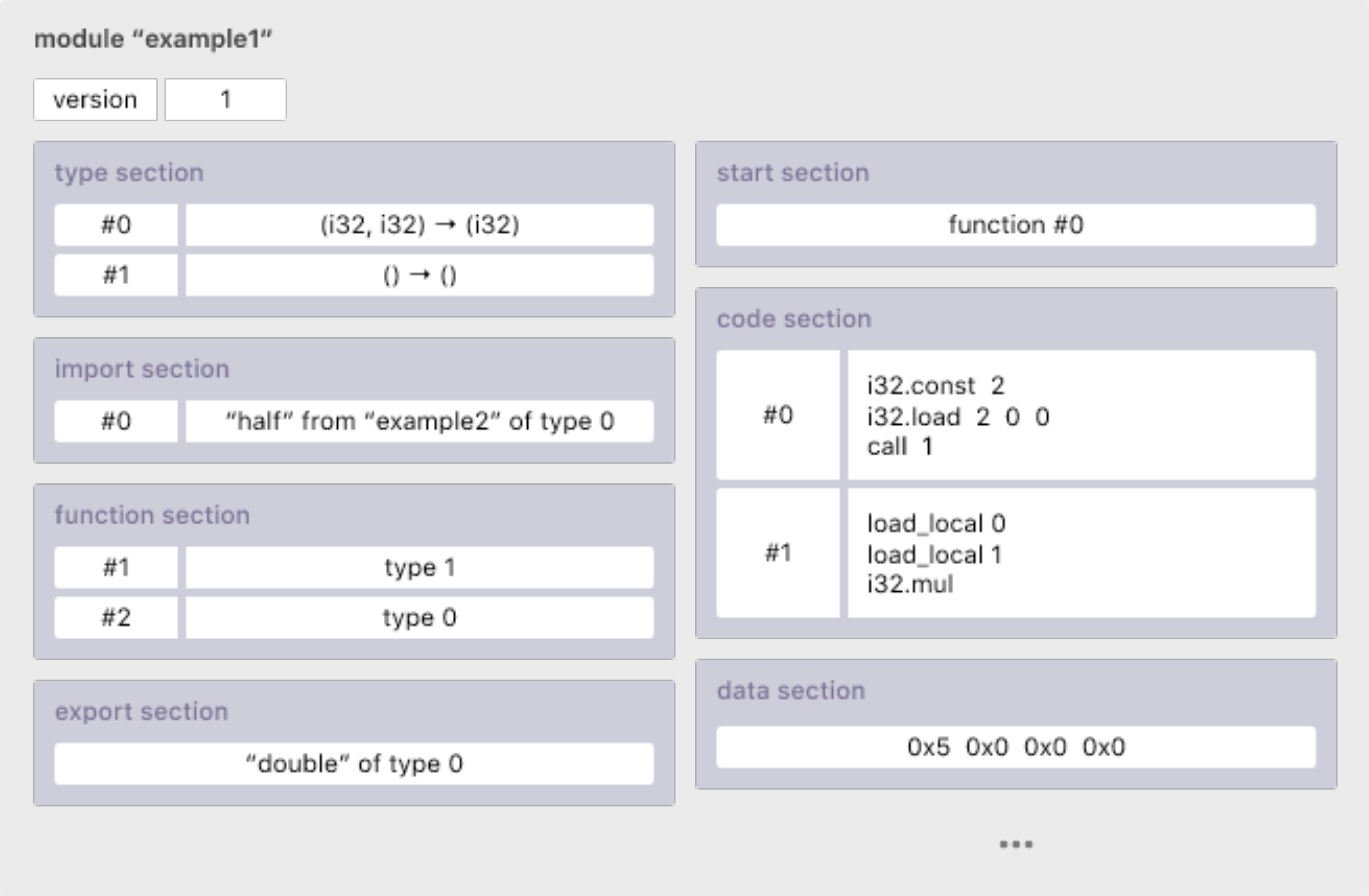
IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser	for Android
			49			10.2			
		55	60	10.1	47	10.3		4.4	
11	[3] 15	56	61	11	48	11	all	56	61
	16	57	62	TP	49				
		58	63		50				
		59	64						

MS Edge status: Preview Release

[3] Can be enabled via the Experimental JavaScript Features flag

https://caniuse.com/#feat=wasm

Module definition



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

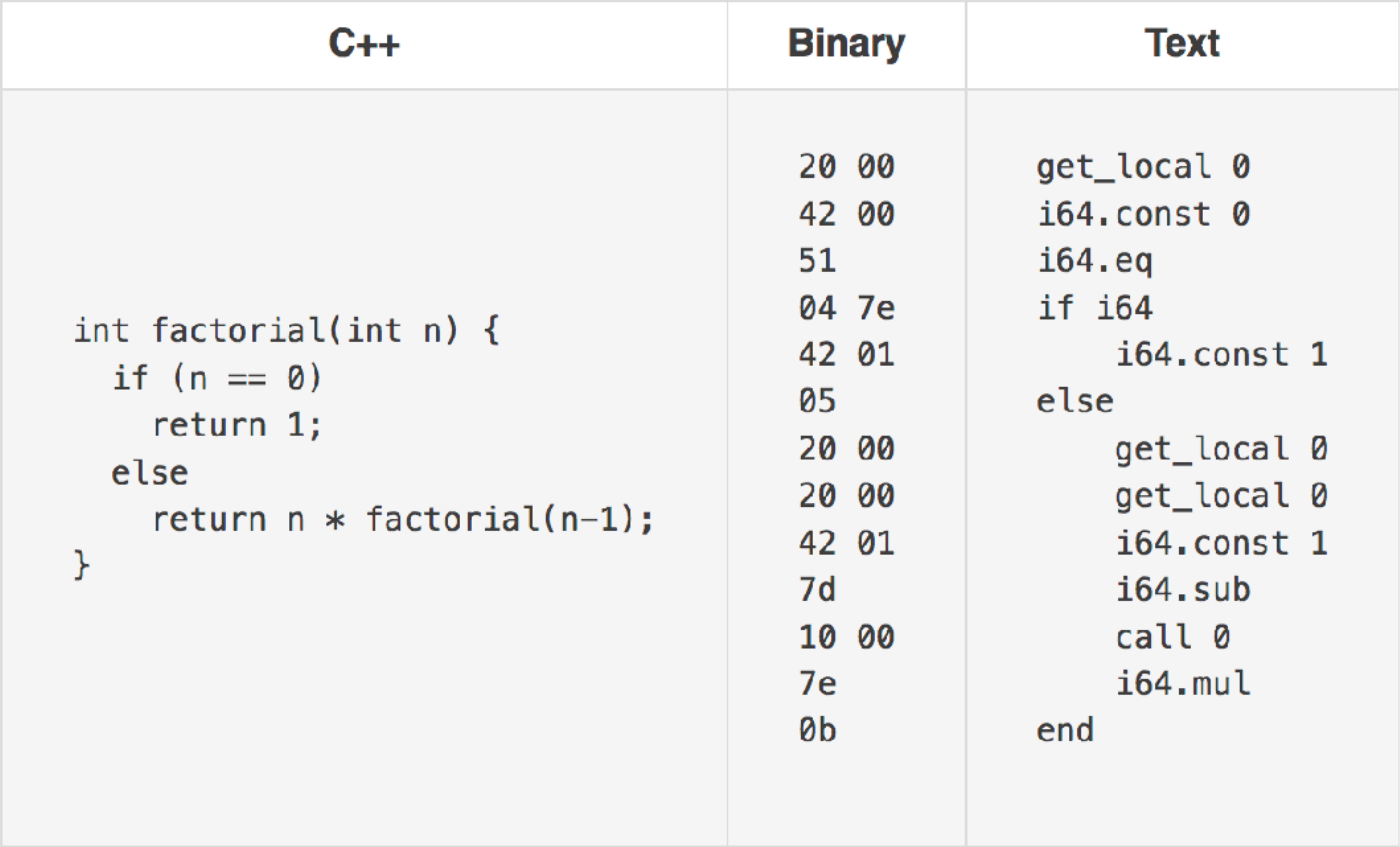
data section

0x5 0x0 0x0 0x0

...

https://rsms.me/wasm-intro

WAT:
text format



C++	Binary	Text

int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
	20 00
	42 00
	51
	04 7e
	42 01
	05
	20 00
	20 00
	42 01
	7d
	10 00
	7e
	0b
		get_local 0
		i64.const 0
		i64.eq
		if i64
		    i64.const 1
		else
		    get_local 0
		    get_local 0
		    i64.const 1
		    i64.sub
		    call 0
		    i64.mul
		end

http://webassembly.org/docs/text-format/#linear-instructions

JS API

```
fetch("add.wasm")
  .then(res => res.arrayBuffer())
  .then(buf =>
     WebAssembly.compile(buf))
  .then(bin =>
     WebAssembly.instantiate(bin, {})
  .then(mod => mod.add(1, 1));
```

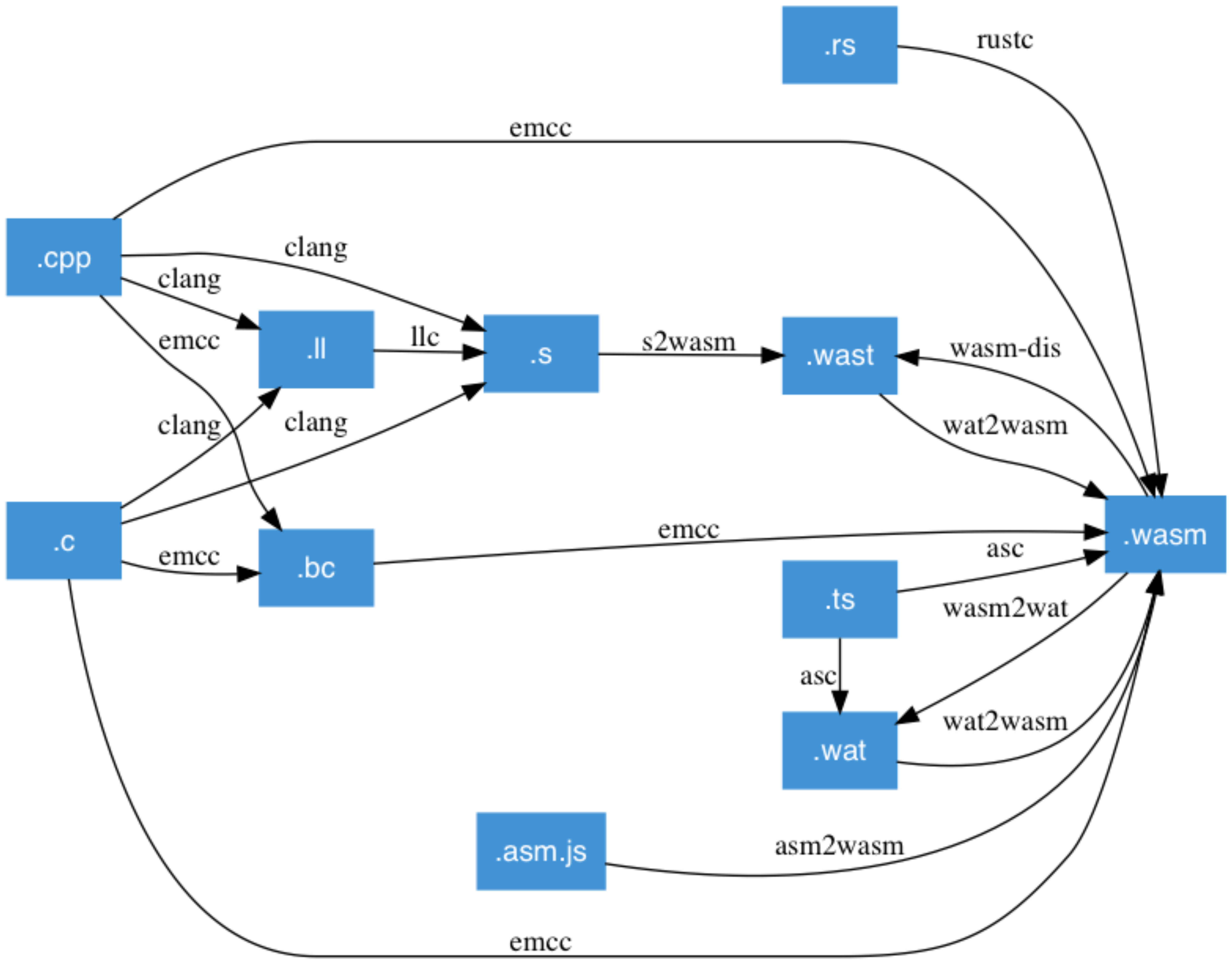
WebEmbedding API

```
WebAssembly
  .compileStreaming("add.wasm")
  .then(bin =>
    WebAssembly.instantiate(bin, {}))
  .then(mod => mod.add(1, 1));

WebAssembly
  .instantiateStreaming("add.wasm",
                        {})
  .then(mod => mod.add(1, 1);
```

Languages and Tools

Tool	Language	Compiler website
Emscripten	C / C++	http://kripken.github.io/emscripten-site/
Clang	C / C++	https://llvm.org/
Rustc	Rust	https://rust-lang.org/
AssemblyScript	TypeScript	https://github.com/AssemblyScript/assemblyscript
Binaryen	IR(LLVM / TS / Rust MIR)	https://github.com/WebAssembly/binaryen
Wabt	S-expression / text	https://github.com/WebAssembly/wabt
Unity	Unity(C#)	http://unity3d.com/
mono-wasm	C#	https://github.com/lrz/mono-wasm

WASM Tool chains



Emscripten
vs
Clang

- I **have not compared** in performance
- Emscripten has advantage in
  - Porting project
  - Library support
  - File system support
  - Easy installation
- Clang has advantage in
  - Creating relatively small modules

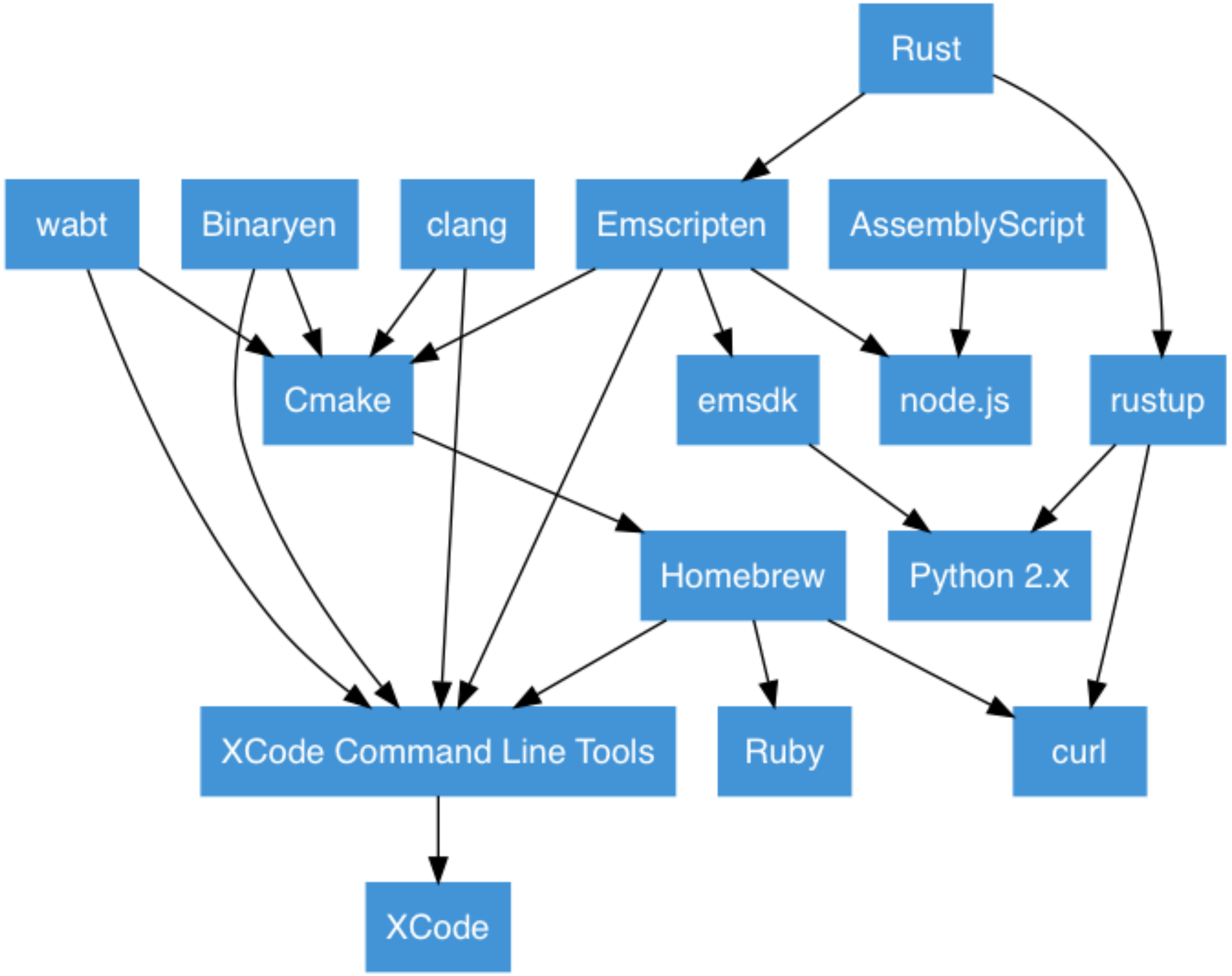
Tool installation: Emscripten

```
% URL="https://s3.amazonaws.com/
mozilla-games/emscripten/releases/
emsdk-portable.tar.gz"

% curl $URL | tar zx -
% cd emsdk-portable
% ./emsdk install latest
% ./emsdk activate latest
% source ./emsdk_env.sh
```

Tool installation: LLVM x Clang

```
% curl http://releases.llvm.org/5.0.0/llvm-5.0.0.src.tar.xz | tar Jxf -
% cd llvm-5.0.0.src/tools
% curl http://releases.llvm.org/5.0.0/cfe-5.0.0.src.tar.xz | tar Jxf -
% cd ../..
% mkdir build
% cd build
% cmake -DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=WebAssembly \
   ../llvm-5.0.0.src
% make
% make install
```

Tool dependencies (In macOS)



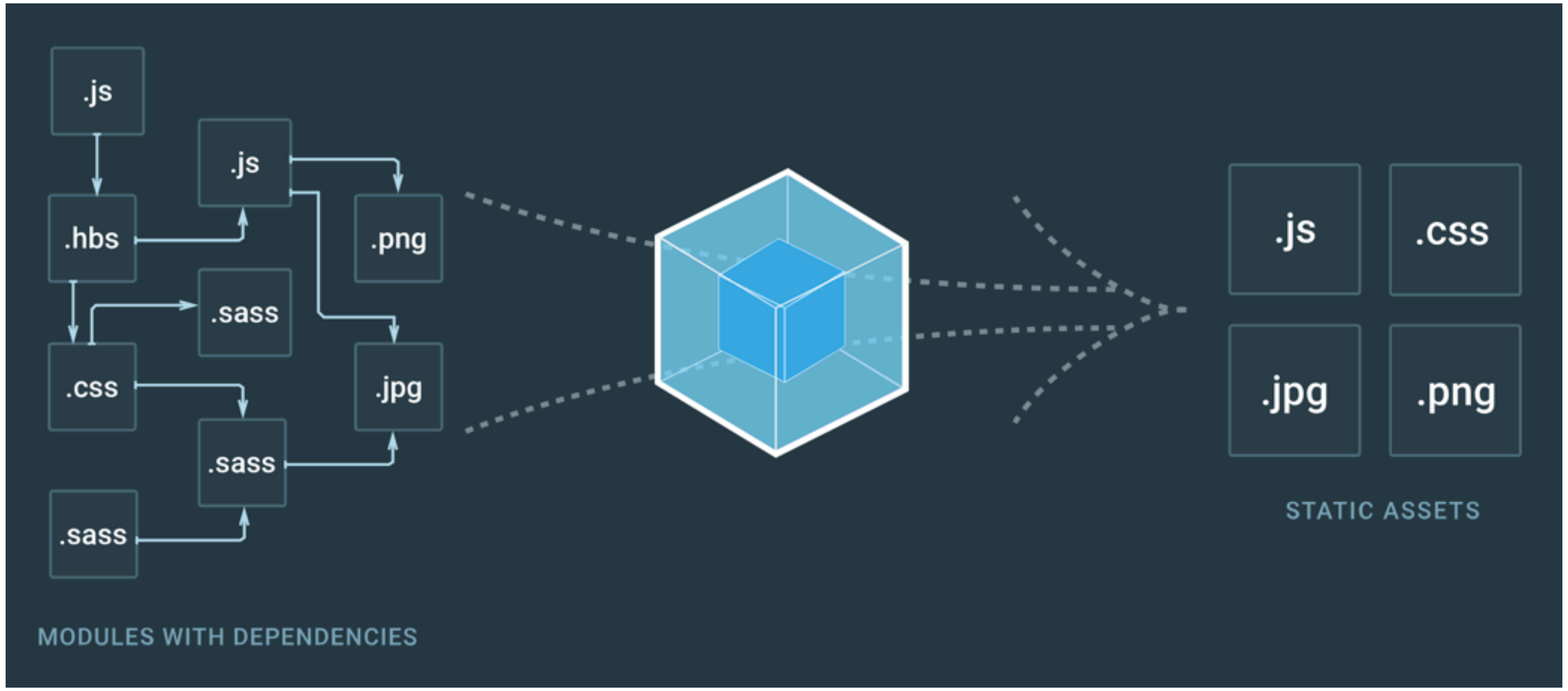
Env. setup to dev. with Rust

1. Install XCode
2. Install XCode Command Line Tools
3. Install Homebrew
4. Install cmake with Homebrew
5. Install emsdk
6. Install Emscripten
7. Install rustup
8. Add target architecture "wasm32-unkown-emscripten"

Tool installation: rustc & wasm32

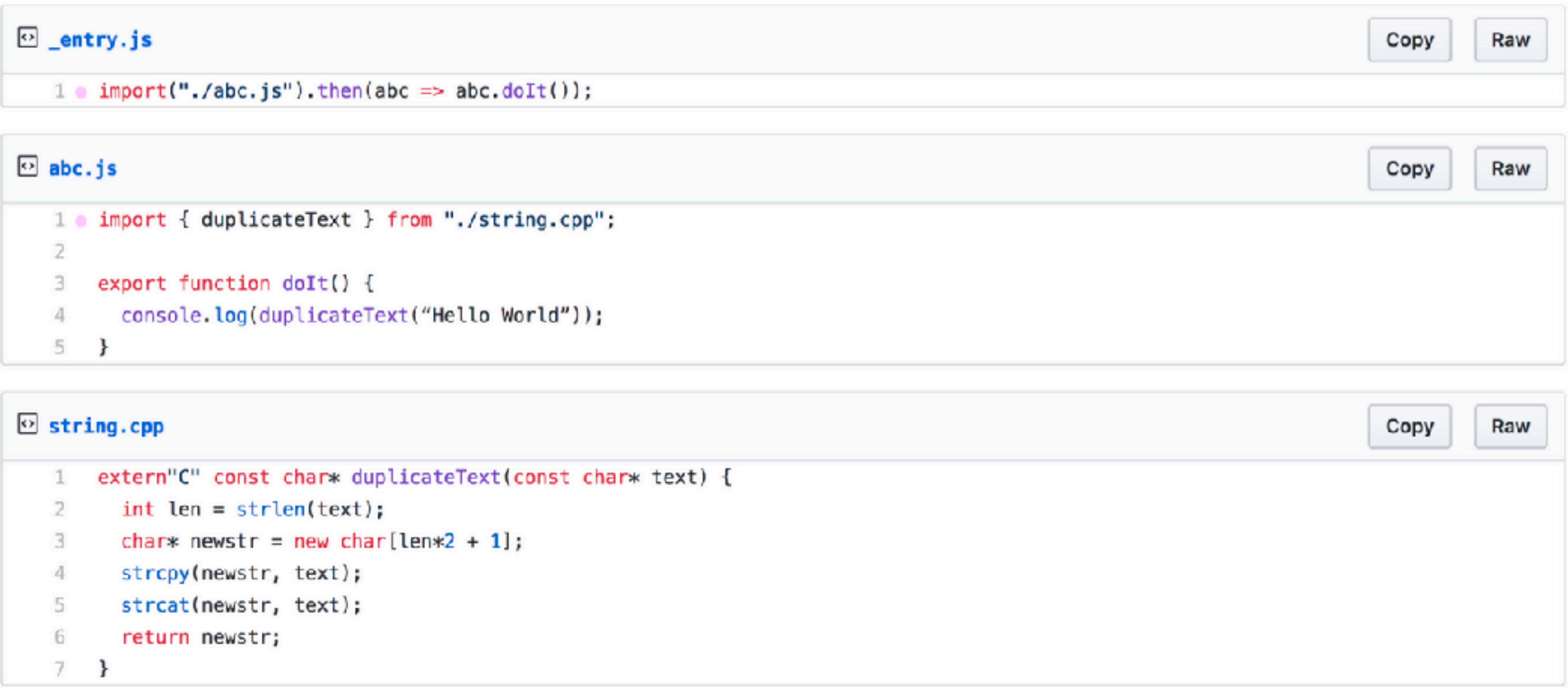
```
#install Emscripten prior to this

% curl https://sh.rustup.rs -sSf | sh
% rustup target \
  add wasm32-unknown-emscripten
% source ~/.cargo/env
```

Webpack



- http://webpack.js.org/
- A bundler:
  - Resolve module dependencies
  - Packs multiple files into 1 file

MOSS Program funded Webpack to implement first-class support for WASM



_entry.js

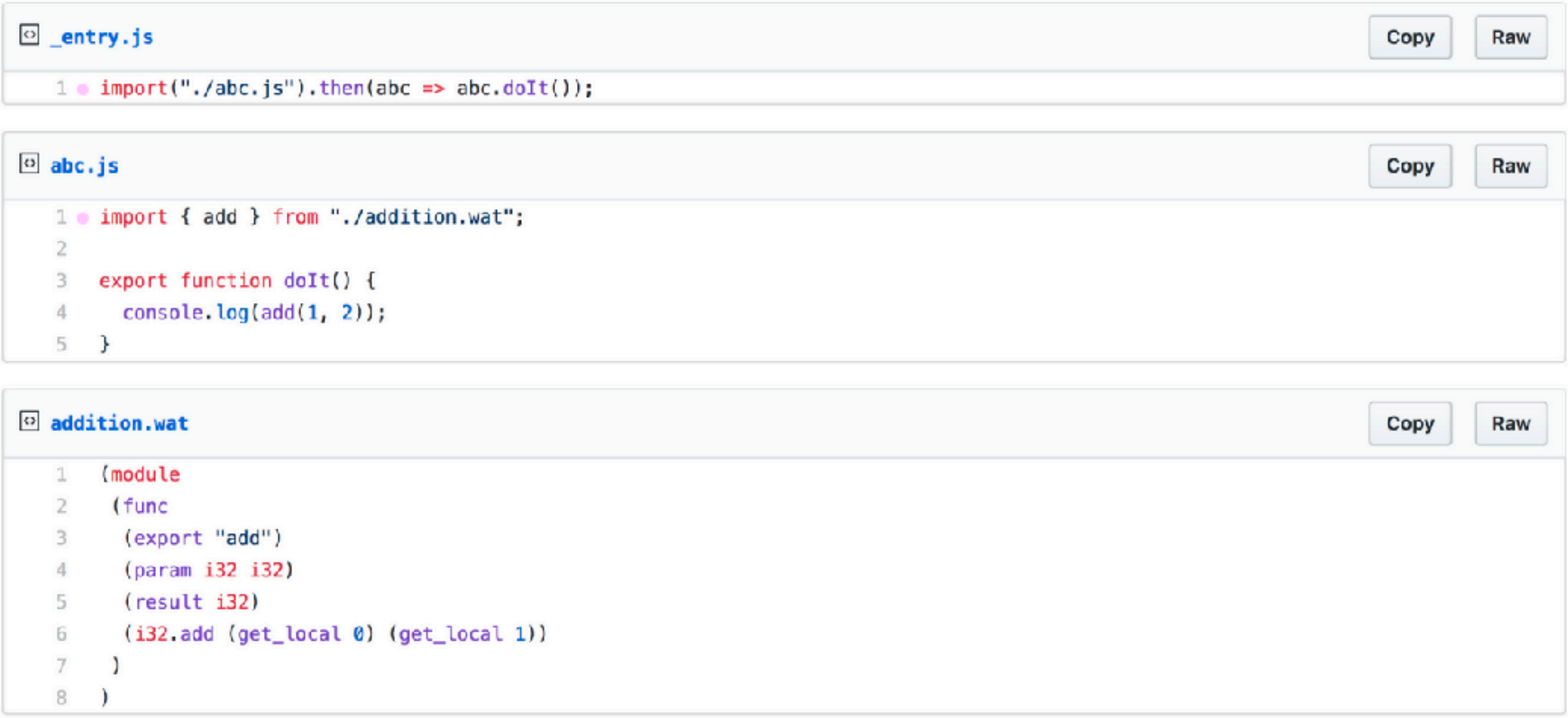
```js
import("./abc.js").then(abc => abc.doIt());
```

abc.js

```js
import { duplicateText } from "./string.cpp";

export function doIt() {
  console.log(duplicateText(“Hello World”));
}
```

string.cpp

```cpp
extern"C" const char* duplicateText(const char* text) {
  int len = strlen(text);
  char* newstr = new char[len*2 + 1];
  strcpy(newstr, text);
  strcat(newstr, text);
  return newstr;
}
```



_entry.js

```js
import("./abc.js").then(abc => abc.doIt());
```

abc.js

```js
import { add } from "./addition.wat";

export function doIt() {
  console.log(add(1, 2));
}
```

addition.wat

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

https://medium.com/webpack/webpack-awarded-125-000-from-moss-program-f63eeaaf4e15

Loader

- WebPack: bundler framework
- Loader
  - Defines a transformation applied on the source module
  - E.g. css-loader, ts-loader, babel-loader
- Loaders for development with WASM
  - wasm-loader
  - cpp-loader, rs-wasm-loader

wasm-loader

https://github.com/ballercat/wasm-loader

```javascript
import makeFactorial
  from 'wasm/factorial';

makeFactorial().then(instance => {
  const factorial =
    instance.exports.factorial;

  console.log(factorial(1)); // 1
  console.log(factorial(2)); // 2
  console.log(factorial(3)); // 6
});
```

```
// instantiate with feeding imports
makeFactorial({
  'global': {},
  'env': {
    'memory': new WebAssembly.Memory({initial: 100, limit: 1000}),
    'table': new WebAssembly.Table({initial: 0, element: 'anyfunc'})
  }
}).then(instance => { /* code here */ });

// default imports
{
  'global': {},
  'env': {
    'memory': new Memory({initial: 10, limit: 100}),
    'table': new Table({initial: 0, element: 'anyfunc'})
  }
}
```

cpp-loader

https://github.com/arcanis/cpp-loader

```
import { addition, range }
  from './lib.cc';

console.log(addition(2, 2));
console.log(range(10, 100));

/*
Note:
- This loader generates asm.js
- Available only on Windows
*/
```

rust-wasm-loader

https://github.com/ianjsikes/rust-wasm-loader

```
const wasm = require('./main.rs');
wasm.initialize().then(module => {
  const doub =
     module.cwrap('doub',
                  'number',
                  ['number']);
  console.log(doub(21));
});
```

AssemblyScript

TypeScript -> WASM

- TypeScript compiler API + Binaryen's WASM backend
- Compiler itself are written in TypeScript
- What to except
  - All types must be annotated
  - Optional function parameters require an initializer expression
  - Union types (except classType | null representing a nullable), any and undefined are not supported by design
  - The result of logical && / || expressions is always bool

Installation

```
$ npm install assemblyscript
```

Compiler usage

```
$ asc [options] entryFile

$ asc -o main.wasm main.ts
$ asc -o main.wasm --textFile main.wat main.ts
$ asc --noRuntime -o main.wasm --textFile main.wat main.ts
$ asc --noRuntime -O -o main.wasm --textFile main.wat main.ts
```

Sample code

```
export function add(a: i32, b: i32): i32 {
  return a + b;
}
```

Generated WASM file

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

Only "export function" is exported

```
export function add(a: i32, b: i32): i32 {
  return a + b;
}

function invertSign(a: i32): i32 {
  return -a;
}

export function sub(a: i32, b: i32): i32 {
  return add(a, invertSign(b));
}
```

Only "export function" is exported

```
(module
  (type (;0;) (func (param i32 i32) (result i32)))
  (type (;1;) (func (param i32) (result i32)))
  (func $add (type 0) (param i32 i32) (result i32)
// snip
  (func $invertSign (type 1) (param i32) (result i32)
// snip
  (func $sub (type 0) (param i32 i32) (result i32)
// snip
  (memory (;0;) 1)
  (export "add" (func $add))
  (export "sub" (func $sub))
  (export "memory" (memory 0)))
```

Constructor: memory initializer

```
export class Point {
  public x: i32
  public y: i32
  constructor(x: i32, y: i32) {
    this.x = x;
    this.y = y;
  }
}
```

```
(func $Point (type 2)
  (param i32 i32 i32) (result i32)
  block (result i32)  ;; label = @1
    get_local 0
    get_local 1
    i32.store
    get_local 0
    get_local 2
    i32.store offset=4
    get_local 0
  end)
```

Methods are translated into functions

```
export class Point {
  public x: i32
  public y: i32
  constructor(x: i32, y: i32) {
    this.x = x;
    this.y = y;
  }
  public norm(): i32 {
    return add(this.x, this.y);
  }
  public distance(point: Point): i32 {
    return this.norm() - point.norm();
  }
}
```

```
(func $Point#norm (type 1)
  param i32) (result i32)
  get_local 0
  i32.load
  get_local 0
  i32.load offset=4
  call $add)
(func $Point#distance (type 0)
  (param i32 i32) (result i32)
  get_local 0
  call $Point#norm
  get_local 1
  call $Point#norm
  i32.sub)
```

Summary

- All major modern browsers provide default support to WebAssembly MVP
- Web Embedding: http://webassembly.org/docs/web/
- WebPack will support WASM as its first class language
- AssemblyScript: a TypeScript subset compiler emits WASM