

Source-to-LLVM and running LLVM in the browser

Bobbie Soedirgo (T1)

Why LLVM?

- Industry grade compiler toolchain
- Modular, easier to build a frontend for relative to e.g. gcc
- Can target many architectures
- De facto choice if your language wants to target Wasm
 - Alternative: cranelift
 - WAVM: compile Wasm to LLVM?!



Architecture

- `js-slang`: `program.js` (Source) → `program.json`
- `sourcec`: `program.json` (ESTree) → `program.ll`
- `llvm-wasm`: `program.ll` (LLVM IR) → `program.wasm`

Source-to-LLVM (sourcec)

Source §1

```
program ::= import-directive ... statement ...
import-directive ::= import { import-names } from string ;
import-names ::= ε | name ( , name ) ...
statement ::= const name = expression ;
            | function name ( parameters ) block
            | return expression ;
            | if-statement
            | block
            | expression ;
parameters ::= ε | name ( , name ) ...
if-statement ::= if ( expression ) block
                | else ( block | if-statement )
block ::= { statement ... }
```

```
expression ::= number
            | true | false
            | string
            | name
            | expression binary-operator expression
            | unary-operator expression
            | expression ( expressions )
            | ( name | ( parameters ) ) => expression
            | ( name | ( parameters ) ) => block
            | expression ? expression : expression
            | ( expression )
binary-operator ::= + | - | * | / | % | === | !==
                  | > | < | >= | <= | && | ||
unary-operator ::= !
expressions ::= ε | expression ( , expression ) ...
```

What's missing?

```
program ::= import-directive ... statement ...
import-directive ::= import { import-names } from string;
import-names ::= ε | name( , name ) ...
statement ::= const name = expression ;
            | function name ( parameters ) block
            | return expression ;
            | if-statement
            | block
            | expression ;
parameters ::= ε | name( , name ) ...
if-statement ::= if ( expression ) block
               | else ( block | if-statement )
block ::= { statement ... }
```

```
expression ::= number
             | true | false
             | string
             | name
             | expression binary-operator expression
             | unary-operator expression
             | expression ( expressions )
             | ( name | ( parameters ) ) => expression
             | ( name | ( parameters ) ) => block
             | expression ? expression : expression
             | ( expression )
binary-operator ::= + | - | * | / | % | === | !==
                  | > | < | >= | <= | && | ||
unary-operator ::= !
expressions ::= ε | expression ( , expression ) ...
```

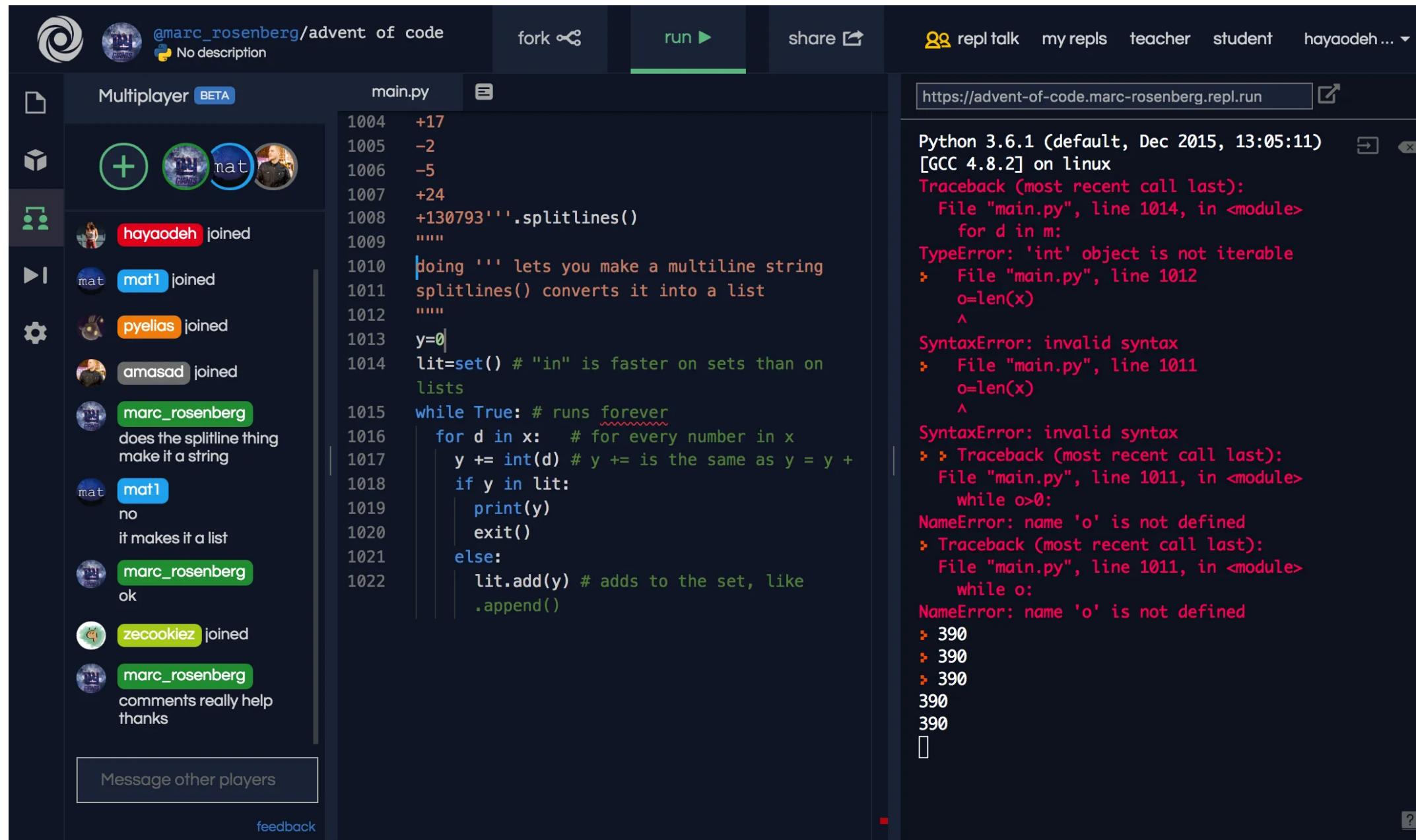
How it works

- Types:
 - undefined, boolean, number, function
 - Boxed values
- Environment:
 - Chain of frames, gets passed when entering a new scope
- Conditionals:
 - Phi values
- Functions:
 - Hoisted to top level (including closures)
 - Takes a function pointer and frame pointer as argument

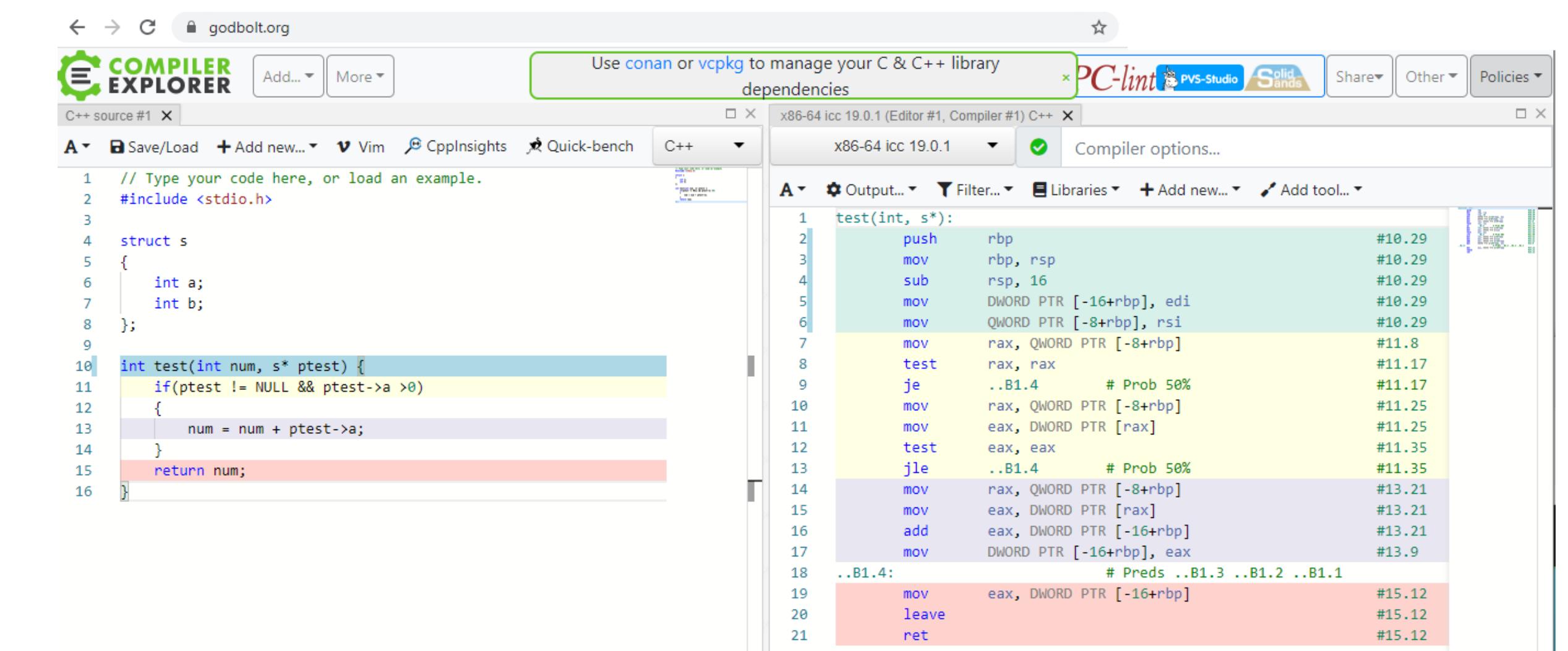
LLVM in the browser (llvm-wasm)

Why run in the browser?

Convenience & reproducibility



Replit



Godbolt

Why run in the browser?

Internet connectivity

- Computer labs, connection issues
- Subpar internet
- Can even run on you phone!

Prior arts

llvm.js

- Compiles LLVM IR to asm.js
- Uses an old (JS) version of Emscripten

Run LLVM Assembly In Your Browser

This is a demo of compiling and running [LLVM](#) assembly in JavaScript.

Press the button to see it compile and run a tiny "hello world" program. You can also try modifying the LLVM assembly and pressing the button again - note though that [LLVM assembly](#) is typed, so if you add characters to the string for example, you will need to adjust its length, both where it is defined and where it is used (otherwise you will get errors, which will show up in the output area).

```
target triple = "i386-pc-linux-gnu"

@.str = private constant [15 x i8] c"hello, world!\0A\00"

define i32 @main() {
entry:
%str = getelementptr inbounds [15 x i8]* @.str, i32 0, i32 0
%call = call i32 (i8*, ...)* @printf(i8* %str)
ret i32 1
}

declare i32 @printf(i8*, ...)
```

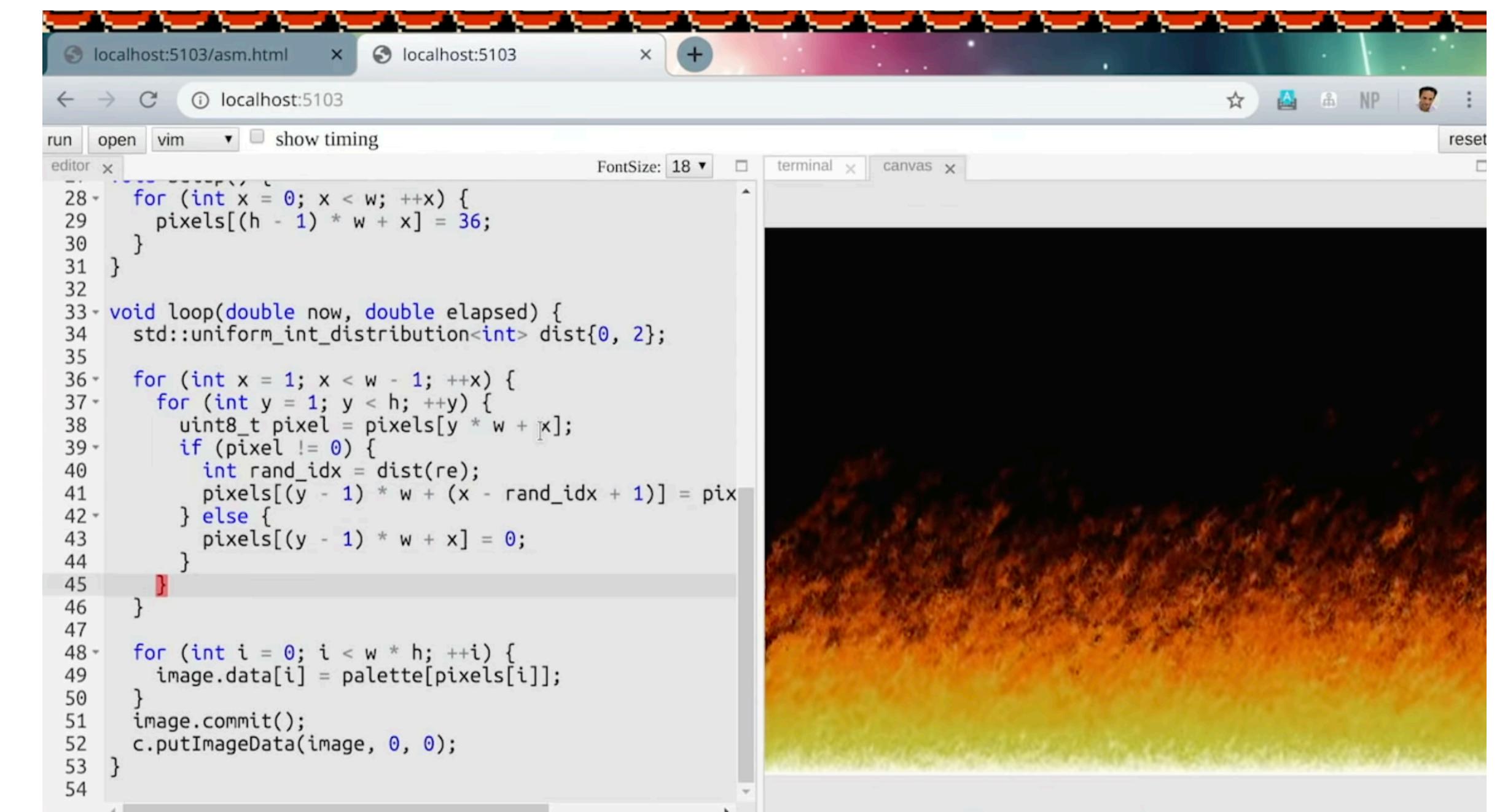
compile and run

Fork me on GitHub

Prior arts

wasm-clang

- Compiles clang to Wasm/WASI
- Using hacked LLVM source
- <https://youtu.be/5N4b-rU-OAA>



The screenshot shows a browser-based development environment with multiple tabs and panes. The main pane displays a Wasm assembly code editor with the following content:

```
28  for (int x = 0; x < w; ++x) {
29    pixels[(h - 1) * w + x] = 36;
30  }
31
32
33 void loop(double now, double elapsed) {
34   std::uniform_int_distribution<int> dist{0, 2};
35
36   for (int x = 1; x < w - 1; ++x) {
37     for (int y = 1; y < h; ++y) {
38       uint8_t pixel = pixels[y * w + x];
39       if (pixel != 0) {
40         int rand_idx = dist(re);
41         pixels[(y - 1) * w + (x - rand_idx + 1)] = pixel;
42       } else {
43         pixels[(y - 1) * w + x] = 0;
44       }
45     }
46
47
48   for (int i = 0; i < w * h; ++i) {
49     image.data[i] = palette[pixels[i]];
50   }
51   image.commit();
52   c.putImageData(image, 0, 0);
53 }
54 }
```

To the right of the editor is a canvas viewer displaying a colorful, abstract image with a gradient from dark blue to bright orange and yellow, resembling a flame or a sunset.

How it works

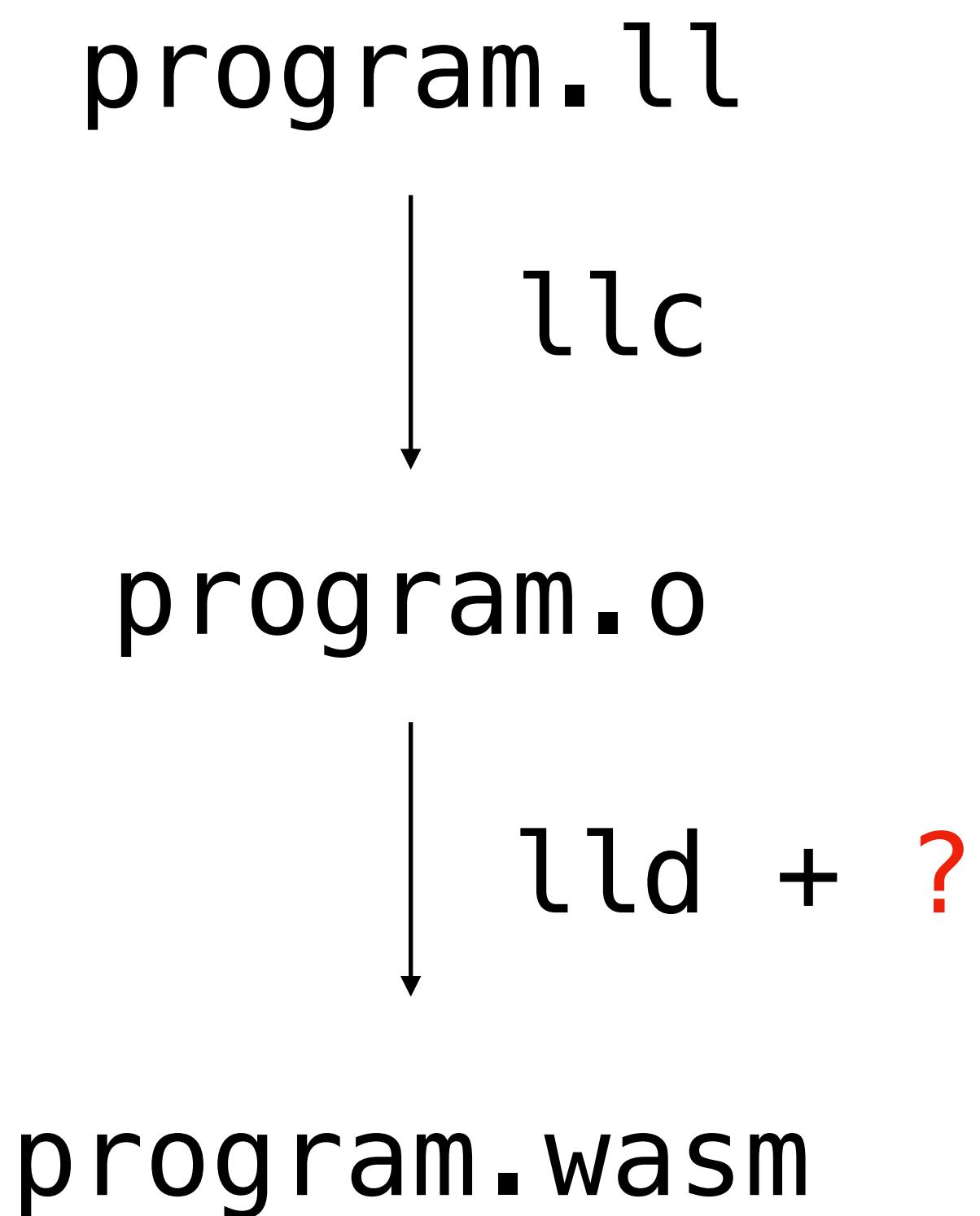
Which LLVM tools do we need?

- `llc`: LLVM static compiler
 - `program.ll` → `program.o`
- `lld`: LLVM linker
 - `program.o + ?` → `program.wasm`
- Compiled to Wasm + JS with Emscripten



How it works

Compilation



Running the Wasm module

WASI

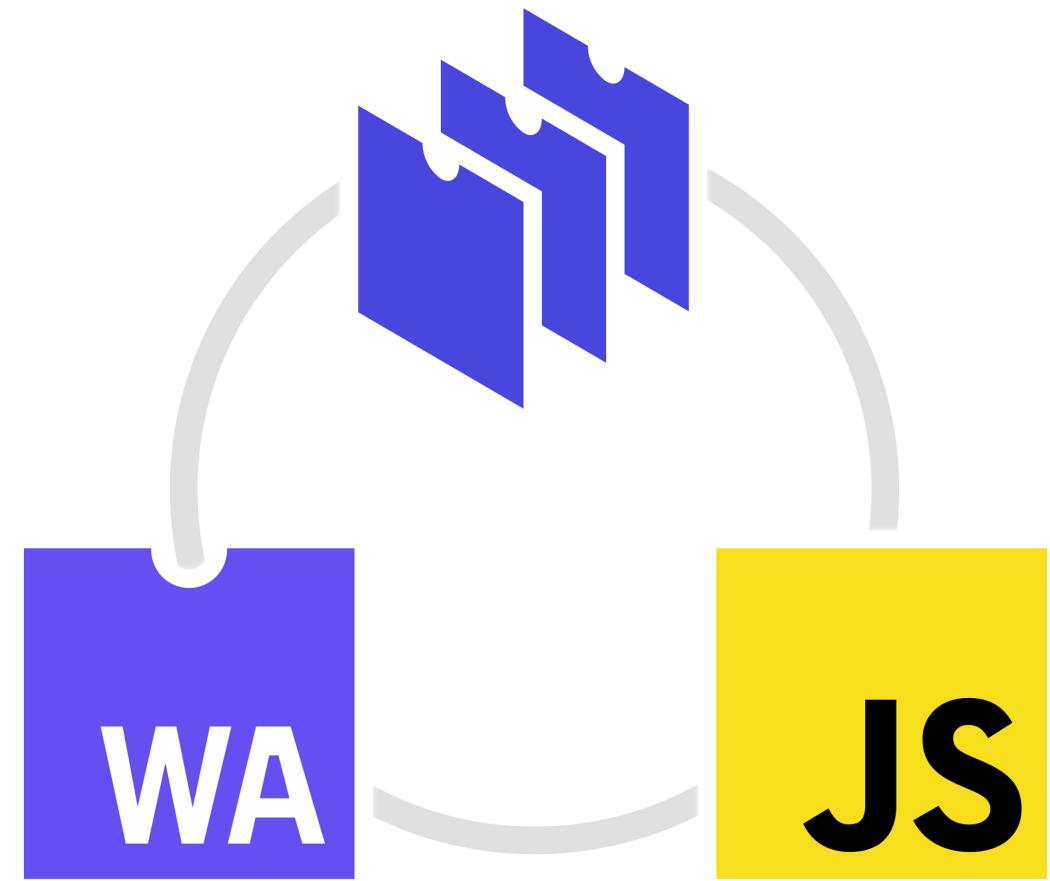
- WebAssembly System Interface
- Provides a syscall interface to common libc functions
 - printf
 - malloc
 - write
- Can run outside of browser (Wasmtime, Wasmer)



Running the Wasm module

`wasmer-js`

- WASI is not natively supported in browsers yet
- `wasmer-js`: WASI polyfill



Demo

Future extensions

Source-to-LLVM in the browser

- Emitting LLVM IR directly
 - Kinda painful
- Linking to LLVM support libraries
 - Need to know which libraries are needed
 - Need to integrate it with the LLVM C/C++ API/wrapper

Proper tail calls

- Hinges on the Wasm tail call proposal
- <https://github.com/WebAssembly/tail-call>

Garbage collection

- Hinges on the Wasm GC proposal
- <https://github.com/WebAssembly/gc>
- Alternative if not targeting Wasm:
 - Boehm-Demers-Weiser: <https://github.com/ivmai/bdwgc>