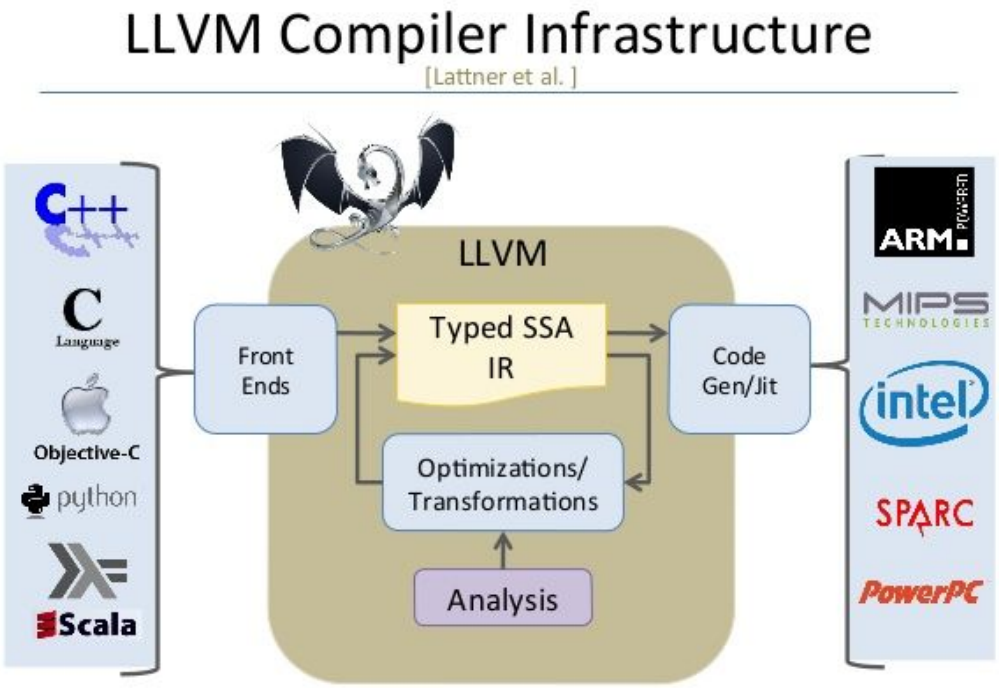
Webscripten

LLVM IR to WebAssembly compiler

T4
Donald Lee
Jin Jing

What is LLVM?


LLVM Compiler Infrastructure
[Lattner et al.]
C++
C
Language
Objective-C
python
Scala
Front Ends
LLVM
Typed SSA IR
Optimizations/ Transformations
Analysis
Code Gen/Jit
ARM
MIPS TECHNOLOGIES
intel
SPARC
PowerPC

What is LLVM?

- Set of production quality, reusable libraries
- Implement modern techniques
- Focus on compile time, performance of generated code
- Supports many different targets

What is LLVM IR

- Core of LLVM
- Intermediate representation - somewhere between source code and machine code, for a specific architecture
- Biggest advantage: acts as intermediary, independence from source/target
- Other IRs: Java bytecode, Microsoft CIL
- Official spec: https://llvm.org/docs/LangRef.html

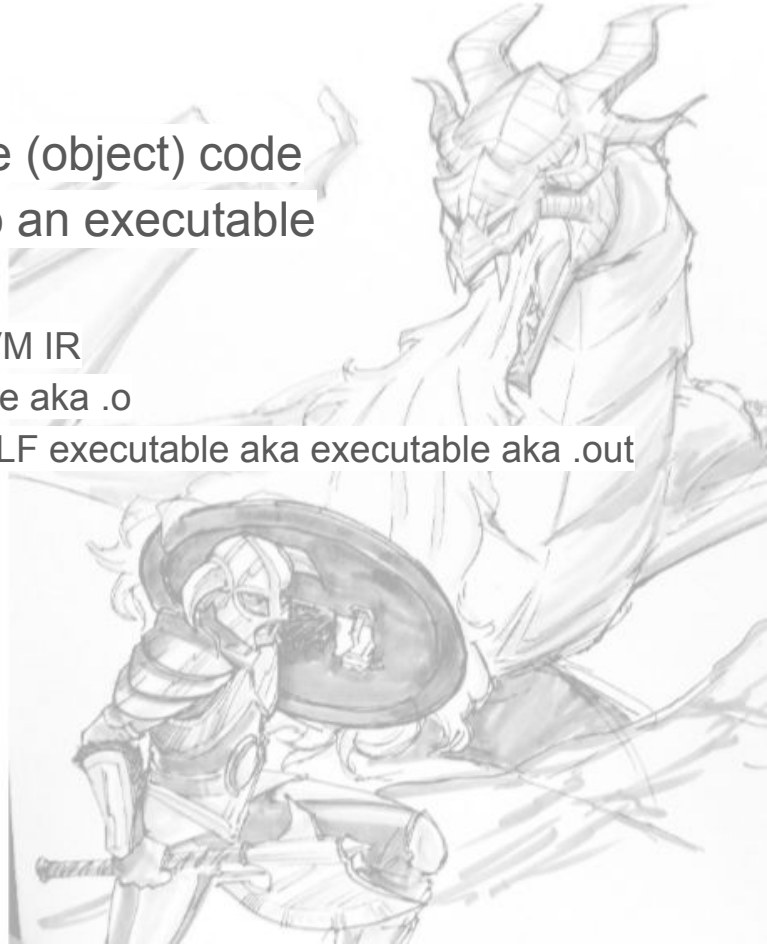
```
; ModuleID = 'mul.c'
source_filename = "mul.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [4 x i8] c

; Function Attrs: noinline nounwind optnone ssps
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  store i32 0, i32* %3, align 4
  br label %4
```

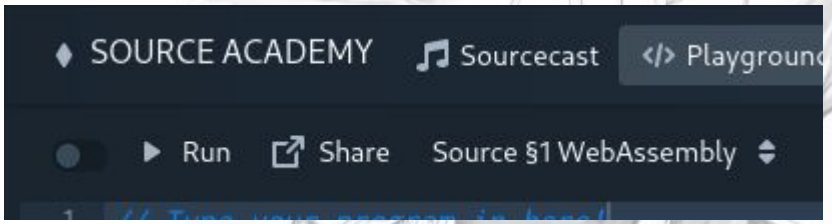
Tools of LLVM

- llc: LLVM static Compiler: LLVM IR -> machine (object) code
- lld: LLVM Linker: Links several object files into an executable
- Example for Clang on x86:
  - Clang frontend compiles C/CPP (source code) to LLVM IR
  - llc compiles LLVM IR to ELF relocatable aka object file aka .o
  - lld links object files (with libc, and other libraries) to ELF executable aka executable aka .out
- But these tools do not work in the browser...



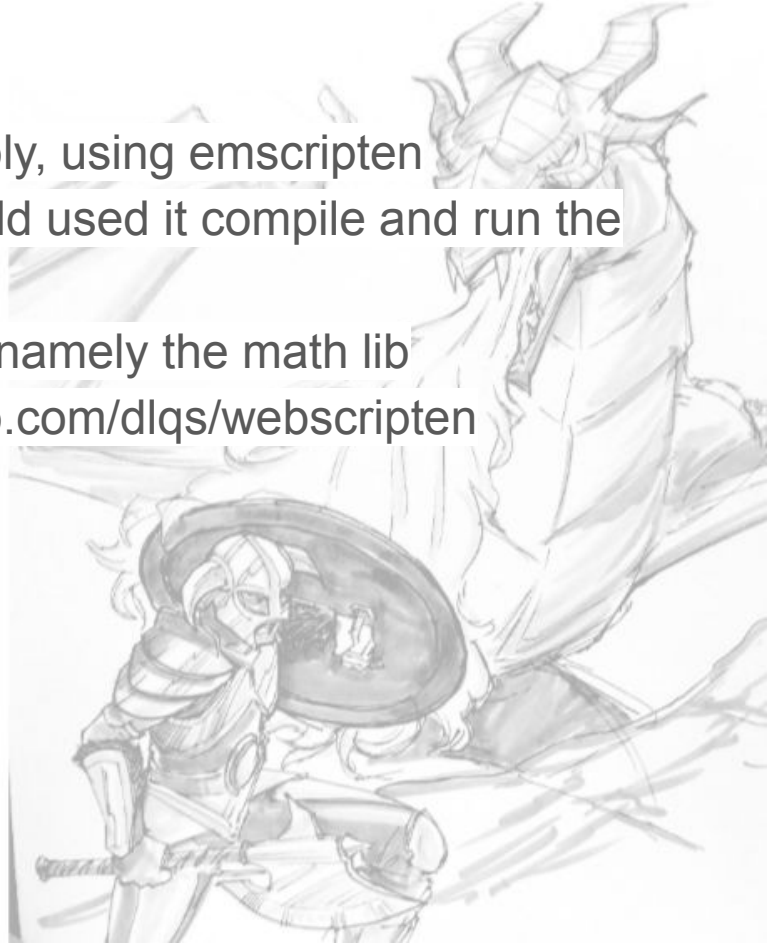
What is webscripten really?

- We want to compile Source -> WebAssembly in the browser (source-academy)
- llvm-sauce (another group) compiles Source -> LLVM IR
- So we’re compiling LLVM IR -> WebAssembly, in WebAssembly
- Similar idea: Sourceror (another group) compiles directly to WebAssembly

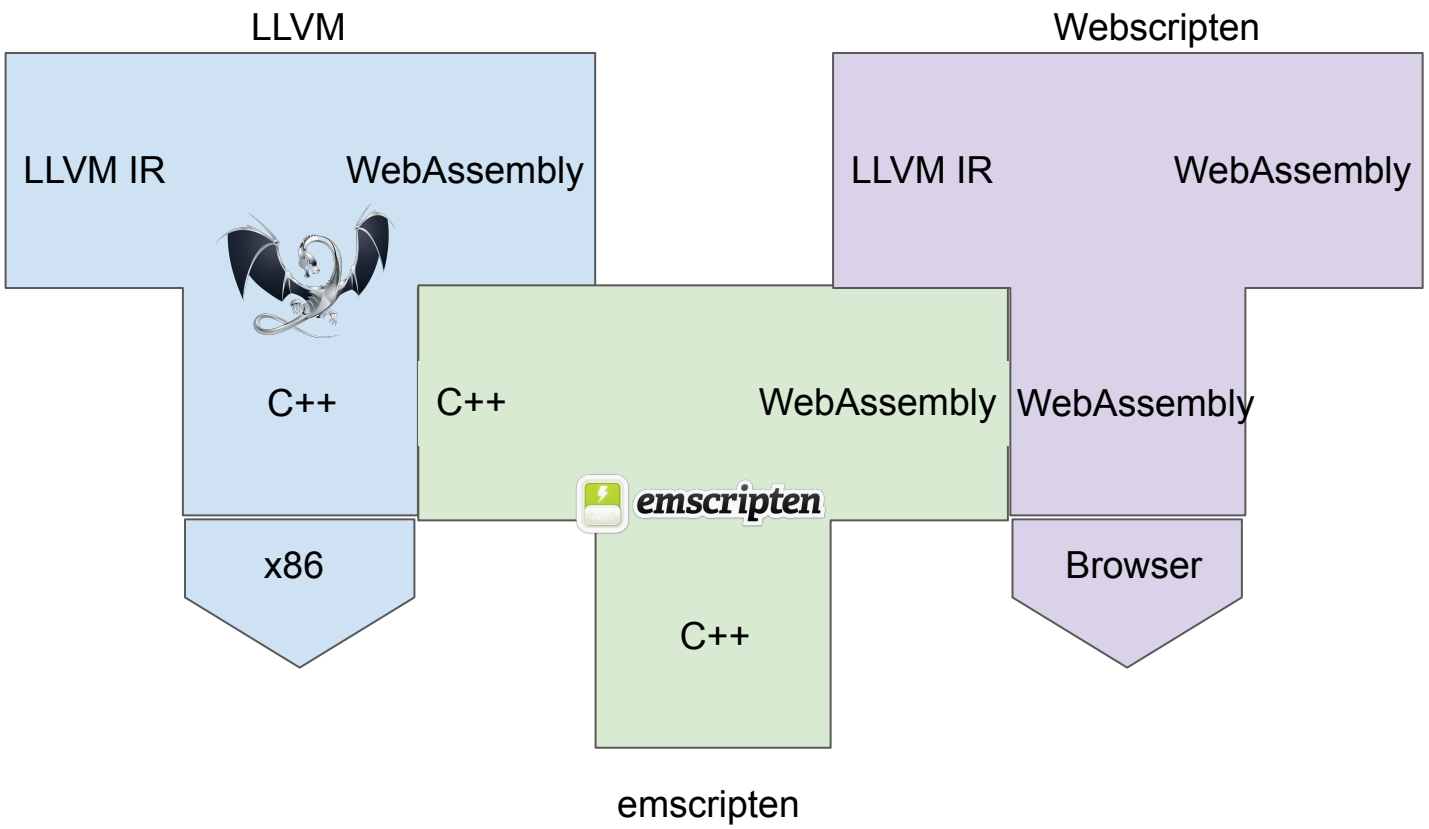


What we did

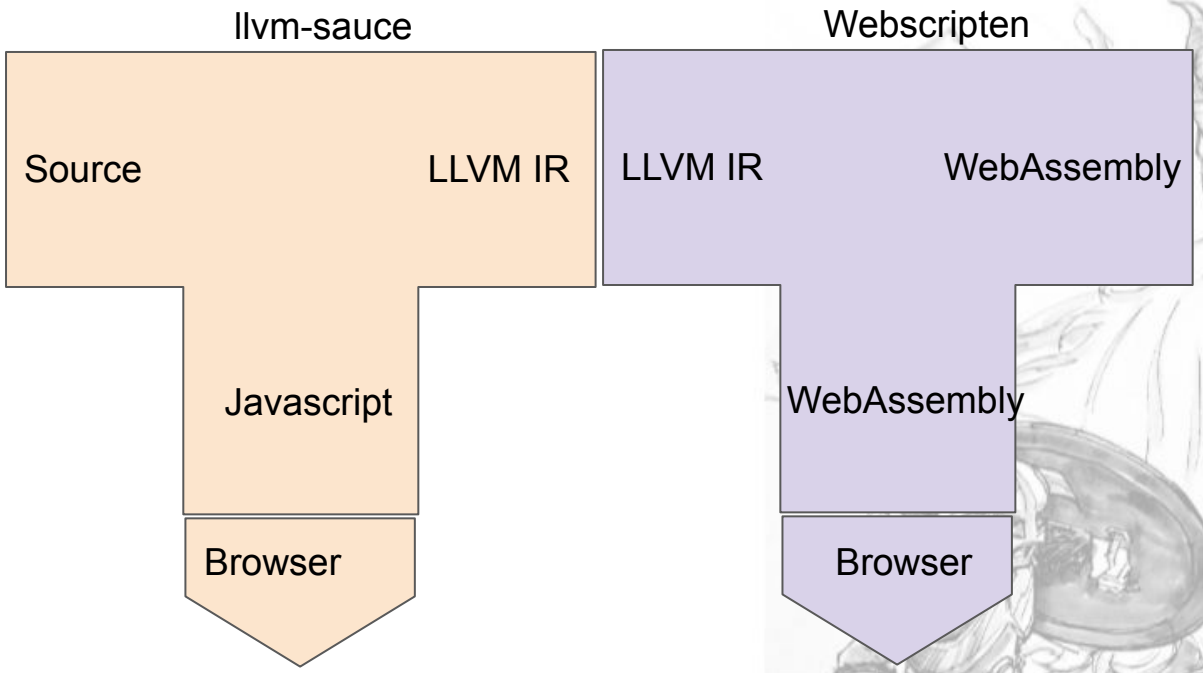
- Compile LLVM tools from C++ to WebAssembly, using emscripten
- Build a toolchain from wasm versions of llc + lld used it compile and run the final wasm
- Build a way to link external libraries to wasm, namely the math lib
- Runs standalone in the browser: https://github.com/dlqs/webscripten



What is webscripten really?

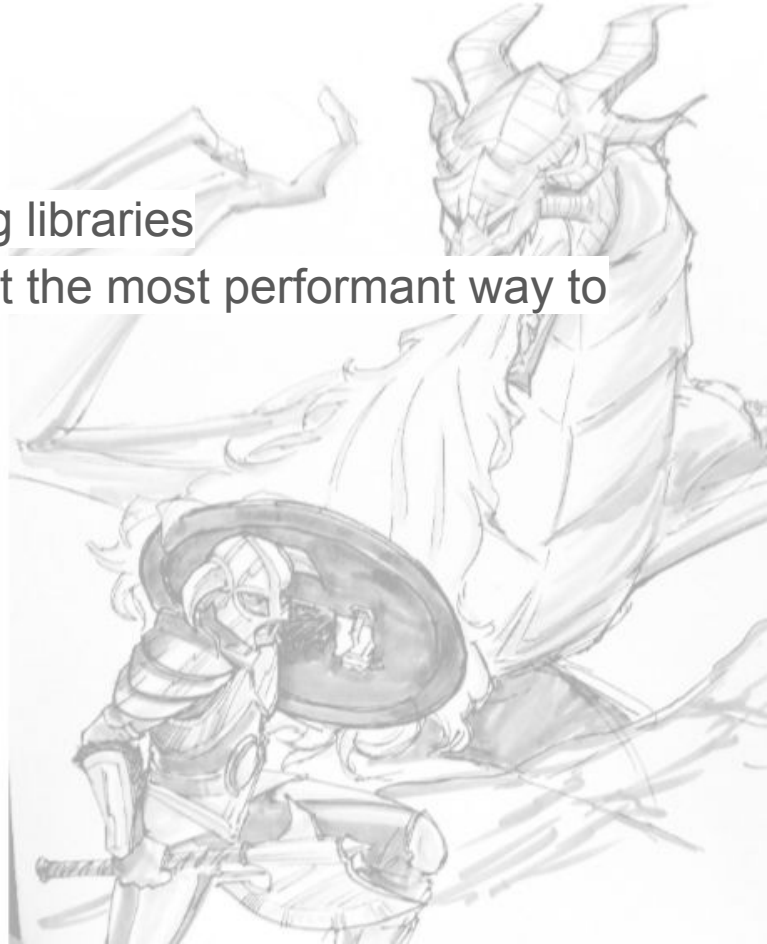


What is webscripten really?



Difficulties

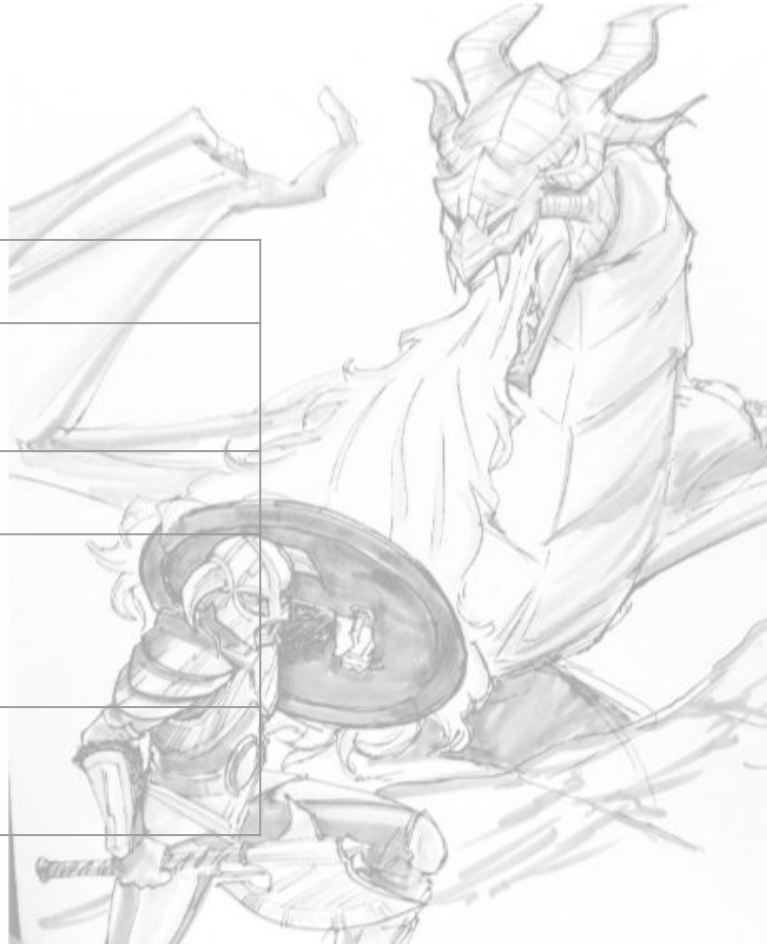
- Cross compiling the LLVM project
- Large binary sizes of llc and lld and supporting libraries
- Running 3 .wasm modules in your browser not the most performant way to compile code



Benchmarks

- Excluding compile time

milliseconds	webscripten	js-slang
direct recursive fibonacci (40)	672	28510
ackerman(3, 9)	345	4353
10,000,000 multiplication operations	111	4474
10,000,000 log operations	336	6375



How

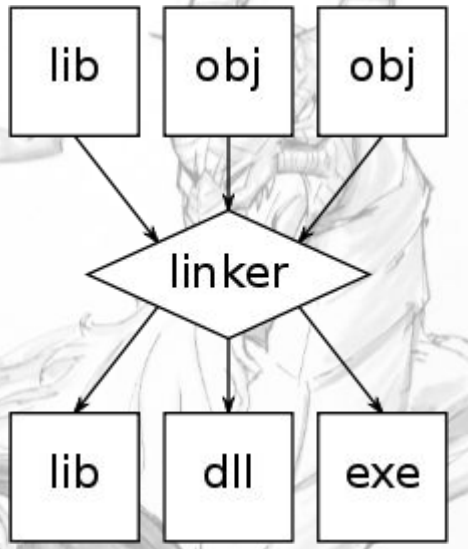
- First compiled the LLVM project
- emscripten did most of the heavy lifting by compiling to WebAssembly
- Metacircular (compiler compiling itself?)
- How to run a “process” in the browser?
- fs api provides a fake filesystem for the process to interact with
- Does input/output as well

```
function runLLC(code, staticPath) {
  return new Promise((resolve, reject) => {
    function preRun() {
      this.FS.writeFile('./a.ll', code)
    }
    function postRun() {
      const exists = this.FS.analyzePath('./a.o').exists
      if (exists) {
        const uint8 = this.FS.readFile('./a.o', { encoding: 'binary' })
        const hex = util.Uint8ArrayToHex(uint8)
        resolve(hex)
```

Linker Overview

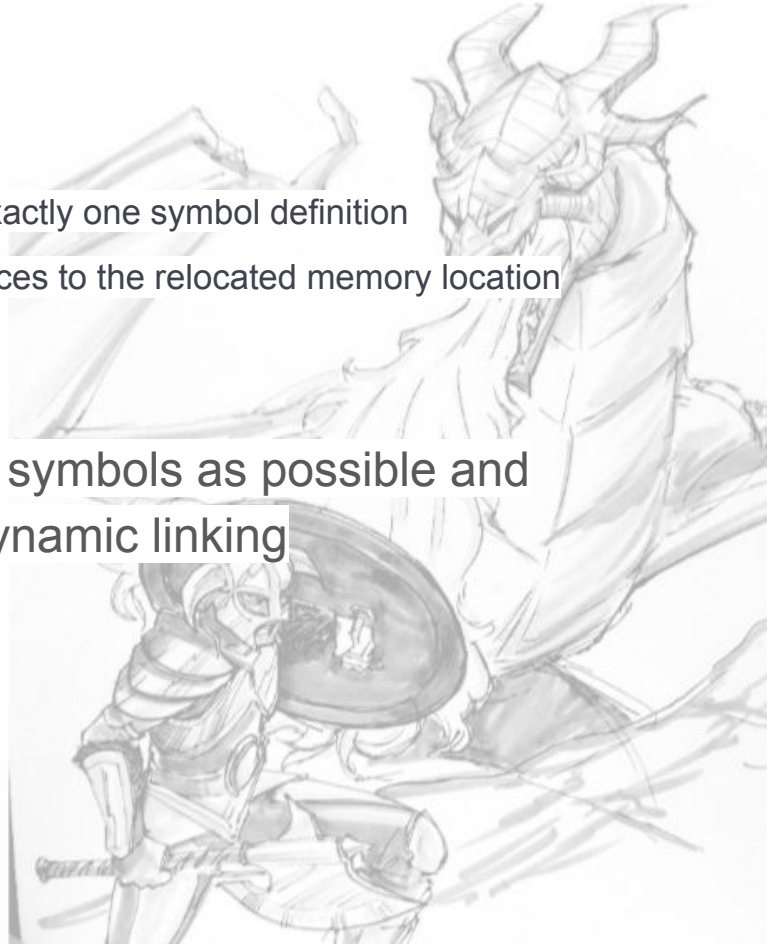
- Linkers take multiple object files/ libraries and combines them into a single executable/ library/ dynamic linking library(dll)

- Two kinds of linking:
  - Static Linking, happens during compile time (in our case, “link time”)
  - Dynamic Linking, happens during run time



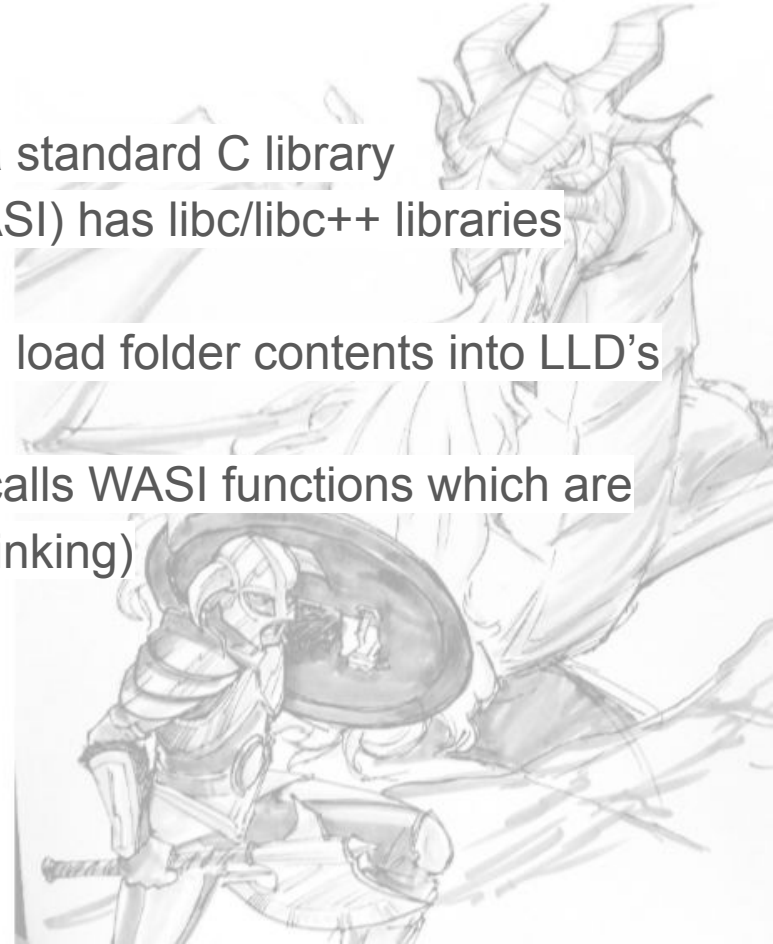
More on Linking

- Static Linking
  - **Symbol resolution –** Tries to match each symbol with exactly one symbol definition
  - **Relocation –** Relocates code and modify symbol references to the relocated memory location
- Dynamic Linking
  - Add the required libraries to runtime environment
- In our case, we let LLD statically link as many symbols as possible and ignore the remaining undefined symbols for dynamic linking



Static Library linking

- Problem: LLVM itself doesn't provide a libc - a standard C library
- Solution: WebAssembly System Interface(WASI) has libc/libc++ libraries readily compiled in a sysroot folder
- Before linking, we fetch the sysroot folder and load folder contents into LLD’s virtual file system
- However, the the definitions of libc functions calls WASI functions which are not yet defined (hence the need for dynamic linking)



WebAssembly Dynamic Linking

Linking javascript functions to functions defined in IR

In C

```
extern double math_sin(double);
```

```
declare double @math_sin(double)
```

In LLVM IR

WebAssembly Dynamic Linking

Linking javascript functions to functions defined in IR eg. math_sin

```
exports.math_abs = Math.abs
exports.math_acos = Math.acos
exports.math_acosh = Math.acosh
exports.math_asin = Math.asin
exports.math_asinh = Math.asinh
exports.math_atan = Math.atan
exports.math_atan2 = Math.atan2
exports.math_atanh = Math.atanh
exports.math_cbrt = Math.cbrt
```

```
const importObject = {
  ...wasi.getImports(module),
  env: {
    ...math,
  },
}

let instance = await WebAssembly.instantiate(module, importObject)
```

Demo

https://dlqs.github.io/webscripten/demo/dist/index.html

Future work: integration with llvm-sauce

- This completes the toolchain
- Source -> WebAssembly compiler, which is able to harness LLVM tools
  - Currently llvm-sauce cannot run standalone on the browser due to its dependency of LLVM itself, requires cmake

Future work: llvm tools

- llvm-opt: LLVM IR optimizer
  - Optimizations: dead code elimination, constant propagation
  - Challenges: performance

Future work: dynamic linking

- Other source libraries
  - RUNES
  - CURVES
- Libraries that pass higher order functions
  - Possible way to implement
    - Motivation: Emscripten has API to add javascript functions into WebAssembly function table as WebAssembly functions
      - Indirect function calls in WebAssembly by accessing index of function table
      - Javascript can read/modify the WebAssembly function table but the functions must be exported WebAssembly functions
      - Emscripten creates a new WebAssembly instance with function as an import and extracts the WebAssembly function using javascript

Links

- Repo: https://github.com/dlqs/webscripten
- Demo: https://dlqs.github.io/webscripten/demo/dist/index.html
- llvm-sauce: https://github.com/jiachen247/llvm-sauce
- LLVM: https://llvm.org/
- Sick art: https://deviantart.com/kieronogorman/art/Skyrim-Dragon-Fighting-396244272