Build your own WebAssembly Compiler

Colin Eberhardt, Scott Logic
Why do we need WebAssembly?
JavaScript is a compilation target
> WebAssembly or wasm is a new portable, size- and load-time-efficient format suitable for compilation to the web.
JavaScript code is much more expensive, byte for byte, than an image, because of the time spent parsing and compiling it.

It's possible to parse and compile wasm as fast as it comes over the network, which makes it much more like an image than JavaScript code.

Game changer!
Why create a WebAssembly compiler?
Most Loved, Dreaded, and Wanted Languages

<table>
<thead>
<tr>
<th>Loved</th>
<th>Dreaded</th>
<th>Wanted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rust</td>
<td>83.5%</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>73.1%</td>
<td></td>
</tr>
<tr>
<td>TypeScript</td>
<td>73.1%</td>
<td></td>
</tr>
<tr>
<td>Kotlin</td>
<td>72.6%</td>
<td></td>
</tr>
<tr>
<td>WebAssembly</td>
<td>69.5%</td>
<td></td>
</tr>
<tr>
<td>Swift</td>
<td>69.2%</td>
<td></td>
</tr>
<tr>
<td>Clojure</td>
<td>68.3%</td>
<td></td>
</tr>
<tr>
<td>Elixir</td>
<td>68.2%</td>
<td></td>
</tr>
</tbody>
</table>

https://insights.stackoverflow.com/survey/2019
Bucket List

☑ Create an open source project
☑ Meet Brendan Eich
☑ Write an emulator
☐ Create my own language and a compiler
var y = 0
while (y < 100)
  y = (y + 1)
var x = 0
while (x < 100)
  x = (x + 1)

var e = ((y / 50) - 1.5)
var f = ((x / 50) - 1)

var a = 0
var b = 0
var i = 0
var j = 0
var c = 0

while (((i * i) + (j * j)) < 4) && (c < 255))
  i = (((a * a) - (b * b)) + e)
  j = (((2 * a) * b) + f)
  a = i
  b = j
  c = (c + 1)
endwhile
A simple wasm module
const magicModuleHeader = [0x00, 0x61, 0x73, 0x6d];
const moduleVersion = [0x01, 0x00, 0x00, 0x00];

export const emitter: Emitter = () => Uint8Array.from([...
magicModuleHeader,
...moduleVersion])

- wasm modules are binary
- Typically delivered to the browser as a .wasm file
const wasm = emitter();
const instance = await WebAssembly.instantiate(wasm);

- Instantiated asynchronously via the JS API
- Runs alongside the JavaScript virtual machine
- This compiles the wasm module, returning the executable
  - ... which currently does nothing!
An ‘add’ function
wasm has a relatively simple instruction set

- Four numeric types
  - More complex types can be constructed in memory (more on this later ...)
- Stack machine
- WebAssembly has no built in I/O
header: 0x00 0x61 0x73 0x6d  version: 0x01 0x00 0x00 0x00

type (0x01): (i32, i32) => (i32), (i64, i64) => ()

import (0x02): “print”, “sin”

function (0x03): type 0, type 2, type 1

etc ...

code (0x0a): code for fn 1, code for fn 2, code for fn 3

etc ...
const code = [
    Opcodes.get_local /** 0x20 */,
    ...unsignedLEB128(0),
    Opcodes.get_local /** 0x20 */,
    ...unsignedLEB128(1),
    Opcodes.f32_add /** 0x92 */
];

const functionBody = encodeVector([...
    ...encodeVector([]) /** locals */,
    ...code,
    Opcodes.end /** 0x0b */
]);

const codeSection = createSection(Section.code, encodeVector([[functionBody]]));
const { instance } = await WebAssembly.instantiate(wasm);

console.log(instance.exports.add(5, 6));

// 11
Building a compiler
```javascript
var a = 0
var b = 0
var i = 0

var e = ((y / 50) - 1.5)
var f = ((x / 50) - 1)

while (((i * i) + (j * j)) < 4) && (c < 255))
{
i = (((a * a) - (b * b)) + e)
j = (((2 * a) * b) + f)
a = i
b = j
}
c = (c + 1)
endwhile

setpixel x y c
```
code → Tokeniser → tokens → Parser → AST → Emitter → wasm
chasm v0.1

print 12

print 46.1
Tokenizer
patterns

"^[0-9]+$"

"^(print|var)$"

"^\s+$"

output

[]

input

" print 23.1"
<table>
<thead>
<tr>
<th>patterns</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>^[.0-9]+</code></td>
<td><code>[]</code></td>
</tr>
<tr>
<td>`^(print</td>
<td>var)`</td>
</tr>
<tr>
<td><code>^\s+</code></td>
<td></td>
</tr>
</tbody>
</table>

**input**

```
print 23.1
```
patterns

"^[0-9]+"

"^(print|var)"

"\s+"

output

[
{
"type": "keyword",
"value": "print",
"index": 1
}
]

input

" print 23.1"
patterns

"^[0-9]+$"

"^(print|var)"

"\s+"

input

" print 23.1"

output

[
  {
    "type": "keyword",
    "value": "print",
    "index": 1
  }
]
patterns

"^[0-9]+"

"^(print|var)"

"\s+"

input

" print 23.1"

output

[  
  
  
  ]

  
  
  {  
    "type": "keyword",  
    "value": "print",  
    "index": 1  
  },  

  
  
  {  
    "type": "number",  
    "value": "23.1",  
    "index": 7  
  }

]
patterns

"^[.0-9]+$"

"^(print|var)"

"\\s+"

input

" print 23.1"

output

[

{
  "type": "keyword",
  "value": "print",
  "index": 1
},
{
  "type": "number",
  "value": "23.1",
  "index": 7
}
]
- Removes whitespace
- Basic validation of syntax
export const parse: Parser = tokens => {
    const iterator = tokens[Symbol.iterator]();
    let currentToken = iterator.next().value;

    const eatToken = () =>
        (currentToken = iterator.next().value);

    [...]

    const nodes: StatementNode[] = [];
    while (index < tokens.length) {
        nodes.push(parseStatement());
    }

    return nodes;
};

tokens

[  
  {  
"type": "keyword",
"value": "print",
"index": 1
  },
  {  
"type": "number",
"value": "23.1",
"index": 7
  }
]
export const parse: Parser = tokens => {
  const iterator = tokens[Symbol.iterator]();
  let currentToken = iterator.next().value;

  const eatToken = () =>
    (currentToken = iterator.next().value);

  [...]

  const nodes: StatementNode[] = [];
  while (currentToken) {
    nodes.push(parseStatement());
  }

  return nodes;
};
const parseStatement = () => {
  if (currentToken.type === "keyword") {
    switch (currentToken.value) {
      case "print":
        eatToken();
        return {
          type: "printStatement",
          expression: parseExpression()
        };
    }
  }
};

tokens
[
{
  "type": "keyword",
  "value": "print",
  "index": 1
},
{
  "type": "number",
  "value": "23.1",
  "index": 7
}
]
```javascript
const parseExpression = () => {
    let node: ExpressionNode;
    switch (currentToken.type) {
        case "number":
            node = {
                type: "numberLiteral",
                value: Number(currentToken.value)
            };
            eatToken();
            return node;
    }
    return null;
}
```

```json
(tokens
[
  {
    "type": "keyword",
    "value": "print",
    "index": 1
  },
  {
    "type": "number",
    "value": "23.1",
    "index": 7
  }
]
)

(ANTLR 4
[AST
  {
    "type": "printStatement",
    "expression": {
      "type": "numberLiteral",
      "value": "23.1"
    }
  }
]
)
Emitter
const codeFromAst = ast => {
    const code = [];

    const emitExpression = node => {
        switch (node.type) {
            case "numberLiteral":
                code.push(OpCodes.f32_const);
                code.push(...ieee754(node.value));
                break;
        }
    }

    ast.forEach(statement => {
        switch (statement.type) {
            case "printStatement":
                emitExpression(statement.expression);
                code.push(OpCodes.call);
                code.push(...unsignedLEB128(0));
                break;
        }
    });

    return code;
}
Demo Time!
" print 42"

tokens

```json
[
  {
    "type": "keyword",
    "value": "print",
    "index": 1
  },
  {
    "type": "number",
    "value": "42",
    "index": 7
  }
]
```

AST

```json
[
  {
    "type": "printStatement",
    "expression": {
      "type": "numberLiteral",
      "value": 42
    }
  }
]
```

wasm

```
0x43 f3.const
0xcd 42 (IEE754)
0xcc
0xbb8
0x41
0x10 call
0x00 0 (LEB 128)
```
print \((42 + 10) / 2\)
print \((42 + 10) / 2\)
const parseExpression = () => {
    let node: ExpressionNode;
    switch (currentToken.type) {
        case "number":
            [...]
            case "parens":
                eatToken();
                const left = parseExpression();
                const operator = currentToken.value;
                eatToken();
                const right = parseExpression();
                eatToken();
                return {
                    type: "binaryExpression",
                    left, right, operator
                };
    }
};
const codeFromAst = ast => {
    const code: number[] = [];

    const emitExpression = (node) => {
        traverse(node, (node) => {
            switch (node.type) {
                case "numberLiteral":
                    code.push(Opcodes.f32_const);
                    code.push(...ieee754(node.value));
                    break;
                case "binaryExpression":
                    code.push(binaryOpcode[node.operator]);
                    break;
            }
        });
    }
    ast.forEach(statement => [...]);
    return code;
};

const binaryOpcode = {
    "+": Opcodes.f32_add,
    "-": Opcodes.f32_sub,
    "*": Opcodes.f32_mul,
    "/": Opcodes.f32_div,
    "==": Opcodes.f32_eq,
    ">": Opcodes.f32_gt,
    "<<": Opcodes.f32_lt,
    "&&": Opcodes.i32_and
};
Demo Time!
chasm v0.3 - variables and while loops
var f = 23
print f

(func (local f32)
  f32.const 23
  set_local 0
  get_local 0
  call 0)
while (f < 10) {
    ...
}
endwhile

(block
    (loop
        [loop condition]
        i32.eqz
        br_if 1
        [nested statements]
        br 0
    )
)
Demo Time!
chasm v1.0 - setpixel
Program Memory

Execution Stack

import / export

push / pop

JavaScript Host
Demo Time!
Recap

- WebAssembly is a relatively simple virtual machine
- It’s a fun playground
- <aside> TypeScript is great! </aside>
- Creating a (simple) compiler isn’t that hard
- A good way to ‘exercise’ your programming skills
- There is a _lot_ of creative energy being poured into WebAssembly
- Hopefully _you_ have been inspired?
Bucket List

☑ Create an open source project
☑ Meet Brendan Eich
☑ Write an emulator
☑ Create my own language and a compiler
Bucket List

☑ Create an open source project
☑ Meet Brendan Eich
☑ Write an emulator
☑ Create my own language and a compiler
☐ ... that supports strings, arrays, functions, lambdas, objects, ...
Build your own WebAssembly Compiler

Colin Eberhardt, Scott Logic

https://github.com/ColinEberhardt/chasm