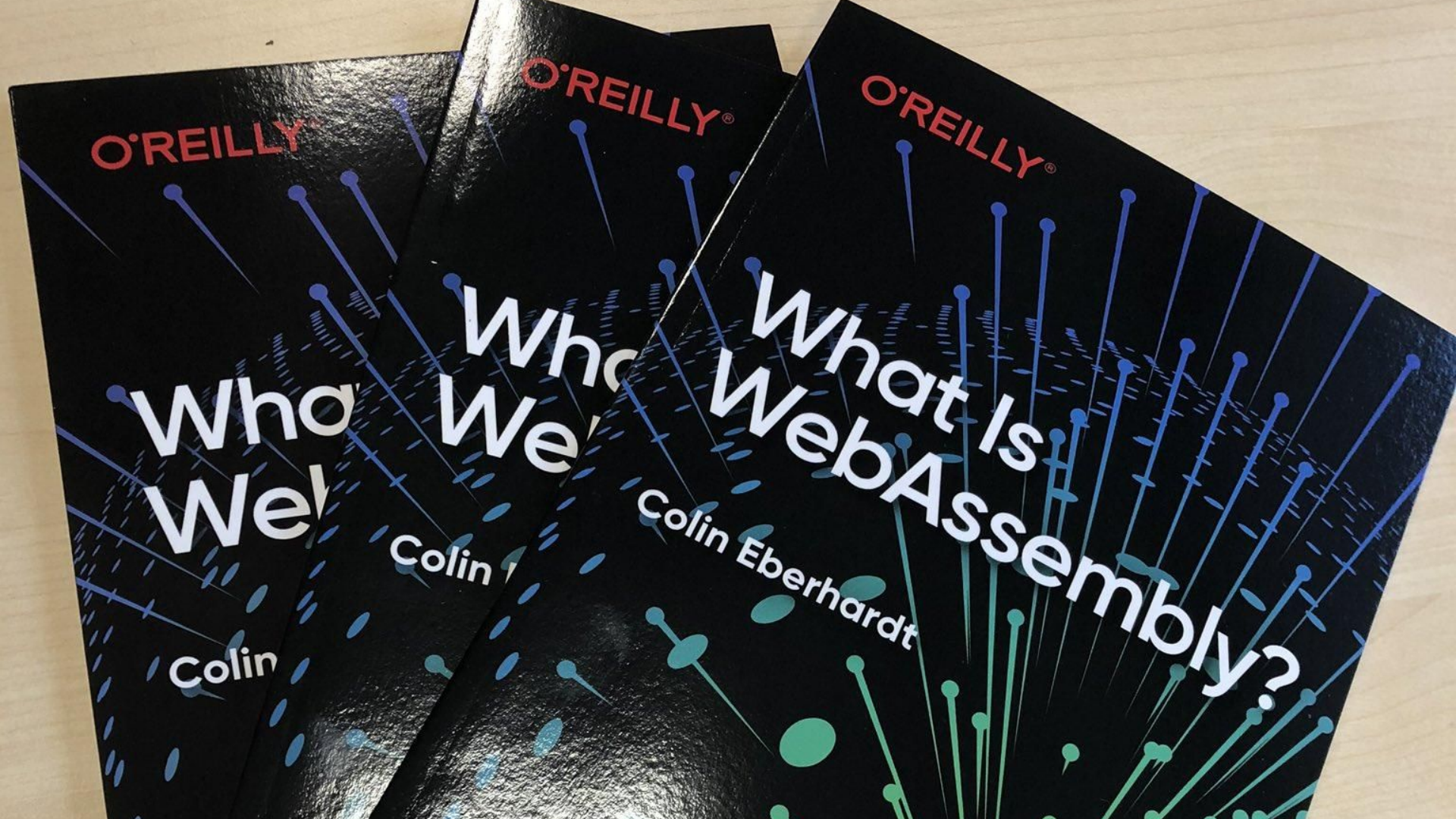
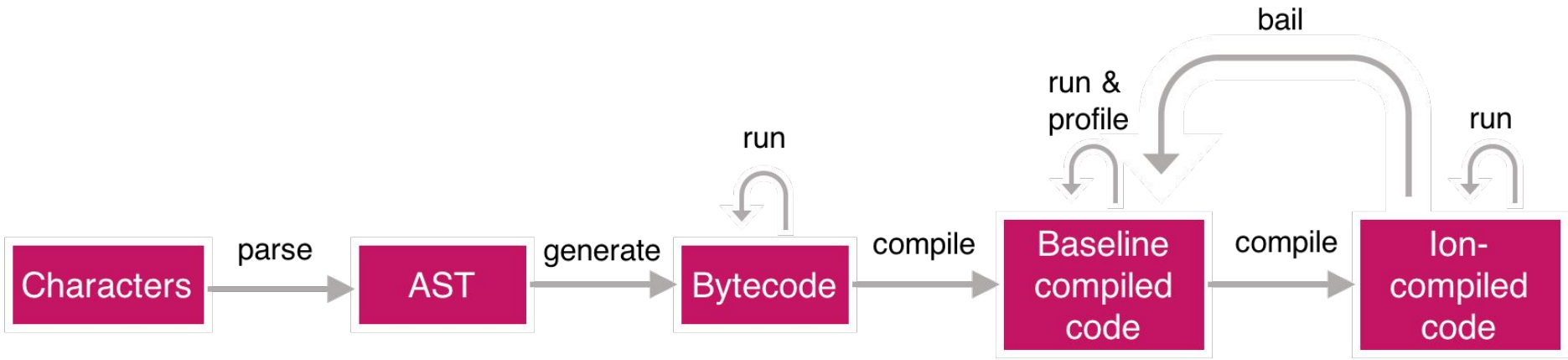
Build your own WebAssembly Compiler

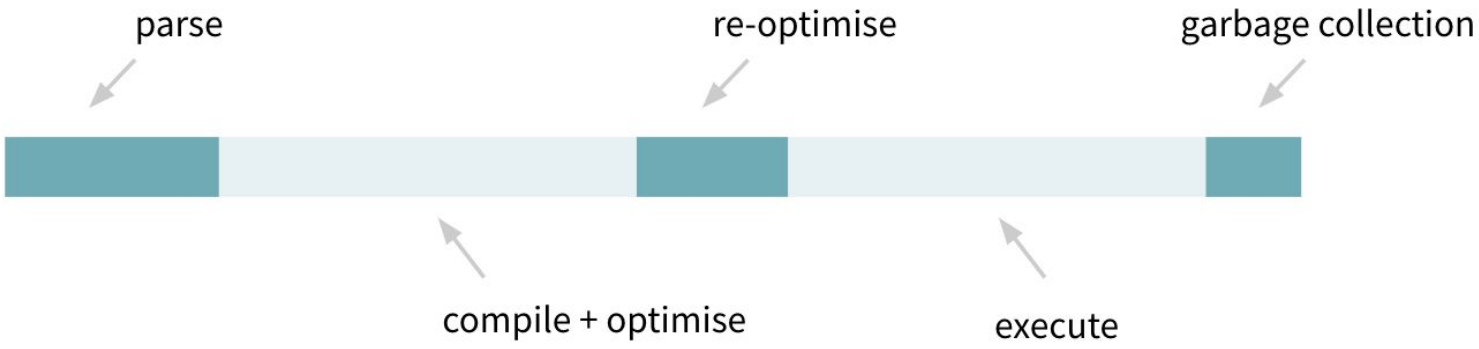
Colin Eberhardt, Scott Logic


O'REILLY
What Is WebAssembly?
Colin Eberhardt

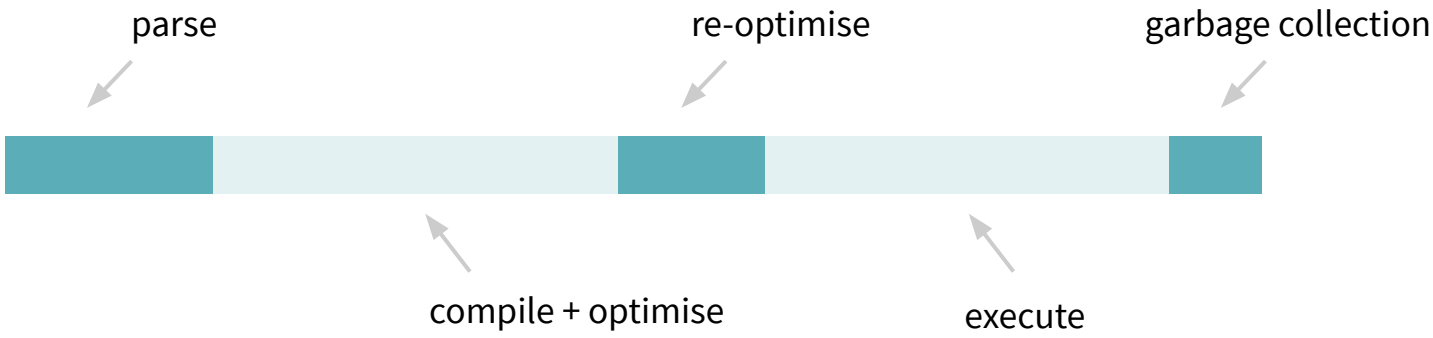
Why do we need WebAssembly?

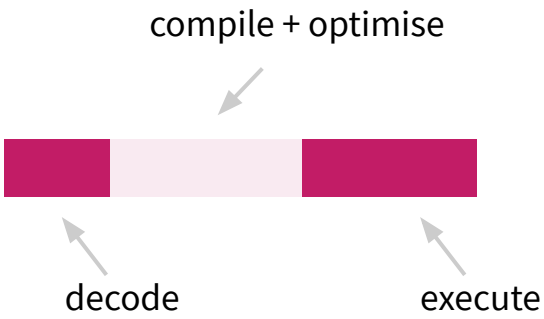

JavaScript is a compilation target


Characters
parse
AST
generate
Bytecode
run
compile
Baseline compiled code
run & profile
bail
compile
Ion-compiled code
run


parse
re-optimise
garbage collection
compile + optimise
execute

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


parse
re-optimise
garbage collection
compile + optimise
execute


compile + optimise
decode
execute

Yehuda Katz 🥨 

@wycats

JavaScript code is much more expensive, byte for byte, than an image, because of the time spent parsing and compiling it.

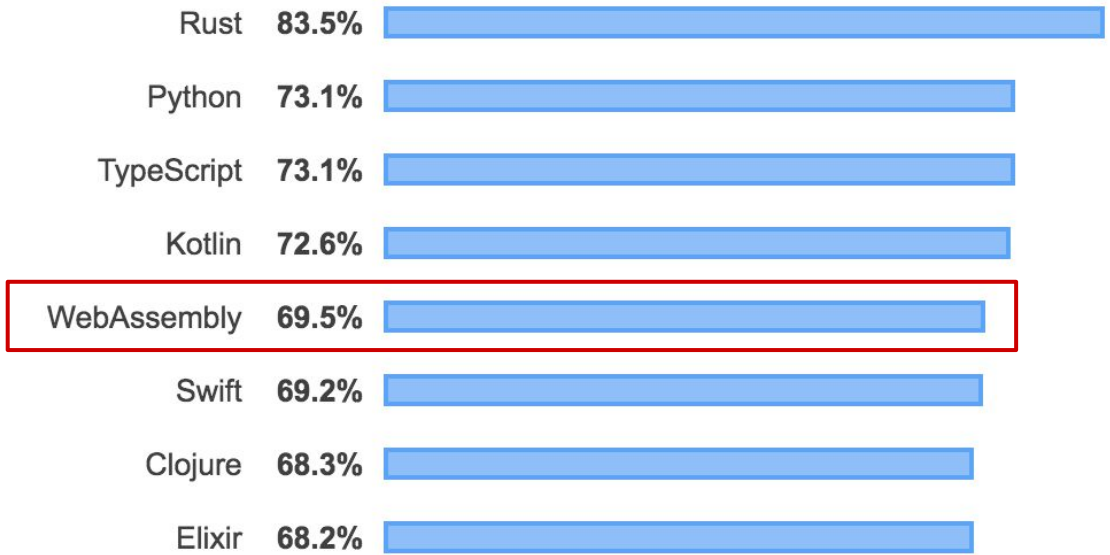
It's possible to parse and compile wasm as fast as it comes over the network, which makes it much more like an image than JavaScript code.

Game changer!

Why create a WebAssembly compiler?

Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted

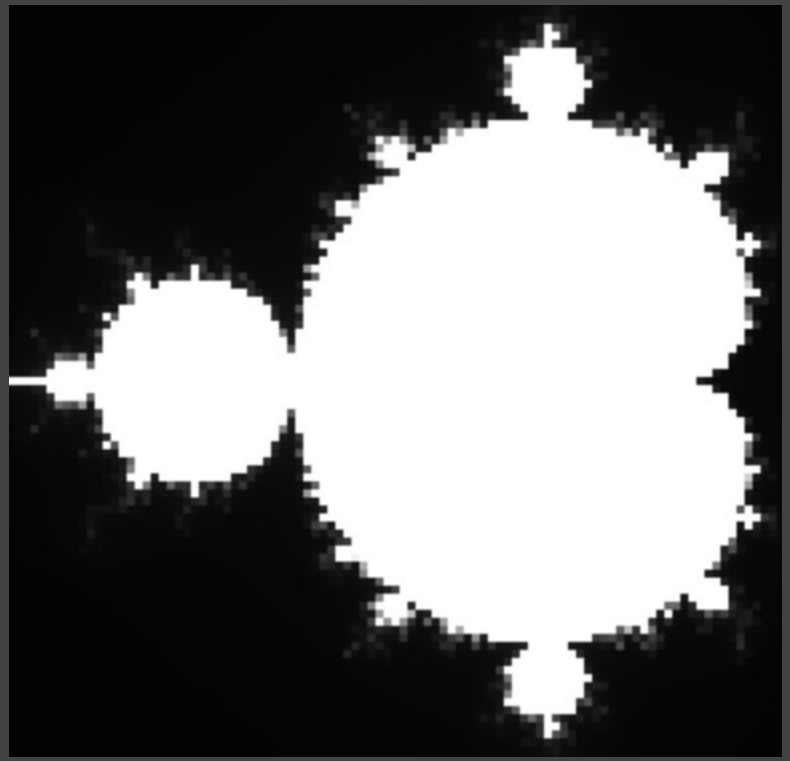


https://insights.stackoverflow.com/survey/2019

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

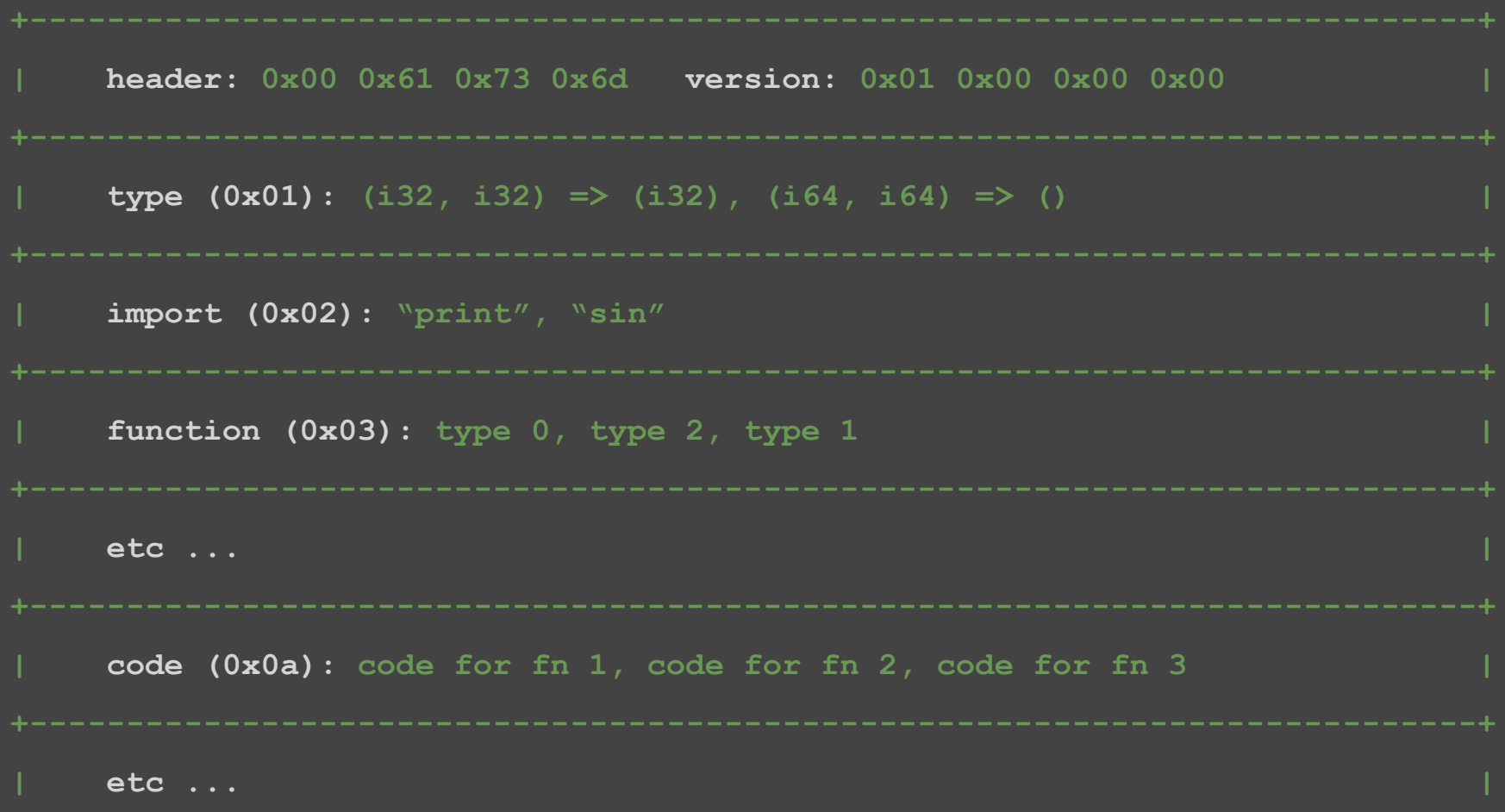
```
const wasm = emitter();

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

```
(module
 (func (param f32) (param f32) (result f32)
   get_local 0
   get_local 1
   f32.add)
 (export "add" (func 0))
)
```

- 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]));
```

get_local 0

get_local 1

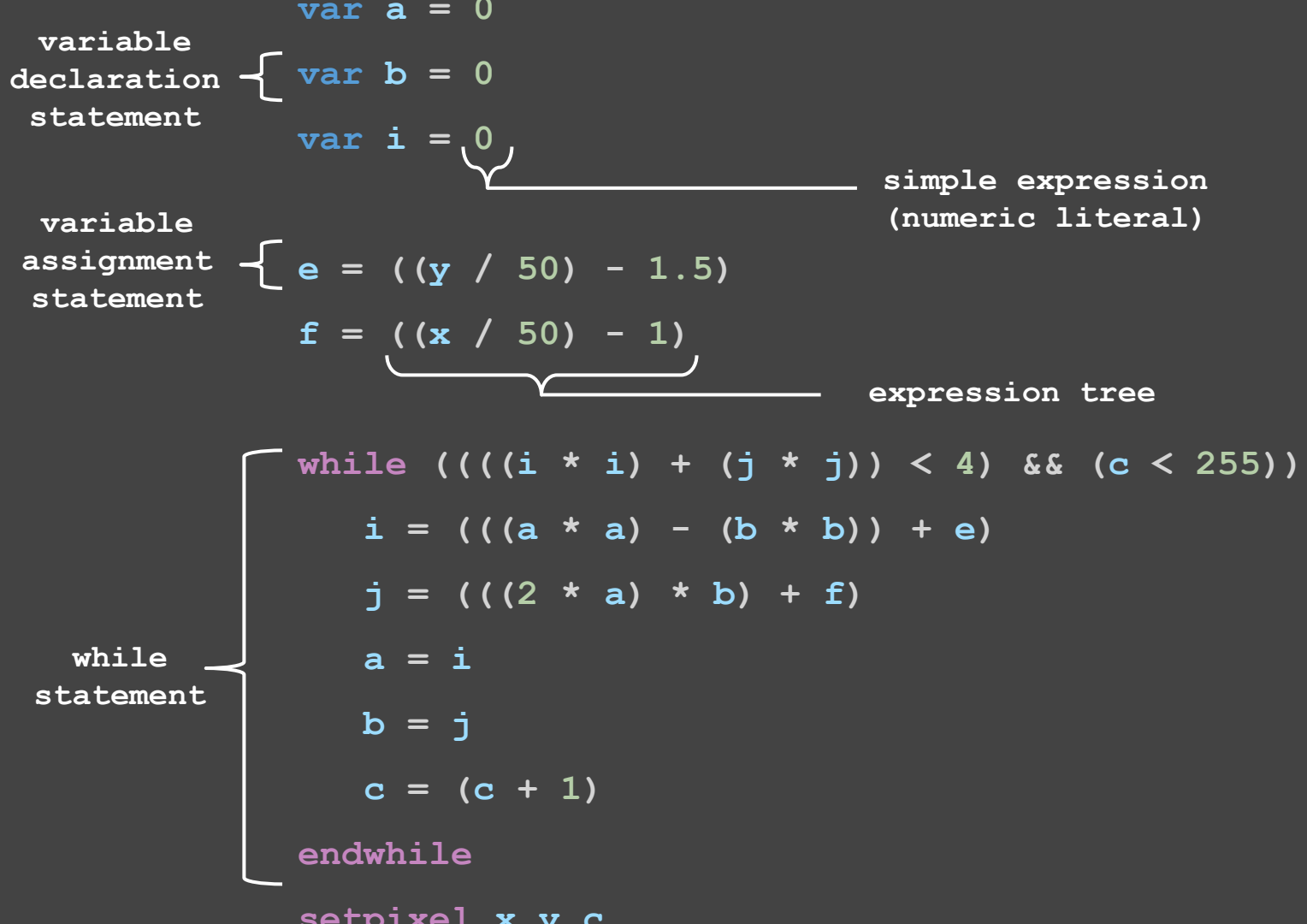
f32.add

function encoding

```
const { instance } = await WebAssembly.instantiate(wasm);
console.log(instance.exports.add(5, 6));
// 11
```

```
$ xxd out.wasm
00000000: 0061 736d 0100 0000 0107 0160 027d 7d01  .asm.......`.}}.
00000010: 7d03 0201 0007 0701 0372 756e 0000 0a09  }........add....
00000020: 0107 0020 0020 0192 0b                   ... . ...
```

Building a compiler


var a = 0
variable declaration statement
var b = 0
var i = 0
simple expression (numeric literal)
variable assignment statement
e = ((y / 50) - 1.5)
f = ((x / 50) - 1)
expression tree
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)
while statement
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"
```

patterns

output

"^[.0-9]+"

"^(print|var)"

"^\\s+"

[]

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
  }
]
```

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
 }
]
```

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


- Removes whitespace
- Basic validation of syntax

Parser

parser

```
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

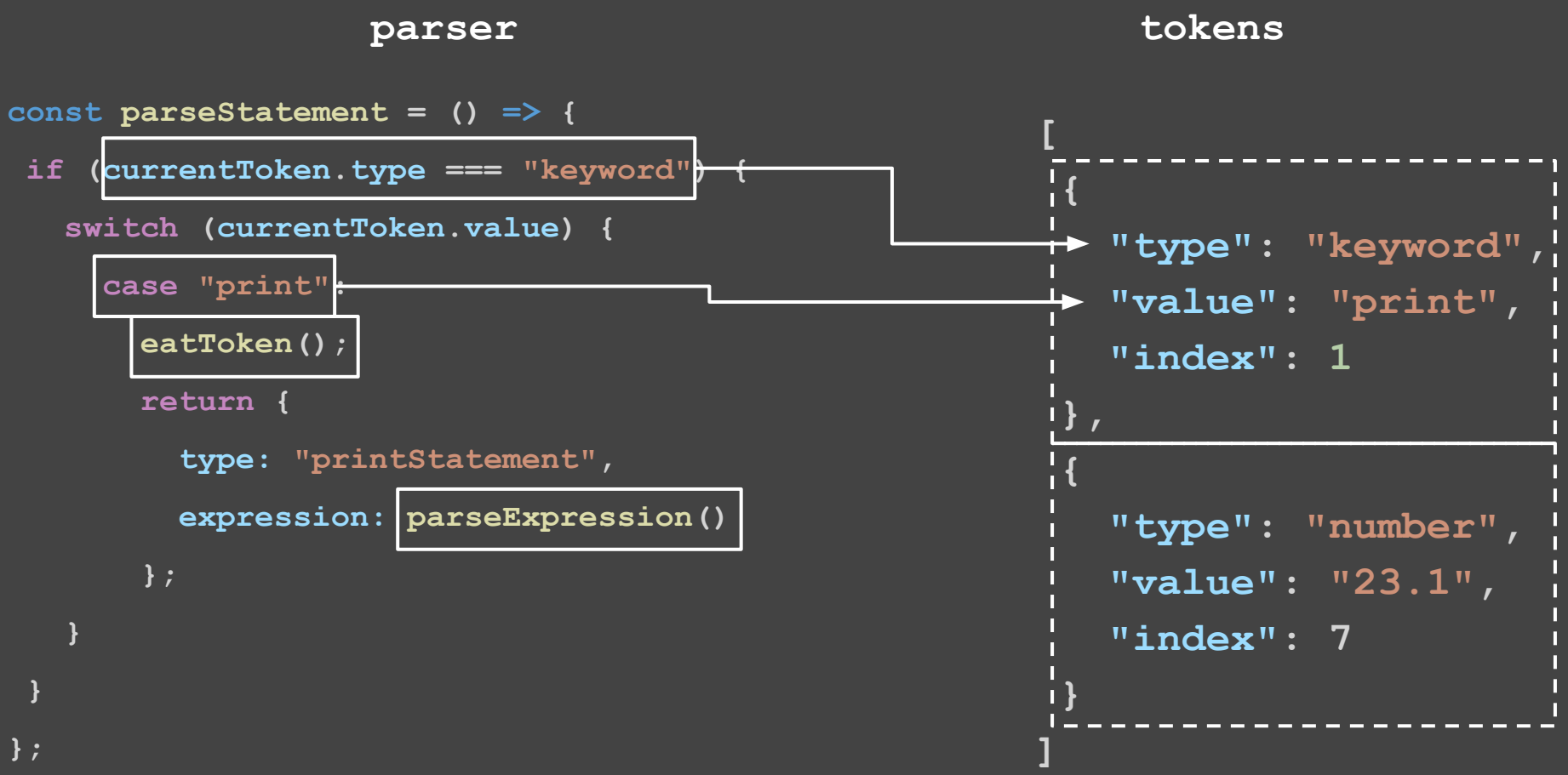
```
[
 {
    "type": "keyword",
    "value": "print",
    "index": 1
 },
 {
    "type": "number",
    "value": "23.1",
    "index": 7
 }
]
```

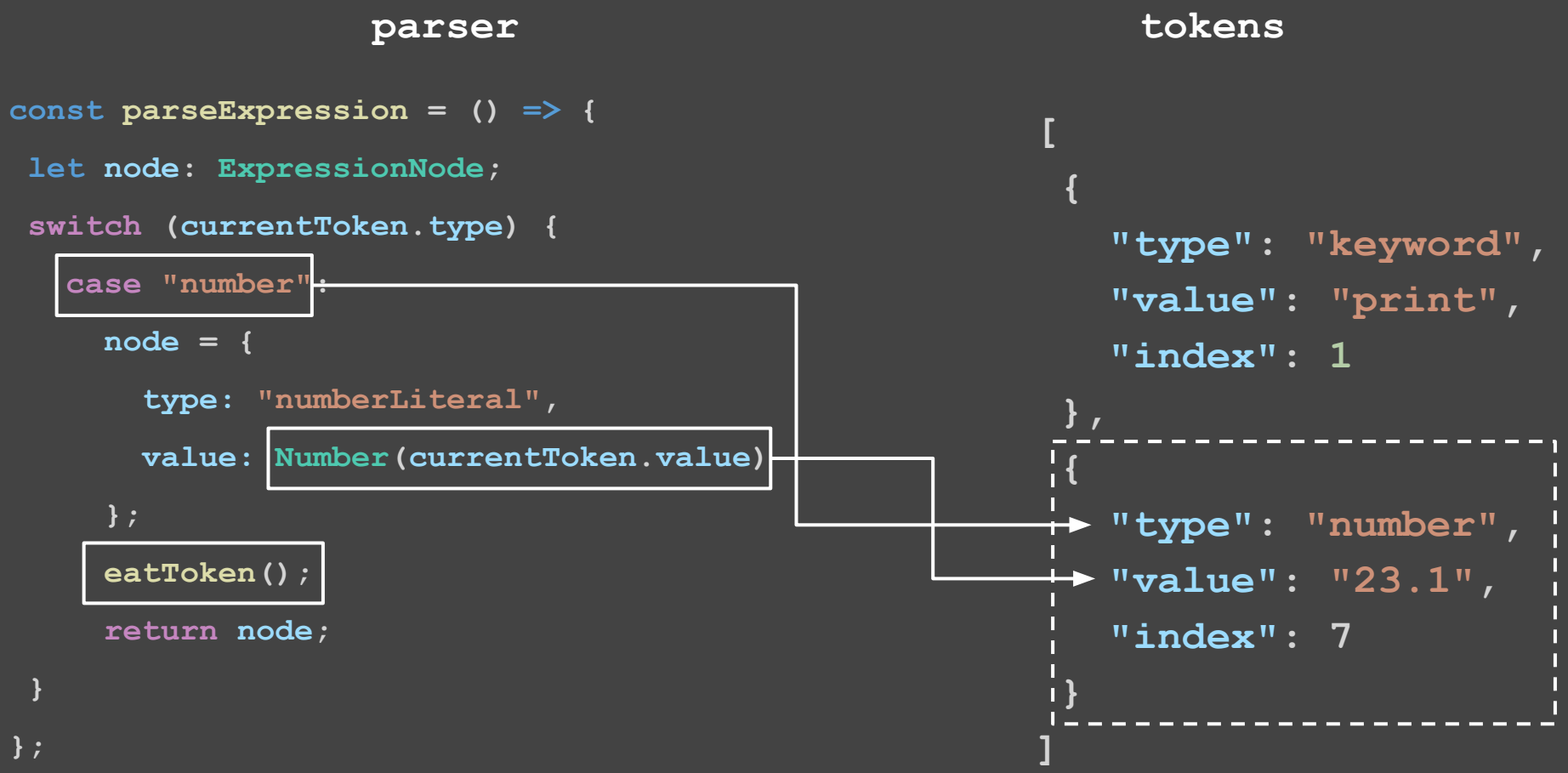
parser

```
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;
};
```

tokens

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


parser
tokens
const parseStatement = () => {
if (currentToken.type === "keyword") {
switch (currentToken.value) {
case "print":
eatToken();
return {
type: "printStatement",
expression: parseExpression()
};
}
};
[
{
"type": "keyword",
"value": "print",
"index": 1
},
{
"type": "number",
"value": "23.1",
"index": 7
}
]


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

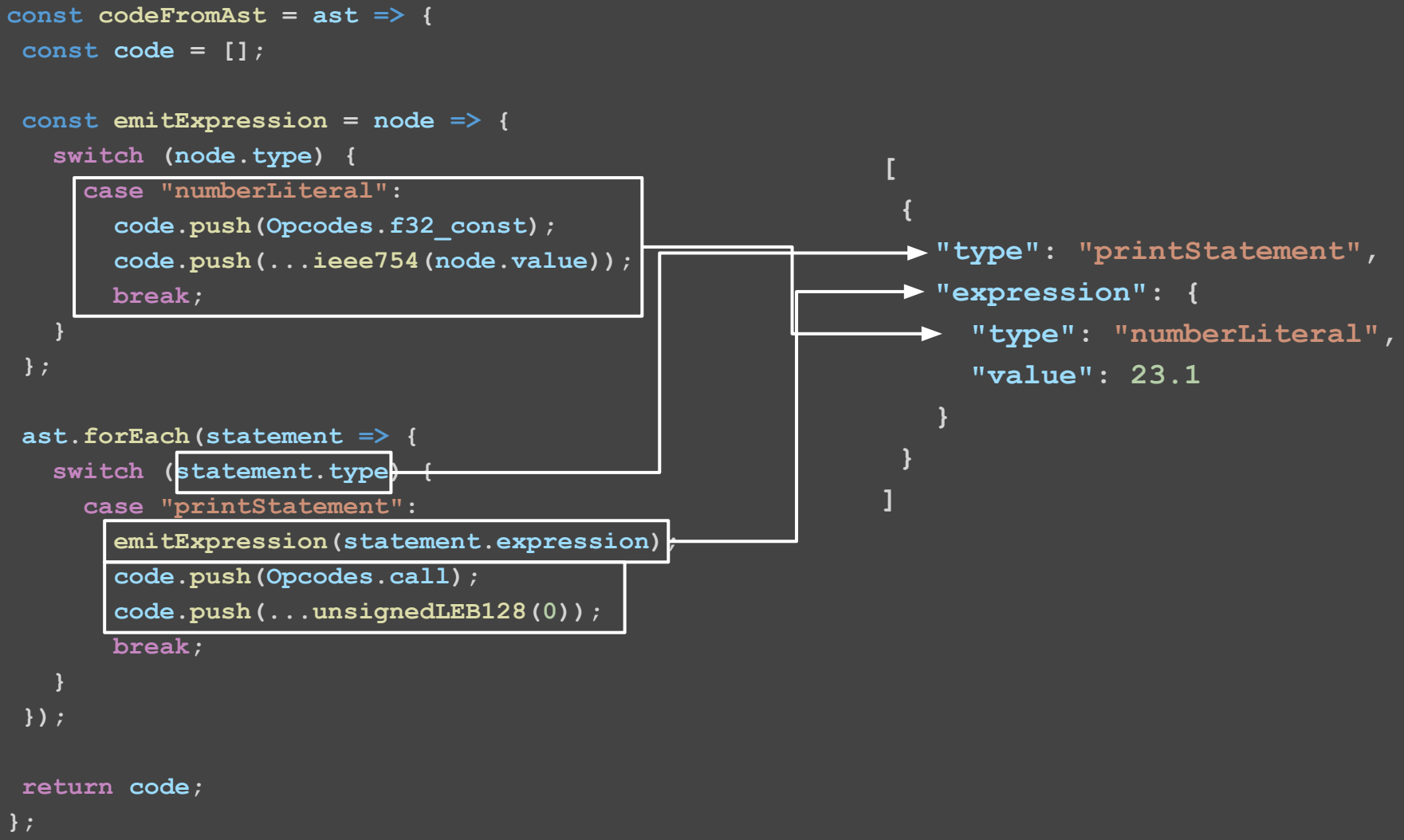
tokens

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

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;
};
[
{
"type": "printStatement",
"expression": {
"type": "numberLiteral",
"value": 23.1
}
}
]

Demo Time!

" print 42"

tokens

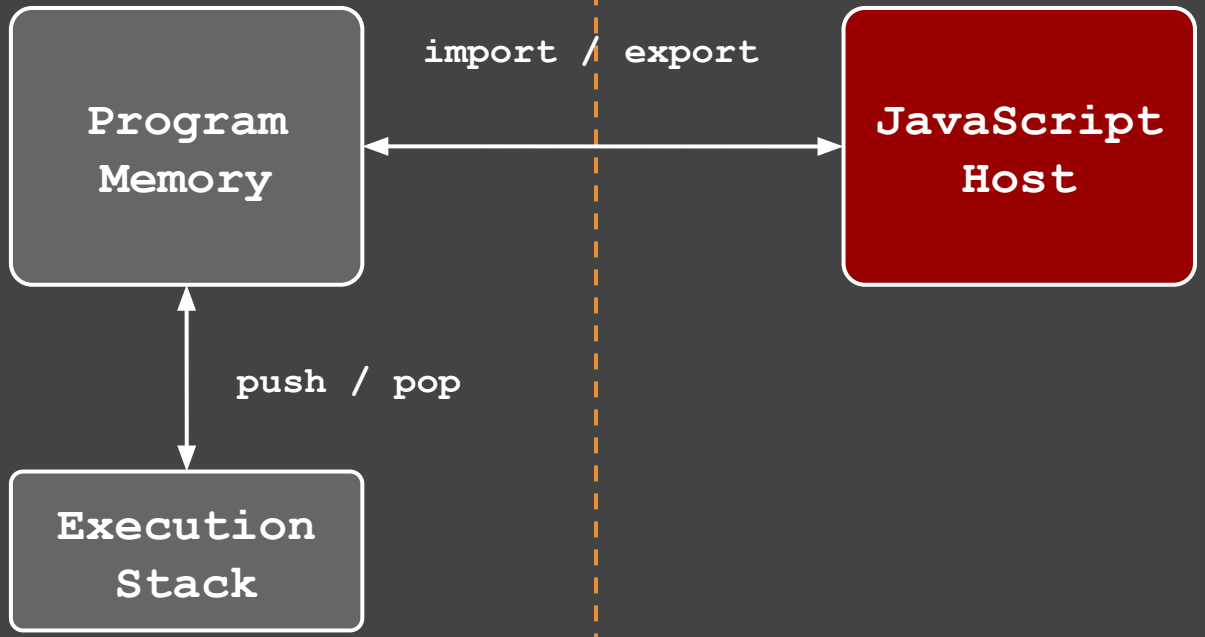
```
[
 {
   "type": "keyword",
   "value": "print",
   "index": 1
 },
 {
   "type": "number",
   "value": "42",
   "index": 7
 }
]
```

AST

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

wasm

```
0x43  f3.const
0xcd  42 (IEE754)
0xcc
0xb8
0x41
0x10  call
0x00  0 (LEB 128)
```


Program Memory
import / export
JavaScript Host
push / pop
Execution Stack

chasm v0.2 - expressions

```
print ((42 + 10) / 2)
```

```
print ((42 + 10) / 2)
```

```
[
 { "type": "keyword", "value": "print" },
 { "type": "parens", "value": "(" },
 { "type": "parens", "value": "(" },
 { "type": "number", "value": "42" },
 { "type": "operator", "value": "+" },
 { "type": "number", "value": "10" },
 { "type": "parens", "value": ")" },
 { "type": "operator", "value": "/" },
 { "type": "number", "value": "2" },
 { "type": "parens", "value": ")" }
]
```

```
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
      };
  }
};
```

```
print ((42 + 10) / 2)
```

```
[{
   type: "printStatement",
   expression: {
     type: "binaryExpression",
     left: {
       type: "binaryExpression",
       left: {
         type: "numberLiteral",
         value: 42
       },
       right: {
         type: "numberLiteral",
         value: 10
       },
       operator: "+"
     },
     right: {
       type: "numberLiteral",
       value: 2
     },
     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;
};
```

depth-first
post-order traversal
(left, right, root)

```
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
```

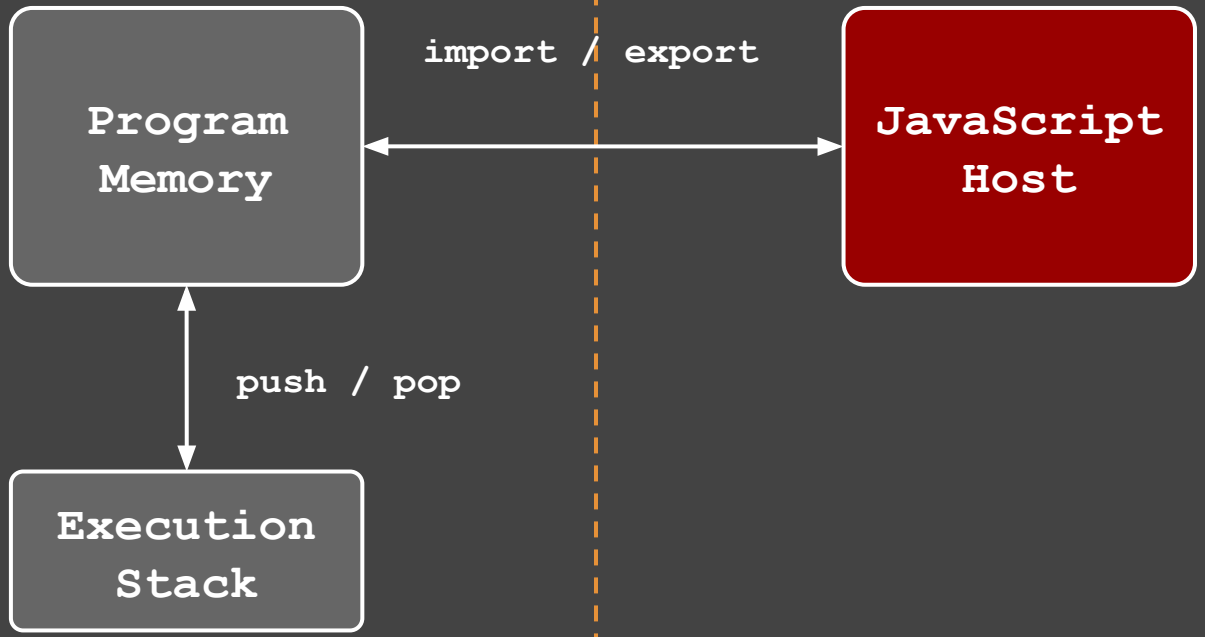
```
(func (local f32)
  f32.const 23
  set_local 0
  get_local 0
  call 0)
```

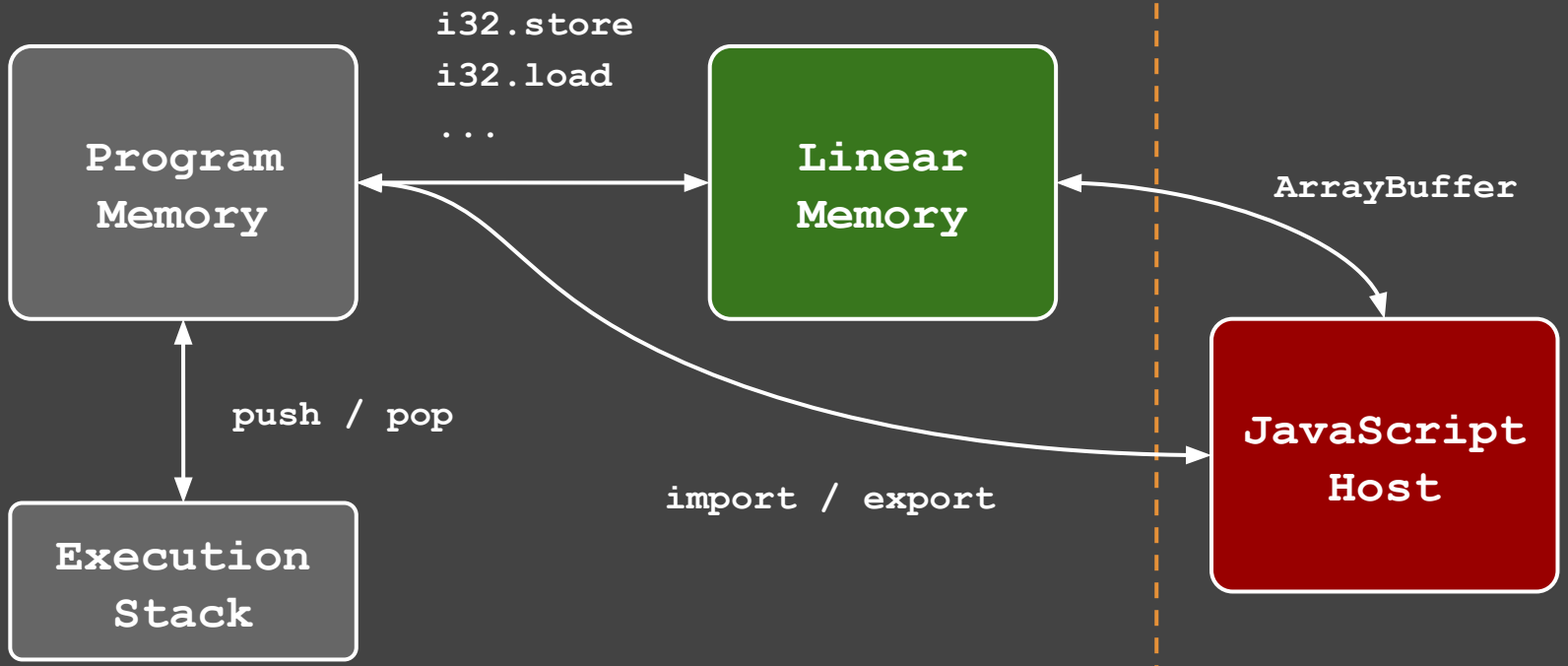
```
while (f < 10)
    ...
endwhile
```

```
(block
 (loop
    [loop condition]
    i32.eqz
    br_if 1
    [nested statements]
    br 0)
 )
```

Demo Time!

chasm v1.0 - setpixel


Program Memory
import / export
JavaScript Host
push / pop
Execution Stack


i32.store
i32.load
...
Program Memory
Linear Memory
ArrayBuffer
push / pop
import / export
Execution Stack
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