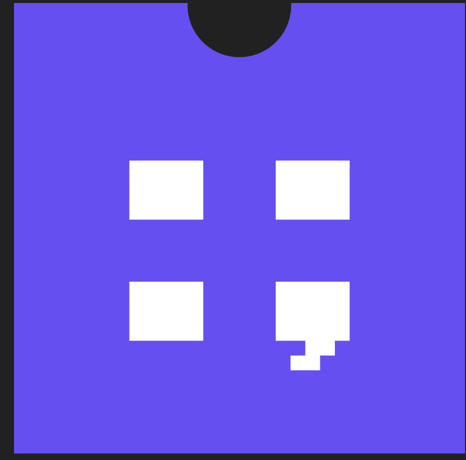


Exploring WebAssembly with Forth (and vice versa)

Artisanal, minimal, just-in-time compilation for the web and beyond

Remko Tronçon
https://mko.re
@remko@mas.to
@remko

Forth

About Forth

- Extremely minimal, stack-oriented programming language & interactive environment
- Released by Charles H Moore in 1970
- Applications
    - Spacecraft controllers
    - Embedded systems
    - Open Firmware (Apple, IBM, Sun, OLPC XO-1)
    - 80's games (Electronic Arts)
    - CollapseOS

Forth

Introduction

```
50 DUP + CONSTANT LARGE
50       CONSTANT SMALL

\ Draw a square
: SQUARE ( n -- )
  4 0 DO
    DUP FORWARD
    90 RIGHT
  LOOP
  DROP
;

SMALL SQUARE
```

- Stack oriented (RPN)
- Syntax?
  - Constants & Variables
  - Comments
  - Function definitions
  - Loops & conditions

Forth

Interpreter loop

```
50 DUP + CONSTANT LARGE
50       CONSTANT SMALL

\ Draw a square
: SQUARE ( n -- )
  4 0 DO
    DUP FORWARD
    90 RIGHT
  LOOP
  DROP
;

SMALL SQUARE
```

- Interpreter loop:
  - Read until next space
  - Known word in dictionary?
    → Execute
  - Number?
    → Push on stack

Forth

:

```
50 DUP + CONSTANT LARGE
50       CONSTANT SMALL

\ Draw a square
: SQUARE ( n -- )
  4 0 DO
    DUP FORWARD
    90 RIGHT
  LOOP
  DROP
;

SMALL SQUARE
```

:

- Read name until next space
- Create new 'in-progress' word in dictionary
- Put interpreter in '**Compilation mode**'

Forth

Compiler mode

```
50 DUP + CONSTANT LARGE
50       CONSTANT SMALL

\ Draw a square
: SQUARE ( n -- )
  4 0 DO
    DUP FORWARD
    90 RIGHT
  LOOP
  DROP
;

SMALL SQUARE
```

- Interpreter loop (Compilation mode):
  - Read until next space
  - Known word in dictionary?
    → Add execution to current word
  - Number?
    → Add code for stack push to current word

Forth

Immediate words

```
50 DUP + CONSTANT LARGE
50       CONSTANT SMALL

\ Draw a square
: SQUARE ( n -- )
  4 0 DO
    DUP FORWARD
    90 RIGHT
  LOOP
  DROP
;

SMALL SQUARE
```

- *Immediate* words
  - Execute during compilation mode
  - (→ Consume input stream until)
  - ; → Finalize word & switch to interpreter mode
  - DO, LOOP → Keep track of jump locations
  - Create your own language constructs

Forth

```
50 DUP + CONSTANT LARGE
50       CONSTANT SMALL

\ Draw a square
: SQUARE ( n -- )
  4 0 DO
    DUP FORWARD
    90 RIGHT
  LOOP
  DROP
;

SMALL SQUARE
```

- Simple intererpreter loop
  - Integrated compiler
- No syntax, everything in word definitions
- Extensible compiler
- Attractive to create a compiler/ interpreter for new low-level systems

WebAssembly

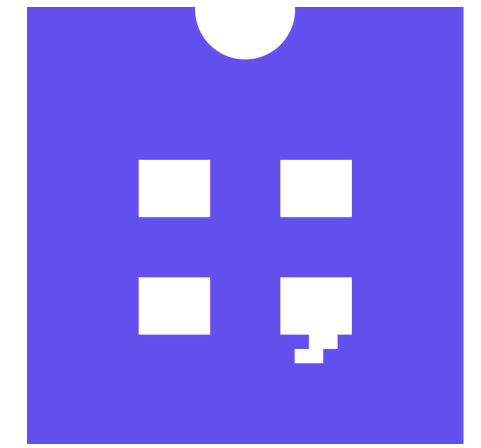
About WebAssembly

- Open standard for portable binary code
- Supported by most browsers & languages
  - Run any language in your browser
- Not web-specific

WAForth

WAForth

About



- Small Forth system, hand-written in WebAssembly, compiling to WebAssembly
- Goals
  - ✅ **WebAssembly-first**: As much as possible in WebAssembly
    - No I/O or module loading
  - ✅ **Simple**: 1 file
  - ✅ **Complete**: ANS Core & (most) Core Extension words
  - ❓ **Speed**
  - ❓ **Binary size** (14kB)
  - ❓ **Ease of use**

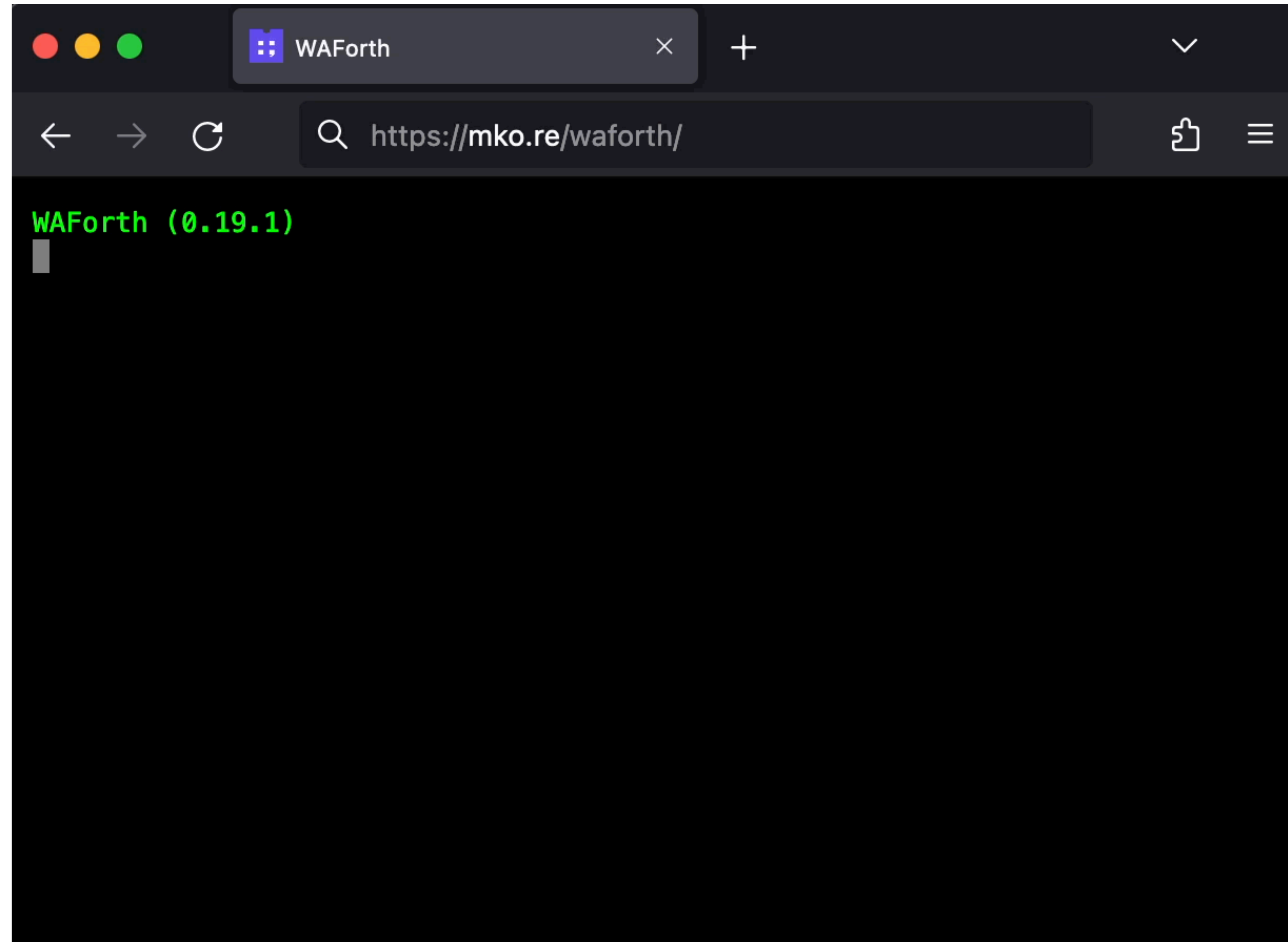
WAForth

Running Forth in JavaScript

```
import WAForth, { withLineBuffer } from "waforth";

(async () => {
  // Create the UI
  document.body.innerHTML = `<button>Go!</button><pre></pre>`;
  const btn = document.querySelector("button");
  const log = document.querySelector("pre");

  // Initialize WAForth
  const forth = new WAForth();
  forth.onEmit = withLineBuffer((c) =>
    log.appendChild(document.createTextNode(c)));
  await forth.load();

  // Bind "prompt" call to a function that pops up a JavaScript
  // prompt, and pushes the entered number back on the stack
  forth.bind("prompt", (stack) => {
    const message = stack.popString();
    const result = window.prompt(message);
    stack.push(parseInt(result));
  });

  // Load Forth code to bind the "prompt" call to a word,
  // and call the word
  forth.interpret(`
( Call "prompt" with the given string )
: PROMPT ( c-addr u -- n )
  S" prompt" SCALL
;

( Prompt the user for a number, and write it to output )
: ASK-NUMBER ( -- )
  S" Please enter a number" PROMPT
  ." The number was" SPACE .
;
`);

  btn.addEventListener("click", () => {
    forth.interpret("ASK-NUMBER");
  });
})();
```

WAForth

Interactive Console



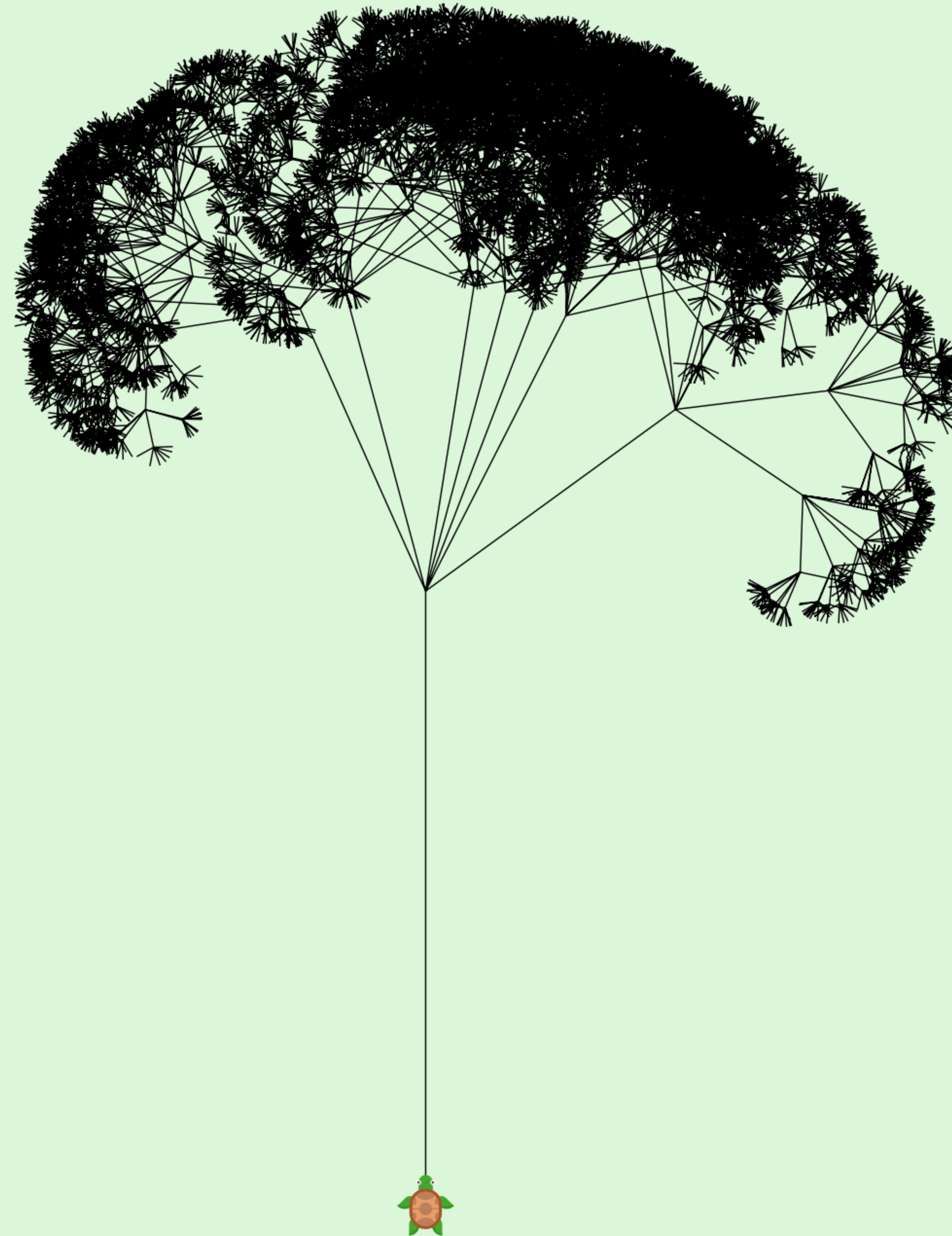
https://mko.re/waforth

WAForth

Thurtle

Plant

```
450 CONSTANT SIZE
7   CONSTANT BRANCHES
160 CONSTANT SPREAD

VARIABLE RND
HERE RND !

: RANDOM ( -- n )
  RND @ 75 * 74 + 65537 MOD
  DUP RND !
;

: CHOOSE ( n1 -- n2 )
  RANDOM 65537 */MOD SWAP DROP
;

: PLANT ( size angle -- )
  OVER 10 < IF 2DROP EXIT THEN
  DUP RIGHT
  OVER FORWARD
  BRANCHES 0 DO
    OVER 2/
    SPREAD CHOOSE SPREAD 2/ -
    RECURSE
  LOOP
  PENUP SWAP BACKWARD PENDOWN
  LEFT
;

1 SETPENSIZE
SIZE 0 PLANT
```

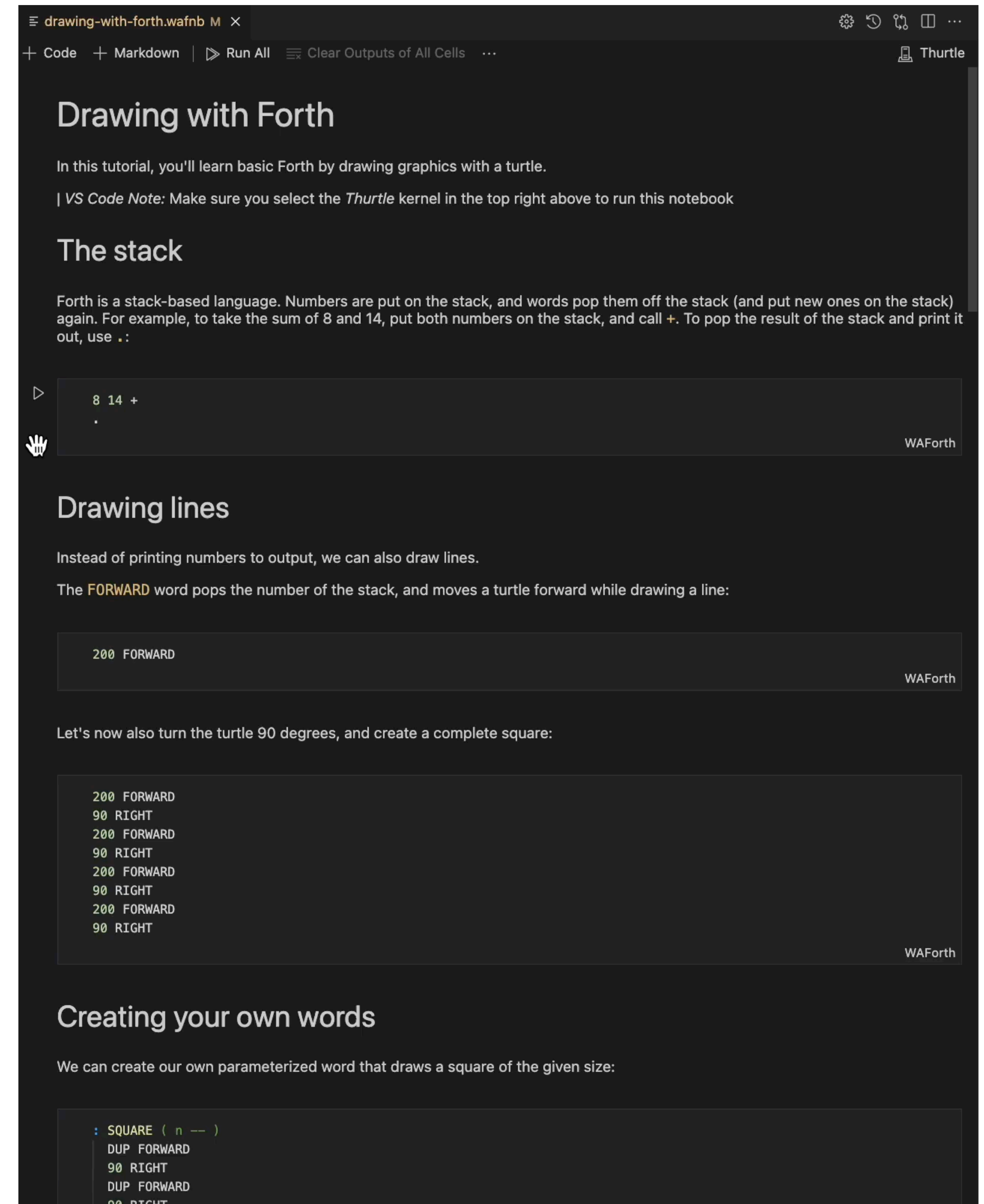


https://mko.re/thurtle

WAForth

Notebook

- VS Code Notebook Extension
- Standalone
  - https://mko.re/wafnb/drawing-with-forth/
  - 60k standalone HTML file



WAForth Internals

WAForth Internals

Interpreter

- WebAssembly Text Format
  - S-Expressions
- WebAssembly Binary Toolkit (WABT) `wat2wasm`
  - Flatten code into sequence binary instructions

```
(block $endLoop
  (loop $loop
    ;; Parse the next name in the input stream
    (call $parseName)
    (local.set $wordLen) (local.set $wordAddr)

    ;; Break the loop if we didn't parse anything
    (br_if $endLoop (i32.eqz (local.get $wordLen)))

    ;; Find the name in the dictionary
    ;; Besides the code address (aka token), also returns a constant whether the word
    ;; was found (!= 0), and if so, whether it was immediate (1) or not (-1).
    (call $find (local.get $wordAddr) (local.get $wordLen))
    (local.set $findResult) (local.set $findToken)
    (if (local.get $findResult)
      (then
        ;; Name found in the dictionary.
        (block
          ;; Are we interpreting? Then jump out of this block
          (br_if 0 (i32.eqz (call $getState)))
          ;; Is the word immediate? Then jump out of this block
          (br_if 0 (i32.eq (local.get $findResult) (i32.const 1)))

          ;; We're compiling a non-immediate.
          ;; Compile the execution of the word into the current compilation body.
          (call $compileExecute (local.get $findToken))
          (br $loop))

        ;; We're interpreting, or this is an immediate word
        ;; Execute the word.
        (local.set $tos (call $execute (local.get $tos) (local.get $findToken))))
      (else
          ;; Name is not in the dictionary. Is it a number?
          ;; `readNumber` leaves the unparsed count and the parsed number on the stack.
          (if (i32.eqz (call $readNumber (local.get $wordAddr) (local.get $wordLen)))
            ;; It's a number.
            (then
              (local.set $number)

              ;; Are we compiling?
              (if (call $getState)
                (then
                  ;; We're compiling. Add a push of the number to the current compilation body.
                  (call $compilePushConst (local.get $number))))
                (else
                  ;; We're not compiling. Put the number on the stack.
                  (call $push (local.get $number)))))
            ;; It's not a number either. Fail.
            (else
              (drop)
              (call $failUndefinedWord (local.get $wordAddr) (local.get $wordLen))))))
      (br $loop))))
```

WAForth Internals

Compiler

- Hard-coded binary header of WebAssembly module with 1 function
  - Placeholders to be filled in
- `: TWICE 2 * ;`
  - Reset placeholders
  - CP → End of header
- In compilation mode, add raw binary opcodes to header (@ CP)

```
(data (i32.const 0x1000
  "\00\61\73\6D" ;; Header
  "\01\00\00\00" ;; Version

  "\01" "\12" ;; Type section
    "\03" ;; #Entries
      "\60\01\7f\01\7f" ;; (func (param i32) (result i32))
      "\60\02\7f\7f\01\7f" ;; (func (param i32) (param i32) (result i32))
      "\60\01\7f\02\7F\7f" ;; (func (param i32) (result i32) (result i32))


  "\02" "\20" ;; Import section
    "\02" ;; #Entries
    "\03\65\6E\76" "\05\74\61\62\6C\65" ;; 'env' . 'table'
      "\01" "\70" "\00" "\FB\00\00\00" ;; table, funcref, flags, initial size
    "\03\65\6E\76" "\06\6d\65\6d\6f\72\79" ;; 'env' . 'memory'
      "\02" "\00" "\01" ;; memory

  "\03" "\02" ;; Function section
    "\01" ;; #Entries
    "\FA" ;; Type 0

  "\09" "\0a" ;; Element section
    "\01" ;; #Entries
    "\00" ;; Table 0
    "\41\FC\00\00\00\0B" ;; i32.const ..., end
    "\01" ;; #elements
      "\00" ;; function 0

  "\0A" "\FF\00\00\00" ;; Code section (padded length)
  "\01" ;; #Bodies
    "\FE\00\00\00" ;; Body size (padded)
    "\01" ;; #locals
      "\FD\00\00\00\7F")  ;; # #i32 locals (padded)
```

CP →

WAForth Internals

Compiler

- In compilation mode, add raw binary opcodes to header (@ CP)
- Control flow instructions

Interpreter loop

```
;; Find the name in the dictionary
;; Besides the code address (aka token), also returns a constant whether the word
;; was found (!= 0), and if so, whether it was immediate (1) or not (-1).
(call $find (local.get $wordAddr) (local.get $wordLen))
(local.set $findResult) (local.set $findToken)
(if (local.get $findResult)
  (then
    ;; Name found in the dictionary.
    (block
      ;; Are we interpreting? Then jump out of this block
      (br_if 0 (i32.eqz (call $getState)))
      ;; Is the word immediate? Then jump out of this block
      (br_if 0 (i32.eq (local.get $findResult) (i32.const 1)))

      ;; We're compiling a non-immediate.
      ;; Compile the execution of the word into the current compilation body.
      (call $compileExecute (local.get $findToken))
```

```
(func $compileExecute (param $index i32)
  (call $emit (i32.const 0x41)) ;; `i32.const` instruction
  (call $emit (local.get $index))
  (call $emit (i32.const 0x11)) ;; `call_indirect` instruction (index is on the stack)
  (call $emit (i32.const 0x1))) ;; function type operand of `call_indirect` instruction

(func $emit (param $v i32)
  (i32.store8 (global.get $cp) (local.get $v))
  (global.set $cp (i32.add (local.get $cp) (i32.const 1)))
```

```
i32.const <index>     0x41 <index>
call_indirect 1       0x11 0x1
```

```
(call_indirect 1 (i32.const <index>))
```

WAForth Internals

Loader

- `: TWICE 2 * ;`
  - Load generated binary module into runtime
  - Pass pointer to generated code to host
  - Host uses WebAssembly API to load binary module
  - Record current function index into dictionary + Return interpreter to execution mode

WAForth

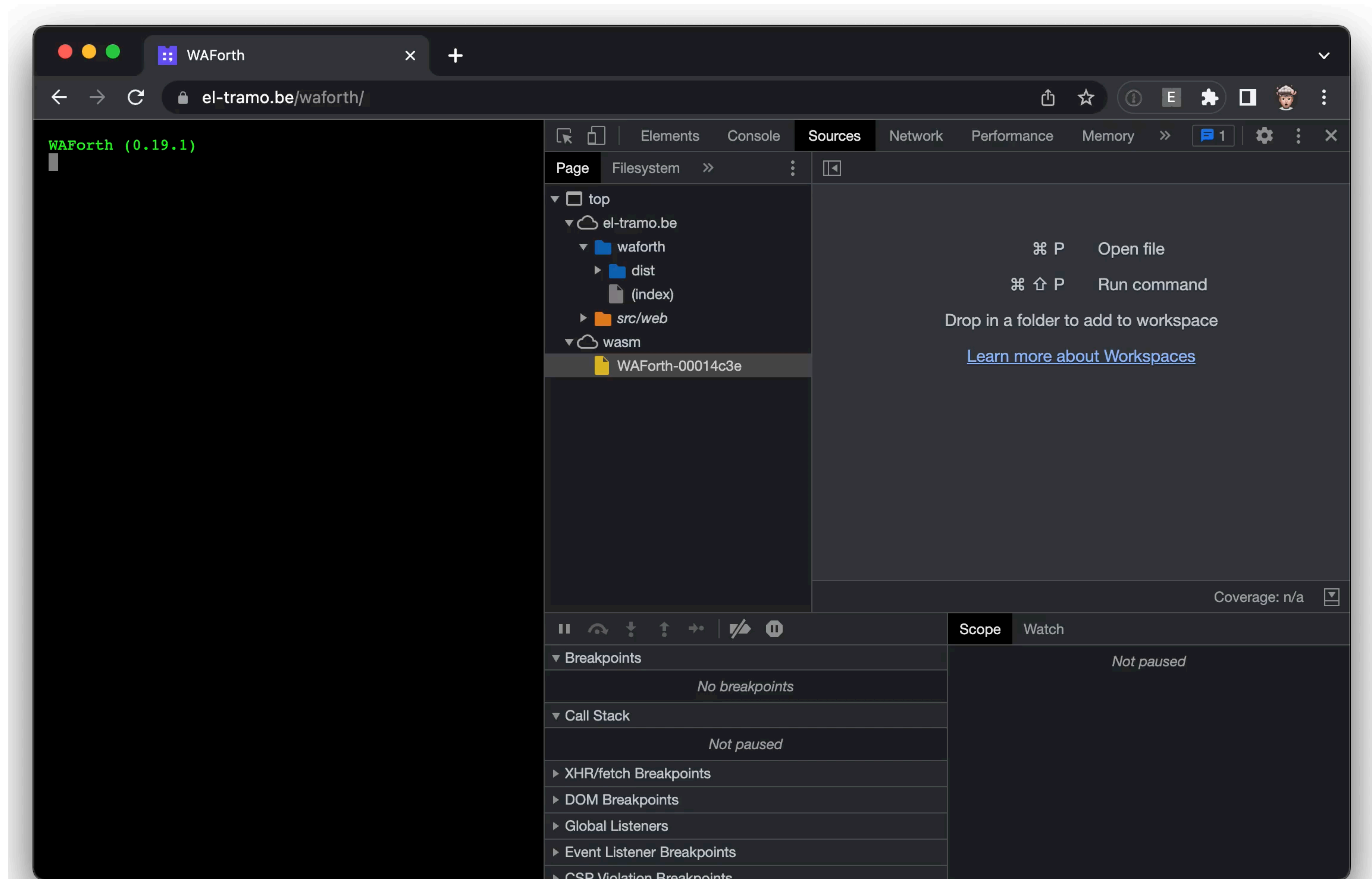
```
;; Load a webassembly module.
;; Parameters: WASM bytecode memory offset, size
(import "host" "load" (func $load (param i32 i32)))

(func $semicolon
  ...
  (call $load (i32.const 0x1000) (i32.sub (global.get $cp) (i32.const 0x1000)))
```

JavaScript

```
function load(offset, length) => {
  const module = new WebAssembly.Module(new Uint8Array(
    forthSystemModule.memory.buffer,
    offset,
    length
  ));
  new WebAssembly.Instance(module, { env: { table, memory } });
}
```

WAForth

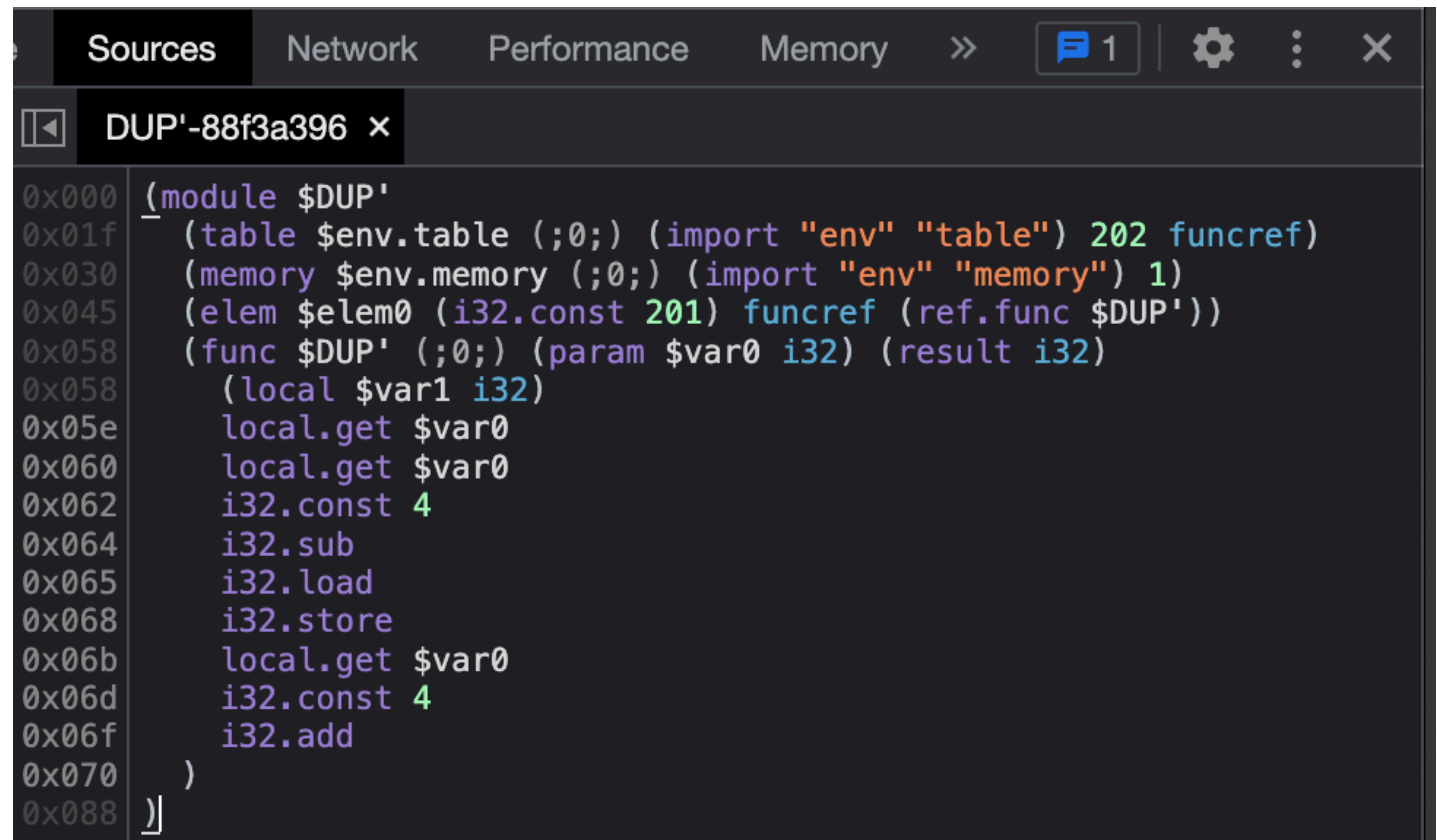
Interactive System



WAForth

Writing WebAssembly in Forth

```
CODE DUP' ( n -- n n )
  [ 0 ] $LOCAL.GET

  [ 0 ] $LOCAL.GET
  [ 4 ] $I32.CONST
  $I32.SUB
  $I32.LOAD
  $I32.STORE

  [ 0 ] $LOCAL.GET
  [ 4 ] $I32.CONST
  $I32.ADD
;CODE
```



WebAssembly Function Tables

- Most Forths use threaded code
- WebAssembly does not allow random or dynamic jumps
- WebAssembly Function Tables
  - Indirect calls (to index in dictionary)
- Subroutine threading
  - Less efficient

WebAssembly Function Table

index	function	type
…		
23	$foo	(i32, i32) → i32
…		

```
(call $foo)

(call_indirect (i32.const 23))
```

Measurements

Benchmark

Sieve of Eratosthenes

```
: prime? HERE + C@ 0= ;
: composite! HERE + 1 SWAP C! ;
: sieve
  HERE OVER ERASE
  2
  BEGIN
    2DUP DUP * >
  WHILE
    DUP prime? IF
      2DUP DUP * DO
        I composite!
      DUP +LOOP
    THEN
    1+
  REPEAT
  DROP
  1 SWAP 2 DO I prime? IF DROP I THEN LOOP .
;
```

JavaScript

```
function sieve(n) {
  const nums = new Uint8Array(n + 1);
  for (let i = 2; i * i <= n; i++) {
    if (!nums[i]) {
      for (let j = i * i; j <= n; j += i) {
        nums[j] = 1;
      }
    }
  }
  let lastPrime = 0;
  for (let i = 2; i < n; i++) {
    if (!nums[i]) {
      lastPrime = i;
    }
  }
  return lastPrime;
}
```

Benchmark

Speed

- Run sieve 90.000.000 times

	Time (s)
jsForth (Chrome 107)	55
jsForth (Safari 16.1)	110
WAForth (Safari 16.1)	7,65
WAForth (Chrome 107)	5,73
WAForth (Firefox 108.1)	5,74
Gforth	5,49
JavaScript	0,55
Raw WebAssembly	0,46
C + Emscripten	0,46
C	0,40

Benchmark

Size

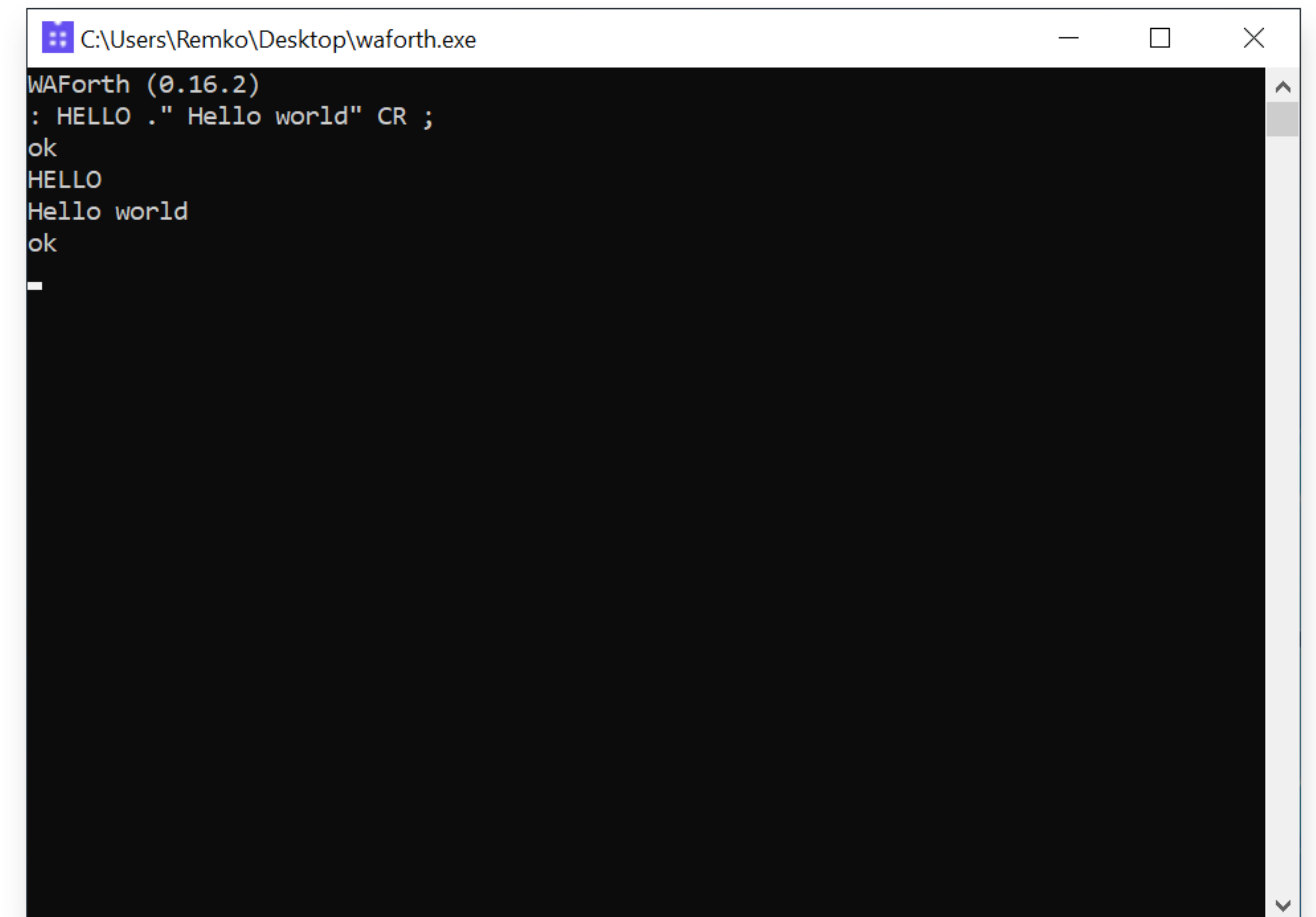
	Size (Code/Runtime)
jsForth	1.1MB (JS) + browser
WAForth	33kB (15k WebAssembly + 18k JS + overhead) + browser
Raw WebAssembly (Sieve only)	190b + browser
C + Emscripten (Sieve only)	27kB (8k WebAssembly + 19kB JS + overhead) + browser
Gforth	350kB
C (Sieve only)	33kB

Beyond the web

Beyond the Web

WAForth Standalone

- Avoid need for browser
- WebAssembly standalone implementations
  - WABT Interp (C++)
  - Wasmtime (Rust)
  - WAMR
- WebAssembly C API
- WAForth Standalone
  - 200 lines of implementation-independent C



WAForth Standalone

Benchmark

	Time (s)	Size
WAForth Standalone (WABT interp)	320	986k (+ libc++)
WAForth Standalone (wasmtime)	34,56	19Mb
WAForth (Chrome 107)	5,73	33kB (15k WebAssembly + 18k JS + overhead) + browser
Gforth	5,49	350kB
JavaScript	0,55	
Raw WebAssembly	0,46	190B + browser
C + Emscripten	0,46	27kB (8k WebAssembly + 19kB JS + overhead) + browser
C	0,40	33kB

From JIT to AOT

WAForthC

- Get rid of WebAssembly runtime
- Use WAForth to run Forth code once (using WABT)
- Keep track of compiled words
- Combine current state & compiled words in 1 WebAssembly binary module
- Use WABT wasm2c to compile into C
- Use host compiler to create native executable
  - Still has all Forth functionality, but no compiler
- Bonus: Cross-compilation

```
$ waforthc --cc=arm-linux-gnueabi-gcc --ccflag=-static
--ccflag=-O2 --output=hello --init=SAY_HELLO hello.fs↩
```

```
$ cat hello.fs↩
: SAY_HELLO  ." Hello, Forth" CR ;
SAY_HELLO

$ waforthc --output=hello hello.fs↩
Hello, Forth
```

```
$ ./hello↩
SAY_HELLO↩
Hello, Forth
ok
4 2 * .↩
8 ok
: SAY_BYE ." Bye" CR ;
Compilation is not available in native compiled mode
```

WAForthC

Benchmark

	Time (s)	Size
WAForth Standalone (WABT interp)	320	986k (+ libc++)
WAForth Standalone (wasmtime)	34,56	19Mb
WAForth (Chrome 107)	5,73	33kB (15k WebAssembly + 18k JS + overhead) + browser
Gforth	5,49	350kB
Waforthc (wasm2c)	3,31	116kB
JavaScript	0,55	
Raw WebAssembly	0,46	190B + browser
C + Emscripten	0,46	27kB (8k WebAssembly + 19kB JS + overhead) + browser
C	0,40	33kB

WAForthC

Future work

- Post-processing combined module
  - Replace indirect word calls by direct word calls
  - Dead-code elimination

Conclusion

- Explore Forth & WebAssembly

The **Forth**
is my ally, and a
powerful ally it is.