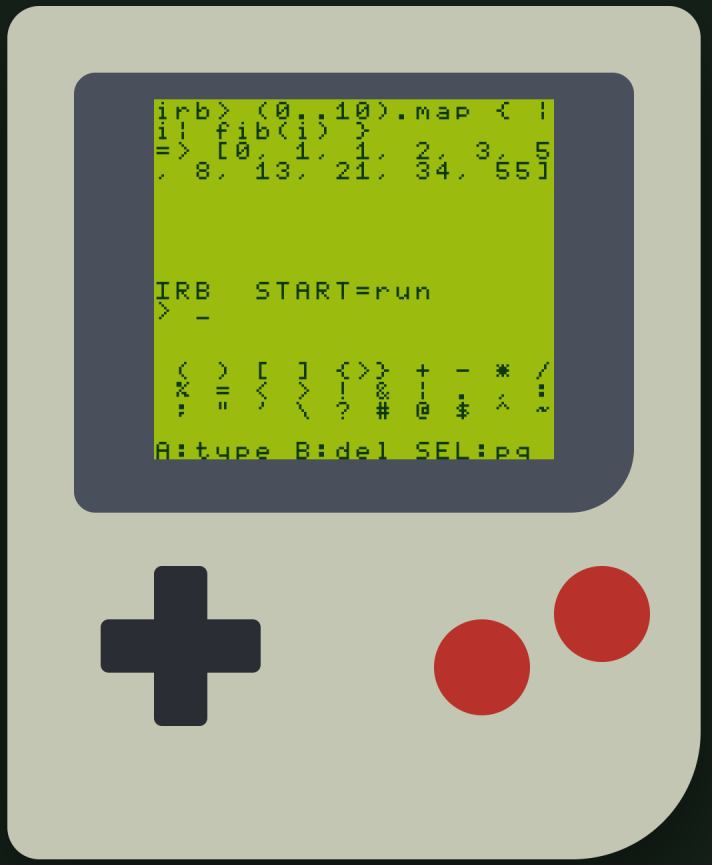
RUBYKAIGI 2027

GBRuby

Ruby on the original Game Boy: 4 MHz, 8 KiB of RAM, no host

[Speaker name] · [@handle]



THE ONE-LINE PITCH

You type Ruby.
The Game Boy compiles it.

On the cartridge

Lexer, parser, compiler, verifier and VM all run on the 8-bit CPU

Persistent IRB

Locals, objects, defs and classes survive between inputs

No host needed

Once the ROM boots, no PC is involved at all

THE MACHINE

Nintendo Game Boy (DMG), 1989

4.19 MHz
Sharp SM83, 8-bit, no FPU, no multiply

8 KiB
Work RAM. All of it. VM, stack, caches

2 MiB
ROM: 128 banks × 16 KiB through an MBC5

128 KiB
Battery SRAM: 16 banks × 8 KiB. The heap lives here

70,224
CPU cycles per LCD frame (59.73 fps)

16 KiB
Visible ROM window at a time. Everything else is a bank switch away

WHAT IT DOES

A real Ruby subset, a real IRB

Language

Classes, modules, singleton methods, big Integers, eval and Binding, Regexp with MatchData, Enumerable and external enumerators, Struct, closures, exceptions, pattern matching, kwargs, &., backtraces with line numbers

76 demos + tutorial

Stored as Ruby source in ROM, compiled on the cart when you pick one; a 16-lesson tutorial and help for 530 methods

Games

Life, Tetris and Snake: game loop in Ruby, board math in native kernels

Four keyboards

On-screen, browser (WASM emulator), terminal, and a link-cable keyboard

Hardware API

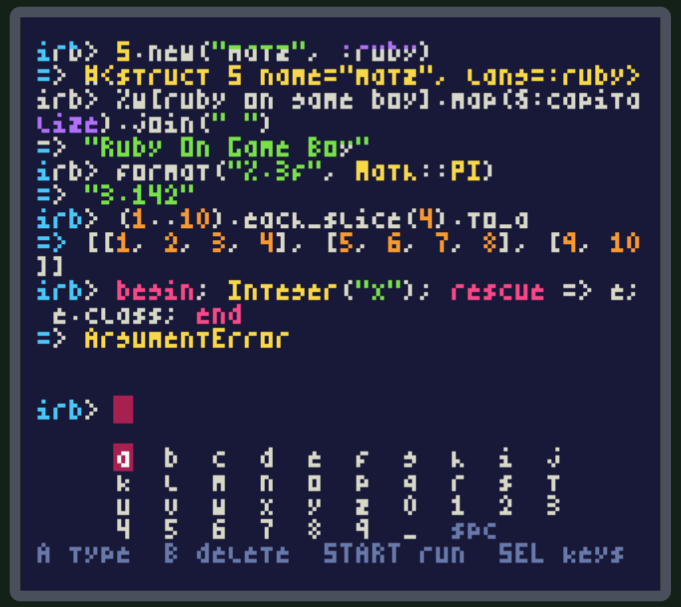
BG scrolling, Window, Tile.define, 40 sprites, Music in MML, pressed?, wait_vbl, beep

Color + dense fonts

Syntax colors on a Game Boy Color; a Super Game Boy palette; a 3×5 font gives 38×22 text on a 160×144 screen

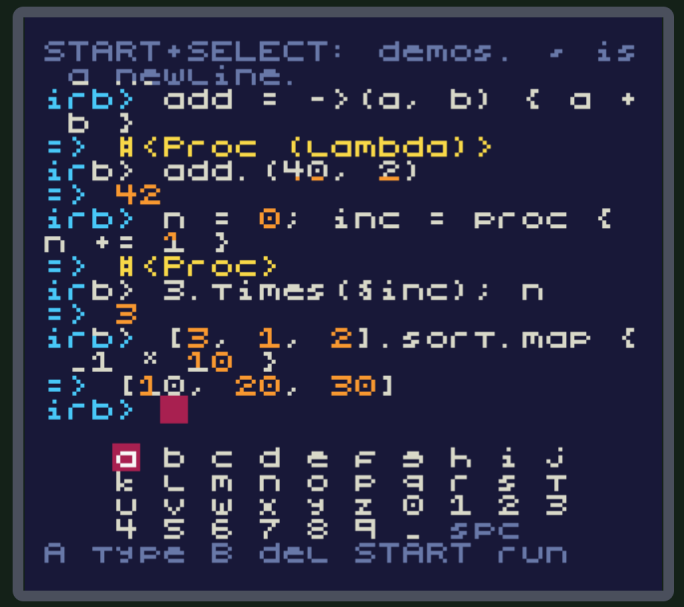
LIVE IRB, ON THE HANDHELD

Three fonts, syntax colors, one 160×144 screen



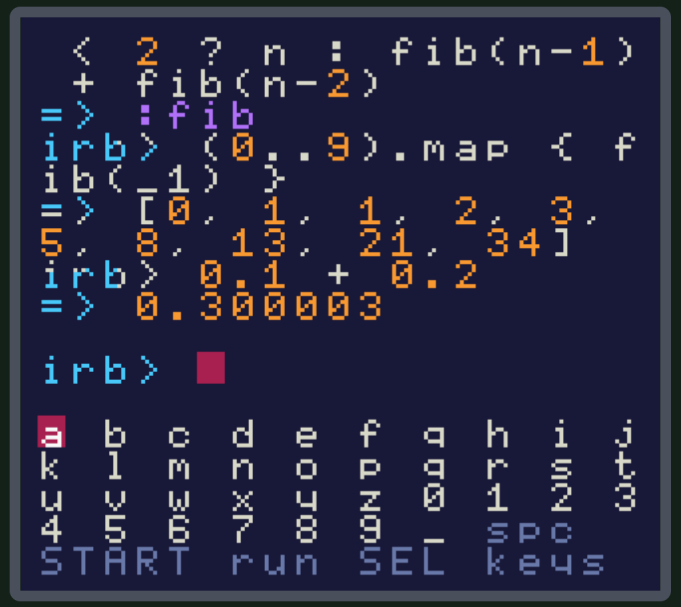
3×5 · 38×22 · default

Struct, format, rescue



5×5 · 25×22

lambdas and closures

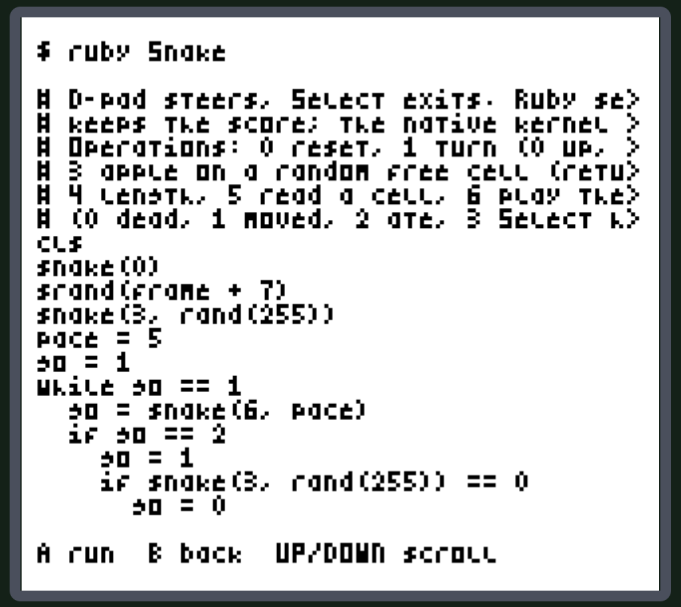


8×8 · 19×17

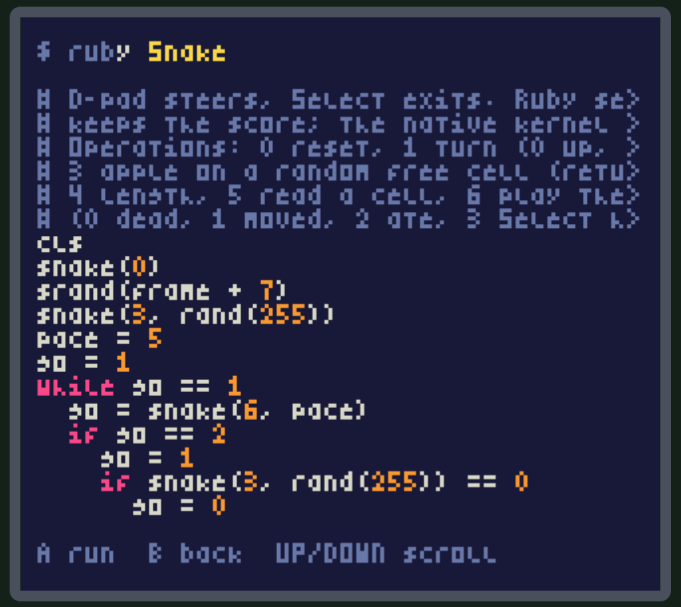
endless def, _1, floats

ONE CARTRIDGE, TWO MACHINES

Game Boy Color, and still a Game Boy



DMG · 1989



CGB · 1998

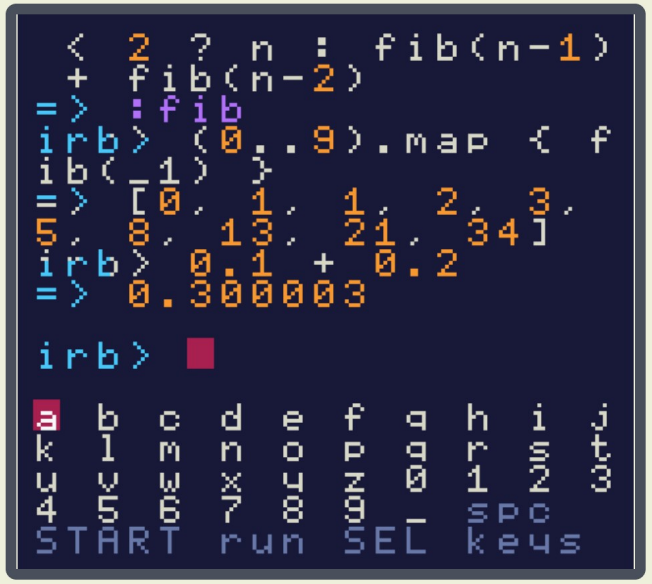


0x143 = 0x80

```
bg_palette(4, 3, 31, 25, 0) · tile_palette(65, 4) · gbc? # ⇒ true
```

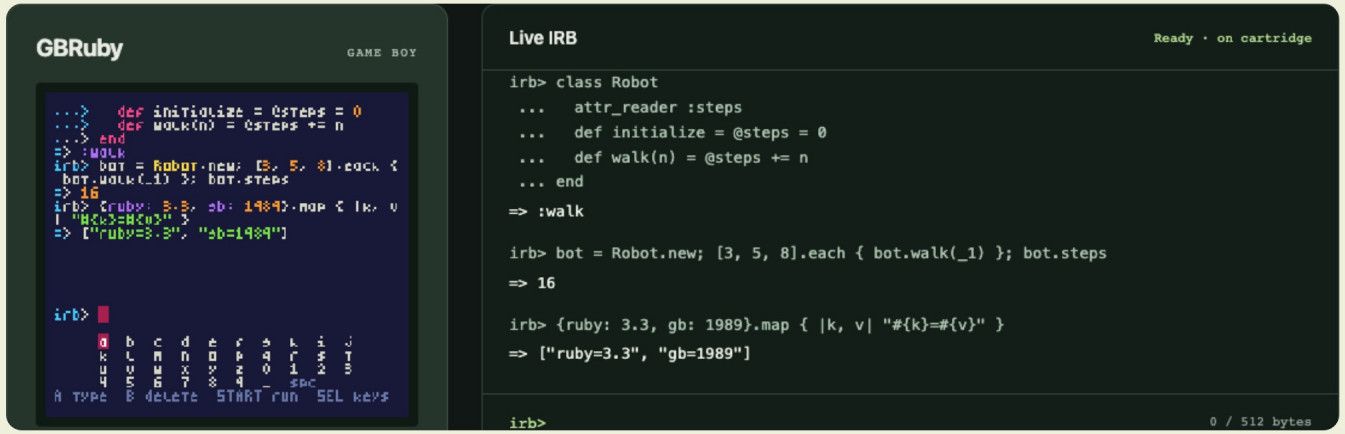
TWO WAYS IN

Type on the D-pad, or in a browser



On-screen keyboard

D-pad picks, A types, SELECT flips the page, START runs



Browser, same ROM

Source goes into a WRAM mailbox at 0xD600; the cart compiles it

HOW IT WORKS

Six stages, all on the cartridge

1
Source
Bytes from ROM, SRAM, WRAM or the keyboard

→

2
Lexer
Streaming, two tokens of lookahead. Names become 16-bit hashes

→

3
Parser
Recursive descent plus Pratt. No AST: it emits as it goes

→

4
Emitter
Append-only bytes with backpatching, in a 6 KiB buffer

→

5
Verifier
One linear pass: lengths, branch targets, a shadow stack

→

6
Dispatcher
Bytecode VM in ROM bank 3: a 150-entry switch

> Single pass: the parser calls emit_* the moment it recognises a fragment. There is no tree to hold in 8 KiB.

> Verify once, run fast: the VM’s trusted loop skips the bounds checks the verifier already proved.

VALUES

Every Ruby value is 4 bytes




```
typedef struct {
  uint8_t tt, flags;
  union { int16_t i; uint16_t sym_id;
          uint16_t rom_idx, ref; } u;
} gbr_value;
```

nil is **all zero bytes**, so clearing 24 registers is a single memset.

tt	meaning
0–2	nil, false, true
3	Integer: int16; overflow becomes a heap big Integer
4	Symbol: 16-bit id
5	String literal: index into ROM pool
8	Float: 24-bit, fraction spills into flags
16+	Heap: String, Array, Hash, Proc, Range, Object, Exception

MEMORY MAP

Three kinds of memory, three jobs

WRAM · 8 KiB

Hot and tiny

- The VM: 556 B (64-slot stack, a 16-frame window, 24 registers)
- Page cache: 4 × 256 B of bytecode
- Console mirror, 20 × 18
- CPU stack: 1,504 B

SRAM · 128 KiB

The heap, and it persists

- Banks 1, 2, 4: the 24 KB Ruby heap
- Banks 3, 15: 16 KB of compiled bytecode
- Bank 13: call frames past the 16th, up to 255 deep
- Banks 10, 14: functions, classes and methods

ROM · 2 MiB

Code, in 16 KiB slices

- Bank 3: the dispatcher
- Banks 2, 4, 6, 7, 8, 31: compiler
- Bank 29: stdlib as bytecode
- Banks 26–27: demo sources

ROM BANKING

The limit is 16 KiB per bank, not 2 MiB

~827 KB

of the 2 MiB ROM is used

52 / 128

banks in use, each packed to within 1 KB by the build

The dispatcher must fit in bank 3 (audit cap 14,800 B). Hitting the cap is how features get evicted:

Case-arm trampoline

Case stays in bank 3's switch; its body calls bank 30

Range routing

One range test sends opcodes 122–138 to banks 30 and 15

Out-migration

Move whole handlers (the regex NFA) to emptier banks

THE HEAP

Refcounting with a one-byte counter



The counter is a **saturating uint8**.

At 254 references an object becomes **immortal**: it can never be freed. 255 means pinned.

No cycle collector. Size-class free lists with boundary tags in each of three SRAM banks. Every run starts from an empty 24 KB heap.

Object	Costs	Cap
String	5 B + its bytes	255 bytes
Array	items block, grows	127 values
Hash	pairs block, grows	127 pairs
Instance	4 B per ivar its class uses	32 ivars
Proc	its captured scope	—

STEP BY STEP

```
x = 1 + 2
puts x
```

17 bytes of Ruby → 9 bytes of bytecode → a “3” on the LCD

LEX PARSE EMIT VERIFY DISPATCH SM83

The lexer turns bytes into tokens

```
IDENT x · 0x0078 | = | INT 1 | + | INT 2 | NEWLINE | IDENT puts · 0x7C26
IDENT x · 0x0078 | NEWLINE | EOF
```

Names are never stored

Each identifier becomes a 16-bit hash:

```
h = (h << 4) ^ ch ^ (h >> 3)
```

Streaming, no token array

Two tokens of lookahead (current and next), read straight from ROM, SRAM or the keyboard buffer.

LEX | PARSE | EMIT | VERIFY | DISPATCH | SM83

The parser compiles while it reads

Tokens seen	What the parser decides	Emits
x =	New local: x lives in slot 0	—
1 + 2	Pratt loop: both sides are literals, fold to 3 (fits int8)	PUSH_INT_3
NEWLINE	Store into slot 0; the statement's value is unused	SET_LOCAL 0, POP
puts x	puts is builtin 0 (from builtin_spec.yml); x is slot 0	GET_LOCAL_0
NEWLINE	One argument, result discarded: use the void form	CALL_BUILTIN_VOID 0, 1
EOF	End of program	HALT

No AST. What survives between statements lives in small side tables: locals, literal pool, functions.

LEX | PARSE | EMIT | VERIFY | DISPATCH | SM83

Nine bytes of bytecode



1-byte opcode (solid) plus 0–4 operand bytes (outlined), little-endian. 150 opcodes in total.

The constant 3 lives inside the opcode: PUSH_INT_0 to PUSH_INT_7 cost one byte and no operand fetch.

LEX PARSE EMIT **VERIFY** DISPATCH SM83

Prove it once, then trust it

pc	instruction	stack depth
0	PUSH_INT_3	0 → 1
1	SET_LOCAL 0	1 → 1
3	POP	1 → 0
4	GET_LOCAL_0	0 → 1
5	CALL_BUILTIN_VOID	1 → 0
8	HALT	0 · max depth 1

One linear pass with a **shadow stack**. It checks:

- every instruction length
- branch targets land on an instruction start
- the stack never underflows or overflows
- slot, argc and pool indexes are in range

Push and pop counts come from opcode_spec.yml, turned into C by a Ruby script at build time.

LEX PARSE EMIT VERIFY **DISPATCH** SM83

The dispatch loop, in bank 3

```
for (;;) {
  /* poll cancel every 32 ops */
  opcode_budget--;
  op = page_cache_fast_byte_at(pc);
  if (op ⩾ 122 && op ⩽ 138) {
    /* floats, regex, blocks */
    banked_handler(op); continue;
  }
  if (op < OP_INT_ADD) goto low_fastpath;
  switch (op) { /* 150 cases */ }
}
```

op	path
08	fast path
13 00	switch
03	switch
14	fast path
2D 00 01	switch, then a banked call to bank 10
00	finish

Bytecode is read through a 4 × 256 B page cache in WRAM. Moving its reader into fixed bank 0 made a VM loop **33% faster** (321 → 214 frames).

LEX PARSE EMIT VERIFY DISPATCH SM83

What the CPU actually runs

```
; switch (op), as compiled by SDCC
ld  e, c        ; e = op
ld  d, #0x00
ld  hl, #table
add hl, de      ; 2 bytes
add hl, de      ; per entry
ld  e, (hl)
inc hl
ld  h, (hl)
ld  l, e        ; hl = table[op]
jp  (hl)
table: .dw case_0, case_1, …
```

~2,900

CPU cycles per bytecode op: about 24 ops per LCD frame

9

instructions just for opcode_budget--, because SDCC keeps locals on the stack

SDCC has no computed goto (&&label), so the switch's jump table is the threaded-code substitute.

SPECIALISED OPCODES

Same program, a third smaller, and not shipped

RELEASE CART · 9 B

```
08 13 00 03 14 2D 00 01 00
```

```
PUSH_INT_3
SET_LOCAL 0
POP
GET_LOCAL_0
CALL_BUILTIN_VOID puts, 1
HALT
```

FULL PROFILE · 6 B

```
08 45 03 54 00 00
```

```
PUSH_INT_3
STORE_LOCAL_0
POP
PUTS_INT_LOCAL 0
HALT
```

The fused handlers exist and pass the tests. They are switched off on the cartridge because bank 3 has no room left for them.

BONUS TRACE

A block that never becomes an object

```
3.times { |i|
  puts i * 2
}
```

compiles as if you had written

```
i = 0
while i < 3
  puts i * 2
  i += 1
end
```

pc	bytes	meaning
0	05 13 00 03	i = 0
4	30 00 03 0B 00	unless i < 3, jump +11 to HALT (one fused op)
9	14 07 1E	i * 2
12	2D 00 01	puts
15	95 00	INC_LOCAL_VOID: i += 1
17	2E F0 FF	JMP −16, back to pc 4
20	00	HALT

CURIOSITY #1

0.1 + 0.2
⇒ 0.300003

Floats are a 24-bit software format packed into the 4-byte value: 1 sign bit, 7-bit exponent, 16-bit fraction. About 5 decimal digits, up to 1.8e19.

FROM POWER-ON TO PLAY

The demos ship as source, not bytecode



1 BOOT

Splash, then the demo list



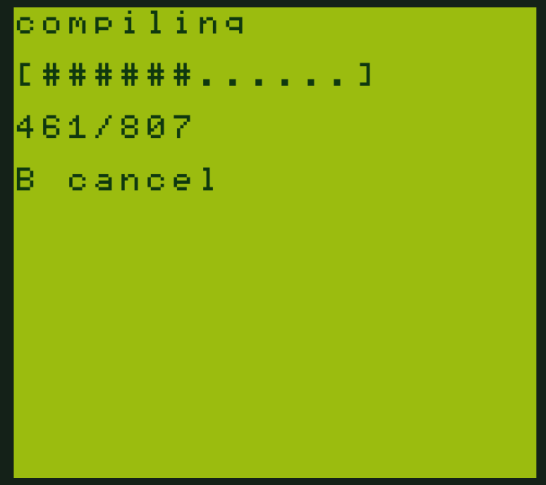
2 PICK

72 demos; SELECT opens IRB



3 READ

The Ruby source, from ROM



4 COMPILE

807 bytes of Ruby, on the cart

GAMES

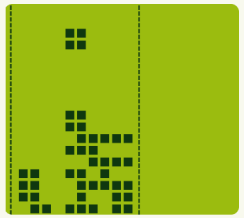
Ruby runs the game, C runs the board



21.6×

Life

3 generations: 6,924 → 320 frames. One bit per cell, a row's neighbours counted at once



5.8×

Tetris

First piece: 2,160 → 372 frames. The well is eighteen 10-bit rows



9.2×

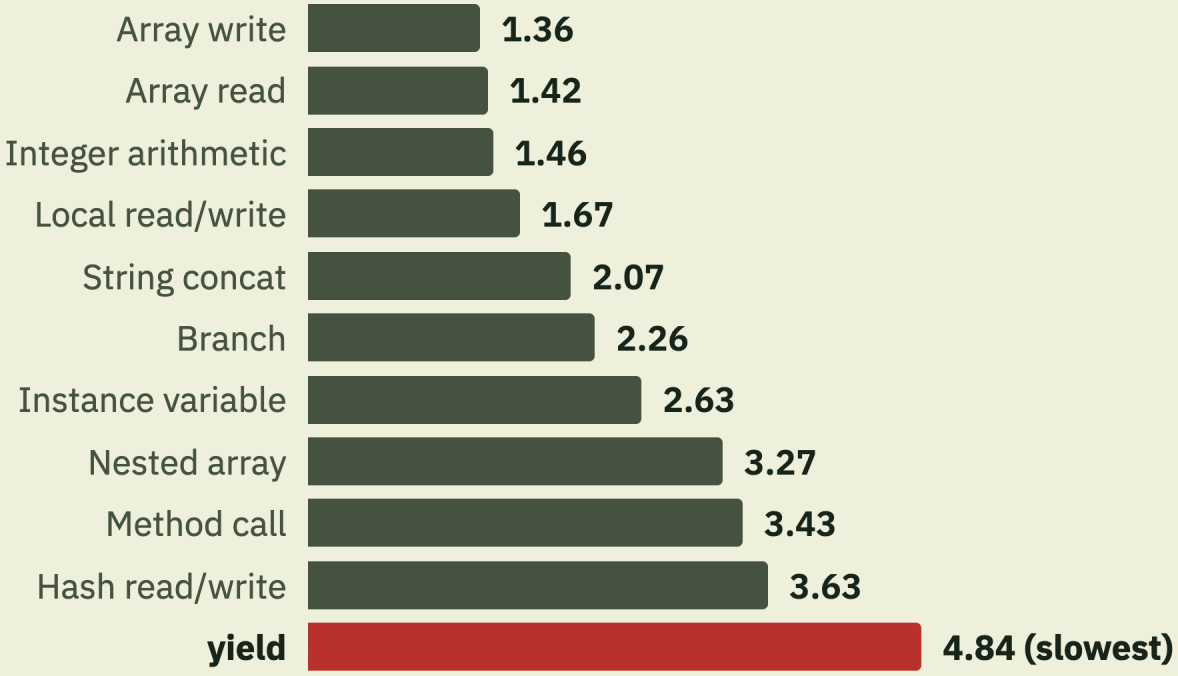
Snake

One step: 55 → 6 frames. Each cell stores a 2-bit exit arrow, so the tail just follows the arrows

Why: one Ruby loop iteration with a four-way pressed? check costs about 4 LCD frames. Input, randomness and score stay in Ruby; all three boards fit in 267 B of WRAM.

PERFORMANCE

LCD frames per operation



The heap is not the problem.

Arrays in banked SRAM are among the cheapest operations. Call frames, yield and linear hash lookup are the hogs.

The model predicted the opposite; measuring settled it.

Measured on the all-C VM. The SM83 fast tier on the next slide cuts most rows 5–7×.

Blocks on values the compiler cannot type: 0.63 frames per element in C, 0.22 once the iterator step moved into the asm tier — faster than an inline loop (0.39).

300-iteration loops, cost per op after subtracting the empty loop.

HYBRID VM

Hand-written SM83 for the hot path: 5–7×

Frames per 1,000 loops	C VM	Hybrid	Speed-up
while loop	86	16	5.4×
Integer arithmetic	458	79	5.8×
Local reads/writes	377	53	7.1×
Array read	655	98	6.7×
Hash	1,461	277	5.3×
Method call	1,388	211	6.6×
Object method + ivar	1,441	218	6.6×
yield	2,128	341	6.2×
each on an untyped Array (per 1,000 elements)	815	222	3.7×
String concat	1,452	825	1.8×

C decides, assembly does.

C admits every opcode first. The tier finishes it, or exits with no side effects and C runs it again.

4,767 lines of SM83: integers, locals, cells, indexing, calls, yield, iterators, ivars, refcounts.

String concat gains least: heap allocation still lives in C.

Cell locals (binding mode) joined the tier last: 454 → 75 frames per 1,000 loops.

SPINEL'S BENCHMARKS

28 programs from matz/spinel, run on a Game Boy

28 / 28

print exactly what Ruby 3.3 prints, at a scale the cartridge holds

Ruby 3.3 on a laptop: **0.02–11 ms**.
The Game Boy: **4.5 seconds to 15 minutes**.

pidigits 255 digits in 15.5 min · bigint_fib fib(20000), 4,180 digits, in 11.4 min

Same programs, first run → today

benchmark	first	today
nqueens	52.7 s	13.8 s
so_lists	48.2 s	16.3 s
csv_process	49.9 s	18.3 s
mandelbrot	73.0 s	30.7 s
io_wordcount	53.4 s	25.0 s
template	56.9 s	29.7 s
partial_sums	26.4 s	19.0 s

WAR STORIES

When one byte breaks six tests

One byte of lexer state

Adding one byte to remember the previous token shifted the WRAM layout and broke 6 cart tests. Rolled back.

Two locals, 11 demos

Two more local variables grew bank 6 by 653 B and broke 11 unrelated demos.

HELLO → ELLO

The CPU stack grew into the heap and ate a letter. The fix: shrink the class tables from 16 to 9.

A static that moved Snake

One static uint8_t in bank 30 changed Snake’s timing. New rule: no BSS in bank-30 code.

The unsolved one

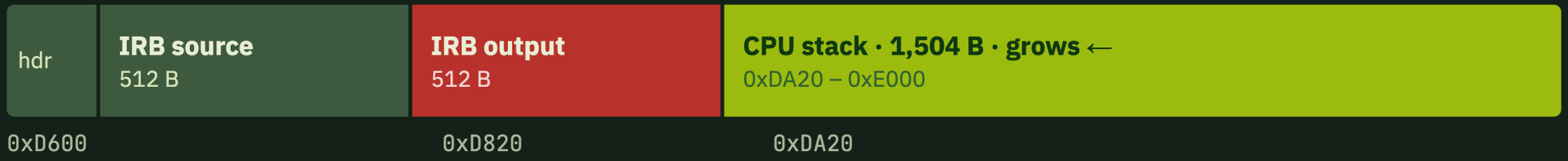
Moving an 80 B helper out of bank 3 made a blocks demo panic with BAD_OPCODE. Root cause never found.

19 green, ROM crashes

19 specs passed on the host and crashed on the cartridge. Now every spec runs against the exact cart flags too.

WAR STORIES

The overflow that printed the right answer



208 B per paren

The recursive-descent parser spent ~208 B of stack per nesting level. About six levels ran past 0xDA20.

Silent

The answer came out right while the IRB output buffer below was being overwritten. Found by painting the stack with 0xA5.

Smaller frames + a guard

Cold branches split out of the recursive path (64 → 38 B), and a clean error 0x310D before the stack runs out. 23 nested parens now fit. The buffers can’t move: clients rely on them.

MISCELLANY

Small things I’m weirdly proud of

A keyboard that never shipped

The link port speaks WorkBoy, an unreleased Bandai keyboard. Escape can interrupt a tight loop.

Time without a clock

No RTC on the cart. Time.now is a 16 Hz timer that starts at Time.set.

Multiply by lookup

256-entry ROM tables make an integer multiply ~30 cycles instead of ~200.

A VM inside the VM

73 stdlib methods ship as bytecode in bank 29 and run as a sub-VM with a 1,024-op budget.

Interpolation, unbuilt

puts "x=#{a}" never builds a string: it prints each piece, then a newline.

Ruby builds the Ruby

Ruby 3.4 scripts turn YAML specs into the opcode, builtin and keyword tables.

TESTING

How you test a Game Boy from a laptop

1,216
golden Ruby specs, each also checked for heap leaks

4,447
one-line probes compared with Ruby 3.3 on the ROM

80
cart test runs: 40 scripts on a DMG and on a Game Boy Color

3
CI hosts that build the ROM byte-identically

1,296
programs, asm tier vs all-C: same output, same heap

> Each spec stores four numbers: compile error, verify error, run error and a checksum of the screen.

> The ROM reports results through a WRAM mailbox at 0xC0A0, magic "MRBC".

HOW MUCH RUBY?

4,593 one-liners, checked against Ruby 3.3

first run

158 / 489

→

today

4447 / 4447

Wrong answers fixed

-5.abs, s << "b", {a: 1} == {a: 1}, "hello" =~ /ll/, flatten

Syntax

a, b = b, a · ||= · &:sym · _1 · kwargs · case/in · heredocs

Objects

singleton methods, eval and Binding, big Integers, MatchData, next and peek

No misses left: `p 1_000_000`, `def o.hi` and `eval("x + 1", b)` all print what Ruby prints.

HONEST LIMITS

What it is not (yet)

Missing on the cart

- Fibers, threads, lazy enumerators
- Refinements, File and IO, Rational
- Ruby's Enumerable through a redefined built-in each
- IEEE double precision

Hard caps

- 16 locals per scope, 255 nested calls
- Strings 255 B, arrays 127, hashes 127
- Big Integers fit in one 8 KB bank
- Floats about 5 digits
- 24 KB heap, 8,000-byte programs

Not on real hardware

Tagged v1.0-emulator: it has only ever run in emulators. A flash cart is the next step.

888 commits · 81k lines of C · 4.8k lines of SM83 · 72 demos

```
irb> puts "Thank you!"
Thank you!
⇒ nil
```

ありがとうございました

[repo URL] · [@handle] · Play it in your browser: [demo URL]