

phases of compiler design

The classical six-phase model, measured against a compiler that reports its own phase structure as data.

ZERO 0.3.4

build 5b3a90a

CAPTURED 2026-08-21

WHY THIS PAPER

does the textbook still describe a real compiler?

Every syllabus teaches six phases in a straight line, front to back. We took one modern compiler, ran it, and asked it to describe itself.

6

phases the textbook names

2

cross-cutting activities beside them

8

phases Zero 0.3.4 actually reports

The two lists don't line up — and where they diverge is the finding, not an inconvenience.

Zero's own phase list · `zero check --json` → `compilerPhases`

BACKGROUND

the model we were taught

Six phases in sequence, with a symbol table and an error handler running alongside all of them.

● PHASE 1

Lexical analysis

● PHASE 2

Syntax analysis

● PHASE 3

Semantic analysis

● PHASE 4

Intermediate code generation

● PHASE 5

Code optimization

● PHASE 6

Target code generation

CROSS-CUTTING

Symbol table management — a structure the front end rebuilds every compile.

CROSS-CUTTING

Error detection and reporting — diagnostics rendered as prose, for a person.

THE SUBJECT

zero compiles a graph, not a file

Text enters once, through an ingestion gate, and becomes a persistent typed graph. Every compile after that reads stored facts instead of re-deriving them.

108

declarations

109

symbols

176

types

974

source-map rows

That's the symbol table — not rebuilt per compile, but stored, versioned and queryable. It's the largest program in our corpus, and the graph is the artifact.

8 TARGETS

DIRECT EMITTERS

NO C BRIDGE

--json on every command

zero query --json --full on *p08_lexer*

METHODOLOGY

we ran the compiler and wrote down what it said

One harness drives every command, on every program, for every target — and writes one JSON file.

8

corpus programs, 6 to 304
lines

10

deliberately broken
programs

8

advertised build targets

64

builds attempted

Per program: tokens · parse · check · query · source-map · time · size
· mem · test · run, plus one build per target. Prose, tables and charts all
read that same file, so the text and the evidence can't drift apart.

THE CORPUS

eight programs, smallest to largest

PROGRAM	LINES	TOKENS	GRAPH NODES	RELATIVE SIZE	BUILDS
p01_hello	6	35	13		8/8
p02_arith	31	181	125		8/8
p03_control	54	237	179		8/8
p04_shapes	85	371	231		5/8
p05_errors	54	228	150		3/8
p06_memory	101	665	413		5/8
p07_generics	45	223	131		5/8
p08_lexer	304	1,471	974		5/8

zero tokens --json · zero query --json · 680 lines, 2,216 nodes, 2,201 edges in total

FINDING 1

the phases didn't disappear. they moved.

Lexical, syntactic and semantic analysis run once, at an ingestion gate that admits a program into the graph. The steady-state compile path is lowering, codegen and link — nothing else.

RUNS ONCE · AT THE GATE

RESOLVE

PARSE

INTERFACE

CHECK

Characters are scanned here and never again. Package compilation reads a binary graph store, so the scanner is absent from the steady-state path.

RUNS EVERY COMPILE

LOWER

CODEGEN

OBJECT

LINK

Note that resolve is reported *before* parse. That isn't our typo — it's what a compiler does when the names are already bound.

`zero check --json → compilerPhases, in the order reported`

FINDING 2

lowering is the whole clock

Every reported phase except lowering returns 0 ms. The front end isn't fast — it isn't running.



100%

of measurable phase time is
lowering

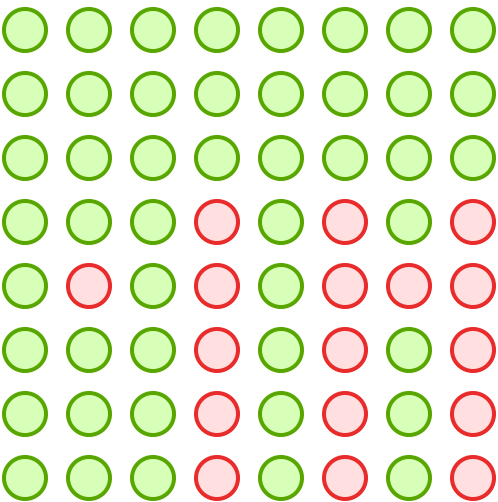
53 ms of 53 ms, summed
over all 8 programs.

zero time --json, cold cache · summed across the corpus · 0 ms is the compiler's own reported figure, not a rounding artifact of ours

FINDING 3

the two ends accept different languages

17 of 64 builds failed with BLD004 — after zero check had already passed them.



One dot per build. 8 programs × 8 targets.

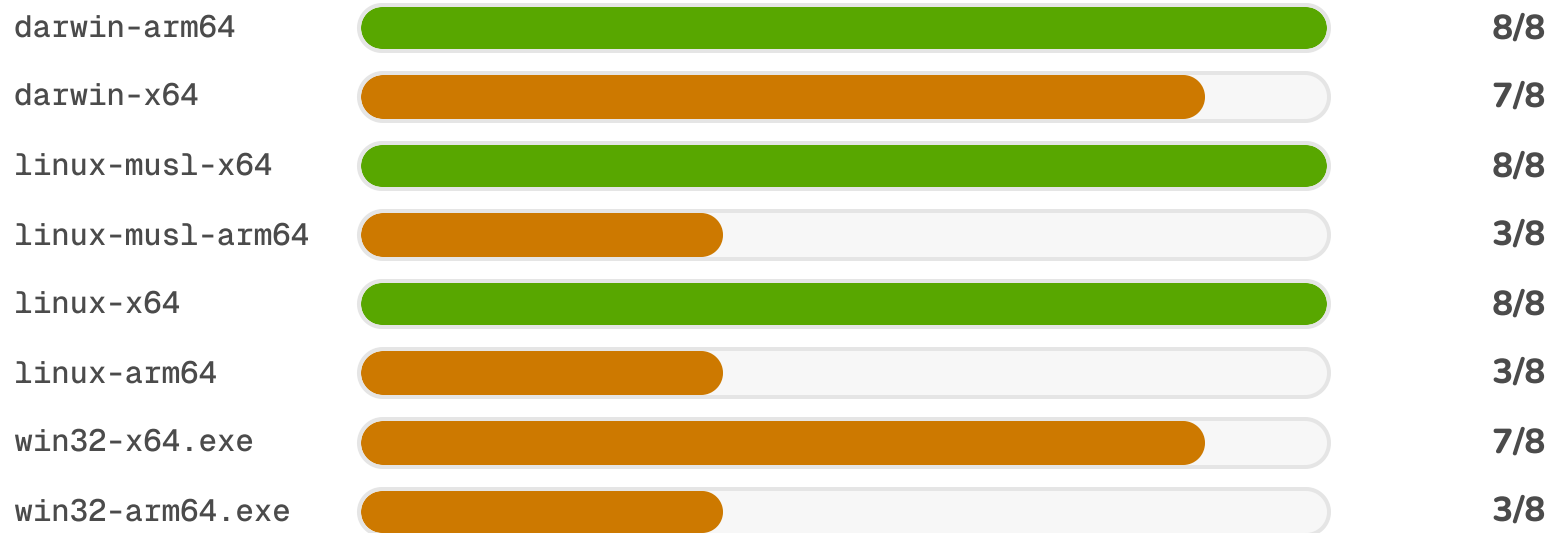
47 / 64 builds produced a binary		17 refused after passing the check	
WHY THE BACKEND REFUSED		BUILDS	
BLD004 · record		●	9
BLD004 · IR_VALUE_CHECK		●	4
BLD004 · unsupported instruction		●	3
BLD004 · IR_VALUE_RESCUE		●	1

zero build --target <t> after a clean zero check · *match* could not be lowered on any target on this build

FINDING 3, CONTINUED

same graph, different emitter

Backend completeness belongs to the target, not the program.
Four targets built everything; three built fewer than half.



ONE PROGRAM, TWO FATES

p05_errors cross-compile to an **16,632-byte** binary for Linux — and is refused outright on the Windows host it was compiled on.

3 BUILT

5 REFUSED

```
zero build --target <t> --emit exe · zero size --json
```

FINDING 4

one command, two verdicts that disagree

zero check --json publishes a top-level verdict and a nested one. On some programs they contradict each other, and the schema presents the wrong one first.

WHAT THE TOP LEVEL SAYS

```
"ok": true,  
"diagnostics": []
```

A consumer reading the field the schema presents as *the* verdict stops here.

WHAT THE SAME PAYLOAD ALSO SAYS

```
"targetReadiness": {  
  "buildable": false,  
  "stage": "lower"  
}
```

...carrying a **BLD004** that names the exact construct it can't lower.

2 of our 10 error cases hit this. The compiler knew. It just answered the question twice.

FINDING 5

what it costs to be readable by a program

The same question — is this program correct? — answered for a person and for a machine, on a six-line program.

141 bytes
the whole program

4 bytes
zero check — the prose answer

17,896 bytes
zero check --json — the same answer

That's **4,474×** more bytes to say the same thing — and on our largest program the JSON reaches 189,741 bytes. The trade isn't tokens for tokens. It's tokens for *actionability*: a span, a typed repair, and a field a program can branch on without parsing English.

ten broken programs, three gates

CASE	INTENDED PHASE	REFUSED AT	CODE
e01_lexical	lexical	IMPORT	PAR100
e02_syntax	syntax	IMPORT	PAR100
e03_name	resolve	IMPORT	NAM003
e04_type	check	IMPORT	TYP002
e05_mutability	check	IMPORT	TYP009
e06_effect	check	IMPORT	ERR003
e07_memory	check	IMPORT	MEM003
e08_target	target	CHECK	TAR002
e09_lowering	lower	BUILD	BLD004
e10_match	—	BUILD	BLD004

7

stopped at the gate

1

stopped at check

2

got as far as build

0 of 10 left anything in the store. The gate holds.

zero import → zero check → zero build, in that order

THE ANALYTICAL CORE

six phases, mapped onto what zero reports

CLASSICAL PHASE	ZERO REPORTS IT AS	WHERE IT RUNS	HOW TO SEE IT
1. Lexical analysis	parse	INGESTION	zero tokens --json
2. Syntax analysis	parse	INGESTION	zero parse --json
3. Semantic analysis	resolve interface check	BOTH	zero check --json
4. Intermediate code generation	lower	COMPILE-PATH	zero size --json (loweredIrBytes)
5. Code optimization	lower codegen	COMPILE-PATH	zero build --profile release-small tiny
6. Target code generation	codegen object link	COMPILE-PATH	zero build --emit exe && zero size --json

Optimization is the one phase Zero never reports separately – it’s selected by build profile, so its cost is folded into *lower* and *codegen*.

phases 1 and 2 run in your browser

Zero 0.3.4 has no WebAssembly target — its own report marks `browserCompiler` as removed — so the real compiler can't run client-side. We reimplemented the scanner in TypeScript instead, then checked it against the compiler token by token.

2,666

tokens compared, field by field

8/8

programs matching exactly

8/8

function sets recovered

Kind, text, line, column, offset and length — all six fields, on every token. So phases 1–2 run live on whatever you type. Phases 3–6 are replayed from the capture, and the terminal says so out loud the moment you edit the buffer.

WHY IT MATTERS

the diagnostics are written for a program

The textbook's second cross-cutting activity is error reporting, and it assumes a human reader. Zero's assumes a caller that has to decide what to do next.

CLASSICAL

A sentence. To act on it, you parse English and hope the wording is stable across releases.

ZERO

code

span

expected

actual

fixSafety

repair

Every field is branchable. That's what the extra 17,892 bytes buy.

The six phases were designed when the compiler's reader was a person at a terminal. Zero is designed for a compiler whose reader is another program.

`zero explain --json · zero fix --plan --json`

what this evidence can't tell you

Everything here describes one build of one compiler on one machine. We'd rather say that than let a reader over-read the numbers.

ONE VERSION

Zero 0.3.4, build 5b3a90a, captured 2026-08-21. It's a preview-stage compiler; the BLD004 gaps are unfinished work, not a design position.

ONE MACHINE

13th Gen Intel(R) Core(TM) i5-13450HX, 16 cores. Timings are medians of repeated runs, but they're still this host's timings, and cross-compilation reports itself as not ready.

A SMALL CORPUS

8 programs, 680 lines. Enough to show the shape, far too little to characterise a language.

MILLISECOND RESOLUTION

0 ms means “under the reporting floor”, not “free”. The claim is that the work moved, not that it vanished.

CONCLUSION

a vocabulary, not an order

The six phases still name the right work. They no longer describe the sequence a compiler runs them in.

WHAT SURVIVES

Lexing, parsing, typing, lowering and emission are all still there, still separable, still worth naming separately.

WHAT CHANGED

They're split across an ingestion gate and a compile path. The symbol table became a database. Diagnostics became records.

WHAT THAT COSTS

Two ends that accept different languages, 17 refused builds, and a verdict field that can be wrong.

Teach the six phases as a decomposition of the *work*. Then show a compiler that reports its own — because that's where the model earns its keep, and where it stops.

THANK YOU

read the whole thing

The paper, the dataset, the corpus and a working
playground — zero.khe.money

PDF · A4

EPUB · REFLOW

8 PROGRAMS, REPRODUCIBLE