Go, lowered to machine code

marmot is a Go compiler built on MLIR and LLVM. Keep your Go code and your go.mod, switch the compiler, and get small, fast native binaries and errors that show you the fix.

Get started | Read the benchmarks | See the progress

Experimental: marmot compiles real Go programs, including itself, and checks every test program against Go, but it is not ready for production.

Go

Your code, unchanged.

```
var s int32
for _, x := range xs {
    s += x
}
return s
```

MLIR, with the go dialect

Still Go: an index into a slice, with its bounds check.

```
%v9 = llvm.extractvalue %a0[0] : !llvm.struct<(ptr, i64, i64)>
%v10 = llvm.extractvalue %a0[1] : !llvm.struct<(ptr, i64, i64)>
%v11 = go.index %v9[%v7] len %v10, i32 : i64
%v12 = llvm.load %v11 {alignment = 4 : i64, go.unsafe} : !llvm.ptr -> i32
%v13 = llvm.add %v1, %v12 : i32
```

LLVM IR

The check is gone; four int32 values at a time.

```
%wide.load = load <4 x i32>, ptr %5, align 4
%wide.load12 = load <4 x i32>, ptr %6, align 4
%wide.load13 = load <4 x i32>, ptr %7, align 4
%wide.load14 = load <4 x i32>, ptr %8, align 4
%9 = add <4 x i32> %wide.load, %vec.phi
```

arm64 machine code

Two 128 bit loads, four vector adds.

```
ldp q4, q5, [x8, #-0x20]
ldp q6, q7, [x8], #0x40
add.4s v0, v4, v0
add.4s v1, v5, v1
add.4s v2, v6, v2
add.4s v3, v7, v3
```

Against go build

Measured on an Apple M3 Pro, newest measurement of each. Every program's output is checked against Go. Shorter bars are better.

Measurement	marmot	go
hello world with println, -release binary size	**201 KB**	1.83 MB
hello world with fmt.Println, -release binary size	**698 KB**	2.43 MB
run time, 45 programs, -release geometric mean; 37 faster than Go, 8 slower	**1.54x faster**	1x
first build of cmd/marmot, empty caches 325 packages, 2026-11-14	**4.40 to 4.57 s**	6.14 to 6.18 s
rebuild, nothing changed 325 packages, 2026-11-14	**0.05 s**	0.15 s
one line edit, leaf package 325 packages, 2026-11-14	**0.59 s**	0.66 to 0.68 s
one line edit, deep package 325 packages, 2026-11-14	**0.72 to 0.76 s**	1.18 to 1.23 s

Every number and how it was measured

Errors that show the fix

Marmots stand guard and whistle when something is wrong. marmot's errors show the source, mark the exact span, label every place involved, and suggest a fix that `marmot fix` can apply.

Off a terminal they are the plain lines the go tool prints. Under a coding agent they are JSON, with the spans, the source around them and the fixes as edits. Every error has a stable code, and all 44 are explained.

shapes

```
shapes $ marmot build .
error[M0014]: cannot use Square{…} (value of struct type Square) as Shape
              value in variable declaration: Square does not implement Shape
              (method Area has pointer receiver)
  --> ./main.go:12:16
   |
 5 | type Shape interface{ Area() float64 }
   |      ----- `Shape` is declared here
 ...
 9 | func (q *Square) Area() float64 { return q.side * q.side }
   |         ------- `Area` has a pointer receiver
 ...
12 |     var s Shape = Square{2}
   |                   ^^^^^^^^ `Square` does not implement `Shape`
   |
   = note: `Square` lacks 1 method of `Shape`:
             `Area` has a pointer receiver, so only `*Square` has it
help: pass a pointer (check it)
   |
12 -     var s Shape = Square{2}
12 +     var s Shape = &Square{2}
   |

error: build failed with 1 error
for more about an error, run `marmot explain M0014`
```

One command for the whole loop

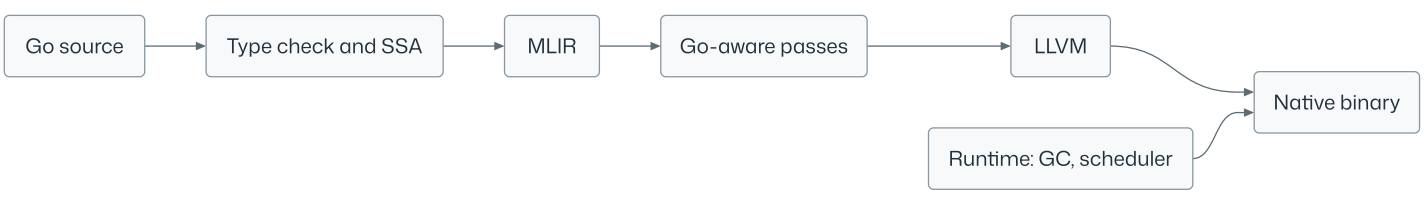
marmot works like the go command, and does what people usually install next to it.

`build`	Compile a program. Dev builds for fast rebuilds, -release for whole program optimization.	`run`	Compile and run a program.
`test`	Run package tests like go test, with each failure shown at its line.	`check`	Run vet, lint, gopls, vuln, tidy, cross builds and more in one cached pass.

`fix`	Apply the suggested fixes of errors and findings, and keep only what still builds.	`fmt`	Format source and fix imports, as gofmt and goimports do.
`watch`	Build, run or test again after every save.	`repl`	Evaluate Go expressions and statements as you type.
`bench`	Run benchmarks against go test and report which changes are significant.	`prof`	Read a pprof profile in the terminal, as a tree or line by line.
`size`	Show what a binary is made of, and why a function is in it.	`generate`	Run go:generate directives, restoring cached outputs.
`explain`	Describe an error code, with the mistake and the fix.	`doctor`	Check the installation and say what to fix.
`mcp`	Serve build, run, test, fix and explain to coding agents.		

Go's front end, MLIR and LLVM behind it

marmot keeps the parts of the Go toolchain that define the language: the type checker, the SSA builder and Go's own standard library. It replaces the parts that generate code. A small go dialect keeps Go's meaning in MLIR long enough to remove bounds checks, keep objects on the stack and call interface methods directly.



How marmot is put together

A marmot is a close relative of the gopher, and it starts with M, like MLIR.

Source on GitHub