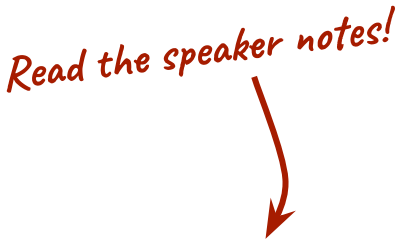
Analysis and transformation tools for Go code modernization

Alan Donovan
Go team, Google

GopherCon ‘25
New York
August 27, 2025



Go 1 and the Future of Go Programs

Table of Contents

Introduction

The release of Go version 1, Go 1 for short, is a major milestone in the development of the language. Go 1 is a stable platform for the growth of programs and projects written in Go.

Go 1 defines two things: first, the specification of the language; and second, the specification of a set of core APIs, the "standard packages" of the Go library. The Go 1 release includes their implementation in the form of two compiler suites (gc and gccgo), and the core libraries themselves.

It is intended that programs written to the Go 1 specification will continue to compile and run correctly, unchanged, over the lifetime of that specification. At some indefinite point, a Go 2 specification may arise, but until that time, Go programs that work today should continue to work even as future "point" releases of Go 1 arise (Go 1.1, Go 1.2, etc.).

Compatibility is at the source level. Binary compatibility for compiled packages is not guaranteed between releases.

Every Go release brings new gifts…

Go 1.18: generics

Go 1.20: strings.Cut{Prefix,Suffix}

Go 1.21: extended forward compatibility; min, max builtins; slices, maps packages

Go 1.22: range over int

Go 1.23: range over func

Go 1.24: iterators such as strings.{SplitSeq,Lines,FieldsSeq}, testing.B.Loop

Example

```
func f(content string) {
	for _, line := range strings.Split(content, "\n") {
		found := false
		for _, field := range strings.Fields(line) {
			if field == "-" {
				found = true
				break
			}
		}
		if found {
			for i := 0; i < 3; i++ {
				fmt.Println(line)
			}
		}
	}
}
```

Example

Ranging over SplitSeq is more efficient

```
func f(content string) {
	for _, line := range strings.Split(content, "\n") {
		found := false
		for _, field := range strings.Fields(line) {
			if field == "-" {
				found = true
				break
			}
		}
		if found {
			for i := 0; i < 3; i++ {
				fmt.Println(line)
			}
		}
	}
}
```

Example

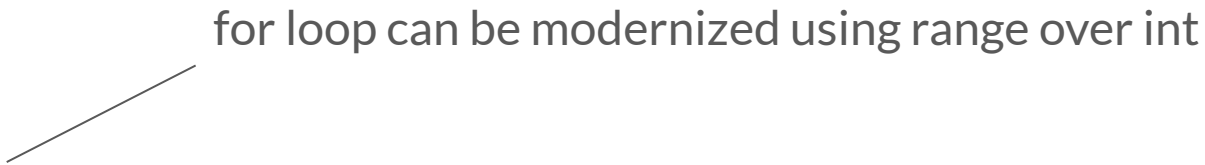
```
func f(content string) {
	for _, line := range strings.SplitSeq(content, "\n") {
		found := false
		for _, field := range strings.Fields(line) {
			if field == "-" {
				found = true
				break
			}
		}
		if found {
			for i := 0; i < 3; i++ {
				fmt.Println(line)
			}
		}
	}
}
```

Loop can be simplified using slices.Contains

Example

```
func f(content string) {
	for _, line := range strings.SplitSeq(content, "\n") {
		found := slices.Contains(strings.Fields(line), "-")

		if found {
			for i := 0; i < 3; i++ {
				fmt.Println(line)
			}
		}
	}
}
```



Example

```
func f(content string) {
	for _, line := range strings.SplitSeq(content, "\n") {
		found := slices.Contains(strings.Fields(line), "-")









		if found {
			for range 3 {
				fmt.Println(line)
			}
		}
	}
}
```

Example

```
func f(content string) {
	for _, line := range strings.SplitSeq(content, "\n") {
		found := slices.Contains(strings.Fields(line), "-")

		if found {
			for range 3 {
				fmt.Println(line)
			}
		}
	}
}
```

`strings.SplitSeq?`

> Write a Go function to **split a string into lines** and report the lengths of the **longest and shortest** ones.

`max`, `min?`

strings.SplitSeq?

> Write a Go function to **split a string into lines** and report the lengths of the **longest and shortest** ones.

max, min?

```
lines := strings.Split(s, "\n")
...
for _, line := range lines[1:] {
       currentLength := len(line)
       if currentLength > longest {
           longest = currentLength
       }
       if currentLength < shortest {
           shortest = currentLength
       }
}
```

not: strings.SplitSeq

not: longest = max(longest, len(line))

not: shortest = min(shortest, len(line))

(Gemini 2.5 Flash)

> Write a Go benchmark that calls a function f() in the loop.

```
... for i := 0; i < b.N; i++ { f() } ...
```

> Use the latest features and idioms of Go.

```
... same answer ...
```

> What about b.Loop?

```
b.Loop is a method that allows for a more flexible and modern way...
```

Was hoping for one of:

```
for range b.N { f() }
for b.Loop()  { f() }
```

(Gemini 2.5 Flash, heavily edited for space.)

> So why didn’t you suggest it?

`Because, while it's a more modern and idiomatic way to write benchmarks, the traditional for i := 0; i < b.N; i++ loop is still the most fundamental and widely understood pattern for Go benchmarks.`

> Could you use a range loop?

`Yes, but it's not the most direct way to solve this problem…`

> What would ‘for range b.N’ do?

`It is a new feature in Go 1.22 … a more concise way of writing a traditional for loop.`

> That seems better, doesn't it?

`Yes … It represents a subtle but significant improvement in the language.` *[bullet list of why it’s better]*

> So why didn’t you offer it then??

(Gemini 2.5 Flash, heavily edited for space.)

> Write a Go benchmark that calls a function f() in the loop.

```
... for i := 0; i < b.N; i++ { f() } ...
```

> Use the latest features and idioms of Go.

```
... same answer ...
```

(Gemini 2.5 Pro. heavily edited for space.)

> What about the for loop up to b.N? Can it be made simpler?

No, that loop is the standard, required structure for all Go benchmarks and cannot be simplified…
The for loop syntax is the convention that the benchmarking tool looks for. *[a lie]*

> What about for range b.N?

No, you cannot use for range b.N. That syntax is not valid.
The range keyword … cannot be used to loop up to an integer value.

> Are you sure?

Yes, I'm sure. Only slices or arrays, strings, maps, or channels. *[outdated information]*

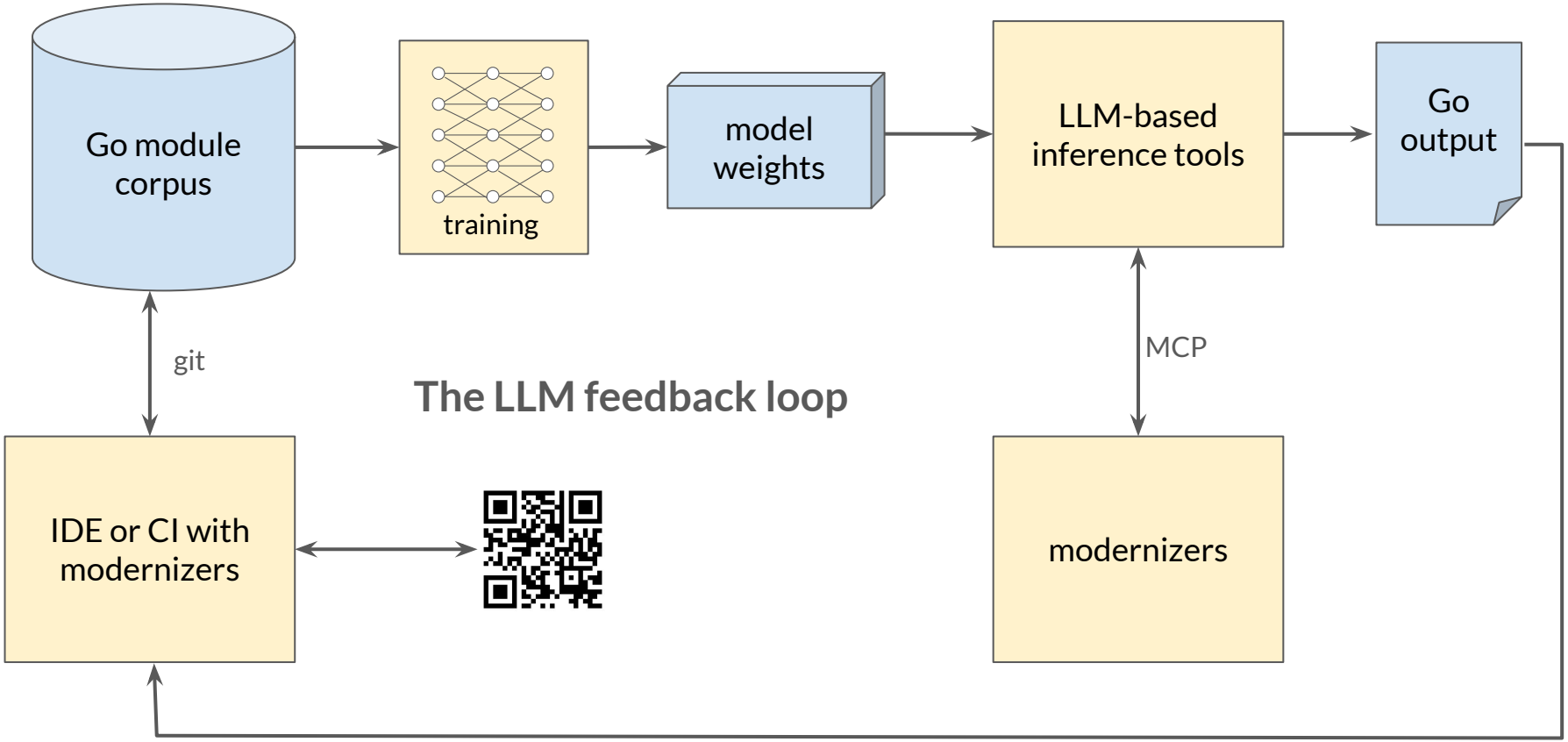
> What about go1.23?

As of Go 1.23, you still cannot. … The proposal to add for range over an integer was also considered for the Go 1.23 release cycle but was ultimately not included. *[another lie]*

> So why do the Go 1.22 release notes say "For loops may now range over integers."?

Excellent question! You are correct that the Go 1.22 release notes mention this, but …
GOEXPERIMENT=rangefunc … *[ever more sophisticated lies]*

(Gemini 2.5 Pro. heavily edited for space.)


Go module corpus
training
model weights
LLM-based inference tools
Go output
git
MCP
The LLM feedback loop
IDE or CI with modernizers
modernizers

How do we implement modernizers?

analysis package

Version: v0.35.0 Latest | Published: Jul 11, 2025 | License: BSD-3-Clause | Imports: 8 | Imported by: 5,680

Details Valid go.mod file | Redistributable license | Tagged version | Stable version | Learn more about best practices

Repository cs.opensource.google/go/x/tools

Links Report a Vulnerability | Open Source Insights

Documentation

Overview

Package analysis defines the interface between a modular static analysis and an analysis driver program.

Background

A static analysis is a function that inspects a package of Go code and reports a set of diagnostics (typically mistakes in the code), and perhaps produces other results as well, such as suggested refactorings or other facts. An analysis that reports mistakes is informally called a "checker". For example, the printf checker reports mistakes in fmt.Printf format strings.

A "modular" analysis is one that inspects one package at a time but can save information from a lower-level package and use it when inspecting a higher-level package, analogous to separate compilation in a toolchain. The printf checker is modular: when it discovers that a function such as log.Fatalf delegates to fmt.Printf, it records this fact, and checks calls to that function too, including calls made from another package.

By implementing a common interface, checkers from a variety of sources can be easily selected, incorporated, and reused in a wide range of driver programs including command-line tools (such as vet), text editors and IDEs, build and test systems (such as go build,

```
var Analyzer = &analysis.Analyzer{
      Name:     "…",
      Doc:      "…",
      Run:      run,
}

func run(pass *analysis.Pass) (any, error) {
      for _, file := range pass.Files {
            … inspect file, report problems …
      }
      return nil, nil
}
```

```
package ioutil // "io/ioutil"

import "os"

// ReadFile reads the file named by filename and returns the contents.
// A successful call returns err == nil, not err == EOF. Because ReadFile
// reads the whole file, it does not treat an EOF from Read as an error
// to be reported.
//
// Deprecated: As of Go 1.16, this function simply calls [os.ReadFile].
func ReadFile(filename string) ([]byte, error) {
      return os.ReadFile(filename)
}
```

```
package readfile; import ( … )

var Analyzer = &analysis.Analyzer{
      Name:     "readfile",
      Doc:      "report uses of deprecated ioutil.ReadFile",
      Run:      run,
}

func run(pass *analysis.Pass) (any, error) {
      for _, file := range pass.Files {
            ast.Inspect(file, func(n ast.Node) bool {
                  if call, ok := n.(*ast.CallExpr); ok &&
                        isFunctionNamed(
                              typeutil.Callee(pass.TypesInfo, call), "io/ioutil", "ReadFile") {
                        pass.ReportRangef(call, "ioutil.ReadFile is deprecated; use os.ReadFile")
                  }
                  return true
            })
      }
      return nil, nil
}
```

(`isFunctionNamed` helper omitted for space)

```
package main

import (
	"example.com/analyzers/readfile"
	"golang.org/x/tools/go/analysis/singlechecker"
)

func main() {
	singlechecker.Main(readfile.Analyzer)
}
```

```
$ go build -o readfile ./main.go

$ ./readfile ./my/...
my/file.go:44:15: ioutil.ReadFile is deprecated; use os.ReadFile

$ ./readfile -c=1 ./my/...
my/file.go:44:15: ioutil.ReadFile is deprecated; use os.ReadFile
43    func load(filename string) (any, error) {
44          data, err := ioutil.ReadFile(filename)
45          if err != nil {
```

-c=1: “show one line of context”

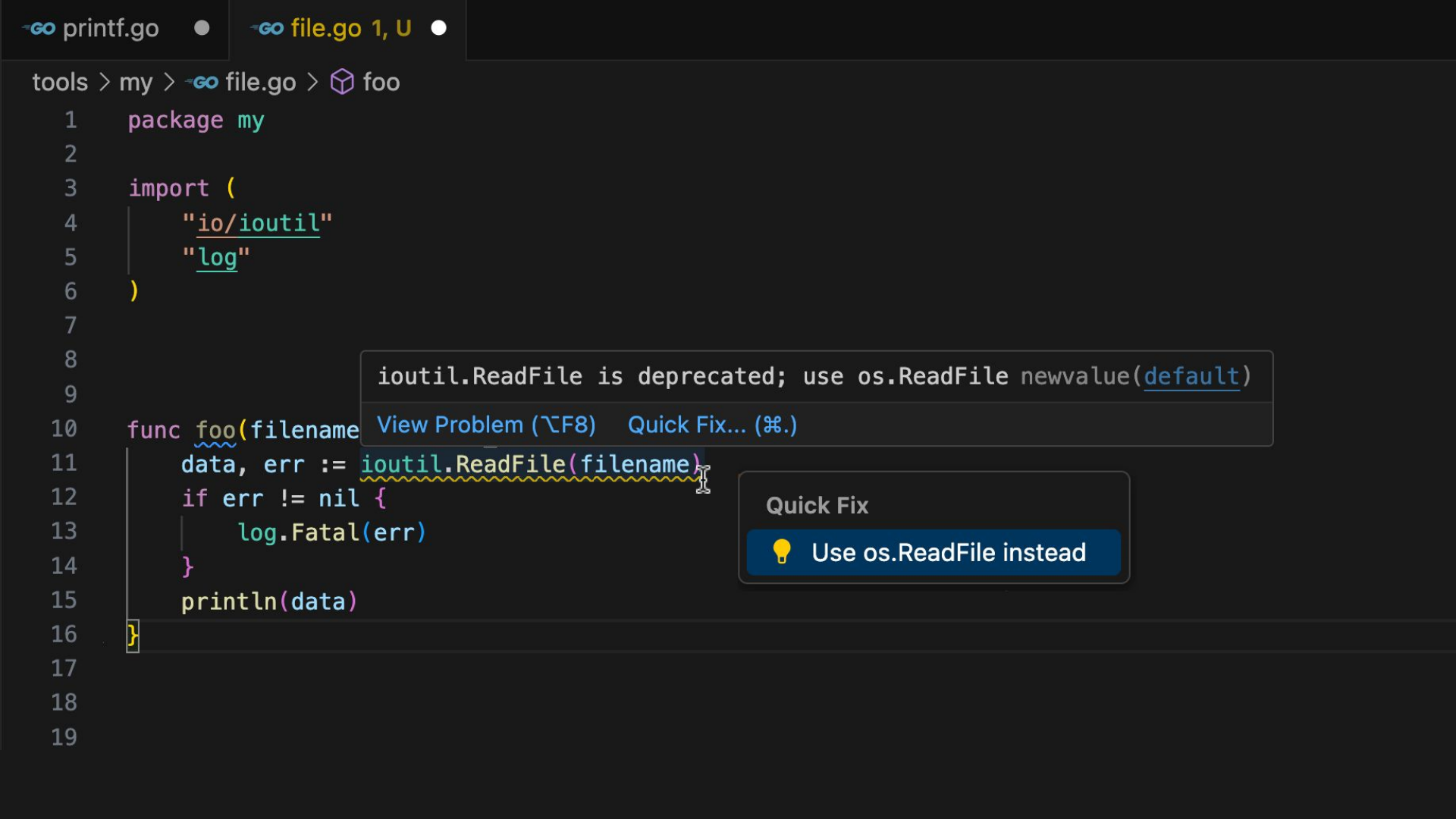
```
...

pass.Report(analysis.Diagnostic{
	Pos:     call.Pos(),
	End:     call.End(),
	Message: "ioutil.ReadFile is deprecated; use os.ReadFile",
	SuggestedFixes: []analysis.SuggestedFix{{
		Message: "Use os.ReadFile instead",
		TextEdits: []analysis.TextEdit{{
			Pos:     call.Fun.Pos(),
			End:     call.Fun.End(),
			NewText: []byte("os.ReadFile"),
		}},
	}},
})

...
```

```
$ ./readfile -fix -diff ./my/...
--- my/file.go (old)
+++ my/file.go (new)
@@ -53,7 +53,7 @@
 }

 func foo(filename string) {
-     data, err := ioutil.ReadFile(filename)
+     data, err := os.ReadFile(filename)
      if err != nil {
           log.Fatal(err)
      }
```


printf.go
file.go 1, U
tools > my > file.go > foo
1 package my
2
3 import (
4 "io/ioutil"
5 "log"
6)
7
8
9
10 func foo(filename
11 data, err := ioutil.ReadFile(filename)
12 if err != nil {
13 log.Fatal(err)
14 }
15 println(data)
16 }
17
18
19
ioutil.ReadFile is deprecated; use os.ReadFile newvalue(default)
View Problem (⌥F8)
Quick Fix... (⌘.)
Quick Fix
Use os.ReadFile instead

A *modernizer* is:

… an **Analyzer**
… that **suggests a fix**
… that uses **newer features** of Go
… and is **safe to apply** unreservedly.

modernize package

Index

Categories of modernize diagnostic:

- forvar: remove x := x variable declarations made unnecessary by the new semantics of loops in go1.22.
- slicescontains: replace 'for i, elem := range s { if elem == needle { ...; break }}' by a call to slices.Contains, added in go1.21.
- minmax: replace an if/else conditional assignment by a call to the built-in min or max functions added in go1.21.
- sortslice: replace sort.Slice(s, func(i, j int) bool { return s[i] < s[j] }) by a call to slices.Sort(s), added in go1.21.
- efaceany: replace interface{} by the 'any' type added in go1.18.
- mapsloop: replace a loop around an m[k]=v map update by a call to one of the Collect, Copy, Clone, or Insert functions from the maps package, added in go1.21.
- fmtappendf: replace []byte(fmt.Sprintf...) by fmt.Appendf(nil, ...), added in go1.19.
- testingcontext: replace uses of context.WithCancel in tests with t.Context, added in go1.24.
- omitzero: replace omitempty by omitzero on structs, added in go1.24.
- bloop: replace "for i := range b.N" or "for range b.N" in a benchmark with "for b.Loop()", and remove any preceding calls to

rangeint

```
for i := 0; i < n; i++ {
        ...
}
```

→

```
for i := range n {
        ... i ...
}
```

or

```
for range n {
        ...
}
```

rangeint: changes final value of i

```
var i int

for i = 0; i < n; i++ {
        ...
}

print(i) // prints n
```

→

```
var i int

for i = range n {
        ...
}

print(i) // ⚠️ prints n-1
```

rangeint: forgot that := and = are not the only assigns

```
for i := 0; i < n; i++ {
        ...
        i += k
}
```

→

```
for range n { // eliminated var i
        ...
        i += k // oops, didn't notice this
}
```

rangeint: changes type of loop var

```
const n = 3.0

for i := 0; i < n; i++ {
        var _ int = i // i is an int
}
```

→

```
for i := range n {
        var _ int = i // ⚠️ i is a float64
}
```

Correct:

```
for i := range int(n) {
```

rangeint: ignores changes to loop bound

```
// loop may run fewer than n times
for i := 0; i < n; i++ {
        ...
        if cond { n-- }
}
```

→

```
// ⚠️ loop always runs n times
for range n {
        ...
        if cond { n-- }
}
```

What about comments?

```
found := false
for _, field := range fields {
        // Dash needs special handling.
        if field == "-" {
                found = true
                break // no need to continue
        }
}
```

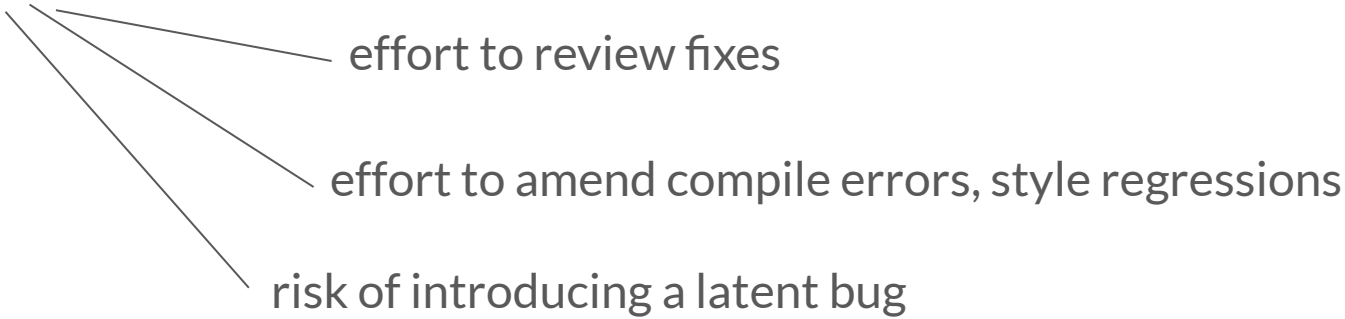
→

```
found := slices.Contains(fields, “-”)
```

Cost/benefit analysis

Modernizers don't fix bugs
=> the **benefit** of each fix is small

To reduce the cost/benefit ratio to an acceptable level
=> the **costs** must be *very* small!



How can we make modernization safer?
How can we make it self-service?

```
package ioutil // "io/ioutil"

import "os"

// ReadFile reads the file named by filename and returns the contents.
// A successful call returns err == nil, not err == EOF. Because ReadFile
// reads the whole file, it does not treat an EOF from Read as an error
// to be reported.
//
// Deprecated: As of Go 1.16, this function simply calls [os.ReadFile].
func ReadFile(filename string) ([]byte, error) {
      return os.ReadFile(filename)
}
```

```
// Package oldmath is the bad old math package.
package oldmath

import "newmath"

// Sub returns x - y.
// Deprecated: the parameter order is confusing.
func Sub(y, x int) int {
	return newmath.Sub(x, y)
}

// Inf returns positive infinity.
// Deprecated: there are two infinite values; be explicit.
func Inf() float64 {
	return newmath.Inf(+1)
}

// Neg returns -x.
// Deprecated: this function is unnecessary.
func Neg(x int) int {
	return newmath.Sub(0, x)
}
```

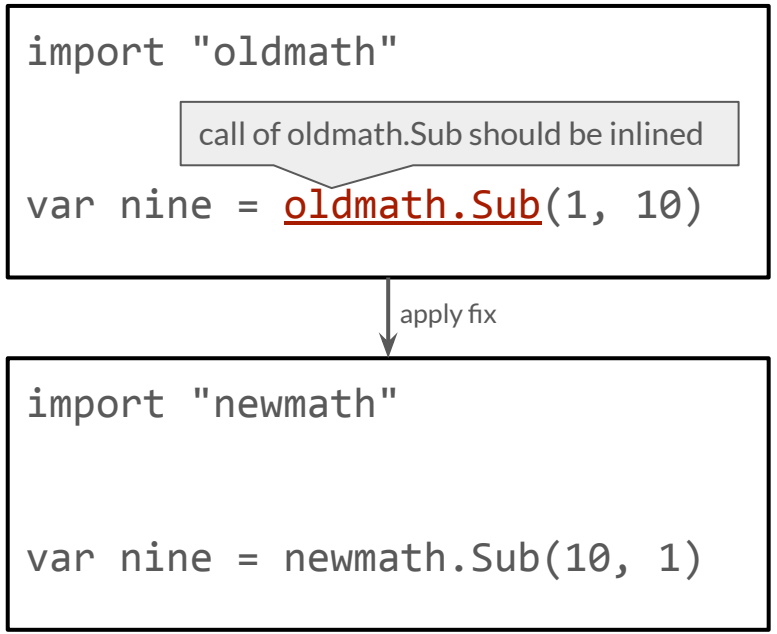
```
// Package oldmath is the bad old math package.
package oldmath

import "newmath"

// Sub returns x - y.
//go:fix inline
func Sub(y, x int) int {
	return newmath.Sub(x, y)
}

// Inf returns positive infinity.
//go:fix inline
func Inf() float64 {
	return newmath.Inf(+1)
}

// Neg returns -x.
//go:fix inline
func Neg(x int) int {
	return newmath.Sub(0, x)
}
```



Source-level inlining

```
//go:fix inline
func Sub(y, x int) int {
    return newmath.Sub(x, y)
}
```

naive inlining: changes order of side effects!

```
print(Sub(f(), g()))  →  print(newmath.Sub(g(), f()))
```
// ⚠️

safe inlining: preserves behavior

```
print(Sub(f(), g()))  →  var y = f(); print(newmath.Sub(g(), y))
```

See golang.org/x/tools/internal/refactor/inline for more detail on this problem

```
$ go fix -diff ./my/...                    # Go 1.26?
--- my/file.go (old)
+++ my/file.go (new)
@@ -53,7 +53,7 @@

-    print(Sub(f(), g()))
+    var y = f(); print(newmath.Sub(g(), y))
```

```
//go:fix inline
func p() {
    q() // ignore error
}
```

```
p()                 →   q() // ignore error
```

comment has gone missing!

```
print(Sub(f(), g()))  →  var y = f(); print(newmath.Sub(g(), y))
```

safe—but bad style if f() and g() don’t actually have effects!

Modernizers

`golang.org/x/tools/gopls/internal/analysis/modernize/cmd/modernize`

- Gopls includes modernizers for a couple dozen new features of Go.
- Modernizers are hard to get right. We'll add them only for the most important features.
- Try it out! Apply fixes interactively in your IDE, or *en masse* using a batch command:

```
$ modernize -fix -test packages...
```

Auto-Inliner

`golang.org/x/tools/internal/gofix/cmd/gofix`

- `//go:fix inline` is a different paradigm: safe by design, self-service.
- What other kinds of migrations fit into this declarative paradigm?
- Try it out! Annotate your own codebase and then apply the fixes.

```
$ gofix -fix -test packages...
```

Both tools will appear in a future release of the `go fix` command.

Thanks!