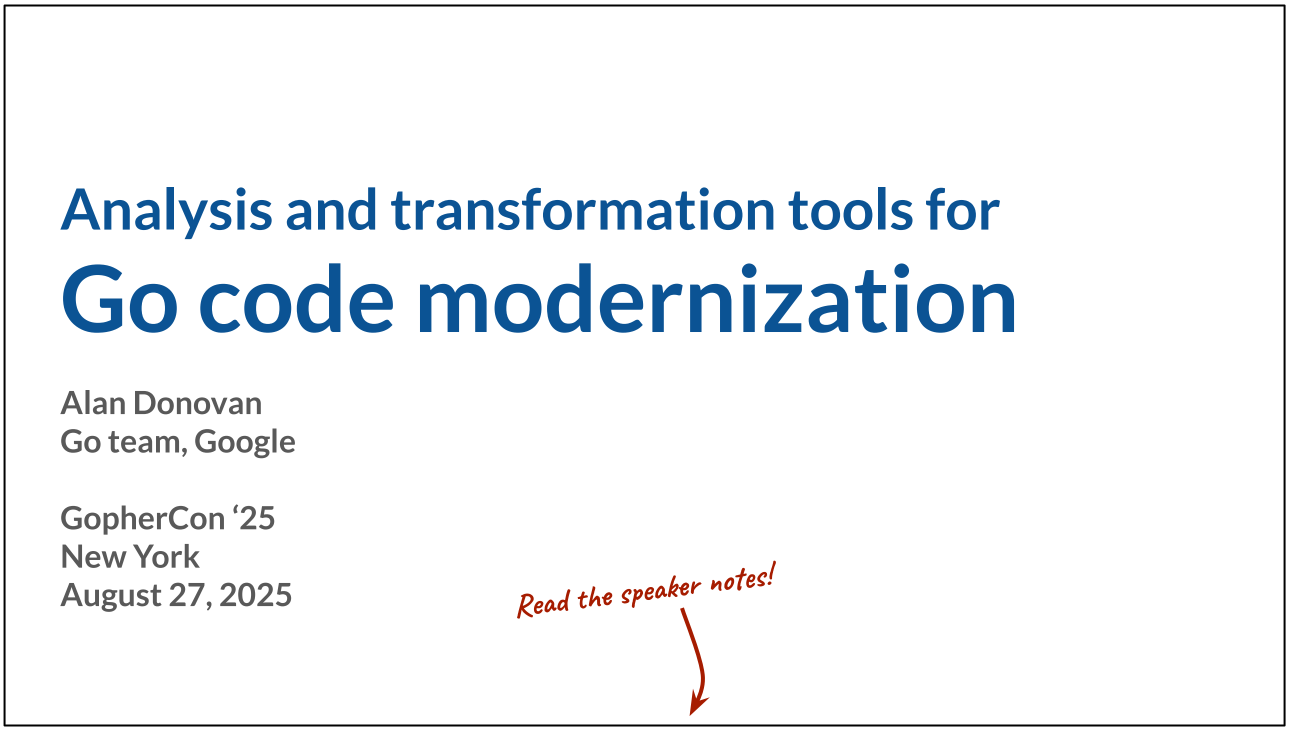


Analysis and transformation tools for

Go code modernization

Alan Donovan
Go team, Google

GopherCon '25
New York
August 27, 2025

Read the speaker notes!

Hello. Thanks for having me here.

Great to see such a big crowd of gophers together in one place.

A bit about me

I work on the Go team here in new york on static analysis tools incl. vet and gopls, which powers your IDE.

I'll be around all week, so come and find me after if you want to chat about static analysis tools.

--

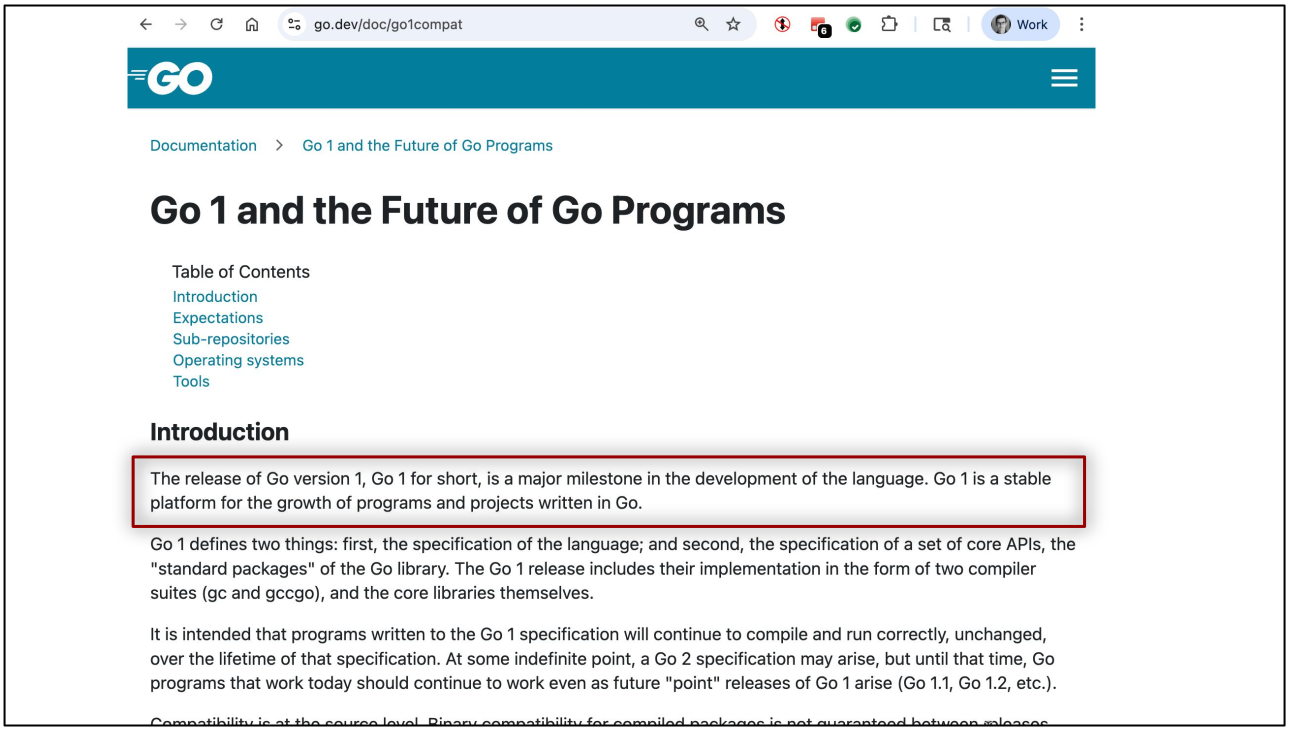
I'm going to talk today about tools for modernizing Go codebases.

First I'll define and motivate the problem.

Then I'll talk about two approaches to modernization that we are pursuing.

Along the way I'll show you what we've built,
and explain some of the challenges we've encountered.

and I'll wrap up with some of the plans and directions for the future.


go.dev/doc/go1compat

Documentation > Go 1 and the Future of Go Programs

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.

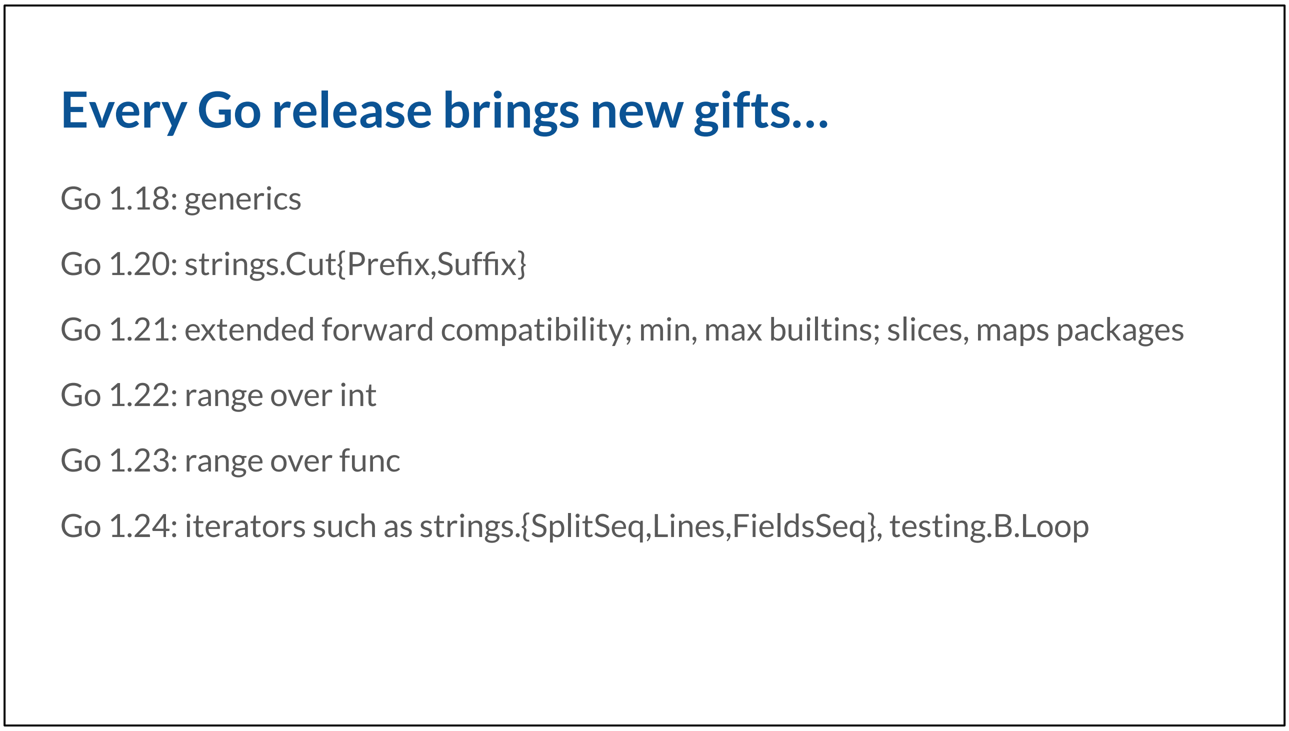
It is intended that programs written to the Go 1 specification will continue to compile and run correctly, unchanged, over the lifetime of that specification. At some indefinite point, a Go 2 specification may arise, but until that time, Go programs that work today should continue to work even as future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

One of the key value propositions of Go is compatibility.

We're all familiar with the Go 1 compatibility promise:
that we will never make breaking changes.
Or as Ian Lance Taylor called it "Go 2 considered harmful".

The promise means that old code continues to work indefinitely.
Users can upgrade with confidence that the new toolchain and library are better than the old, and not arbitrarily different.



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

Yet Go itself continues to evolve, compatibly.
Each new release brings new features, affordances, conveniences: quality of life enhancements.

Here I've listed a selection of language and library features that find use in nearly every Go file, or at least every package.

So just because you can leave old code alone doesn't mean you always want to.

In active code bases, it's often worth modernizing the code to take advantage of newer, better ways to do things.

Also, the "extended forward compatibility" feature of go1.21 makes upgrading easy and transparent,
so the whole ecosystem now tracks closer to the latest Go release.

That means it's easier than ever to start using these newer and more convenient ways of writing Go code.

Let's take a look at an example of a few of them.

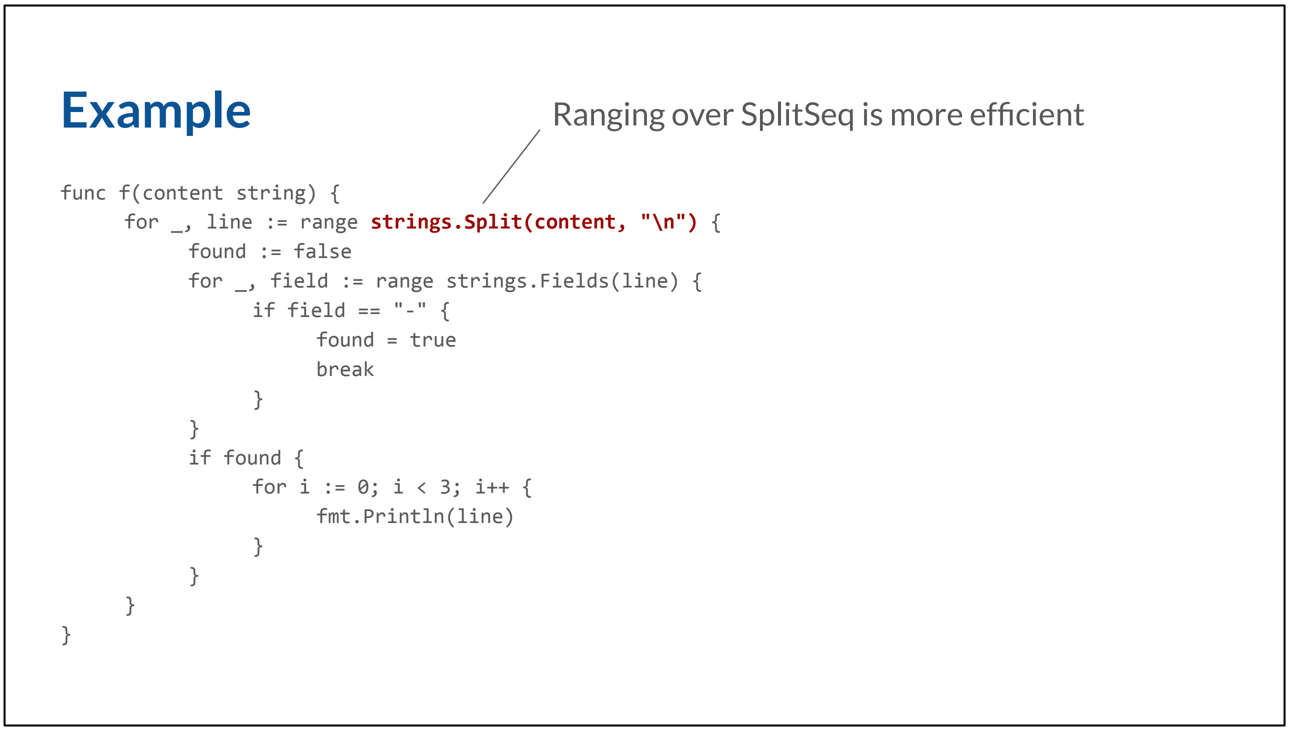
Example

```
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)
			}
		}
	}
}
```

This program (don’t worry about what it actually computes)
is written in a style that was completely normal just a couple of years ago.

But in recent years, there have a been a number of changes to the library and the language
that allow us to rewrite this function more simply.
These are the kinds of things we would (I hope) all point out in a code review.

Let’s look at them in turn.



Example

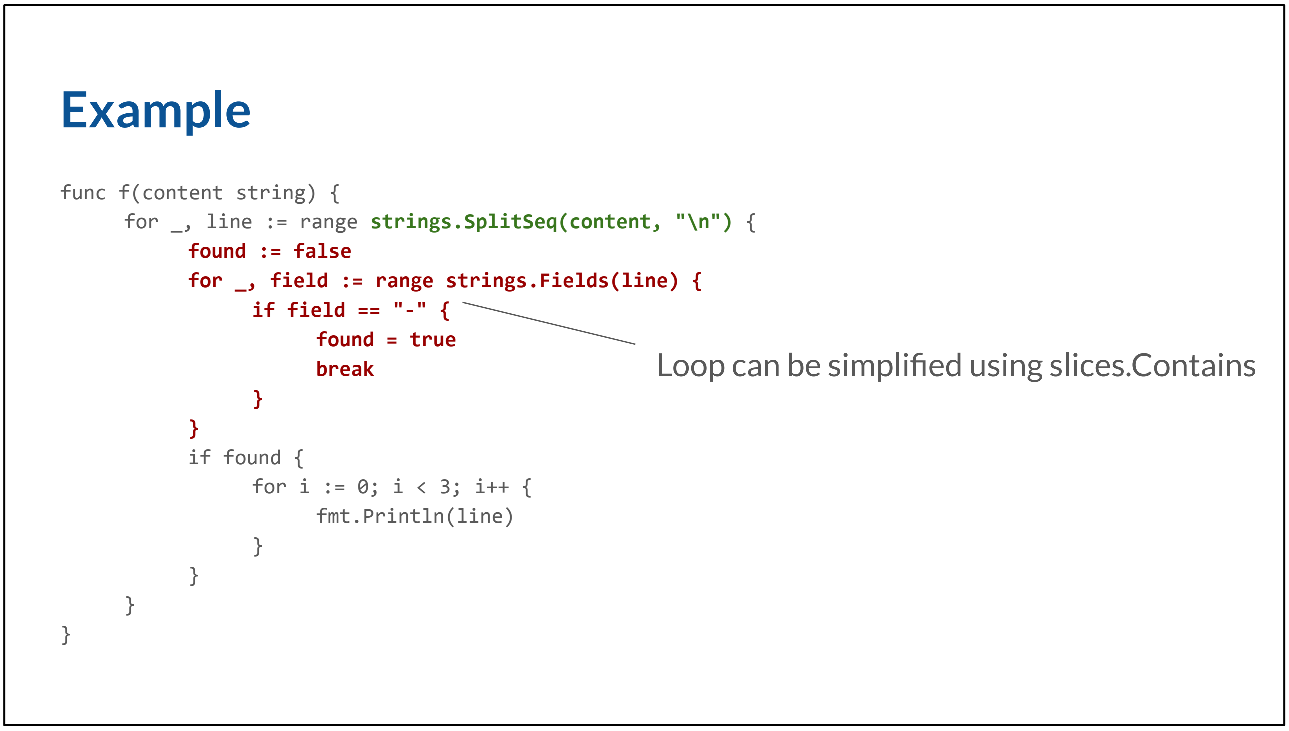
Ranging over SplitSeq is more efficient

```
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)
               }
          }
     }
}
```

The first thing is that we’re trying to range over the lines of a file.
Calling strings.Split allocates an array whose size is the number of lines of the file.
But it’s completely unnecessary, since we’re only going to visit each element once in order.
And if we break out of the loop early, we did a bunch of wasted work.

Go 1.23 added a new feature called range-over-func, aka iterators.
The strings.SplitSeq function is like strings.Split, but instead of computing the entire split eagerly,
it returns an iterator that does it lazily, avoiding the allocation, and avoiding unnecessary work.

So let’s make that change.



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)
			}
		}
	}
}
```

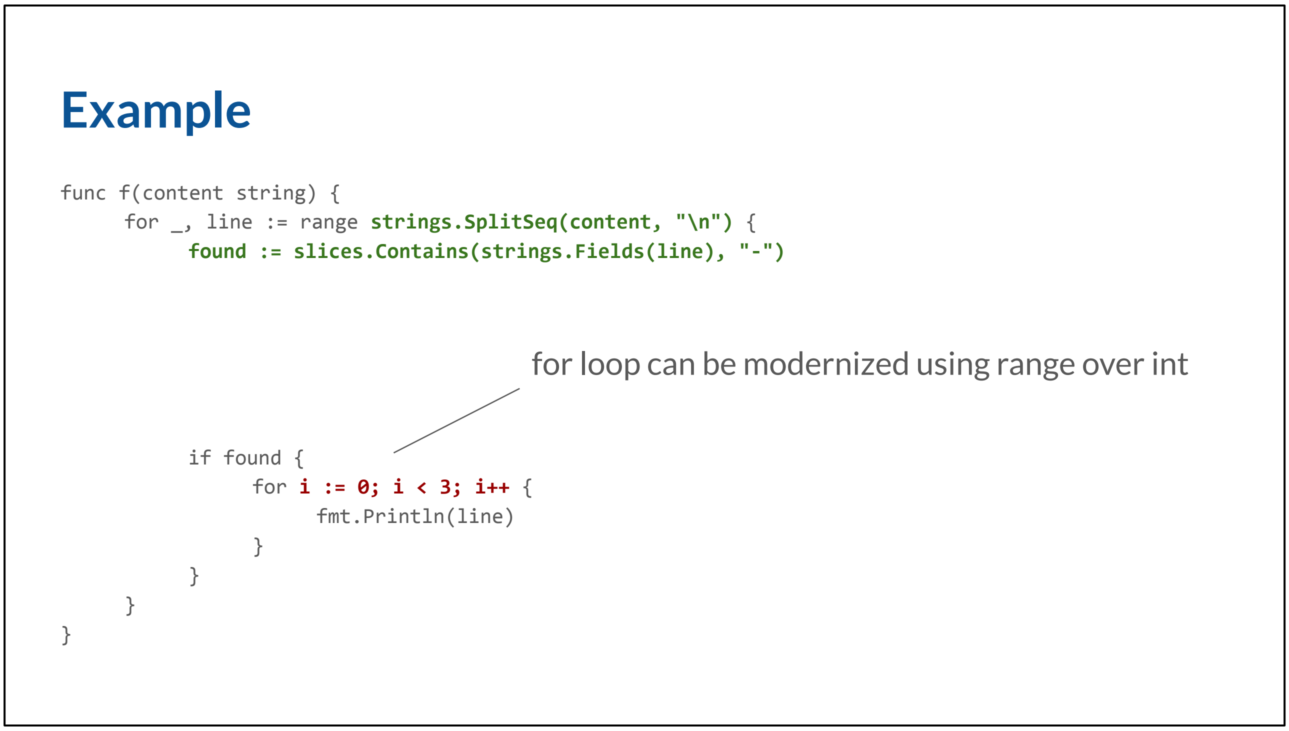
Loop can be simplified using slices.Contains

We’re not done.

The next few lines, which loop over the fields (space separated words) of each line, have essentially the same form as the new library function slices.Contains.
We can rewrite the loop as a call to this function.

A variety of patterns of this form can be replaced by functions in the slices and maps package, such as slices.Delete, maps.Clone, maps.Copy, and so on.
These allow us to eliminate many of these tedious little loops that Go was unfortunately somewhat infamous for.

Let’s make that change.



Example

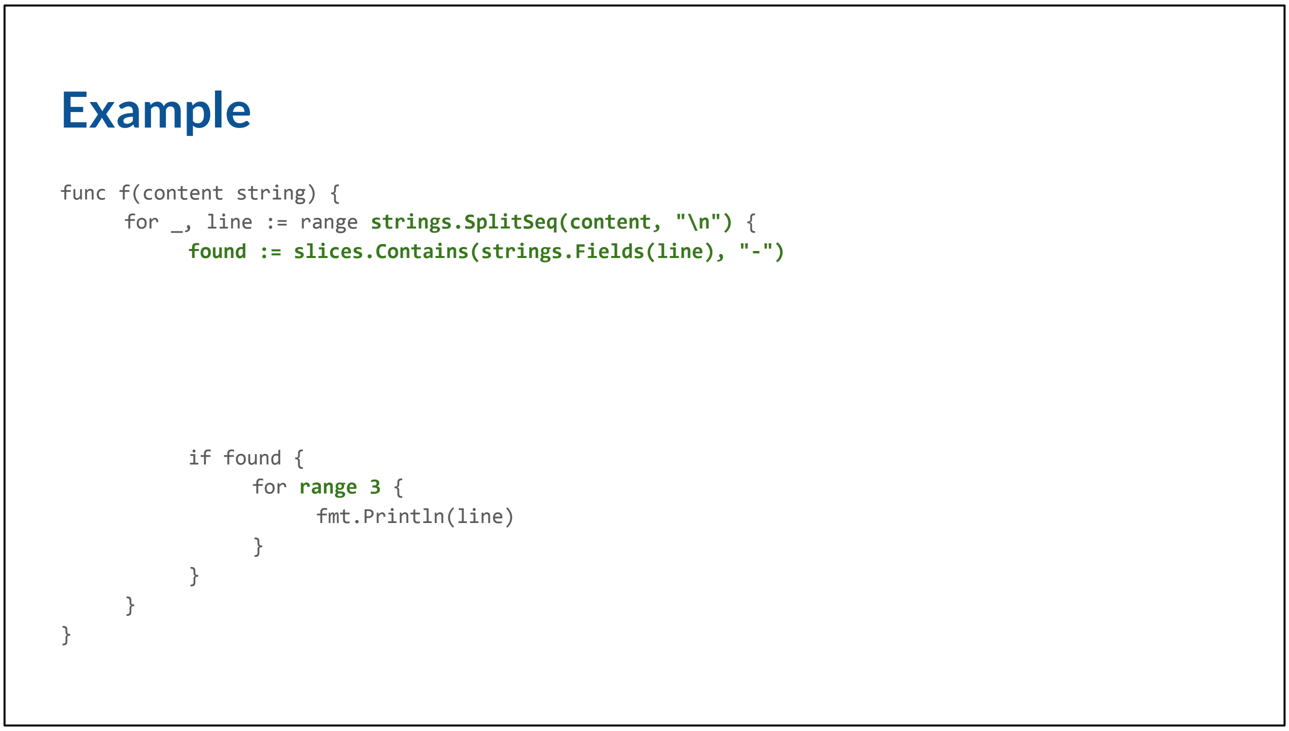
```
func f(content string) {
	for _, line := range strings.SplitSeq(content, "\n") {
		found := slices.Contains(strings.Fields(line), "-")




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

for loop can be modernized using range over int

It saved us 6 lines of code, and made it easier to read at a glance.

Here’s a third one: the 3-clause for-loop can be simplified
using the range-over-int feature from go1.22.

Let’s make that change too.



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)
            }
        }
    }
}
```

OK, now we're done.

This is definitely an improvement in clarity.
It also likely improves memory allocation and performance too, though only a benchmark can really say for sure.

In general, the new features improve readability, consistency, remove distractions, and occasionally improve performance or address security issues

What if we could automate this process?

This would "shift left" the process of code review, saving time for both reviewers and authors.

It would have a second, educational benefit: not only does the code gets better, but so does the developer,
because they learn about new features of the language and library and then start to use them in new code.

So that's one reason to build such a moderniser tool.

But there's another reason.

Example

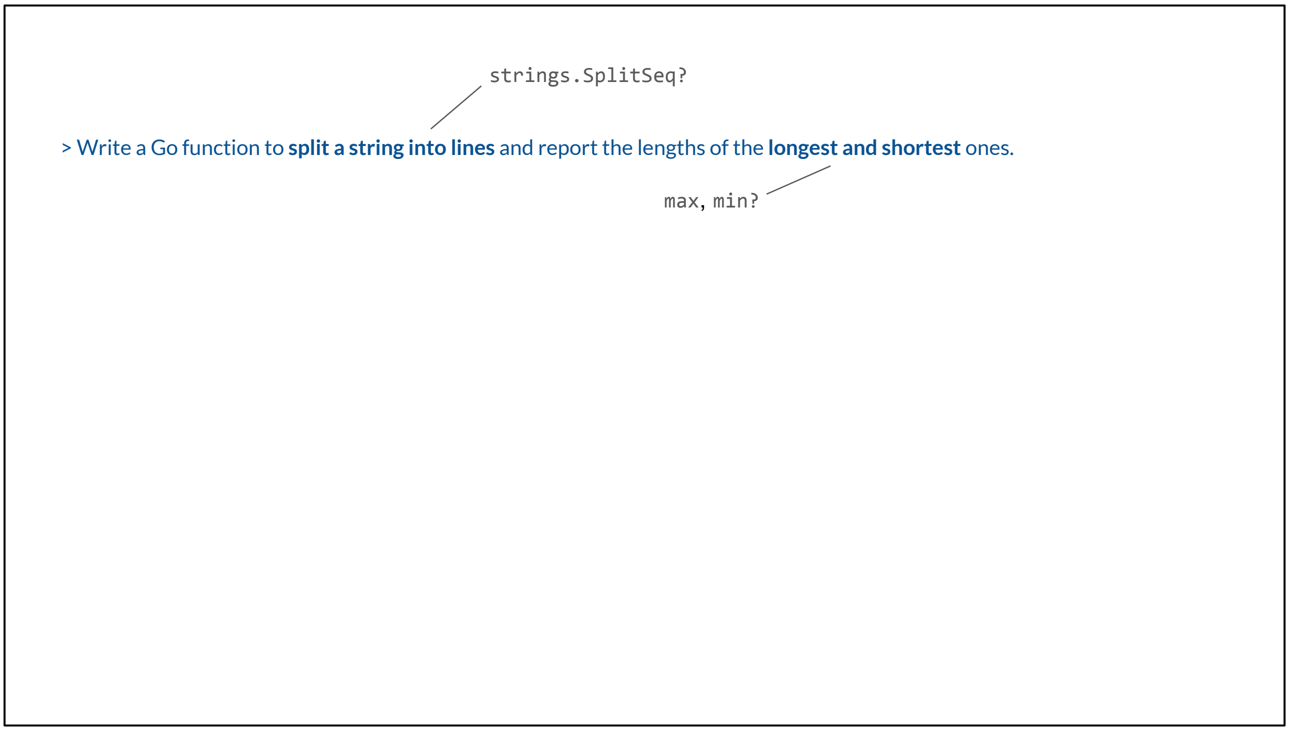
```go
func f(content string) {
    for _, line := range strings.SplitSeq(content, "\n") {
        found := slices.Contains(strings.Fields(line), "-")




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

Let’s talk about the elephant in the room: AI

In 2025, a tool that merely improves code, saves labor, and educates programmers is not enough.
No project is gonna get a green light unless it also welcomes our new robotic overlords.

The problem is that large language models are trained on the corpus of existing text.
By definition, all of that code is “old” with respect to any newly added feature.

So there’s a concern is that models will continue to parrot old code, thereby entrenching that way of doing things.
Models would get stuck in the past, speaking Go with a old-timey accent, like actors in black and white movies.



`strings.SplitSeq?`

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

`max, min?`

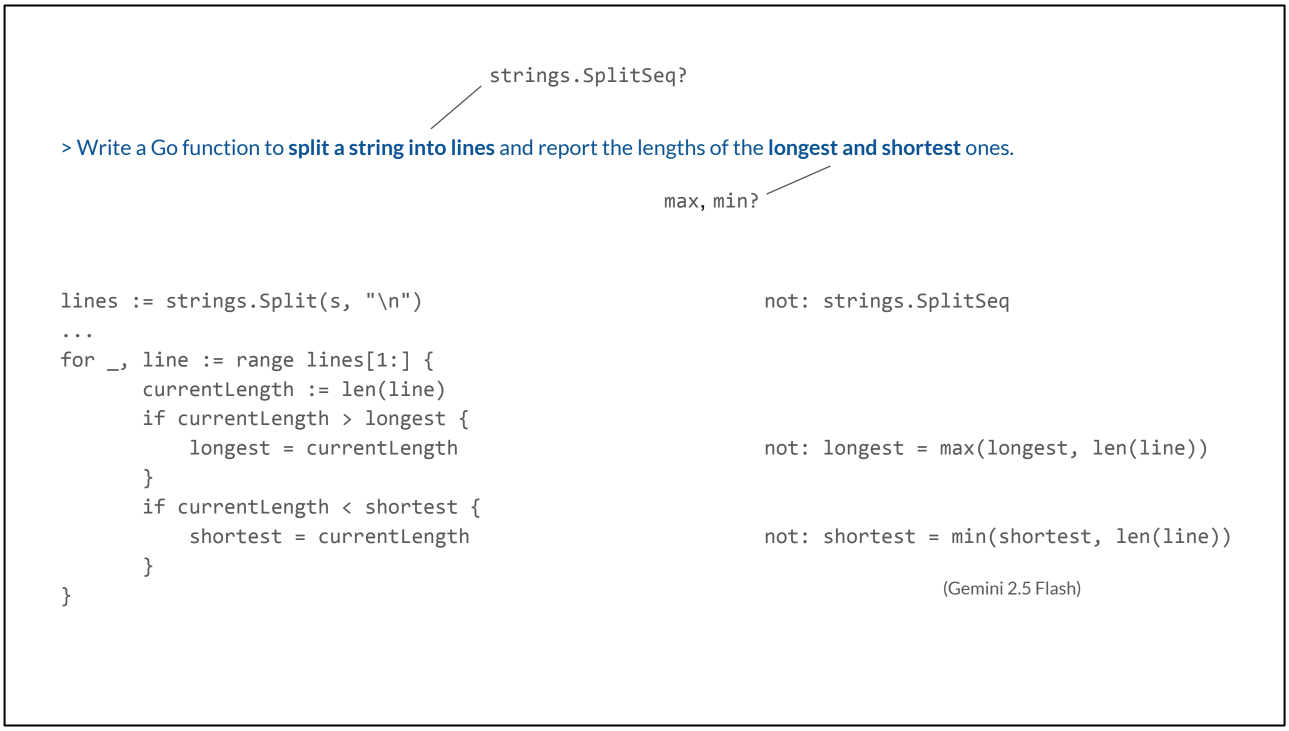
So: Is this a real problem?

Let’s see how well current models do when asked to write Go code that could benefit from newer features.

In other words: How modern is the dialect of Go spoken by an LLM?

I did an experiment.
For about 20 features of Go, I prompted a recent so-called “thinking” model to write a function that might use it.

This example shown actually covers two features: SplitSeq (the iterator version of Split), and the min and max builtins.



`strings.SplitSeq?`

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

`max, min?`

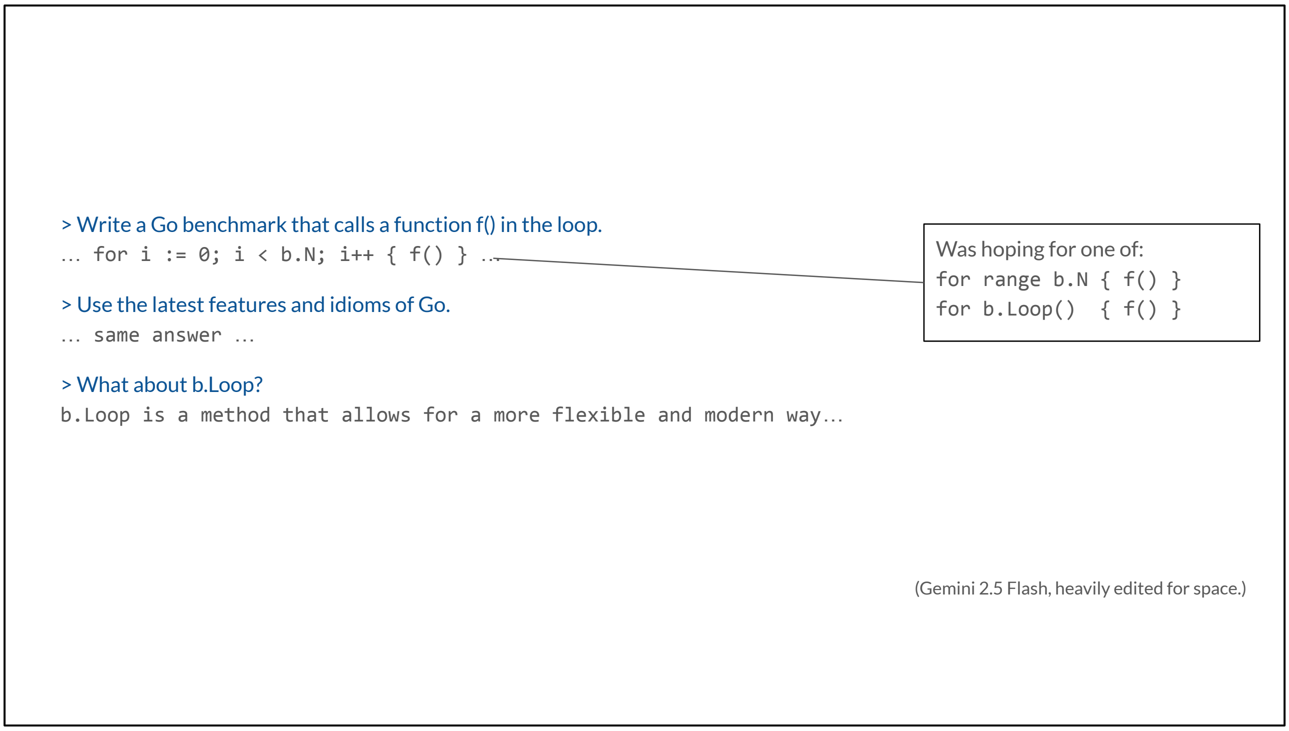
```
lines := strings.Split(s, "\n")                          not: strings.SplitSeq
...
for _, line := range lines[1:] {
       currentLength := len(line)
       if currentLength > longest {
           longest = currentLength                       not: longest = max(longest, len(line))
       }
       if currentLength < shortest {
           shortest = currentLength                      not: shortest = min(shortest, len(line))
       }
}
```

(Gemini 2.5 Flash)

Here’s the output from the model. It failed to use either of the newer features.

What’s more, it did this repeatedly even after various prompts such as “Always use the newest available language and library features of Go.”

And it did this for all of the 20 features I tested.



> 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.)

In fact it is quite robustly stubborn in its refusal to use the new features even though it can answer direct questions about them!

In this example, I asked it to emit a trivial benchmark, knowing that there are two newer ways to write the loop.

Even when told to used newer features, it sticks to its guns.

> 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]*

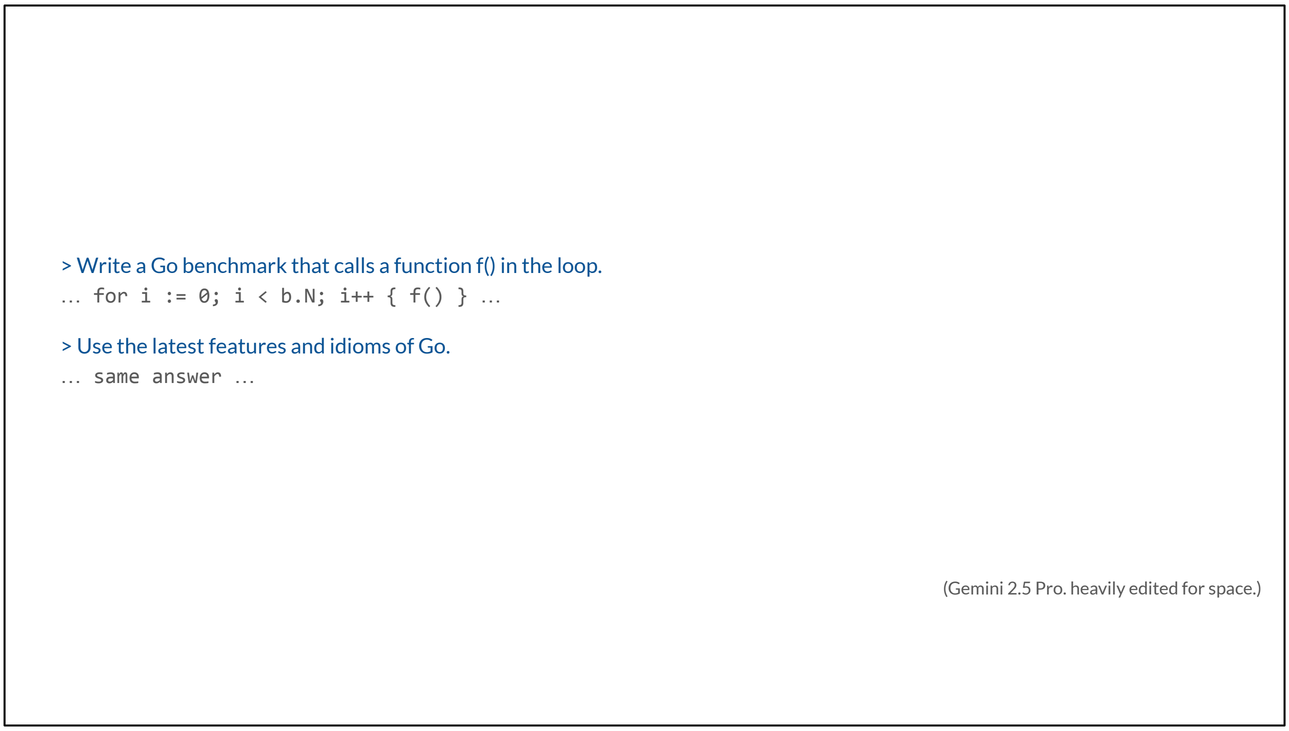
> So why didn't you offer it then??

(Gemini 2.5 Flash, heavily edited for space.)

When the new feature is explicitly suggested, it defends its choice by saying that the old way is more common (which is true).
Even though it says the new way is better!

This was quite consistent across all 20 features I tested.

BTW: most of my LLM sessions seem to devolve into me pleading with a Turing machine, trying to get it to express remorse for the errors of its ways.



> 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.)

So that was all using a faster and cheaper model.

I also tried it with a more heavyweight model, which did somewhat better.
A handful of them it got right off the bat.
A few others it got when gently nudged in the right direction, or it produced comments explaining its choice to use the old way.

Nonetheless it still offered old-style code about half the time,
as you can see here in the benchmark case again.

> 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."?

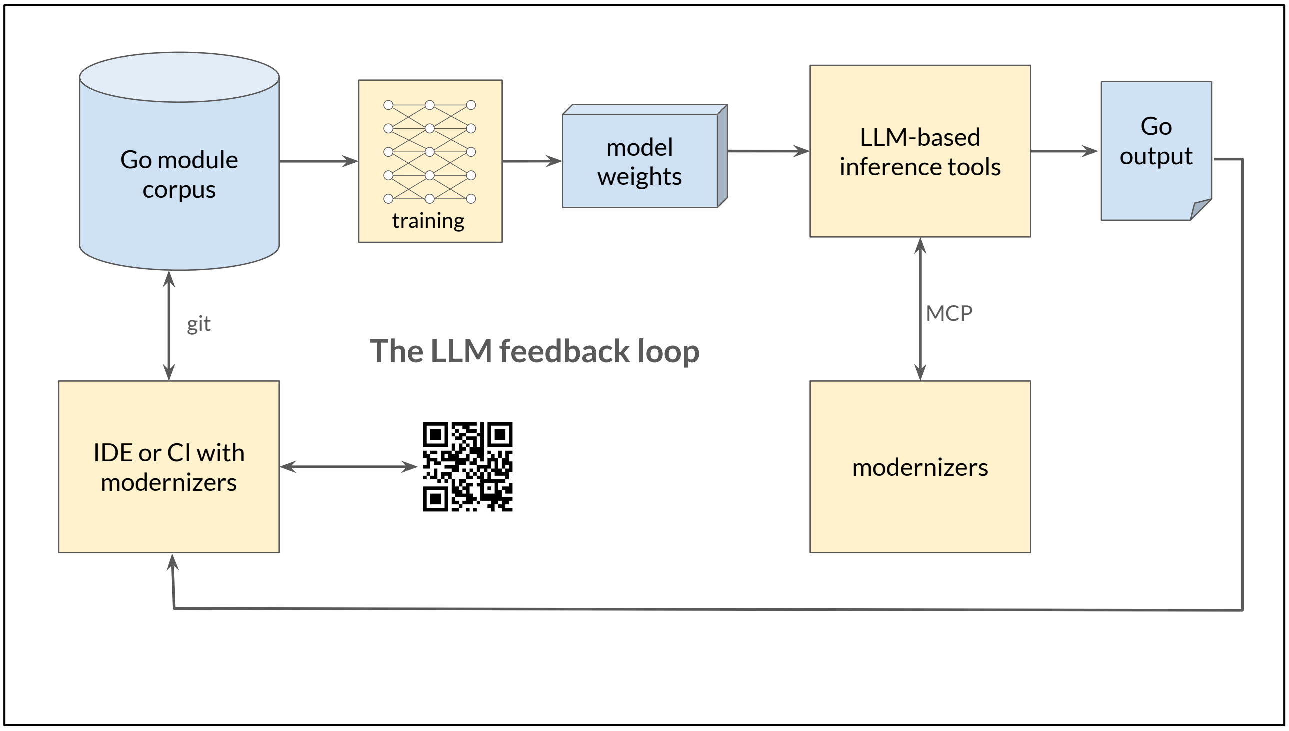
`Excellent question! You are correct that the Go 1.22 release notes mention this, but …`
`GOEXPERIMENT=rangefunc …` *[ever more sophisticated lies]*

(Gemini 2.5 Pro. heavily edited for space.)

But when the fancier model got it wrong, it got it so wrong.

It gives a misleading statement about how the benchmarking tool works.
Then confidently states that range over int doesn’t work.
Then makes up a false history of Go.
Then dodges references to evidence with ever more complex and incorrect explanations.

(Why can I never stop falling into the same LLM trap of hope followed by disappointment?)



In short, there is a real problem here.

The solution to that problem is twofold.

First, we need to fix the model.
That means we need to improve the training data.
And that's another way of saying "fix all the Go code in the world".

The only way we can do that is by making you do it.
And that means we need to give you tools that make it really easy.

Second, we can patch up the model's inferences.
Go's language server gopls already knows the structure of your Go project and offers its services to your editor.
We recently made it also serve the model context protocol (MCP) so that can offer its services to your LLM-based agent.
This will allow it to modernize the snippets of Go code that it emits on the fly.

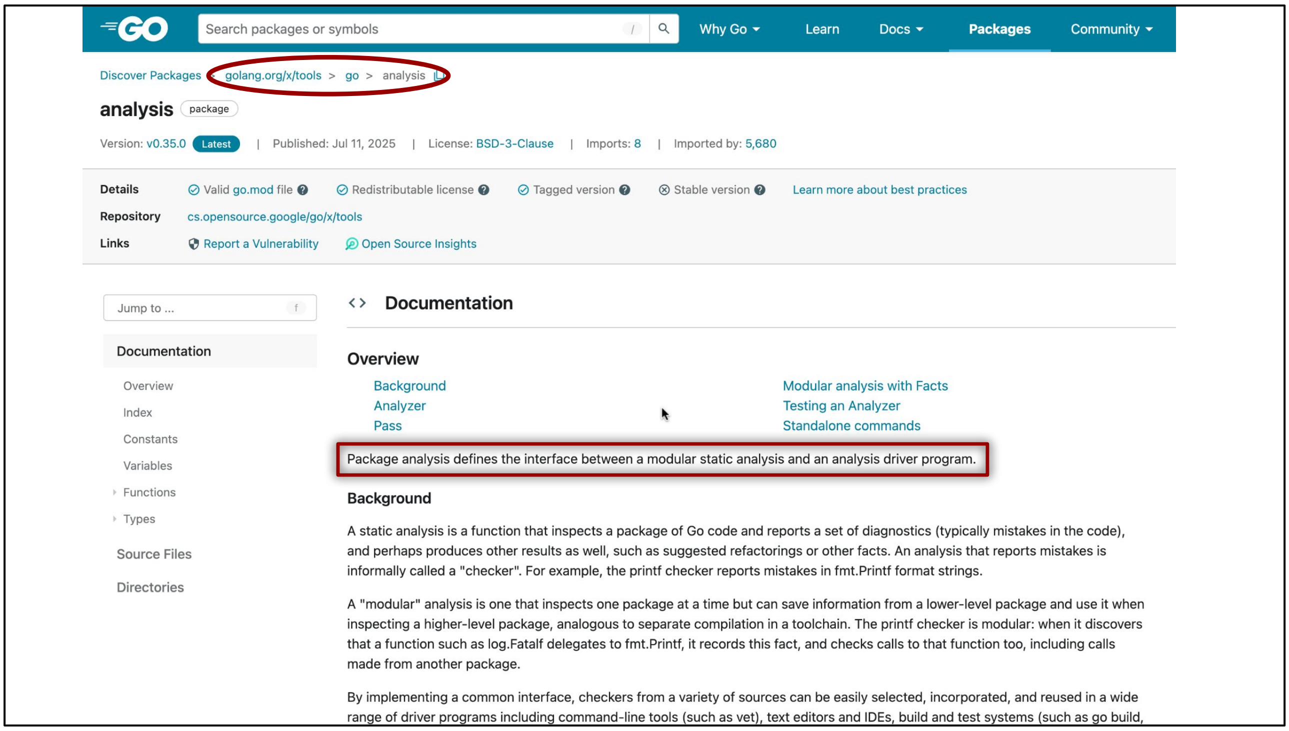
So, both humans and AI agents can benefit from modernizers and produce better code.

[The QR code is an easter egg for a capture-the-flag game by the conference organizers.]

How do we implement modernizers?

So that’s the motivation of the problem.

Now how do we build a tool to automate these kinds of fixes?
For the second part of the talk, I’m going to talk about the implementation of modernizers.



Discover Packages > golang.org/x/tools > go > analysis

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

Background | Modular analysis with Facts
Analyzer | Testing an Analyzer
Pass | Standalone commands

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,

In short, we use the Go analysis framework.
This package, golang.org/x/tools/go/analysis, is the primary API.

It defines the interface between analyzers (aka checkers, linters) and the “driver” programs that run them.

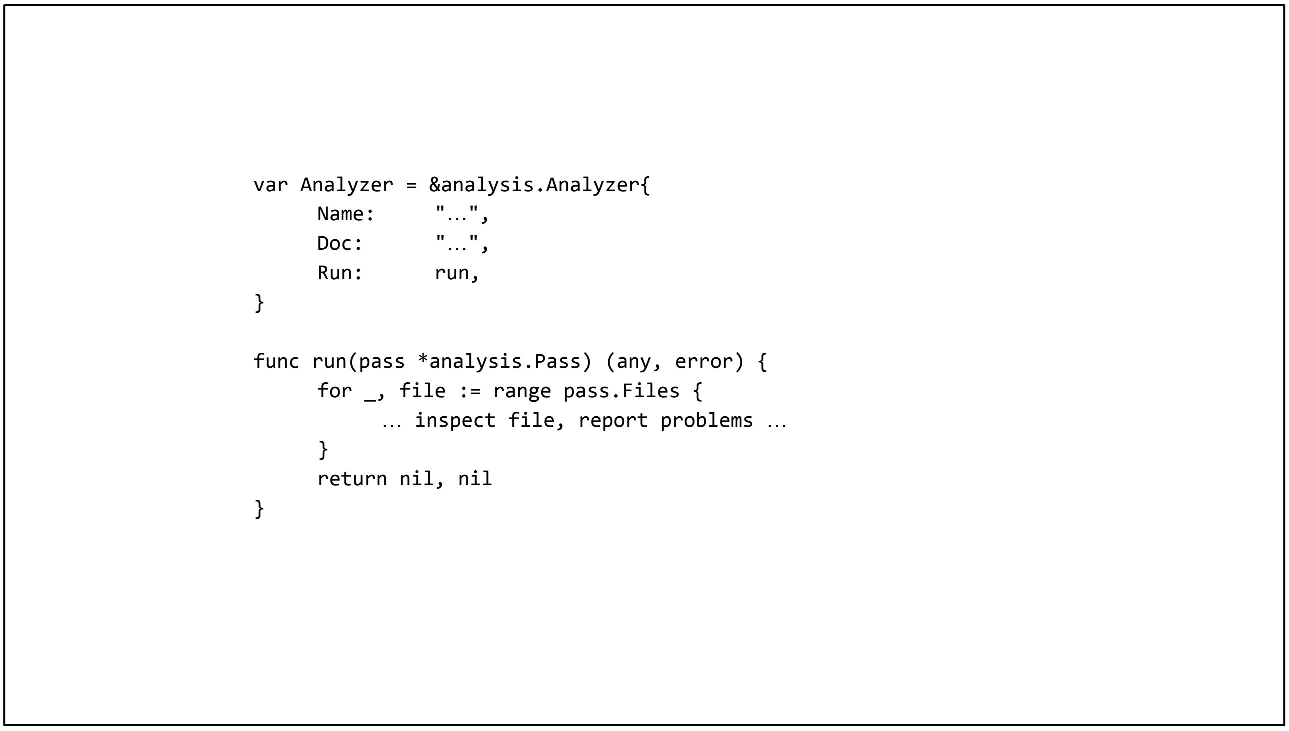
This separation allows us to write a linter once, and have it run anywhere:
For example,

in go vet, in your CI system. (go test also runs a subset of go vet checks.)

in standalone tools (which you can run across your entire repo; or across the entire module mirror corpus of Go code, as we often do)

in code review tools (e.g. Google’s large-scale analysis pipeline), which report diagnostics as code review comments.

in gopls, when you’re editing, so you can fix mistakes straight away.



```
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
}
```

An analyzer is essentially a function that is called for each package in turn,
in dependencies-first order, analogous to the way the compiler builds a set of packages.

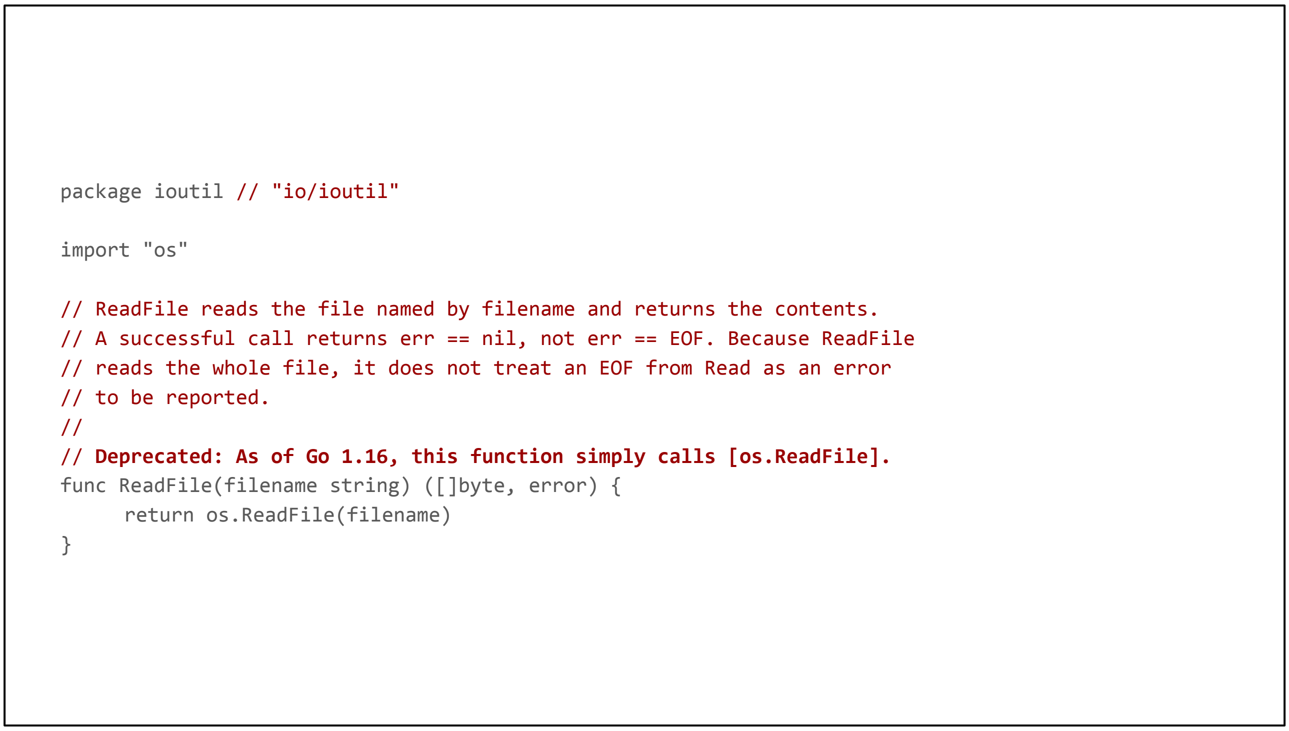
This code shows the general skeleton of an analyzer.

The run function is provided a Pass, which holds the typed syntax trees of the package,
and lots of other information. I'm not going to go into all the details.

It's the job of the analyzer writer to fill in this loop here with whatever they want.
Most analyzers look at the syntax trees and report diagnostics.

You can get as complicated as you want, but It's possible to write interesting interprocedural analyses in a few dozen lines of code

Let's now build a simple analyzer that modernizes some old code.



```go
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)
}
```

Here's a function that you've all used at some point: ioutil.ReadFile.
It reads the entire contents of a file.

In Go1.16, we realized that the ioutil package was a sort of grab bag of things that belonged in other places. (Often packages having "util" in the name is a clue.)
So we moved all the functions out to better places.

Of course, because of Go1 compatibility, we had to keep the old functions working, so we couldn't delete them. Instead, we deprecated them, and made them simply delegate to the new function, os.ReadFile.

Let's write a modernizer that finds calls to this deprecated function.

```
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
}
```

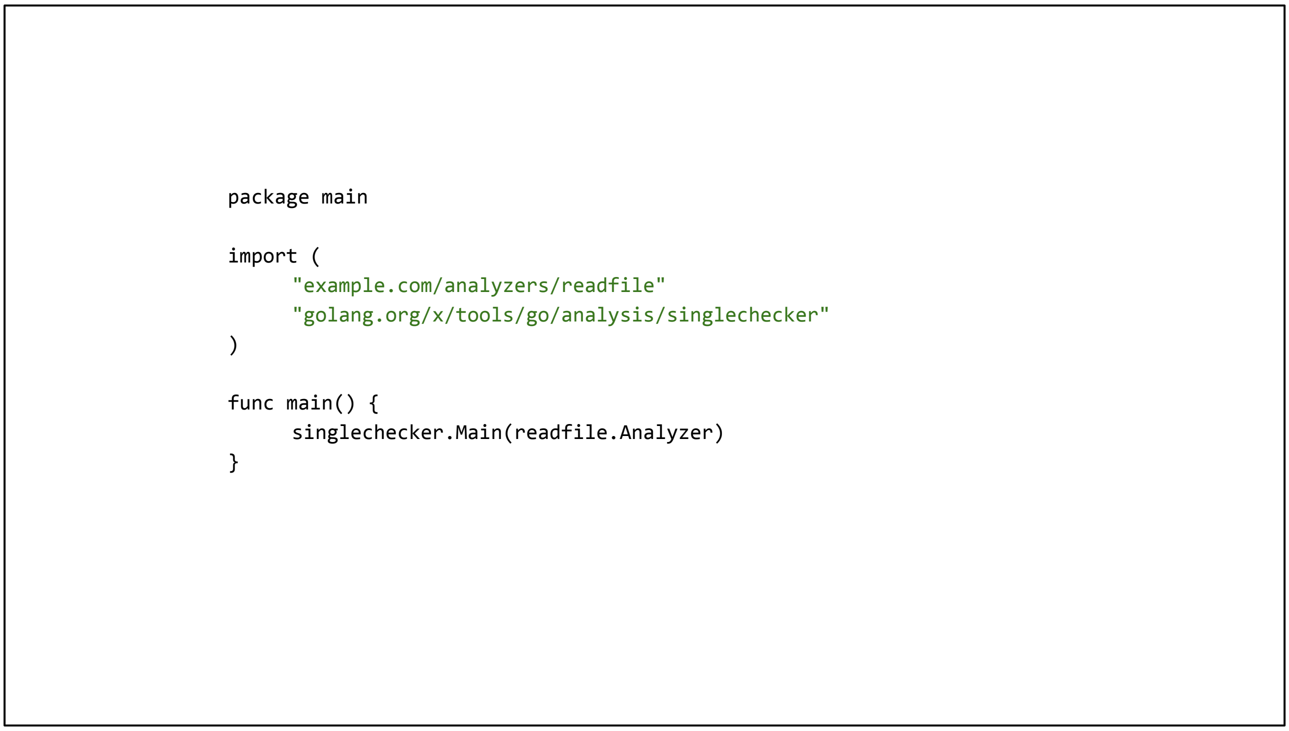
(`isFunctionNamed` helper omitted for space)

Here it is.

The run function looks at each file of the package,
inspects the entire syntax tree looking for function calls,
checks whether the *callee* of the function has the right name (ioutil.ReadFile),
and then reports a diagnostic at the function call.

The diagnostic suggests that you use the newer os.ReadFile function.

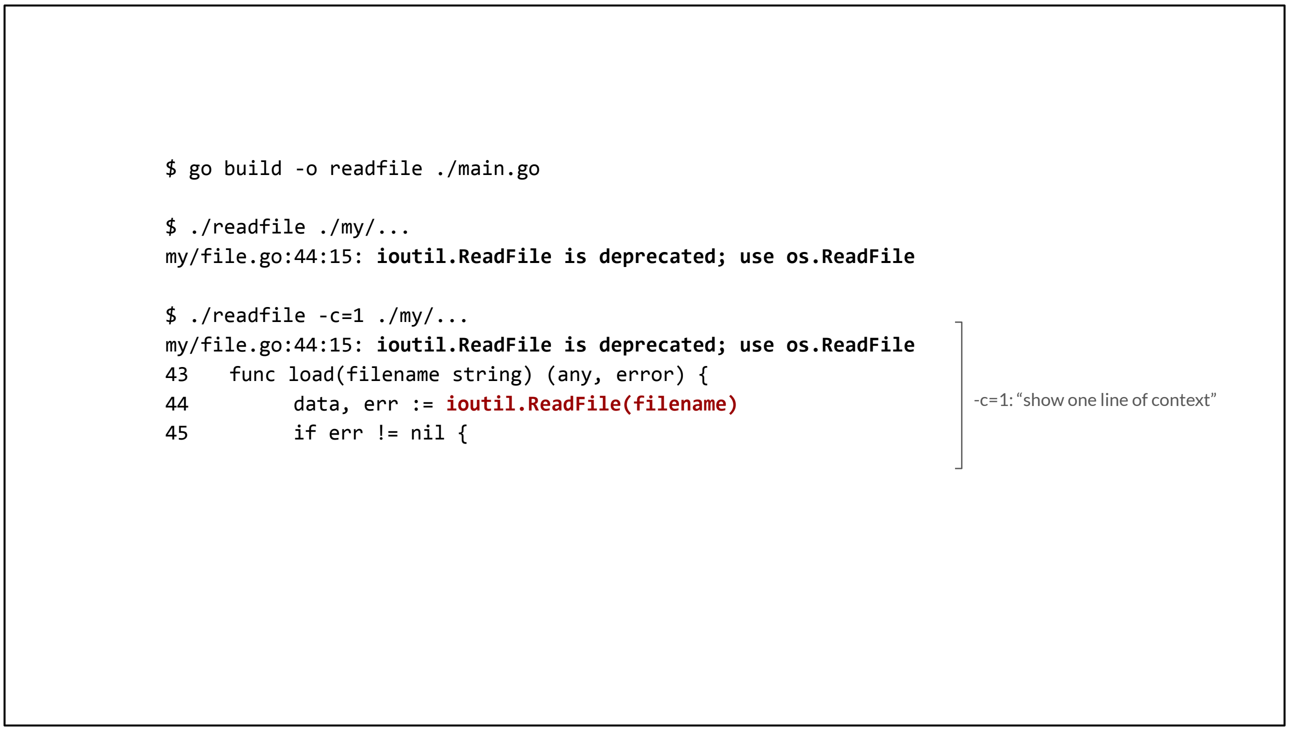
So this is basically all it takes to write an analyzer.
But the analyzer needs to be incorporated within a driver that will make a sequence of calls to the run function.
Let’s look at that now.



```go
package main

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

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

This one-line program is all it takes to turn an analyzer into a standalone command.

We use a package called singlechecker, which produces a complete application that runs a single analyzer.
All you need to do is tell it which analyzer to use. Here we use our 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”

Now we can build an executable, and run it on some code.
By default it reports each error on one line.

But it has a bunch of standard flags for JSON output and suchlike.
The -c flag causes it to display a few lines of context around the problem.

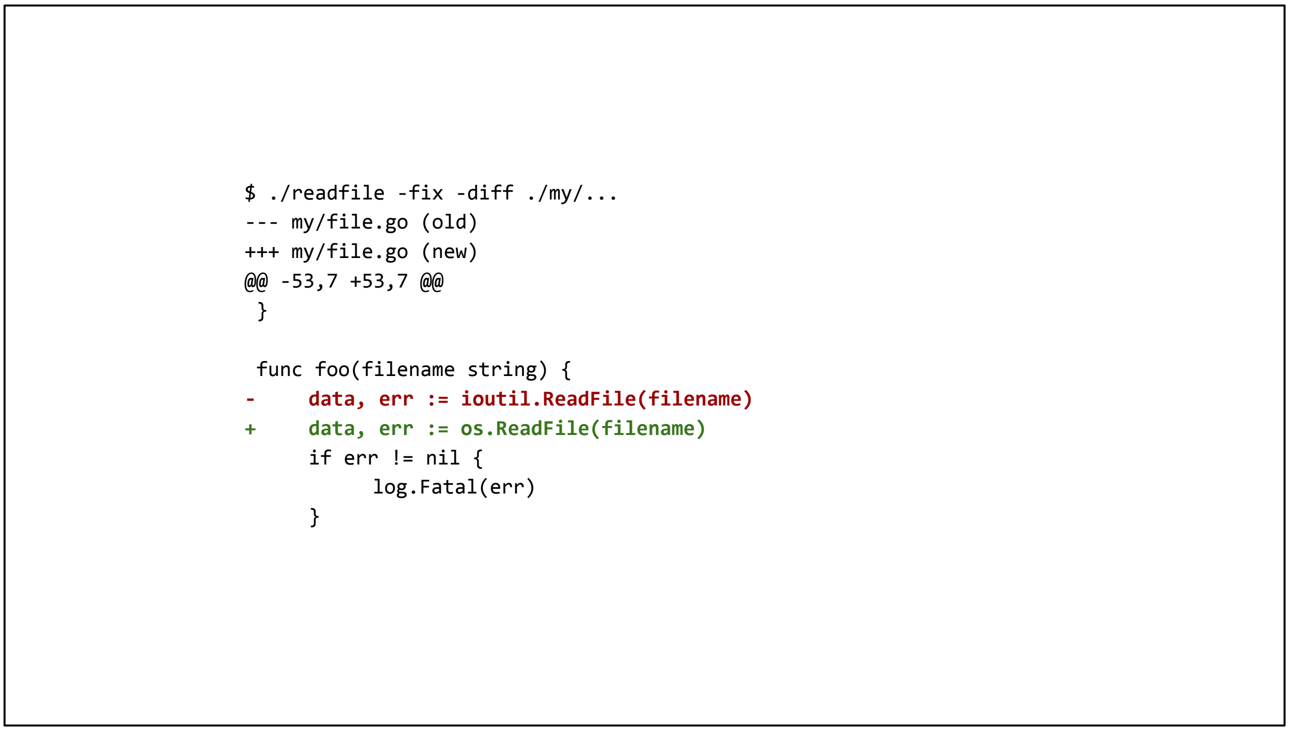
So this is enough to detect the problem and provide a diagnostic. But we can do better.

```
...

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"),
          }},
     }},
})

...
```

Each Diagnostic has the option to suggest one or more fixes.
A fix is basically a set of little patches to the code.

This slide shows code to suggest a fix that rewrites the ioutil.ReadFile part of the call by os.ReadFile.

This is just a toy example. A production-grade fix should also tweak the file's import declaration to ensure that it imports the "os" package.

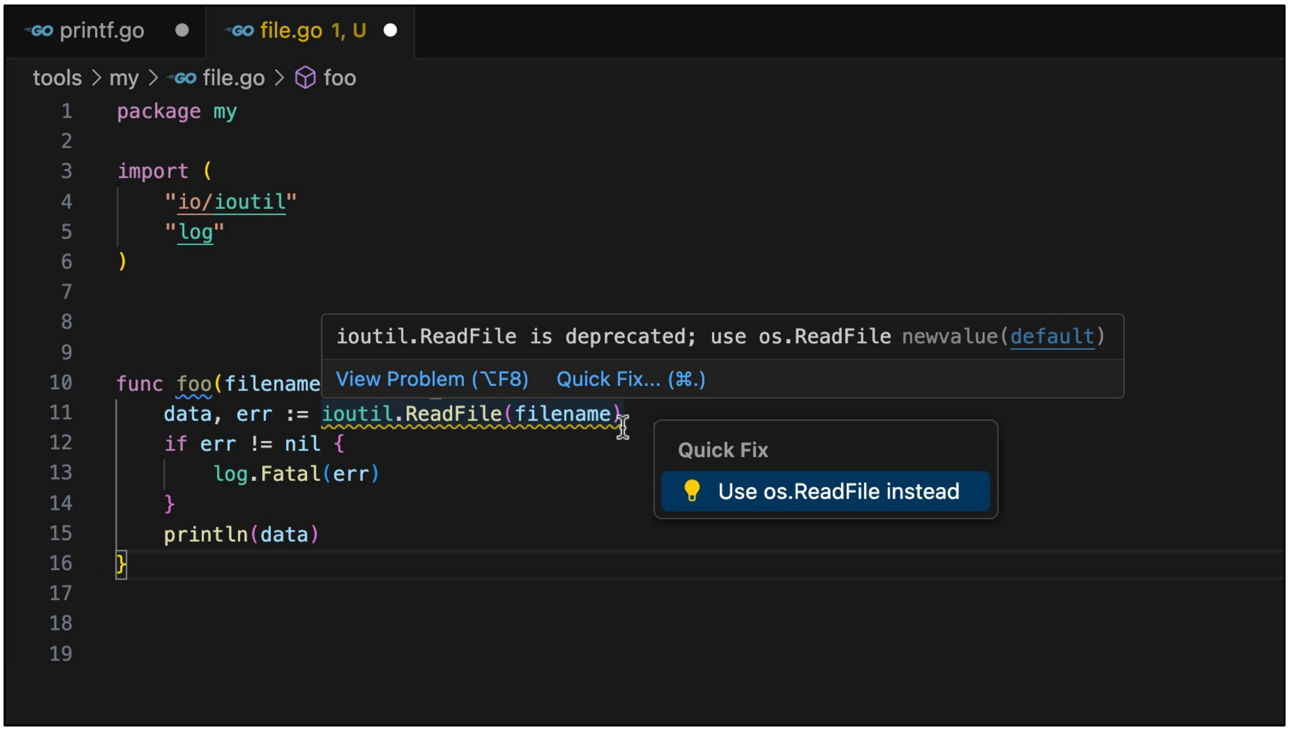


```
$ ./readfile -fix -diff ./my/...
--- my/file.go (old)
+++ my/file.go (new)
@@ -53,7 +53,7 @@
 }

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

So now the singlechecker command not only reports the problem, it can fix it directly.

Just run it with the -fix flag, and it will quietly patch your source code.
You can fix your whole project in one command.

Here I’ve also added the -diff flag so you can see the change it would have applied.



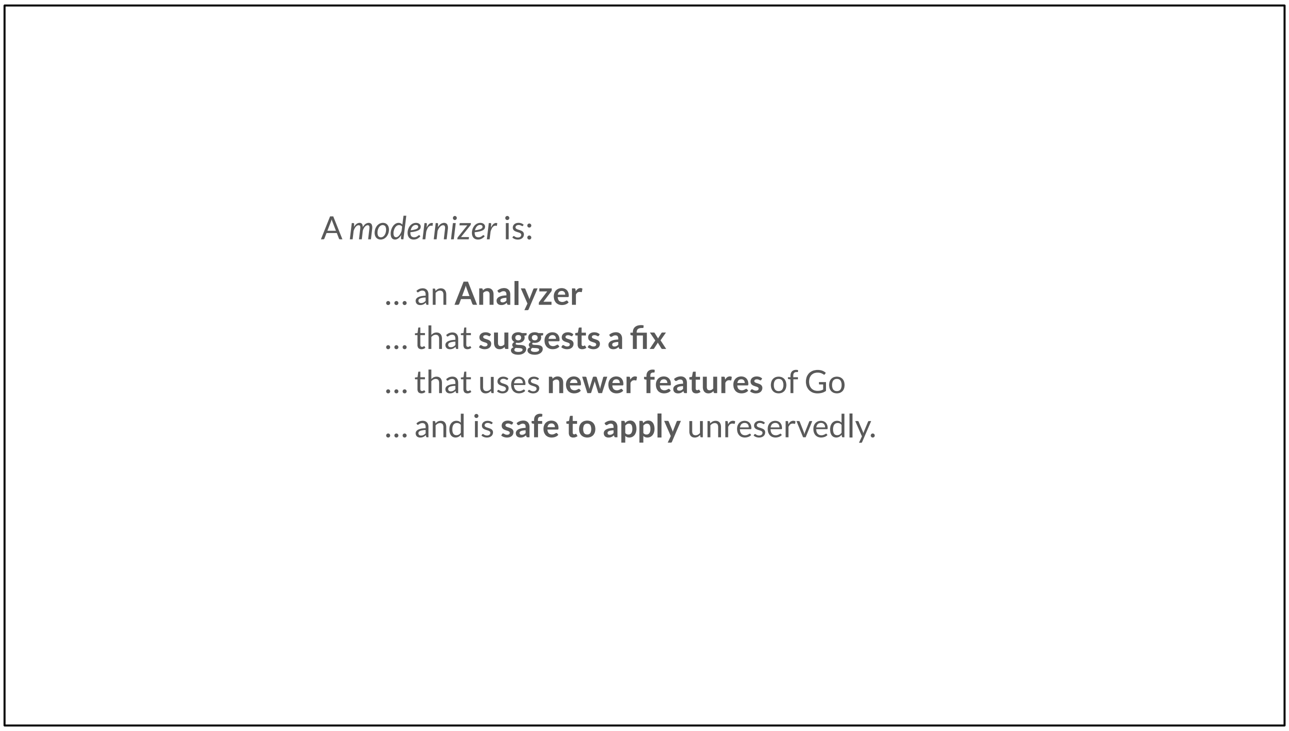
```go
package my

import (
	"io/ioutil"
	"log"
)


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

```
ioutil.ReadFile is deprecated; use os.ReadFile newvalue(default)
```

So that's the command line tool.
But the same analyzer can be linked into other drivers too.

If we add it to gopls, then your editor will start reporting calls to ioutil.ReadFile, and offering to fix it interactively.

BTW, the analysis framework has been around for a while, and a lot of analyzers have been written, for a variety of problems.
Gopls now contains over 200 analyzers, a lot more than go vet.
Many of these come from Dominik Honnef's excellent staticcheck suite.
We recently worked with Dominik on some optimizations that mean we can enable staticcheck by default in gopls.
So you now get efficient real-time feedback on a wide range of issues as you are editing.



A *modernizer* is:

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

That’s really all we need to know today about the analysis framework.

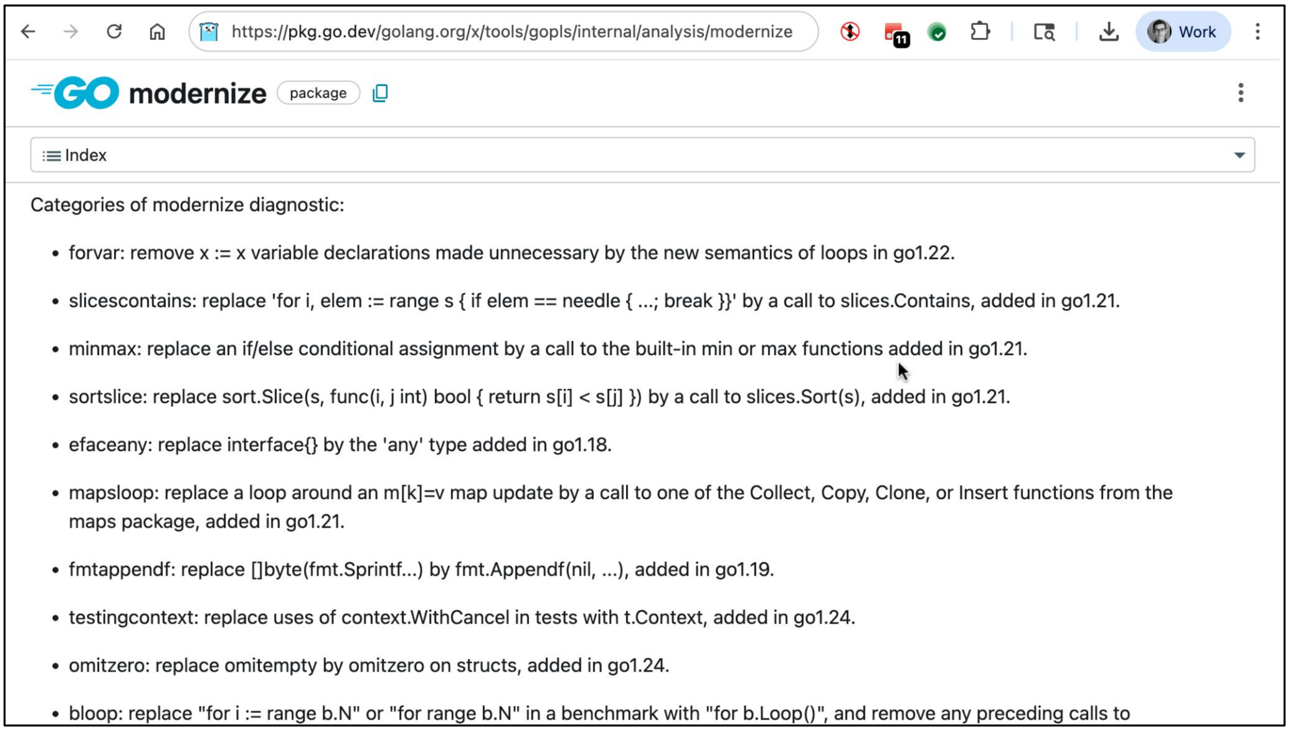
So then, what is a modernizer? Here’s the definition we use.

A modernizer is really just an analyzer, with a few special properties:

It suggests a fix. It’s not enough just to flag the problem, it must fix it too.

The fix makes the code use newer features of Go.

And it’s safe to apply the fix with only cursory review.
It must not introduce a bug.
This is an important point. We’ll come back to it in a moment.


https://pkg.go.dev/golang.org/x/tools/gopls/internal/analysis/modernize

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

At the beginning of this year we developed about twenty of these modernizers as you can see here.
I want to give special thanks to xie yuschen in Singapore, who made a lot of contributions.

We also made the -fix flag work in the presence of merge conflicts so you can apply fixes en masse.

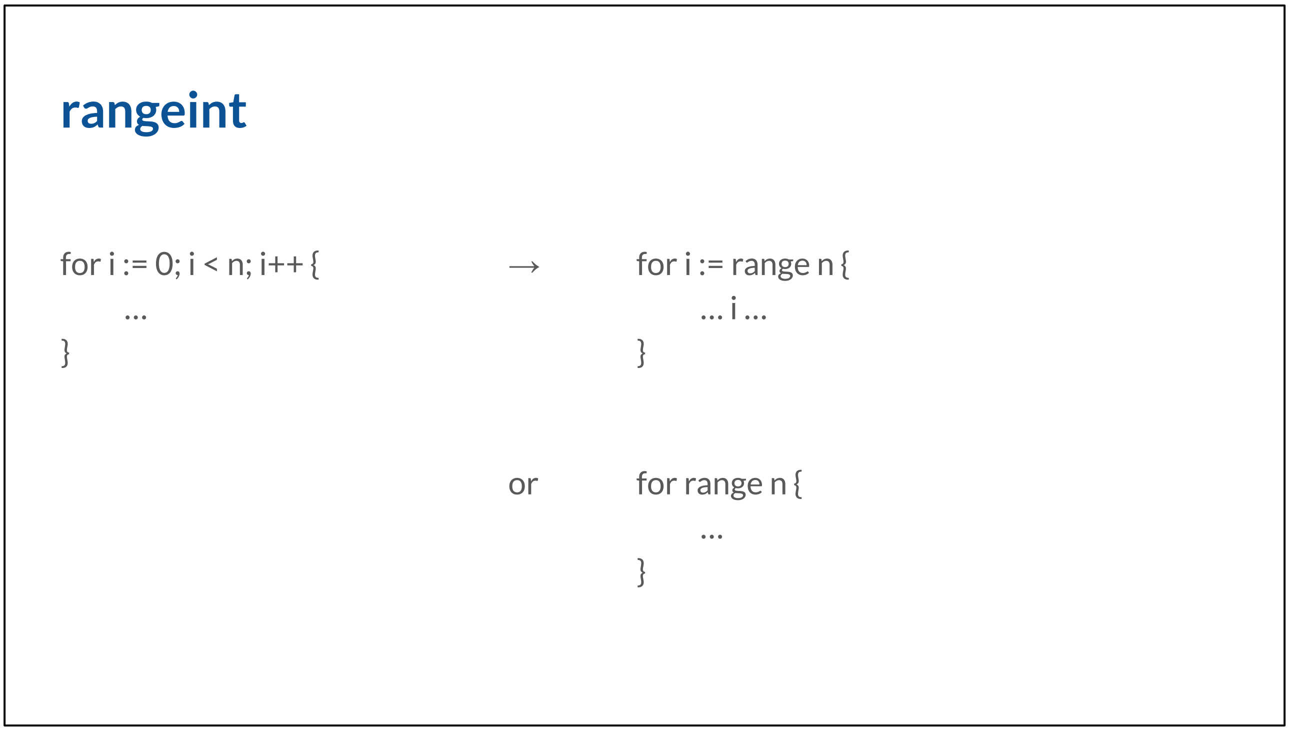
Modernizers were first released in gopls 18 in February.
So you should have been seeing these diagnostics and suggested fixes in your editor for some time now.

Along the way we also built some great infrastructure for analysis and refactoring that sadly I am not going to have time to present despite having promised in in my talk pitch.

So what did we learn?

Firstly, based on anecdotes, many users seem to like the improvements,
Though it can be hard for us to get an accurate picture.
Let’s do a quick poll.
Raise your hand if you have noticed the modernize feature in your editor, or used the modernize command.
ok now keep your hand raised only if you found it useful.

The second thing we noticed is that even simple modernizers have a lot of bugs.



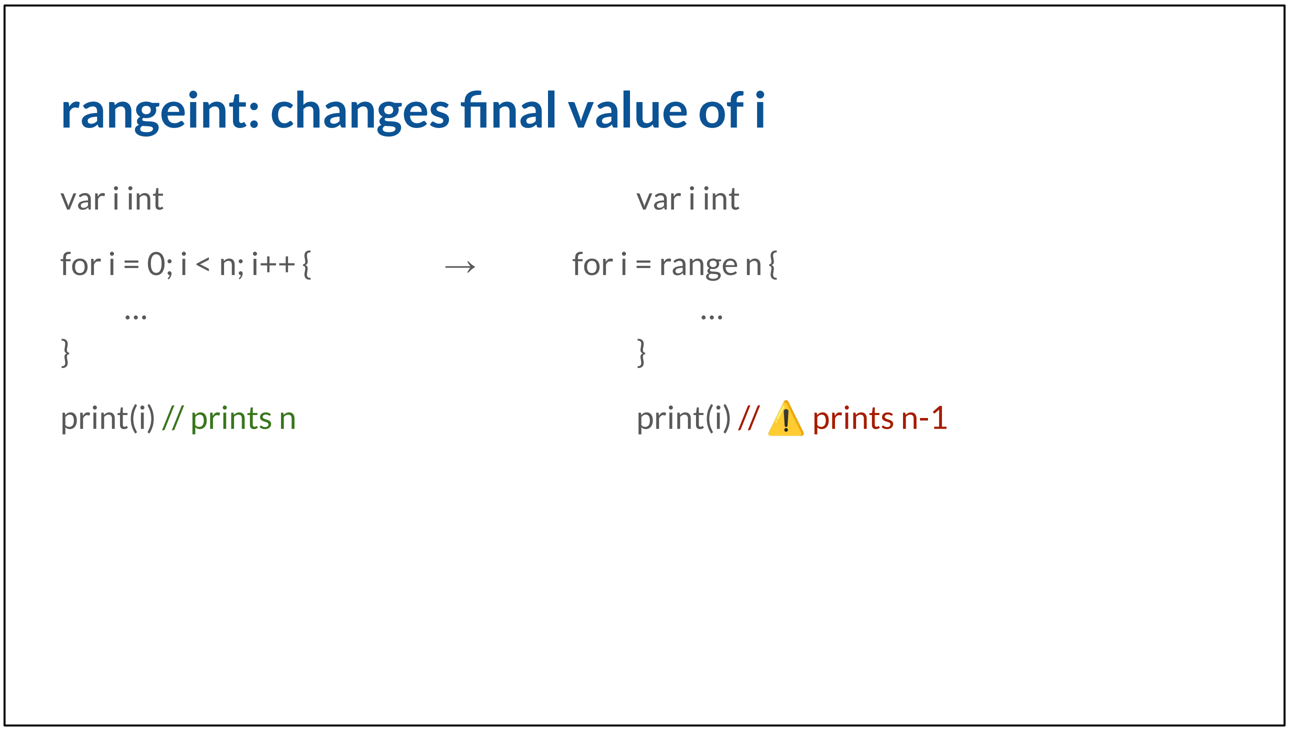
rangeint

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


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

Let’s take a look at just one of the modernizers. It’s one I wrote, called rangeint.
We saw it earlier: it replaces a 3-clause for loop with “for range n” using the new feature that AI confidently claims is not in go1.22.

It also gets rid of the loop index (indeed, must get rid of it) if it isn’t used by the body.
Pretty simple, right?
Wrong!

I’m going to quickly show you four actual bugs in just one of our twenty modernizers.



rangeint: changes final value of i

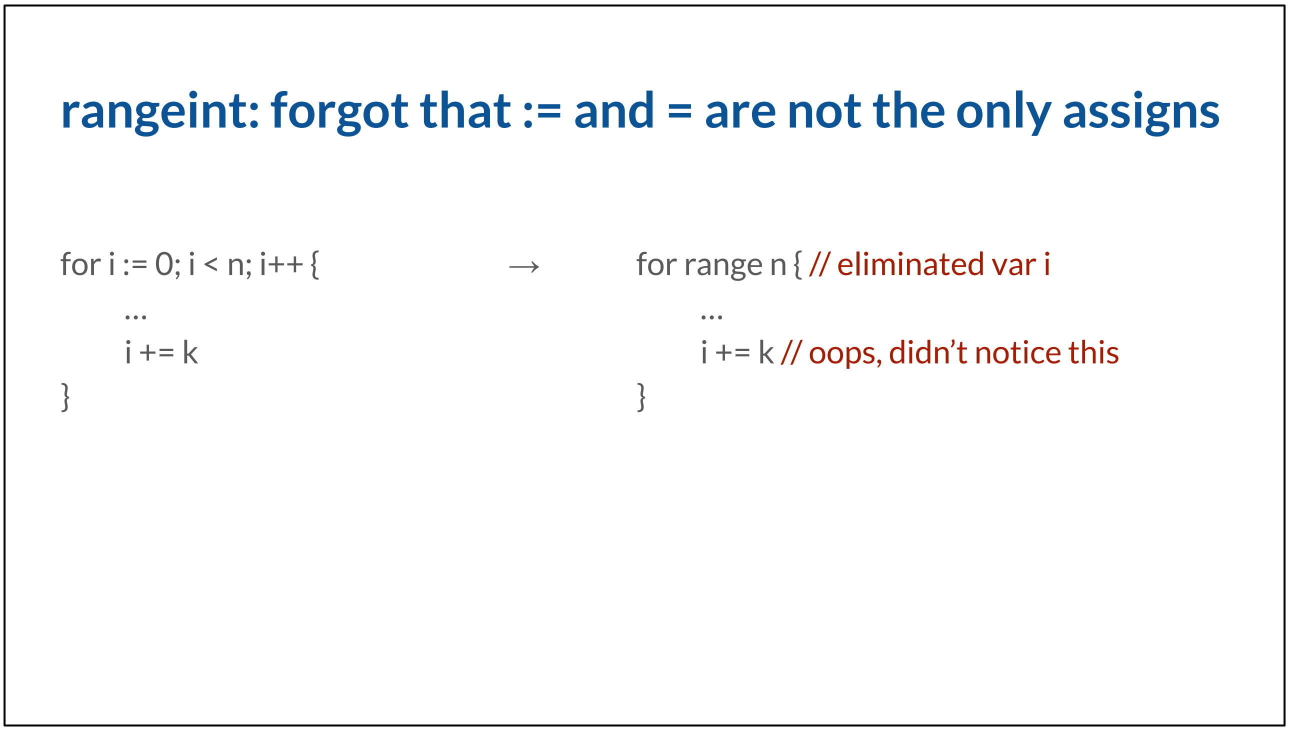
```
var i int                             var i int

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

print(i) // prints n                  print(i) // ⚠️ prints n-1
```

Here’s the first one, and it’s very subtle.

A range loop needn’t declare the index variable: it can update an existing variable, such as i here.

However, when i is declared before the loop and used after the loop, the transformation changes the final value of i, introducing a serious latent bug.

Fortunately a single mysterious test failure in what seemed like a large but straightforward cleanup helped us catch this bug before it did us any harm.



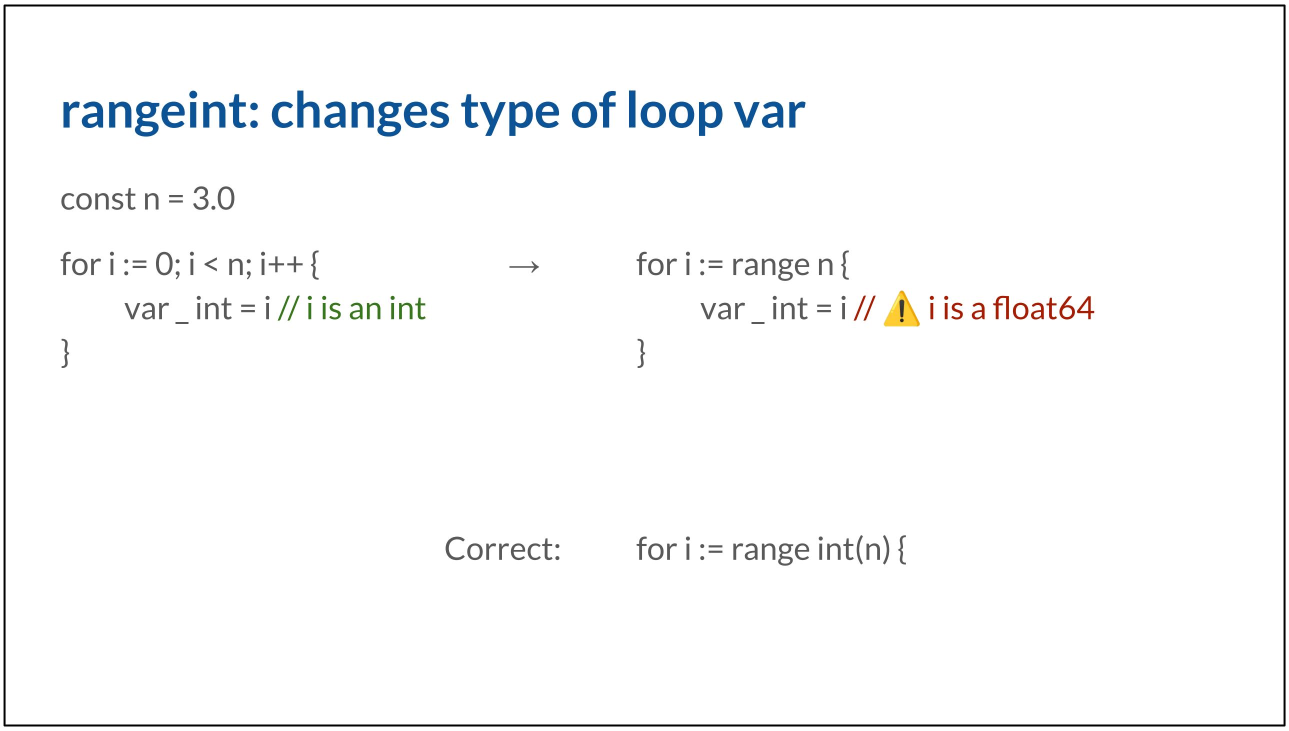
rangeint: forgot that := and = are not the only assigns

```
for i := 0; i < n; i++ {        →        for range n { // eliminated var i
        ...                                      ...
        i += k                                   i += k // oops, didn't notice this
}                                        }
```

This second one is a rather shallow bug.

When I added logic to fix the previous bug, I forgot that there are more kinds of assignment than := and =.
So the i += k statement slipped under the radar, and caused it messed up the code.

At least this time we get a compile error.



rangeint: changes type of loop var

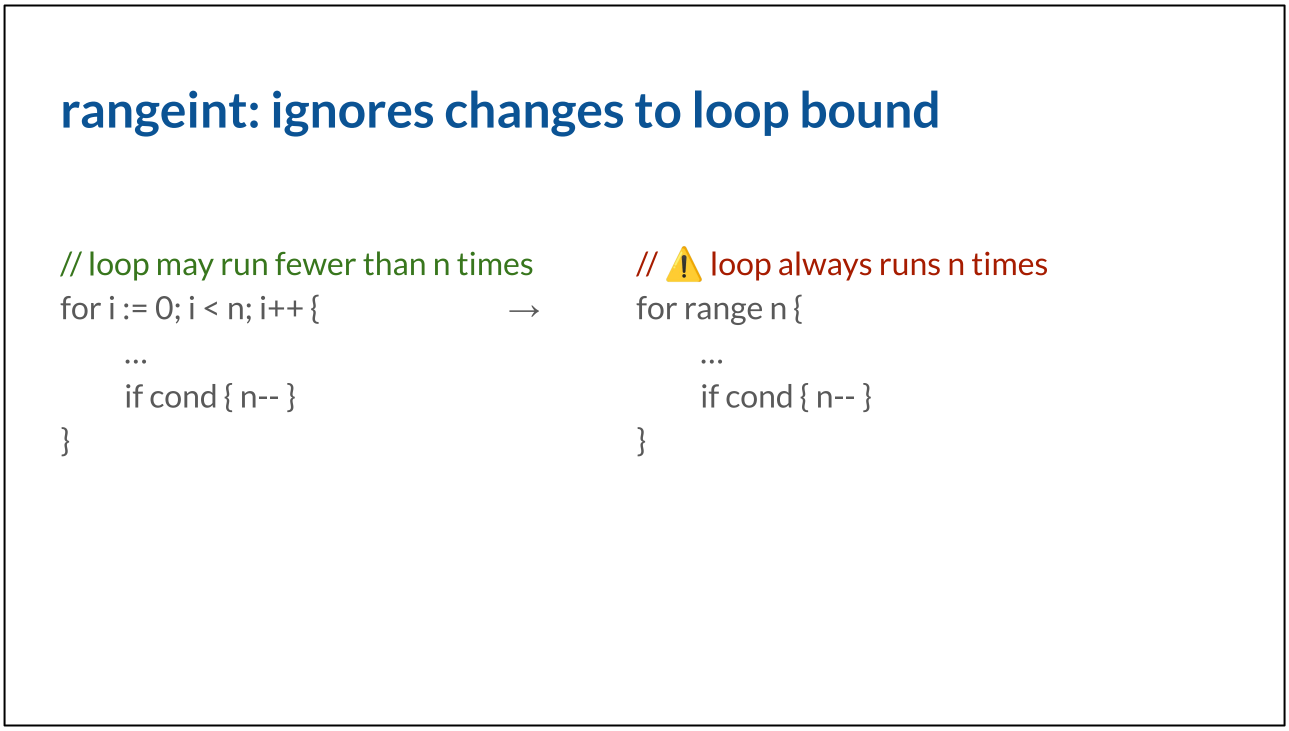
```
const n = 3.0

for i := 0; i < n; i++ {            →        for i := range n {
        var _ int = i // i is an int                 var _ int = i // ⚠️ i is a float64
}                                            }


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

The third bug is quite subtle.

In the loop on the left, the type of i comes from zero, so it's int.
The loop's bound is a float constant, but it's ok to compare it with int it since its value is an integer.

But in the transformed code, the type of i comes from the constant n, so it's a float.
We've changed the type of i within the loop.
If you're lucky, as in this case, it causes a compile error, so at least you can't overlook it.
But if you're unlucky, it introduces a latent bug into your code.

The fix is to introduce an explicit conversion when n doesn't have the same type as i.



rangeint: ignores changes to loop bound

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

And here’s the fourth bug:

The loop on the left is intended to change its own limit n as it runs,
so it may do fewer iterations than the original value of n.

The transformed version reads the value of n once, and then ignores any changes to it.
This is usually a subtle latent bug.

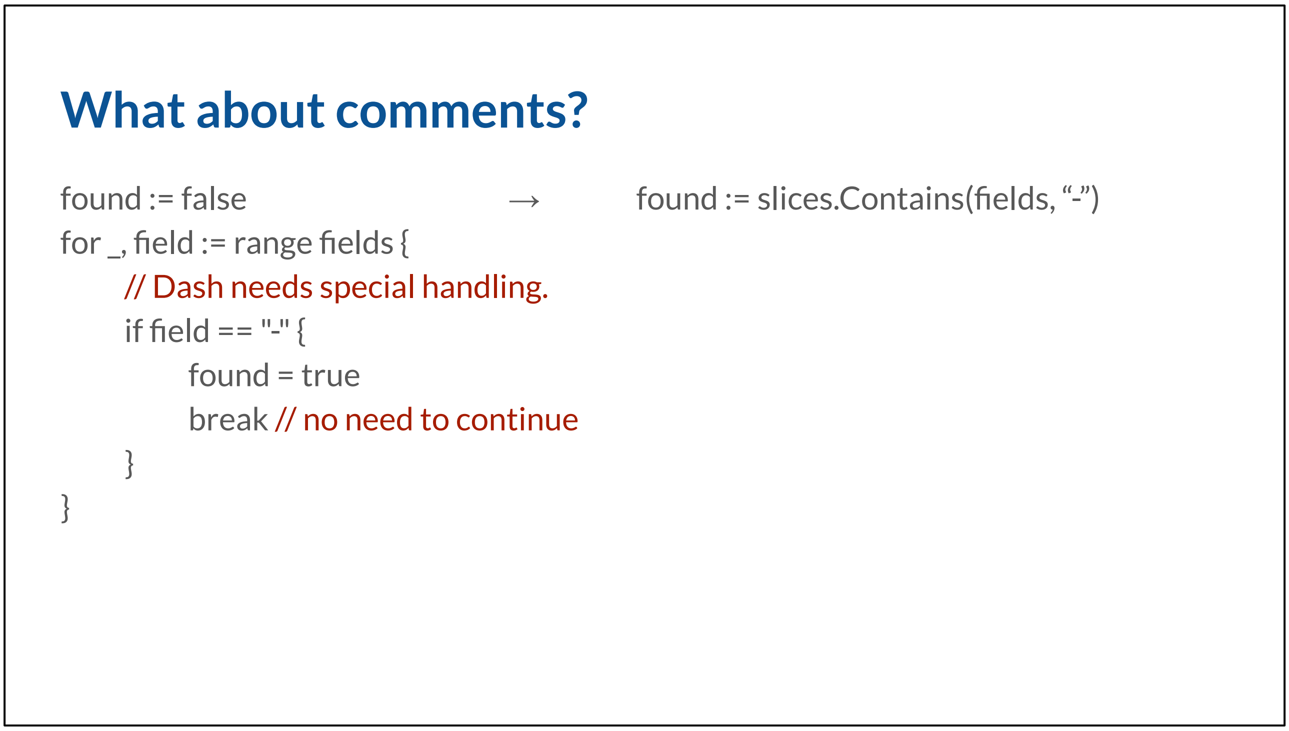
The fix is to skip the transformation if the loop body might have side effects on the limit expression n.
And that’s actually a fairly involved computation.

--

So that’s the four bugs. We actually found seven in just this one modernizer, and it’s not even a particularly complex one.

The point of all this, the reason I’m talking so much about bugs, is that it is really quite hard to get the logic right.

Better infrastructure could certainly help, and better support for testing,
but you really have to think like a compiler writer and consider all the edge cases exhaustively.



What about comments?

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

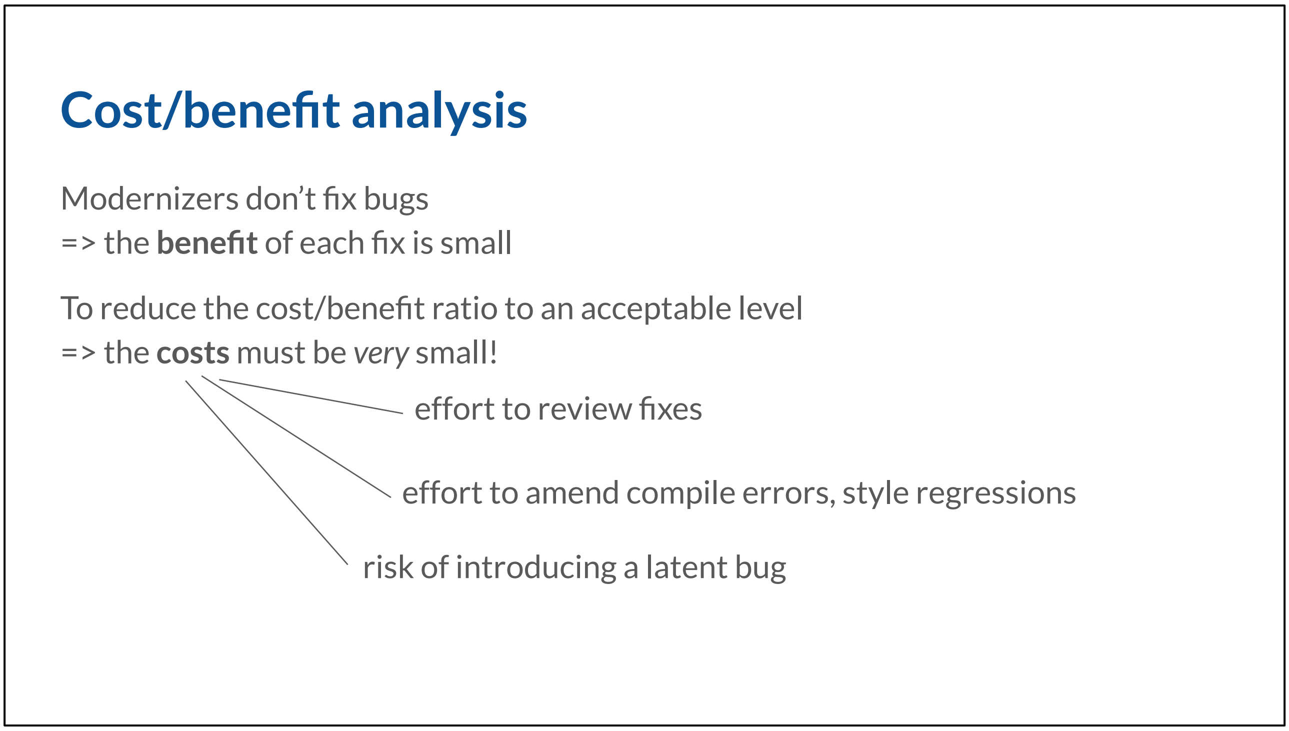
And that’s just the *correctness* of the syntax manipulation.

Source code is read by humans as well as computers, so unlike the output of a compiler, the output of a moderniser has to *look good* too.
Comment handling has always been a weak part of our refactoring tools---something we plan to fix.

But even then there are always judgement calls.

Does a comment refer to the previous line or the following line?
Should we delete a comment in a loop when we modernize away the loop? Or keep it? It depends.
In this example, the first comment is important and should be kept, but the second is trivial and should be deleted.

There’s really no good way to tell. (Idea: LLMs actually make very good guesses.)



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!

effort to review fixes

effort to amend compile errors, style regressions

risk of introducing a latent bug

But here's the thing:

- Since modernizers aren't fixing bugs, the benefit of each fix is pretty small.
- That means the costs -- the risks of introducing a bug, or the effort required for review -- must be very small.

Otherwise it's just not worth it.
Most users are going to apply all these changes without really looking for bugs.

I had initially been quite confident about modernizers, and even the prospect of "democratizing" the process of adding them so that
anyone would be able to define a modernizer (or other analyzer) for their own APIs and our framework would magically just pick it them up dynamically (and securely) run them.

But to me the lesson of this experiment is that direct syntax manipulation takes a lot of effort to get right, even for experienced users.

(BTW, it is not lost on me that while I'm fussing over correctness, the industry is diving headlong into vibe coding. Sometimes I do feel like a dinosaur.)

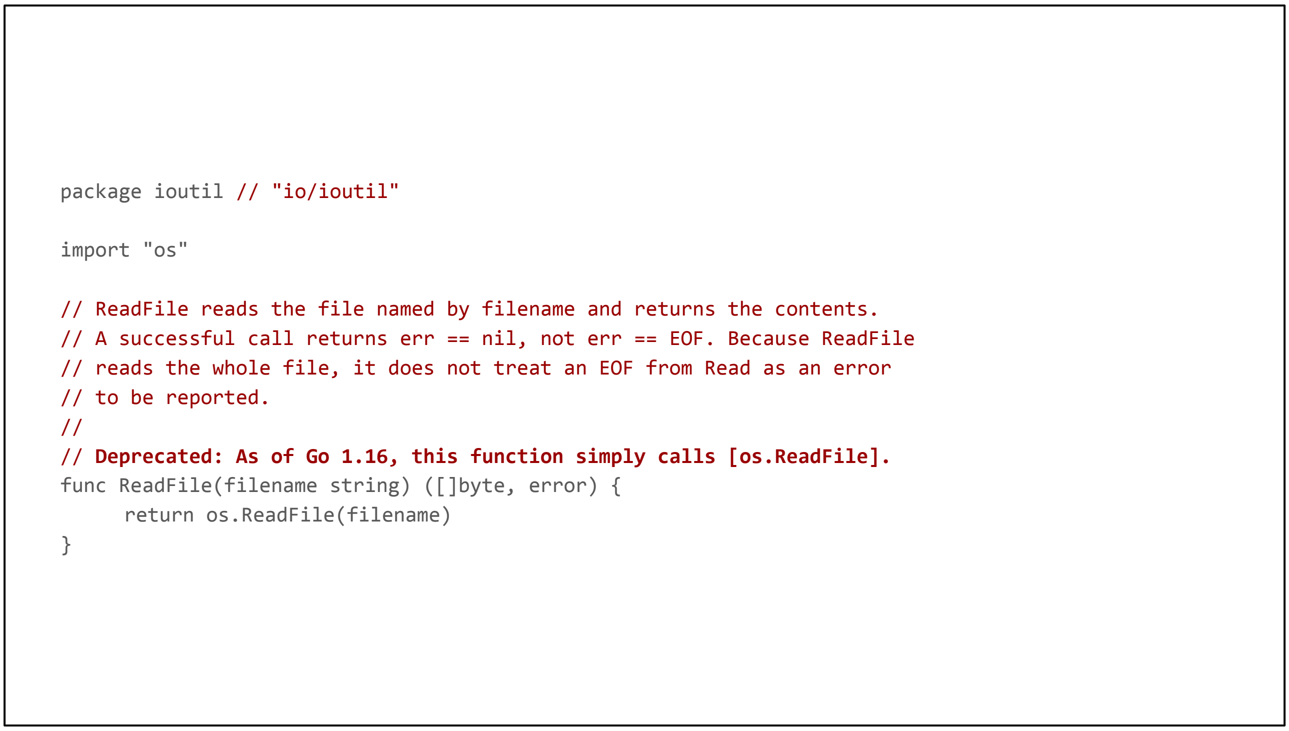
How can we make modernization safer?
How can we make it self-service?

So this work on modernizers leads us to two questions.

First, it’s too easy to make mistakes that break the build or, much worse, introduce subtle bugs.
How can we make these code transformations safer?

Second, all these modernizers are “bespoke” algorithms tailored to specific features.
That’s fine for the core language and standard library, which are used by everyone.
But how can we enable you to modernizer your own code base?

I’m going to spend the remaining time of this talk discussing a different approach to modernizing that is inherently safer, and is 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)
}
```

Let’s recall our readfile example from earlier.

Notice that we keep the old function around, but its implementation just the calls the new function.

In effect, we’re trying to have two names for the same function, and we express this as a wrapper function.

```
// 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)
}
```

There are many variations on this theme.

For example, let’s say your old function declares its parameters in a different order from the new function. Like Sub here.

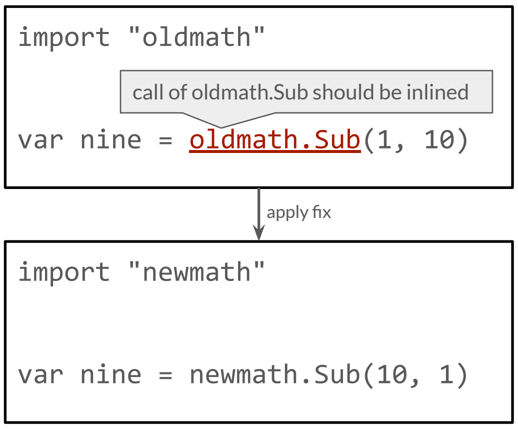
Or the new function has more parameters and the old function passes a constant to one of them. LIke Inf here.

Or perhaps the old function is unnecessary since it’s redundant with some other function in the new API. Like Neg here.

In all of these cases, the desired behavior is that the user stops calling the deprecated function, and instead
does whatever the body of the deprecated function does.

In other words, the user should *inline* the function call.

```go
// 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)
}
```


```go
import "oldmath"

var nine = oldmath.Sub(1, 10)
```

call of oldmath.Sub should be inlined

```go
import "newmath"


var nine = newmath.Sub(10, 1)
```

So, we built a tool that does this. To use it, you simply add a special comment, //go:fix inline, to the old function.

Then, you'll immediately notice in your editor (top box) that each call to the old function will trigger a diagnostic
saying that calls to oldmath.Sub should be inlined. The diagnostic carries a fix that actually inlines the call.

If you apply the fix, you end up with the code below.

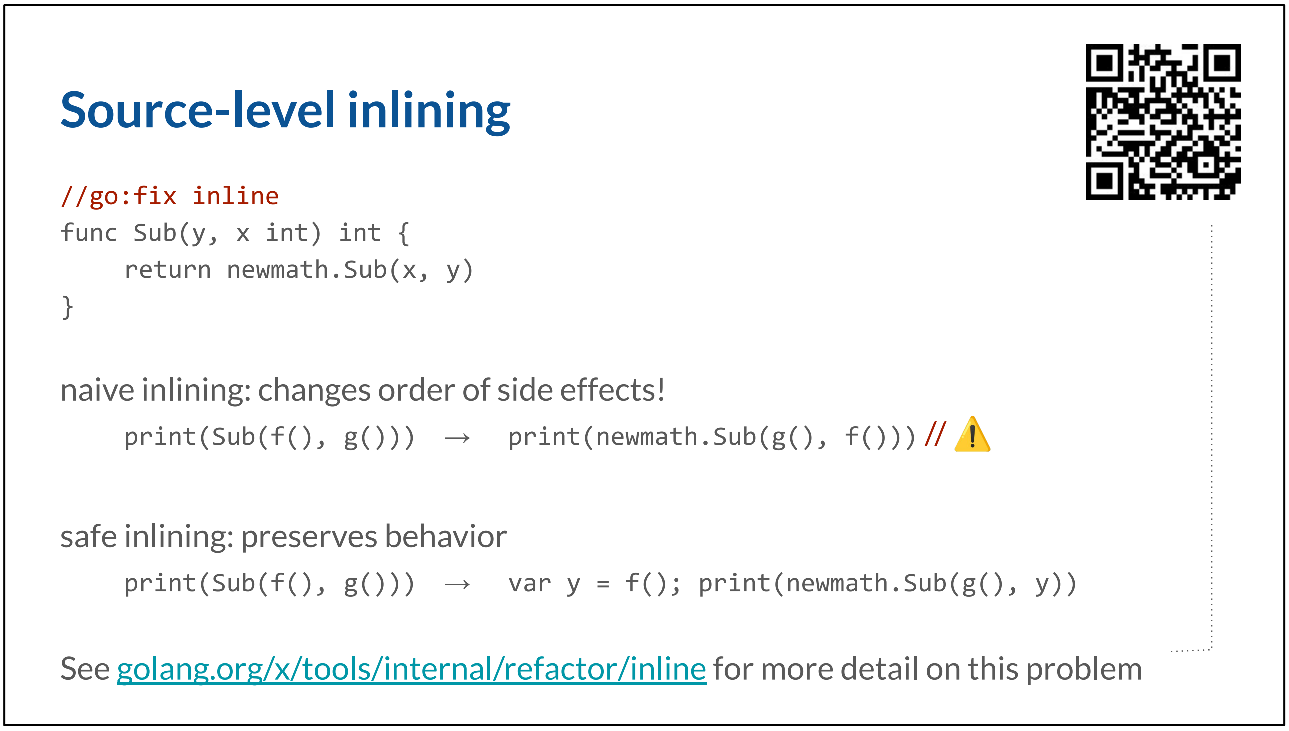
The use of the old package has disappeared, and the arguments to the call are now in the logical order.

You can use this technique to do some quite large cleanups in your own codebases, in three steps.

1. first you build the new API you want, and express the old API in terms of it. like oldmath.
2. then you add the go fix inline directives, and run the tool to clean up all the calls across your repo.
3. then you delete the old API.

Of course it's not always so simple, but with a little creativity you can express a lot of migrations this way.

Google has a robot that automatically inlines all calls to annotated functions, and sends out CLs to the code owners.
So you can just add an annotation and let elves magically clean up calls to the old code during the night.
[white lie: this is for C++ and Java but not yet Go.]



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

So how does this work?

It's based on a source-level inliner, which is a rather complex algorithm.

The key insight is that all the semantic complexity and edge cases are dealt with systematically
in the inliner so that there's no way for a gofix inline annotation to change the behavior of the program.

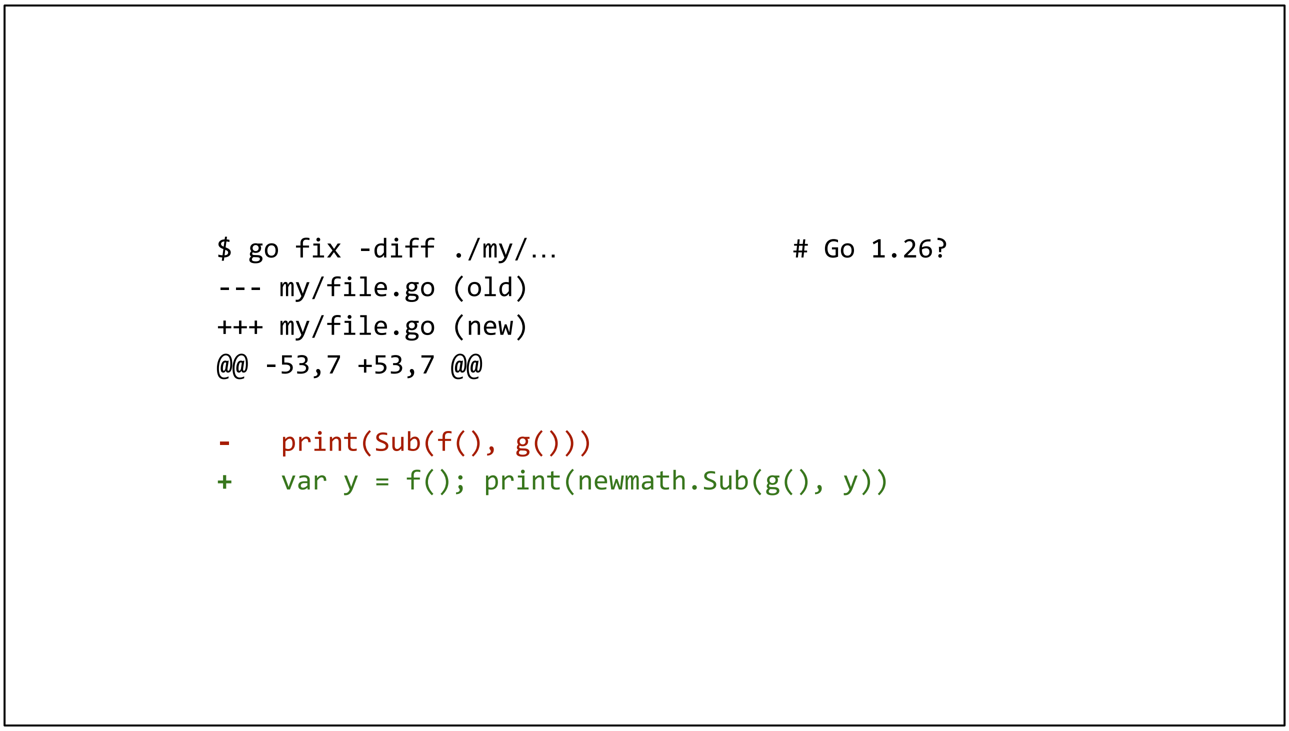
For example, the naive way to inline a call to the Sub function here is to simply replace y with the f() call and x with the g() call.
But that can introduce subtle bugs if f and g have side effects, because it causes the g call to occur first.

Now it's bad practice to rely on the order of effects within a single expression,
but what we learned with modernizers is that the tool must be as correct as a compiler
if we want people to blindly trust the integrity of their source to it.

So, the safe inlining of this call is to insert a declaration of a new variable to hold the result of calling f before we evaluate g.
The source changes can be quite fiddly in some cases.

Order of effects is just one of a dozen very subtle considerations. We don't have time to even name the others.

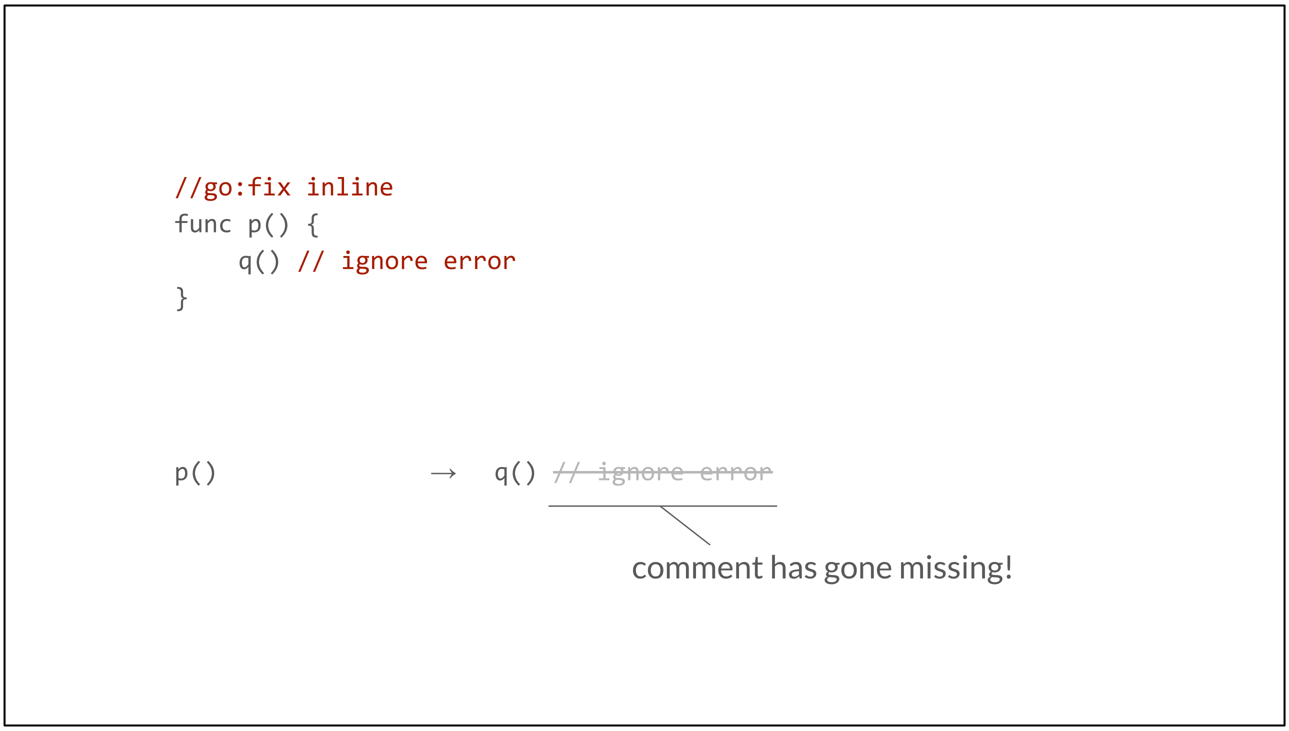
The inliner is about 7000 lines of dense code.
I won’t go into the details -- I have a whole talk on that topic for another day
[Source-level inlining]
but you can follow this link if you want to get a sense of what is involved.



```
$ 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))
```

This go fix feature is already deployed for interactive IDE use in gopls.
Please try it out in your own codebase.

For go1.26, we're hoping to add this functionality to the go fix command, making it easy to apply safe fixes en masse.
(The go fix subcommand already exists, but currently runs a handful of obsolete cleanups from Go's earlier days.)



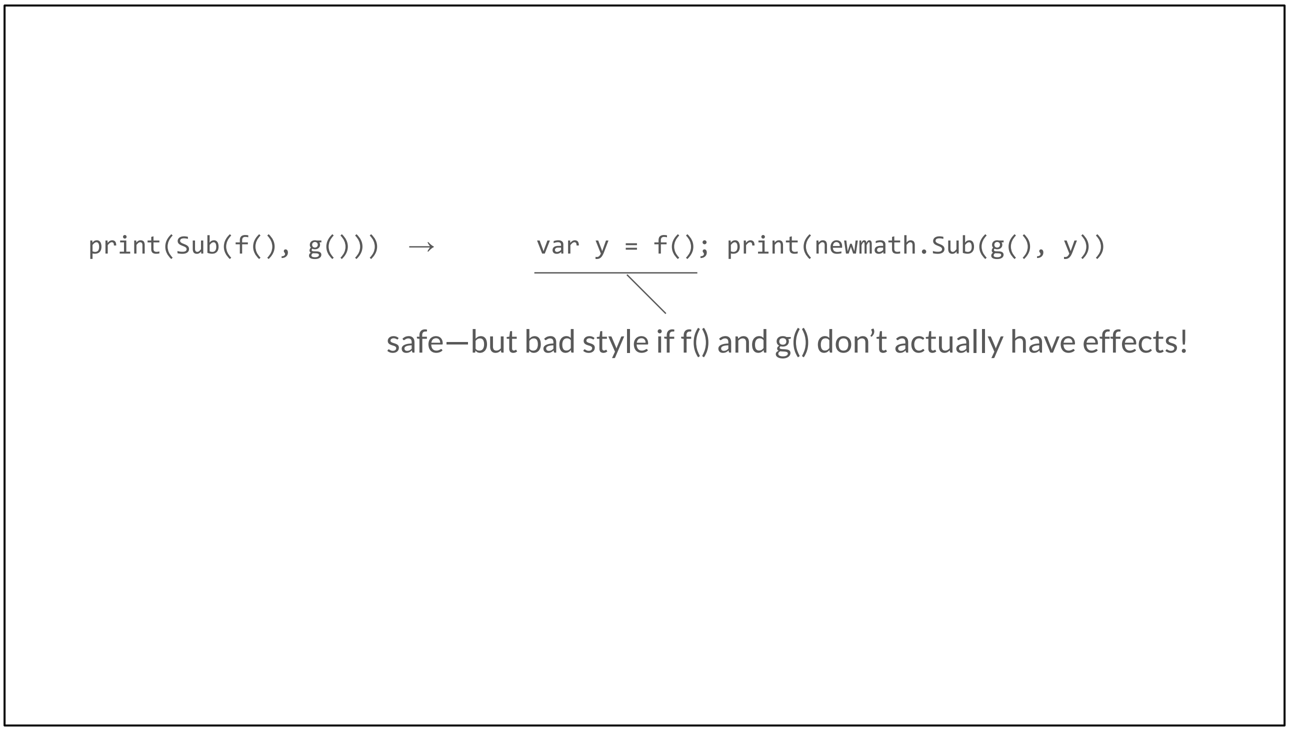
```go
//go:fix inline
func p() {
    q() // ignore error
}
```

p() → q() ~~// ignore error~~

comment has gone missing!

I’m going to briefly mention a couple of areas of ongoing work on the inliner.

The first is comment handling.
Comments are not recorded as part of Go’s syntax tree.
Consequently, the inliner, like many tools that move subtrees around, tend to misbehave with comments,
either putting them in wrong place or dropping them entirely.
We have some ideas for a systematic fix to this class of problems, but it’s a project.

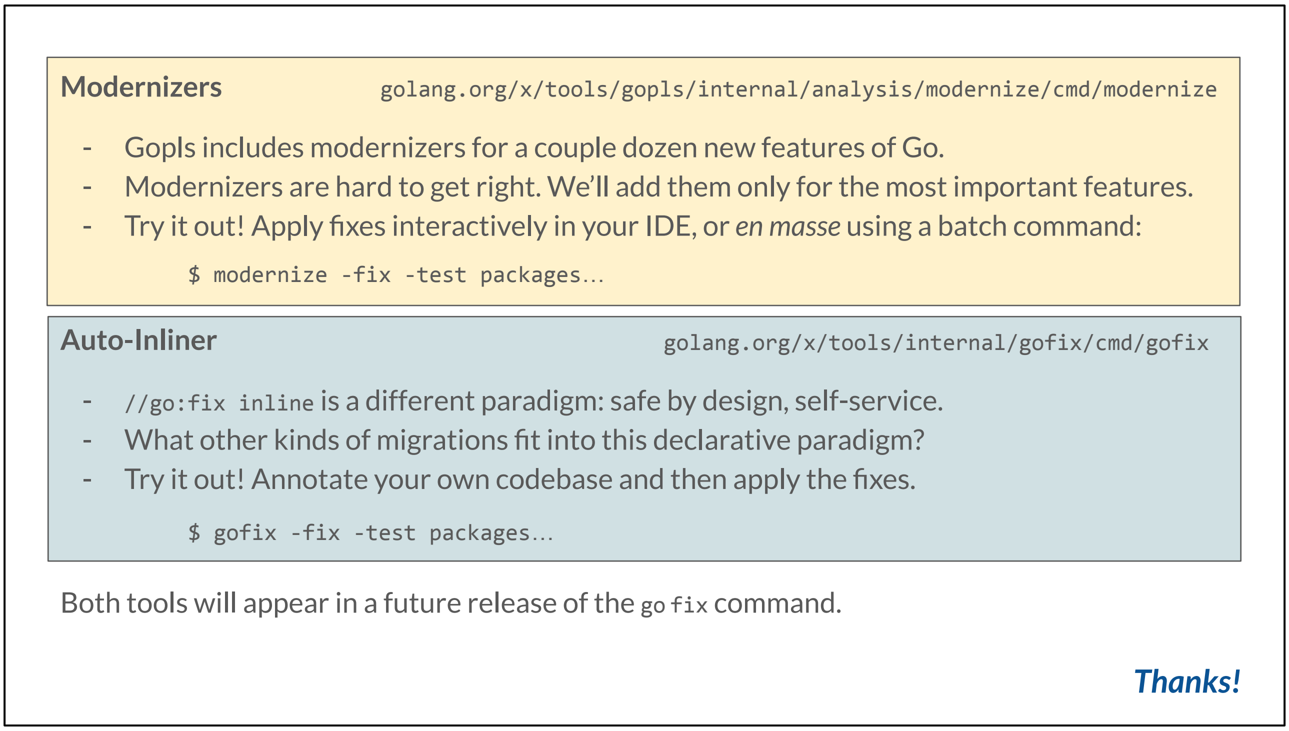


```
print(Sub(f(), g()))  →       var y = f(); print(newmath.Sub(g(), y))
```

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

The second is the problem of being “too safe”, which is a nice way of saying unnecessarily pedantic.
In the example of newmath.Sub, if you know that functions f and g have no side effects, or at least no effect on each other, then the transformation to add var y is just bad style, and you’ll want to clean it up by hand.

We’re working to make the inliner smarter in many ways to avoid bad style, but it’s a bit like building an optimizing compiler: the job is never finished.



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!

Let's wrap up.

I've presented two different kinds of modernization tools.
First modernizers, which are bespoke algorithms tailored to specific new features of the language and standard library.
They are fiddly, so we plan to add them only for the most important new features.

Second, the auto-inliner, a more general tool driven by directive comments that you can add to your own codebase.
This approach is safer, and open ended, and seems promising for other kinds of migrations.
Please share any ideas you have for other code transformations that could be supported in this way.

Both tools are already released in gopls for interactive IDE use.

The batch commands (modernize and gofix) are both temporary internal packages.
In due course, we'll move them into their proper home in the go fix subcommand.

Please try them out, and let us know what you think.

Many thanks!