Cognit-Go

Cognitive Complexity in Go

Scott Nicholas Allan Smith

London Gophers

2024-09-18

About Me

- Clojure + a bit of React/TS for the last year at Riverford Organic Farmers
- Go, Python, Java, React/TS for the previous two years at Open Cosmos
- Worked on microservices & monoliths, cloud & local & embedded applications
- Growing along the path to becoming a tech lead or staff/principle engineer
- Aspiring to give more technical & conference talks
- For fun: kayaking, woodworking, reading, learning languages, eating/cooking food

Where Are We Going?

Aims

- Introduce some micro (function) level complexity measures
- Calculate these on some Go code snippets for comparison
- Discuss linting for them in fresh or not-so-fresh codebases
- Explain why cognitive complexity can be a good tool in your toolbox

Where Are We Going?

Non-Aims

- Have everyone linting for cognitive complexity tomorrow in their codebases
- Claim to solve all of your code complexity problems with this one weird trick

A Taxonomy of Complexities *

* Less pretentiously: types of complexity **

** or not even really complexity https://www.youtube.com/watch?v=vzfy4EKwG_Y

A Taxonomy of Complexities

Macro level

Essential - complexity that is inherent in the domain that you're working on

Something that the business has decided must happen because it's "valuable"

e.g. we deliver to a certain customer on a given weekday and never to any customer on a Sunday, except for near Christmas where the weekday is moved around so each customer gets their delivery, as close to Christmas as possible including potentially on Sundays

A Taxonomy of Complexities

Macro level

Incidental - complexity introduced due to e.g. technological/code choices

Not a business rule/doesn't add value
Must happen because of some technology choice (good or bad)

e.g. we're a team of 5 engineers with 10000 daily users

To know the customers delivery date you have to call out to the
delivery date microservice who is the only one who knows delivery date logic

We look after 10 microservices each

Basically the classic Krazam https://www.youtube.com/watch?v=y8OnoxKotPQ

A Taxonomy of Complexities

At a *micro* (code/per function) *level*

- **Cyclomatic**
- **Cognitive**

both measure complexity on the scale of individual functions (or code snippets)

Cyclomatic Complexity

A mathematical concept

Formally, the number of linearly independent paths through a piece of code

More informally, counts the "decision points" in a block of code

Consequently ~ number of tests you would have to write to cover all paths in a function

We'll see some examples of how to calculate this in a bit

Cognitive Complexity

From G. Ann Campbell at SonarSource - corporate, so make of that what you will *

Tries to account for intuition about "hard to understand" code in its rules

(i.e. more vibes based than formal)

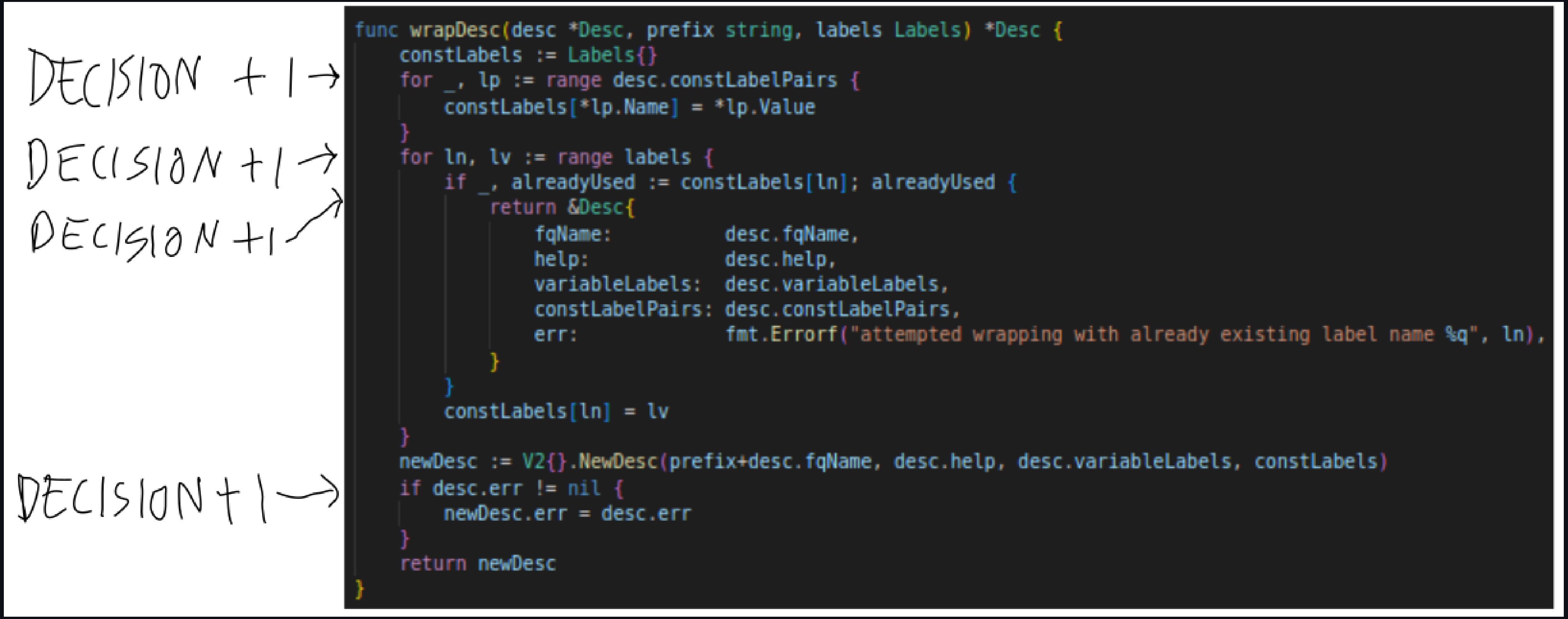
Key departures from cyclomatic complexity

- switch/case +1 to complexity, regardless of cases, rather than + number of cases
- nested "decisions" count progressively more towards complexity
- boolean operations are counted "per group" rather than "per operation"
- +1 to complexity on recursion

This will become much clearer with the following examples

* full white paper here https://www.sonarsource.com/docs/CognitiveComplexity.pdf

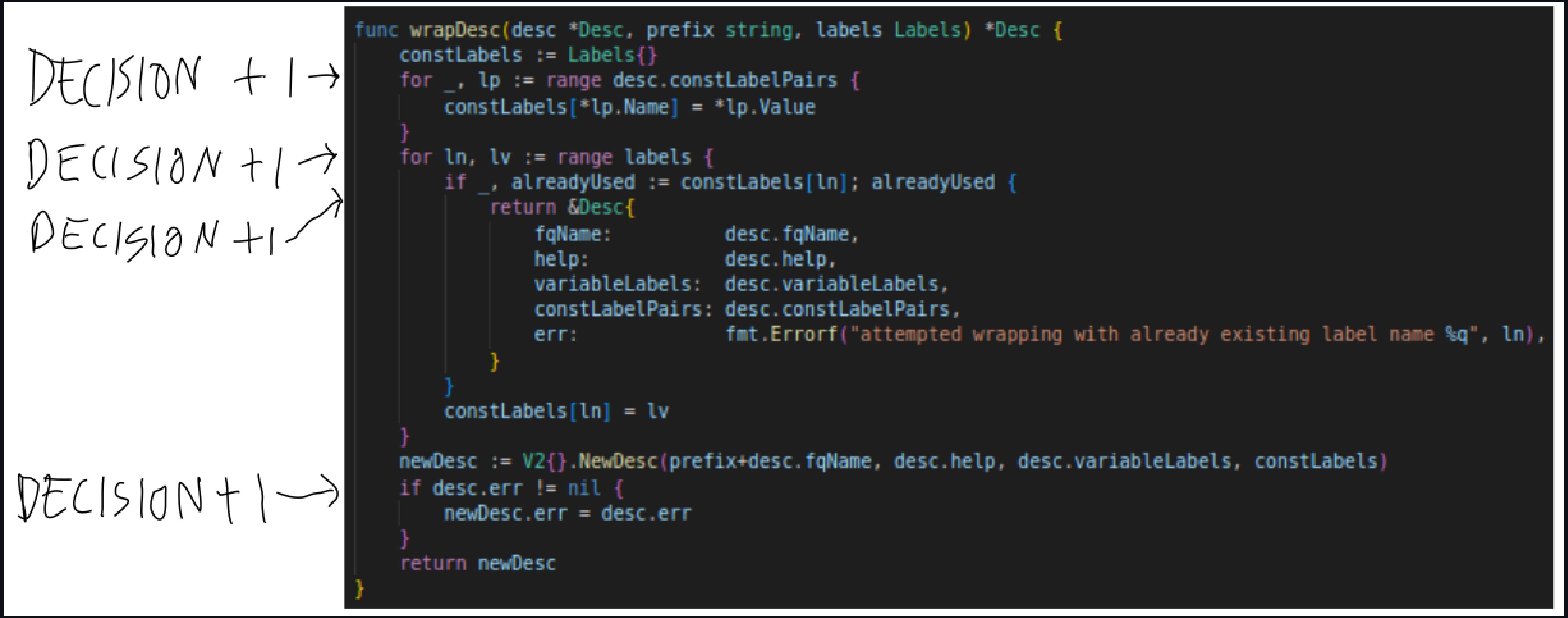
Calculating Cyclomatic Complexity of Go Code



```go
func wrapDesc(desc *Desc, prefix string, labels Labels) *Desc {
    constLabels := Labels{}
    for _, lp := range desc.constLabelPairs {
        constLabels[*lp.Name] = *lp.Value
    }
    for ln, lv := range labels {
        if _, alreadyUsed := constLabels[ln]; alreadyUsed {
            return &Desc{
                fqName:          desc.fqName,
                help:            desc.help,
                variableLabels:  desc.variableLabels,
                constLabelPairs: desc.constLabelPairs,
                err:             fmt.Errorf("attempted wrapping with already existing label name %q", ln),
            }
        }
        constLabels[ln] = lv
    }
    newDesc := V2{}.NewDesc(prefix+desc.fqName, desc.help, desc.variableLabels, constLabels)
    if desc.err != nil {
        newDesc.err = desc.err
    }
    return newDesc
}
```

All code shamelessly stolen & possibly slightly adapted from Prometheus https://github.com/prometheus unless stated otherwisse

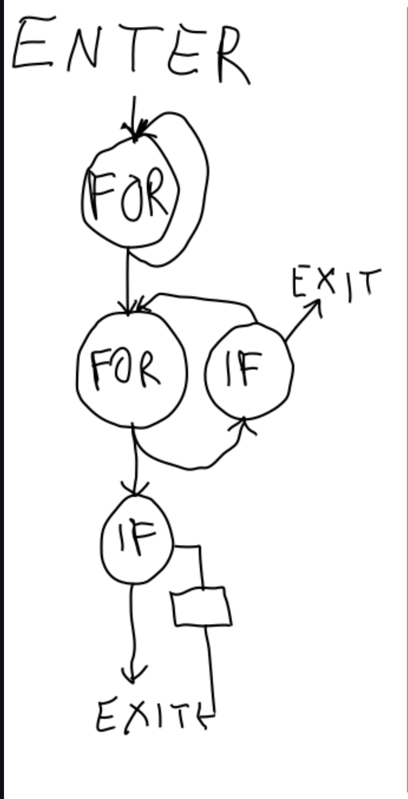
Calculating Cyclomatic Complexity of Go Code

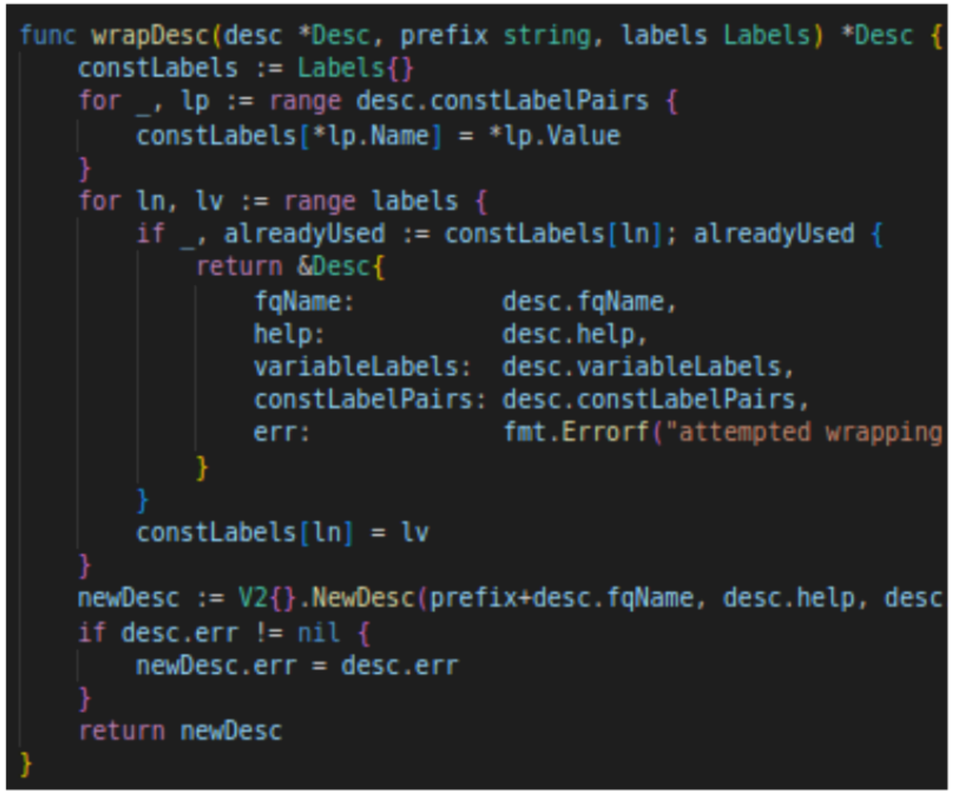


```go
func wrapDesc(desc *Desc, prefix string, labels Labels) *Desc {
    constLabels := Labels{}
    for _, lp := range desc.constLabelPairs {
        constLabels[*lp.Name] = *lp.Value
    }
    for ln, lv := range labels {
        if _, alreadyUsed := constLabels[ln]; alreadyUsed {
            return &Desc{
                fqName:          desc.fqName,
                help:            desc.help,
                variableLabels:  desc.variableLabels,
                constLabelPairs: desc.constLabelPairs,
                err:             fmt.Errorf("attempted wrapping with already existing label name %q", ln),
            }
        }
        constLabels[ln] = lv
    }
    newDesc := V2{}.NewDesc(prefix+desc.fqName, desc.help, desc.variableLabels, constLabels)
    if desc.err != nil {
        newDesc.err = desc.err
    }
    return newDesc
}
```

`for` implies an `if` - `if` no elements left to iterate over

Calculating Cyclomatic Complexity of Go Code

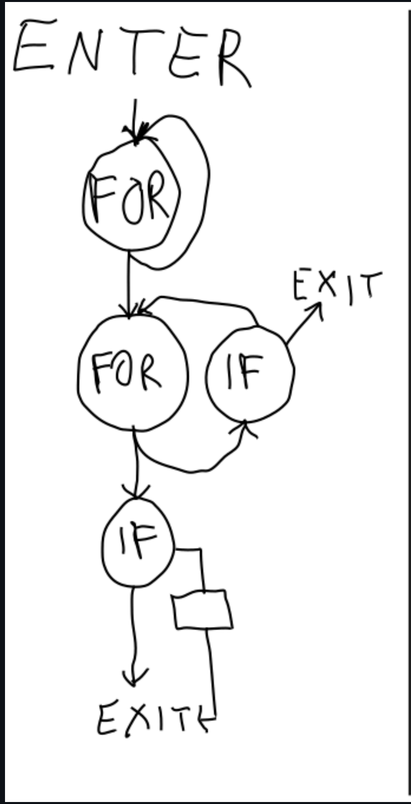


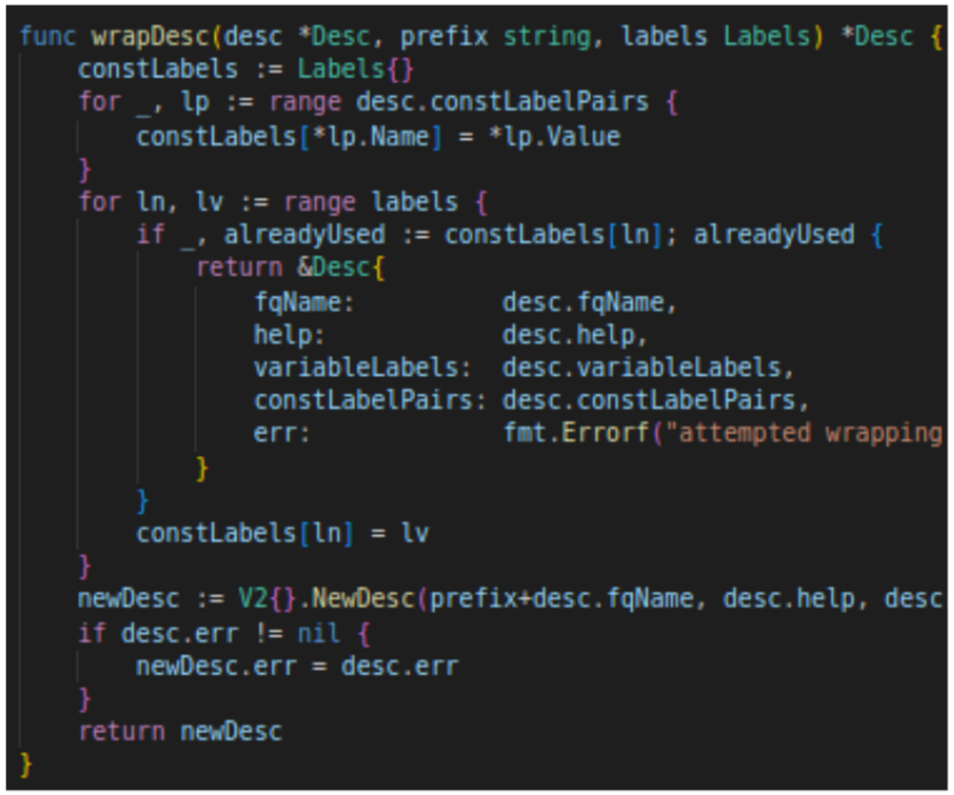


```
func wrapDesc(desc *Desc, prefix string, labels Labels) *Desc {
    constLabels := Labels{}
    for _, lp := range desc.constLabelPairs {
        constLabels[*lp.Name] = *lp.Value
    }
    for ln, lv := range labels {
        if _, alreadyUsed := constLabels[ln]; alreadyUsed {
            return &Desc{
                fqName:          desc.fqName,
                help:            desc.help,
                variableLabels:  desc.variableLabels,
                constLabelPairs: desc.constLabelPairs,
                err:             fmt.Errorf("attempted wrapping
            }
        }
        constLabels[ln] = lv
    }
    newDesc := V2{}.NewDesc(prefix+desc.fqName, desc.help, desc
    if desc.err != nil {
        newDesc.err = desc.err
    }
    return newDesc
}
```

Paths through the code

Calculating Cyclomatic Complexity of Go Code



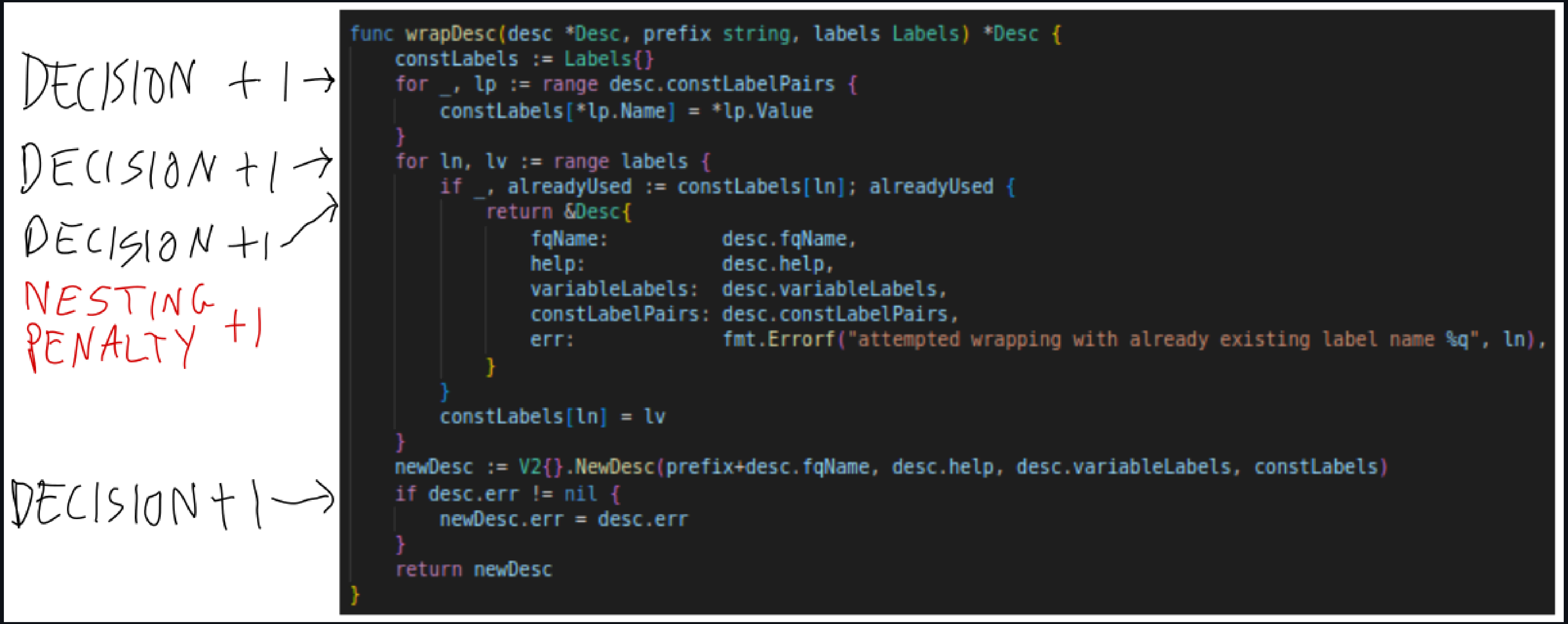


```go
func wrapDesc(desc *Desc, prefix string, labels Labels) *Desc {
    constLabels := Labels{}
    for _, lp := range desc.constLabelPairs {
        constLabels[*lp.Name] = *lp.Value
    }
    for ln, lv := range labels {
        if _, alreadyUsed := constLabels[ln]; alreadyUsed {
            return &Desc{
                fqName:          desc.fqName,
                help:            desc.help,
                variableLabels:  desc.variableLabels,
                constLabelPairs: desc.constLabelPairs,
                err:             fmt.Errorf("attempted wrapping
            }
        }
        constLabels[ln] = lv
    }
    newDesc := V2{}.NewDesc(prefix+desc.fqName, desc.help, desc
    if desc.err != nil {
        newDesc.err = desc.err
    }
    return newDesc
}
```

Linearly independent paths = # of decisions + 1 $\Longrightarrow$ 5

Calculating Cognitive Complexity of Go Code

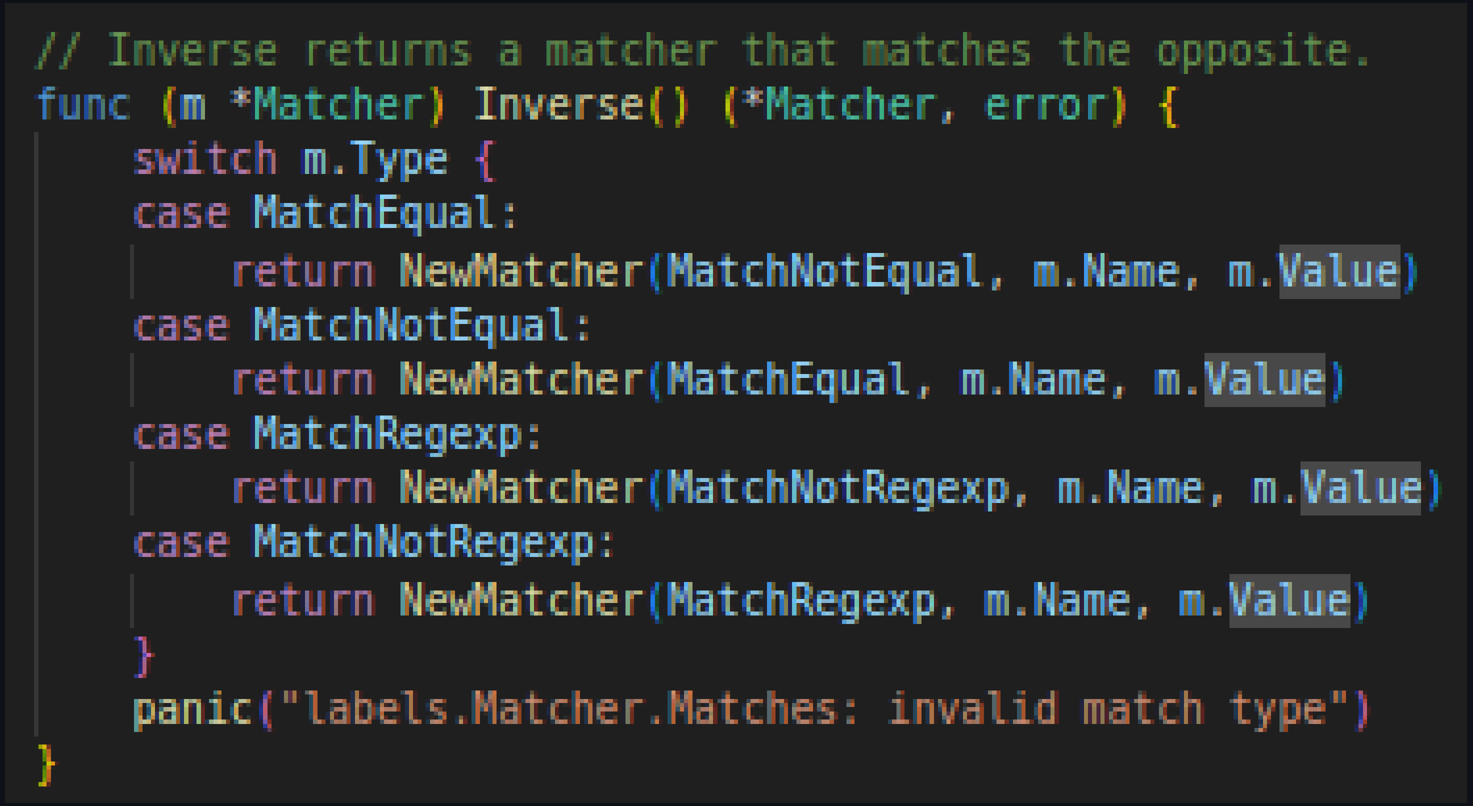
Basically, cyclomatic + adjustments for cognitive



```go
func wrapDesc(desc *Desc, prefix string, labels Labels) *Desc {
    constLabels := Labels{}
    for _, lp := range desc.constLabelPairs {
        constLabels[*lp.Name] = *lp.Value
    }
    for ln, lv := range labels {
        if _, alreadyUsed := constLabels[ln]; alreadyUsed {
            return &Desc{
                fqName:          desc.fqName,
                help:            desc.help,
                variableLabels:  desc.variableLabels,
                constLabelPairs: desc.constLabelPairs,
                err:             fmt.Errorf("attempted wrapping with already existing label name %q", ln),
            }
        }
        constLabels[ln] = lv
    }
    newDesc := V2{}.NewDesc(prefix+desc.fqName, desc.help, desc.variableLabels, constLabels)
    if desc.err != nil {
        newDesc.err = desc.err
    }
    return newDesc
}
```

5 paths + 1 nesting penalty ⟹ 6

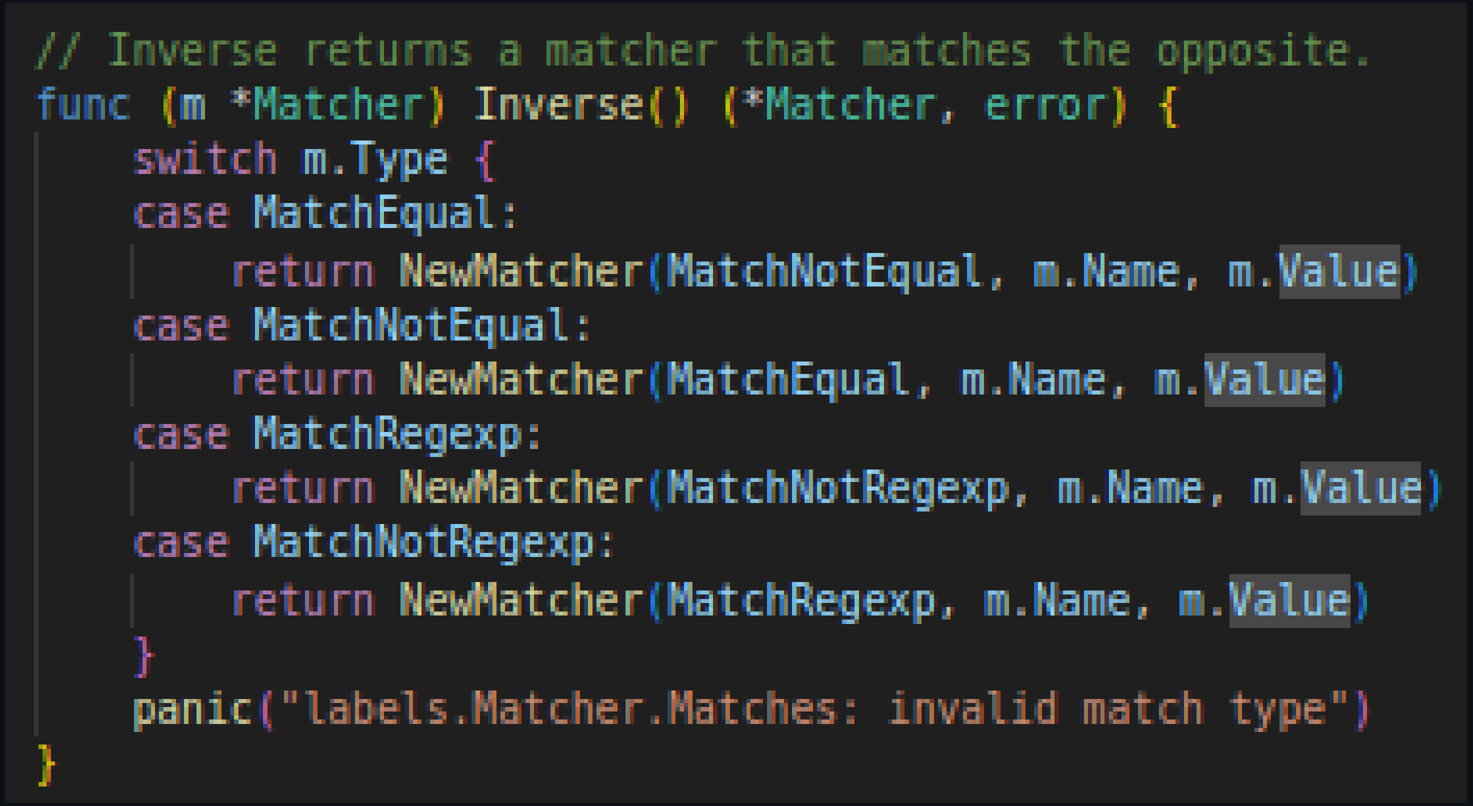
Cyclomatic vs Cognitive Complexity - Switch



```go
// Inverse returns a matcher that matches the opposite.
func (m *Matcher) Inverse() (*Matcher, error) {
    switch m.Type {
    case MatchEqual:
        return NewMatcher(MatchNotEqual, m.Name, m.Value)
    case MatchNotEqual:
        return NewMatcher(MatchEqual, m.Name, m.Value)
    case MatchRegexp:
        return NewMatcher(MatchNotRegexp, m.Name, m.Value)
    case MatchNotRegexp:
        return NewMatcher(MatchRegexp, m.Name, m.Value)
    }
    panic("labels.Matcher.Matches: invalid match type")
}
```

4 paths through switch + 1 path after $\Longrightarrow$ 5 cyclomatic complexity

Cyclomatic vs Cognitive Complexity - Switch

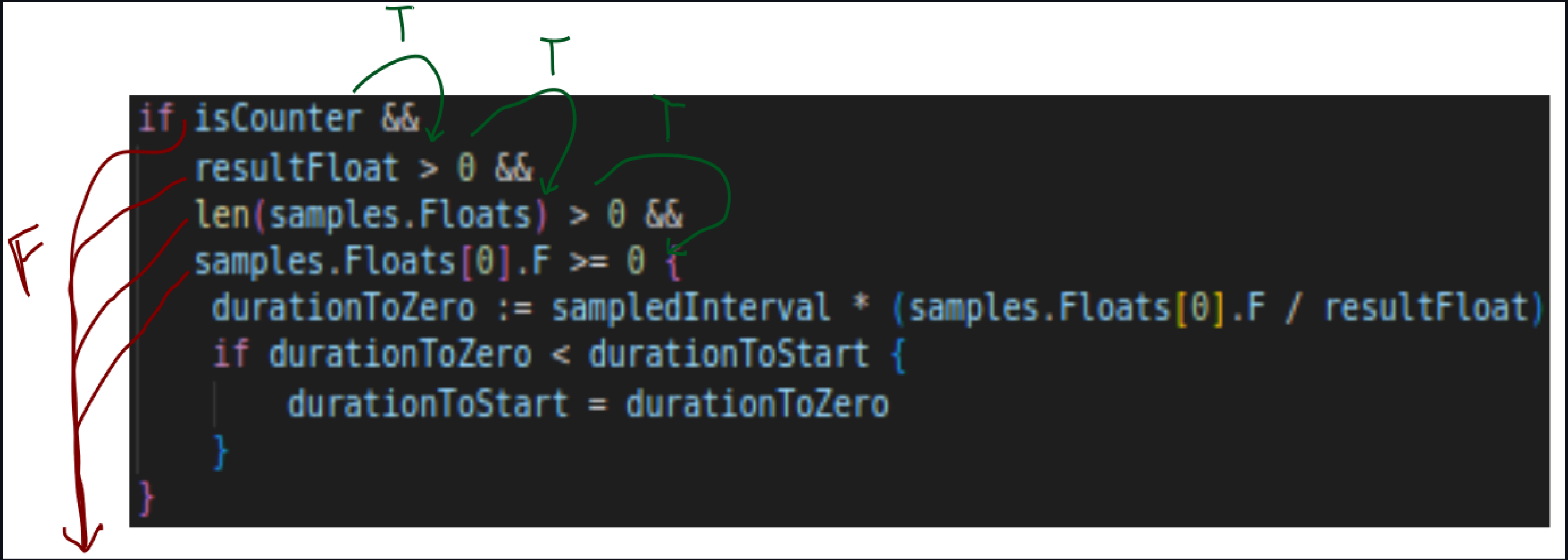


```go
// Inverse returns a matcher that matches the opposite.
func (m *Matcher) Inverse() (*Matcher, error) {
    switch m.Type {
    case MatchEqual:
        return NewMatcher(MatchNotEqual, m.Name, m.Value)
    case MatchNotEqual:
        return NewMatcher(MatchEqual, m.Name, m.Value)
    case MatchRegexp:
        return NewMatcher(MatchNotRegexp, m.Name, m.Value)
    case MatchNotRegexp:
        return NewMatcher(MatchRegexp, m.Name, m.Value)
    }
    panic("labels.Matcher.Matches: invalid match type")
}
```

switch always counts as +1 $\Longrightarrow$ ~~4~~ 1 + 1 $\Longrightarrow$ 2 cognitive complexity

Cyclomatic vs Cognitive Complexity - && ||

Can take a minute to grok path counting through boolean expressions

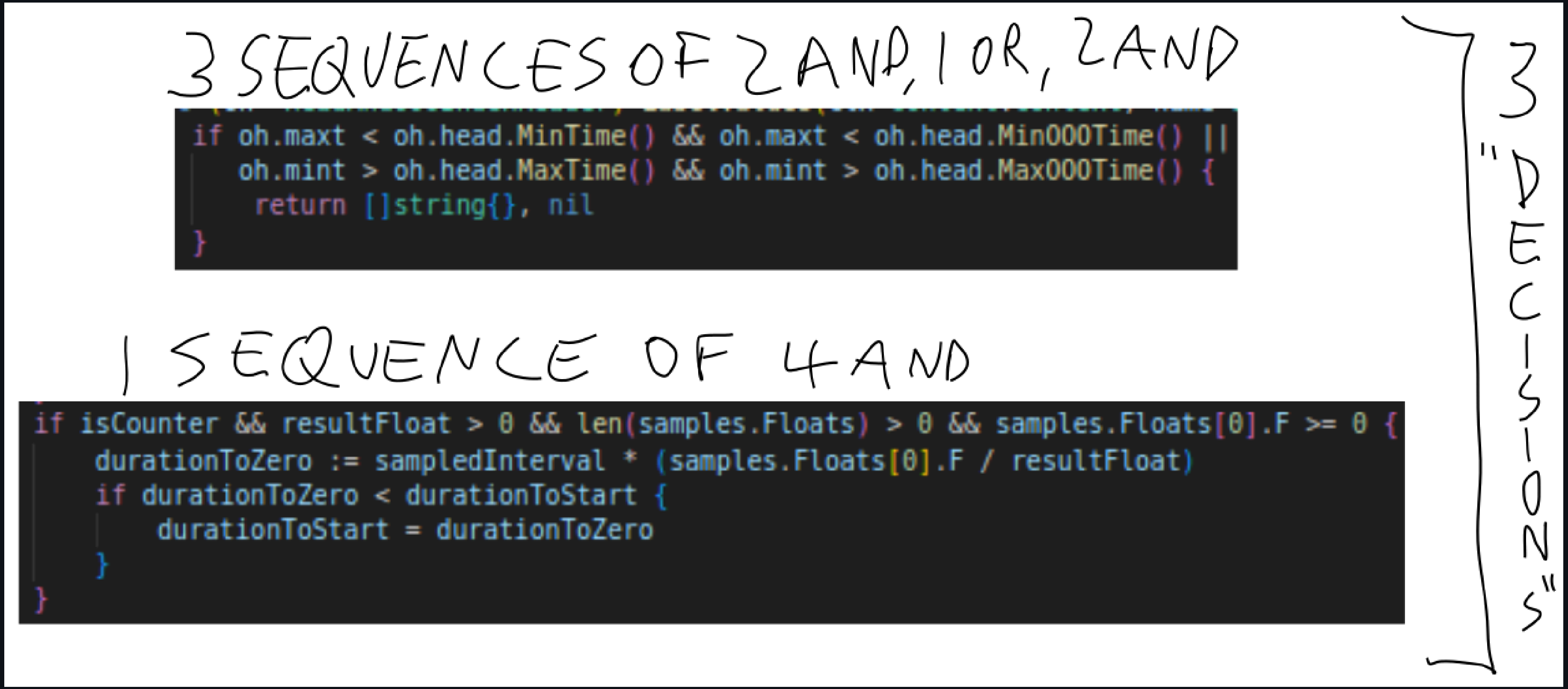


```
if isCounter &&
    resultFloat > 0 &&
    len(samples.Floats) > 0 &&
    samples.Floats[0].F >= 0 {
     durationToZero := sampledInterval * (samples.Floats[0].F / resultFloat)
     if durationToZero < durationToStart {
         durationToStart = durationToZero
     }
}
```

4 paths, 1 from if, 3 from logical operators $\Longrightarrow$ +4 cyclomatic complexity

Cyclomatic vs Cognitive Complexity - && ||

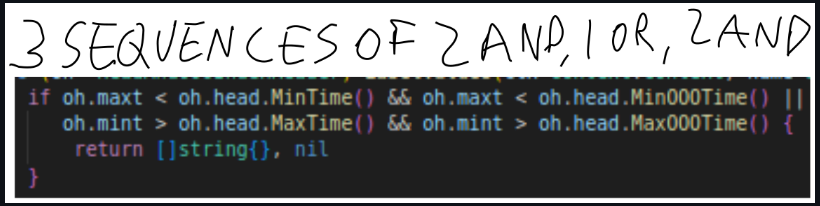
Does this align with our intuition about sequences of logical operations?



```
if oh.maxt < oh.head.MinTime() && oh.maxt < oh.head.MinOOOTime() ||
   oh.mint > oh.head.MaxTime() && oh.mint > oh.head.MaxOOOTime() {
    return []string{}, nil
}
```

```
if isCounter && resultFloat > 0 && len(samples.Floats) > 0 && samples.Floats[0].F >= 0 {
    durationToZero := sampledInterval * (samples.Floats[0].F / resultFloat)
    if durationToZero < durationToStart {
        durationToStart = durationToZero
    }
}
```

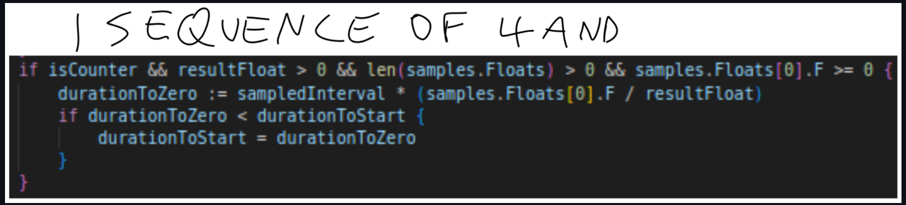
Cyclomatic vs Cognitive Complexity - && ||


3 SEQUENCES OF 2 AND, 1 OR, 2 AND

```
if oh.maxt < oh.head.MinTime() && oh.maxt < oh.head.MinOOOTime() ||
    oh.mint > oh.head.MaxTime() && oh.mint > oh.head.MaxOOOTime() {
    return []string{}, nil
}
```

`a && b || c && d` 3 decisions ⟹ +3 cyclomatic complexity

`a && b || c && d` 3 sequences of like operations ⟹ +3 cognitive complexity

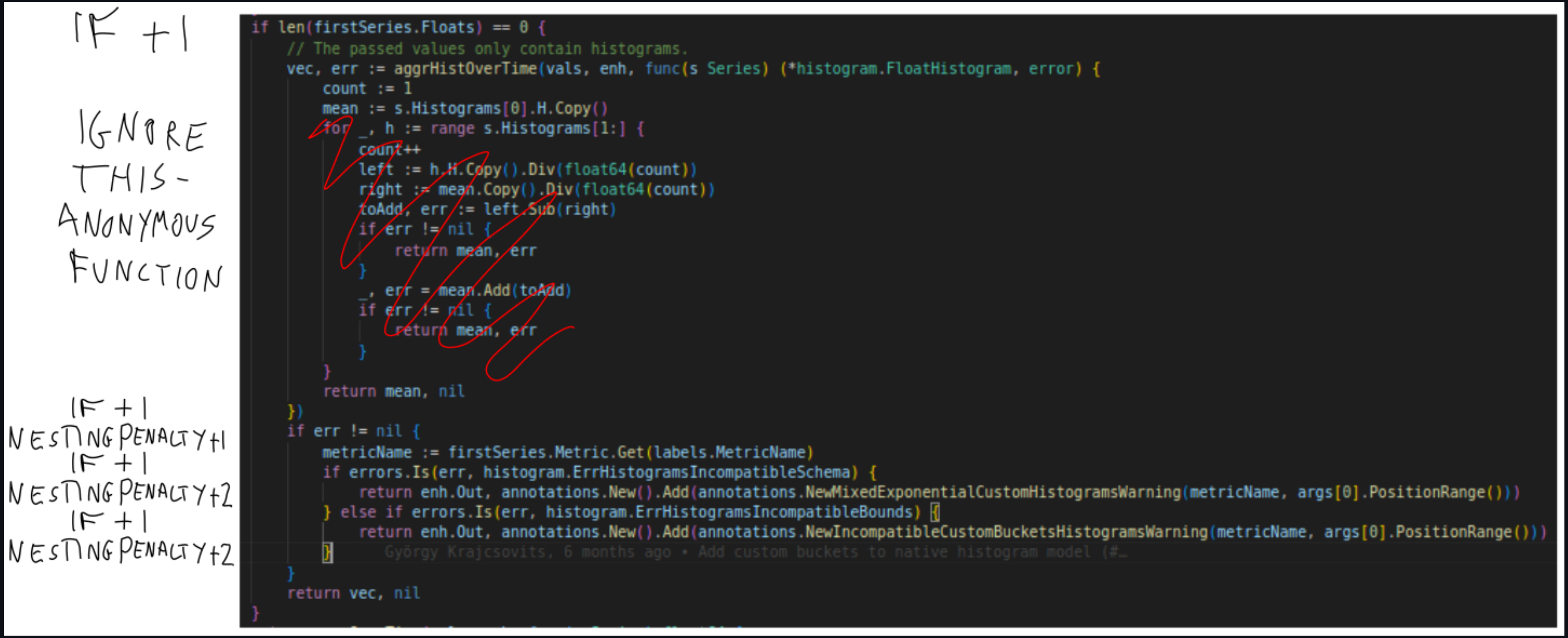

1 SEQUENCE OF 4 AND

```
if isCounter && resultFloat > 0 && len(samples.Floats) > 0 && samples.Floats[0].F >= 0 {
    durationToZero := sampledInterval * (samples.Floats[0].F / resultFloat)
    if durationToZero < durationToStart {
        durationToStart = durationToZero
    }
}
```

`a && b && c && d` 3 decisions ⟹ +3 cyclomatic complexity

`a && b && c && d` 1 sequence of like operations ⟹ +1 cognitive complexity

Contrasting Cyclomatic and Cognitive Complexity



```go
if len(firstSeries.Floats) == 0 {
    // The passed values only contain histograms.
    vec, err := aggrHistOverTime(vals, enh, func(s Series) (*histogram.FloatHistogram, error) {
        count := 1
        mean := s.Histograms[0].H.Copy()
        for _, h := range s.Histograms[1:] {
            count++
            left := h.H.Copy().Div(float64(count))
            right := mean.Copy().Div(float64(count))
            toAdd, err := left.Sub(right)
            if err != nil {
                return mean, err
            }
            _, err = mean.Add(toAdd)
            if err != nil {
                return mean, err
            }
        }
        return mean, nil
    })
    if err != nil {
        metricName := firstSeries.Metric.Get(labels.MetricName)
        if errors.Is(err, histogram.ErrHistogramsIncompatibleSchema) {
            return enh.Out, annotations.New().Add(annotations.NewMixedExponentialCustomHistogramsWarning(metricName, args[0].PositionRange()))
        } else if errors.Is(err, histogram.ErrHistogramsIncompatibleBounds) {
            return enh.Out, annotations.New().Add(annotations.NewIncompatibleCustomBucketsHistogramsWarning(metricName, args[0].PositionRange()))
        }     György Krajcsovits, 6 months ago • Add custom buckets to native histogram model (#…
    }
    return vec, nil
}
```

Each level of nesting incurs +1 penalty

Hence this snippet has cognitive complexity 9 already!

Why Should I Care? Fits-In-My-Headness

Both measures ~ proxy for function size

More flexible than "function length" limit

Long functions of many simple operations are still low complexity

Helps you to optimise for "fits-in-my-head"-ness *

* https://www.youtube.com/watch?v=4Y0tOi7QWqM

Why Should I Care? Chunking

The more branching/decisions in a function...

... the less likely it'll fit in working memory

... the longer it will take to understand

We want to split this into smaller "chunks" of logic

- how we hold larger systems in our mind from smaller ones (*chunking*)
- naming chunks (functions) is the most "to hand" way to document intent

Ideas on this slide put into words by *The Programmer's Brain* by *Felienne Hermans*

https://www.manning.com/books/the-programmers-brain

Linting For Cognitive Complexity in New Go Code

Luckily, this is super easy!

`golangci-lint` (you are using that, right?) has inbuilt support via the `gocognit` linter

https://golangci-lint.run/usage/linters/#gocognit

Don't forget to set a nice low value!

My own recommendation: try 10, if it's too limiting, try 15 or 20

Ulysses Pact

From https://en.wikipedia.org/wiki/Ulysses_pact

> A Ulysses pact or Ulysses contract is a freely made decision that is designed and intended to bind oneself in the future

Ulysses Pact in Software Development

An example could be

- We believe `golangci-lint`'s default linters are valuable because they
  - capture potential bugs much earlier than tests, etc...
  - improve the quality/reliability/etc... of our code

Consequently

- We always use `golangci-lint`'s default linters
- We never disable linting for any of those linters in comments
- We enforce the above in CI

Ulysses Pact - More Background

> The term refers to the pact that Ulysses ... made with his men as they approached the Sirens. Ulysses wanted to hear the Sirens' song although he knew that doing so would render him incapable of rational thought. He put wax in his men's ears so that they could not hear and had them tie him to the mast so that he could not jump into the sea. He ordered them not to change course under any circumstances and to keep their swords upon him and to attack him if he should break free of his bonds.
> Upon hearing the Sirens' song, Ulysses was driven temporarily insane and struggled with all of his might to break free so that he might join the Sirens, which would have meant his death. His men, however, kept their promise, and they refused to release him.

Ulysses Pact in Greenfield Projects

There's always eventually the temptation to ignore the linter

It's just one function... It's not that complex, right?...

Surely I/the new started will understand it in 6 months...

Decision taken 10s/100s/1000s of times in a codebase

Many things are a little harder to grok ⇨ lots of slow down

I like to call this *death by a thousand **what**?s*

Basically

- we find a rule that we believe will always be beneficial for the codebase
- we set up some way to enforce that rule (e.g. a linter)
- we bind ourselves always to follow that rule in future

Cognitive Complexity Linting in Legacy Go Code

Oh no, there's a big pile of legacy code with huge cognitive complexity!

How can we force the complexity of this codebase down over time?

Using a **Ratchet** / **Ratcheting***

- Find a metric that you want to improve around a codebase
- Introduce a tool that can measure that metric automatically
- Establish the current value of that metric
- Put that tool into CI, blocking merges/etc... if it increases

* https://martinfowler.com/articles/useOfMetrics.html#MetricsAsARatchet or https://bugroll.com/how-to-refactor-incredibly-bad-code-a8443cb0da86

Cognitive Complexity Linting in Legacy Go Code

As a bonus, a *self-tightening ratchet* *

- Dynamically measure the current value of the metric (e.g. on `main`)
- Do not allow a commit/branch to increase the metric above this value

Now the metric can be decreased, but never increased, forcing improvement over time!

* I've seen this used a few times, but never named

To Summarise

Key Takeaways

- A function level complexity measure that roughly aligns with intuition
- Linting is easily available for Go codebases (and many other languages)
- IMO limiting cognitive complexity is a good, intelligent way to limit function size
    - helps us to optimise for "fits-in-my-head"-ness of functions
    - forces us to think about how we should chunk our code

To Summarise

Concepts / Names

- A **Ulysses Pact** is an agreement we make now that binds us in future
- A **ratchet** can be used to improve any measureable in a codebase over time

Thanks!

For listening

Please give brutally honest feedback and criticism

https://uk.linkedin.com/in/scottnasmith