Generating Generators

GopherCon Europe 2022

Tamir Bahar (He/Him)

@tmr232

I'm New to Go

My Workflow

- Data is key
- Get your hands dirty
- See concrete data
- Print everything
- Panic on errors

```go
func PrintAllBooks(library Library) {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                fmt.Println(book)
            }
        }
    }
}
```

Iterators

- Data-on-Demand™
- Like having a slice

```go
func PrintAllBooks(library Library) {
    it := IterBooks(library)
    for it.Next() {
        fmt.Println(it.Value())
    }
    if it.Error() != nil {
        panic(it.Error())
    }
}
```

Iterators

- Data-on-Demand™
- Like having a slice

```go
func PrintAllBooks(library Library) {
    it := IterBooks(library)
    for it.Next() {
        fmt.Println(it.Value())
    }
    if it.Error() != nil {
        panic(it.Error())
    }
}
```

Iterators

- Data-on-Demand™
- Like having a slice

```go
func PrintAllBooks(library Library) {
    it := IterBooks(library)
    for it.Next() {
        fmt.Println(it.Value())
    }
    if it.Error() != nil {
        panic(it.Error())
    }
}
```

Iterators

- Data-on-Demand™
- Like having a slice

```go
func PrintAllBooks(library Library) {
    it := IterBooks(library)
    for it.Next() {
        fmt.Println(it.Value())
    }
    if it.Error() != nil {
        panic(it.Error())
    }
}
```

Implementing Iterators

- Much harder than it should be
- Typically 5 main parts

```go
type MyIterator[T any] struct{…}

func NewIterator[T any]() *MyIterator[T] {…}

func (it* MyIterator[T]) Value() T {…}

func (it* MyIterator[T]) Err() error {…}

func (it* MyIterator[T]) Next() bool {…}
```

Implementing Iterators

- Much harder than it should be
- Typically 5 main parts

```go
type MyIterator[T any] struct{…}
func NewIterator[T any]() *MyIterator[T] {…}
func (it* MyIterator[T]) Value() T {…}
func (it* MyIterator[T]) Err() error {…}
func (it* MyIterator[T]) Next() bool {…}
```

Implementing Iterators

- Much harder than it should be
- Typically 5 main parts

```go
type MyIterator[T any] struct{…}

func NewIterator[T any]() *MyIterator[T] {…}

func (it* MyIterator[T]) Value() T {…}

func (it* MyIterator[T]) Err() error {…}

func (it* MyIterator[T]) Next() bool {…}
```

Implementing Iterators

- Much harder than it should be
- Typically 5 main parts

```go
type MyIterator[T any] struct{…}

func NewIterator[T any]() *MyIterator[T] {…}

func (it* MyIterator[T]) Value() T {…}

func (it* MyIterator[T]) Err() error {…}

func (it* MyIterator[T]) Next() bool {…}
```

Implementing Iterators

- We'll use a helper called Closure-Iterator
- Next() -> Advance()
- withValue(value T)
- withError(error)
- exhausted()

```
func rangeIterator(stop int) ClosureIterator[int] {
    current := 0

    return ClosureIterator[int]{
        Advance: func(...) bool {
            if current < stop {
                retval := current
                current++
                return withValue(retval)
            }
            return exhausted()
        },
    }
}
```

Implementing Iterators

- We'll use a helper called Closure-Iterator
- Next() -> Advance()
- withValue(value T)
- withError(error)
- exhausted()

```
func rangeIterator(stop int) ClosureIterator[int] {
    current := 0

    return ClosureIterator[int]{
        Advance: func(...) bool {
            if current < stop {
                retval := current
                current++
                return withValue(retval)
            }
            return exhausted()
        },
    }
}
```

Implementing Iterators

- We'll use a helper called Closure-Iterator
- Next() -> Advance()
- withValue(value T)
- withError(error)
- exhausted()

```
func rangeIterator(stop int) ClosureIterator[int] {
    current := 0

    return ClosureIterator[int]{
        Advance: func(...) bool {
            if current < stop {
                retval := current
                current++
                return withValue(retval)
            }
            return exhausted()
        },
    }
}
```

14

Implementing Iterators – Library Sample

- Remember our book-printer?

```go
func PrintAllBooks(library Library) {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                fmt.Println(book)
            }
        }
    }
}
```

```go
func IterBooks(library Library) ClosureIterator[Book] {
    bookIndex := -1
    shelfIndex := 0
    roomIndex := 0
    return ClosureIterator[Book]{
        Advance: func(...) bool {
            bookIndex++
            for bookIndex >= len(library.Rooms[roomIndex].Shelves[shelfIndex].Books) {
                bookIndex = 0
                shelfIndex++
                for shelfIndex >= len(library.Rooms[roomIndex].Shelves) {
                    shelfIndex = 0
                    roomIndex++
                    if roomIndex >= len(library.Rooms) {
                        return exhausted()
                    }
                }
            }
            return withValue(library.Rooms[roomIndex].Shelves[shelfIndex].Books[bookIndex])
        },
    }
}
```

```
func IterBooks(library Library) ClosureIterator[Book] {
    bookIndex := -1
    shelfIndex := 0
    roomIndex := 0
    return ClosureIterator[Book]{
        Advance: func(...) bool {
            bookIndex++
            for bookIndex >= len(library.Rooms[roomIndex].Shelves[shelfIndex].Books) {
                bookIndex = 0
                shelfIndex++
                for shelfIndex >= len(library.Rooms[roomIndex].Shelves) {
                    shelfIndex = 0
                    roomIndex++
                    if roomIndex >= len(library.Rooms) {
                        return exhausted()
                    }
                }
            }
            return withValue(library.Rooms[roomIndex].Shelves[shelfIndex].Books[bookIndex])
        },
    }
}
```

Special case initialization

Working from the inside out

Tricky indexing

17

Implementing Iterators – Library Sample

- Hard to implement
- Hard to read
- Hard to maintain
- Converting a nested loop to an iterator should not be a challenge

Implementing Iterators - Generators

```go
func IterBooks(library Library) gengen.Generator[Book] {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                gengen.Yield(book)
            }
        }
    }
    return nil
}
```

```go
func PrintAllBooks(library Library) {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                fmt.Println(book)
            }
        }
    }
}
```

Implementing Iterators - Generators

```go
func IterBooks(library Library) gengen.Generator[Book] {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                gengen.Yield(book)
            }
        }
    }
    return nil
}
```

```go
func PrintAllBooks(library Library) {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                fmt.Println(book)
            }
        }
    }
}
```

Implementing Iterators - Generators

```go
func IterBooks(library Library) gengen.Generator[Book] {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                gengen.Yield(book)
            }
        }
    }
    return nil
}
```

```go
func PrintAllBooks(library Library) {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                fmt.Println(book)
            }
        }
    }
}
```

Implementing Iterators - Generators

```go
func IterBooks(library Library) gengen.Generator[Book] {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                gengen.Yield(book)
            }
        }
    }
    return nil
}
```

```go
func PrintAllBooks(library Library) {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                fmt.Println(book)
            }
        }
    }
}
```

Generator Syntax

- All generator functions use `Yield`
- When called, a generator is returned, but no code is executed

```
func IterBooks(library Library) Generator[Book] {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                gengen.Yield(book)
            }
        }
    }
    return nil
}
```

Generator Syntax

```go
func PrintAllBooks(library Library) {
    it := IterBooks(library)
    for it.Next() {
        fmt.Println(it.Value())
    }
    if it.Error() != nil {
        panic(it.Error())
    }
}
```

```go
func IterBooks(library Library) Generator[Book] {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                gengen.Yield(book)
            }
        }
    }
    return nil
}
```

Generator Syntax

```go
func PrintAllBooks(library Library) {
    it := IterBooks(library)
    for it.Next() {
        fmt.Println(it.Value())
    }
    if it.Error() != nil {
        panic(it.Error())
    }
}
```
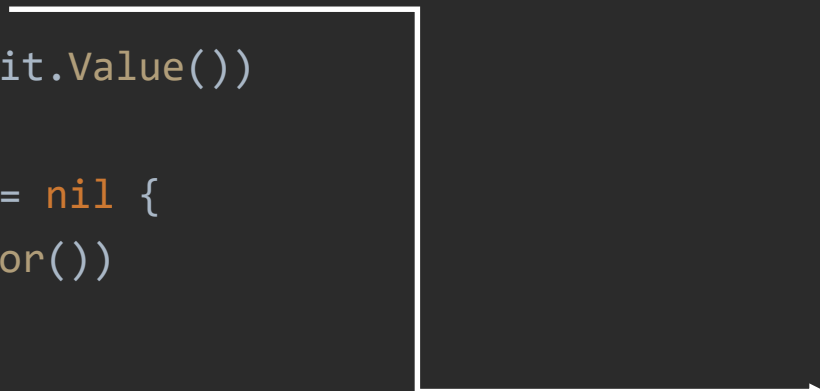
```go
func IterBooks(library Library) Generator[Book] {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                gengen.Yield(book)
            }
        }
    }
    return nil
}
```

26

Generator Syntax

```go
func PrintAllBooks(library Library) {
    it := IterBooks(library)
    for it.Next() {
        fmt.Println(it.Value())
    }
    if it.Error() != nil {
        panic(it.Error())
    }
}
```

```go
func IterBooks(library Library) Generator[Book] {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                gengen.Yield(book)
            }
        }
    }
    return nil
}
```

Generator Syntax

```go
func PrintAllBooks(library Library) {
    it := IterBooks(library)
    for it.Next() {
        fmt.Println(it.Value())
    }
    if it.Error() != nil {
        panic(it.Error())
    }
}
```

```go
func IterBooks(library Library) Generator[Book] {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                gengen.Yield(book)
            }
        }
    }
    return nil
}
```

Generator Syntax

```go
func PrintAllBooks(library Library) {
    it := IterBooks(library)
    for it.Next() {
        fmt.Println(it.Value())
    }
    if it.Error() != nil {
        panic(it.Error())
    }
}
```

```go
func IterBooks(library Library) Generator[Book] {
    for _, room := range library.Rooms {
        for _, shelf := range room.Shelves {
            for _, book := range shelf.Books {
                gengen.Yield(book)
            }
        }
    }
    return nil
}
```

Generating Generators

- Generators are great, but they aren't Go...

- But we can change that...

- Using Code Generation!

- Lucky for us – Go has AMAZING tooling

Generating Generators – Build Tricks

- go generate to generate implementation from definitions

  `//go:generate go run github.com/tmr232/gengen/cmd/gengen`

- Build tags separate pretend-Go from real-Go

  `//go:build gengen` ⟶ `//go:build !gengen`

- Other code copied verbatim

- Only real-Go goes into executable

Generating Generators – Code Transformations

```go
func Empty() gengen.Geneartor[int] {
    return nil
}
```

Generating Generators – Code Transformations

```
func Empty() ClosureIterator[int] {
    return ClosureIterator[int]{
        Advance: func(...) bool {
            return nil
        }
    }
}
```

Generating Generators – Code Transformations

```
func Empty() ClosureIterator[int] {
    return ClosureIterator[int]{
        Advance: func(...) bool {
            return exhausted()
        }
    }
}
```

Generating Generators – Code Transformations

```go
func Error() gengen.Geneartor[int] {
    return MyError{}
}
```

↓

```go
func Error() ClosureIterator[int] {
    return ClosureIterator[int]{
        Advance: func(...) bool {
            return withError(MyError{})
        }
    }
}
```

40

Yielding Values

```go
func HelloWorld() gengen.Generator[string] {
    gengen.Yield("Hello, World!")
    return nil
} ] (} {}
```

func Yield[T any](T) {}

Yielding Values

```
func HelloWorld() ClosureIterator[string] {
    return ClosureIterator[string]{
        Advance: func(...) bool {
            gengen.Yield("Hello, World!")
            return nil
        },
    }
}
```

Yielding Values

```go
func HelloWorld() ClosureIterator[string] {
    return ClosureIterator[string]{
        Advance: func(...) bool {
            // gengen.Yield("Hello, World!")
            return withValue("Hello, World!")
            return nil
        },
    }
}
```

44

Yielding Values

```
func HelloWorld() ClosureIterator[string] {
    next := 0
    return ClosureIterator[string]{
        Advance: func(...) bool {
            switch next {
            case 0:
                goto Label0
            case 1:
                goto Label1
            }
        Label0:
            // gengen.Yield("Hello, World!")
            next = 1
            return withValue("Hello, World!")
        Label1:
            return nil
        },
    }
}
```

45

Yielding Values

```
func HelloWorld() ClosureIterator[string] {
    next := 0
    return ClosureIterator[string]{
        Advance: func(...) bool {
            switch next {
            case 0:
                goto Label0
            case 1:
                goto Label1
            }
        Label0:
            // gengen.Yield("Hello, World!")
            next = 1
            return withValue("Hello, World!")
        Label1:
            return nil
        },
    }
}
```

46

Yielding Values

```
func HelloWorld() ClosureIterator[string] {
    next := 0
    return ClosureIterator[string]{
        Advance: func(...) bool {
            switch next {
            case 0:
                goto Label0
            case 1:
                goto Label1
            }
        Label0:
            // gengen.Yield("Hello, World!")
            next = 1
            return withValue("Hello, World!")
        Label1:
            return nil
        },
    }
}
```
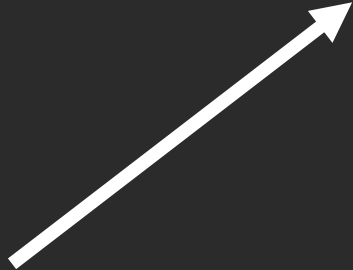
47

Yielding Values

```
func HelloWorld() ClosureIterator[string] {
    next := 0
    return ClosureIterator[string]{
        Advance: func(...) bool {
            switch next {
            case 0:
                goto Label0
            case 1:
                goto Label1
            }
        Label0:
            // gengen.Yield("Hello, World!")
            next = 1
            return withValue("Hello, World!")
        Label1:
            return exhausted()
        },
    }
}
```

48

Using Goto

- Go's goto is safe

- Can't skip variable declarations

- Can't enter blocks

```
    goto skipDeclatation
    msg := "Hello, World!"
skipDeclaration:
    // What is the value of `msg`?
    fmt.Println(msg)
```

```
goto intoBlock
if cond {
intoBlock:
    // Does the condition hold?
    doSomething()
}
```

Using Goto – Variable Declarations

- Move all variables into the state-block
- As a bonus – preserves state across calls to `Next()`

```
func HelloWorld() ClosureIterator[string] {
    // State-Block
    // All vars declared here
    next := 0
    return ClosureIterator[string]{
        Advance: func(...) bool {
            switch next {
            case 0:
                goto Label0
            case 1:
                goto Label1
            }
        Label0:
            next = 1
            return withValue("Hello, World!")
        Label1:
            return exhausted()
        },
    }
}
```

Using Goto – Blocks

- Blocks are for scoping & control-flow
- We eliminated scoping
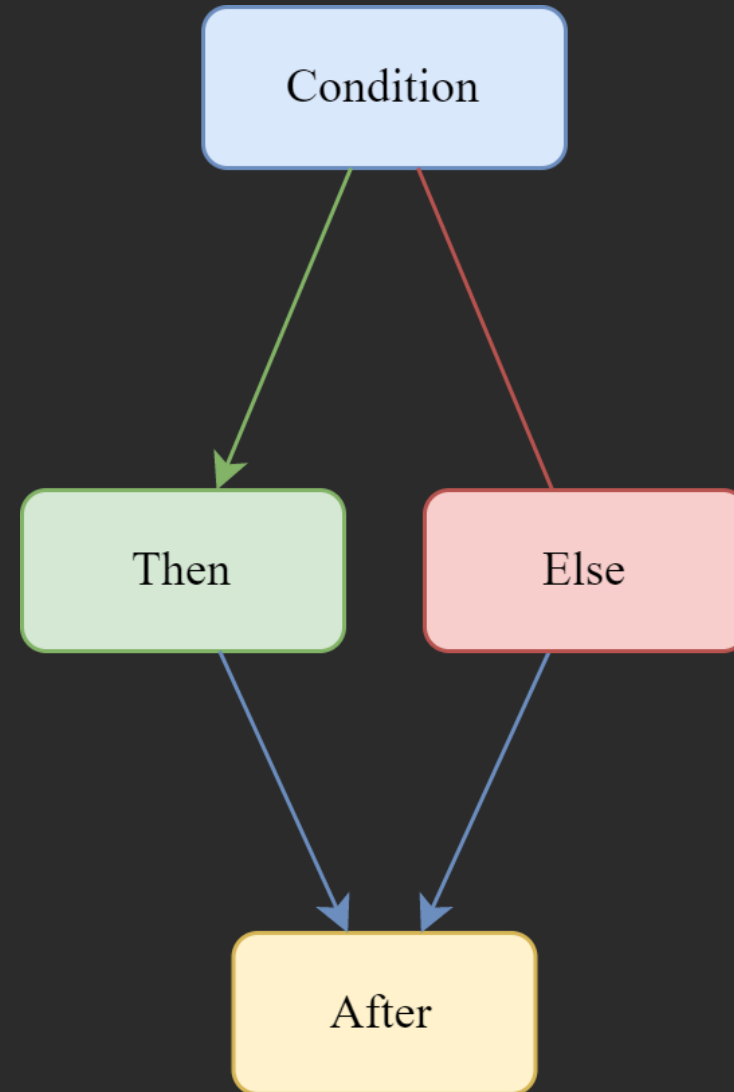- We can transform blocks away too!

Control Flow - if

```
if alpha {

    gengen.Yield("a")
    gengen.Yield("b")
    gengen.Yield("c")

} else {

    gengen.Yield("1")
    gengen.Yield("2")
    gengen.Yield("3")

}
```
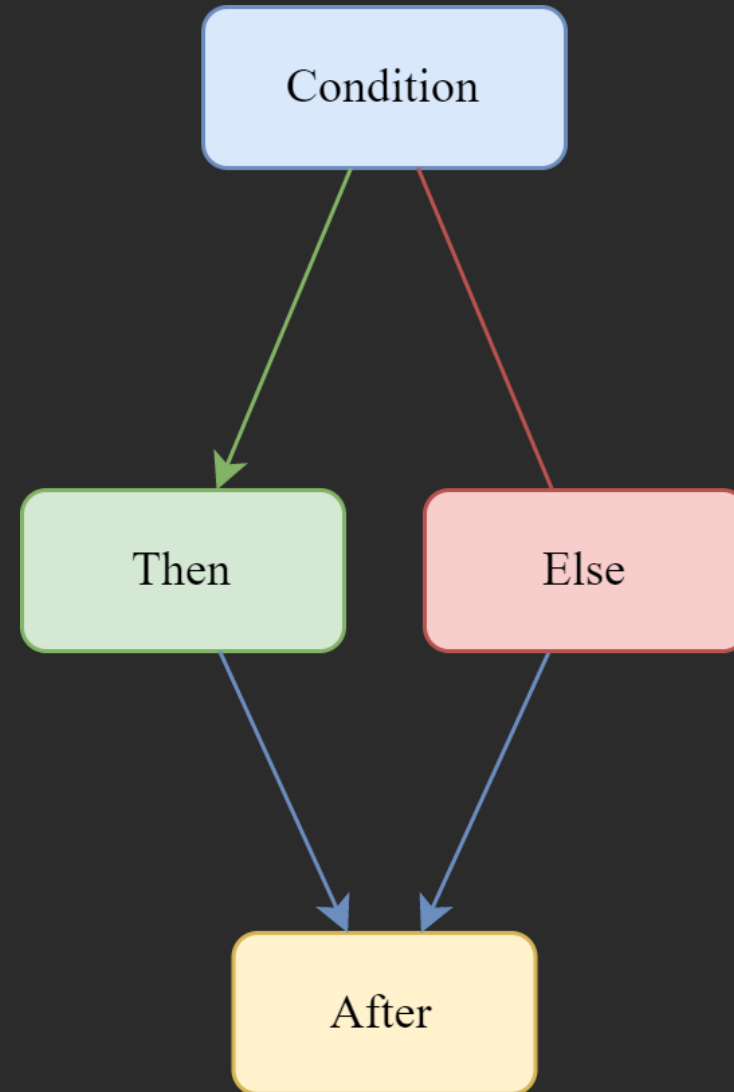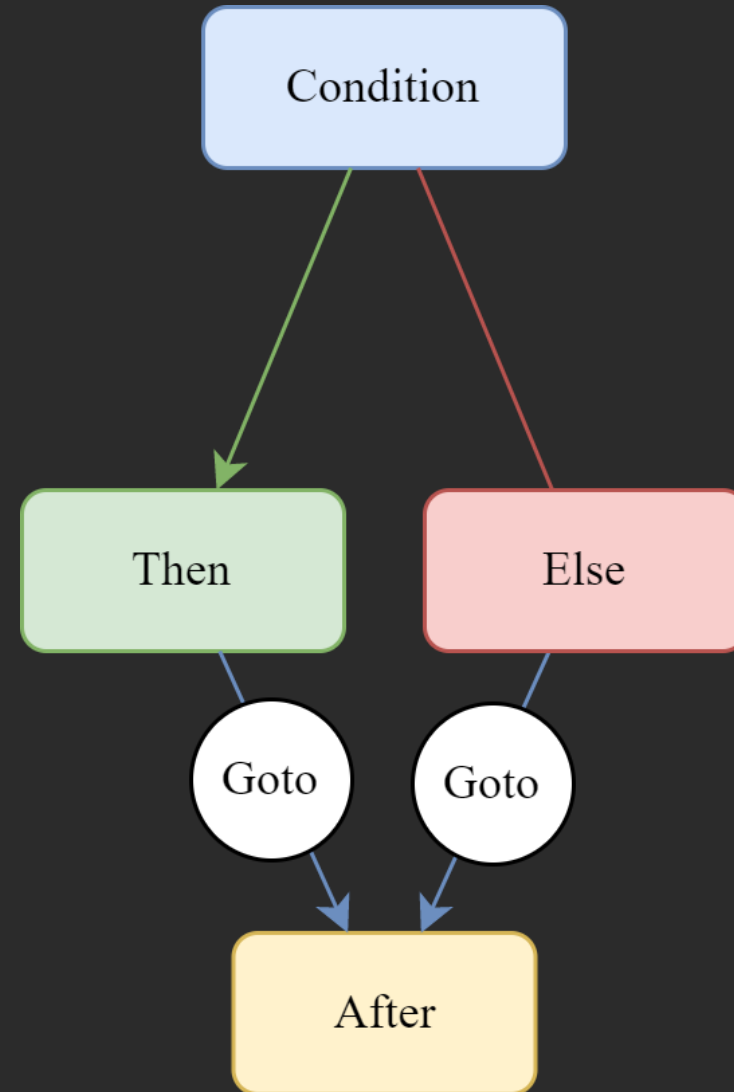
Control Flow - if

```
if alpha {
    thenLabel:
        gengen.Yield("a")
        gengen.Yield("b")
        gengen.Yield("c")

    } else {
    elseLabel:
        gengen.Yield("1")
        gengen.Yield("2")
        gengen.Yield("3")

    }
afterLabel:
```

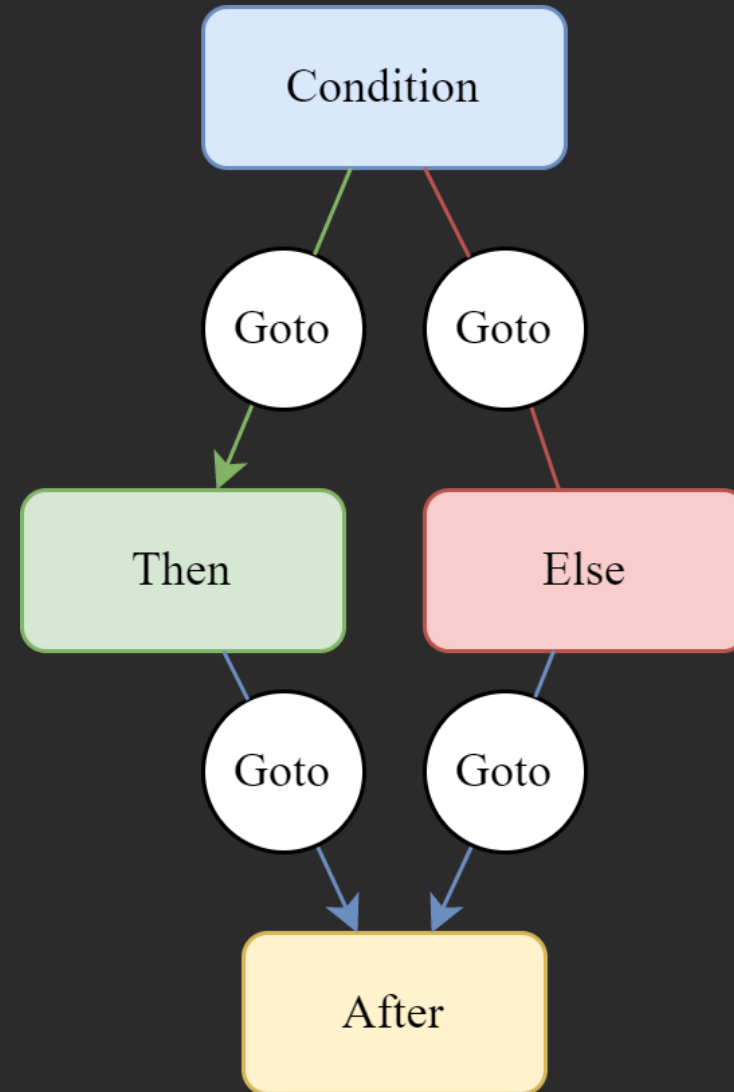# Control Flow - if

```
if alpha {
    thenLabel:
        gengen.Yield("a")
        gengen.Yield("b")
        gengen.Yield("c")
        goto afterLabel
} else {
    elseLabel:
        gengen.Yield("1")
        gengen.Yield("2")
        gengen.Yield("3")
        goto afterLabel
}
afterLabel:
```


54

Control Flow - if

```
if alpha {
    goto thenLabel
} else {
    goto elseLabel
}
thenLabel:
    gengen.Yield("a")
    gengen.Yield("b")
    gengen.Yield("c")
    goto afterLabel
elseLabel:
    gengen.Yield("1")
    gengen.Yield("2")
    gengen.Yield("3")
    goto afterLabel
afterLabel:
```

Control Flow - forever

```
n := 0
for {

    gengen.Yield(n)
    n++

}
```

Control Flow - forever

```
n := 0
for {
loopHead:
    gengen.Yield(n)
    n++

}
afterLoop:
```

Control Flow - forever

```
n := 0
for {
loopHead:
    gengen.Yield(n)
    n++
    goto loopHead
}
afterLoop:
```

Control Flow - forever

```
    n := 0
loopHead:
    gengen.Yield(n)
    n++
    goto loopHead
afterLoop:
```

Control Flow - while

```
n := 0
for n < 10 {
    gengen.Yield(n)
    n++
}
```

Control Flow - while

```
n := 0
for {
    if n < 10 {
        gengen.Yield(n)
        n++
    } else {
        break
    }
}
```

Control Flow - while

```
loopHead:
    if n < 10 {
        goto loopBody
    } else {
        goto afterLoop
    }
loopBody:
    gengen.Yield(n)
    n++
afterLoop:
```
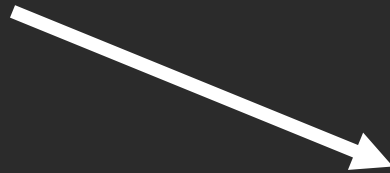
Control Flow – C-Style Loop

```go
for n := 0; n < 10; n++ {
    gengen.Yield(n)
}
```

Control Flow – C-Style Loop

```go
for n := 0; n < 10; n++ {
    gengen.Yield(n)
}
```

```go
n := 0
for n < 10 {
    gengen.Yield(n)

    n++
}
```

Control Flow – C-Style Loop

```go
for n := 0; n < 10; n++ {
    gengen.Yield(n)
}
```

```go
n := 0
for n < 10 {
    gengen.Yield(n)
    // continue jumps here!
    n++
}
```

Control Flow – for range

```go
for index, item := range slice {
    gengen.Yield(item)
}
```

Control Flow – for range

```
for index, item := range slice {
    gengen.Yield(item)
}
```

```
iter := SliceAdaptor(slice)
for iter.Next() {
    index, item := iter.Value()
    gengen.Yield(item)
}
```

Control Flow - Continued

- Apply to remaining control structures
- defer cannot be transformed
- (Also – what will it mean in a generator?)

Demo Time!

```go
//go:build gengen

package demo

import (
    "github.com/tmr232/gengen"
)

//go:generate go run gengen

func Fibonacci() gengen.Generator[int] {
    a := 1
    b := 1
    for {
        gengen.Yield(a)
        a, b = b, a+b
    }
}
```

Run:
```
$ go generate -tags gengen
```

69

Demo Time!

```go
//go:build !gengen

package demo

import "github.com/tmr232/gengen"

func Fibonacci() gengen.Generator[int] {
    var a int
    var b int
    __next := 0
    return &gengen.GeneratorFunction[int]{
        Advance: func(...) bool {
            switch __next {
            case 0:
                goto __Next0
            case 1:
                goto __Next1
            }
        __Next0:
            a = 1
            b = 1
        __Head1:
            __next = 1
            return withValue(a)
        __Next1:
            a, b = b, a+b
            goto __Head1
        },
    }
}
```

Demo Time!

```go
func main() {
    fib := Fibonacci()
    for i := 0; i < 10 && fib.Next(); i++ {
        fmt.Println(fib.Value())
    }
}
```

Run:
  $ go run

71

Demo Time!

```go
func main() {
    fib := Fibonacci()
    for i := 0; i < 10 && fib.Next(); i++ {
        fmt.Println(fib.Value())
    }
}
```

Run:
```
$ go run
1
1
2
3
5
8
13
21
34
55
```

Generating Generators

GopherCon Europe 2022

Tamir Bahar (He/Him)

@tmr232