

Generating Generators

GoLab 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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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()  
        },  
    }  
}
```

Implementing Iterators – Library Sample

- Remember our book-printer?

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

```

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
```

Special case initialization

```
    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])
```

```
        },
```

```
    }
```

```
}
```

Working from the inside out

Tricky indexing

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

```
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  
}
```

```
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

```
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  
}
```

```
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

```
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  
}
```

```
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

```
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  
}
```

```
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

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

```
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

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




```
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

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

```
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

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


```
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

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


```
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

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

```
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

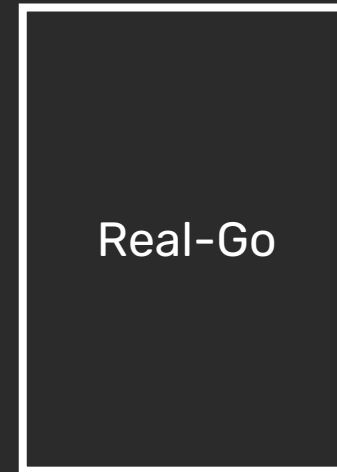
Generating Generators – Build Tricks

`//go:build gengen`



`go generate -tags gengen`

`//go:build !gengen`



Generating Generators – Build Tricks

Pretend Go (Generators)

- `//go:build` gengen

Real Go (Implementations)

- `//go:build` !gengen

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

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

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

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

Generating Generators – Code Transformations

- Copy into Closure Iterator
- return nil -> exhausted()

```
func Empty() ClosureIterator[int] {  
    return ClosureIterator[int]{  
        AdvanceEmptyGenerator() ClosureIterator[int] {  
            return exhausted()  
        }  
    }  
}
```

Generating Generators – Code Transformations

```
func Empty() ClosureIterator[int] {  
    return ClosureIterator[int]{  
        AdvanceEmpty(func gen genB genC genD genE genF genG genH genI genJ genK genL genM genN genO genP genQ genR genS genT genU genV genW genX genY genZ genAA genAB genAC genAD genAE genAF genAG genAH genAI genAJ genAK genAL genAM genAN genAO genAP genAQ genAR genAS genAT genAU genAV genAW genAX genAY genAZ genBA genBB genBC genBD genBE genBF genBG genBH genBI genBJ genBK genBL genBM genBN genBO genBP genBQ genBR genBS genBT genBU genBV genBW genBX genBY genBZ genCA genCB genCC genCD genCE genCF genCG genCH genCI genCJ genCK genCL genCM genCN genCO genCP genCQ genCR genCS genCT genCU genCV genCW genCX genCY genCZ genDA genDB genDC genDD genDE genDF genDG genDH genDI genDJ genDK genDL genDM genDN genDO genDP genDQ genDR genDS genDT genDU genDV genDW genDX genDY genDZ genEA genEB genEC genED genEE genEF genEG genEH genEI genEJ genEK genEL genEM genEN genEO genEP genEQ genER genES genET genEU genEV genEW genEX genEY genEZ genFA genFB genFC genFD genFE genFF genFG genFH genFI genFJ genFK genFL genFM genFN genFO genFP genFQ genFR genFS genFT genFU genFV genFW genFX genFY genFZ genGA genGB genGC genGD genGE genGF genGG genGH genGI genGJ genGK genGL genGM genGN genGO genGP genGQ genGR genGS genGT genGU genGV genGW genGX genGY genGZ genHA genHB genHC genHD genHE genHF genHG genHH genHI genHJ genHK genHL genHM genHN genHO genHP genHQ genHR genHS genHT genHU genHV genHW genHX genHY genHZ genIA genIB genIC genID genIE genIF genIG genIH genII genIJ genIK genIL genIM genIN genIO genIP genIQ genIR genIS genIT genIU genIV genIW genIX genIY genIZ genJA genJB genJC genJD genJE genJF genJG genJH genJI genJJ genJK genJL genJM genJN genJO genJP genJQ genJR genJS genJT genJU genJV genJW genJX genJY genJZ genKA genKB genKC genKD genKE genKF genKG genKH genKI genKJ genKK genKL genKM genKN genKO genKP genKQ genKR genKS genKT genKU genKV genKW genKX genKY genKZ genLA genLB genLC genLD genLE genLF genLG genLH genLI genLJ genLK genLL genLM genLN genLO genLP genLQ genLR genLS genLT genLU genLV genLW genLX genLY genLZ genMA genMB genMC genMD genME genMF genMG genMH genMI genMJ genMK genML genMM genMN genMO genMP genMQ genMR genMS genMT genMU genMV genMW genMX genMY genMZ genNA genNB genNC genND genNE genNF genNG genNH genNI genNJ genNK genNL genNM genNN genNO genNP genNQ genNR genNS genNT genNU genNV genNW genNX genNY genNZ genOA genOB genOC genOD genOE genOF genOG genOH genOI genOJ genOK genOL genOM genON genOO genOP genOQ genOR genOS genOT genOU genOV genOW genOX genOY genOZ genPA genPB genPC genPD genPE genPF genPG genPH genPI genPJ genPK genPL genPM genPN genPO genPP genPQ genPR genPS genPT genPU genPV genPW genPX genPY genPZ genQA genQB genQC genQD genQE genQF genQG genQH genQI genQJ genQK genQL genQM genQN genQO genQP genQQ genQR genQS genQT genQU genQV genQW genQX genQY genQZ genRA genRB genRC genRD genRE genRF genRG genRH genRI genRJ genRK genRL genRM genRN genRO genRP genRQ genRR genRS genRT genRU genRV genRW genRX genRY genRZ genSA genSB genSC genSD genSE genSF genSG genSH genSI genSJ genSK genSL genSM genSN genSO genSP genSQ genSR genSS genST genSU genSV genSW genSX genSY genSZ genTA genTB genTC genTD genTE genTF genTG genTH genTI genTJ genTK genTL genTM genTN genTO genTP genTQ genTR genTS genTT genTU genTV genTW genTX genTY genTZ genUA genUB genUC genUD genUE genUF genUG genUH genUI genUJ genUK genUL genUM genUN genUO genUP genUQ genUR genUS genUT genUU genUV genUW genUX genUY genUZ genVA genVB genVC genVD genVE genVF genVG genVH genVI genVJ genVK genVL genVM genVN genVO genVP genVQ genVR genVS genVT genVU genVV genVW genVX genVY genVZ genWA genWB genWC genWD genWE genWF genWG genWH genWI genWJ genWK genWL genWM genWN genWO genWP genWQ genWR genWS genWT genWU genWV genWW genWX genWY genWZ genXA genXB genXC genXD genXE genXF genXG genXH genXI genXJ genXK genXL genXM genXN genXO genXP genXQ genXR genXS genXT genXU genXV genXW genXX genXY genXZ genYA genYB genYC genYD genYE genYF genYG genYH genYI genYJ genYK genYL genYM genYN genYO genYP genYQ genYR genYS genYT genYU genYV genYW genYX genYY genYZ genZA genZB genZC genZD genZE genZF genZG genZH genZI genZJ genZK genZL genZM genZN genZO genZP genZQ genZR genZS genZT genZU genZV genZW genZX genZY genZZ
```

Generating Generators – Code Transformations

```
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

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



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


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 withValue("Hello, World!")
        return exhausted()
        Advance: func(...) bool {
            // gengen.Yield("Hello, World!")
            return withValue("Hello, World!")
            return nil
        },
    }
}

func Yield[T any](T) {}

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

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

Yielding Values

```
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

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

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  
        },  
    }  
}
```

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  
            },  
    }  
}
```

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  
            },  
    }  
}
```

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()  
        },  
    }  
}
```


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()`



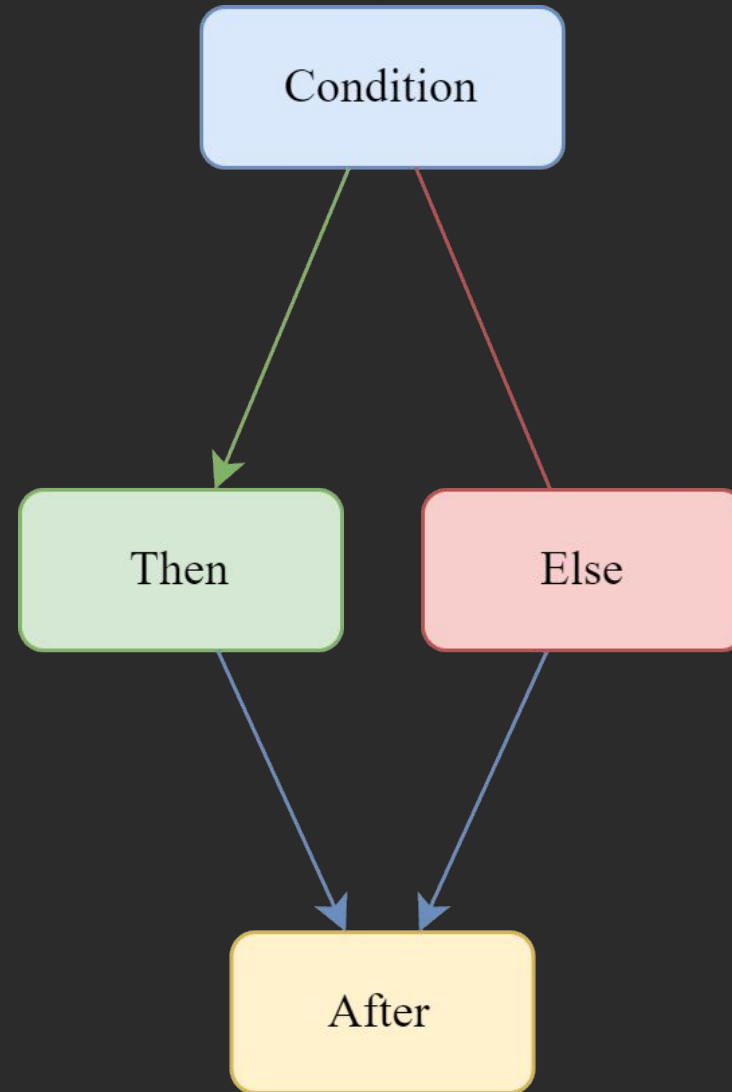
```
func HelloWorld() ClosureIterator[string] {  
    // State-Block  
    // All vars declared here  
    next := 0  
    return ClosureIterator[string]{  
        Advance: func(...) bool {  
            switch next {  
            case 0:  
                goto Label0  
            case 1:  
     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

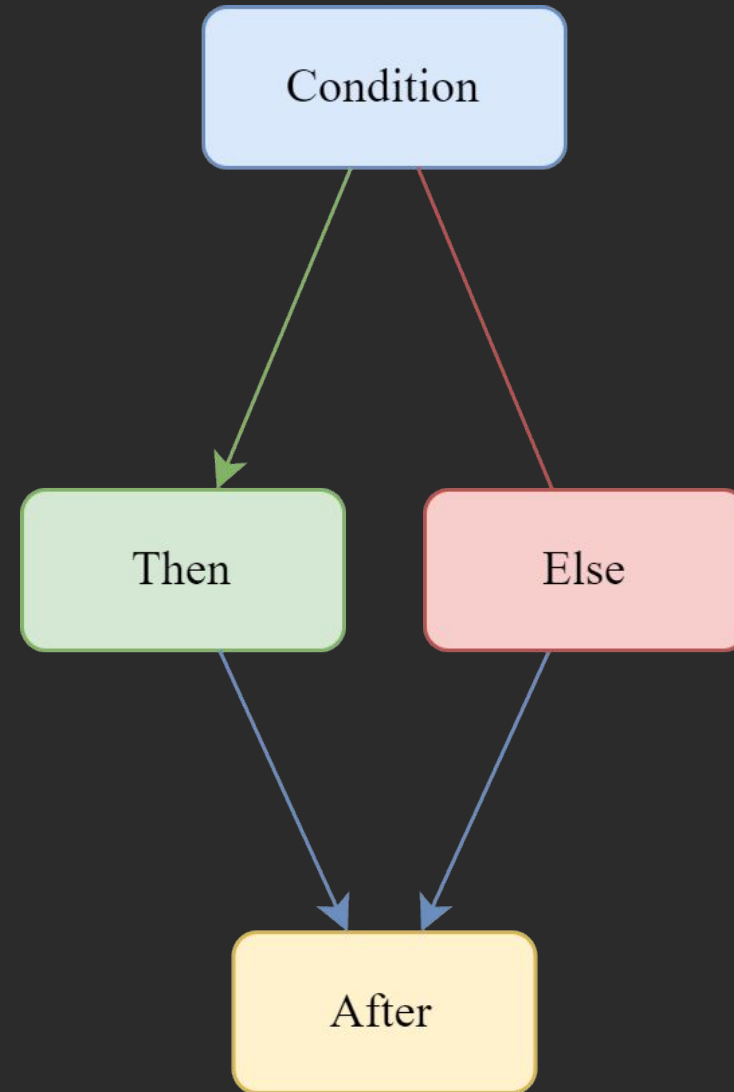
Control Flow - if

```
if alpha {  
  
    gengen.Yield("a")  
    gengen.Yield("b")  
    gengen.Yield("c")  
  
} else {  
  
    gengen.Yield("1")  
    gengen.Yield("2")  
    gengen.Yield("3")  
  
}
```



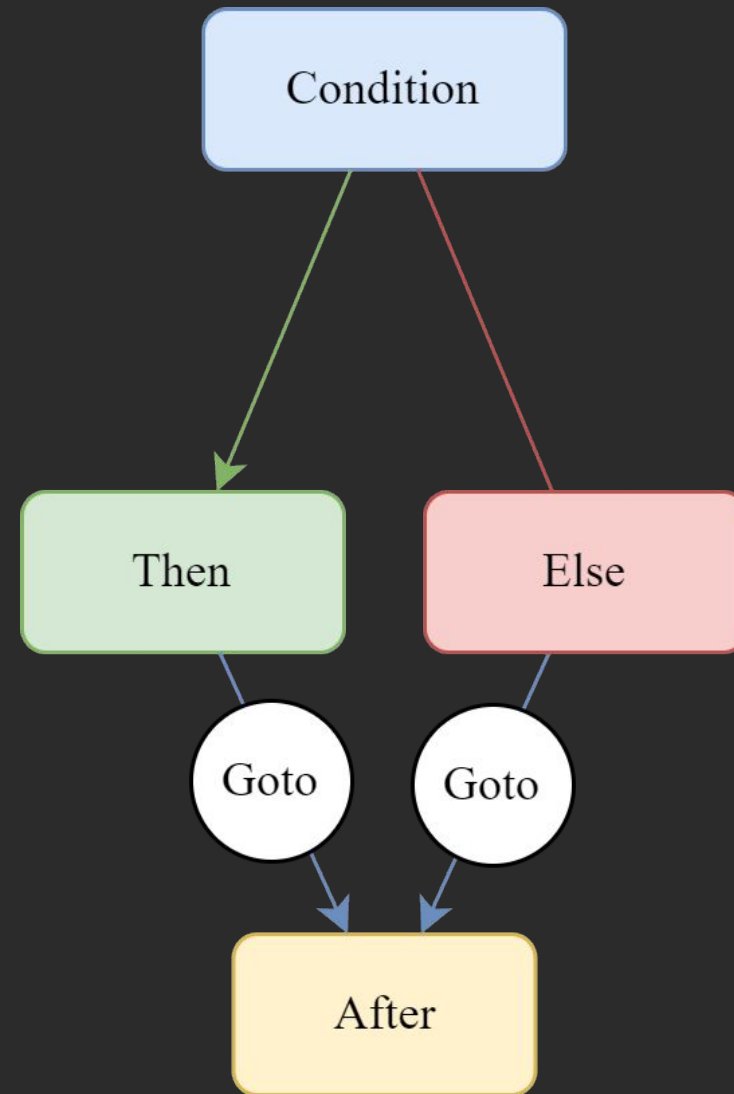
Control Flow - if

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



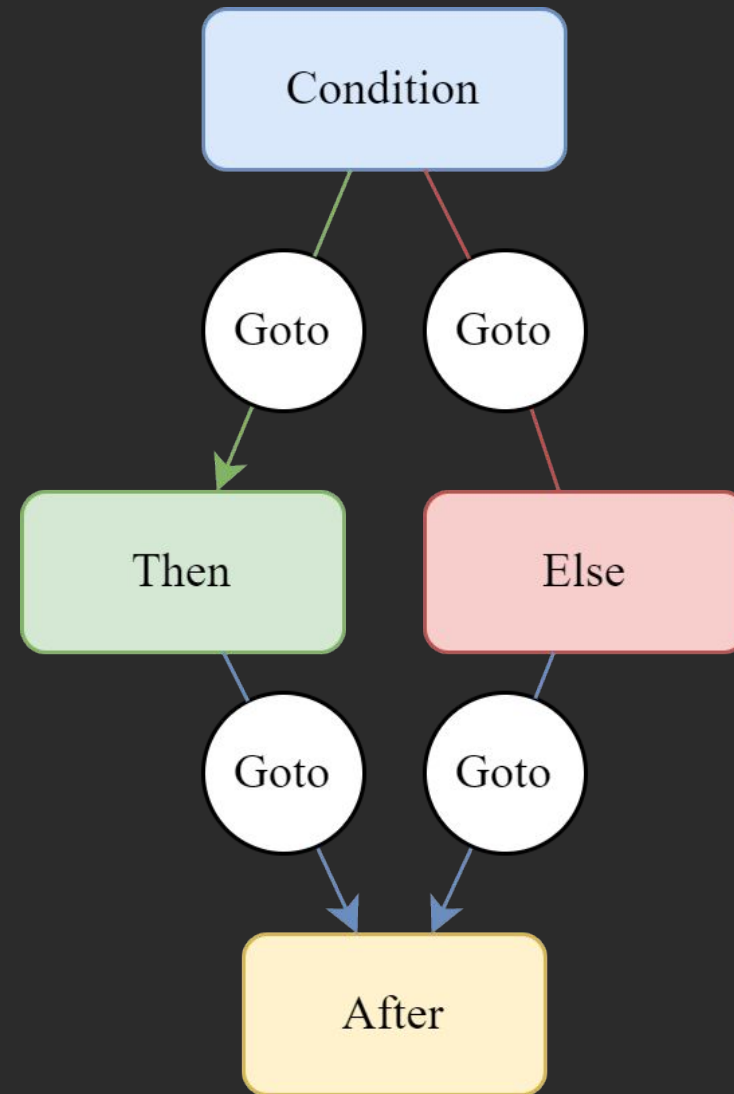
Control Flow - if

```
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:
```



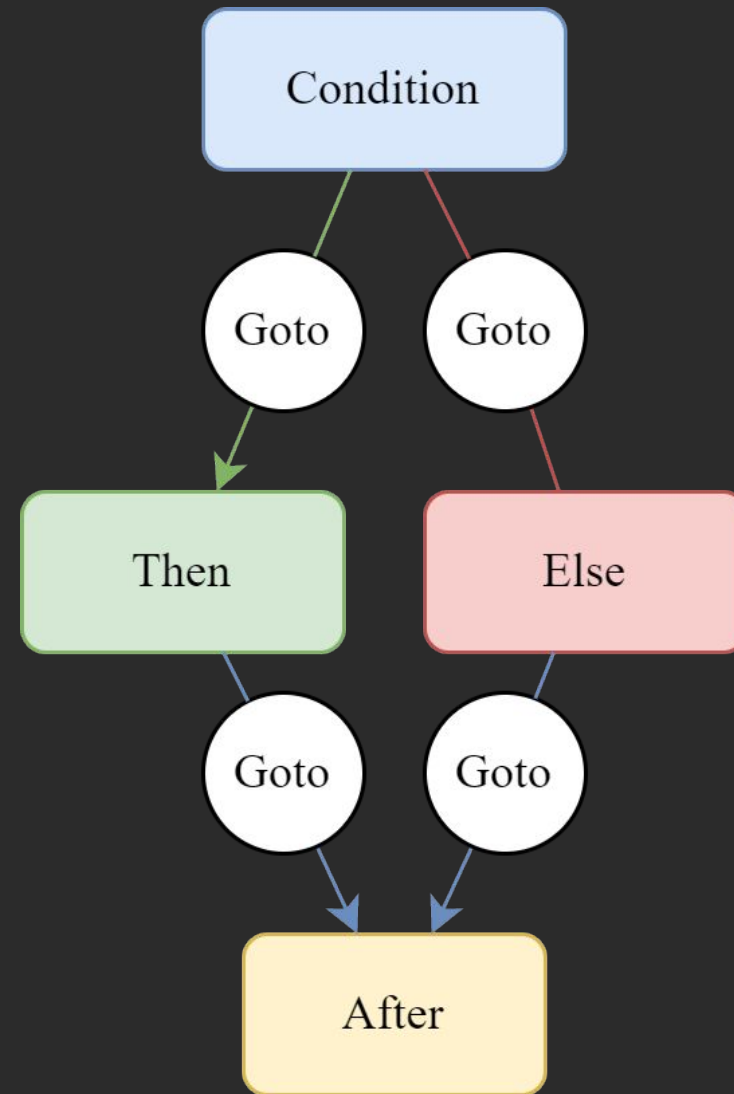
Control Flow - if

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



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++
    goto loopHead
afterLoop:
```

Control Flow – C-Style Loop

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


Control Flow – C-Style Loop

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



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

Control Flow – C-Style Loop

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



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

Control Flow – for range

```
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: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
```

Demo Time!

```
//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!

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

Run:

```
$ go run
```


Demo Time!

```
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



GoLab 2022

Tamir Bahar (He/Him)



@tmr232

Generating Generators - Links

-   @tmr232
- github.com/tmr232/gengen
- github.com/tmr232/gengen-demo
- pkg.go.dev/github.com/tmr232/gengen