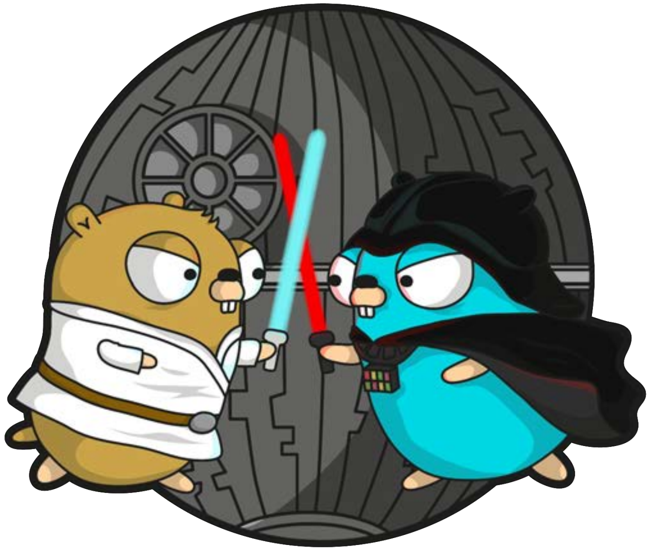


Coroutines and Go

Raghav Roy

```
whoami
```

What I will be covering

- Coroutines as generalised subroutines

What I will be covering

- Coroutines as generalised subroutines
- How it started

What I will be covering

- Coroutines as generalised subroutines
- How it started
- Classifying coroutines

What I will be covering

- Coroutines as generalised subroutines
- How it started
- Classifying coroutines - Building up to Full Coroutines

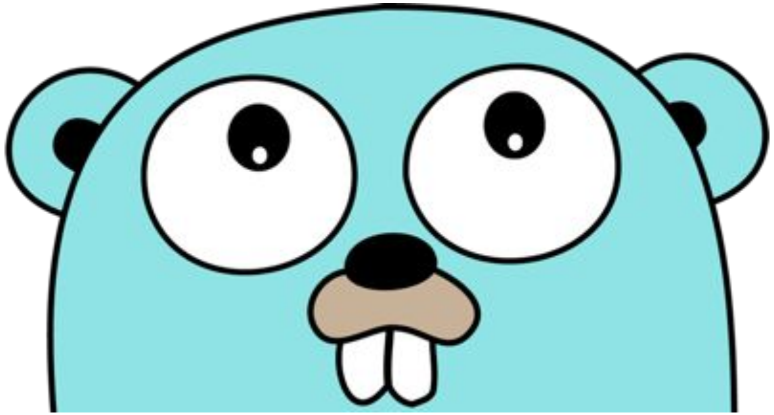
What I will be covering

- Coroutines as generalised subroutines
- How it started
- Classifying coroutines - Building up to Full Coroutines
- Coroutines in Go

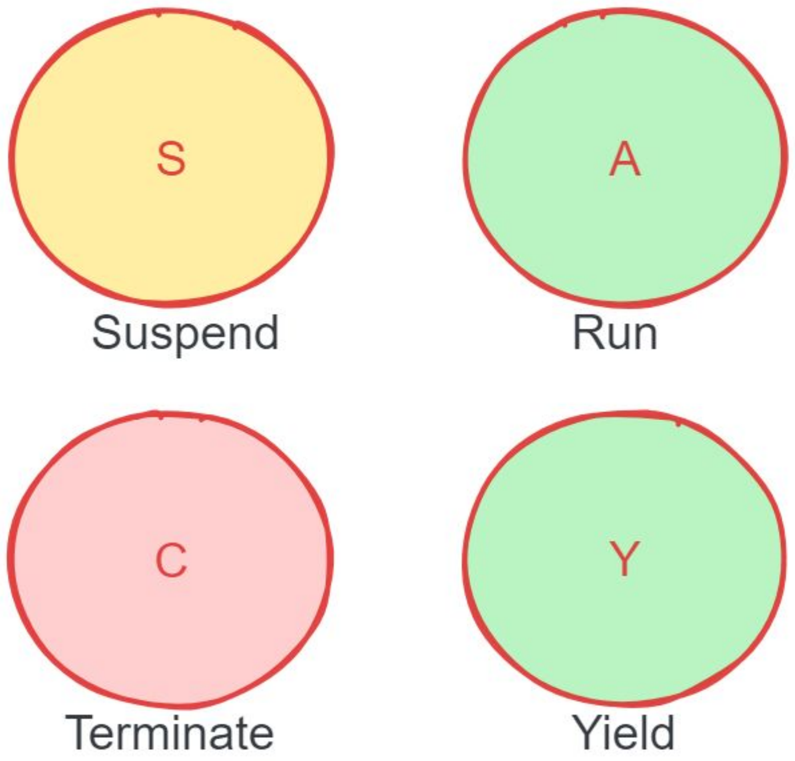
What I will be covering

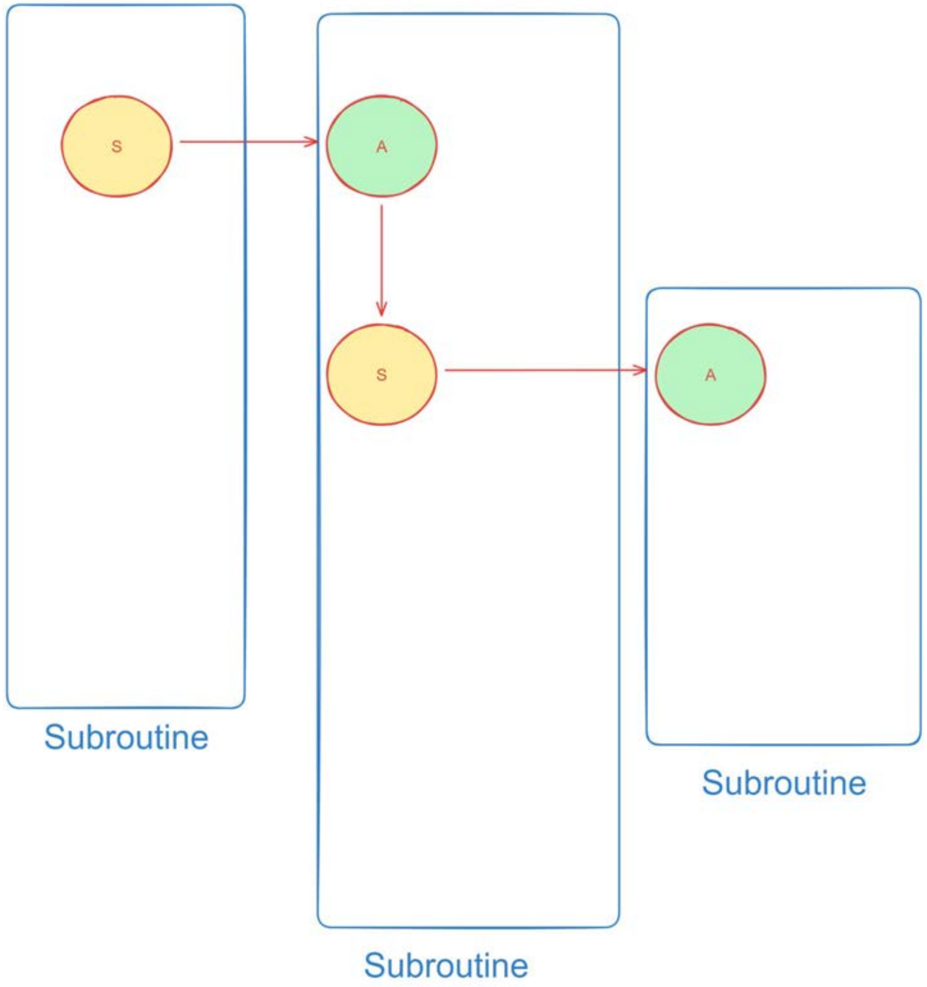
- Coroutines as generalised subroutines
- How it started
- Classifying coroutines - Building up to Full Coroutines
- Coroutines in Go
- Go runtime changes to support them natively

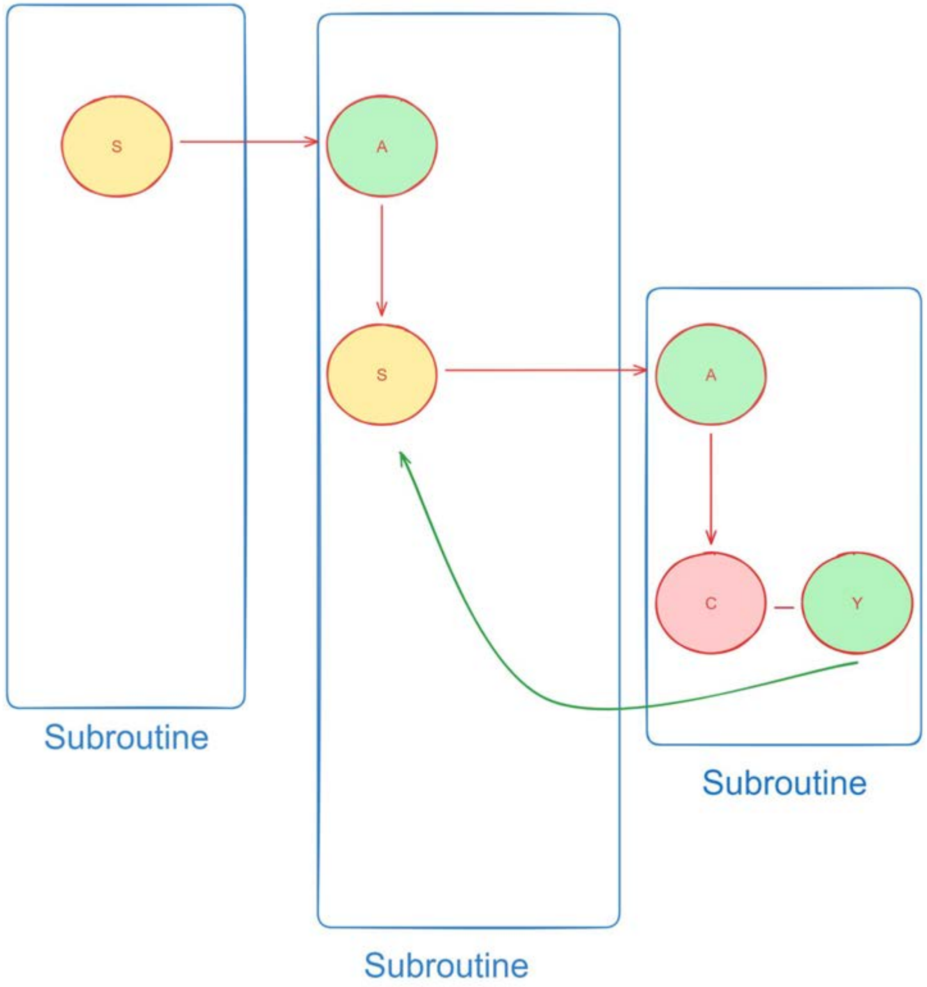
Brushing up on some Basics

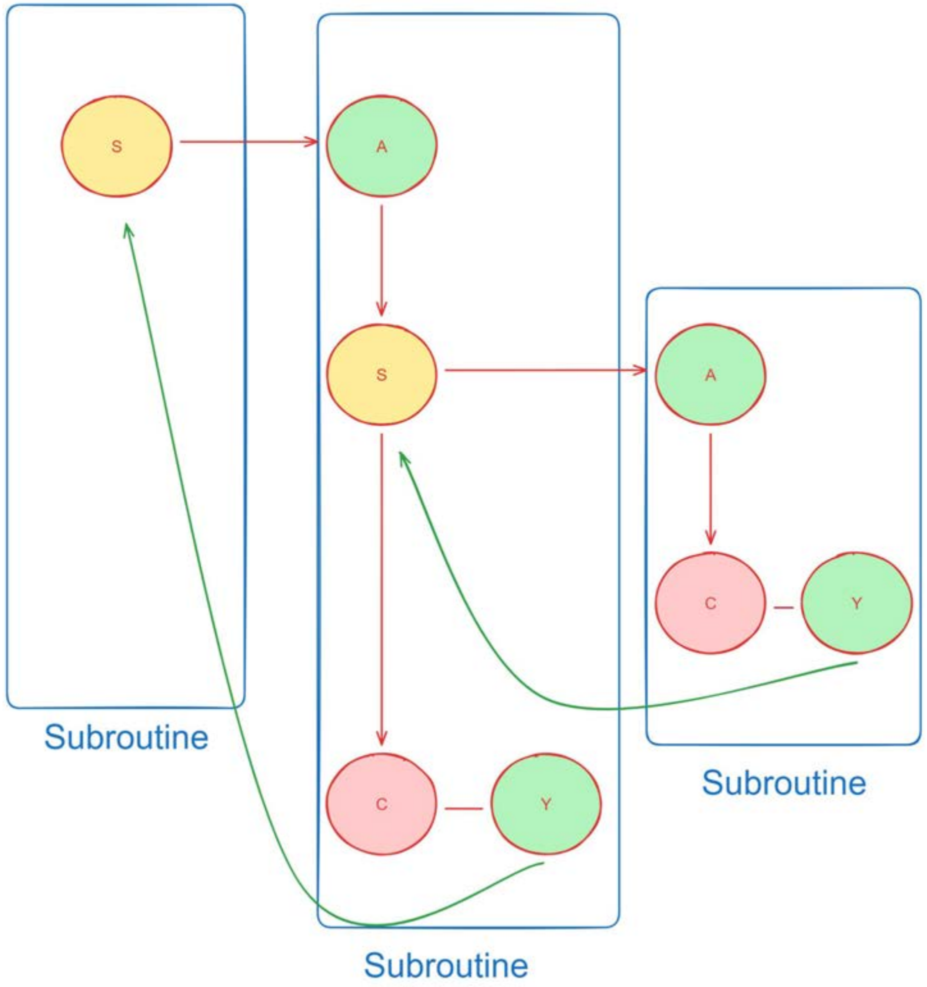


What are Subroutines?


S
Suspend
A
Run
C
Terminate
Y
Yield


S
A
S
A
Subroutine
Subroutine
Subroutine


S
A
S
A
C
Y
Subroutine
Subroutine
Subroutine


S
A
S
A
C
Y
C
Y
Subroutine
Subroutine
Subroutine

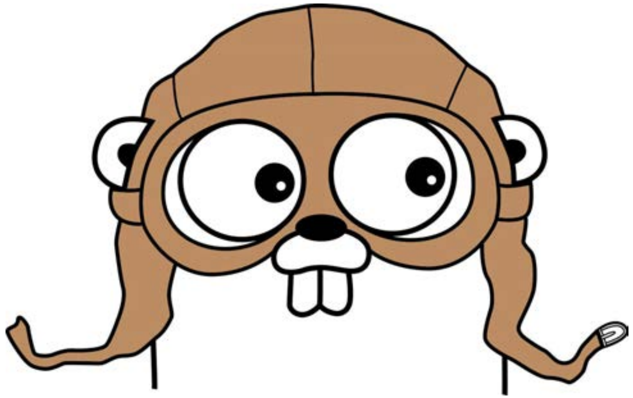
Eager and Closed

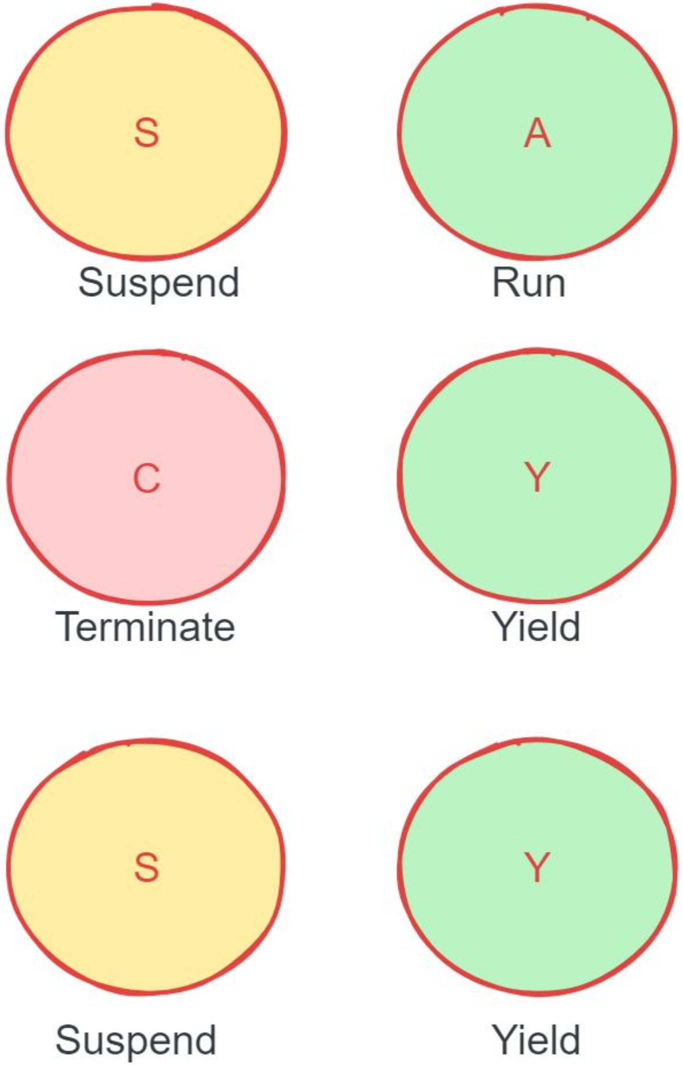
- **Eager**: Expression is evaluated as soon as it is encountered

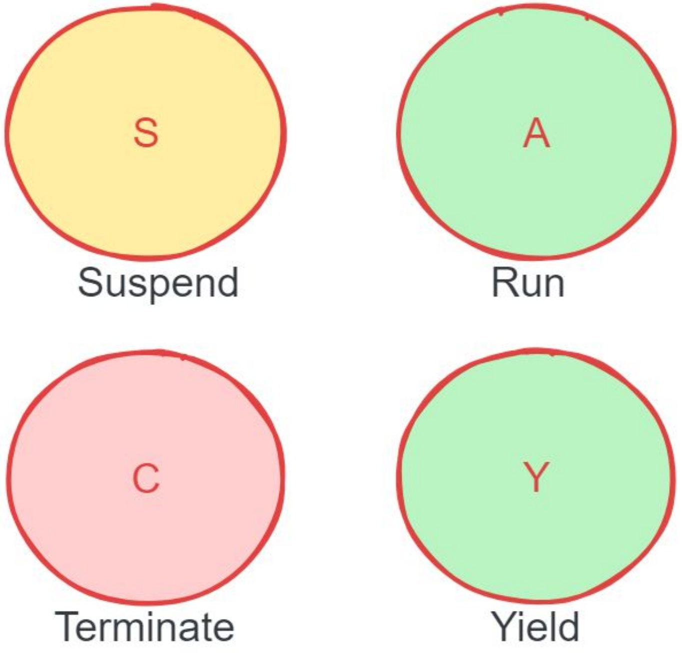
Eager and Closed

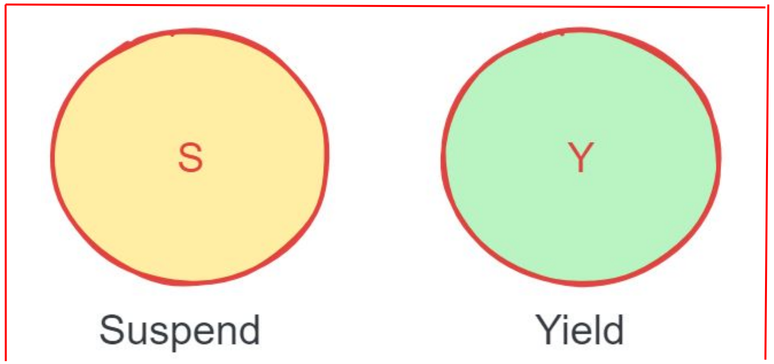
- **Eager**: Expression is evaluated as soon as it is encountered
- **Closed**: Only returns after it has evaluated the expression

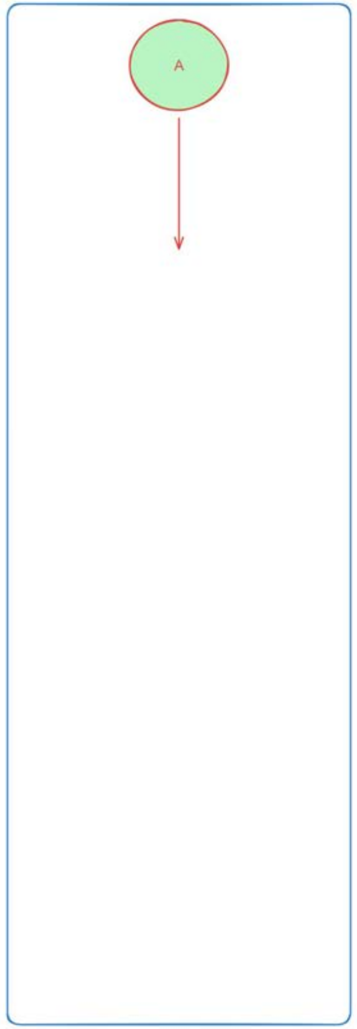
Coroutines as generalised Subroutines



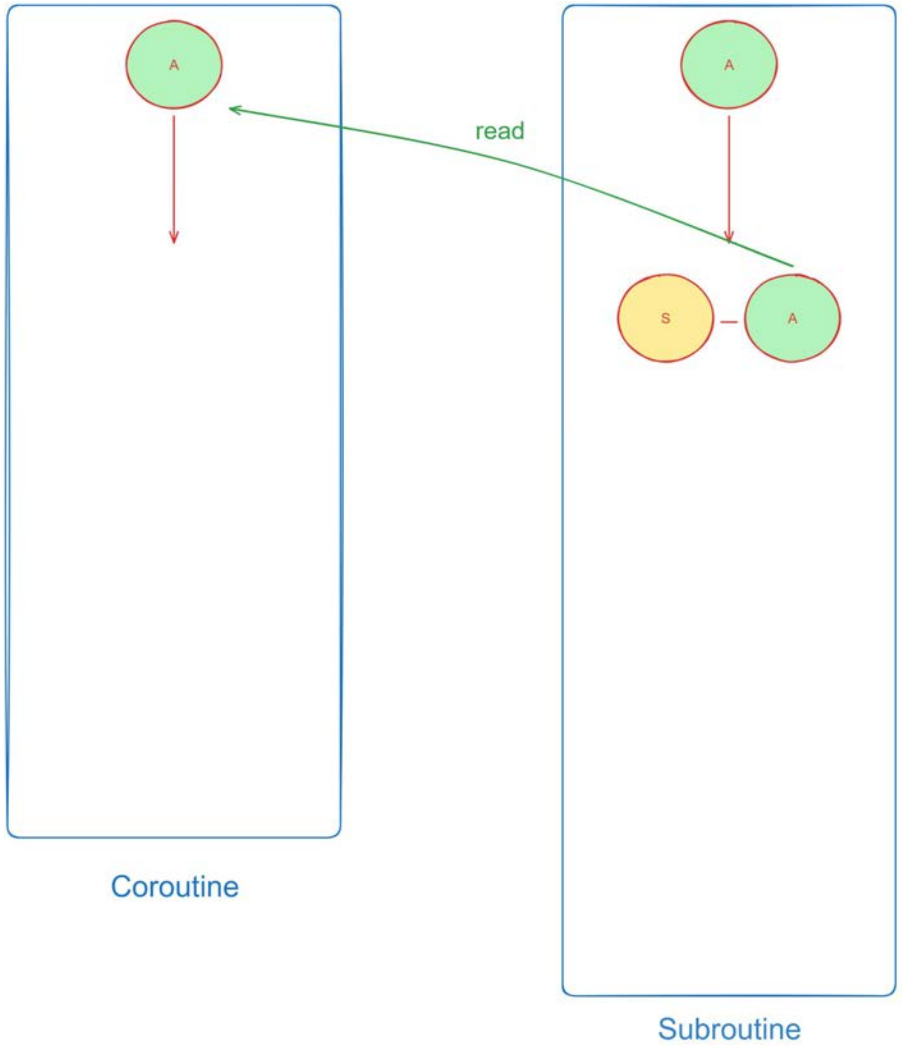

S
Suspend
A
Run
C
Terminate
Y
Yield
S
Suspend
Y
Yield

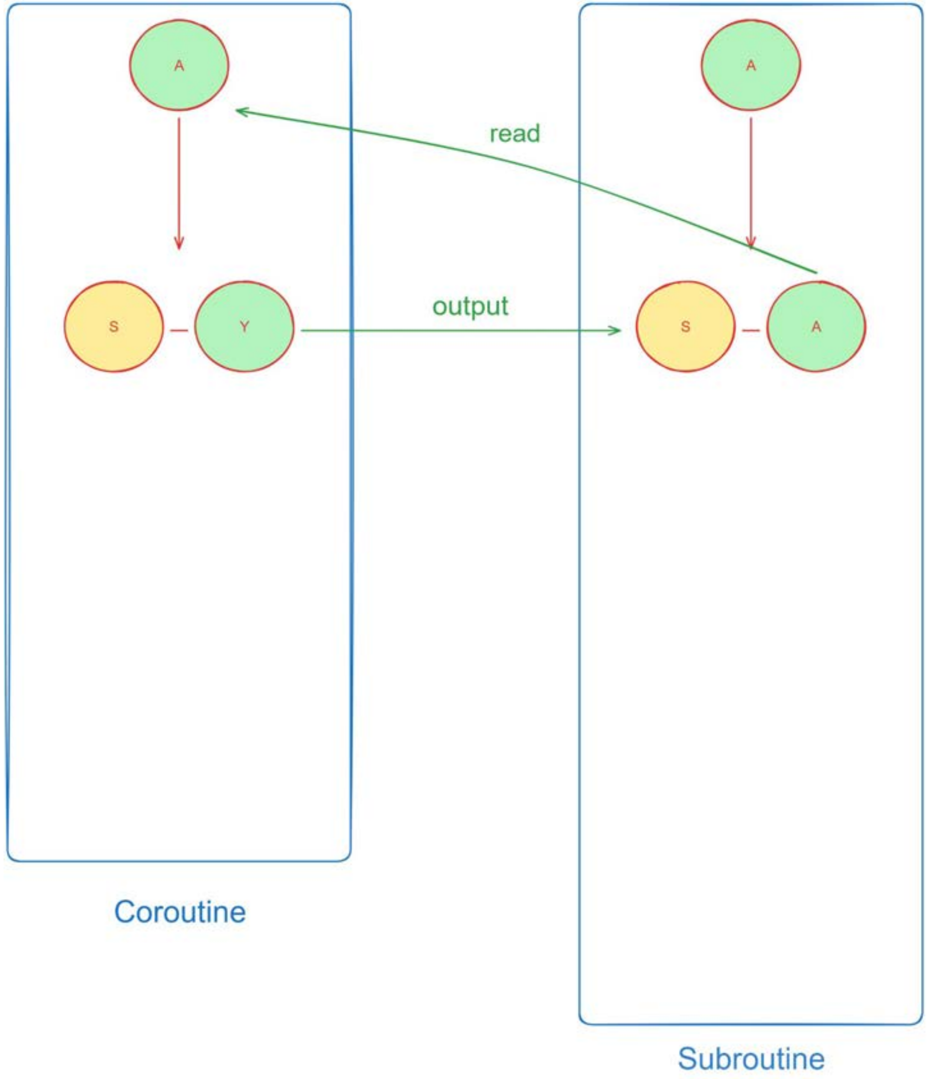

S
Suspend
A
Run
C
Terminate
Y
Yield

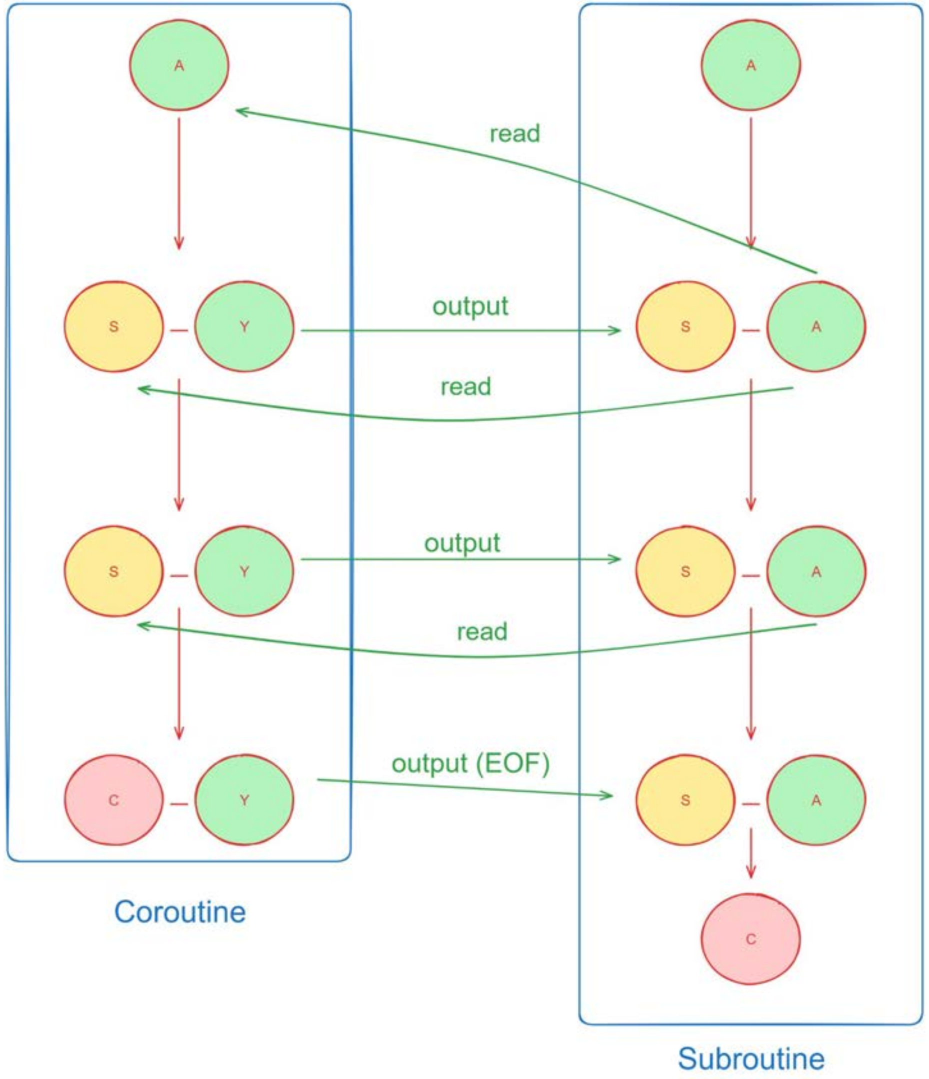

S
Suspend
Y
Yield



Subroutine


A
read
A
S
A
Coroutine
Subroutine

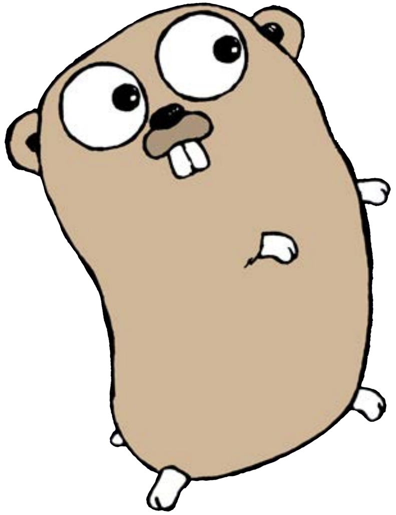

A
read
A
output
S
Y
S
A
Coroutine
Subroutine


A
read
S
Y
output
S
A
read
S
Y
output
S
A
read
C
Y
output (EOF)
S
A
C
Coroutine
Subroutine

Coroutines are like functions that **return multiple times** and keep their state

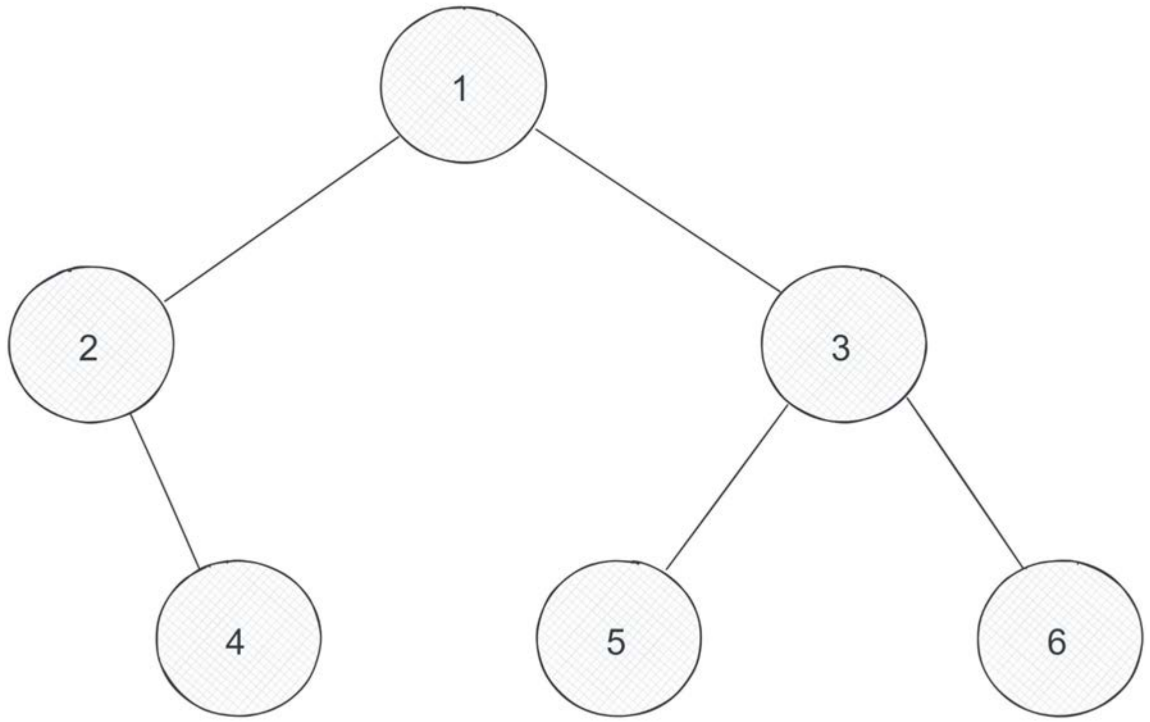
Coroutines are like functions that **return multiple times** and keep their state (which would include the values of local variables plus the command pointer)

Coroutines are like functions that **return multiple times** and keep their state (which would include the values of local variables plus the command pointer) so they can **resume** from where they **yielded**



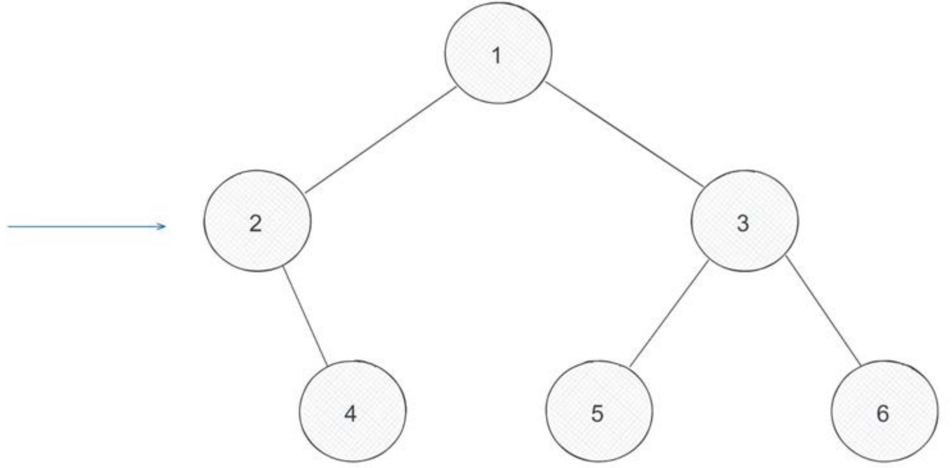
Let’s look at an example

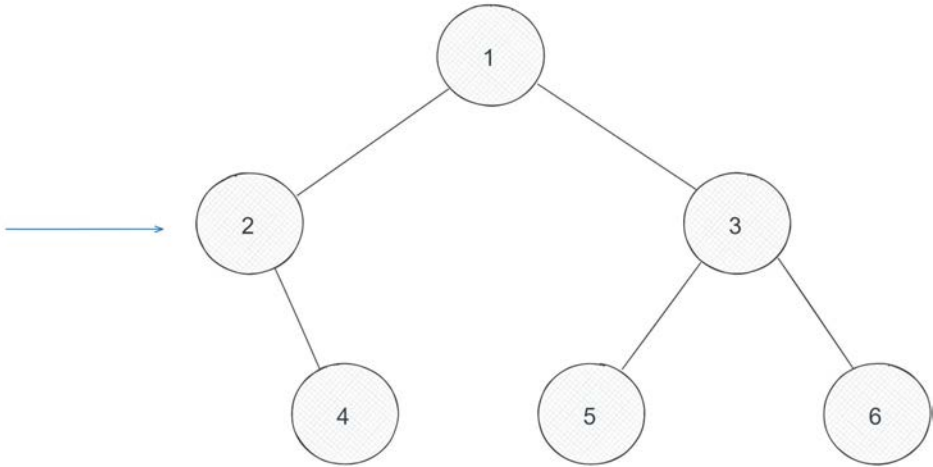
Comparing Binary Trees!

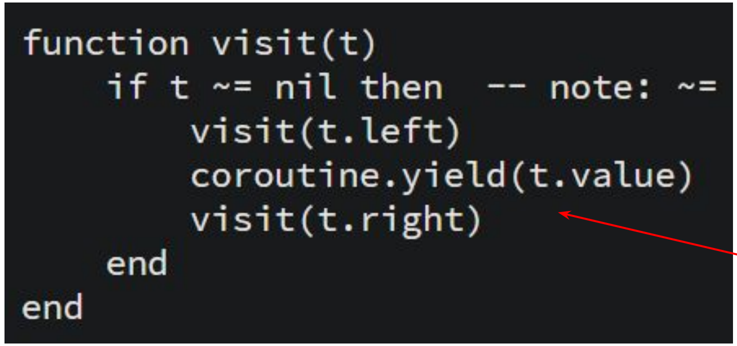


```lua
function visit(t)
    if t ~= nil then  -- note: ~=
        visit(t.left)
        coroutine.yield(t.value)
        visit(t.right)
    end
end
```

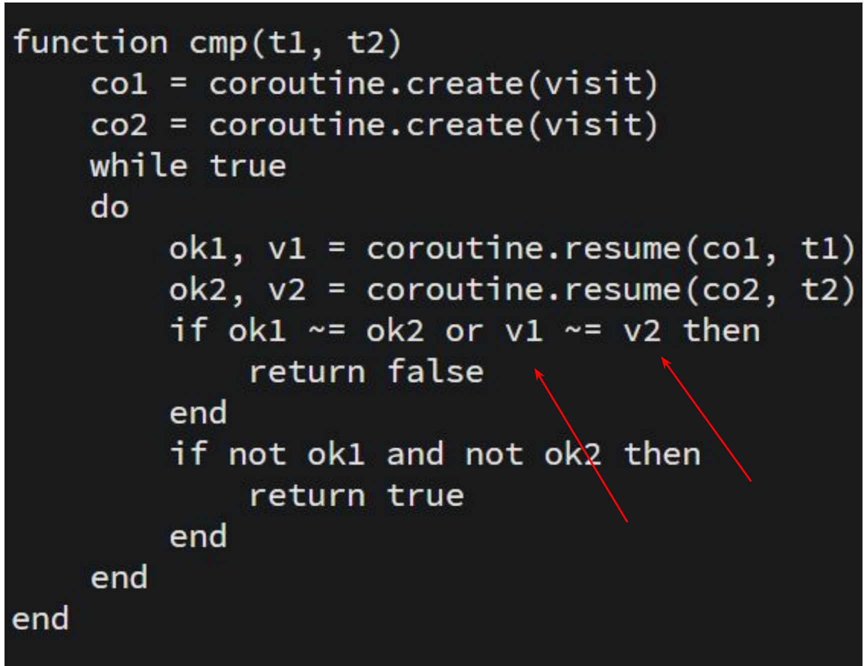
```lua
function cmp(t1, t2)
    co1 = coroutine.create(visit)
    co2 = coroutine.create(visit)
    while true
    do
        ok1, v1 = coroutine.resume(co1, t1)
        ok2, v2 = coroutine.resume(co2, t2)
        if ok1 ~= ok2 or v1 ~= v2 then
            return false
        end
        if not ok1 and not ok2 then
            return true
        end
    end
end
```



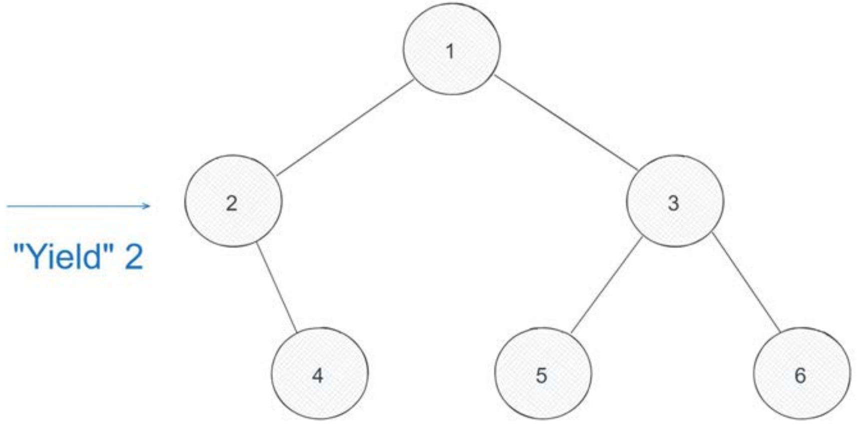


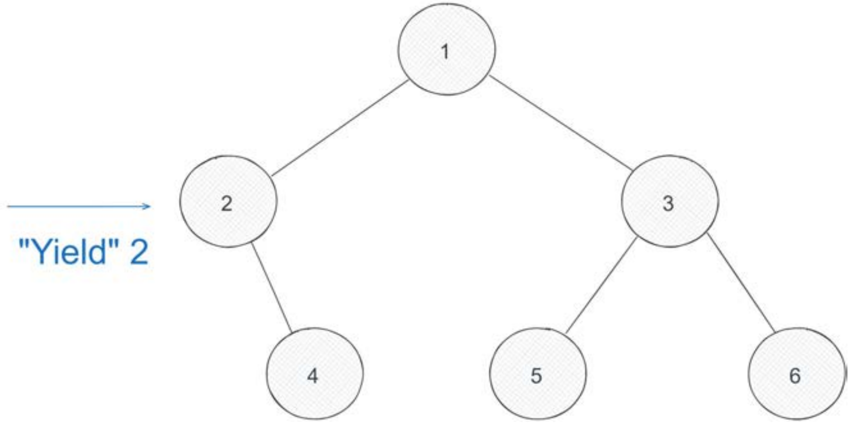


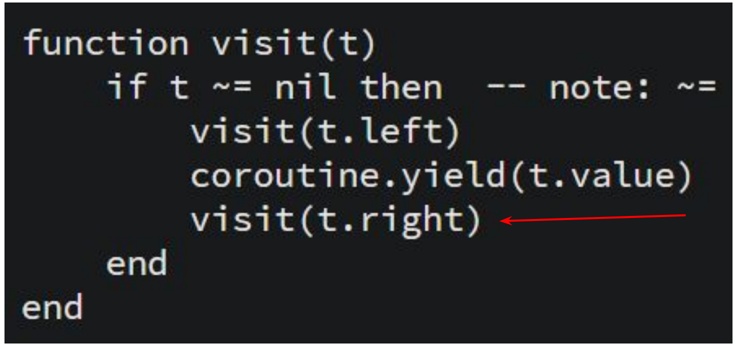
```
function visit(t)
    if t ~= nil then  -- note: ~=
        visit(t.left)
        coroutine.yield(t.value)
        visit(t.right)
    end
end
```



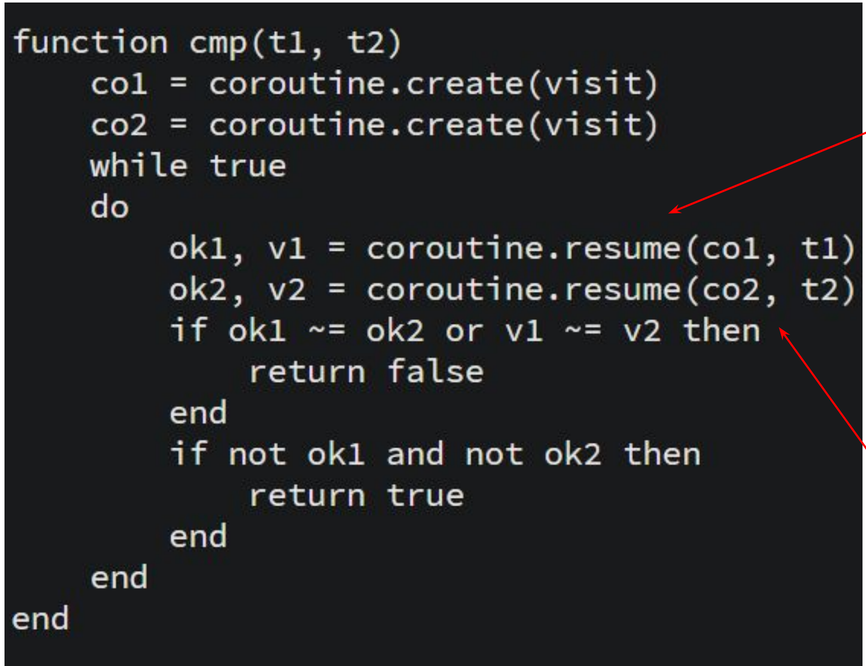
```
function cmp(t1, t2)
    co1 = coroutine.create(visit)
    co2 = coroutine.create(visit)
    while true
    do
        ok1, v1 = coroutine.resume(co1, t1)
        ok2, v2 = coroutine.resume(co2, t2)
        if ok1 ~= ok2 or v1 ~= v2 then
            return false
        end
        if not ok1 and not ok2 then
            return true
        end
    end
end
```



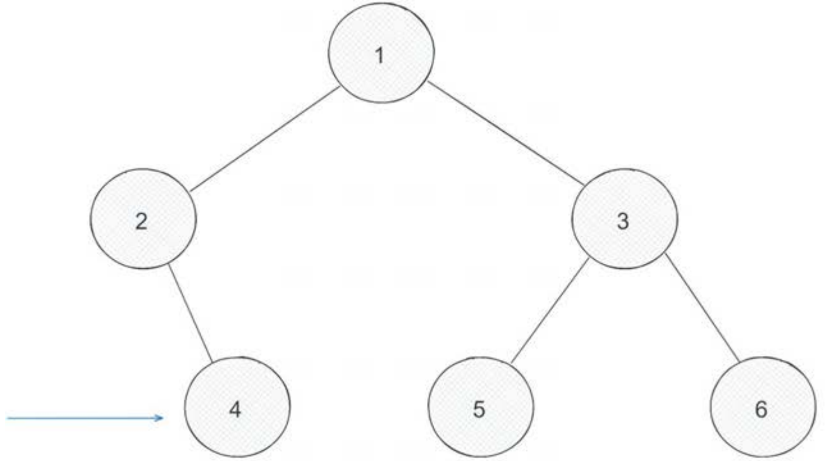


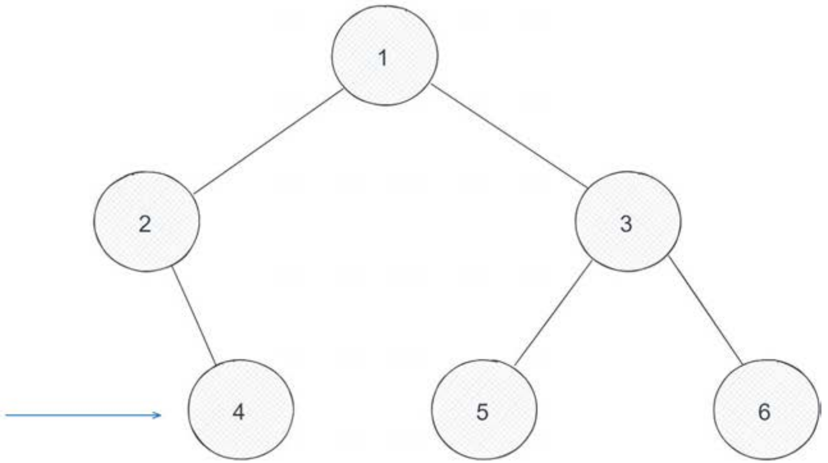


```lua
function visit(t)
    if t ~= nil then  -- note: ~=
        visit(t.left)
        coroutine.yield(t.value)
        visit(t.right)
    end
end
```

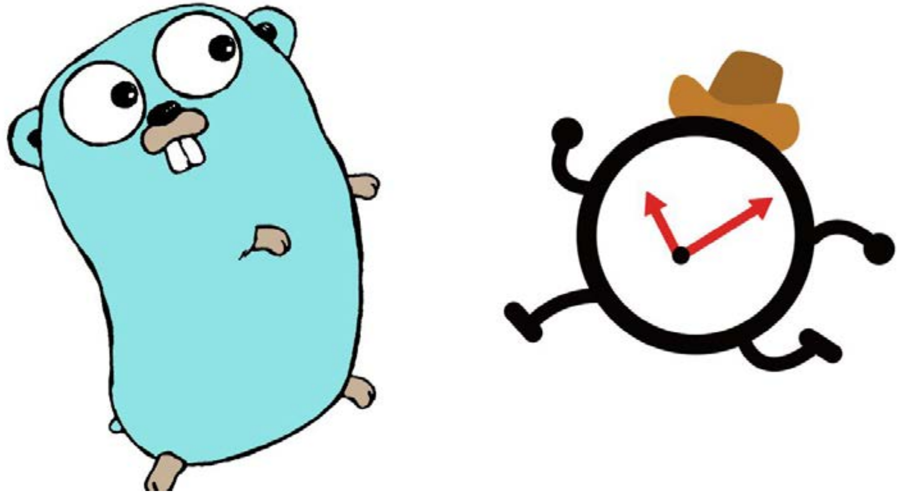


```lua
function cmp(t1, t2)
    co1 = coroutine.create(visit)
    co2 = coroutine.create(visit)
    while true
    do
        ok1, v1 = coroutine.resume(co1, t1)
        ok2, v2 = coroutine.resume(co2, t2)
        if ok1 ~= ok2 or v1 ~= v2 then
            return false
        end
        if not ok1 and not ok2 then
            return true
        end
    end
end
```



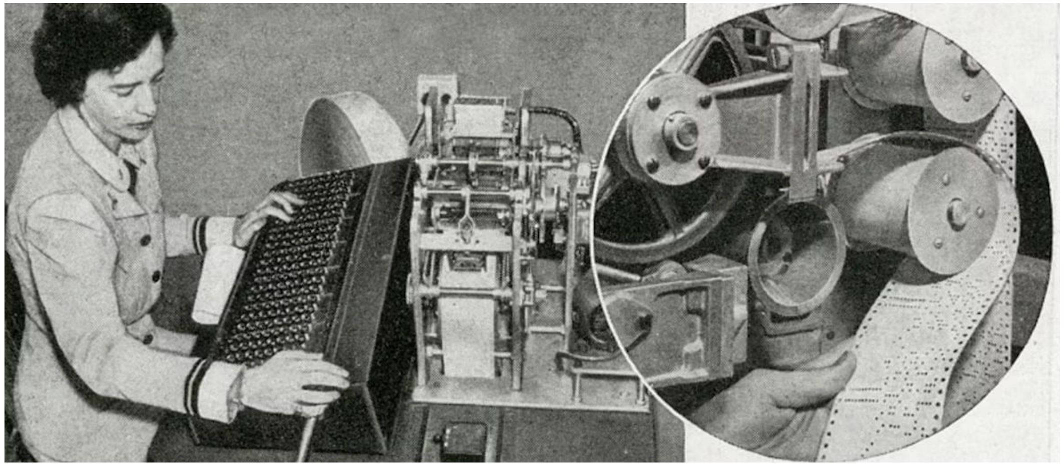


Let’s go back in time



It’s 1958 …

- You want to compile your COBOL program in the modern nine-path COBOL compiler

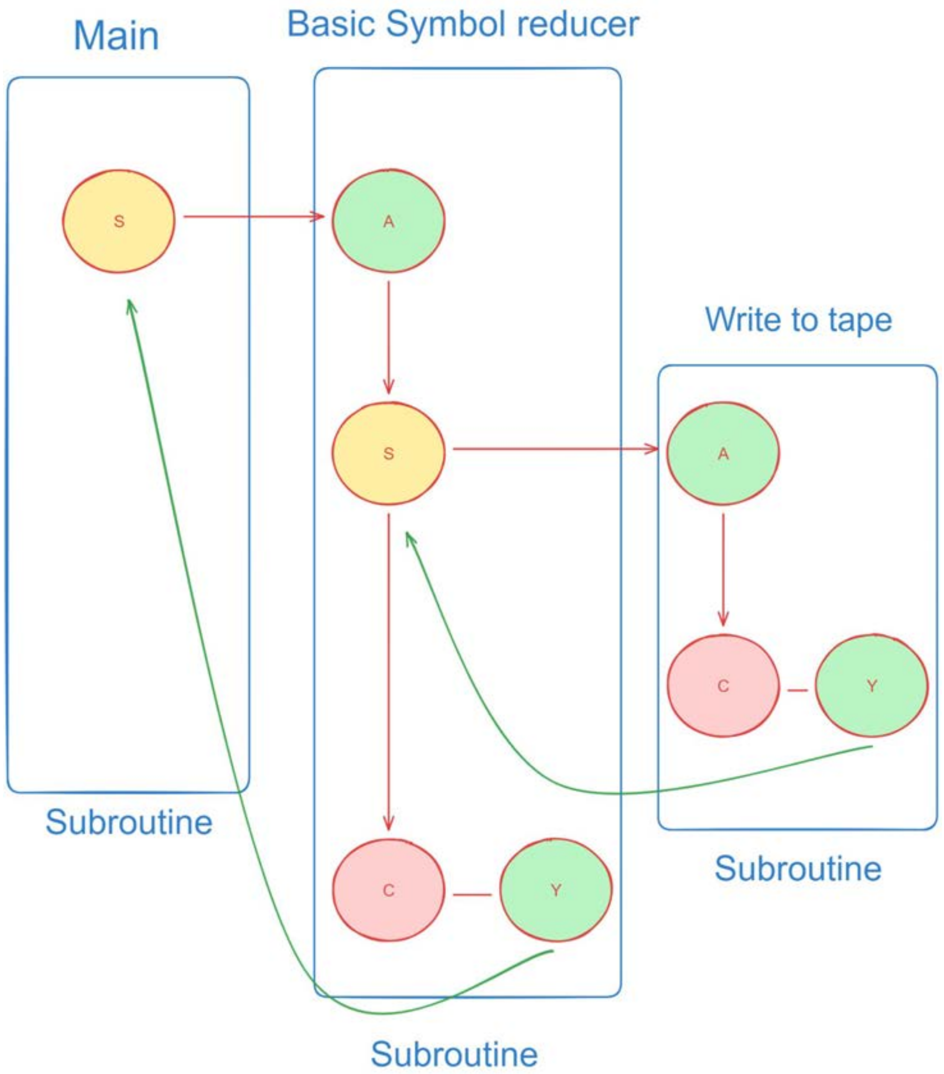


It’s 1958 …

- You want to compile your COBOL program in the modern nine-path COBOL compiler
- You take your main program punched-card, pass it to the **Basic Symbol Reducer** which will eat the punched card, and it will spew the **tokens onto the tape**

It’s 1958 ...

- You want to compile your COBOL program in the modern nine-path COBOL compiler
- You take your main program punched-card, pass it to the **Basic Symbol Reducer** which will eat the punched card, and it will spew the **tokens onto the tape**
- It then goes back to the **main** routine, which calls the **Name Reducer** (Name Lookup today) which puts its output in the **next tape**



Main
Basic Symbol reducer
Write to tape
S
A
S
A
C
Y
C
Y
Subroutine
Subroutine
Subroutine

It’s 1958 …

- And this keeps going till you have the result of the execution and a bunch of extra tapes that you don’t need anymore.

It’s 1958 …

- Conway thought there had to be a better way to pass a token from a lexer to the parser without all this expensive piece of machinery

It’s 1958 …

- Subroutines were just a **special case** of more generalised coroutines, that didn’t need to write on tape

It’s 1958 …

- Subroutines were just a **special case** of more generalised coroutines, that didn’t need to write on tape

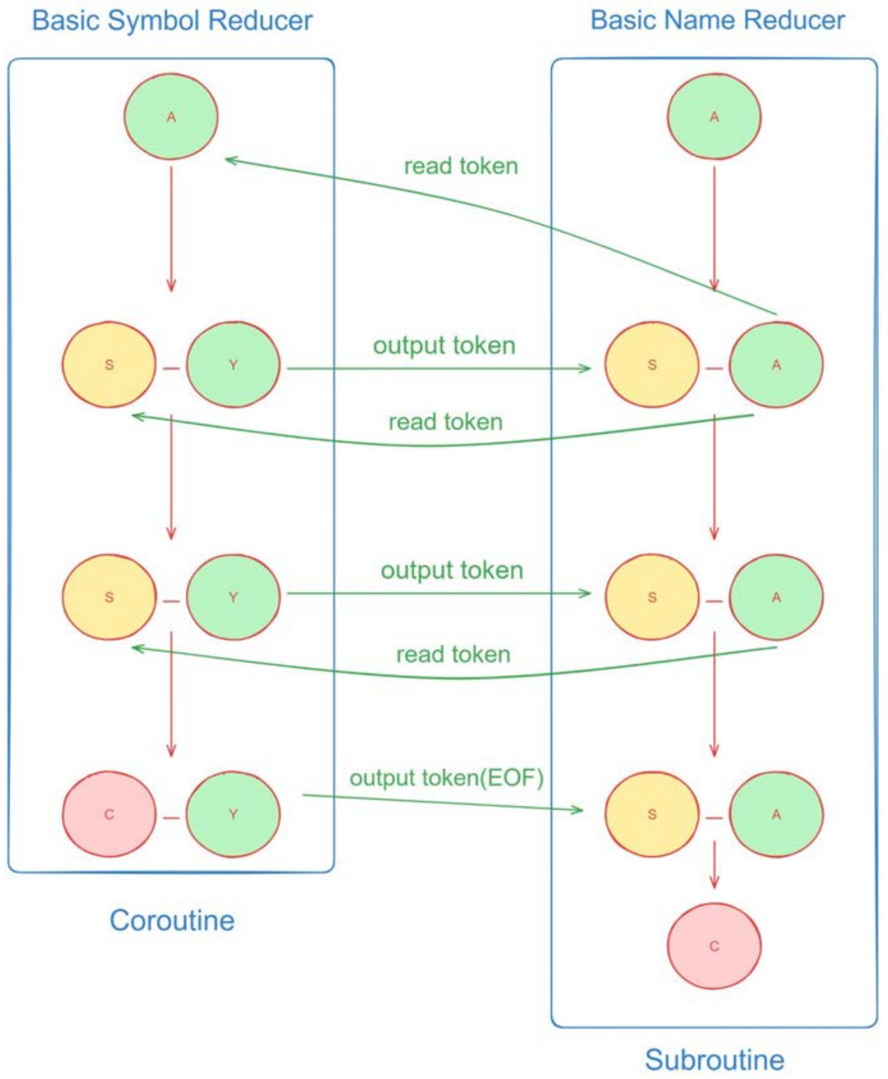
  (ie, they didn’t need to “return”)

It’s 1958 ...

- Subroutines were just a **special case** of more generalised coroutines, that didn’t need to write on tape

  (ie, they didn’t need to “return”)

- Instead pass the information more directly, **bypassing** this “machinery”



Basic Symbol Reducer
Basic Name Reducer
A
A
read token
S
Y
output token
S
A
read token
S
Y
output token
S
A
read token
C
Y
output token(EOF)
S
A
C
Coroutine
Subroutine

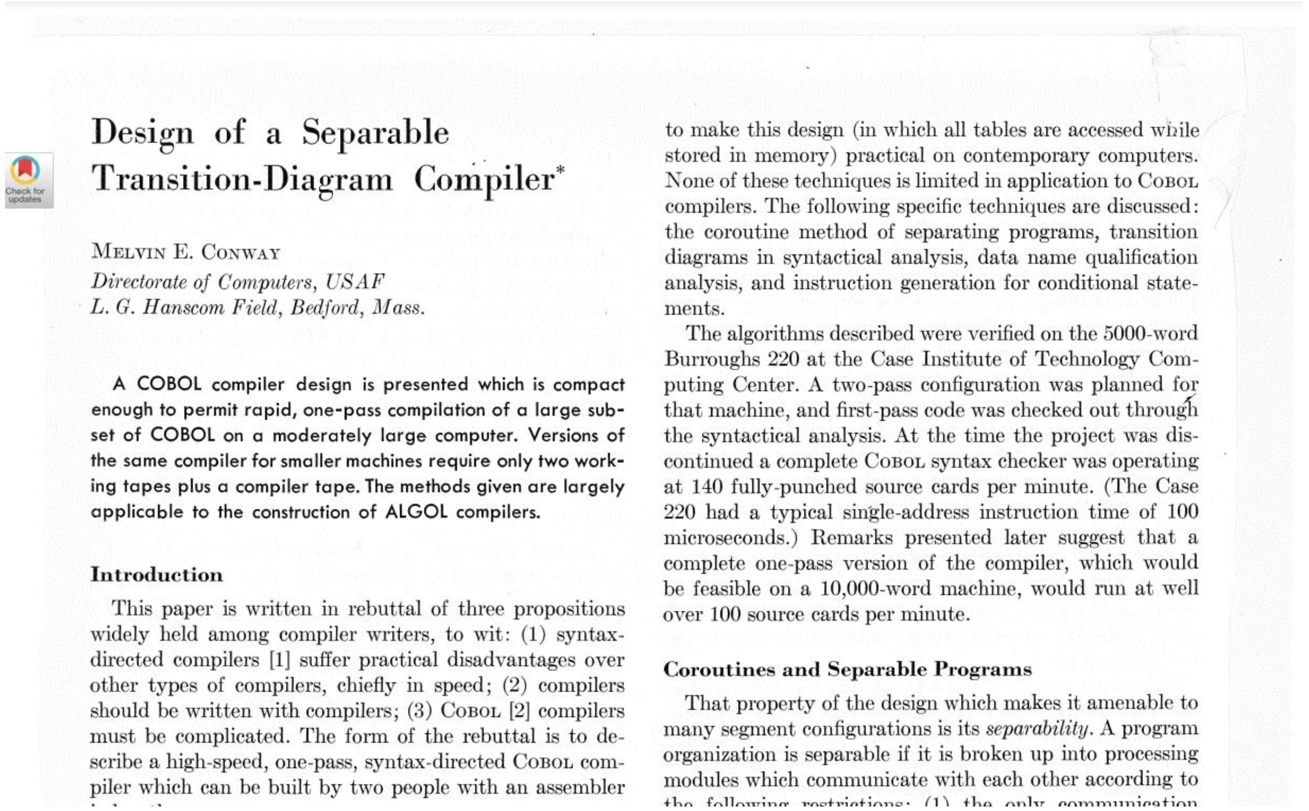
It’s 1958 …

- This way, **raising** the level of abstraction, actually led to a **less costly** control structure, leading to the **one-pass** COBOL compiler.

It’s 1958 …

“Negative Cost Abstraction”

Side note - The paper that coined the term “Coroutines”



Design of a Separable Transition-Diagram Compiler*

Melvin E. Conway
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) Cobol [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed Cobol compiler which can be built by two people with an assembler

to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to Cobol compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete Cobol syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to

So, where are Coroutines now?

- Considering all we’ve talked about so far, coroutines should have been a common pattern that is provided by most languages.

- Considering all we’ve talked about so far, coroutines should have been a common pattern that is provided by most languages.
- But with rare exceptions such as Simula, few languages do

- Considering all we’ve talked about so far, coroutines should have been a common pattern that is provided by most languages.
- But with rare exceptions such as Simula, few languages do, and those that do, generally provide **limited variants** of coroutines, (we discuss this a little later)

Problems with Coroutines

- A lack of a uniform view of this concept

Problems with Coroutines

- A lack of a uniform view of this concept
- No precise definitions for it

Problems with Coroutines

- Another reason why coroutines are not provided as a facility in most mainstream languages was the advent of **Algol-60**

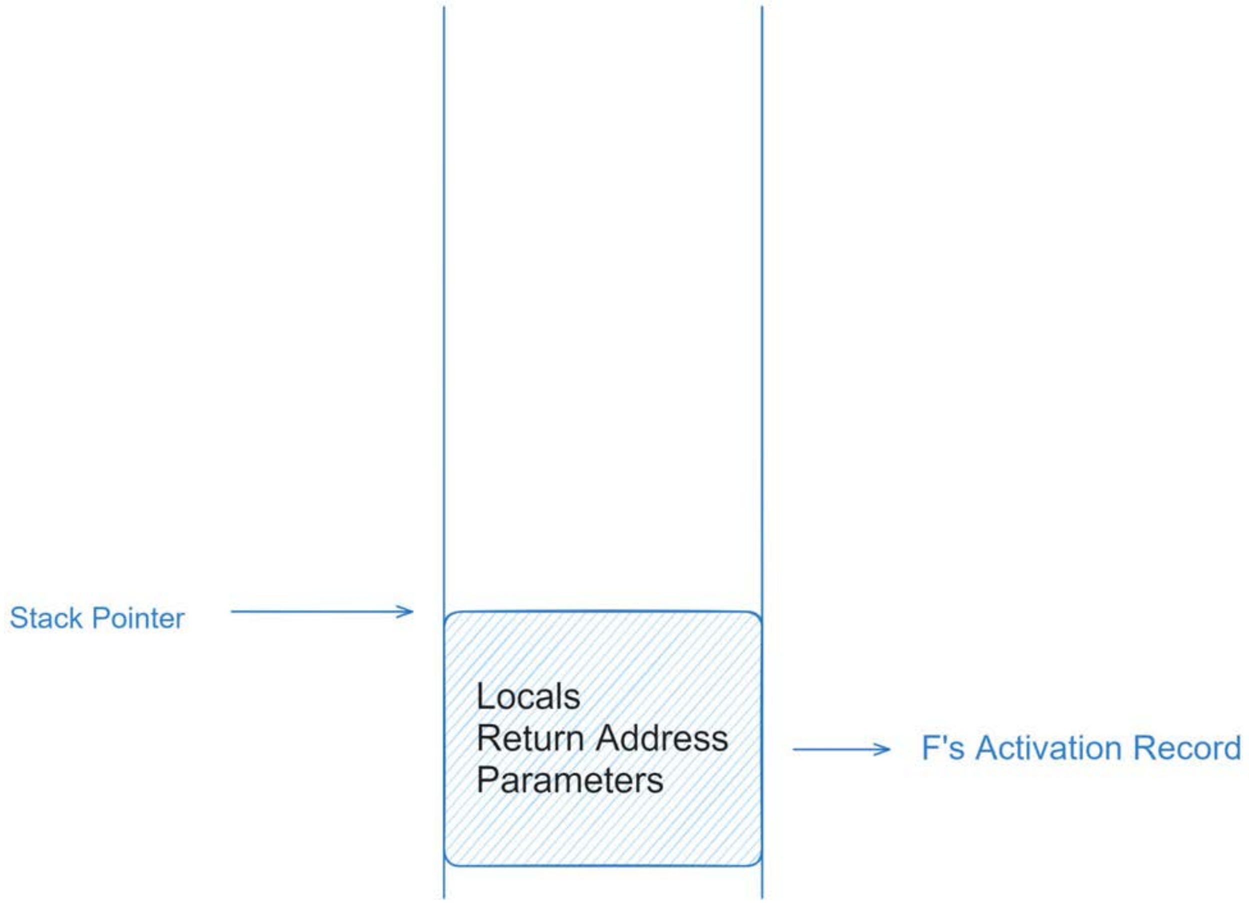
Problems with Coroutines

- Another reason why coroutines are not provided as a facility in most mainstream languages was the advent of **Algol-60**
- And with it, **block scoped variables**

Problems with Coroutines

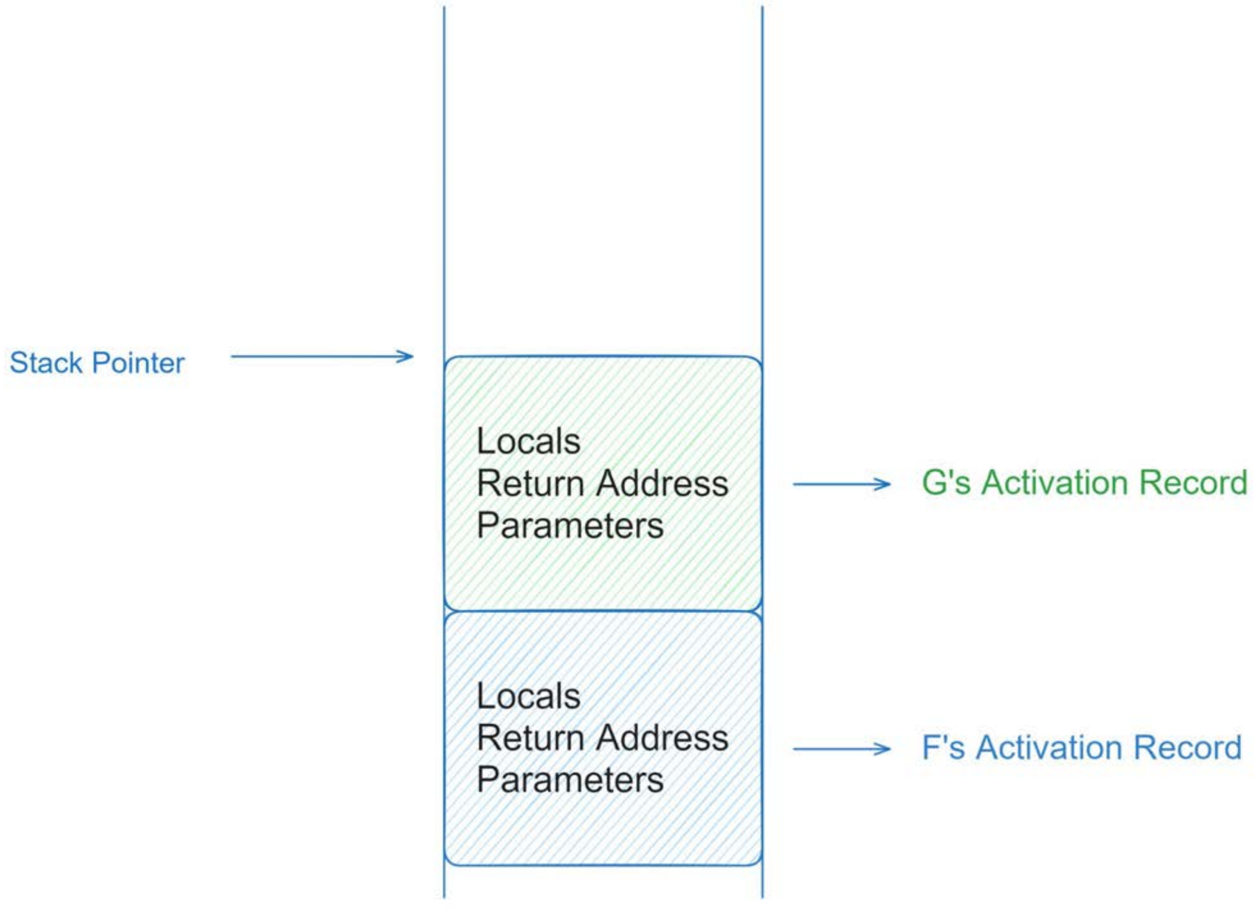
- Another reason why coroutines are not provided as a facility in most mainstream languages was the advent of **Algol-60**
- And with it, **block scoped variables**, you no longer had parameters and return values stored as global memory, but rather **relative to a stack pointer**

Normal Functions



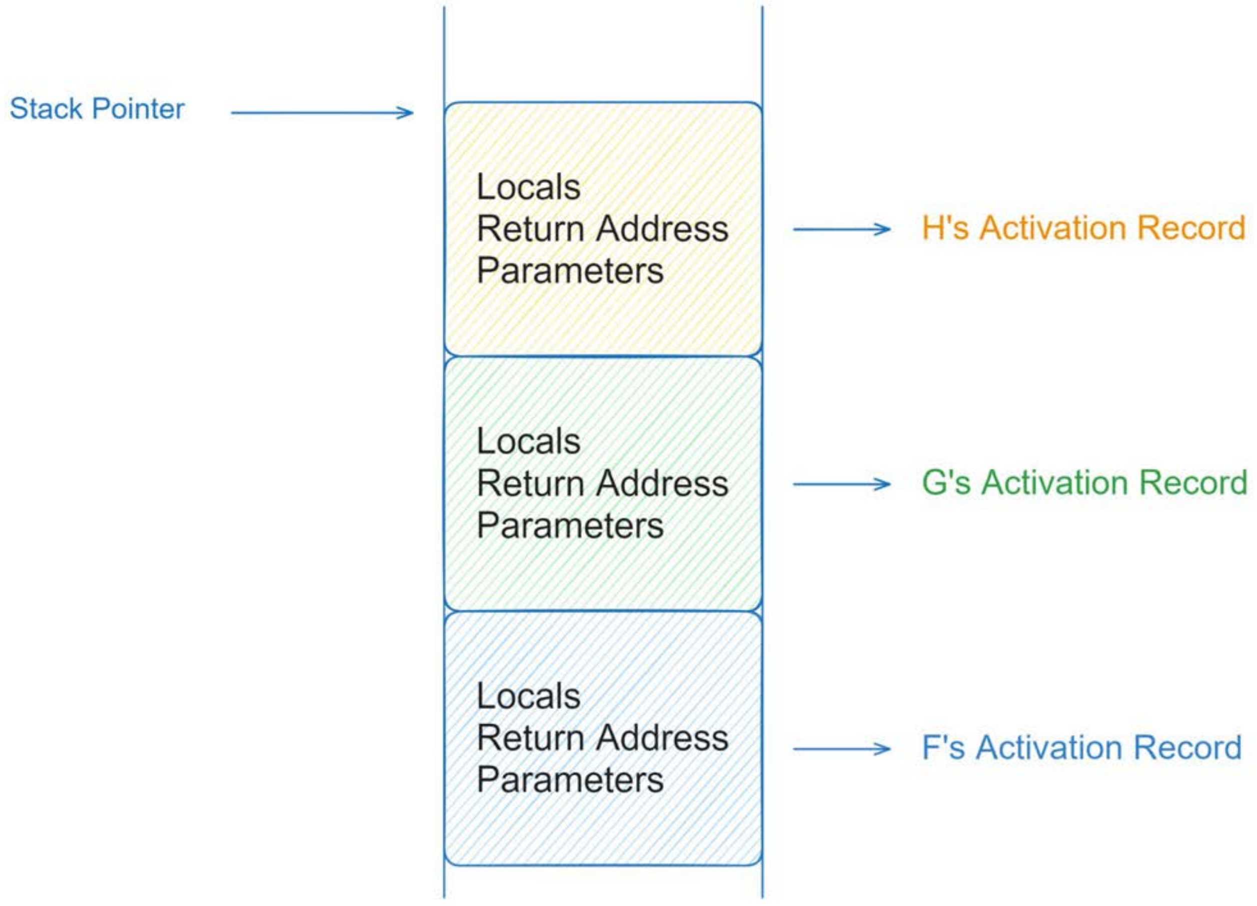
Thread Stack

Normal Functions

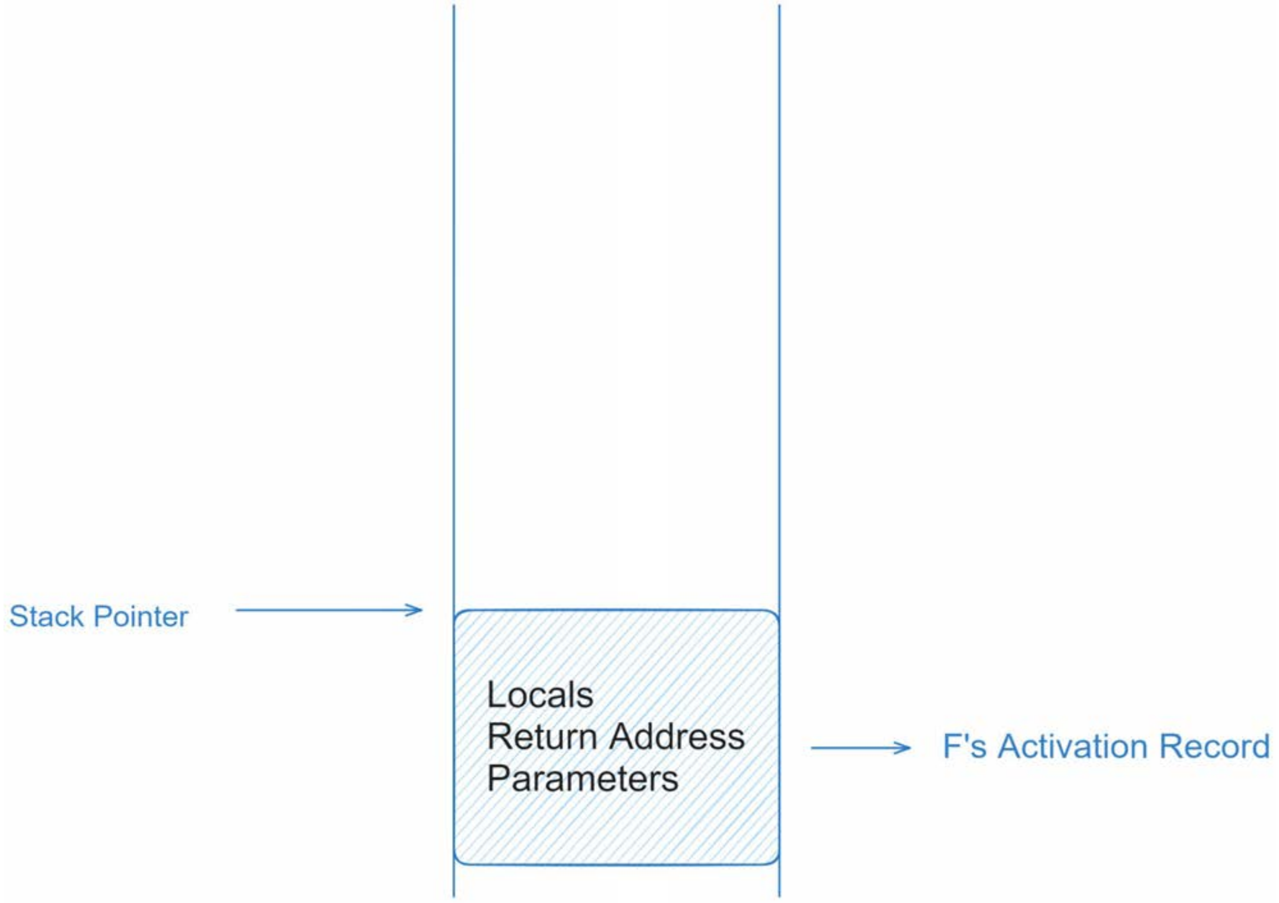


Thread Stack

Normal Functions

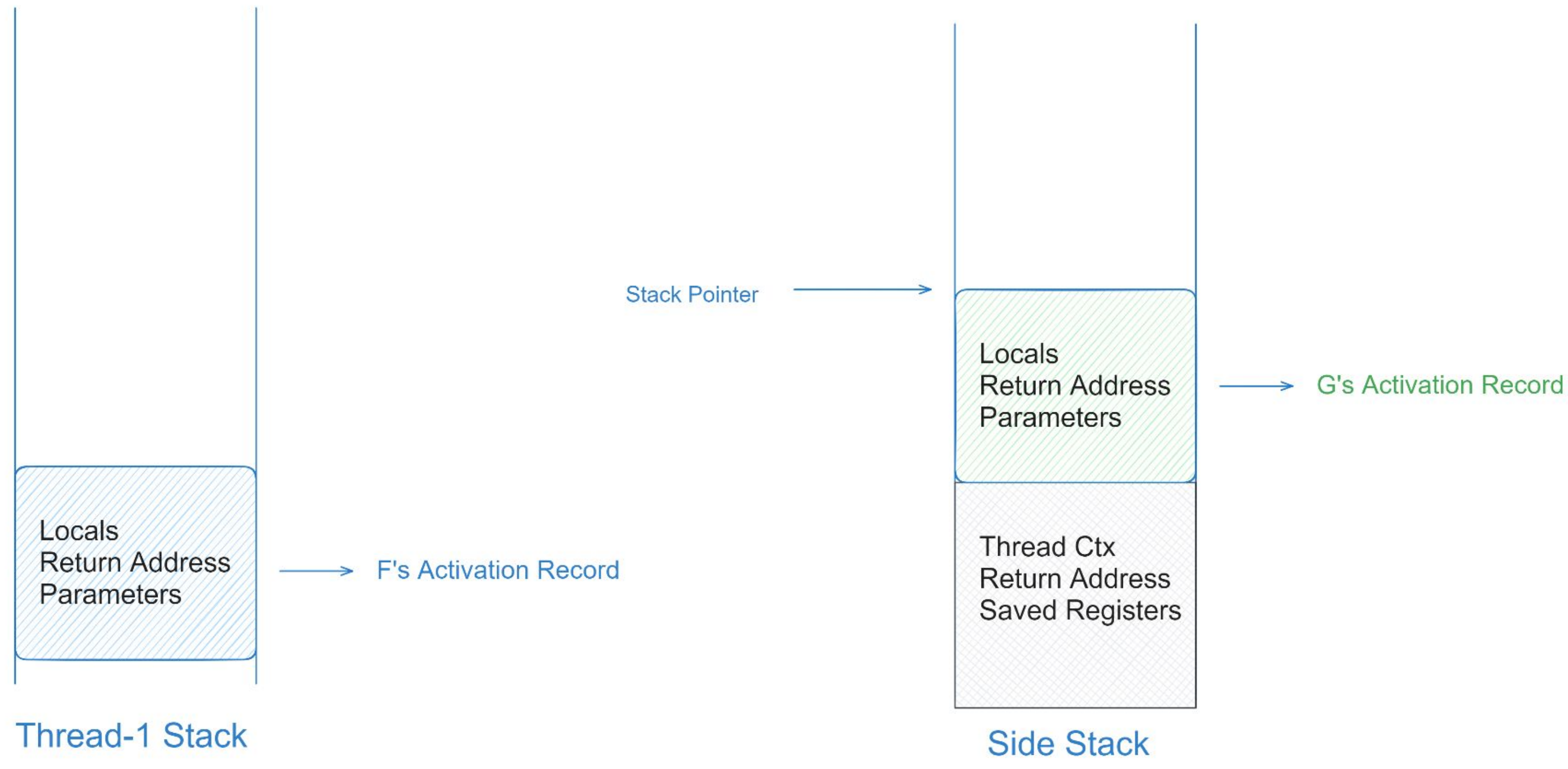


Thread Stack

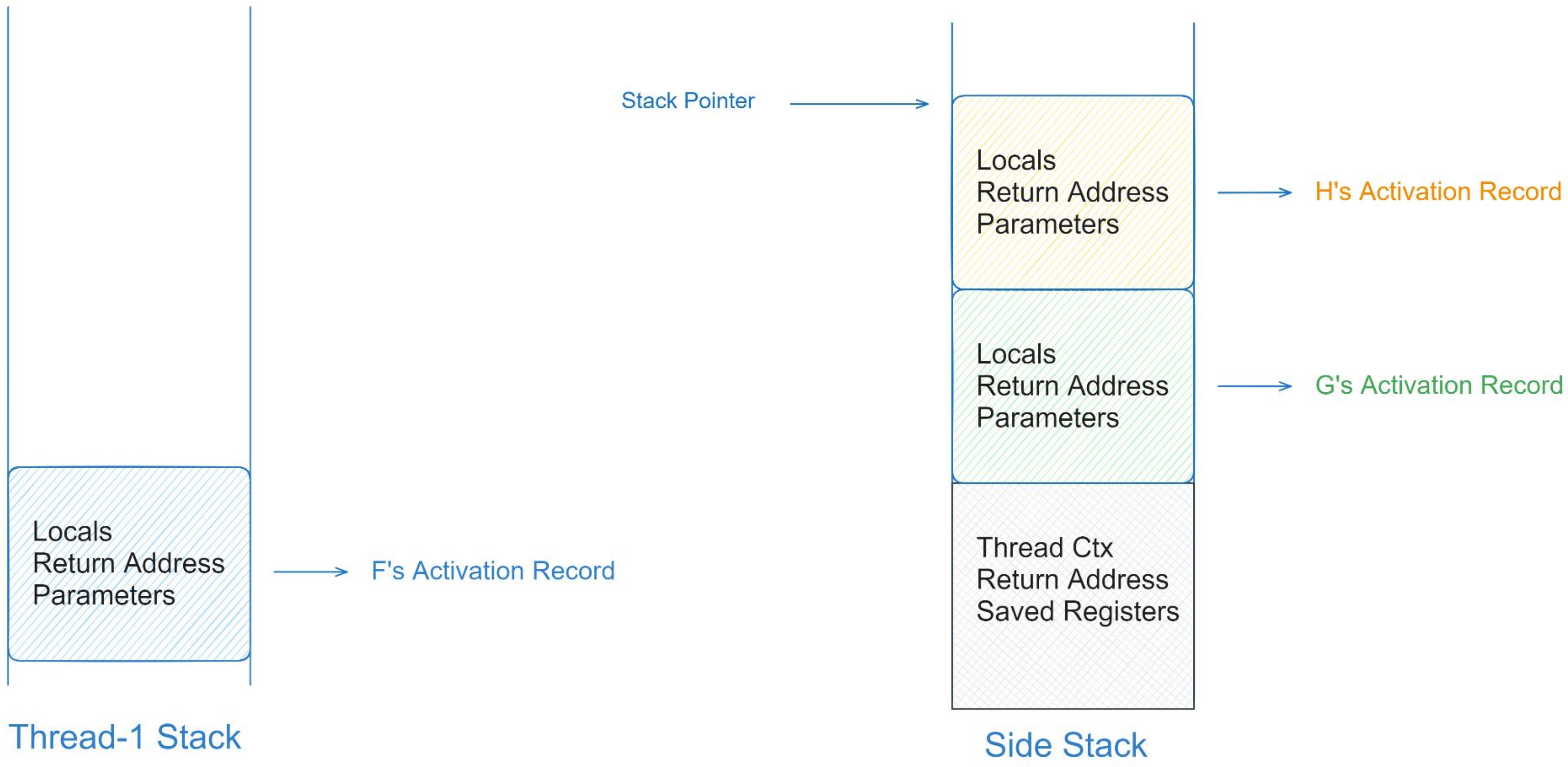


Thread-1 Stack

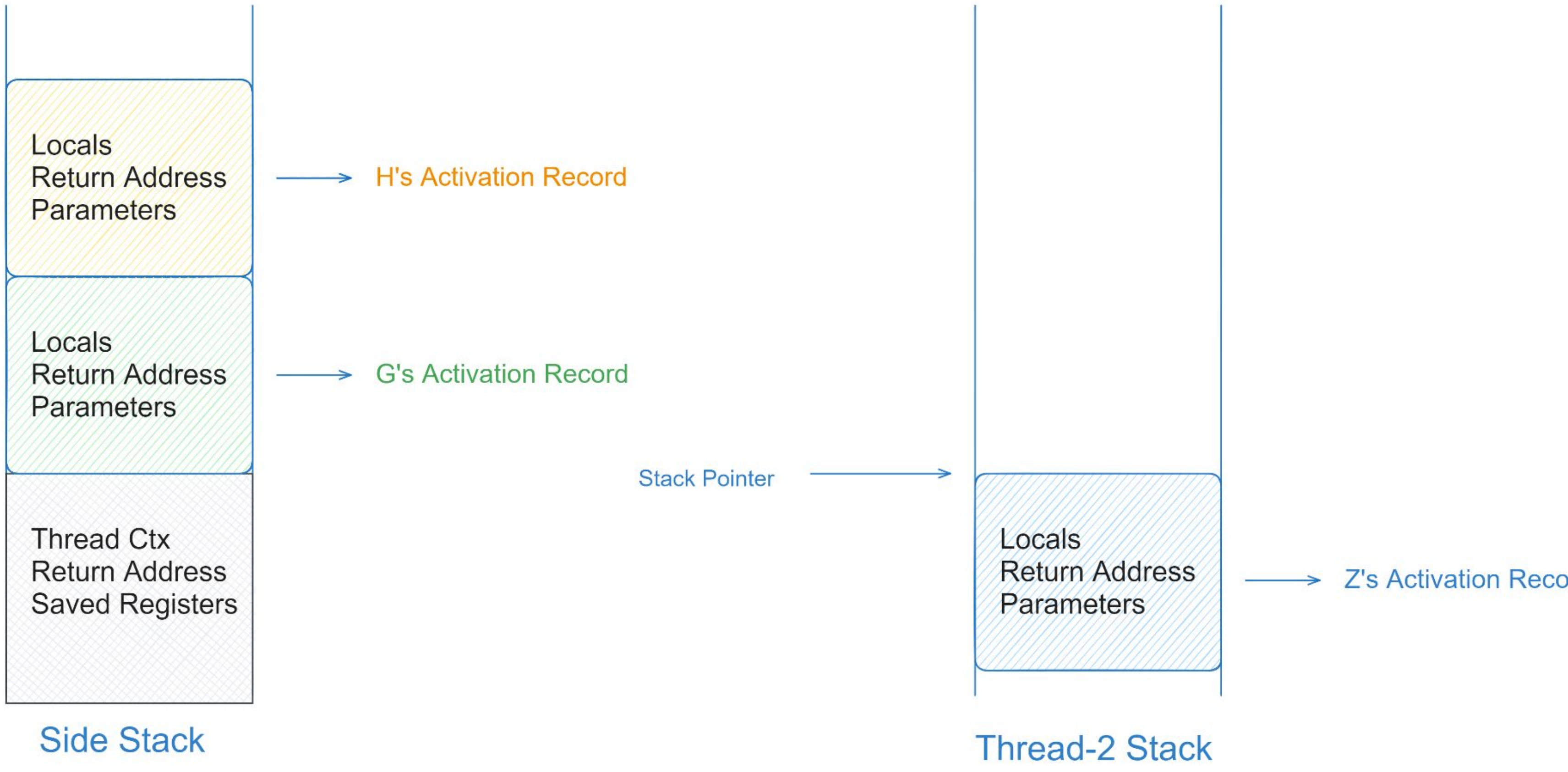
Coroutines using side stacks



Coroutines using side stacks



Coroutines using side stacks



Problems with Coroutines

- This almost mimics **heavy multithreading** and increases memory footprint, rather than being a **cheap abstraction** like a function that a coroutine is meant to be.

Quickly, let’s look at the fundamental characteristics of a Coroutine

Characteristics

Marlin’s doctoral thesis, widely acknowledged as a reference for this mechanism, summarizes -

Characteristics

Marlin’s doctoral thesis, widely acknowledged as a reference for this mechanism, summarizes -

- “The values of data local to a coroutine **persist between successive calls**”

Characteristics

Marlin's doctoral thesis, widely acknowledged as a reference for this mechanism, summarizes -

- "The values of data local to a coroutine **persist between successive calls**"
- "The execution of a coroutine is **suspended** as control leaves it, only to carry on where it left off when **control re-enters** the coroutine at some later stage."

Now that we have the basics and the history out of the way

… and hopefully, have made a case for its usefulness let’s build up to what a coroutine can look like in Go

Classifying Coroutines

I promise this is relevant, please bear with me

Classifying Coroutines

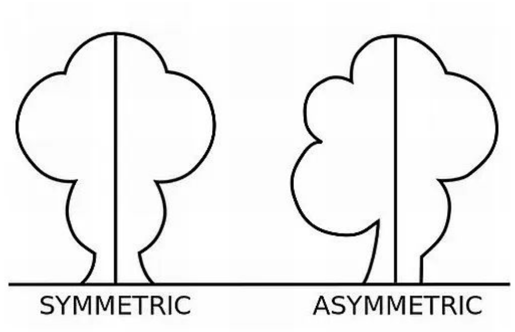
- By doing this we will see what we mean by a “**Full Coroutine**”

Classifying Coroutines

- By doing this we will see what we mean by a “**Full Coroutine**”
- And how some languages like Python and Kotlin don’t actually provide this

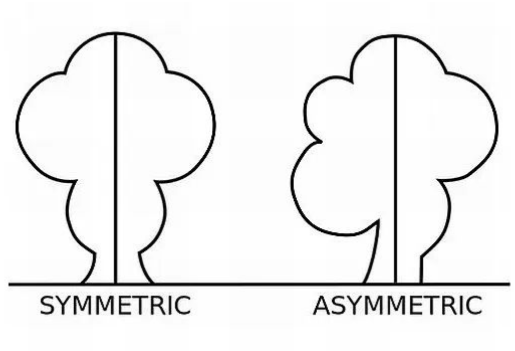
Control Transfer Mechanism - Asymmetric, Symmetric

- **Symmetric** coroutines provide a single control-transfer operation that allows coroutines to **explicitly pass control** among themselves.



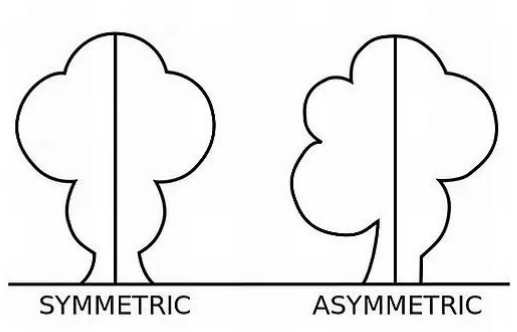
Control Transfer Mechanism - Asymmetric, Symmetric

- **Symmetric** coroutines provide a single control-transfer operation that allows coroutines to **explicitly pass control** among themselves.
- **Asymmetric** coroutine mechanisms provide two control-transfer operations:



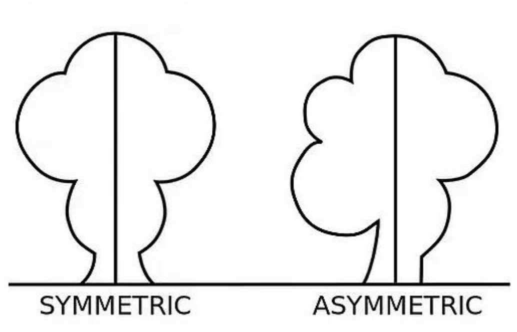
Control Transfer Mechanism - Asymmetric, Symmetric

- One for **invoking** a coroutine and one for **suspending** it, the latter returning control to the coroutine invoker.



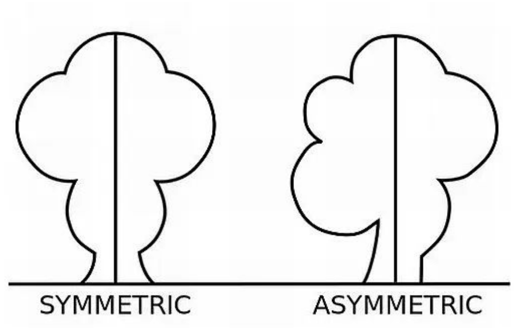
Control Transfer Mechanism - Asymmetric, Symmetric

- Coroutine mechanisms that support concurrent programming usually provide symmetric coroutines



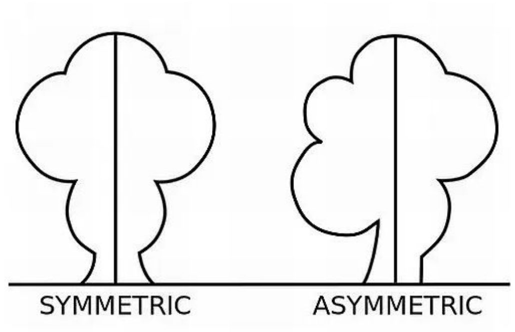
Control Transfer Mechanism - Asymmetric, Symmetric

- Coroutine mechanisms that support concurrent programming usually provide symmetric coroutines
- On the other hand, coroutine mechanisms intended for constructs that produce **sequences of values** typically provide asymmetric coroutines



Control Transfer Mechanism - Asymmetric, Symmetric

- But symmetric coroutines can be implemented **using asymmetric** coroutines that are **easier to write and maintain**.



First-Class versus Constrained Coroutines

- A coroutine mechanism provided as **first-class objects** that are fully programmable has a huge influence on its **expressive power**.

First-Class versus Constrained Coroutines

- A coroutine mechanism provided as **first-class objects** that are fully programmable has a huge influence on its **expressive power**.
- Coroutine objects that are **constrained** within language bounds cannot be directly manipulated by the programmer.

First-Class versus Constrained Coroutines

Appear in an expression

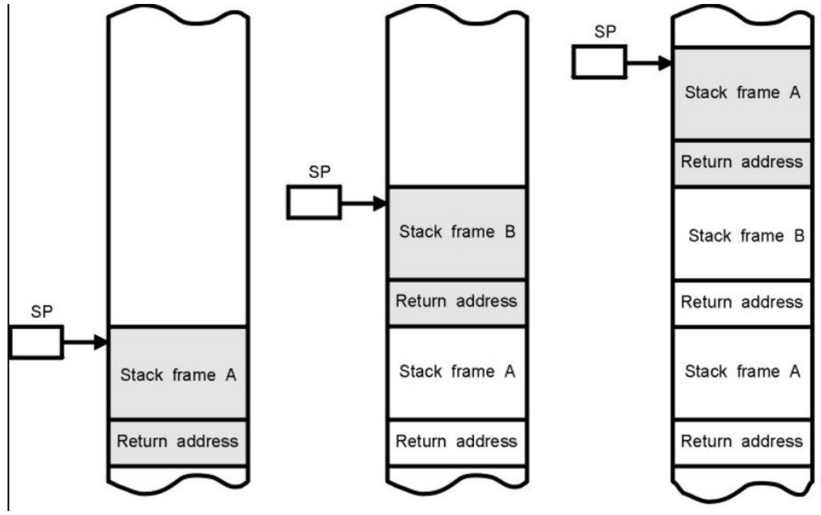
Be assigned to a variable

Be used as an argument

Be returned by a function call

Finally, Stackfulness

- **Stackful** coroutines allow coroutines to suspend their execution from within **nested** functions.



Finally, Stackfulness

- **Stackless** coroutines like in Python and Kotlin are **not Full Coroutines**.

With this, we show that a Full Coroutine would have to be “Stackful” and be provided as “First Class objects”

Full Coroutines

- **Full Coroutines** can be used to implement Generators, Iterators, Goal Oriented Programming and **Cooperative Multitasking**

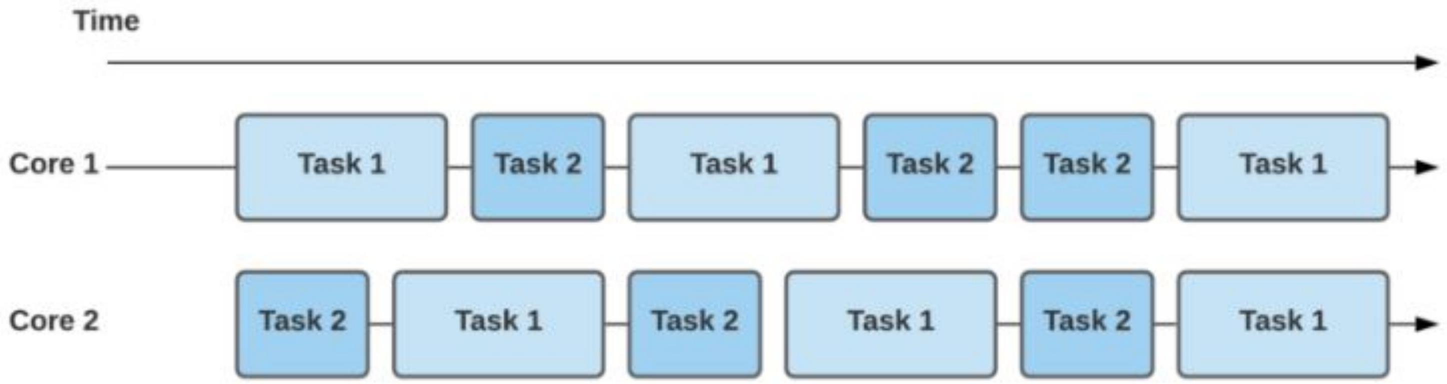
Full Coroutines

- **Full Coroutines** can be used to implement Generators, Iterators, Goal Oriented Programming and **Cooperative Multitasking**
- And just providing **asymmetric** coroutine mechanisms is sufficient as they can implement symmetric coroutines and are **much easier to implement**.

Cooperative Multitasking - a small caveat

Cooperative Multitasking

- In a cooperative multitasking environment, the interleaving of concurrent tasks is **deterministic**.



Cooperative Multitasking

- In a cooperative multitasking environment, the interleaving of concurrent tasks is **deterministic**.
- There is a fairness problem that arises when concurrent tasks execute time-consuming operations - non-preemption

Cooperative Multitasking

- In user-level multitasking, coroutines are part of the **same program** and collaborate to achieve a common goal

Cooperative Multitasking

- In user-level multitasking, coroutines are part of the **same program** and collaborate to achieve a common goal
- Since fairness problems are restricted to the collaborative environment, they are more easily identified and reproduced, and not difficult to implement.

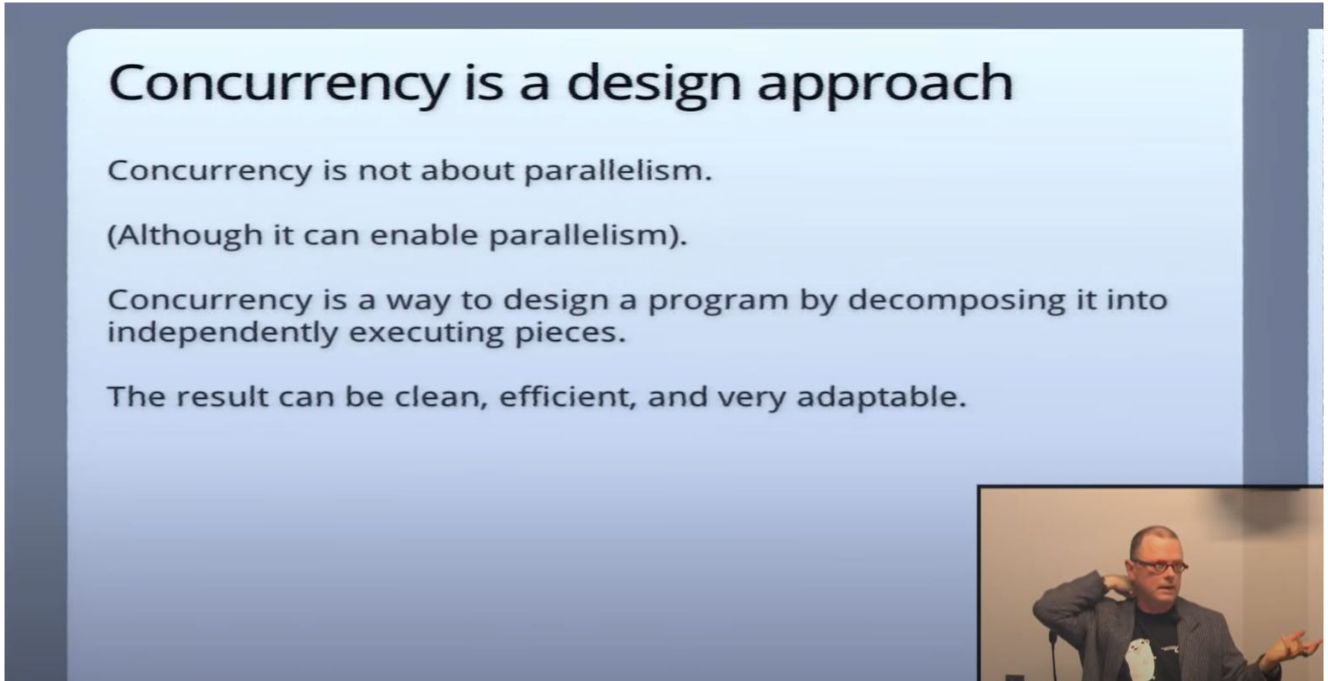
Why Coroutines in Go?

Coroutines in Go

- Coroutines are not directly served by existing Go concurrency libraries

Coroutines in Go

- Coroutines are not directly served by existing Go concurrency libraries
- As an example, In Rob Pike's talk "Lexical Scanning in Go", They ran in separate **goroutines connected by a channel**.



Concurrency is a design approach

Concurrency is not about parallelism.

(Although it can enable parallelism).

Concurrency is a way to design a program by decomposing it into independently executing pieces.

The result can be clean, efficient, and very adaptable.

Coroutines in Go

- **Full goroutines** proved to be a bit too much. The **parallelism** provided by the goroutines caused **races**.

Coroutines in Go

- **Full goroutines** proved to be a bit too much. The **parallelism** provided by the goroutines caused **races**.
- Proper **coroutines** would have **avoided the races** and been more efficient than goroutines because of **concurrency constructs**.

Difference between Coroutines, Threads and Generators

Difference between Coroutines, Threads and Generators

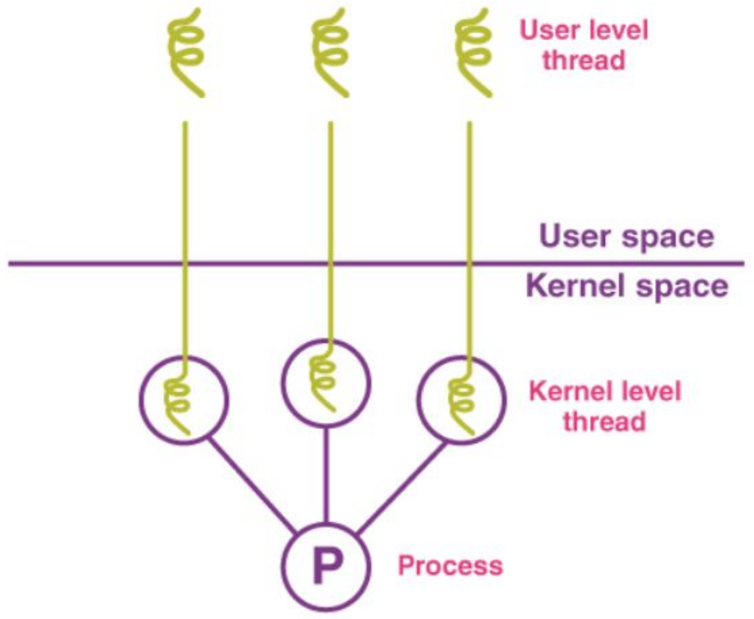
to get it out of the way

Coroutines, Threads and Generators

- **Coroutines** provide concurrency without parallelism: when one coroutine is running, the others are not.

Coroutines, Threads and Generators

- **Threads** provide more power than coroutines, but with more cost.



Coroutines, Threads and Generators

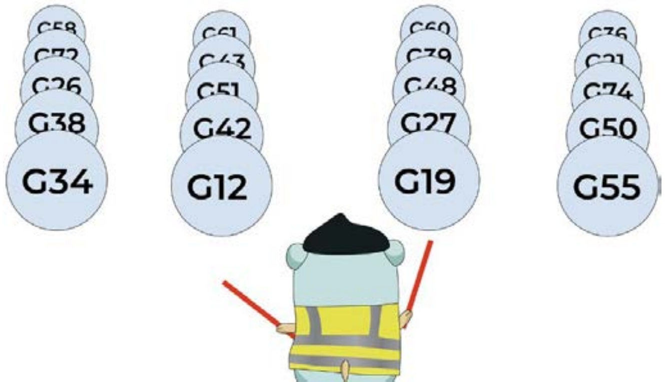
- **Threads** provide more power than coroutines, but with more cost.
- With Parallelism, the cost is the **overhead of scheduling**, including more expensive context switches.

Coroutines, Threads and Generators

- **Threads** provide more power than coroutines, but with more cost.
- With Parallelism, the cost is the **overhead of scheduling**, including more expensive context switches.
- The need to add **preemption** for this.

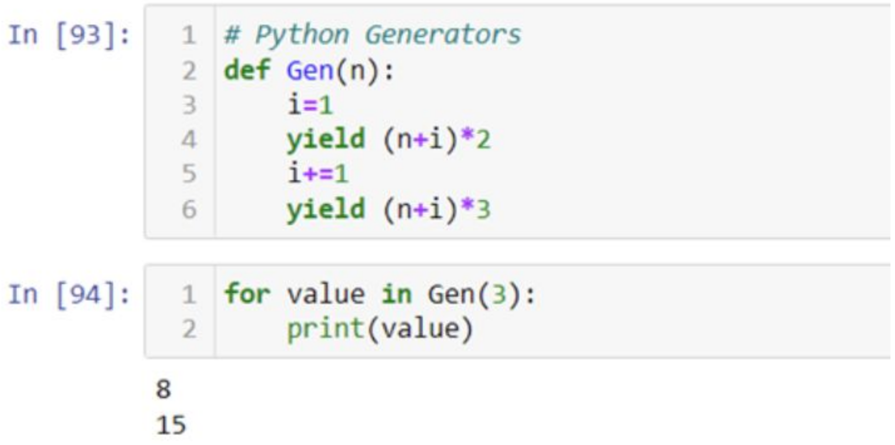
Coroutines, Threads and Generators

- **Goroutines** are cheap threads: a goroutine switch is closer to a few hundred nanoseconds.



Coroutines, Threads and Generators

- **Generators** (like in python) provide less power than coroutines - Stackless.



```
In [93]:
# Python Generators
def Gen(n):
    i=1
    yield (n+i)*2
    i+=1
    yield (n+i)*3
```

```
In [94]:
for value in Gen(3):
    print(value)
```

```
8
15
```

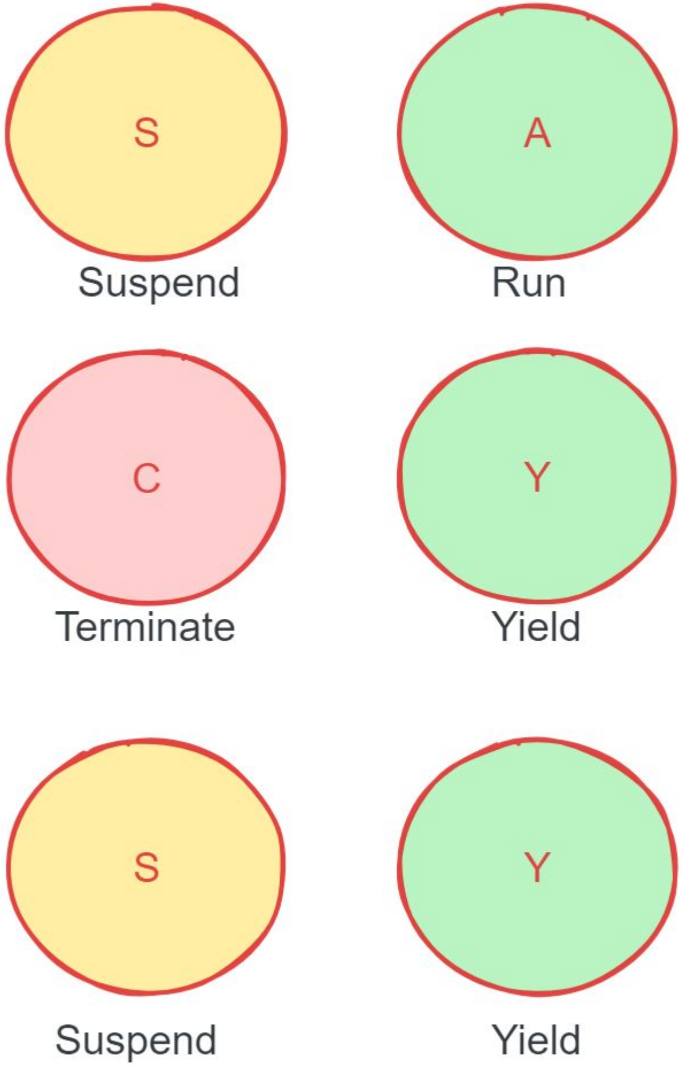
Let’s build an API for Coroutines in Go by using definitions available today

Let’s build an API for Coroutines in Go by using definitions available today

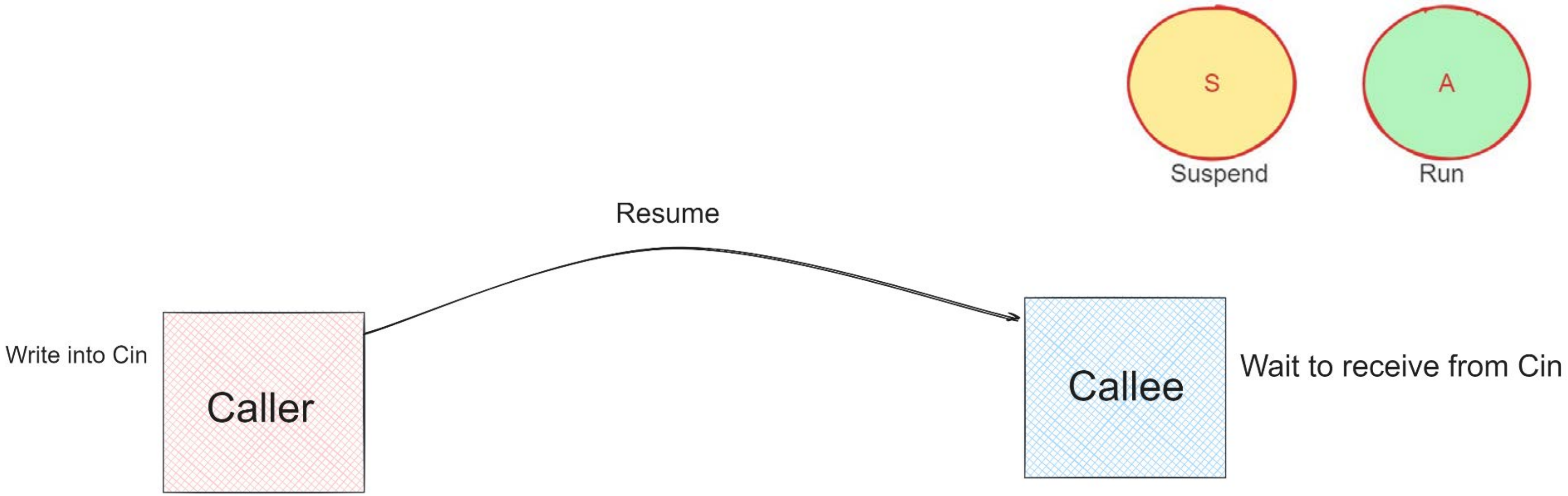
(This part of the talk is borrowed from Russ’ Research Proposal for implementing Coroutines)

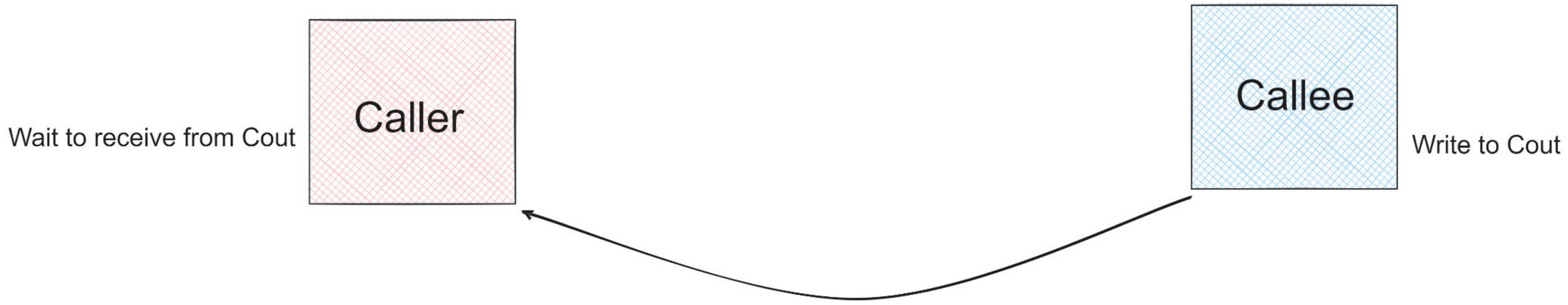
API for Coroutines

- It is very neat that we can do this using existing **Go definitions, Goroutines and Channels** because of how channels work with blocking Goroutines (Goroutine-safe), and Go’s support for **function values**


S
Suspend
A
Run
C
Terminate
Y
Yield
S
Suspend
Y
Yield

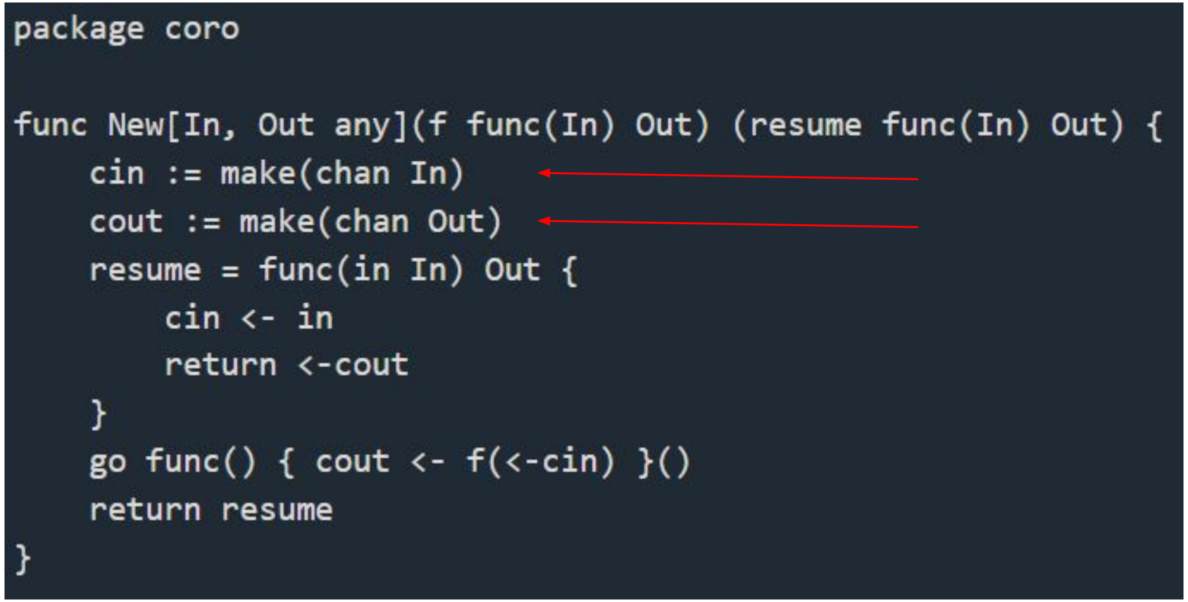
We start with a simple implementation of the package coro.


S
Suspend
A
Run
Resume
Write into Cin
Caller
Callee
Wait to receive from Cin


Wait to receive from Cout
Caller
Callee
Write to Cout

API for Coroutines

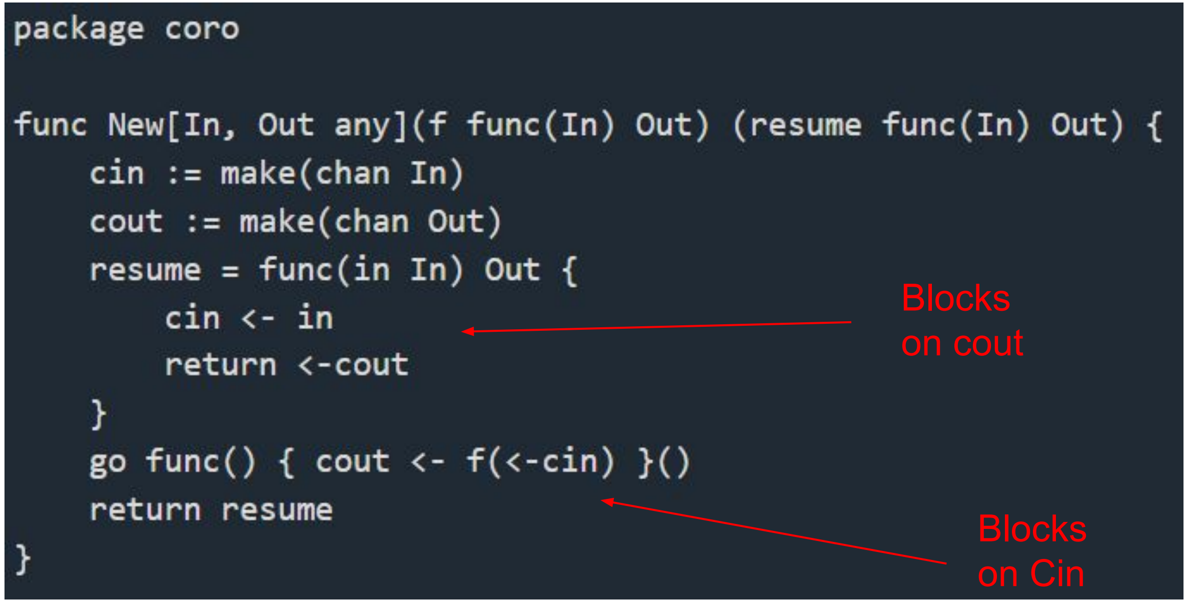
- This will define a function New that takes a function as an argument and one result, it allocates channels, **creates a goroutine** to run f, and returns the **resume** function.



```
package coro

func New[In, Out any](f func(In) Out) (resume func(In) Out) {
    cin := make(chan In)
    cout := make(chan Out)
    resume = func(in In) Out {
        cin <- in
        return <-cout
    }
    go func() { cout <- f(<-cin) }()
    return resume
}
```

API for Coroutines

- This will define a function New that takes a function as an argument and one result, it allocates channels, **creates a goroutine** to run f, and returns the **resume** function.



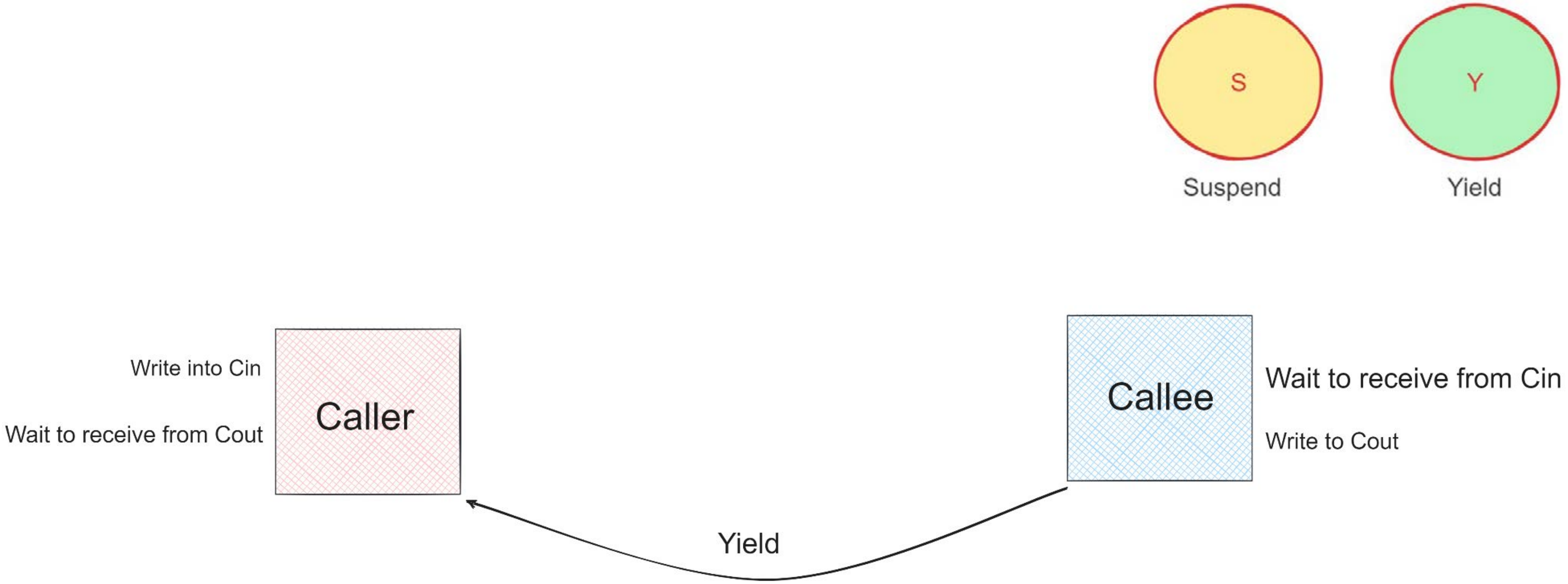
```
package coro

func New[In, Out any](f func(In) Out) (resume func(In) Out) {
    cin := make(chan In)
    cout := make(chan Out)
    resume = func(in In) Out {
        cin <- in
        return <-cout
    }
    go func() { cout <- f(<-cin) }()
    return resume
}
```

API for Coroutines

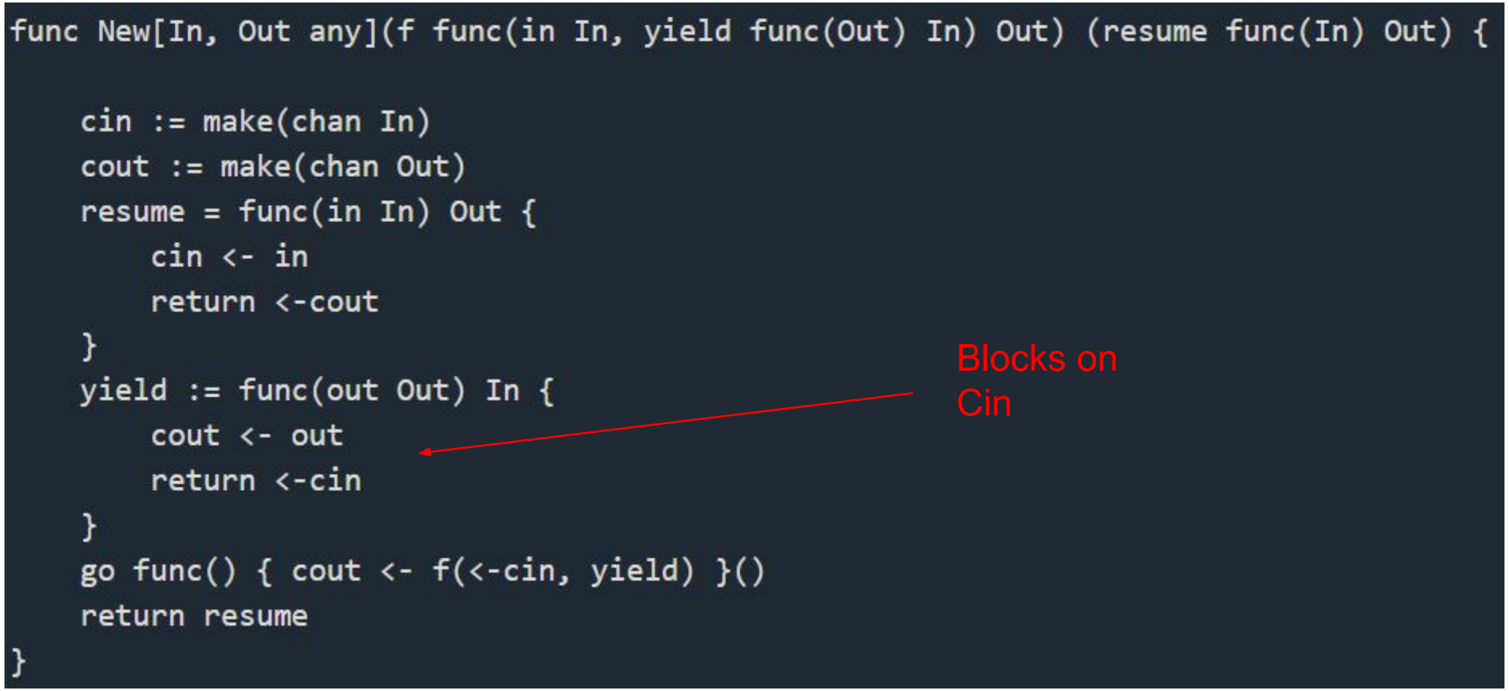
- This will define a function New that takes a function as an argument and one result, it allocates channels, **creates a goroutine** to run f, and returns the **resume** function.
- The new goroutine **blocks** on <-cin - No Parallelism

API for Coroutines

- This will define a function New that takes a function as an argument and one result, it allocates channels, **creates a goroutine** to run f, and returns the **resume** function.
- The new goroutine **blocks** on <-cin - No Parallelism
- Let’s add the definition for “**yield**” to suspend a function and return its value to the coroutine that “resumed” it.


S
Suspend
Y
Yield
Write into Cin
Wait to receive from Cout
Caller
Callee
Wait to receive from Cin
Write to Cout
Yield

API for Coroutines



```
func New[In, Out any](f func(in In, yield func(Out) In) Out) (resume func(In) Out) {

    cin := make(chan In)
    cout := make(chan Out)
    resume = func(in In) Out {
        cin <- in
        return <-cout
    }
    yield := func(out Out) In {
        cout <- out
        return <-cin
    }
    go func() { cout <- f(<-cin, yield) }()
    return resume
}
```

Blocks on Cin

API for Coroutines

- **Note:** “This is just an addition of a send-receive pair and there is still no parallelism”

Let’s pause for a bit, are these actually Coroutines?

Are these Coroutines?

- Yes and no

Are these Coroutines?

- Yes and no
- They are full goroutines, and they can do everything an ordinary goroutine can

Are these Coroutines?

- Yes and no
- They are full goroutines, and they can do everything an ordinary goroutine can
- **coro.New** creates goroutines with access to “resume” and “yield” operations.

Are these Coroutines?

- Yes and no
- They are full goroutines, and they can do everything an ordinary goroutine can
- **coro.New** creates goroutines with access to “resume” and “yield” operations.
- Unlike with the ‘go’ statement, we are **adding new concurrency** to the program **without parallelism**.

Are these Coroutines?

- “If you have just one main goroutine and run 10 **go statements**, then all 11 goroutines can be **running at once”**.

Are these Coroutines?

- “But if you have one main goroutine and run 10 **coro.New** calls, there are now 11 control flows but the parallelism of the program is what it was before”:

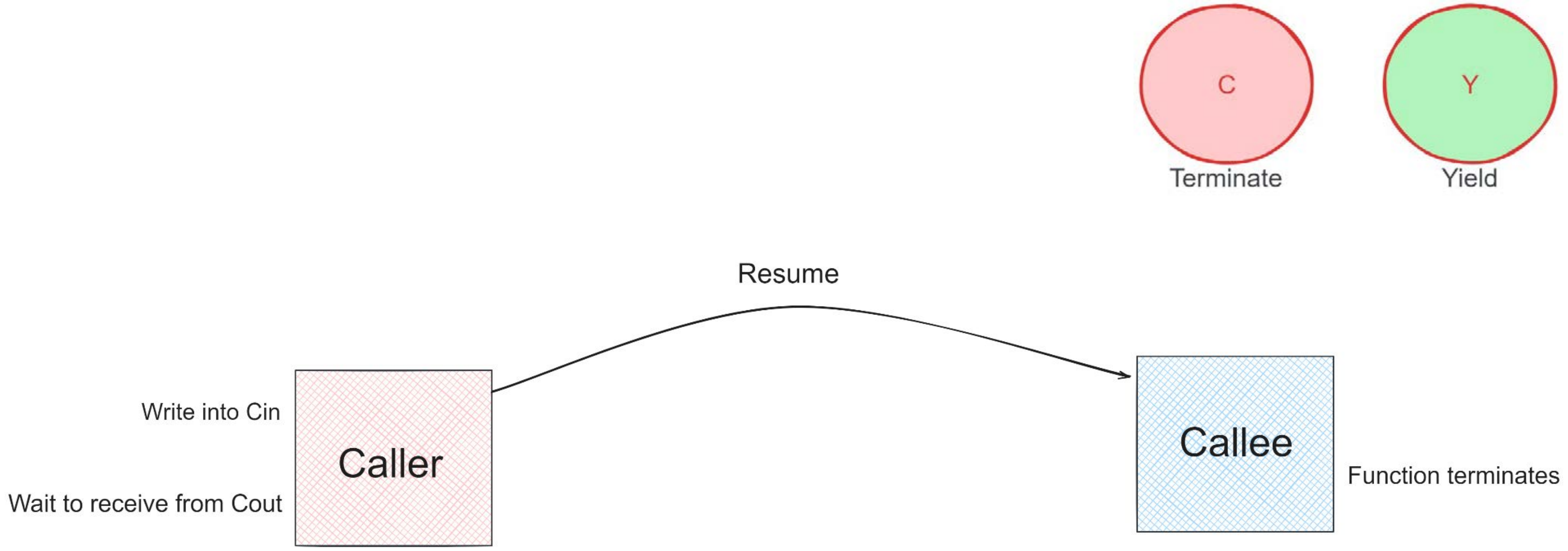
Are these Coroutines?

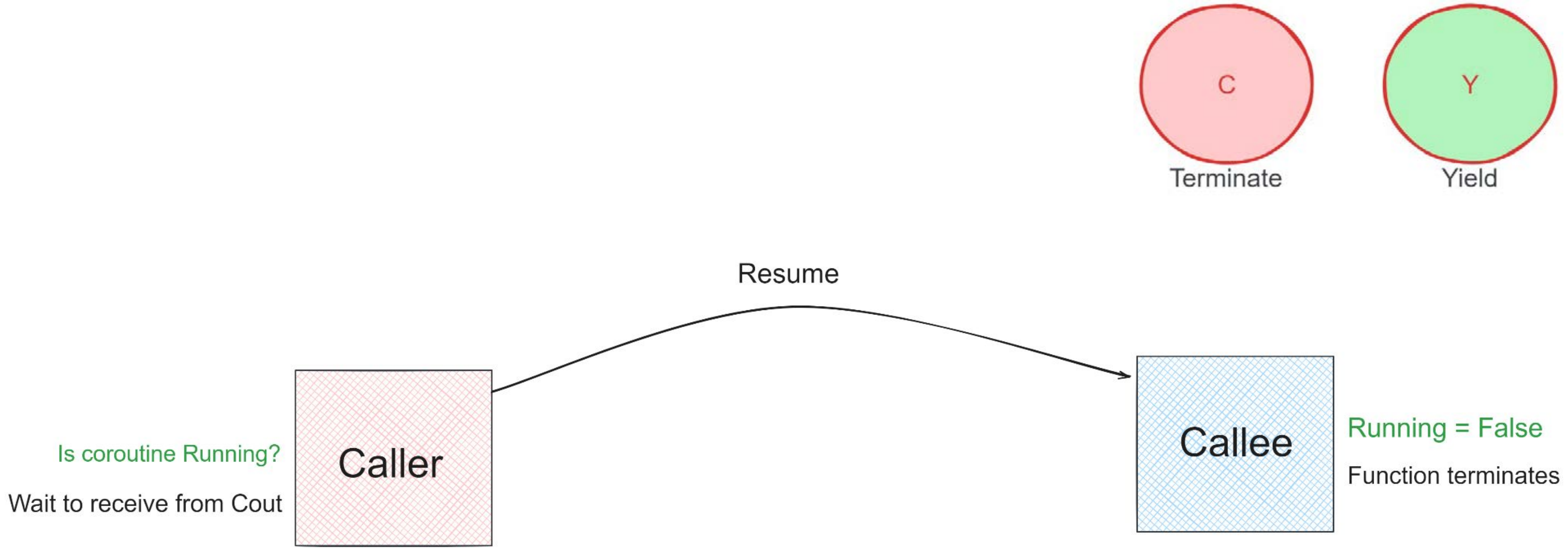
- “But if you have one main goroutine and run 10 **coro.New** calls, there are now 11 control flows but the parallelism of the program is what it was before”: only one.

Are these Coroutines?

- “**go**” creates a new concurrent, parallel control flow, while **coro.New** creates a new **concurrent, non-parallel** control flow”

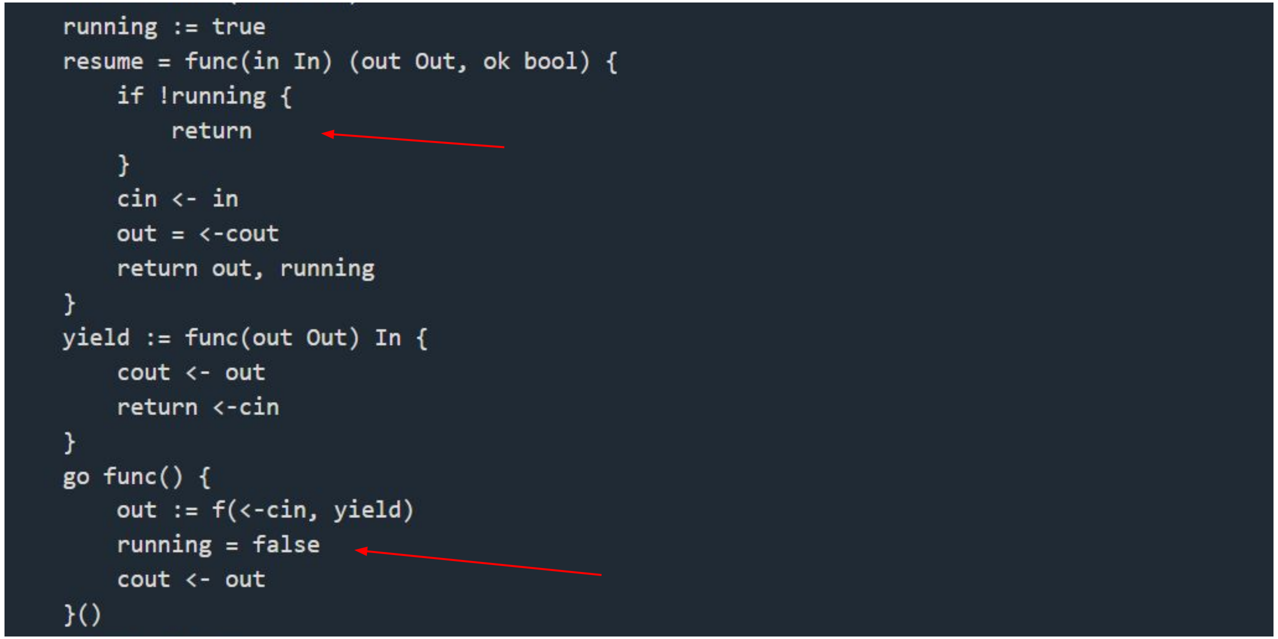
Back to implementing our coro API - *Improvements*


C
Terminate
Y
Yield
Resume
Write into Cin
Caller
Wait to receive from Cout
Callee
Function terminates


C
Terminate
Y
Yield
Resume
Caller
Is coroutine Running?
Wait to receive from Cout
Callee
Running = False
Function terminates

API for Coroutines

- Allow resume to be called after the **function is done**: right now it will deadlock.



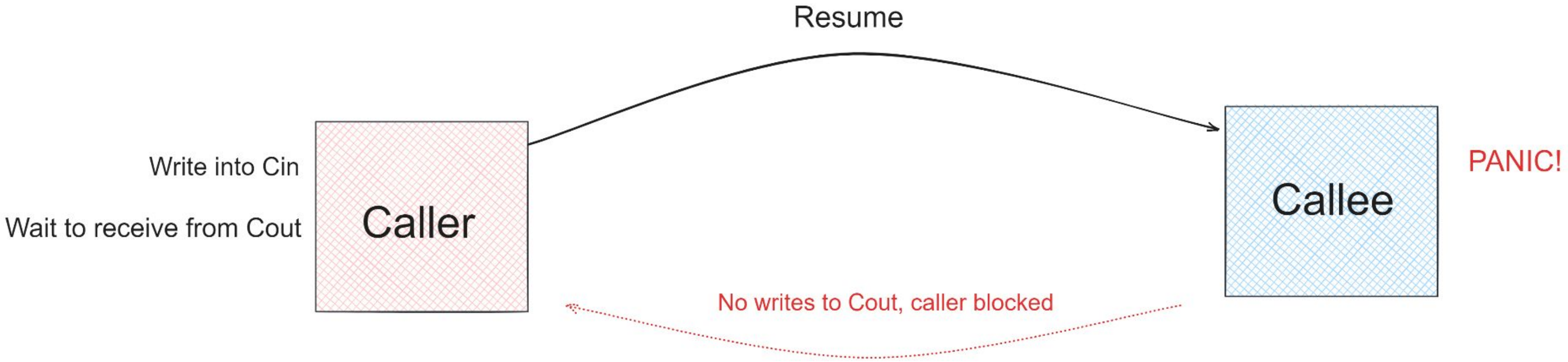
```
running := true
resume = func(in In) (out Out, ok bool) {
    if !running {
        return
    }
    cin <- in
    out = <-cout
    return out, running
}
yield := func(out Out) In {
    cout <- out
    return <-cin
}
go func() {
    out := f(<-cin, yield)
    running = false
    cout <- out
}()
```

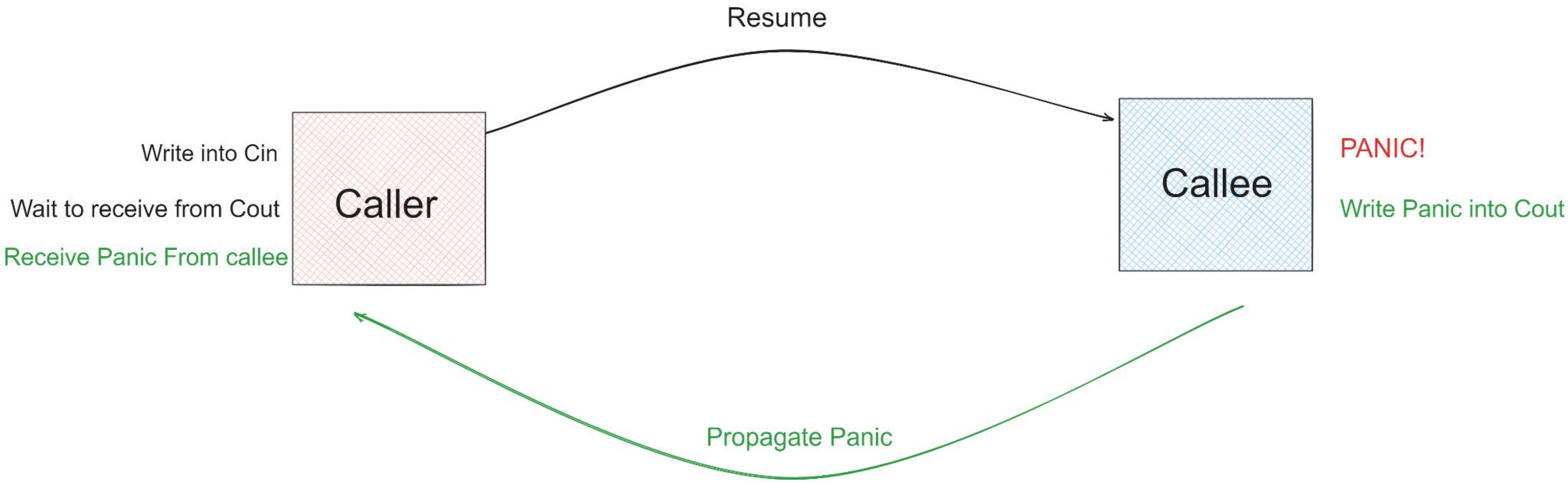
API for Coroutines

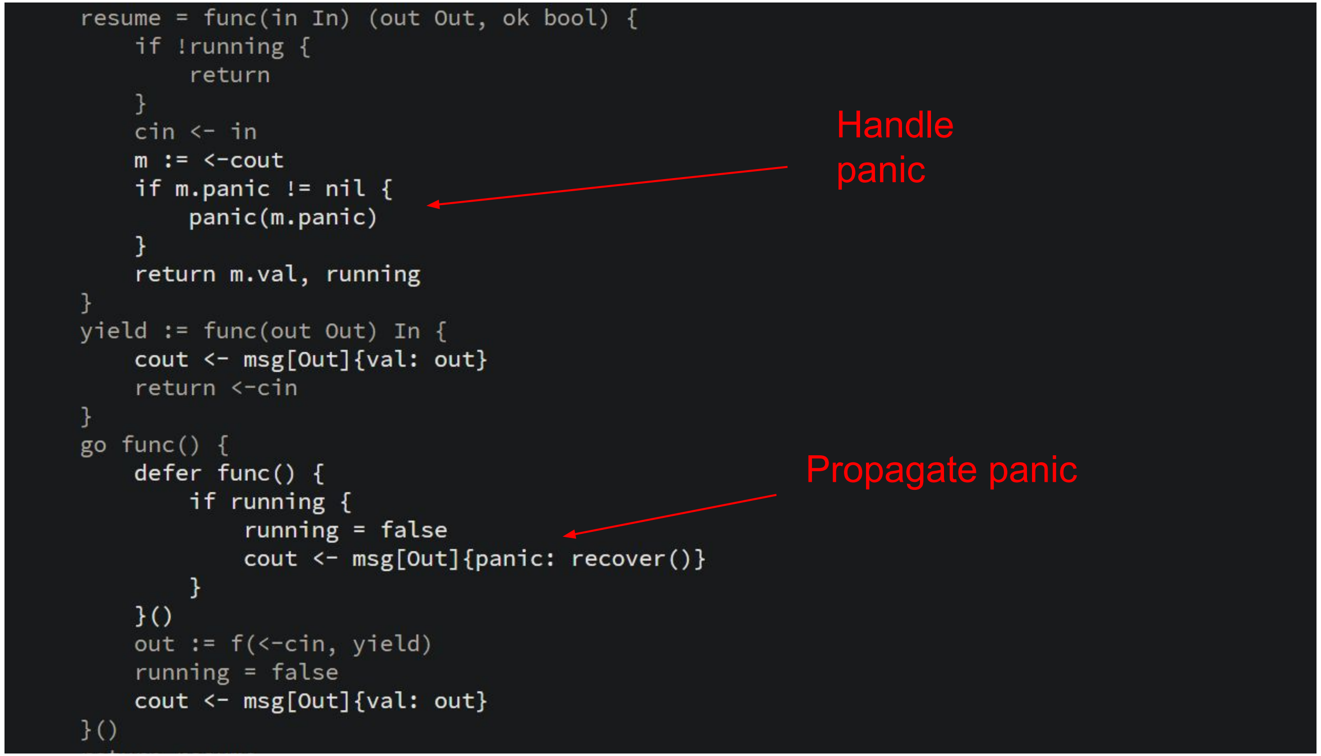
- **Pass panics** from a coroutine back to its caller

API for Coroutines

- **Pass panics** from a coroutine back to its caller
- If a panic occurs in a coroutine context, we have the caller **blocked waiting** for news.


Resume
Write into Cin
Wait to receive from Cout
Caller
Callee
PANIC!
No writes to Cout, caller blocked


Resume
Write into Cin
Wait to receive from Cout
Receive Panic From callee
Caller
Callee
PANIC!
Write Panic into Cout
Propagate Panic



```go
resume = func(in In) (out Out, ok bool) {
    if !running {
        return
    }
    cin <- in
    m := <-cout
    if m.panic != nil {
        panic(m.panic)
    }
    return m.val, running
}
yield := func(out Out) In {
    cout <- msg[Out]{val: out}
    return <-cin
}
go func() {
    defer func() {
        if running {
            running = false
            cout <- msg[Out]{panic: recover()}
        }
    }()
    out := f(<-cin, yield)
    running = false
    cout <- msg[Out]{val: out}
}()
```

API for Coroutines

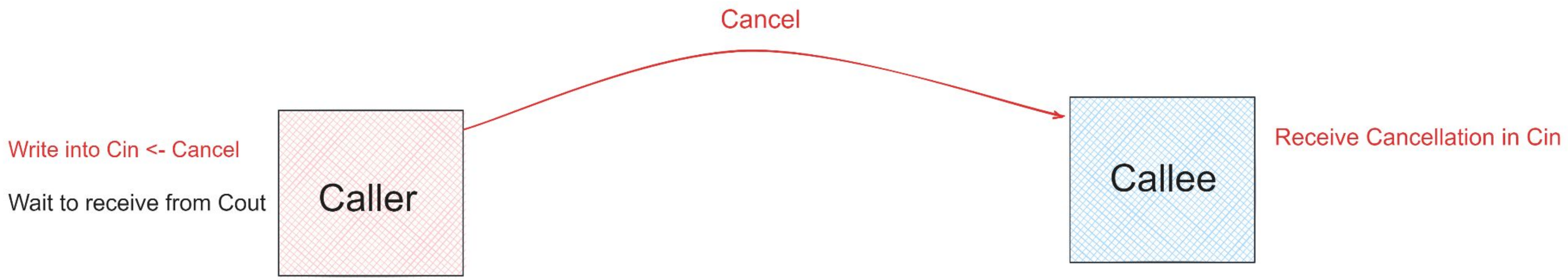
- We need some way to signal to the coroutine that it’s **no longer needed**.

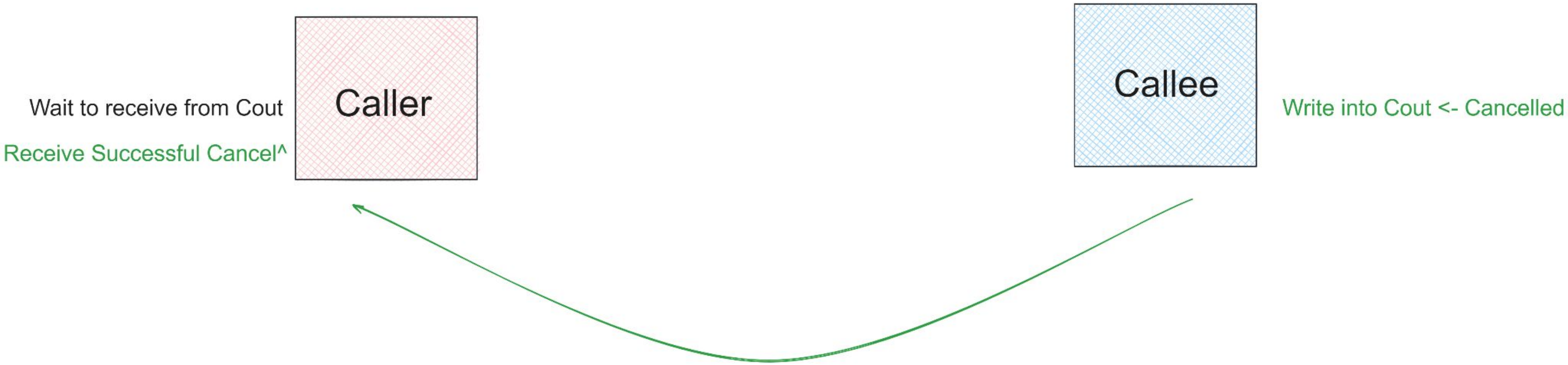
API for Coroutines

- We need some way to signal to the coroutine that it’s **no longer needed**.
- Perhaps because the caller is panicking, or because the caller is simply returning.

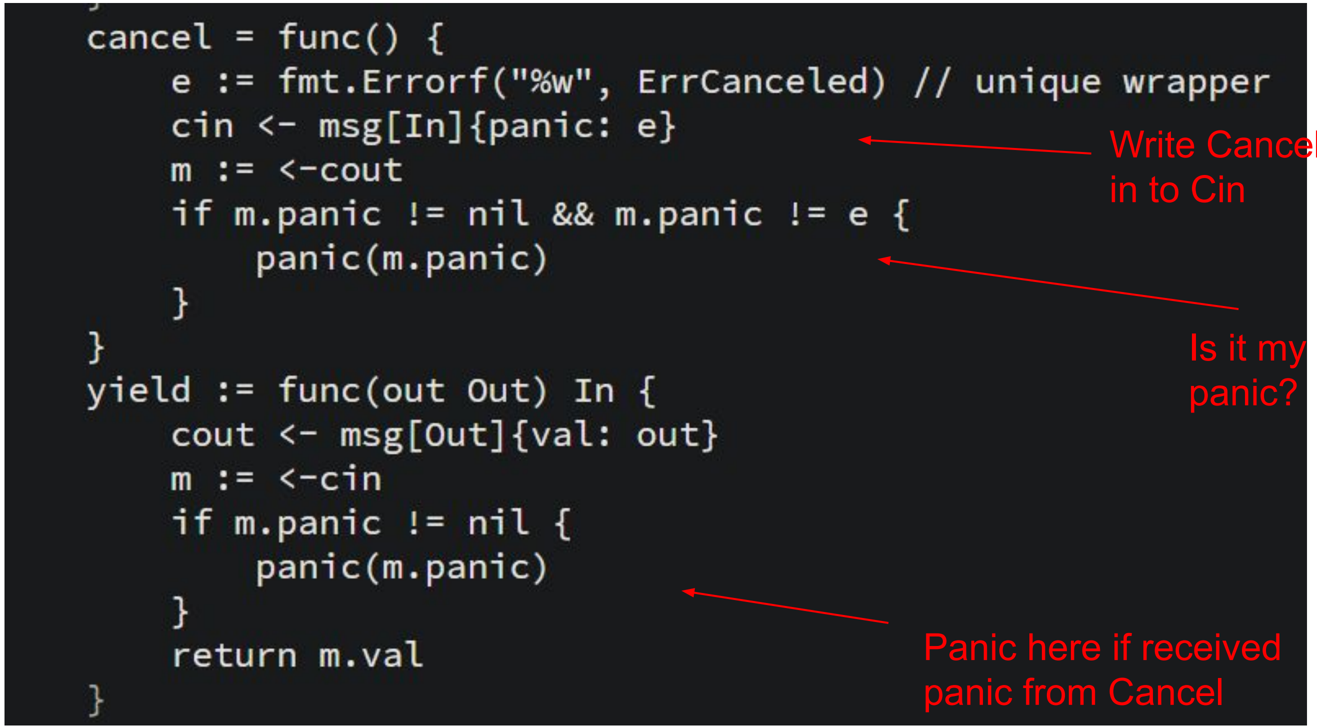
API for Coroutines

- We need some way to signal to the coroutine that it’s **no longer needed,**
- Perhaps because the caller is panicking, or because the caller is simply returning.
- To do that, we can change coro.New to return a **cancel func** as well


Cancel
Write into Cin <- Cancel
Wait to receive from Cout
Caller
Callee
Receive Cancellation in Cin


Wait to receive from Cout
Receive Successful Cancel^
Caller
Callee
Write into Cout <- Cancelled

API for Coroutines



```
cancel = func() {
    e := fmt.Errorf("%w", ErrCanceled) // unique wrapper
    cin <- msg[In]{panic: e}
    m := <-cout
    if m.panic != nil && m.panic != e {
        panic(m.panic)
    }
}
yield := func(out Out) In {
    cout <- msg[Out]{val: out}
    m := <-cin
    if m.panic != nil {
        panic(m.panic)
    }
    return m.val
}
```

Write Cancel in to Cin

Is it my own panic?

Panic here if received panic from Cancel

Runtime Changes?

Runtime Changes

- While we have a definition of coroutines that can be implemented using pure Go,

Runtime Changes

- While we have a definition of coroutines that can be implemented using pure Go,
- Russ builds on the use of an optimized runtime implementation

Runtime Changes

- Some perf data he collected
- “On my 2019 MacBook Pro, passing values back and forth using the channel-based coro.New in this post requires approximately **190ns** per switch”

Runtime Changes

- He changes the compiler such that it can **mark send-receive pairs** and leave hints for the runtime to fuse them into a single operation.

Runtime Changes

- He changes the compiler such that it can **mark send-receive pairs** and leave hints for the runtime to fuse them into a single operation.
- That would let the channel runtime **bypass the scheduler** and jump directly to the other coroutine.

Runtime Changes

- He changes the compiler such that it can **mark send-receive pairs** and leave hints for the runtime to fuse them into a single operation.
- That would let the channel runtime **bypass the scheduler** and jump directly to the other coroutine.
- This implementation required about **118ns** per switch, **38%** faster.

Runtime Changes

- Another change he talks about is adding a direct coroutine switch to the runtime, **avoiding channels entirely**

Runtime Changes

- Another change he talks about is adding a direct coroutine switch to the runtime, **avoiding channels entirely**
- That implementation took **20ns** per switch.

Runtime Changes

- Another change he talks about is adding a direct coroutine switch to the runtime, **avoiding channels entirely**
- That implementation took **20ns** per switch.
- This is about **10X** faster than the original channel implementation.

Conclusions

Conclusions

We covered quite a bit, thanks for making it here!

Conclusions

- We were able to show that having a **Full Coroutine facility** in Go makes it even more powerful for implementing very **robust and generalised** concurrency patterns.

Conclusions

- We were able to show that having a **Full Coroutine facility** in Go makes it even more powerful for implementing very **robust and generalised** concurrency patterns.
- We covered the basics of **Coroutine fundamentals**, its history and why it’s **not as prolific** as it should be today in mainstream languages.

Conclusions

- We were able to show that having a **Full Coroutine facility** in Go makes it even more powerful for implementing very r**obust and generalised** concurrency patterns.
- We covered the basics of **Coroutine fundamentals**, its history and why it’s n**ot as prolific** as it should be today in mainstream languages.
- We showed what Full Coroutines are, as a function of the different **classifications** of it

Conclusions

- Why we need to have Coroutines in Go, how they **differ from Goroutines** and how they would **differ from existing implementations** in some other languages.

Conclusions

- Why we need to have Coroutines in Go, how they **differ from Goroutines** and how they would **differ from existing implementations** in some other languages.
- We then implemented a coroutine API using **existing Go definitions**, and build on it to make it **robust**.

Conclusions

- Why we need to have Coroutines in Go, how they **differ from Goroutines** and how they would **differ from existing implementations** in some other languages.
- We then implemented a coroutine API using **existing Go definitions**, and build on it to make it **robust**.
- We showed what **runtime changes** can be made to make the implementation even more **efficient**.

References

- Coroutines for Go - Russ Cox
- C++ Coroutines - a negative overhead abstraction - Gor Nishanov
- Generators, Coroutines and Other Brain Unrolling Sweetness - Adi Shavit
- Happy birthday, amazing Grace Hopper
- Lexical Scanning in Go - Rob Pike
- Design of a Separable Transition-Diagram Compiler
- Revisiting Coroutines

Artworks: Renée French, Takuya Ueda, Quasylite

Thank you!