

MPP 2026/27 (0077A, 9CFU)

Models for Programming Paradigms

Roberto Bruni Filippo Bonchi
http://www.di.unipi.it/~bruni/

https://didawiki.di.unipi.it/doku.php/magistraleinformatica/mpp/start

16b - Google Go

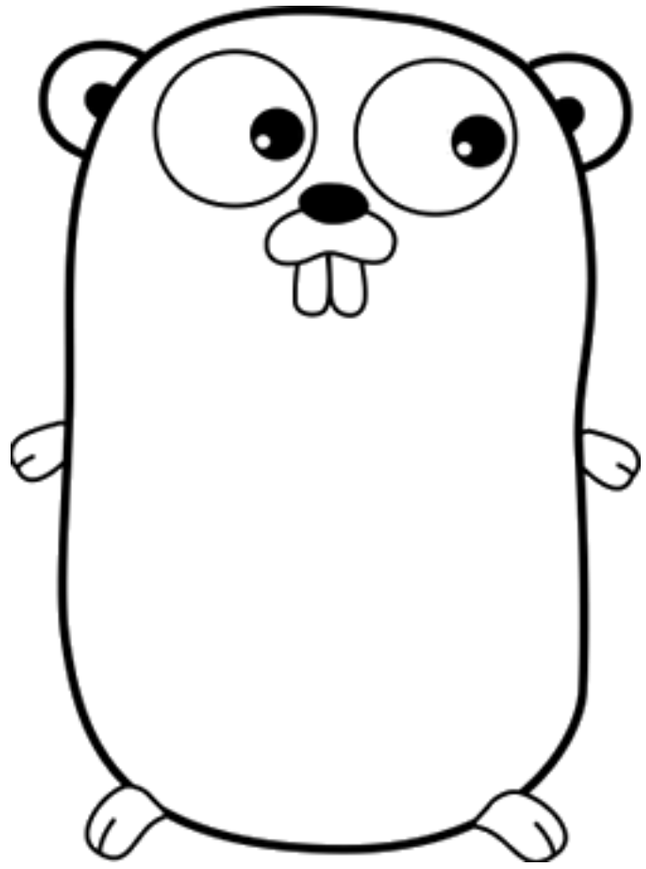
Google Go

concurrency oriented programming

Google Go

http://golang.org/





Go features

facilitate building reliable and efficient software

open source

compiled, garbage collected

functional and OO features

statically typed (light type system)

concurrent

Go principles

C, C++, Java:
too much typing (writing verbose code)
and too much typing (writing explicit types)
(and poor concurrency)

Python, JS:
no strict typing, no compiler issues
runtime errors that should be caught statically

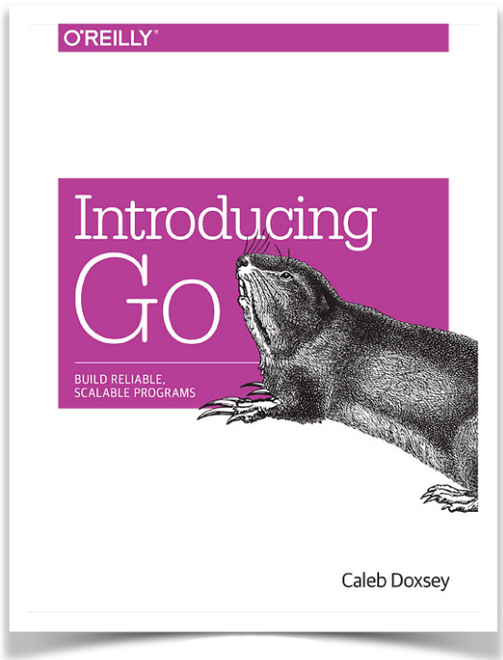
Google Go:
compiled, static types, type inference
(and nice concurrency primitives)

Go project

designed by Ken Thompson, Rob Pike, Robert Griesemer

2007: started experimentation at Google
nov 2009: first release (more than 250 contributors)
may 2012: version 1.0 (two yearly releases since 2013)
feb 2025: version 1.24.0

C. Doxsey, Introducing Go (2016). Ch: 1-4, 6-7, 10



Go concurrency

any function can be executed in a separate lightweight thread

```
go f(x)
```

goroutines run in the same address space

package `sync` provides basic synchronisation primitives
programmers are encouraged NOT TO USE THEM!

do not communicate by sharing memory
instead, share memory by communicating

use built-in high-level concurrency primitives:
channels and **message passing**
(inspired by process algebras)

Go channels

channels can be created and passed around

```
var ch = make(chan int)
```

creates a channel for transmitting integers

```
ch1 = ch
```

aliasing: `ch1` and `ch` now refers to the same channel

```
go f(ch)
go g(ch)
```

`f` and `g` share the channel `ch`

Directionality

channels are alway created bidirectional

```
var ch = make(chan int)
```

channel types can be annotated with directionality

```
var rec <-chan int
```

`rec` can only be used to receive integers

```
var snd chan<- int
```

`snd` can only be used to send integers

```
rec = ch
snd = ch
```

are valid assignments

```
rec = snd // invalid!
```

Go communication

to send a value (like *ch!2*) `ch <- 2`

to receive and store in `x` (like *ch?x*) `x = <- ch`

to receive and throw the value away `<- ch`

to close a channel (by the sender) `close(ch)`

to check if a channel has been closed (by the receiver)

```
x,ok = <- ch  // either value,true or 0,false
```

Go sync communication

by default the communication is **synchronous**

BOTH send and receive are BLOCKING!

asynchronous channels can be created
by allocating a buffer of fixed size

```
var ch = make(chan int, 100)
```

creates an **asynchronous channel** of size 100

receive on asynchronous channel is of course still blocking

send is blocking only if the buffer is full

no dedicated type for asynchronous channels:
buffering is a property of values not of types

Go communication

to choose between different options

```
select {
    case x = <- ch1: { … }
    case ch2 <- v: { … }
    // both send and receive actions
    default: { … }
}
```

the selection is made pseudo-randomly among enabled cases

if no case is enabled, the default option is applied

if no case is enabled, and no default option is given
the select blocks until (at least) one case is enabled

Example

non-blocking receive

```
select {
    case x = <- ch: { … }
    default: { … }
}
```

receives on `x` from `ch`, if data available

otherwise proceeds

Example

wait for first among many (senders)

```
select {
    case x = <- ch1: { … }
    case x = <- ch2: { … }
    case x = <- ch3: { … }
}
```

receives on `x` from any of `ch1`, `ch2`, `ch3`, if data available

otherwise waits

Example

wait for first among many (receivers)

```
select {
    case ch1 <- v : { … }
    case ch2 <- v : { … }
    case ch3 <- v : { … }
}
```

sends `v` to any of `ch1`, `ch2`, `ch3`, if available to receive

otherwise waits

Name mobility

channels can be sent over channels (like in π-calculus)

```
var mob = make(chan chan int)
```

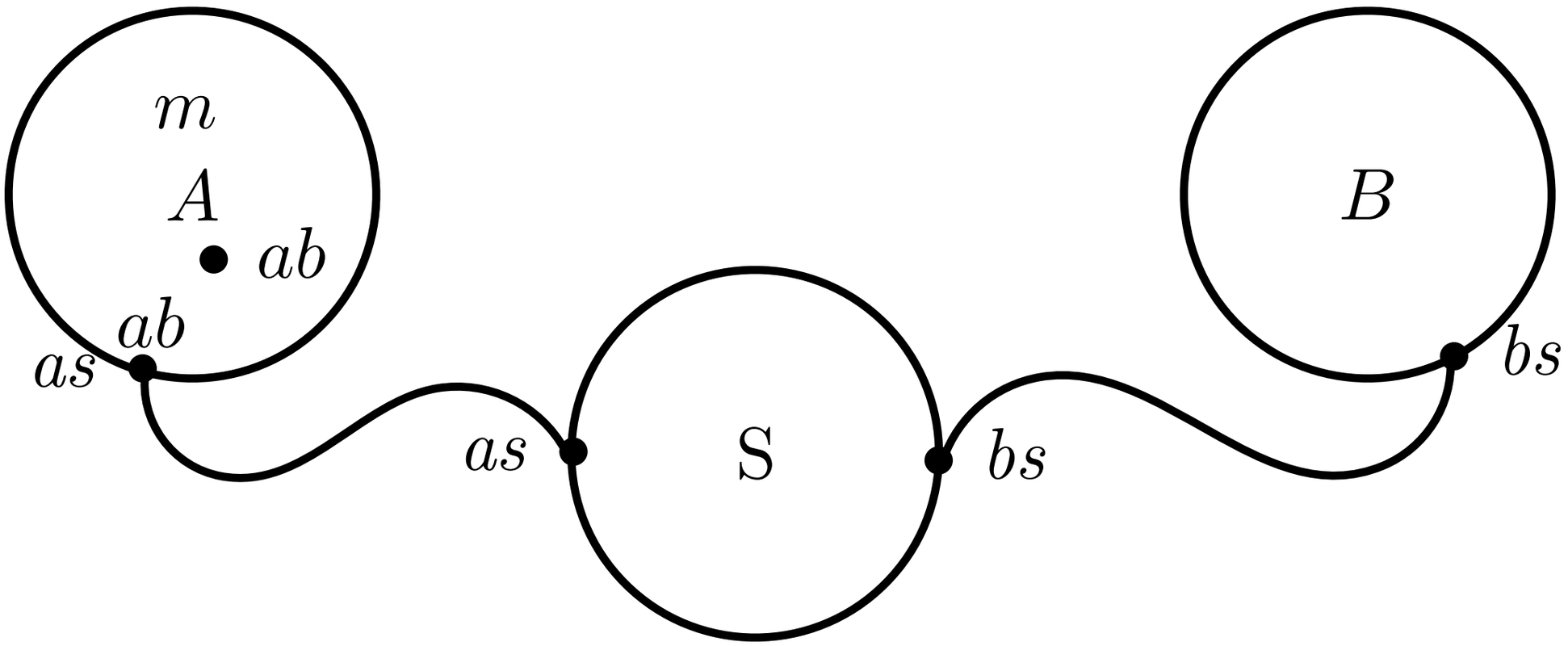
a channel for communicating channels

```
mob <- ch
```

send the channel `ch` over `mob`

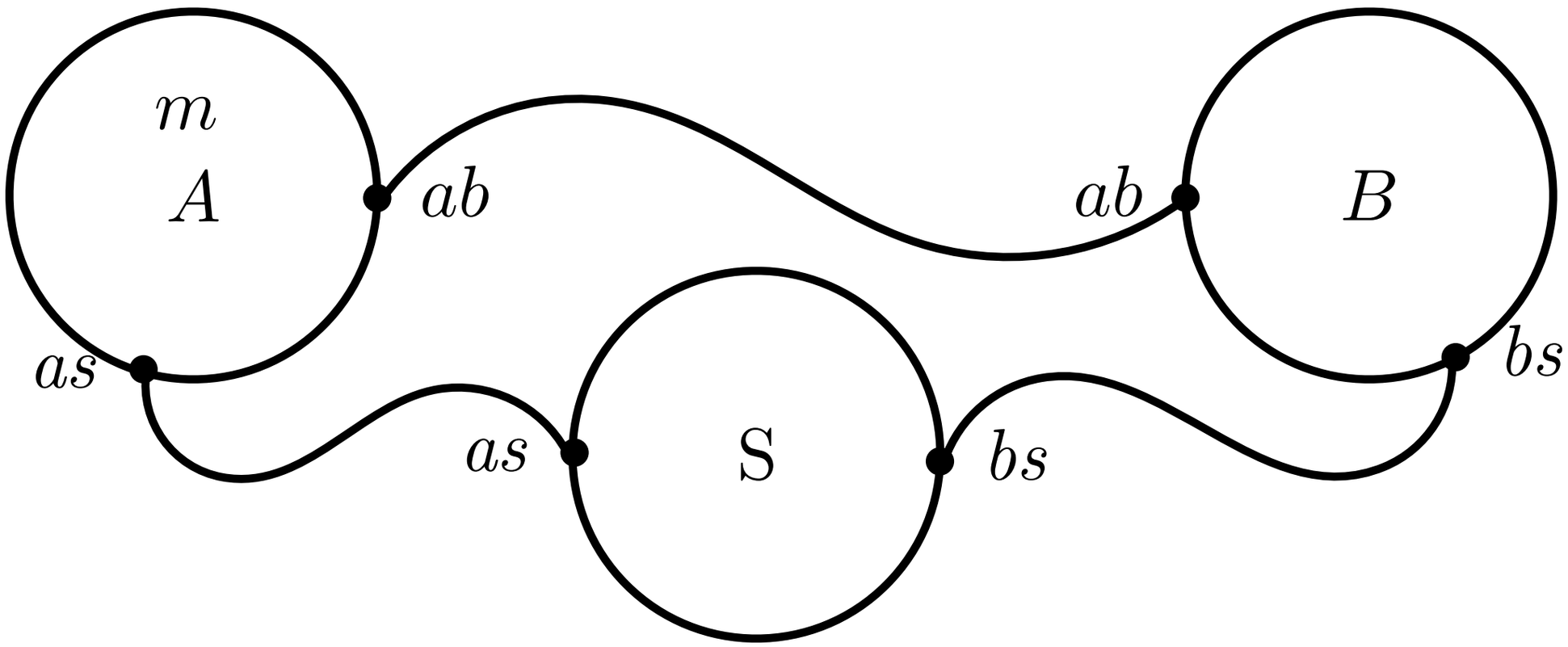
Name mobility: secrecy



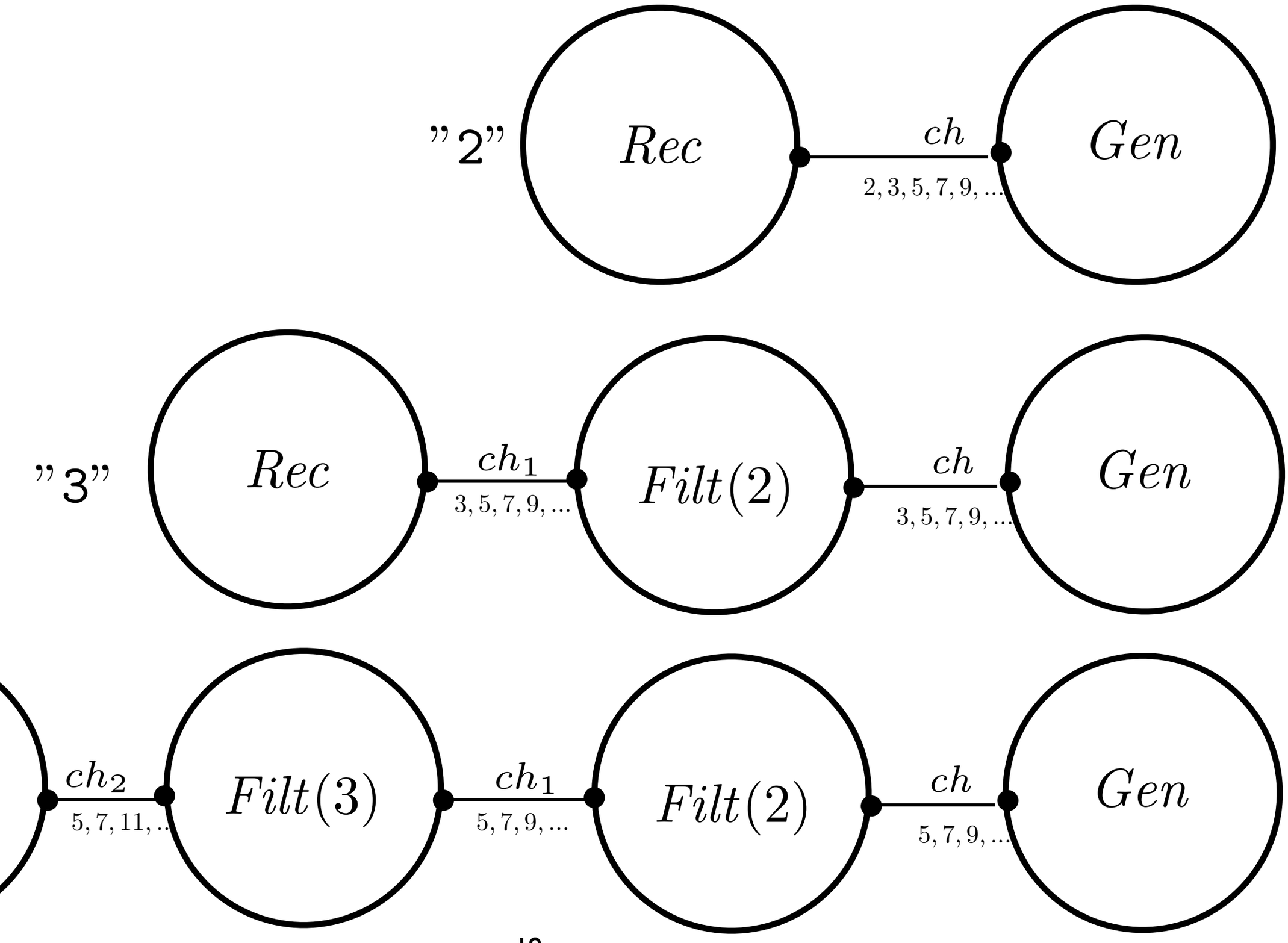


Name mobility: secrecy



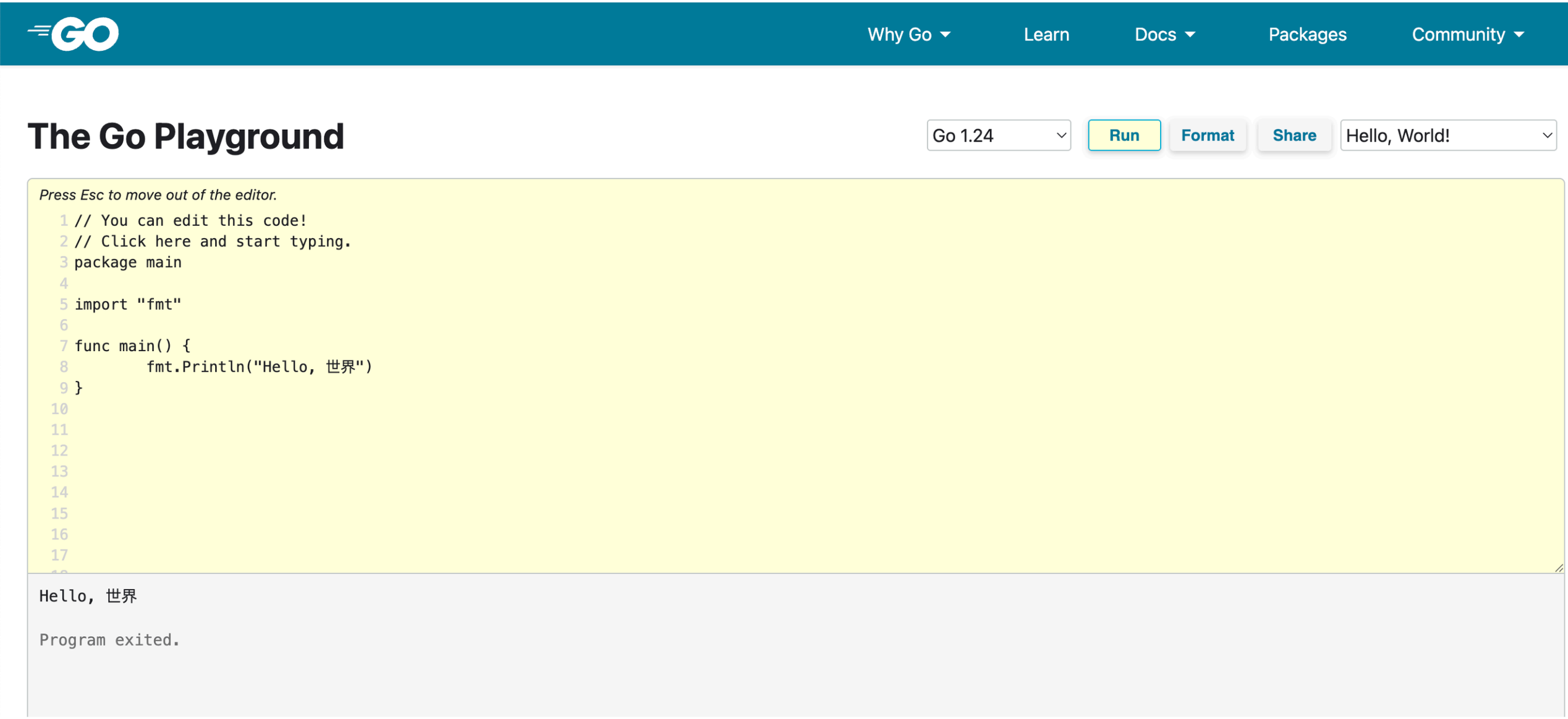


Concurrent prime sieve



Go playground

https://go.dev/play/



Basics

The Go Playground

Go 1.24 | Run | Format | Share | Hello, World!

Press Esc to move out of the editor.

```
1 package main
2
3 // constants are declared as variables, but using the keyword const
4 const Pi = 3.14
5
6 func main() {
7
8         // multiple variables declaration with default init (0, false, "")
9         // types follow variable names
10        var x, y, z int
11        println(x, y, z)
12 }
13
```

```
0 0 0

Program exited.
```

Basics

```
package main

// constants are declared as variables, but using the keyword const
const Pi = 3.14

func main() {

        // multiple variables declaration with default init (0, false, "")
        // types follow variable names
        var x, y, z int
        println(x, y, z)

        // multiple assignments are possible
        // semi-colons ; are not necessary, use new-line instead
        x = 1
        x, y, z = 2, x+1, y+1
        println(x, y, z)

}

```

```
0 0 0
2 2 1

Program exited.
```

Basics

```
package main

func main() {

	// multiple variables declaration with default init (0, false, "")
	// types follow variable names
	var x, y, z int
	println(x, y, z)

	// multiple assignments are possible
	// semi-colons ; are not necessary, use new-line instead
	x = 1
	x, y, z = 2, x+1, y+1
	println(x, y, z)

	// multiple variables declaration with init
	var i, j int = 1, 2
	println(i, j)

}

```

```
0 0 0
2 2 1
1 2

Program exited.
```

Basics

```go
package main

import "fmt"

// constants are declared as variables, but using the keyword const
const Pi = 3.14

func main() {

        // multiple intializations
        // using := the keyword var can be omitted (:= within function body only)
        // types are inferred
        n, f, s, b, c := 5, 2.5, "Hello", true, 1+2i
        fmt.Printf("%v (%T)\n", n, n)
        fmt.Printf("%v (%T)\n", f, f)
        fmt.Printf("%v (%T)\n", s, s)
        fmt.Printf("%v (%T)\n", b, b)
        fmt.Printf("%v (%T)\n", c, c)
        fmt.Printf("Pi is %v (%T)\n", Pi, Pi)
```

```
5 (int)
2.5 (float64)
Hello (string)
true (bool)
(1+2i) (complex128)
Pi is 3.14 (float64)

Program exited.
```

Basics

```go
package main

func main() {

	// multiple intializations
	// using := the keyword var can be omitted (:= within function body only)
	// types are inferred
	n, f := 5, 2.5

	// Go base types:
	// bool string
	// int  int8  int16  int32  int64
	// uint uint8 uint16 uint32 uint64 uintptr
	// byte (alias for uint8)
	// rune (alias for int32 - "code point" Unicode)
	// float32 float64
	// complex64 complex128

	// cast: T(v)
	n = int(f)
	f = float64(n)
}


```

```
Program exited.
```

Functions

```
1 package main
2
3 // public names (exported) begin with a capital letter
4 func Square(a int) int {
5         return a * a
6 }
7
8 func main() {
9         println(Square(3))
10
11 }
12
13
14
15
```

```
9

Program exited.
```

Functions

```
package main

// functions can return tuples
func Swap(a int, b int) (int, int) {
        return b, a
}

func main() {

        println(Swap(4, 2))

}


```

```
2 4

Program exited.
```

Functions

```
package main

// return values can be given a name
func Split(x, y int) (d, r int) {
        d = x / y
        r = x % y
        return // naked return
}

func main() {

        println(Split(17, 3))
}

```

```
5 2

Program exited.
```

Conditionals

```
1 package main
2
3 func Max(x, y int) int {
4         // if syntax
5         // no (), but {} are mandatory, else {} is optional
6         if x > y {
7                 return x
8         } else {
9                 return y
10        }
11 }
12
13 func main() {
14        println(Max(5, 3))
15 }
16
```

```
5

Program exited.
```

Functions as values

```
package main

import "fmt"

func main() {

        // functions are values too
        twice := func(x int) int { return 2 * x }
        fmt.Printf("twice has type %T\n", twice)

        apply := func(f func(int) int, x int) int { return f(x) }
        fmt.Printf("apply has type %T\n", apply)

        fmt.Println("apply(twice, 3) =", apply(twice, 3))

}
```

```
twice has type func(int) int
apply has type func(func(int) int, int) int
apply(twice, 3) = 6

Program exited.
```

Cycles

```
1 package main
2
3 func main() {
4         // for syntax
5         // no (), but {} are mandatory
6         sum := 0
7         for i := 1; i < 10; i += 2 {
8                 sum += i
9         }
10        println("The sum of odd numbers from 1 to 9 is", sum)
11
12 }
13
14
15
16
```

```
The sum of odd numbers from 1 to 9 is 25

Program exited.
```

Cycles

```
package main

func main() {
        // no while, use for instead
        mul := 2
        for mul < 1000 {
                mul *= mul
        }
        println(mul)

        // for {...} infinite cyle when no condition is present
}
```

```
65536

Program exited.
```

Switch syntax

```
package main

import (
        "fmt"
        "time"
)

func main() {
        t := time.Now()
        fmt.Println("It's", t)

        // switch syntax
        // evaluate cases top-down, first encountered is taken
        // !! unless fallthrough is used the other cases are not considered
        fmt.Println("When's Saturday?")
        today := t.Weekday()
        switch time.Saturday {
        case today + 0:
                println("Today.")
        case today + 1:
                println("Tomorrow.")
        // possibly a default case
        default:
                println("Too far away.")
        }
}

```

```
It's 2009-11-10 23:00:00 +0000 UTC m=+0.000000001
When's Saturday?
Too far away.

Program exited.
```

Switch syntax

```
1 package main
2
3 import (
4         "fmt"
5         "time"
6 )
7
8 func main() {
9         t := time.Now()
10        fmt.Println("It's", t)
11
12        // switch can be used without condition
13        switch {
14        case t.Hour() < 12:
15                println("Good morning!")
16        case t.Hour() < 17:
17                println("Good afternoon.")
18        default:
19                println("Good evening.")
20        }
21 }
22
```

```
It's 2009-11-10 23:00:00 +0000 UTC m=+0.000000001
Good evening.

Program exited.
```

Pointers

```
 1 package main
 2
 3 import "fmt"
 4
 5 func main() {
 6         // Go has pointers, but no pointer arithmetic
 7         i := 4
 8         println("i is", i) // see the initial value of i
 9         p := &i
10         fmt.Printf("p is %v (%T)\n", p, p)
11         fmt.Printf("*p is %v (%T)\n", *p, *p)
12         *p = 21             // set i through the pointer
13         println("i is", i) // see the new value of i
14
15 }
16
```

```
i is 4
p is 0xc000098040 (*int)
*p is 4 (int)
i is 21

Program exited.
```

Struct

```go
package main

import "fmt"

type Vertex struct {
        X int
        Y int
}

func main() {
        // create a Vertex
        v := Vertex{1, 2}
        fmt.Printf("v is %v (%T)\n", v, v)

        // access a field using dot notation
        v.X = v.Y * 2
        fmt.Println("v is", v)
}

```

```
v is {1 2} (main.Vertex)
v is {4 2}

Program exited.
```

Struct

```go
package main

import "fmt"

type Vertex struct {
        X int
        Y int
}

func main() {
        // create a Vertex
        v := Vertex{1, 2}
        fmt.Printf("v is %v (%T)\n", v, v)

        q := v // copy
        fmt.Printf("q is %v (%T)\n", q, q)
        q.X = 1e5
        fmt.Println("q is", q)
        fmt.Println("v is still", v) // v is unchanged
}

```

```
v is {1 2} (main.Vertex)
q is {1 2} (main.Vertex)
q is {100000 2}
v is still {1 2}

Program exited.
```

Struct

```go
package main

import "fmt"

type Vertex struct {
        X int
        Y int
}

func main() {
        // create a Vertex
        v := Vertex{1, 2}
        fmt.Printf("v is %v (%T)\n", v, v)

        p := &v // alias
        fmt.Printf("p is %v (%T)\n", p, p)
        p.X = 1e5 // p.X same as (*p).X
        fmt.Printf("v is %v (%T)\n", v, v)
}


```

```
v is {1 2} (main.Vertex)
p is &{1 2} (*main.Vertex)
v is {100000 2} (main.Vertex)

Program exited.
```

Struct

```go
package main

import "fmt"

type Vertex struct {
        X int
        Y int
}

func main() {
        // field assignment
        v1 := Vertex{1, 2} // has type Vertex
        v2 := Vertex{Y: 1} // X:0 is implicit
        v3 := Vertex{}     // X:0 and Y:0
        fmt.Println(v1, v2, v3)
}

```

```
{1 2} {0 1} {0 0}

Program exited.
```

Methods

```go
package main

import "fmt"
import "math"

type Vertex struct {
    X int
    Y int
}

// A method is a function with a special receiver argument
func (this Vertex) Dist() float64 {
    return math.Sqrt(float64(this.X*this.X + this.Y*this.Y))
}

// A method is a function with a special receiver argument
// (need a pointer to change the state... and avoid copying)
// (value methods can be invoked on pointers and values,
// but pointer methods can only be invoked on pointers)
// (When the value is addressable, the language takes care
// of the common case of invoking a pointer method on a value
// by inserting the address operator automatically)
// methods can be invoked on nil objects
func (this *Vertex) Move(dx, dy int) {
    if this != nil {
        this.X += dx
        this.Y += dy
    }
}

func main() {
    v := Vertex{3, 4}
    // method invocation
    fmt.Println(v)
    fmt.Println(v.Dist())

    v.Move(6, 8)
    fmt.Println(v)
    fmt.Println(v.Dist())
}
```

```go
1 package main
2
3 import (
4         "fmt"
5         "math"
6 )
7
8 type Vertex struct {
9         X int
10        Y int
11 }
12
13 // A method is a function with a special receiver argument
14 func (this Vertex) Dist() float64 {
15        return math.Sqrt(float64(this.X*this.X + this.Y*this
16 }
17
```

```
{3 4}
5
{9 12}
15

Program exited.
```

Methods

```go
package main

import "fmt"
import "math"

type Vertex struct {
    X int
    Y int
}

// A method is a function with a special receiver argument
func (this Vertex) Dist() float64 {
    return math.Sqrt(float64(this.X*this.X + this.Y*this.Y))
}

// An interface type is defined as a set of method signatures
type Distable interface {
    Dist() float64
}

type Point int

func (p Point) Dist() float64 {
    return float64(p)
}

func main() {
    var i1, i2 Distable
    //  A variable of interface type can hold any value that implements those methods
    // (don't need to declare that the type implements the interface)
    i1 = Vertex{3, 4}
    i2 = Point(3)
    fmt.Printf("i1.Dist()=%v\n", i1.Dist())
    fmt.Printf("i2.Dist()=%v\n", i2.Dist())

}
```

```go
1 package main
2
3 import (
4         "fmt"
5         "math"
6 )
7
8 type Vertex struct {
9         X int
10        Y int
11 }
12
13 // A method is a function with a special
14 func (this Vertex) Dist() float64 {
15        return math.Sqrt(float64(this.X*t
16 }
17
18 // An interface type is defined as a set
19 type Distable interface {
```

```
i1.Dist()=5
i2.Dist()=3

Program exited.
```

Hello concurrency

```
1 package main
2
3 func main() {
4         println("Hello")
5         println("World")
6 }
7
```

```
Hello
World

Program exited.
```

Hello concurrency

```
1 package main
2
3 func main() {
4         // launch a goroutine
5         go println("Hello")
6         println("World")
7         // Hey, what happens? Where is Hello?
8         // (when main ends all its goroutines are terminated)
9 }
10
```

```
World

Program exited.
```

Hello concurrency

```
1 package main
2
3 import "time"
4
5 func main() {
6         // launch a goroutine
7         go println("Hello")
8         println("World")
9         time.Sleep(1000)
10        // Here is Hello!
11 }
12
```

```
World
Hello

Program exited.
```

Hello concurrency

```
1 package main
2
3 // let's sync on a channel
4 func main() {
5         done := make(chan bool)
6         // launch a goroutine
7         go func() {
8                 println("Hello")
9                 done <- true // send value true on channel done
10        }()
11        println("World")
12        // wait on channel done, ignore received value
13        <-done
14 }
15
```

```
World
Hello

Program exited.
```

Hello concurrency

```
package main

// Hello takes a channel for exchanging booleans
func Hello(done chan bool) {
        println("Hello")
        done <- true // send value true on channel done
}

func main() {
        // create a channel for sending booleans
        done := make(chan bool)
        go Hello(done) // launch a goroutine
        println("World")
        // wait on channel done, ignore received value
        <-done // receive a value from channel done
        // this way World may get printed before Hello
}

```

```
World
Hello

Program exited.
```

Hello concurrency

```
1 package main
2
3 // Hello takes a channel for exchanging booleans
4 func Hello(done chan bool) {
5         println("Hello")
6         done <- true // send value true on channel done
7 }
8
9 func main() {
10        done := make(chan bool)
11        go Hello(done)
12        <-done
13        // this way Hello gets printed before World
14        println("World")
15 }
16
```

```
Hello
World

Program exited.
```

Hello deadlocks

```
1 package main
2
3 func main() {
4         c := make(chan int) // create a channel for sending integers
5         c <- 245            // send 245 (but sending is blocking!)
6         n := <-c            // receive from c and store the value in n
7         println(n)
8 }
9
```

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        /tmp/sandbox4275027505/prog.go:5 +0x2d

Program exited.
```

Buffering

```
1 package main
2
3 func main() {
4         c := make(chan int, 1) // create a buffered channel for sending integers
5         c <- 245               // send 245 (now sending is not blocking!)
6         n := <-c               // receive from c and store the value in n
7         println(n)
8 }
9
```

```
245

Program exited.
```

Communicating goroutines

```
1 package main
2
3 func main() {
4         c := make(chan int)
5         // do the sending in an anonymous goroutine
6         go func() {
7                 c <- 245
8         }()
9         n := <-c
10        println(n)
11 }
12
13
```

```
245

Program exited.
```

Communicating goroutines

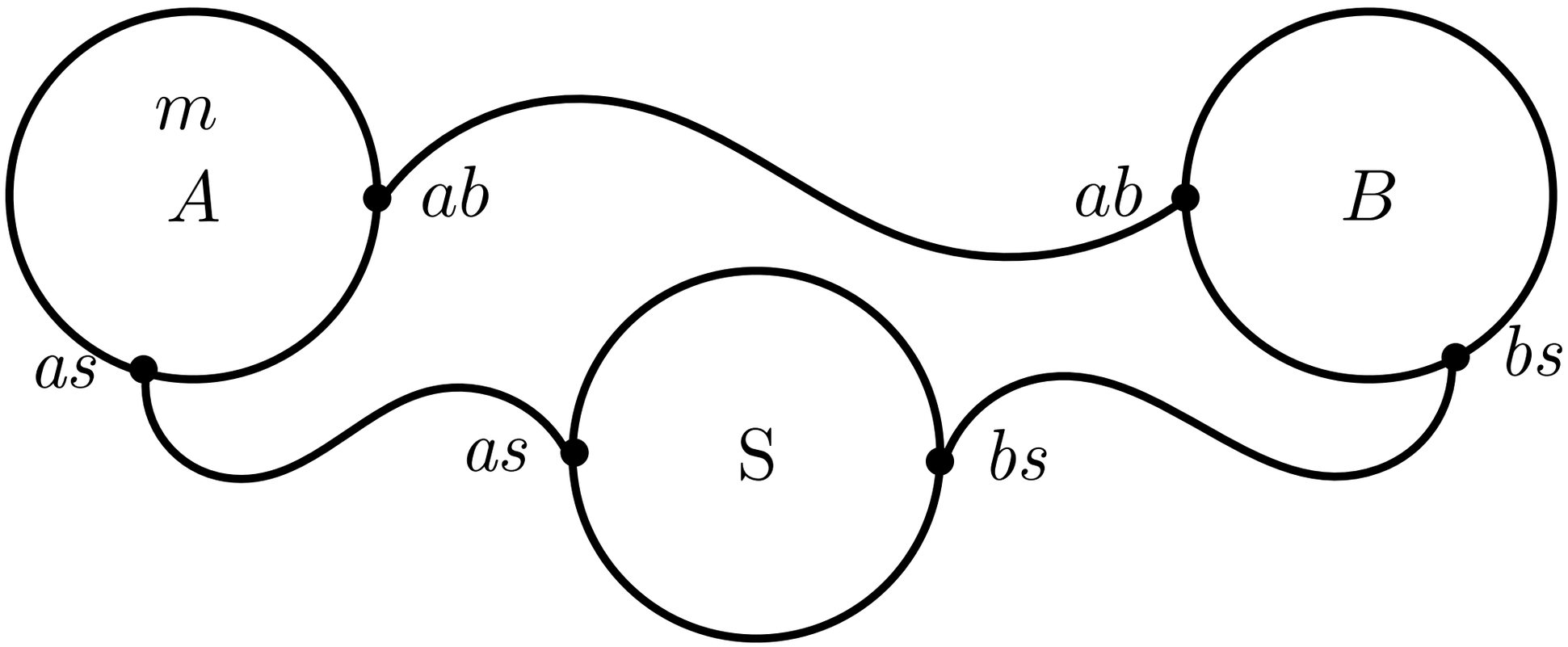
```
1 package main
2
3 func main() {
4         c := make(chan int)
5         // do the sending in an anonymous goroutine
6         go func() {
7                 c <- 245
8         }()
9         // avoid to use variable n
10        println(<-c)
11 }
12
13
```

```
245

Program exited.
```

Name mobility: secrecy





Name mobility

```
func main() {
    as, bs := Serv() // launch server, get secure channels
    go A(as)         // launch A
    B(bs)            // run B
}
```

```
// returns a pair of channels for communicating to the server
func Serv() (as chan chan int, bs chan chan int) {
    // create two channels
    // for sending names of channels for sending integers
    as = make(chan chan int)
    bs = make(chan chan int)
    // launch a goroutine for serving requests
    go func() {
        for {
            // forward messages from as to bs
            c := <-as
            bs <- c
        }
    }()
    return  // naked return
}
```

Name mobility

```
// for N times:
//     creates a channel ch
//     sends the channel to the server on as
//     sends an integer on ch
func A(as chan chan int) {
    for i := 0; i < N; i++ {
        ch := make(chan int)
        fmt.Printf("created %v (%T) for sending %v\n", ch, ch, i)
        as <- ch // send ch to the server
        ch <- i  // send i on ch
    }
}

// for N times:
//     receives a channel ch from the server
//     receives an integer on ch
func B(bs chan chan int) {
    for i := 0; i < N; i++ {
        ch := <-bs
        n := <-ch
        fmt.Printf("received %v on %v\n", n, ch)
    }

}
```

Name mobility

```
1 package main
2
3 import "fmt"
4
5 const N = 3
6
7 // returns a pair of channels for communicating to the server
8 func Serv() (as chan chan int, bs chan chan int) {
9         // create two channels
10        // for sending names of channels for sending integers
11        as = make(chan chan int)
12        bs = make(chan chan int)
13        // launch a goroutine for serving requests
```

```
created 0xc000076150 (chan int) for sending 0
received 0 on 0xc000076150
created 0xc0000761c0 (chan int) for sending 1
received 1 on 0xc0000761c0
created 0xc000076230 (chan int) for sending 2
received 2 on 0xc000076230

Program exited.
```

Closing channels

```
// for N times
//     creates a channel ch
//     sends the channel to the server on as
//     sends an integer on ch
// then closes the communication with the server
func A(as chan chan int) {
    for i := 0; i < N; i++ {
        ch := make(chan int)
        fmt.Printf("created %v (%T) for sending %v\n", ch, ch, i)
        as <- ch // send ch to the server
        ch <- i  // send i on ch
    }
    close(as) // close channel as shared with server
}

// while bs has not been closed
//     receives a channel ch from the server
//     receives an integer on ch
func B(bs chan chan int) {
    // until bs is active
    for ch, ok := <-bs; ok; ch, ok = <-bs {
        n := <-ch
        fmt.Printf("received %v on %v\n", n, ch)
    }
    println("done")

}
```

Closing channels

```go
// returns a pair of channels for communicating to the server
func Serv() (as chan chan int, bs chan chan int) {
    // create two channels
    // for sending names of channels for sending integers
    as = make(chan chan int)
    bs = make(chan chan int)
    // launch a goroutine for serving requests
    go Fwd(as, bs)
    return // naked return
}

func Fwd(as chan chan int, bs chan chan int) {
    // until as is active
    for c, ok := <-as; ok; c, ok = <-as {
        // forward messages from as to bs
        bs <- c
    }
    close(bs) // close channel bs shared with B
}
```

Closing channels

```
1 package main
2
3 import "fmt"
4
5 const N = 3
6
7 func main() {
8         as, bs := Serv() // launch server, get secure channels
9         go A(as)         // launch A
10        B(bs)            // run B
11 }
12
13 // returns a pair of channels for communicating to the server
```

```
created 0xc00009e150 (chan int) for sending 0
created 0xc00009e1c0 (chan int) for sending 1
received 0 on 0xc00009e150
received 1 on 0xc00009e1c0
created 0xc00009e230 (chan int) for sending 2
received 2 on 0xc00009e230
done

Program exited.
```

Range

```go
package main

const N = 5

func Send(c chan<- int) {
    for i := 0; i < N; i++ {
        println("sending", i)
        c <- i
    }
    close(c)
}

func Rec(c <-chan int, d chan<- bool) {
    for n := range c {
        println("got", n)
    }
    // alternatively:
    // done<- true
    close(d)
}

func main() {
    c := make(chan int)
    done := make(chan bool)
    go Send(c)
    go Rec(c, done)
    // alternatively:
    // <-done
    for range done { }
}
```

Press Esc to move out of the editor.

```go
1 package main
2
3 const N = 5
4
5 func Send(c chan<- int) {
6         for i := 0; i < N; i++ {
7                 println("sending", i)
8                 c <- i
9         }
10        close(c)
11 }
```

```
sending 0
sending 1
got 0
got 1
sending 2
sending 3
got 2
got 3
sending 4
got 4

Program exited.
```

Select (many to one)

```go
package main

const N = 3
const M = 2

func Send(c chan<- int, done chan<- bool) {
    for i := 0; i < M; i++ {
        println("sending", i)
        c <- i
    }
    done <- true
}

func Rec(c <-chan int, d <-chan bool, k int, done chan<- bool) {
    for k > 0 {
        select {
        case n := <-c:
            println("got", n)
        case <-d:
            k--
        }
    }
    done <- true
}

func main() {
    c := make(chan int)
    d := make(chan bool)
    done := make(chan bool)
    for i := 0; i < N; i++ {
        go Send(c, d)
    }
    go Rec(c, d, N, done)
    <-done
}
```

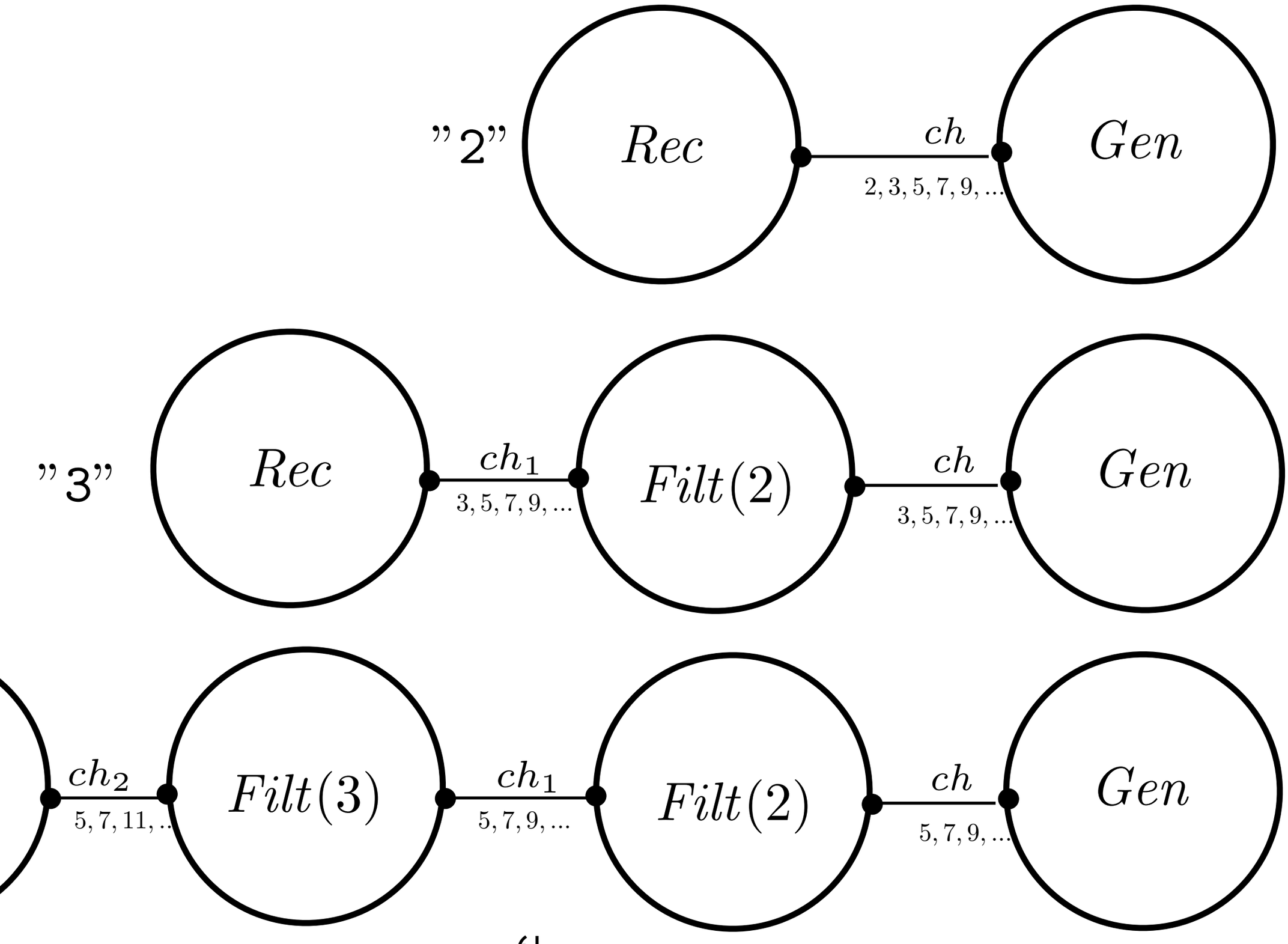
Press Esc to move out of the editor.

```go
1 package main
2
3 const N = 3
4 const M = 2
5
6 func Send(c chan<- int, done chan<- bool) {
7         for i := 0; i < M; i++ {
8                 println("sending", i)
9                 c <- i
10        }
11        done <- true
```

```
sending 0
sending 0
got 0
sending 1
got 0
sending 1
got 1
sending 0
got 1
got 0
sending 1
got 1

Program exited.
```

Concurrent prime sieve



A concurrent prime sieve

```
// A concurrent prime sieve
package main

const N = 4000 // prime numbers to print

// The prime sieve: Daisy-chain Filter processes.
func main() {
    ch := Generate() // Launch Generate goroutine.
    Rec(ch,N)
}

// remember typing :
// chan<- int: used only for sending
// <-chan int: used only for receiving

// Create and return a channel ch
// where the sequence 2, 3, 5, 7, 9, ... will be sent by Gen
func Generate() <-chan int {
    out := make(chan int)
    go Gen(out)
    return out
}
```

A concurrent prime sieve

```
// Send the sequence 2, 3, 5, 7, 9, ... on out
func Gen(out chan<- int) {
    out <- 2
    // note the missing guard!
    for i := 3; ; i = i + 2 {
        out <- i
    }
}
```

A concurrent prime sieve

```
// receives the list of primes and prints each of them
func Rec(in <-chan int, count int) {
    for i := 1; i <= count; i++ {
        // receives a prime
        prime := <-in
        println(prime,"is prime n.",i)
        // install a new filter for the newly received prime
        in = Filter(in, prime) // updates in
    }
}
```

A concurrent prime sieve

```
// Copy the values from channel 'in' to channel 'out'
// removing those divisible by 'prime'
func Filter(in <-chan int, prime int) <-chan int {
    out := make(chan int)
    go Fil(in,out,prime)
    return out
}

func Fil(in <-chan int, out chan<- int, prime int) {
    // println("--- Created filter", prime, "from", in, "to", out)
    for {
        i := <-in // Receive value from 'in'
        // println("--- Filter", prime, "receives", i, "on", in)
        if i%prime != 0 {
            // println("--- Filter", prime, "sends", i, "to", out)
            out <- i // Send 'i' to 'out'
        } else {
            // println("--- Filter", prime, "discards", i)
        }
    }
}
```

A concurrent prime sieve

Press Esc to move out of the editor.

```
// A concurrent prime sieve
package main

const N = 4000 // prime numbers to print

// The prime sieve: Daisy-chain Filter processes.
func main() {
        ch := Generate() // Launch Generate goroutine.
        Rec(ch, N)
}

```

```
37643 is prime n. 3987
37649 is prime n. 3988
37657 is prime n. 3989
37663 is prime n. 3990
37691 is prime n. 3991
37693 is prime n. 3992
37699 is prime n. 3993
37717 is prime n. 3994
37747 is prime n. 3995
37781 is prime n. 3996
37783 is prime n. 3997
37799 is prime n. 3998
37811 is prime n. 3999
37813 is prime n. 4000

Program exited.
```

Exercises

[**Ex. 4**] Write a GoogleGo function that takes one channel `ini` for receiving integers and one channel `ins` for receving strings and returns a channel `outp` where all the messages received on `ini` and `ins` will be paired.
Hint: define a `struct` *to form pairs*

Ex. 4, pairing in Go

```
package main

import "fmt"

type Pair struct {
    N int
    S string
}
```

Ex. 4, pairing in Go

```
package main

import "fmt"

type Pair struct {
    N int
    S string
}

func pairing(ini chan int, ins chan string) (outp chan Pair) {

    return
}
```

Ex. 4, pairing in Go

```
package main

import "fmt"

type Pair struct {
    N int
    S string
}
```

```
func pairing(ini chan int, ins chan string) (outp chan Pair) {
    outp = make(chan Pair)









    return
}
```

Ex. 4, pairing in Go

```
package main

import "fmt"

type Pair struct {
    N int
    S string
}

func pairing(ini chan int, ins chan string) (outp chan Pair) {
    outp = make(chan Pair)
    go func() {
        for {




        }
    }()
    return
}
```

Ex. 4, pairing in Go

```
package main

import "fmt"

type Pair struct {
    N int
    S string
}
```

```
func pairing(ini chan int, ins chan string) (outp chan Pair) {
    outp = make(chan Pair)
    go func() {
        for {
            i := <-ini
            s := <-ins

        }
    }()
    return
}
```

Ex. 4, pairing in Go

```
package main

import "fmt"

type Pair struct {
    N int
    S string
}
```

```
func pairing(ini chan int, ins chan string) (outp chan Pair) {
    outp = make(chan Pair)
    go func() {
        for {
            i := <-ini
            s := <-ins
            outp <- Pair{i, s}
        }
    }()
    return
}
```

Ex. 4, pairing in Go

```
func main() {
    chi := make(chan int)
    chs := make(chan string)

}
```

Ex. 4, pairing in Go

```
func main() {
    chi := make(chan int)
    chs := make(chan string)
    chp := pairing(chi,chs)

}
```

Ex. 4, pairing in Go

```
func main() {
    chi := make(chan int)
    chs := make(chan string)
    chp := pairing(chi,chs)
    chi <- 1
    chs <- "Alice"

}
```

Ex. 4, pairing in Go

```
func main() {
    chi := make(chan int)
    chs := make(chan string)
    chp := pairing(chi,chs)
    chi <- 1
    chs <- "Alice"
    v := <- chp
    fmt.Println("got",v)



}
```

Ex. 4, pairing in Go

```
func main() {
    chi := make(chan int)
    chs := make(chan string)
    chp := pairing(chi,chs)
    chi <- 1
    chs <- "Alice"
    v := <- chp
    fmt.Println("got",v)
    chi <- 2
    chs <- "Bob"
    v = <- chp
    fmt.Println("got",v)
}
```

[**Ex. 5**] Write a GoogleGo function that takes two channels `f` and `q` and tries to send the stream of Fibonacci numbers on `f` but quits when it receives `true` on channel `q`. Write a `main` program to test the function by printing the first 10 Fibonacci numbers.

Ex. 5, Go Fibonacci

```
package main

import "fmt"

func fibonacci(f chan int, q chan bool) {

}
```

Ex. 5, Go Fibonacci

```
package main

import "fmt"

func fibonacci(f chan int, q chan bool) {
    x, y := 0, 1
    for {



    }
}
```

Ex. 5, Go Fibonacci

```
package main

import "fmt"

func fibonacci(f chan int, q chan bool) {
    x, y := 0, 1
    for {
        select {



        }
    }
}
```

Ex. 5, Go Fibonacci

```
package main

import "fmt"

func fibonacci(f chan int, q chan bool) {
    x, y := 0, 1
    for {
        select {
        case f <- x:
            x, y = y, x+y


        }
    }
}
```

Ex. 5, Go Fibonacci

```
package main

import "fmt"

func fibonacci(f chan int, q chan bool) {
    x, y := 0, 1
    for {
        select {
        case f <- x:
            x, y = y, x+y
        case <-q:
            return
        }
    }
}
```

Ex. 5, Go Fibonacci

```
func main() {
    fib := make(chan int)
    quit := make(chan bool)

    fibonacci(fib, quit)
}
```

Ex. 5, Go Fibonacci

```
func main() {
    fib := make(chan int)
    quit := make(chan bool)
    go func(){

    }()
    fibonacci(fib, quit)
}
```

Ex. 5, Go Fibonacci

```
func main() {
    fib := make(chan int)
    quit := make(chan bool)
    go func(){
        for i := 0; i < 10; i++ {
            fmt.Println(<-fib)
        }

    }()
    fibonacci(fib, quit)
}
```

Ex. 5, Go Fibonacci

```
func main() {
    fib := make(chan int)
    quit := make(chan bool)
    go func(){
        for i := 0; i < 10; i++ {
            fmt.Println(<-fib)
        }
        quit <- true
    }()
    fibonacci(fib, quit)
}
```