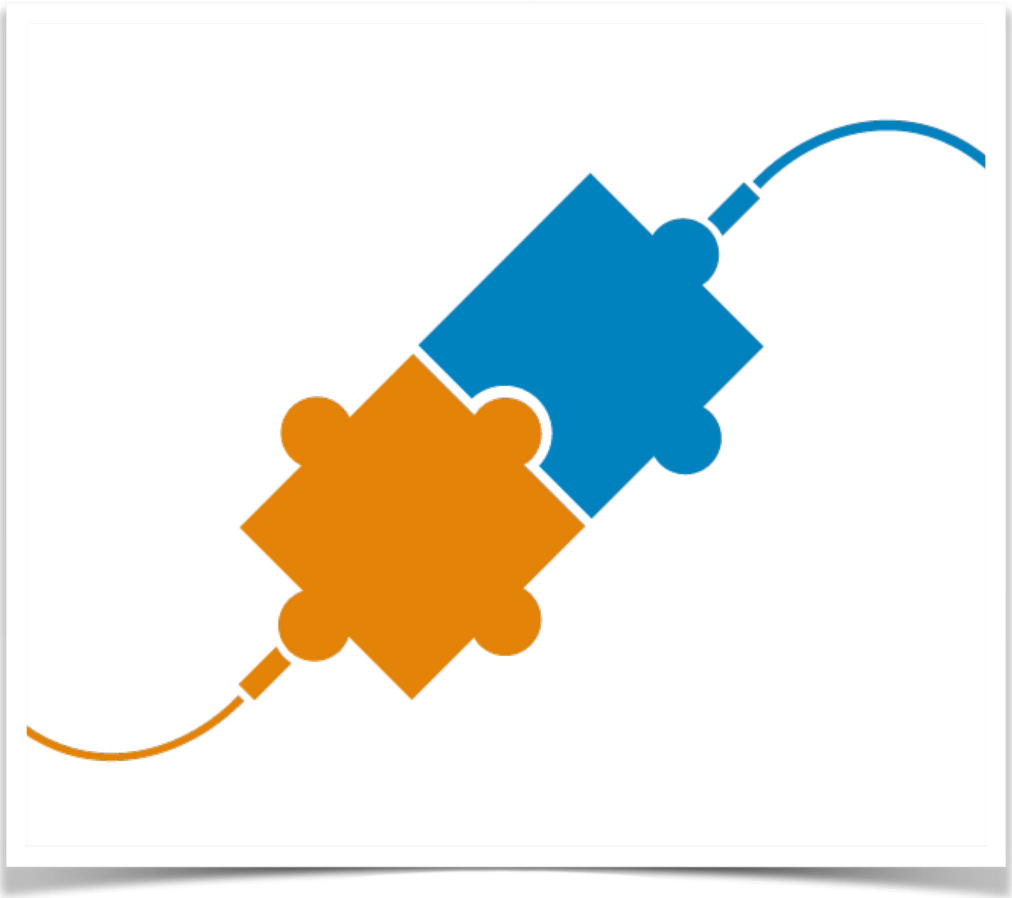


http://didawiki.di.unipi.it/doku.php/
magistraleinformatica/psc/start

PSC 2023/24 (375AA, 9CFU)

Principles for Software Composition

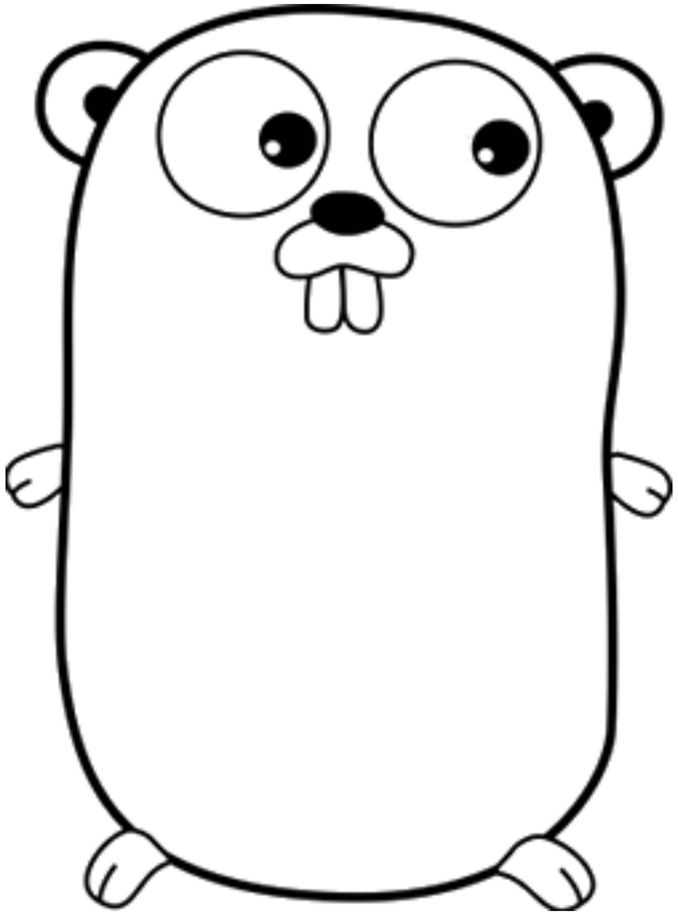
Roberto Bruni
http://www.di.unipi.it/~bruni/

23 - 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

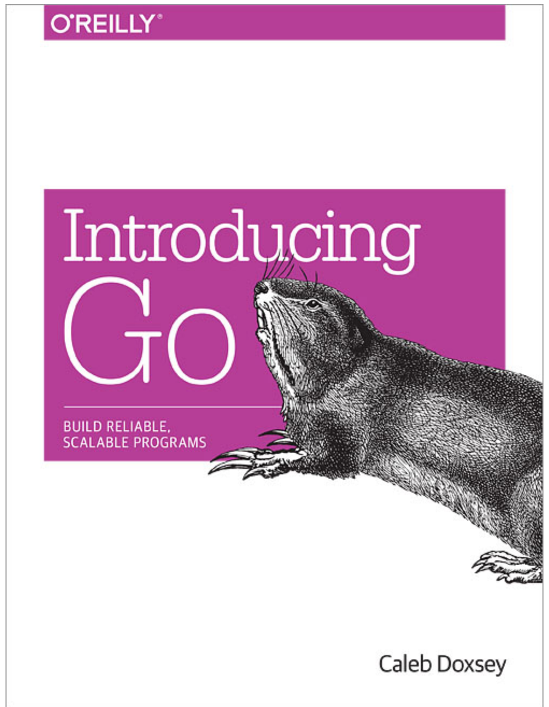
Google Go:
compiled, static types, type inference
(and nice concurrency primitives)

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 2023: version 1.20

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)

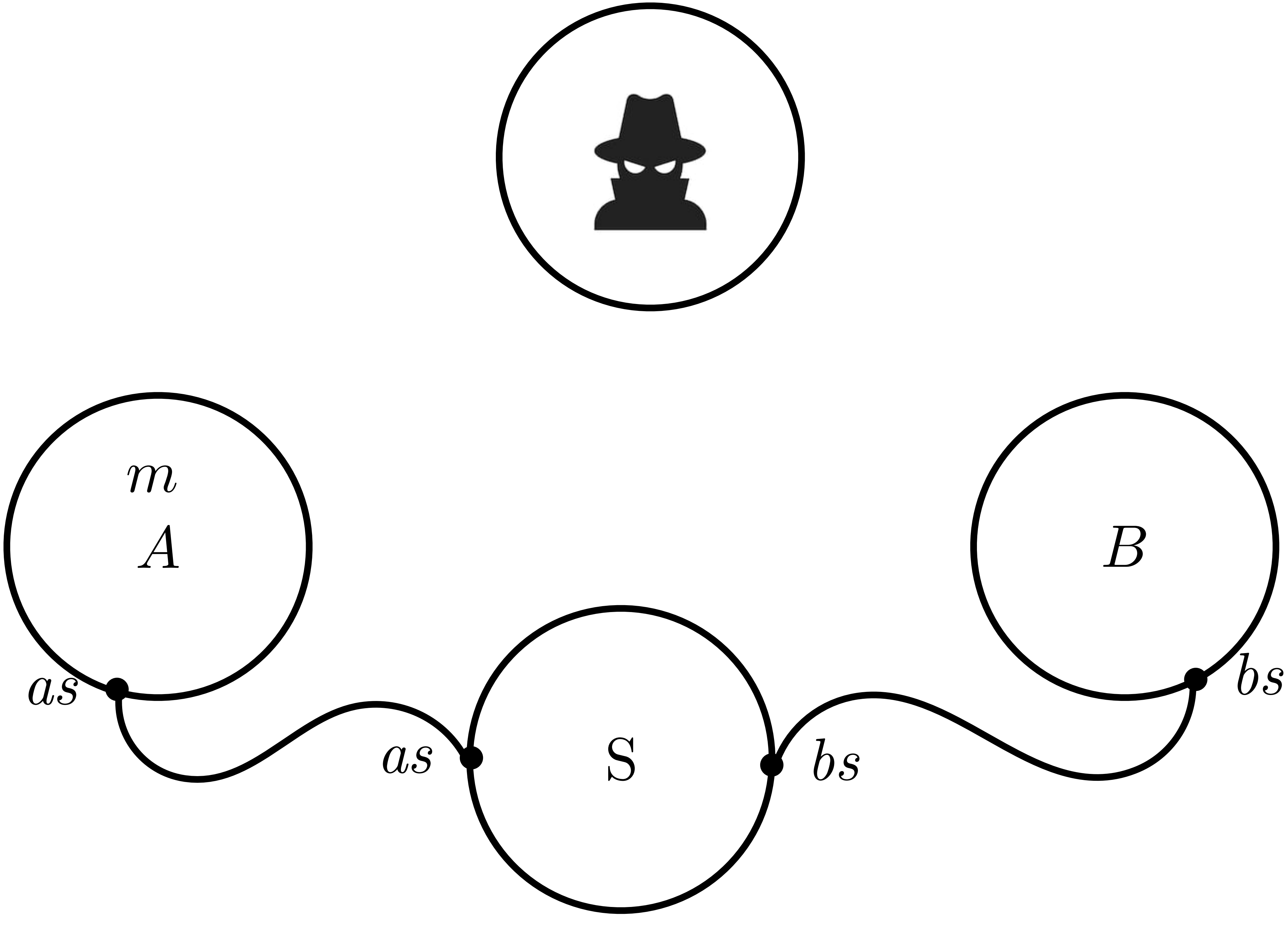
```
var mob = make(chan chan int)
```

a channel for communicating channels

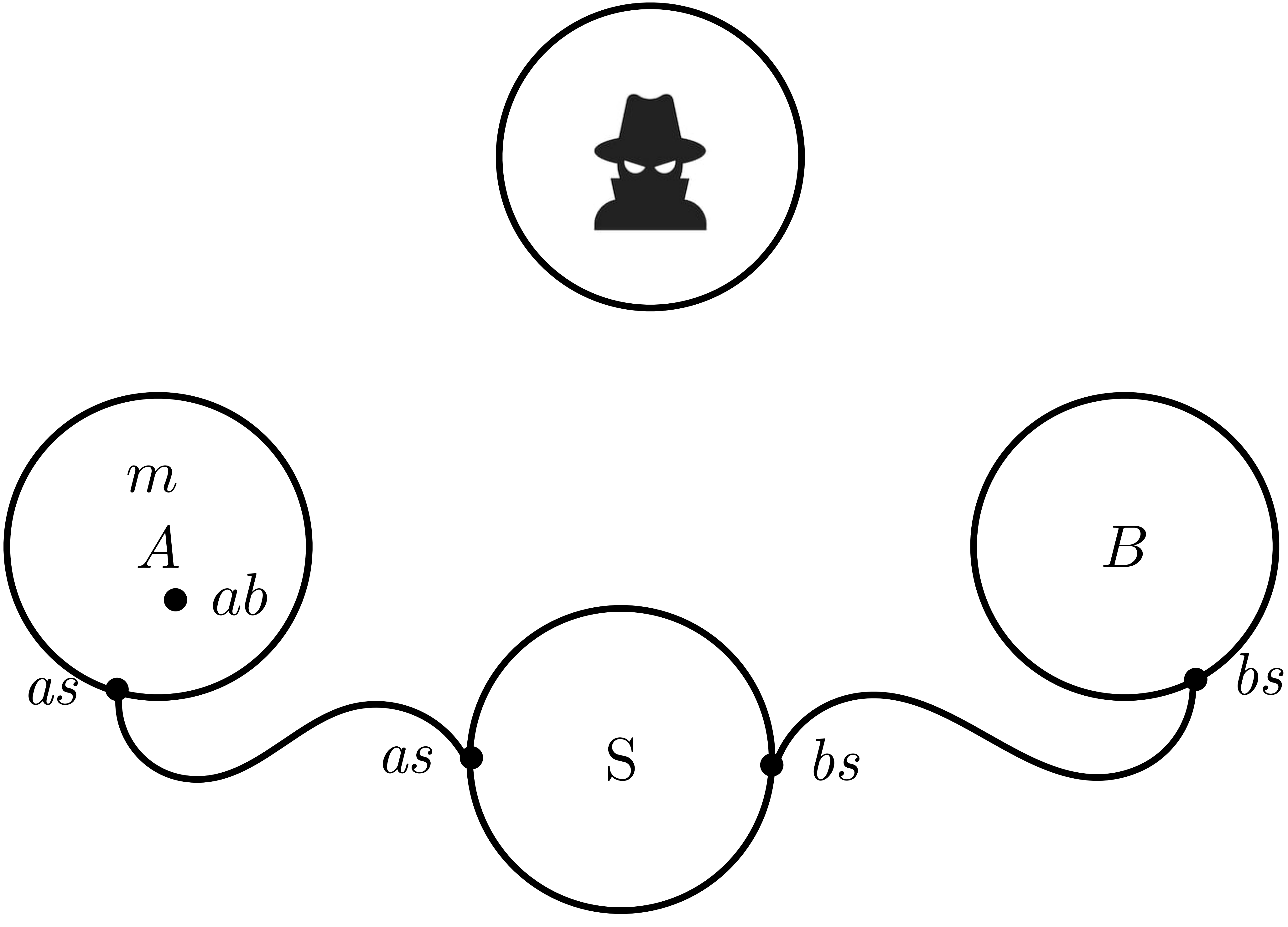
```
mob <- ch
```

send the channel `ch` over `mob`

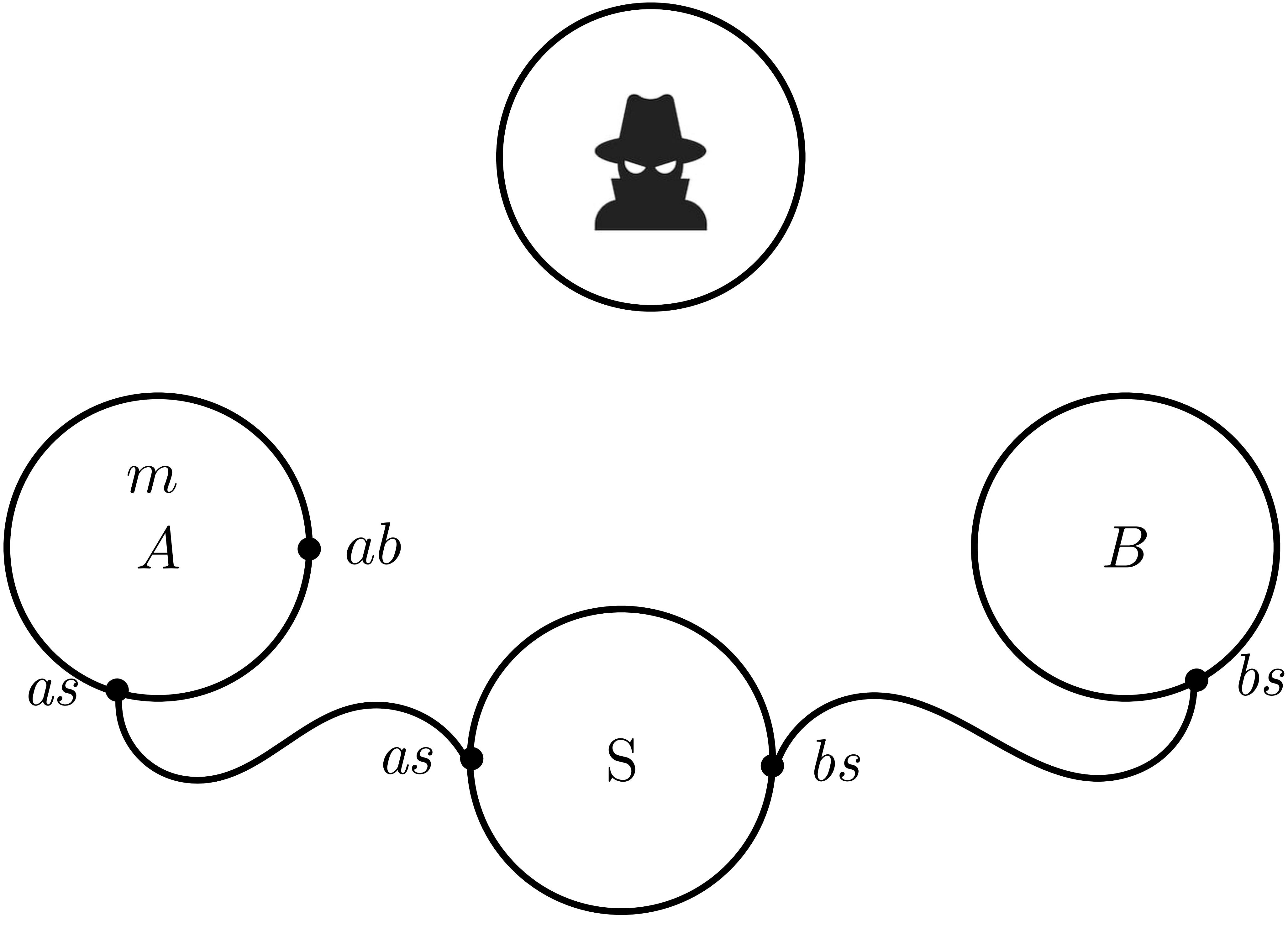
Name mobility: secrecy



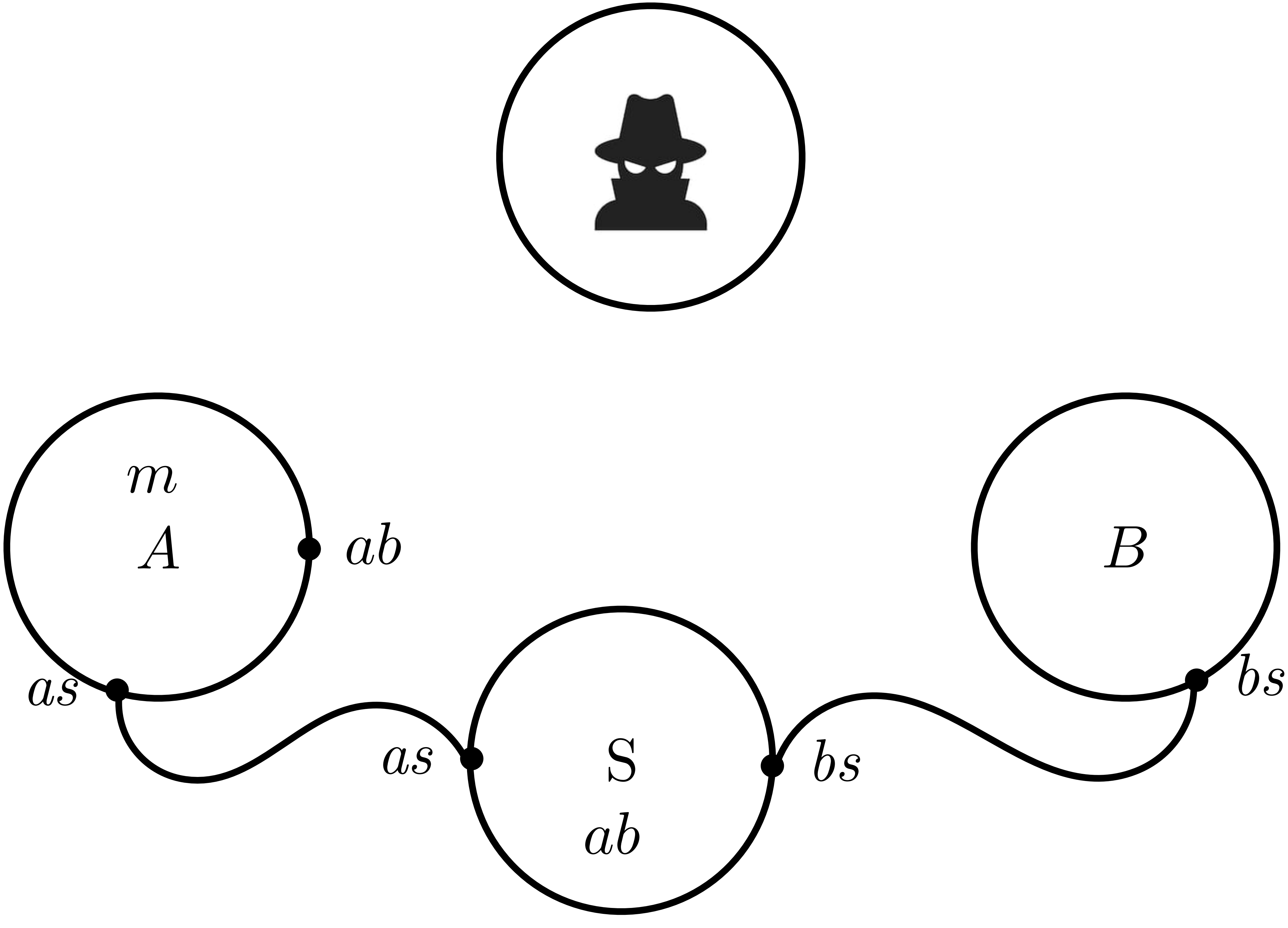
Name mobility: secrecy



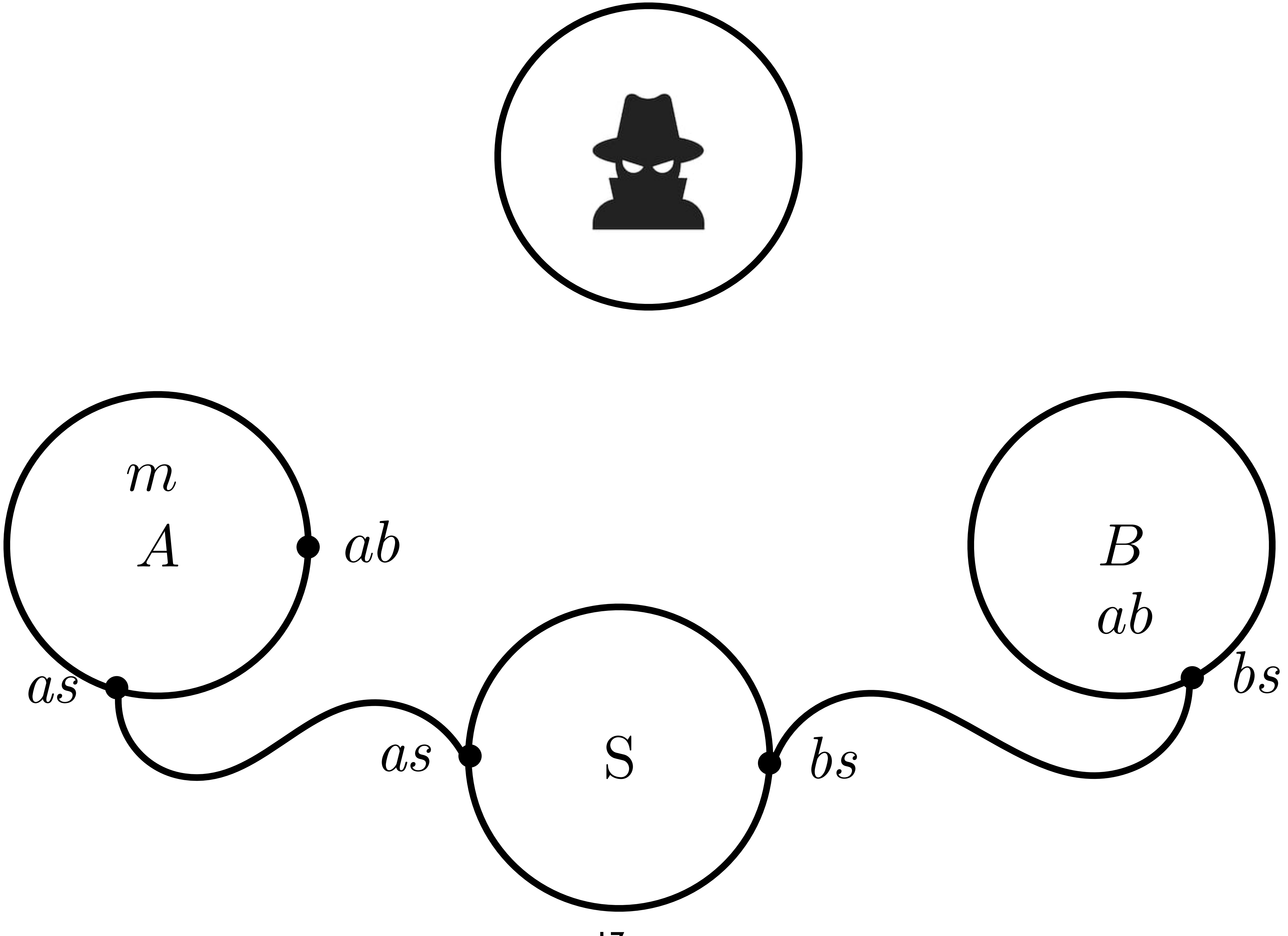
Name mobility: secrecy



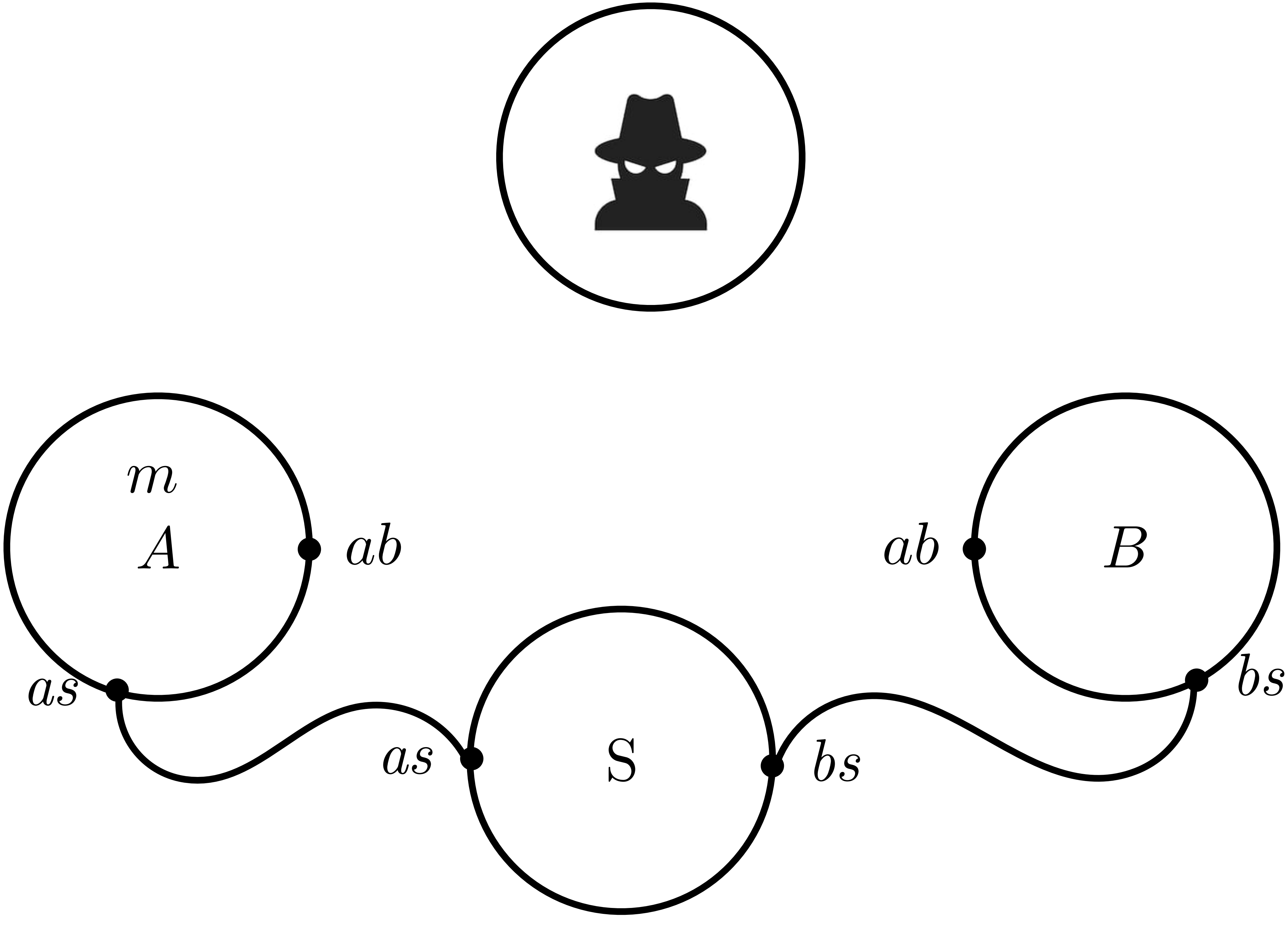
Name mobility: secrecy



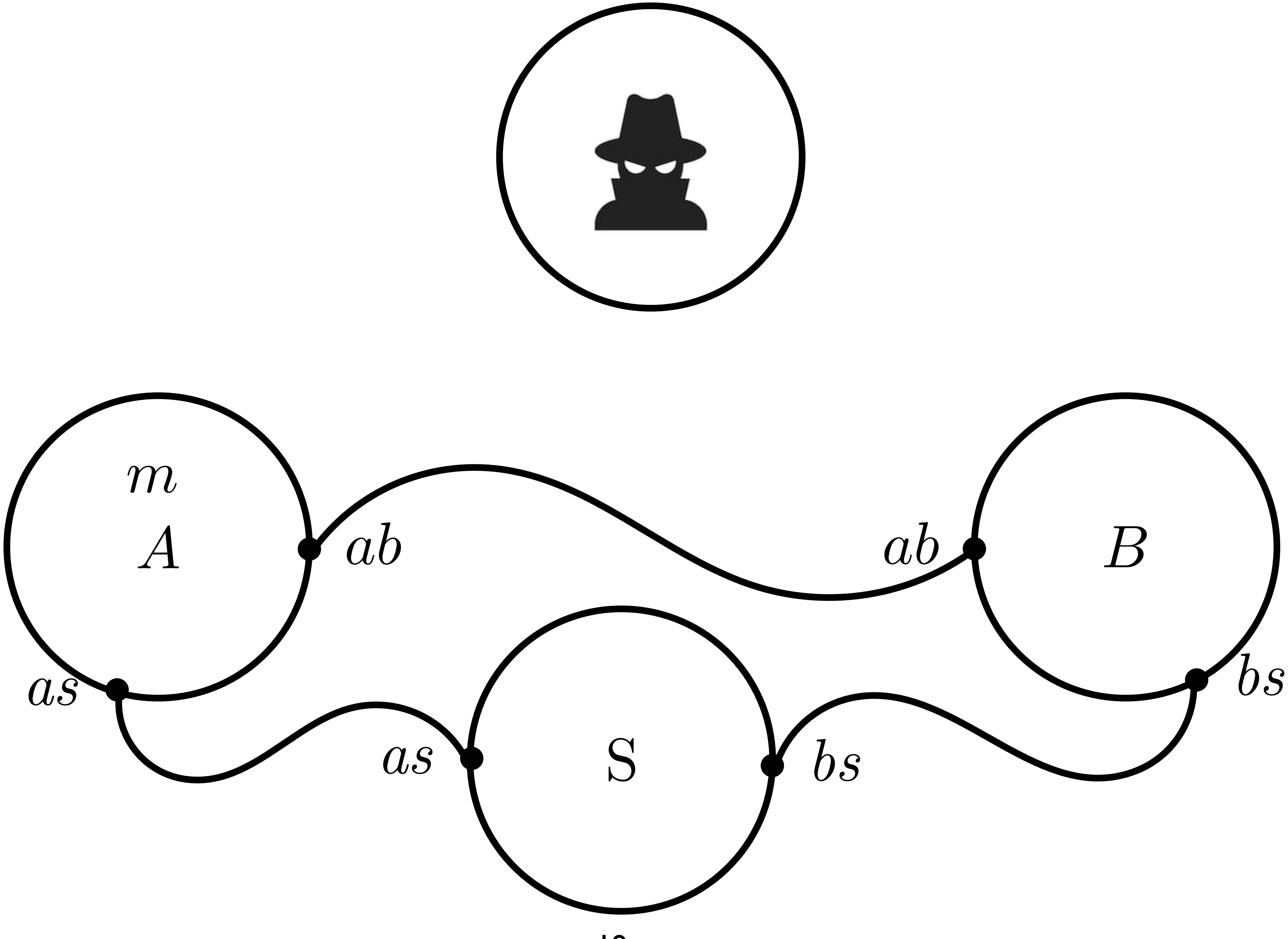
Name mobility: secrecy



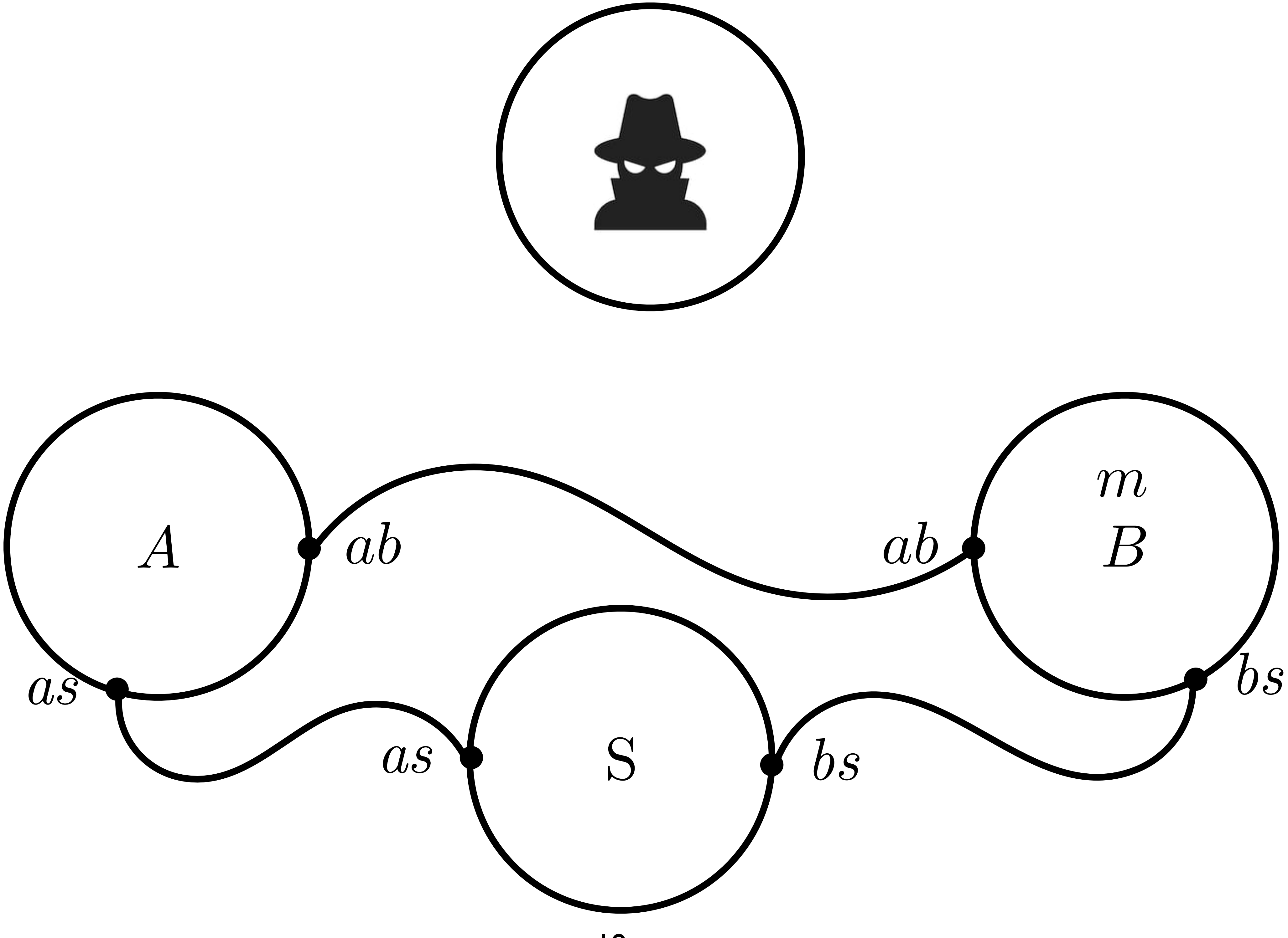
Name mobility: secrecy



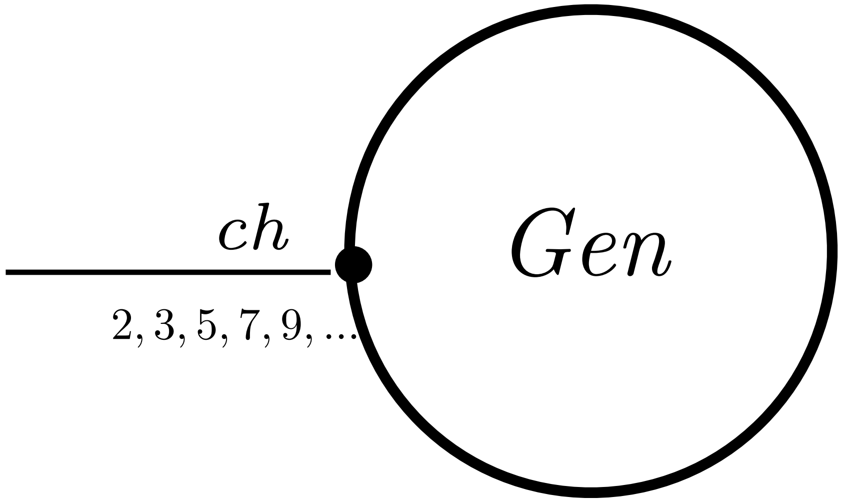
Name mobility: secrecy



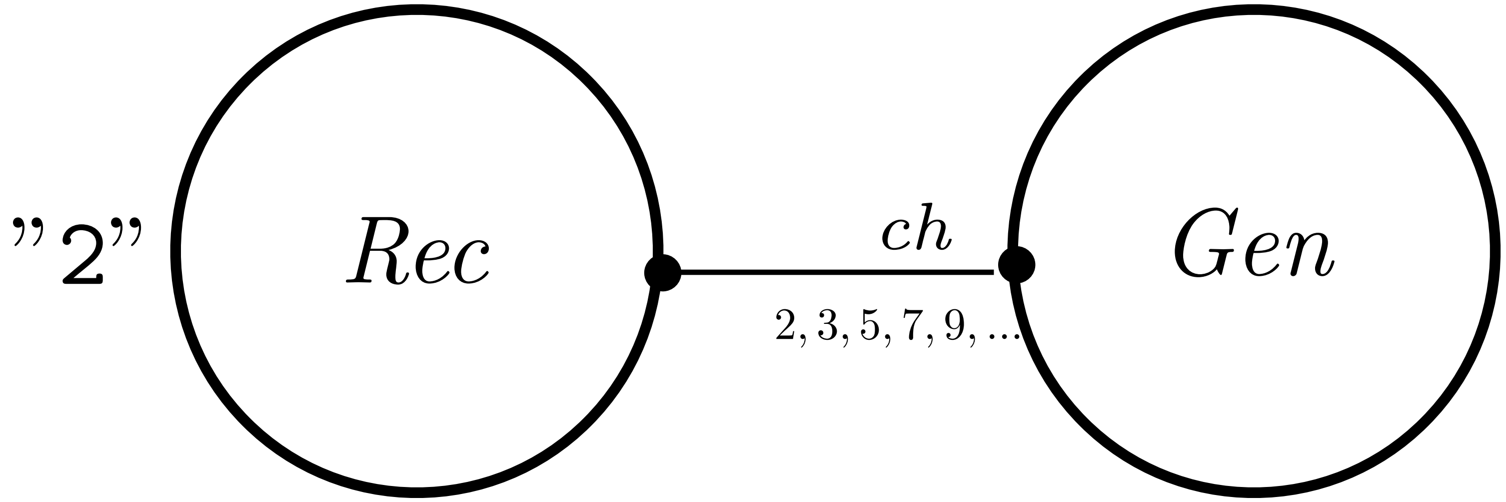
Name mobility: secrecy



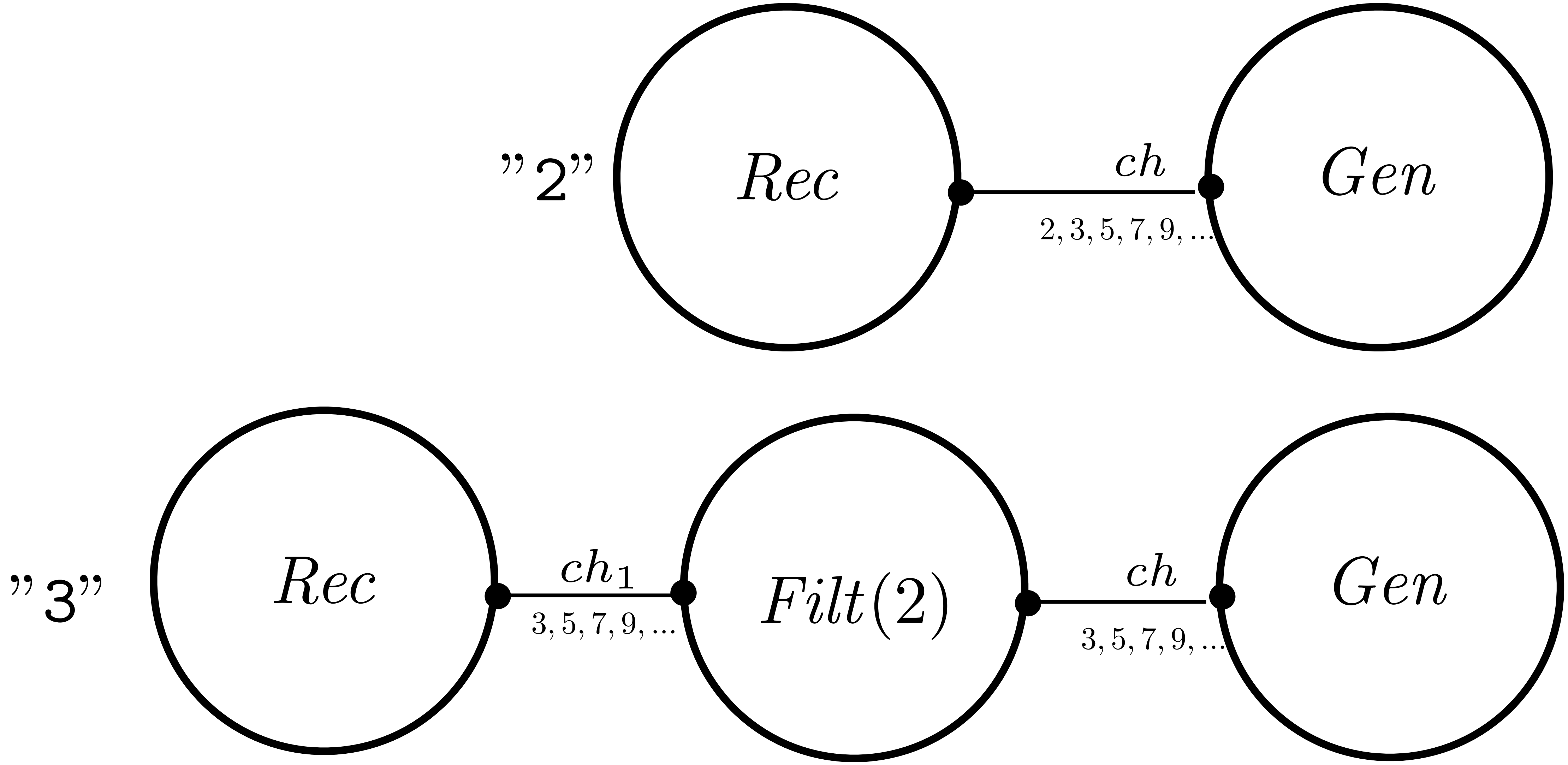
Concurrent prime sieve



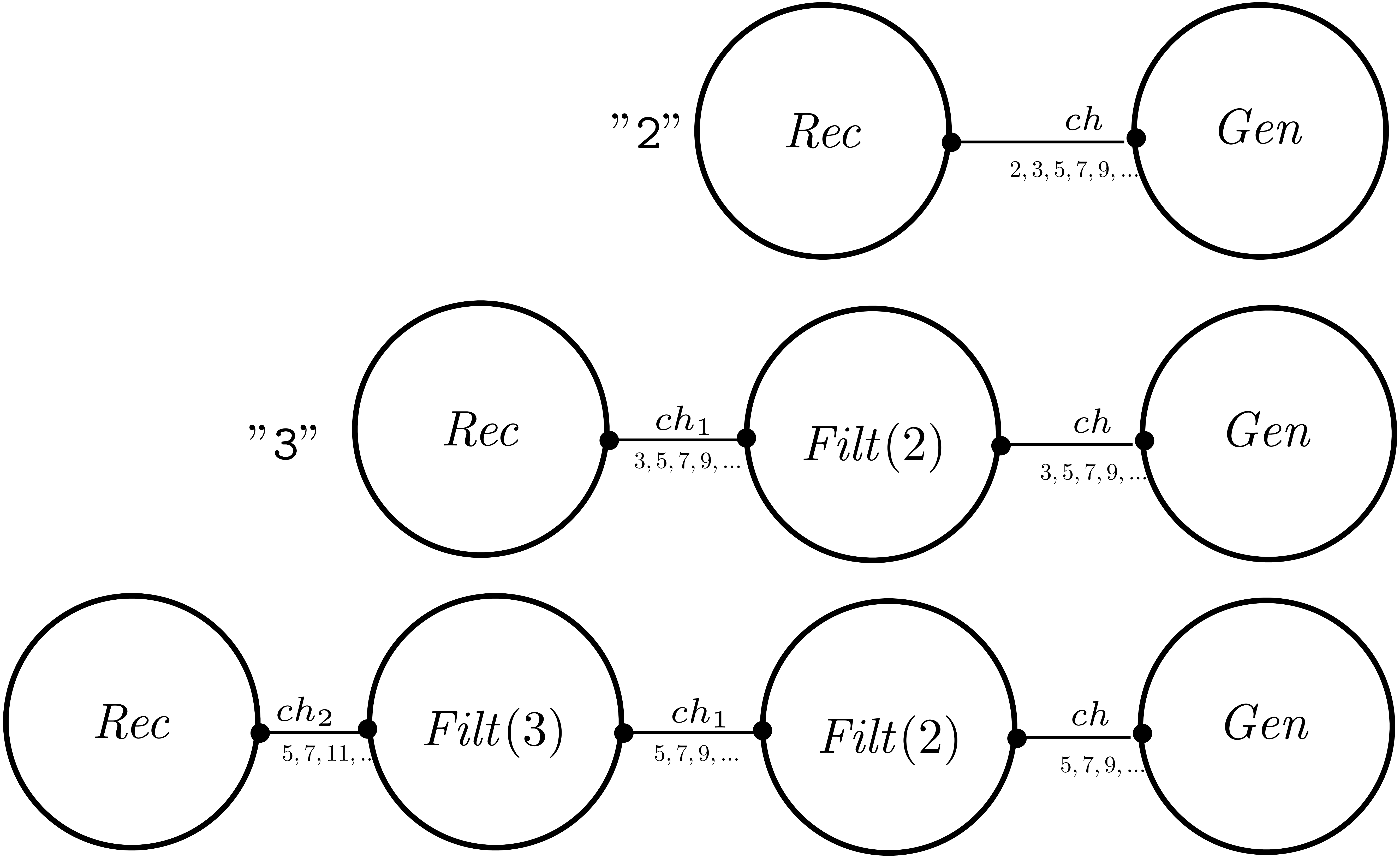
Concurrent prime sieve



Concurrent prime sieve



Concurrent prime sieve



Go playground

```go
// You can edit this code!
// Click here and start typing.
package main

import "fmt"

func main() {
	fmt.Println("Hello, 世界")
}
```

```
Hello, 世界

Program exited.
```