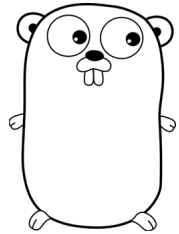
Principles of Concurrent and Distributed Programming

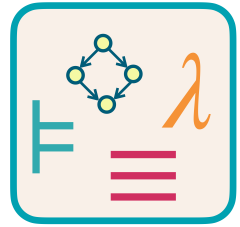
Emilio Tuosto

Academic Year 2025/2026

January 2026

Channel-based concurrency

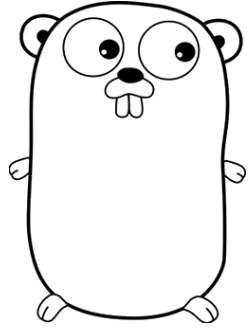




slides are courtesy of R. Bruni and F. Bonchi

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

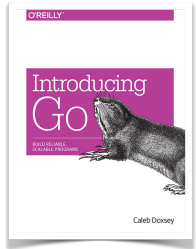
Google Go:
compiled, static types, type inference
(and nice concurrency primitives)

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

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

```go
package main

import "time"

func main() {
        // launch a goroutine
        go println("Hello")
        println("World")
        time.Sleep(1000)
        // Here is Hello!
}

```

```
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

```
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         // create a channel for sending booleans
11         done := make(chan bool)
12         go Hello(done) // launch a goroutine
13         println("World")
14         // wait on channel done, ignore received value
15         <-done // receive a value from channel done
16         // this way World may get printed before Hello
17 }
18
```

```
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

```go
package main

func main() {
        c := make(chan int) // create a channel for sending integers
        c <- 245            // send 245 (but sending is blocking!)
        n := <-c            // receive from c and store the value in n
        println(n)
}
```

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

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

Program exited.
```

Buffering

```go
package main

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

```

```
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

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

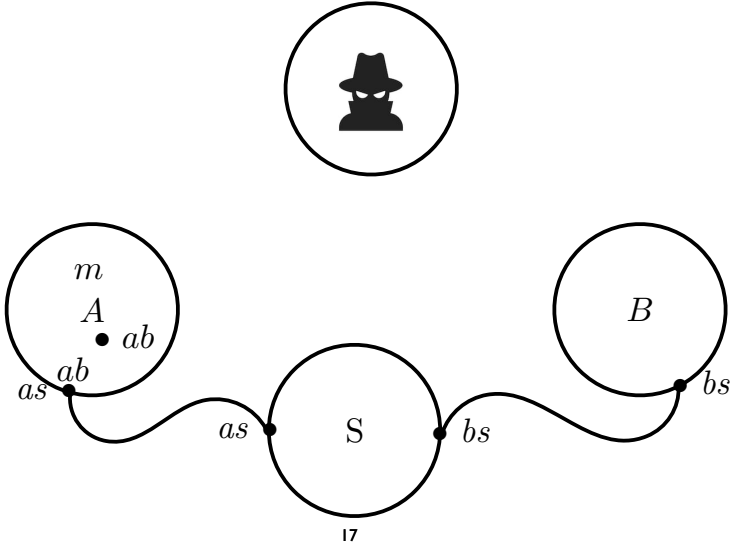
```
var mob = make(chan chan int)
```

a channel for communicating channels

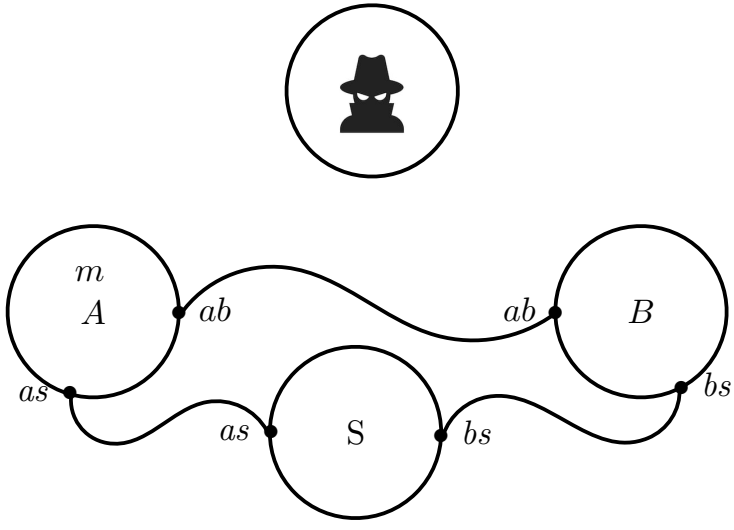
```
mob <- ch
```

send the channel `ch` over `mob`

Name mobility: secrecy



Name mobility: secrecy



Name mobility

```
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

```
// 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
}
```

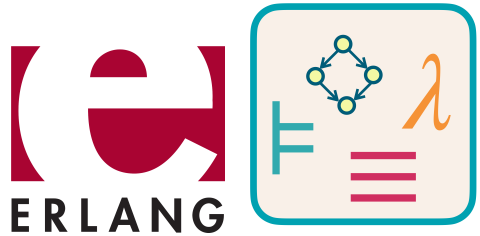
Closing channels

```
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.
```

Actor-based concurrency



wslides are courtesy of R. Bruni and F. Bonchi

Erlang: a concurrent programming language

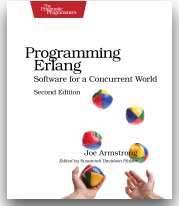
http://www.erlang.org/

Erlang: origins

named after Danish mathematician A. K. Erlang

1986: first experimentation at Ericsson, Sweden
1989: internal use only
1990: sold as a product
1998: open source

Joe Armstrong, “Programming Erlang”, ch.1-5, 11-12



Features

declarative (functional, Prolog) programming

arbitrary size integers, tuples, lists, functions, higher-order

atoms everywhere

dynamically typed

open source

unfriendly syntax

variables are assigned only once

left-to-right evaluation, no pointers, no object-orientation

Features: concurrency

concurrent and distributed programming

asynchronous message passing
(no locks, no mutexes)

fault tolerance

hot swapping code

erlang processes are cheap

automatic memory allocation and garbage collection

can handle large telecom applications

Erl

Erlang: erl

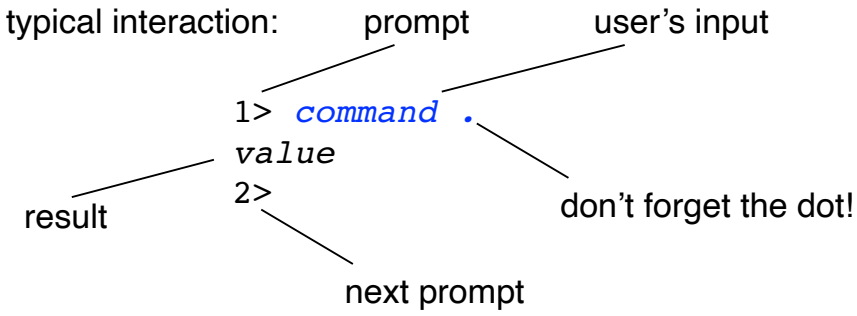
erl is the Erlang VM emulator

interactive shell or interpreter, executing read-eval-print loop

programmers enter expressions / declarations one at a time

they are compiled / executed

erl expressions



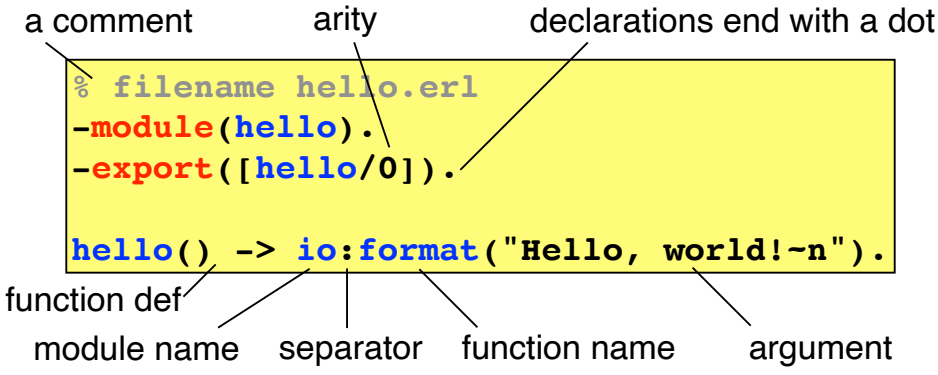
`halt().` to exit the emulator

Erlang modules

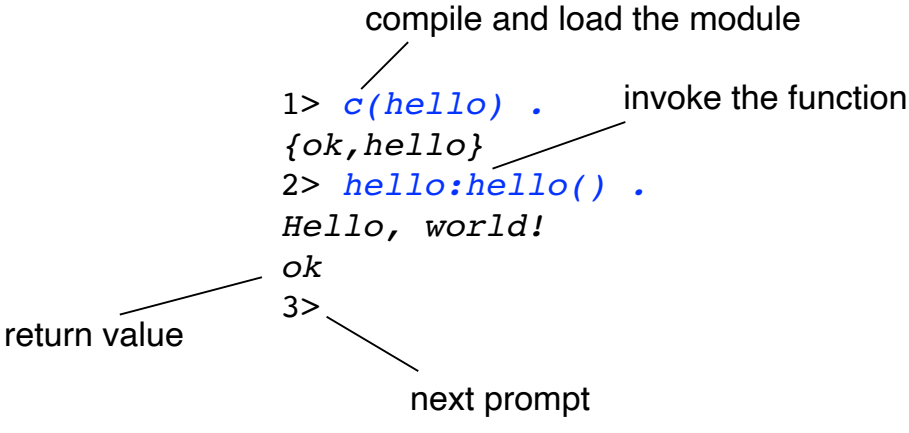
functions are organised in modules

one module for source file

filename is module name with suffix `.erl`



erl: module loading



if you edit `hello.erl` and do `c(hello)` again
the new version of the module replaces the old one

Erlang basics

Function definition

separates function clauses with ;
last clause ends with .

variables start with upper-case letters X Head Tail
variables are local to function clauses

function definitions cannot be nested
non-exported functions are local to the module

pattern matching allowed

guards allowed (keyword `when`)

type-checking is done at runtime

Atoms, tuples, lists

numbers: arbitrary size integers, floating point values
(cannot start with .)

atoms: start with lower-case character
(can be single-quoted if needed, don't use camelCase)

```
true  ok  hello_world.  'this is an atom'
```

tuples: main data constructor
tagged tuples: the first element of the tuple is an atom
we can use pattern matching

```
{}  {movie,"Matrix"}   {movie,Title}
```

lists: can contain elements of any type
we can use pattern matching

```
[]  [1,2,ok]  [H|T]  [X,Y,Z]  [X,Y,Z| Tail]
```

Funs

funs: anonymous functions (lambda expressions)
can have several arguments and clauses

```
fun () -> 42 end

fun (X) -> X+1 end

fun (X,Y) -> {X, fun (Z) -> Z+Y end} end

fun (F,X) -> F(X) end
```

Type test & conversion

```
is_integer(X)
is_float(X)
is_number(X)
is_atom(X)
is_tuple(X)
is_list(X)
is_function(X)
is_pid(X)
…
```

```
atom_to_list(A)
list_to_atom(L)
tuple_to_list(T)
list_to_tuple(L)
…
```

Erlang concurrency

Processes

every Erlang code is executed by a process

processes are implemented by the VM (not by OS threads)

multitasking is preemptive (VM switching and scheduling)

processes need very little memory

switching between processes is very fast

the VM can handle a large number of processes

on multiprocessor/multicore machines, processes can be scheduled to run in parallel on separate CPUs/cores using multiple schedulers

different processes may be reading the same program code at the same time (no variable updates!)

Pids

each process has a process identifier

```
Pid = self()
```

new Erlang processes can be spawned to run functions

```
Pid = spawn(module,function,arguments)
Pid = spawn(fun () -> … end)
Pid = spawn(fun f/0)
Pid = spawn(fun m:f/0)
```

the spawn operation returns immediately
(the return value is the pid of the process)

children pids are available to parent process,
not vice versa (unless passed)

Communication

Messages can be sent to pids

```
Pid ! message
```

called bang

Processes can wait to receive (and select) some message

```
receive
   Pattern1 when Cond1 -> Exp1;
   Pattern2 when Cond2 -> Exp2;
   ...
   Patternk when Condk -> Expk
end
```

Communication

Messages can be sent to pids

```
Pid ! {1,2,3}
```

called bang

Processes can wait to receive (and select) some message

```
receive
    {X} when X>0 -> X;
    {X,Y} when Y>X -> X+Y;
    {X,Y,Z} when Y>X andalso Z>Y -> X+Z;
end
```

First matching clause for first message,
if none, first matching clause for second message,
if none, ...
if none it blocks (all messages are kept)

Communication

Messages can be sent to pids

```
Pid ! {1,2,3}
```

called bang

Processes can wait to receive (and select) some message

```
receive
    {X} when X>0 -> X;
    {X,Y} when Y>X -> X+Y;
    {X,Y,Z} when Y>X andalso Z>Y -> X+Z;
    _ -> 0
end
```

First matching clause for first message
(the last clause, called catch-all, will match anyway)

Communication

Messages can be sent to pids

```
Pid ! {1,2,3}
```

called bang

Processes can wait to receive (and select) some message

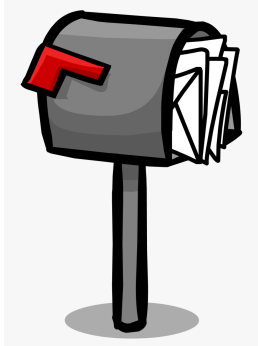
```
receive
    {X} when X>0 -> X;
    {X,Y} when Y>X -> X+Y;
    {X,Y,Z} when Y>X andalso Z>Y -> X+Z;
    after 0 -> 0
end
```

timeout

First matching clause for first message,
if none, first matching clause for second message,
if none, ...
if none it evaluates to 0 (all messages are kept)

Message passing

`receive … end`



Pid

Pid `!` *message*

Message passing

messages are sent asynchronously
(the sender continues immediately)

any value can be sent as a message

each process has a message queue (mailbox)
no size limit, messages are kept until extracted

a message is received when it is extracted from the mailbox

messages are ordered from oldest to newest in the mailbox

the message that is extracted is not necessarily the oldest
(pattern matching can be used, if there is no match
the receiver suspends and keeps waiting)

Reply

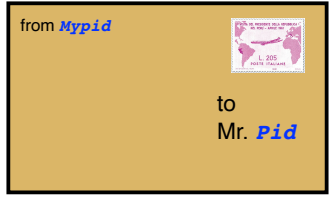
To reply a message, its sender must be known

its pid can be inserted in the message

syntax for tuples

```
Pid ! { Mypid , message }
```

now the receiver *Pid* can reply to *Mypid*



erl session

```
26  %% EXAMPLE: permutations
27
28  perms([]) -> [[]];
29  perms(L)  -> [[H|T] || H <- L, T <- perms(L--[H])].
```

```
99> c(recursion).
recursion.erl:2:2: Warning: export_all flag enabled - all functions
will be exported
{ok,recursion}
100> recursion:perms("abc").
["abc","acb","bac","bca","cab","cba"]
101> recursion:perms("abcdef").
["abcdef","abcdfe","abcedf","abcefd","abcfde","abcfed",
 "abdcef","abdcfe","abdecf","abdefc","abdfce","abdfec",
 "abecdf","abecfd","abedcf","abedfc","abefcd","abefdc",
 "abfcde","abfced","abfdce","abfdec","abfecd","abfedc",
 "acbdef","acbdfe","acbedf","acbefd",
 [...]|...]
```

erl session

```erlang
%% EXAMPLE: length of a list

len([]) -> 0;
len([_|T]) -> 1 + len(T).

tail_len(L) -> tail_len(L,0).

tail_len([],Acc) -> Acc;
tail_len([_|T],Acc) -> tail_len(T,Acc+1).
```

```erlang
%% EXAMPLE: replicate

replicate(0,_) -> [];
replicate(N,Term) when N > 0 -> [Term|replicate(N-1,Term)].

tail_replicate(N,Term) -> tail_replicate(N,Term,[]).

tail_replicate(0,_,List) -> List;
tail_replicate(N,Term,List) when N > 0 -> tail_replicate(N-1, Term, [Term|List]).
```

```erlang
%% EXAMPLE: reverse

reverse([]) -> [];
reverse([H|T]) -> reverse(T)++[H].
% costs too much!!

tail_reverse(L) -> tail_reverse(L,[]).

tail_reverse([],Acc) -> Acc;
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).
```

[Ex. 1] Write a server in erlang to convert temperatures from Celsius degrees to Fahrenheit degrees and vice versa, using the formula $F = 1.8C + 32$. The server receives requests of the form $(Pid, \texttt{cs}, C)$ or $(Pid, \texttt{ft}, F)$ and replies to *Pid* by sending messages in analogous format. The server can be stopped by sending the message `stop`. All the other messages are ignored. Spawn a copy of the server, send it some temperatures to convert, check out the results and stop the server.

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive




    end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
     {Pid,cs,C} ->




    end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
     {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                   convert();




    end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
     {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                   convert();
     {Pid,ft,F} ->



    end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
     {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                   convert();
     {Pid,ft,F} -> Pid ! {self(),cs,(F - 32) / 1.8},
                   convert();


    end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
     {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                   convert();
     {Pid,ft,F} -> Pid ! {self(),cs,(F - 32) / 1.8},
                   convert();
     stop -> true;

    end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
     {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                   convert();
     {Pid,ft,F} -> Pid ! {self(),cs,(F - 32) / 1.8},
                   convert();
     stop -> true;
     _ ->
    end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
     {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                   convert();
     {Pid,ft,F} -> Pid ! {self(),cs,(F - 32) / 1.8},
                   convert();
     stop -> true;
     _ -> convert()
    end.
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2>
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2> Conv = spawn(ex1,convert,[]).
<0.84.0>
3>
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2> Conv = spawn(ex1,convert,[]).
<0.84.0>
3> Conv ! {self(),cs,23}.
{<0.77.0>,cs,23}
4>
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2> Conv = spawn(ex1,convert,[]).
<0.84.0>
3> Conv ! {self(),cs,23}.
{<0.77.0>,cs,23}
4> receive
4>     {Conv,ft,F} -> io:format("23 celsius = ~p fahrenheit~n",[F])
4> end.
23 celsius = 73.4 fahrenheit
ok
5>
```

[**Ex. 2**] Write an erlang function `copy` that receives an integer n and if n is positive it prints n copies of n (one per line). Write an erlang function that receives a list of integers and spawn an instance of `copy` for each integer in the list.

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 ->
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) ->
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 ->
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 -> io:format("~p~n",[M]),
                        copy(N-1,M);
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 -> io:format("~p~n",[M]),
                        copy(N-1,M);
copy(_,_) -> true.
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 -> io:format("~p~n",[M]),
                        copy(N-1,M);
copy(_,_) -> true.

listCopy(L) ->
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 -> io:format("~p~n",[M]),
                        copy(N-1,M);
copy(_,_) -> true.

listCopy(L) -> [                          || N <- L ].
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 -> io:format("~p~n",[M]),
                        copy(N-1,M);
copy(_,_) -> true.

listCopy(L) -> [ spawn(ex2,copy,[N]) || N <- L ].
```

Ex. 2, copy

```
Eshell V10.2.1  (abort with ^G)
1> c(ex2).
{ok,ex2}
2>
```

Ex. 2, copy

```
Eshell V10.2.1  (abort with ^G)
1> c(ex2).
{ok,ex2}
2> ex2:listCopy(lists:seq(1,5)).
1
2
3
4
5
[<0.84.0>,<0.85.0>,<0.86.0>,<0.87.0>,<0.88.0>]
2
3
4
5
3
4
5
4
5
5
3>
```