0. Erlang

Sequential Programming

Facultat d’Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

Why Erlang?

- We need a language to implement some of the algorithms that we will learn
- Erlang is a general-purpose language and runtime environment well suited for scalable distributed programming
  - Developed at Ericsson in late eighties
  - Functional Programming (e.g. Prolog, Lisp)
  - Available as open source
  - Built-in support for concurrency, distribution and fault tolerance

Why Erlang?

- Companies use Erlang in their production systems, e.g. Amazon, Yahoo!, Facebook, WhatsApp, T-Mobile, Motorola, Ericsson, ...
  - https://en.wikipedia.org/w/index.php?title=Erlang_(programming_language)&oldid=754567048#Companies_using_Erlang
- Popular applications use Erlang, e.g. Chef, Ejabberd, CouchDB, GitHub, RabbitMQ, ...
  - https://en.wikipedia.org/w/index.php?title=Erlang_(programming_language)&oldid=754567048#Software_projects_written_in_Erlang

Data Structures

- Literals
  - atoms: foo, my_bar, …
  - numbers
    - integers: 10, -234, 16#A42B
    - floats: 17.3, -56.62
  - nil: []
  - bool: true, false

Data Structures

- Compound
  - tuples: used to store a **fixed** number of items
    - {123, def, abc}
    - {person, 'Joe', 'Armstrong'}
    - {abc, {def, 123}, jkl}
  - lists: used to store a **variable** number of items
    - []: empty list
    - [foo, 12, bar, zot]
  - "..." is short for the list of integers representing the ASCII codes of the enclosed within the quotes
    - "abcdefghi" is [97,98,99,100,101,102,103,104,105]

Variables

- Used to store values of data structures
- Procedure definition
  - No global scope
- Variables can only be bound **once**
  - Assigned a value when introduced
  - The value of a variable can never be changed
- Start with an upper case letter
  - Abc, A_var, Foo

Variables

- Assignment and Pattern matching
  - A = 10; Succeeds: binds A to 10
  - B = {z, Foo, 4}; Succeeds: binds B to {z, Foo, 4}
  - {B, C, D} = {10, foo, bar}
    - Succeeds: binds B to 10, C to foo and D to bar
  - {A, A, B} = {abc, abc, foo}
    - Succeeds: binds A to abc, B to foo
  - {A, A, B} = {abc, def, 123}; Fails
  - [A,B,C] = [1,2,3]
    - Succeeds: binds A to 1, B to 2, C to 3
  - [A,B,C,D] = [1,2,3]; Fails

Pattern matching

- Cons cell: [H | T]
- Used for pattern matching on lists
  - The pattern "[H|T] = List" extracts the head into "H" and tail into "T" of the list "List"
  - [A,B|C] = [1,2,3,4,5,6,7]
    - Succeeds: binds A = 1, B = 2, C = [3,4,5,6,7]
  - [H|T] = [1,2,3,4]
    - Succeeds: binds H = 1, T = [2,3,4]
  - [H|T] = [abc]
    - Succeeds: binds H = abc, T = []

Pattern matching

- [H|T] = []
  - Fails
- {A,_,[B|_],{B}} = {abc,23,[22,x],{22}}
  - Succeeds: binds A = abc, B = 22
  - Note the use of "_", the anonymous (don't care) variable
    - It is used as a place holder where the syntax requires a variable, but the value of the variable is of no interest
  - If you know that X is bound to a tuple with three elements and you need to access the second, do:
    - {_,Y,_} = X
    - Y is now a reference to the second element

Function calls

```
module:func(Arg1, Arg2, ... Argn)
```

- Arg1 .. Argn are any Erlang data structures
- The module/function name must be an atom

```
math2:double(10).
```

- Function calls can be nested

```
math2:double(math2:double(2)).
```

- A function can have zero arguments

```
hello:world().
```

Function calls

- Built In Functions (BIFs)
  - date()
  - time()
  - length([1,2,3,4,5])
  - size({a,b,c})
  - self()
  - ...
- Full list at http://erlang.org/doc/index.html

Function definition

```
func(Pattern1, Pattern2, ...) -> ... ;
func(Pattern1, Pattern2, ...) -> ... ;
...
func(Pattern1, Pattern2, ...) -> ... .
```

- Clauses are scanned sequentially until a match is found
- When a match is found all variables occurring in the head become bound
- Variables are local to each clause

Function & module definition

```
-module(demo).
-export([double/1]).
double(X) -> times(X, 2).
times(X, N) -> X * N.
```

- Functions are defined within Modules and must be exported before they can be called from outside the module
  - double can be called from outside the module, times is local to the module
  - double/1 means the function double with one argument

Function examples

```
-module(mathStuff).
-export([factorial/1, area/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

area({square, Side}) -> Side * Side;
area({circle, Radius}) -> 3.14 * Radius * Radius;
area({triangle, A, B, C}) -> S = (A + B + C)/2,
   math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Other) -> {invalid_object, Other}.
```

Function examples: Conditionals

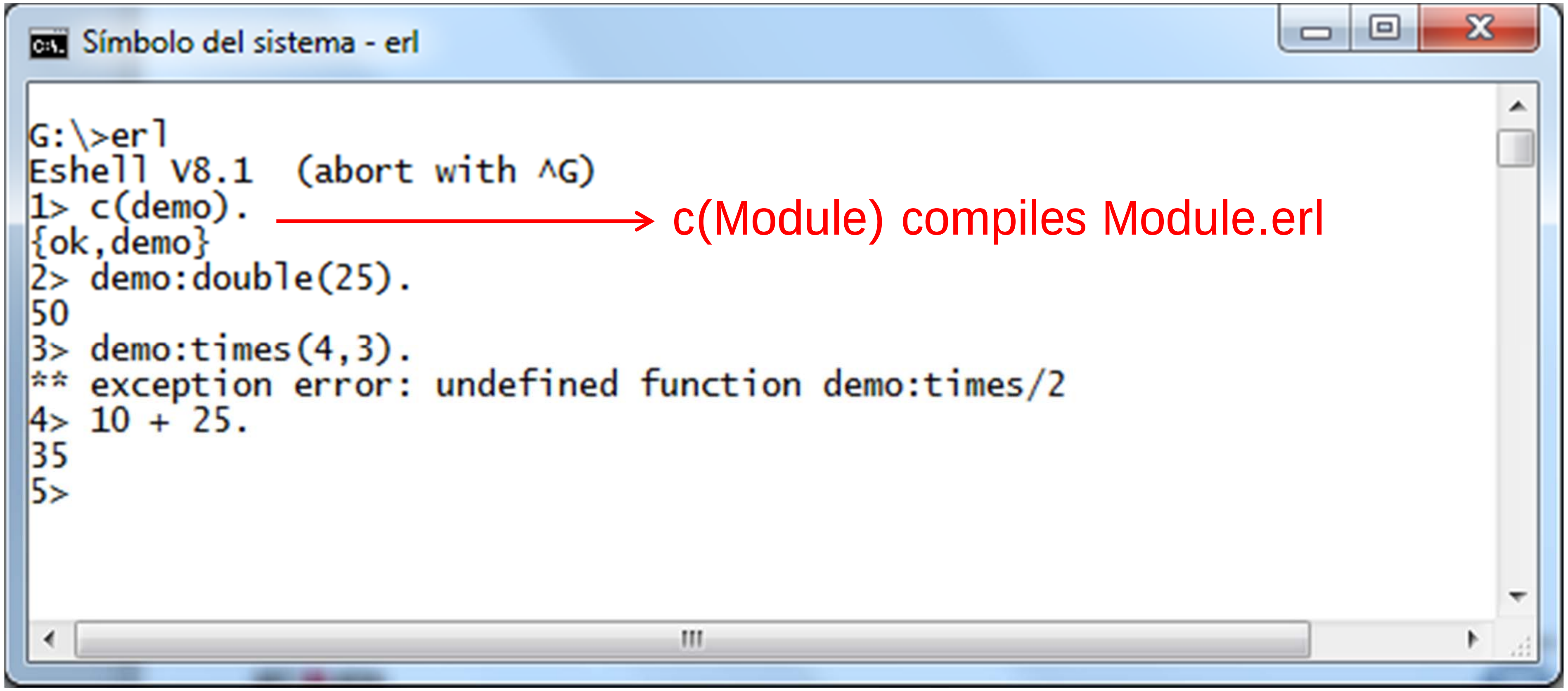
```
fac(N) ->
   if
      N == 0 -> 1;
      N > 0 -> N*fac(N-1)
   end.

sum([]) ->
   0;
sum([H|T]) ->
   H + sum(T).
```

```
sum(L) ->
   case L of
      [] ->
         0;
      [H|T] ->
         H + sum(T)
   end.
```

Runtime system

- The Erlang runtime system gives you an interactive shell
- In the shell you can compile and load modules and call functions
- Run it by itself, inside vim, emacs, or in a IDE such as Eclipse

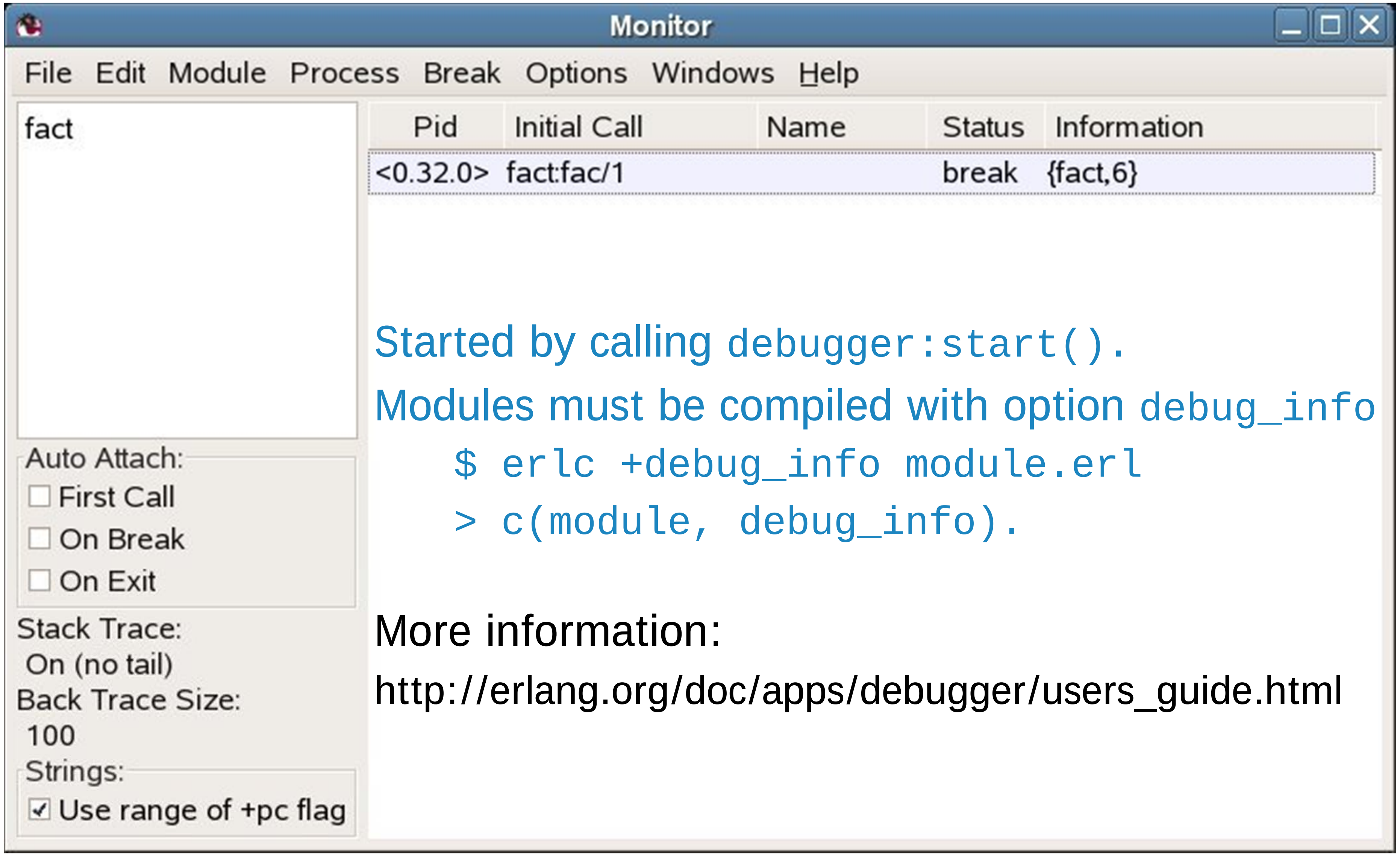
Runtime system



```
G:\>erl
Eshell V8.1  (abort with ^G)
1> c(demo).
{ok,demo}
2> demo:double(25).
50
3> demo:times(4,3).
** exception error: undefined function demo:times/2
4> 10 + 25.
35
5>
```

c(Module) compiles Module.erl

Debugger



fact

Pid	Initial Call	Name	Status	Information
<0.32.0>	fact:fac/1		break	{fact,6}

Auto Attach:
☐ First Call
☐ On Break
☐ On Exit
Stack Trace:
On (no tail)
Back Trace Size:
100
Strings:
☑ Use range of +pc flag

Started by calling `debugger:start().`

Modules must be compiled with option `debug_info`

```
    $ erlc +debug_info module.erl
    > c(module, debug_info).
```

More information:

http://erlang.org/doc/apps/debugger/users_guide.html

0. Erlang

Concurrent Programming

Facultat d’Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

Process creation

- Pid1 invokes:

```
Pid2 = spawn(Module, Function, [Args])
```

- Creates and starts the execution of a new process that runs Function
- Pid2 is process identifier of the new process
  - This is known only to process Pid1

Message passing

- To send a message you need the *process identifier* of the receiver

```
Pid ! Msg
```

  - Sending a message is **asynchronous**
    - No acknowledgement
  - self() - returns the Process Identity (Pid) of the process executing this function
  - Process identifiers can be included in messages just like any data structure

```
P = spawn(wait, hello, []).
P ! "hello".
```

Message passing

- A process can <u>suspend</u> waiting for a message

```
receive
    Msg -> Actions
end.

-module(wait).
-export([hello/0]).
hello() ->
    receive
       X -> io:format("message received: ~s~n", [X])
    end.
```

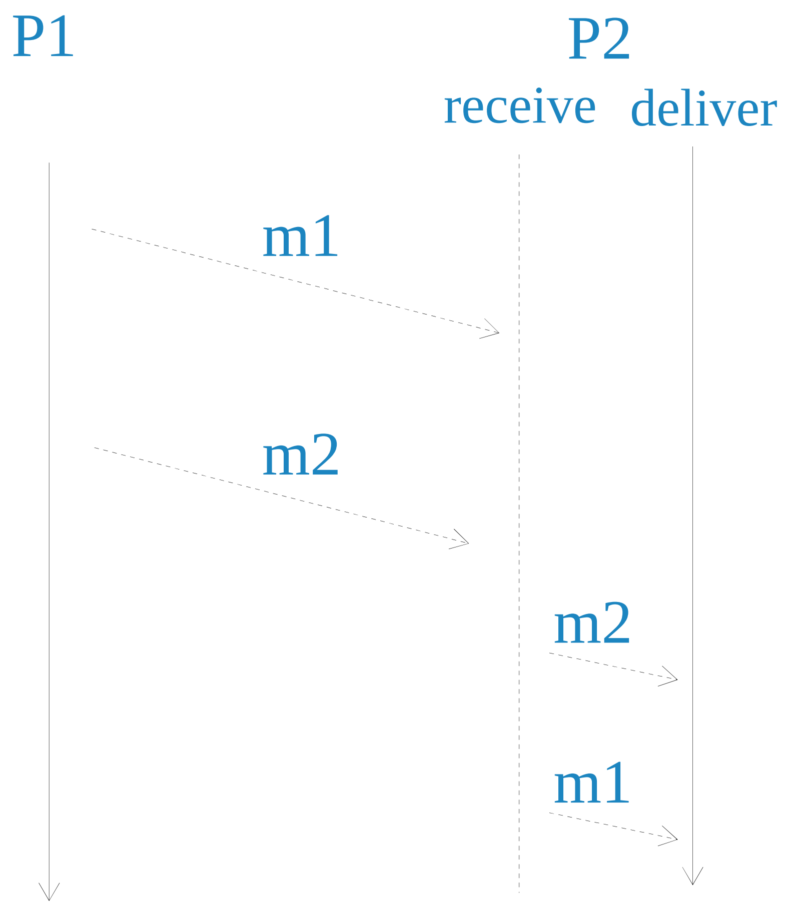
Message passing

- A process will have an ordered sequence of received messages
  - All messages sent to a process are stored in its mailbox in the same order as they arrive

```
receive
    Message1 -> Actions1 ;
    Message2 -> Actions2
end.
```

- The first message that matches one of defined patterns will be delivered

Message passing



messages received
in FIFO order

asynchronous sending,
no acknowledgment

implicit deferral
of message
delivery

```
receive
  m2 -> true
end,
receive
  m1 -> true
end
```

Message passing

- ↑ One can select which messages to handle first
- ↓ Risk of forgetting messages that are left in a growing queue
- Receiving messages from a specific process

```
Pid ! {self(),abc}.

receive
   {Pid,Msg} ->...
end.
```

Registered processes

- Register the process identifiers under names that are known to all processes

```
P = spawn(wait, hello, []).
register(foo, P).
foo ! "hello".
```

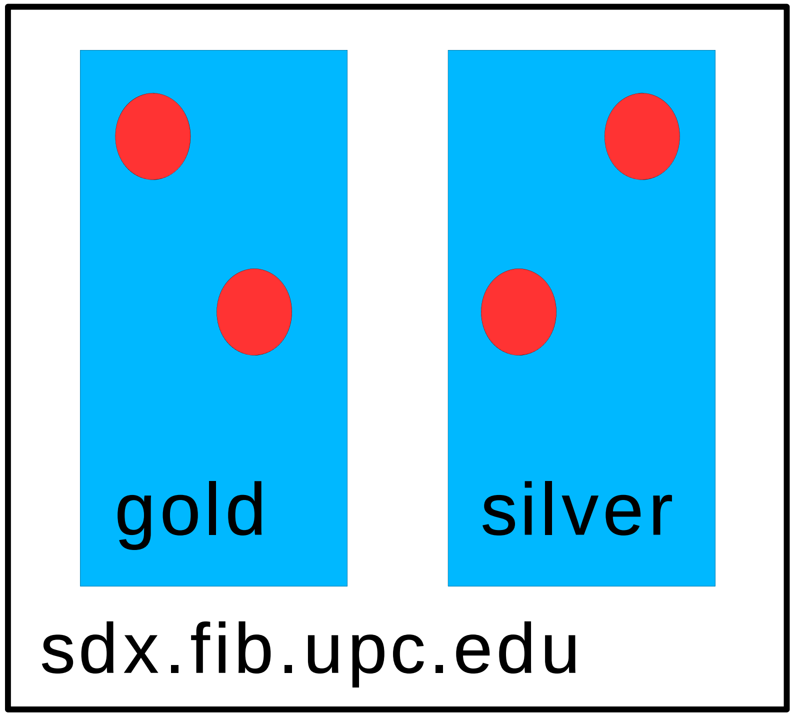
  - Sending to a registered name is different to sending to a process id
    - Sending to a process id will always succeed
      - Even if the process is dead
    - A dead process will be de-registered and sending to a name without registered process will cause an exception

0. Erlang

Distributed Programming

Facultat d’Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

Distributed programming



- Each Erlang instance is a different node in the distributed system

  Node 1: 'gold@sdx.fib.upc.edu'

  Node 2: 'silver@sdx.fib.upc.edu'

Distributed programming

- Create a process in a remote Erlang node

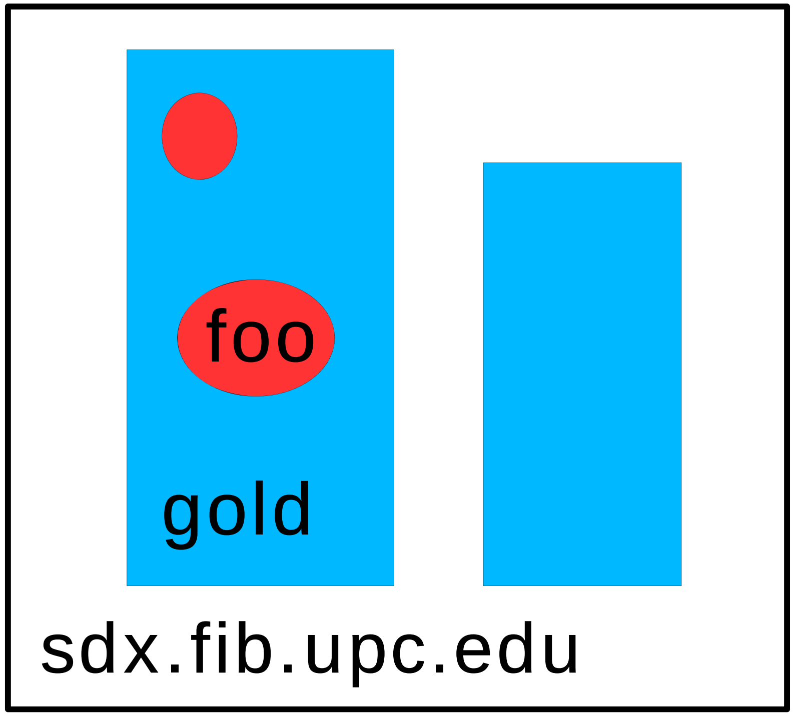
```
P = spawn('gold@sdx.fib.upc.edu', M, F, [A]).
P ! Msg.
```

- Connections are established automatically when another node is referenced
- Process IDs returned by spawn (or received in a message) can be used normally
  - Access transparent: local or remote are the same
  - Location transparent: you do not know where the process is located

Distributed programming

- Send a message to a process that is locally registered on a remote node

```
{Name, Node} ! Message.
```



```
{foo, 'gold@sdx.fib.upc.edu'} ! "hello".
```

Distributed programming

- We will use locally registered names, but Erlang offers also globally registered names

```
global:register_name(foo, Pid).
```

- Send a message to the process globally registered as foo

```
global:send(foo, "Hello").
```

- Connections must be established explicitly

```
net_kernel:connect_node(Node).
```

Distributed programming

- When you start Erlang you should make it network aware by providing a name
  - Using long (-name) or short names (-sname) (*)

    erl -name gold@sdx.fib.upc.edu / erl -name gold

      - node(): gold@sdx.fib.upc.edu

    erl -sname gold@sdx / erl -sname gold

      - node(): gold@sdx

    erl -name gold@127.0.0.1

(*) A node with a long name cannot communicate with a node with a short name

Distributed programming

- If someone connects to a node, it gets connected to all the other nodes
- Use cookies as a mechanism to differentiate clusters of nodes $\Rightarrow$ Nodes with different cookies are not able to communicate together

```
-setcookie mycookie
erlang:set_cookie(node(), mycookie)
```

  - Alternatively, you can have a file *.erlang.cookie* with the cookie in your home folder on all nodes

More information

- Erlang official website: http://www.erlang.org/
- ‘An Erlang Primer’ by Johan Montelius
  - http://people.kth.se/~johanmon/dse/crash.pdf
- ‘Learn You Some Erlang for Great Good!’
  - http://learnyousomeerlang.com/
- ‘Concurrent Programming in Erlang, Part I’
  - http://erlang.org/download/erlang-book-part1.pdf
- Elixir language: http://elixir-lang.org/
  - Runs on the Erlang runtime