

---

## DISTRIBUTED SYSTEMS (COMP9243)

### Lecture 1.5: Erlang

#### Slide 1

- ① Introduction
- ② Sequential programming
- ③ Concurrent programming
- ④ Resources

---

## INTRODUCTION TO ERLANG

**Erlang:** Functional language with built in concurrency support

**OTP:** A large collection of libraries for Erlang

#### Features:

- Concurrency and asynchronous message passing
- Lightweight processes. Fast context switches
- Virtual machine
- ✗ Not suitable for low-level system software

#### History:

- Named after mathematician Agner Erlang
- Originated from Ericsson (maybe Erlang actually stands for ERicsson LANGUage?)
- Used for a lot of telecoms applications: e.g. switches
- Open sourced in 1998

---

## THE ERLANG ENVIRONMENT

```
unix% erl
1> 1 + 2.
3
2> c(demo).
{ok,demo}
3> demo:double(25).
50
4> date().
{2004,2,24}
5> halt().
unix% cat demo.erl
-module(demo).
-export([double/1]).
double(X) -> 2 * X.
unix%
```

#### Slide 3

---

## SEQUENTIAL PROGRAMMING

#### Basics:

- Numbers: Integers (1, -10), Floats (3.1415, -0.23)
  - Hex: 16#AB123 Binary: 2#100110
  - ASCII: \$A (65), \$z (122), etc.
- Atoms: hello, how\_are\_you, 'I am fine'
- Variable: Counter, Good\_server, BadServer
  - Only bound once. Value cannot be changed once bound!!!
- Operators: +, -, \*, /, >, >=, <, <=, ==, /=

#### Slide 4

---

**Slide 5**

Data Structures:

- Tuples: {123, hello, 'Good Morning', {super, 456}}, {}
- Lists: [123, hello, 'Welcome'], [], "abcdefg", ""
- Combinations: [{123, house}, guest, {friends, family}], {123, [1,2,3,4], "building"}
- Others (dict, process dictionary, etc.): see documentation

---

**Slide 6**

Pattern Matching:

Binding variables to values

- ✓ A = 10
- ✓ {B, C, D} = {10, foo, bar}
- ✓ {A, A, B} = {abc, abc, foo}
- ✗ {A, A, B} = {abc, def, 123}
- ✓ [A,B,C] = [1,2,3]
- ✗ [A,B,C,D] = [1,2,3]
- ✓ [A,B|C] = [1,2,3,4,5,6,7]
- ✓ [A|B] = [abc]
- ✗ [A|B] = []
- ✓ {A,\_, B} = {123, 456, 789}

---

Output:

```
io:format(FormatString, ArgList)
```

Examples

**Slide 7**

```
1> io:format("Hello world!~n", []).
Hello world!
ok
2> io:format("arg1:~w, arg2:~w, arg3:~w", [1,2,5]).
arg1:1, arg2:2, arg3:5ok
3>
```

---

Functions:

Function definition (in a module)

```
-module(math).
-export([factorial/1]).
```

**Slide 8**

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

Function use

```
2> math:factorial(5).
120
```

---

### Anonymous Functions:

#### Slide 9

```
F = fun(X) -> X*2 end.  
F(2).
```

---

### Function Evaluation Rules:

- Clauses scanned until a match is found
- All variables in function head are bound
- Variables are local to each clause
- Body evaluated sequentially

#### Slide 10

### Built In Functions:

- In module `erlang`.
  - Do what you cannot (easily) do in Erlang
  - See documentation (<http://www.erlang.org/documentation/doc-5.6.3/lib/kernel-2.12.3/doc/html/erlang.html>)
- 

---

### Guarded Function Clauses:

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

### Examples

#### Slide 11

- `is_number(X)` - X is a number
  - `is_atom(X)` - X is an atom
  - `is_tuple(X)` - X is a tuple
  - `is_list(X)` - X is a list
  - See documentation for more (<http://www.erlang.org/documentation/doc-5.6.3/doc/index.html>)
- 

### Case and If:

```
case X of  
    true -> ...;  
    false -> ...;  
end,  
...
```

#### Slide 12

```
if  
    integer(X) -> ...;  
    tuple(X) -> ...;  
end,  
...
```

---

---

## Recursion and List Traversal:

### Common patterns

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

**Slide 13** `double_list([H|T]) -> [2*H|double_list(T)];`  
`double_list([]) -> [].`

```
member(H, [H|_]) -> true;  
member(H, [_|T]) -> member(H, T);  
member(_, []) -> false.
```

---

```
double_list([H|T]) -> [2*H|double_list(T)];  
double_list([]) -> [].
```

### What happens:

```
double_list([1,2,3]).
```

**Slide 14**

```
double_list([1,2,3]) => [2|double_list([2,3])]  
double_list([2,3]) => [4|double_list([3])]  
double_list([3]) => [6|double_list([])]
```

```
[2,4,6]
```

---

---

## List Comprehensions:

```
List = [ X || X <- L, Filter ]
```

**Slide 15**

### Example:

```
Y = [ 1/X || X <- List, X > 0].
```

---

## SOME USEFUL LIBRARIES

### stdlib:

```
http://www.erlang.org/documentation/doc-5.6.3/lib/  
stdlib-1.15.3/doc/html/index.html
```

**Slide 16**

- `io`: read, write, format, etc.
  - `lists`: append, concat, flatten, reverse, sort, member, etc.
  - `string`: len, equal, concat, substr, strip, etc.
  - `dict`: new, find, store, fetch, update, etc.
  - `math`: sin, cos, tan, exp, log, pow, sqrt, etc.
-

---

## CONCURRENT PROGRAMMING

Processes:

```
Pid = spawn(Mod, Func, Args)
```

### Slide 17

Creates a new process that evaluates the given function with the given arguments

```
F = fun() -> io:format("Hello!") end.  
Pid = spawn(F).
```

---

Message Passing:

A does:

```
B ! {self(), hello, you}
```

### Slide 18

This sends a message {A, hello, you} to process B  
In order to receive the message B does:

```
receive  
  {From, Msg1, Msg2} -> ...  
end
```

---

---

Selective Message Reception:

```
A: C!foo
```

```
B: C!bar
```

```
C:
```

```
receive  
  foo -> true  
end,  
receive  
  bar -> true  
end
```

### Slide 19

→ foo is received before bar no matter what order they were sent in.

---

Timeouts:

Wait a given amount of time (milliseconds)

```
sleep(T) ->  
  receive  
  after  
    T -> true  
  end.
```

### Slide 20

Wait forever

```
suspend() ->  
  receive  
  after  
    infinity -> true  
  end.
```

---

Slide 21

```
0 is special

flush() ->
  receive
    Any -> flush()
  after
    0 -> true
  end.
```

0 means:

- Check message buffer
- If empty execute the given code (true)

## CLOSURES

Values of bound variables are passed along in messages

Slide 22

```
-module(closures).
-export([do_send/4, do_receive/0]).
do_send(Dest, A, B, C) ->
  Dest ! {msg, fun(D) ->
    io:format("A: ~s, B: ~s, C: ~s, D: ~s~n", [A, B, C, D]) end}.
do_receive() ->
  receive
    {msg, F} -> F("woohoo")
  end.

unix% erl
1> B = spawn(fun() -> closures:do_receive() end).
2> closures:do_send(B, "hello", "there", "friend")
A: hello, B: there, C: friend, D: woohoo
```

## ERROR HANDLING

Try - Catch:

Slide 23

```
catch_error(N) ->
  try error_func(N) of
    {ok, Ret} -> io:format("SUCCES: ~w~n", [Ret])
  catch
    throw:Err -> io:format("THROW: ~w~n", [Err]);
    exit:Err -> io:format("EXIT: ~w~n", [Err]);
    error:Err -> io:format("ERROR: ~w~n", [Err])
  after
    io:format("All Done~n")
  end.
```

```
error_func(1) -> throw(woops);
error_func(2) -> exit(woops);
error_func(3) -> erlang:error(woops);
error_func(N) -> {ok, N}.
```

Trap Exit:

Slide 24

```
trapper(N) ->
  process_flag(trap_exit, true),
  Pid = spawn(fun() -> exiter(N) end),
  link(Pid),
  receive
    {'EXIT', Pid, Why} -> io:format("~w exited with ~w~n", [Pid, Why])
  end.

exiter(1) -> exit(1);
exiter(2) -> 1/0;
exiter(N) -> true.
```

---

## DYNAMIC CODE LOADING

```
-module(dyn).  
-export([start/0]).  
start() -> spawn(fun() -> dyn_loop() end).  
dyn_loop() -> io:format("a = ~w~n",[dyn_a:a()]), sleep(), dyn_loop().  
sleep() -> receive after 3000 -> true end.
```

Slide 25

```
-module(dyn_a).  
-export([a/0]).  
a() -> 1.  
  
3> dyn:start().  
a = 1  
a = 1  
% change dyn_a.erl to return 2  
4> c(dyn_a).  
{ok,dyn_a}  
a = 2
```

---

Slide 26

## SOME EXAMPLES?

---

---

## ERLANG RESOURCES

<http://www.erlang.org>

**Documentation** <http://www.erlang.org/doc.html>

### Introductory Course

<http://www.erlang.org/course/course.html>

Slide 27

**Man pages** [http://www.erlang.org/documentation/doc-5.6.3/doc/man\\_index.html](http://www.erlang.org/documentation/doc-5.6.3/doc/man_index.html)

### Erlang Book

<http://www.erlang.org/download/erlang-book-part1.pdf>

### Programming Rules and Conventions

[http://www.erlang.se/doc/programming\\_rules.shtml](http://www.erlang.se/doc/programming_rules.shtml)

**Existing Code** <http://www.erlang.org/examples.html>,

<http://www.erlang.org/user.html>

---

## WATCH THE MOVIE!



Slide 28

<http://www.youtube.com/watch?v=uKfKtXYLG78>

---