16 - Erlang
Erlang

concurrency oriented programming
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

unfamiliar 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
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

typical interaction: prompt user’s input

1> command . user’s input

value don’t forget the dot!

result

next prompt

2>

halt(). to exit the emulator
Erlang modules

functions are organised in modules

one module for source file

filename is module name with suffix .erl

a comment

arity

declarations end with a dot

module name

function name

argument

arity

% filename hello.erl
-module(hello).
-export([hello/0]).

hello() -> io:format("Hello, world!\n").
erl: module loading

compile and load the module

1> c(hello) .
{ok,hello}

invoke the function

2> hello:hello() .
Hello, world!

return value

3> ok

next prompt

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 singlequoted 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: 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

\[ Pid = \text{self()} \]

new Erlang processes can be spawned to run functions

\[ Pid = \text{spawn}(\text{module}, \text{function}, \text{arguments}) \]

\[ Pid = \text{spawn}(\text{fun} () \rightarrow \ldots \text{end}) \]

\[ Pid = \text{spawn}(\text{fun} \ f/0) \]

\[ Pid = \text{spawn}(\text{fun} \ m:f/0) \]

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

\textit{Pid} ! \textit{message}

called bang

Processes can wait to receive (and select) some message

receive ... end
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

\[ Pid \uparrow \{ Mypid, message \} \]

now the receiver \( Pid \) can reply to \( Mypid \)
erl session

```
Last login: Fri Apr 17 11:42:07 on ttys001
host-131-114-219-127:~ bruni$ erl
Erlang/OTP 21 [erts-10.2.1] [source] [64-bit] [smp:12:12] [ds:12:12:10] [async-threads:1] [hipe] [dtrace]
Eshell V10.2.1  (abort with ^G)
1>  
```