Concurrent Programming
CS511

About Erlang

- ▶ Functional language
- ▶ Concurrent/Distributed
  - ▶ No shared memory (message passing)
- ▶ No types at compile time
  - ▶ Dynamically typed
- ▶ Open source
- ▶ Developed in the 80s in Ericsson by Joe Armstrong, Robert Virding and Mike Williams

Runtime System

- Compiled code runs on a virtual machine (BEAM).
- Lightweight processes
- Fast process creation
- Support hot-swapping

More About Erlang

- Open Telecom Platform (OTP) (fault-tolerance, hot code update, ...)
  - Practically "proven" programming patterns
- Supporting tools
  - Debugger
  - Unit testing
  - Dialyzer
  - Mnesia

Typical Applications

- Telecoms
    - Switches (POTS, ATM, IP, ...)
    - GPRS
    - SMS applications
- Internet applications
    - WhatsApp (backbone)
    - Facebook (chat)
    - Amazon (SimpleDB, part of the Amazon Elastic Compute Cloud)
    - Yahoo! (social bookmarking service, Delicious)
    - Online shopping (Klarna AB)
    - T-Mobile
    - Discord
- 3D modelling (Wings3D)

Bibliography

- **Programming Erlang: Software for a Concurrent World**, Joe Armstrong

- **Learn some Erlang for Great Good!**, Fred Hebert
- **Erlang Programming**, Cesarini and Thompson

Note: Some of the material in these slides draw from the above sources

Downloading and Installing

- http://www.erlang.org/downloads
- Prepackaged binaries from https://www.erlang-solutions.com/resources/download.html
- Setting up erlang mode in emacs+flycheck: http://www.lambdacat.com/post-modern-emacs-setup-for-erlang/

Running Erlang

```
 1 $ erl
 2 Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:4:4]
       [async-threads:10] [hipe] [kernel-poll:false]
 3
 4 Eshell V8.0  (abort with ^G)
 5 1> io:format("Hello, world!~n").
 6 Hello, world!
 7 ok
 8 2> q().
 9 ok
10 $
```

A Small Script

```
1 $ cat hello.erl
2 %% Sample module
3 -module(hello).
4 -export([hello/0]).
5 hello() -> "Hello, world!".
6 $ erl
7 Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:4:4]
      [async-threads:10] [hipe] [kernel-poll:false]
8
9 Eshell V8.0  (abort with ^G)
10 1> c(hello).
11 {ok,hello}
12 2> hello:hello().
13 "Hello, world!"
14 3> q().
```

Mind the dot!

Sequential Fragment

- Data types and variables
- Function definitions
- Pattern matching

Numbers

Integers (arbitrarily big)

```
1 7> fact:fact(200).
2 78865786736479050355236321393218506229513597768717326329474 2
3 53324435944996340334292030428401198462390417721213891963883 0
4 25764279024263710506192662495282993111346285727076331723739 6
5 98894392244562145166424025403329186413122742829485327752424 2
6 40757390324032125740557956866022603190417032406235170085879 6
7 17892222278962370389737472000000000000000000000000000000000 0
8 000000000000000
```

Floats

```
1 > 2.78 + 9.6.
2 12.379999999999999
```

▶ IEEE 754 de 64–bits (range: $\pm 10^{308}$)

Atoms

```
1 start_with_a_lower_case_letter
2 'Anything_inside_quotes\n\09'
```

- ▶ Names must begin in lowercase or between apostrophes
- ▶ Heavily used

Characters and Strings

▶ Characters: `$a`, `$n`.

▶ Strings

```
1 9> "hello".
2 "hello"
```

▶ They are, in fact, a list of integers.

```
1 10> "hello\7".
2 [104,101,108,108,111,7]
3 11> is_list("hello").
4 true
```

Tuples

```
1 {}
2 {atom, another_atom, 'PPxT'}
3 {atom, {tup, 2}, {{tup}, element}}
4 {atom, {"hello",5}}
5
6 12> is_tuple({atom, another_atom, 'PPxT'}).
7 true
8 13> is_list({atom, another_atom, 'PPxT'}).
9 false
10 14> is_atom({atom, another_atom, 'PPxT'}).
11 false
```

Modeling Data

```
type 'a tree = Leaf of 'a | Node of ('a tree)*('a tree)
```

▶ Atoms to indicate which constructor is used at top-level
▶ Tuples to collect the arguments of the constructor
▶ Example:

```
Node(Node(Leaf 3,Leaf 4), Leaf 5)
```

becomes

```
{node, {node, {leaf, 3}, {leaf, 4}}, {leaf, 5}}
```

Operators

- Arithmetic: $+$, $-$, $*$, $/$, div, rem
- Equal value: "==" and "/="
- Exact equality (type and value):
  "=:=" and "=/="
- Boolean: and, or, xor, not, andalso, orelse

Operator Examples

```
1 1> 4 == 4.0.          % value is 4 on both sides
2 true
3 2> 4 =:= 4.0.         % value same but type different
4 false
5 3> 1 =/= 0.
6 true
7 4> 1 /= 1.
8 false
9 5> 1 =/= 1.0.
10 true.
```

- ▶ Use == and /=
- ▶ Switch to =:= and =/= only when you need exact equality (type and value)

Lists

```
1 []
2 [1, true]
3 [1 | [true] ]
4 [ok, 10]
```

- ▶ List concatenation: "++"
- ▶ List subtraction: "--"
- ▶ List cons: "|"
- ▶ List comprehension: `[ math:log(A)|| A <- lists:seq(1,10)]`

Operator and List Examples

```
1 3> L1 = [ apple , cherry ].
2 [apple ,cherry]
3 4> L2 = [ lime , grape ].
4 [lime ,grape]
5 5> L3 = L1 ++ L2.
6 [apple ,cherry ,lime ,grape]
7 6> L3 -- [cherry].
8 [apple ,lime ,grape]
9 7> L4 = [ banana | L3 ].
10 [banana ,apple ,cherry ,lime ,grape]
11 8> [ Head | Tail ] = L4.
12 [banana ,apple ,cherry ,lime ,grape]
```

- ▶ Note: = is not assignment; it is matching

Operator and List Examples (cont.)

```
 1 9> b().                 % b() shows all bindings
 2 Head = banana           % Head and Tail have been bound
 3 L3 = [apple,cherry,lime,grape]
 4 L4 = [banana,apple,cherry,lime,grape]
 5 Tail = [apple,cherry,lime,grape]
 6 ok
 7 10> f().                 % f() flushes all bindings
 8 ok
 9 11> { A, B } = { 4.0, 5.2 }.
10 {4.0,5.2}
11 12> b().
12 A = 4.0
13 B = 5.2
14 13> { C, D } = { 4.0, 5.2 }.
15 {4.0,5.2}
16 14> { A, B } == { C, D }.
17 true
18 15> { A, B } =:= { C, D }.
19 true
```

Comparison

- In Erlang all terms are comparable

  number < *atom* < *reference* < *fun* < *port* < *pid* < *tuple* < *map* < *nil* < *list* < *bitstring*

- Integers and floats are compared as usual
- The rest are compared as indicated above

```
1 1> a<2.
2 false
3 2> 2<a.
4 true
```

More on Variables

- Identifiers: `A_long_variable_name`
- Must start with an upper case letter
- Can store values
- Can be bound only once!
- Bound variables cannot change values
- We use the $=$ operator for binding (and also matching!)

More on Variables

```erlang
1 1> a = 3.                % fails because a is not a variable
2 ** exception error: no match of right hand side value 3
3 2> A = 3.                % notice: ends with a period
4 3
5 3> B = 3.                % there's that period again
6 3
7 4> A = B.                % succeeds: A and B both have value 3
8 3
9 5> A = 4.                % fails because A cannot be re-bound
10 ** exception error: no match of right hand side value 4
11 6> X = { hello, goodbye }.     % hello & goodbye are atoms
12 {hello,goodbye}
13 7> { Y, Z } = X.         % binds both Y and Z
14 {hello,goodbye}
15 8> Y.
16 hello
17 9> Z.
18 goodbye
```

More on Variables

```
1 10> X = {Y,Z}.         % succeeds because of value match
2 {hello,goodbye}
3 11> X = {hello,Z}.     % succeeds because of value match
4 {hello,goodbye}
5 12> q()                % notice: forgot the period
6 12> q().               % fails because Erlang reads "q()q()."
7 * 2: syntax error before: q
8 13> q().               % succeeds: quits Erlang shell
9 ok
```

Functions

- ▶ May have several clauses
    - ▶ Function is sequence of pattern matching clauses separated by semicolons – semicolon means "or"
    - ▶ Finish definition with .
- ▶ Function application matches arguments to pattern in some clause

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

- ▶ "when ..." is a clause guard

Example 1

```erlang
1  arith(X, Y) ->
2      io:format("Arguments: ~p ~p~n", [ X, Y ]) ,
3      Sum = X + Y ,
4      Diff = X - Y ,
5      Prod = X * Y ,
6      Quo = X div Y ,
7      io:fwrite("~p ~p ~p ~p~n", [ Sum, Diff, Prod, Quo ]) ,
8      { Sum, Diff, Prod, Quo } .
```

Take note:

- ▶ Function name starts with lowercase letter
- ▶ `io:format` is similar to `printf`
- ▶ Expressions separated by comma
- ▶ Function clause ended by period
- ▶ Final expression is function's return value

Example 2

```erlang
1 what_day(saturday) -> % "saturday" is an atom
2     weekend ;         % notice semicolon
3 what_day(sunday) ->   % "sunday" is an atom
4     weekend ;         % semicolon again
5 what_day(_) ->        % underscore is "don't care" variable
6     weekday .         % period ends function
```

Example 3

```erlang
drivers_license(Age) when Age < 16 ->
    forbidden ;
drivers_license(Age) when Age == 16 ->
    'learners permit' ;
drivers_license(Age) when Age == 17 ->
    'probationary license' ;
drivers_license(Age) when Age >= 65 ->
    'vision test recommended but not required' ;
drivers_license(_) ->
    'full license'.
```

Function Application

- ▶ Function application is call-by-value or eager
- ▶ Clause matches if function name, arguments, and all guards match the input
- ▶ Except for "built-in functions (BIFs)" must specify function's module when calling

```
1 2> drivers_license(16).
2 ** exception error: undefined shell command
      drivers_license/1
3 3> example:drivers_license(16).
4 'learners permit'
```

Function Application

```
1 1> c(example).                    % c() compiles
2 {ok,example}
3 2> drivers_license(16).           % must specify module
4 ** exception error: undefined shell command drivers_license
    /1
5 3> example:drivers_license(16).
6 'learners permit'
7 4> example:drivers_license(15).
8 forbidden
9 5> example:drivers_license(17).
10 'probationary license'
11 6> example:drivers_license(23).
12 'full license'
13 7> example:drivers_license(65).
14 'vision test recommended but not required'
15 8> q().
16 ok
```

Pattern Matching

The factorial definition uses pattern matching over numbers

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

▶ A zero number (first clause)

▶ A number different from zero (second clause)

A more involved example. Function area to compute the area of different geometrical figures.

```
1 area({square, Side})  -> Side * Side ;
2 area({circle, Radius}) -> Radius*Radius*math:pi().
```

▶ Patterns: {square, Side} and {circle, Radius}

▶ {square, Side} matches {square, 4} and binds 4 to variable Side

▶ {circle, Radius} matches {circle, 1} and binds 1 to variable Radius

Anonymous Functions

```
 1 11> F = fun (Y) -> Y+1 end.
 2 #Fun<erl_eval.6.127694169>
 3 12> F(2).
 4 3
 5 13> G = fun () -> 1 end.
 6 #Fun<erl_eval.20.127694169>
 7 14> G().
 8 1
 9 15> G.
10 #Fun<erl_eval.20.127694169>
11 16> [F,G].
12 [#Fun<erl_eval.6.127694169>,#Fun<erl_eval.20.127694169>]
```

Pattern Matching (cont.)

```
1 {B , C , D} = {10 , foo , bar}
```

Succeeds: binds B to 10, C to foo and D to bar

```
1 {A , A , B} = {abc , abc , foo}
```

Succeeds: binds A to abc, B to foo

```
1 {A , A , B} = {abc , def , 123}
```

Fails

```
1 [A ,B ,C ,D] = [1 ,2 ,3]
```

Fails

Pattern Matching (cont.)

```
1 [H|T]= [1,2,3,4]
```

Succeeds: binds H to 1, T to [2,3,4]

```
1 [H|T] = [abc]
```

Succeeds: binds H to abc, T to []

```
1 [H|T] = []
```

Fails

```
1 {A , _ , [ B | _ ] , {B} } = {abc ,23,[22,x],{22}}
```

Succeeds: binds A to abc, B to 22

BIFs

- Much-used modules in Erlang library: `io`, `list`, `dict`, `sets`, `gb_trees`
- You can inspect the source code for these libraries
- Eg. snippet from `/usr/local/lib/erlang/lib/stdlib-3.0/src/lists.erl`

```erlang
1 %% sum(L) returns the sum of the elements in L
2 -spec sum(List) -> number() when
3        List :: [number()].
4
5 sum(L)            -> sum(L, 0).
6
7 sum([H|T], Sum) -> sum(T, Sum + H);
8 sum([], Sum)     -> Sum.
```

Erlang

Shell

Data Types

Modules

Modules

- ▶ Basic compilation unit is a module
    - ▶ Module name = file name (.erl)
- ▶ They contain attributes and function definitions
- ▶ Attributes are of the form `-Name(Attribute).` and describe information about the module
- ▶ Let us create the module `math_examples` as follows.

```erlang
1 -module(math_examples).
2 -export([fact/1,area/1]).
3 -author("E.B").
4
5 fact(0) -> 1;
6 fact(N) when N>0 -> N * fact(N-1).
7
8 area({square, Side})  -> Side*Side ;
9 area({circle, Radius}) -> Radius*Radius*math:pi().
```

Modules

Running the examples.

```
1 1> c(math_examples).
2 {ok,math_examples}
3 2> math_examples:fact(3).
4 6
5 3> math_examples:area({square,4}).
6 16
7 4> math_examples:area({circle,1}).
8 3.141592653589793
```

Modules

▶ A useful compiler option is `export_all`

```
1 5> c(a_module,[export_all]).
2 {ok,a_module}
```

▶ This causes the compiler to ignore the `-export` module attribute and export all functions defined

▶ You can also do the following:

```
1 -module(math_examples).
2 -compile(export_all).
3 -author("E.B").
4
5 fact(0) -> 1;
6 fact(N) when N>0 -> N * fact(N-1).
7
8 area({square, Side})  -> Side*Side ;
9 area({circle, Radius}) -> Radius*Radius*math:pi().
```

Modules – Information About

```
1 1> c( math_examples ).
2 {ok , math_examples }
3 2> math_examples : module_info ().
4 [{ module , math_examples },
5  { exports ,[{ fact ,1},
6             { area ,1},
7             { module_info ,0},
8             { module_info ,1}]},
9  { attributes ,[{ vsn ,[292229879300425682399740783243125416564]
       },
10              { author ," E.B "}]},
11  { compile ,[{ options ,[]},
12             { version ," 7.0 "},
13             { source ," / Users / ebonelli / Documents / erlang /
                  math_examples . erl "}]},
14  { native , false },
15  {md5 , < <219 ,217 ,109 ,113 ,225 ,135 ,117 ,96 ,156 ,42 ,192 ,248 ,50 ,
16         41 ,98 ,116 > >}]
```

Modules – Command Line

▶ Compilation

```
1 $ erlc math_examples.erl
```

This generates a bytecode file (i.e. `.beam` file)

▶ Execution

```
1 $ erl -noshell -run math_examples factLstStr 5 -run
    init stop
```

`factStr` consumes a list of strings and then prints the result by calling `fact`

Records

- Declared as module attributes

Definition
```
-module(records).
-compile(export_all).

-record(robot, {name, type=industrial, hobbies, details=[]})
    .
```

Creation
```
first_robot() ->
    #robot{name="Mechatron", type=handmade,
           details=["Moved by a small man inside"]}.
```

Records are Syntactic Sugar for Tuples

Accessing record fields

```
1 1> c(records).
2 {ok,records}
3 2> records:first_robot().
4 {robot,"Mechatron",handmade,undefined,["Moved by a small man
      inside"]}
```

▶ Note above: a record is just syntactic sugar for a tuple

▶ That means that if we try to access a field, we'll get an error

```
1 3> (records:first_robot())#robot.name.
2 * 1: record robot undefined
```

Accessing the Fields

We must load the record definitions first

```
1 3> rr(records).
2 [robot]
3 4> records:first_robot().
4 #robot{name = "Mechatron",type = handmade,hobbies =
      undefined, details = ["Moved by a small man inside"]}
```

```
1 16> (records:first_robot())#robot.name.
2 "Mechatron"
3 17> Crusher = #robot{name="Crusher", hobbies=["Crushing
      people","petting cats"]}.
4 #robot{name = "Crusher",type = industrial,
5         hobbies = ["Crushing people","petting cats"],
6         details = []}
7 18> Crusher#robot.name.
8 "Crusher"
```

Updating Records

```
1 repairman(Rob) -> Details = Rob#robot.details,
2                   NewRob = Rob#robot{details=["Repaired by
3 repairman"|Details]}, {repaired, NewRob}.
```

```
1 16> c(records).
2 {ok,records}
3 17> records:repairman(#robot{name="Ulbert", hobbies=["trying
      to have feelings"]}).
4 {repaired,#robot{name = "Ulbert",type = industrial, hobbies
     = ["trying
5 to have feelings"], details = ["Repaired by repairman"]}}
```

Records in Header Files

```erlang
%% this is a .hrl (header) file.
-record(included, {some_field,
    some_default = "yeah!",
    unimaginative_name}).
```

To include it in records.erl, add the following line to the module:

```erlang
-include("records.hrl").
```

And an example function

```erlang
included() -> #included{some_field="Some value"}.
```

Records in Header Files

```
1 18> c(records).
2 {ok,records}
3 19> rr(records).
4 [included,robot,user]
5 20> records:included().
6 #included{some_field = "Some value",some_default = "yeah!",
7 unimaginative_name = undefined}
```