# Concurrent Programming

CS511

# About Erlang

- Functional language
- Concurrent
- Distributed
- "Soft" real-time (failure to meet deadline not considered system failure, but can degrade later output/performance)
- Open Telecom Platform (OTP) (fault-tolerance, hot code update, ...)
- Open source
- Developed in the 80s in Ericsson by Joe Armstrong, Robert Virding and Mike Williams

# Typical Applications

- Telecoms
  - Switches (POTS, ATM, IP, ...)
  - GPRS
  - SMS applications
- Internet applications
  - Twitter (backbone)
  - Facebook (chat)
  - Online shopping (Klarna AB)
  - T-Mobile
- 3D modelling (Wings3D)

# Essence

- A simple functional language
- No types at compile time
  - Dynamically typed
- No shared memory (message passing)
- Open Telecom Platform libraries
  - Practically “proven” programming patterns
  - Utilities

# Bibliography

- Programming Erlang: Software for a Concurrent World, Joe Armstrong
- Learn some Erlang for Great Good!, Fred Hebert
- Erlang Programming, Cesarini and Thompson

# Downloading and Installing

- `http://www.erlang.org/downloads` or 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/`

Erlang

Shell

Data Types

# Running Erlang

```
$ erl
Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:4:4] [asyn

Eshell V8.0  (abort with ^G)
1> io:format("Hello, world!~n").
Hello, world!
ok
2> q().
ok
$
```

# A Small Script

```
$ cat hello.erl
-module(hello).
-export([hello/0]).
hello() -> "Hello, world!".
$ erl
Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:4:4] [asyn

Eshell V8.0  (abort with ^G)
1> c(hello).
{ok,hello}
2> hello:hello().
"Hello, world!"
3> q().
```

# More on the Compiler

- Can pass options on to the compiler. Eg.

```
1 > c(hello, [native]).
```

- This uses HiPE (High Performance Native Compiler)

# Runtime System

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

Erlang

Shell

Data Types

# Sequential Fragment

- Data types and variables
- Function definitions
- Pattern matching
- Evaluation strategy
- Tail recursion

# Numbers

## Integers (arbitrarily big)

```
7> fact:fact(200).
788657867364790503552363213932185062295135977687173263294742533244359449963403342920304284011984623904177212138919638830257642790242637105061926624952829931113462857270763317237396988943922445621451664240254033291864131227428294853277524242407573903240321257405579568660226031904170324062351700858796178922227896237038973747200000000000000000000000000000000000000000
```

## Floats

```
> 2.78 + 9.6.
12.37999999999999
```

- 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: `"A String"`. They are, in fact, a list of integers.

```
10> "hello\7".
[104,101,108,108,111,7,104,101,108,108,111,7]
```

# Tuples

```
1 {}
2 {atom, another_atom, 'PPxT'}
3 {atom, {tup, 2}, {{tup}, element}}
```

# Modeling Data

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

- 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}}
```

# 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)]`

# Operators

- Arithmetic: +, -, *, /, `div`, `rem`
- Equal value: "==" and "/="
- Exact equality (type and value):
  "=:=" and "=/="
- List concatenation: "++"
- List subtraction: "--"
- List cons: "|"
- Boolean: `and`, `or`, `xor`, `not`, `andalso`, `orelse`

# Operator Examples

```
1> 4 == 4.0.          % value is 4 on both sides
true
2> 4 =:= 4.0.         % value same but type different
false
3> L1 = [ apple, cherry ].
[apple,cherry]
4> L2 = [ lime, grape ].
[lime,grape]
5> L3 = L1 ++ L2.
[apple,cherry,lime,grape]
6> L3 -- [cherry].
[apple,lime,grape]
7> L4 = [ banana | L3 ].
[banana,apple,cherry,lime,grape]
8> [ Head | Tail ] = L4.
[banana,apple,cherry,lime,grape]
```

# Operator Examples (cont.)

```
9> b().                    % b() shows all bindings
Head = banana              % Head and Tail have been bound
L3 = [apple,cherry,lime,grape]
L4 = [banana,apple,cherry,lime,grape]
Tail = [apple,cherry,lime,grape]
ok
10> f().                   % f() flushes all bindings
ok
11> { A, B } = { 4.0, 5.2 }.
{4.0,5.2}
12> b().
A = 4.0
B = 5.2
13> { C, D } = { 4.0, 5.2 }.
{4.0,5.2}
14> { A, B } == { C, D }.
true
15> { A, B } =:= { C, D }.
true
```

# Comparison

- En Erlang all terms are comparable
- Criteria:

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

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

```
16> a<2.
false
17> 2<a.
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

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

# More on Variables

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

# Records

## Definition

```
1 -record(person, {name = "", phone = [], address}).
```

## Creation

```
1 X = #person{name = "Joe", phone = [1,1,2], address= "here"}
```

## Accessing record fields

```
1 X#person.name
2 X#person{phone = [0,3,1,1,2,3,4]}
```

# Functions

- Separate cases by ;
- Finish definition with .

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

- Function application is call-by-value or eager

# Example

```
1 arith(X, Y) ->
2     io:fwrite("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:fwrite` is similar to `printf`
- Expressions separated by comma
- Function clause ended by period
- Final expression is function's return value

# Function Definition

- May have several clauses
  - function is sequence of pattern matching clauses separated by semicolons – semicolon means “or”
- Function call seeks to match arguments to pattern in some clause

Example:

```
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
```

# Function Examples, I

```
1  drivers_license(Age) when Age < 16 ->
2      forbidden ;
3  drivers_license(Age) when Age == 16 ->
4      'learners permit' ;
5  drivers_license(Age) when Age == 17 ->
6      'probationary license' ;
7  drivers_license(Age) when Age >= 65 ->
8      'vision test recommended but not required' ;
9  drivers_license(_) ->
10     'full license'.
```

- "`when ...`" is a clause guard
- Clause matches if function name, arguments, and all guards match the input

# Function Examples, II

```
$ erl
Erlang/OTP 19...

Eshell V8.0  (abort with ^G)
1> c(example).                  % c() compiles
{ok,example}
2> drivers_license(16).         % must specify module
** exception error: undefined shell command drivers_license/1
3> example:drivers_license(16).
'learners permit'
4> example:drivers_license(15).
forbidden
5> example:drivers_license(17).
'probationary license'
6> example:drivers_license(23).
'full license'
7> example:drivers_license(65).
'vision test recommended but not required'
8> q().
ok
```

# Function Call

Except for "built-in functions," must specify function's module when calling

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

Much-used modules in Erlang library:
`io`, `list`, `dict`, `sets`, `gb_trees`

# 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, Radio}) -> Radio*Radio*math:pi().
```

- Patterns: `{square, Side}` and `{circle, Radio}`
- `{square, Side}` matches `{square, 4}` and binds 4 to variable `Side`
- `{circle, Radio}` matches `{circle, 1}` and binds 1 to variable `Radio`

# Pattern Matching (cont.)

```
{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,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

# Modules

- Basic compilation unit is a module
  - Module name = file name (.erl)
- Modules contain function definitions
  - Like factorial and area (see above)
  - Some functions can be exported to be usable from outside of the module
- Let us create the module `math_examples` as follows.

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

# Modules

Running the examples.

```
1  > c(math_examples).
2  {ok,math_examples}
3  > math_examples:fact(3).
4  6
5  > math_examples:area({square,4}).
6  16
7  > math_examples:area({circle,1}).
8  3.14159265358979 3
```

# List Examples

```
1 > c(list_examples).
2 {ok,list_examples}
3 > list_examples:sum([1,2,3,4]).
4 10
5 > list_examples:len([0,1,0,1]).
6 4
7 > list_examples:append([5,4],[1,2,3]).
8 [5,4,1,2,3]
```

- We will define them recursively (inductively)
  - Base case: empty list ([])
  - Recursive case: a list with at least one element ([X — XS])

# Tail Recursion

- Programming pattern to increase performance
- It helps compilers when optimizing code!
- Inefficient recursive definition

```
1 len([_|XS]) -> 1 + len(XS) ;
2 len([])     -> 0.
```

Observe the evaluation of `len([1,2,3])`

```
1 len([1,2,3]) == 1 + len([2,3])
2 len([1,2,3]) == 1 + (1 + len([3]))
3 len([1,2,3]) == 1 + (1 + (1 + len([]))) %%
4 len([1,2,3]) == 1 + (1 + (1 + 0))
5 len([1,2,3]) == 1 + (1 + 1)
6 len([1,2,3]) == 1 + 2
7 len([1,2,3]) == 3
```

- At the time of reaching the marked line, Erlang needs to keep in memory a long expression
- After that line, it starts shrinking the expression
- Imaging how it will work for a very big list!

# Tail Recursion

- More efficiency by tail recursion
- Space (constant if we assume elements of the list have the same size)
- Efficiency (No returns from recursive calls)
- What is the trick?
  - Use of accumulators (partial results)
  - There are no more computations after the recursive call

# Tail Recursion

- We define `len_a`, the tail recursive version of len
- Function `len_a` has an extra parameters capturing the partial result of the function, i.e., how many elements `len_a` has seen so far

```
1 len_a([_|XS], Acc) -> len_a(XS, Acc+1);
2 len_a([], Acc) -> Acc.
```

We define len based on `len_a` as follows

```
1 len(XS) -> len_a(XS, 0).
```

What about the tail recursive version of sum and append?

# Control Structures

```
1 is_greater_than(X, Y) ->
2     if
3         X>Y ->
4             true;
5         true -> % works as an 'else' branch
6             false
7     end
```

# Control Structures

```
1 is_valid_signal(Signal) ->
2     case Signal of
3         {signal, _What, _From, _To} ->
4             true;
5         {signal, _What, _To} ->
6             true;
7         _Else ->
8             false
9     end.
```