WEEK-1 PRESENTED BY NOLLEH

LEARN U ERLANG

WHAT IS ERLANG?

- immutable variable
  - forbidden

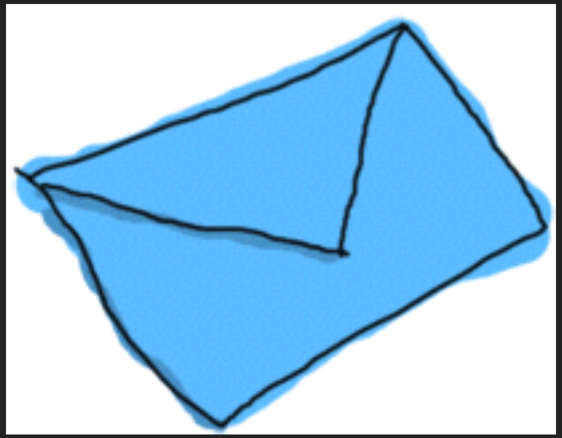
X = 5 + 1
X = X
∴ 5 = 6

- referential transparency
  - means.. composite
  - some cases where it's useful to break

X = TODAY() = 2009/10/22
-- WAIT A DAY --
X = TODAY() = 2009/10/23
X = X
∴ 2009/10/22 = 2009/10/23

WHAT IS ERLANG?

- Functional + Concurrency + Reliability
  - actor model (each actor do job in vm)
    - dark room
    - distinct task
    - message pass



WHAT IS ERLANG?

- development env
  - compiled to byte code, run in vm
  - distribution
- vm + libraries
  - update on running program

DON'T DRINK TOO MUCH KOOL AID

- reminder to keep your feet on the ground
  - scaling ability
    - can do doesn't mean have to
  - propotional manner to # of core ?
    - in few case..
  - choose right tool
    - not silver bullet
  - do not wall your self with erlang
    - ai, wings 3D…

WHAT YOU NEED TO DIVE IN

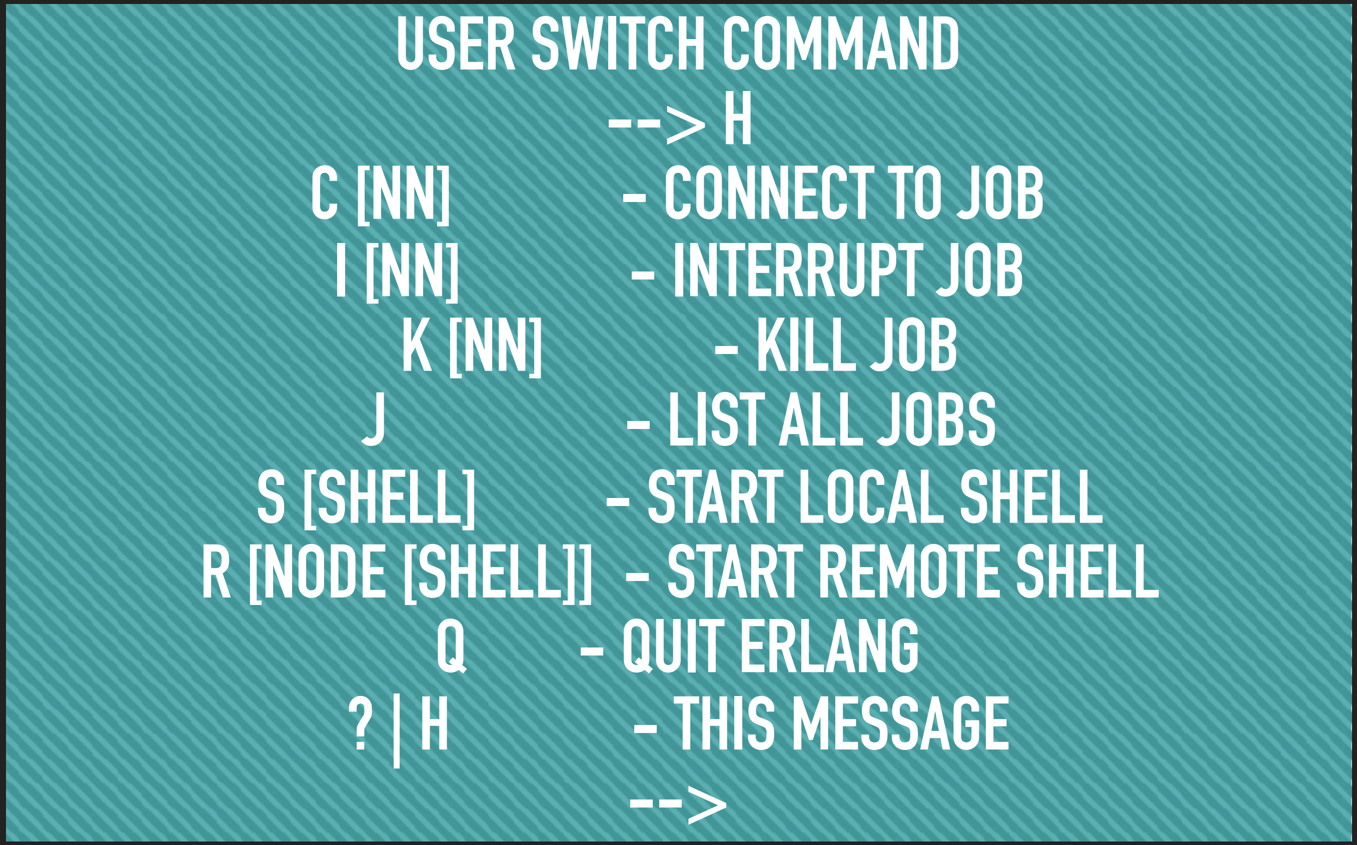
- windows
  - download binaries
- linux
  - (debian based) apt-get install erlang
  - (fedora) yum insall erlang
- osx
  - brew install erlang / port install erlang

WHERE TO GET HELP

- linux
  - erl -man lists
- windows
  - http://erlang.org/doc/
- coding practice
  - http://www.erlang.se/doc/programming_rules.shtml

THE SHELL AND SHELL COMMAND

- shell
  - $erl, werl.exe
- shell command
  - ^A, ^E, tab, help(), ^G



```
USER SWITCH COMMAND
 --> H
  C [NN]            - CONNECT TO JOB
  I [NN]            - INTERRUPT JOB
  K [NN]            - KILL JOB
  J                 - LIST ALL JOBS
  S [SHELL]         - START LOCAL SHELL
  R [NODE [SHELL]]  - START REMOTE SHELL
  Q                 - QUIT ERLANG
  ? | H             - THIS MESSAGE
 -->
```

NUMBERS

- expression terminate with period followed by whitespace
- separate with comma

```
3> 1892 - 1472.                 => 420
4> 5 / 2.                       => 2.5
5> 5 div 2.                     => 2
6> 5 rem 2.                     => 1
10> 2#101010.                   => 42
11> 8#0677.                     => 447
12> 16#AE.                      => 174
```

INVARIABLE VARIABLES

- examples

```
1> One.
1: variable 'One' is unbound
2> One = 1.
1
3> Un = Uno = One = 1.
1
4> Two = One + One.
2
5> Two = 2.
2
6> Two = Two + 1.
** exception error: no match of right hand side value 3
```

INVARIABLE VARIABLES

- assign once
- pretends assign
  - depends on = operator
    - compare (complain) + return
- left hand side term is variable and unbound
  - bind + comparison + keep in the memory
- basis on pattern matching

INVARIABLE VARIABLES

- variable name must begin with capital
  - technically possible begin with underscore
- variable '_'
  - store nothing

ATOMS

- literals, constants with own name for value

```
1> atom.
atom
2> atoms_rule.
atoms_rule
3> atoms_rule@erlang.
atoms_rule@erlang
4> 'Atoms can be cheated!'.
'Atoms can be cheated!'
5> atom = 'atom'.
atom
```

ATOMS

- worked with code that used constants before
  - check chapter2. modules
- best use will come when coupled with other types of data
- referred in atom table
  - 4byte/atom (32bit) , 8byte/atom (64bit)
  - not garbage collected
  - should be seen as tools for developer

BOOLEAN ALGEBRA & COMPARISON OPERATORS

- and / or / xor / not
  - evaluates both side of operator (!= andalso/orelse)

```
1> true and false.
false
2> false or true.
true
3> true xor false.
true
4> not false.
true
5> not (true and true).
false
```

BOOLEAN ALGEBRA & COMPARISON OPERATORS

- =:= , =/=, ==, /=
- =:=, =/=
  - distinct int <=> floats
- ==, /=
  - do not distinct int <=> floats

```
6> 5 =:= 5.
true
7> 1 =:= 0.
false
8> 1 =/= 0.
true
9> 5 =:= 5.0.
false
10> 5 == 5.0.
true
11> 5 /= 5.0.
false
```

BOOLEAN ALGEBRA & COMPARISON OPERATORS

- comparision with different types
  - disallow

```
12> 5 + llama.
** exception error: bad argument in an arithmetic expression
in operator  +/2
called as 5 + llama
```

  - allow

```
13> 5 =:= true.
false
```

BOOLEAN ALGEBRA & COMPARISON OPERATORS

- redherring..

```
14> 0 == false.
false
15> 1 < false.
true
```

- total ordering

number < atom < reference < fun < port < pid < tuple < list <
bit string

TUPLES

- way of organize data (knowing how many data)
- carry 2 -> ignore y
  - x as unbound, unpack tuple (pattern matching)
  - _
    - drop value
    - wild card in pattern matching

```
3> Point = {4,5}.
{4,5}
4> {X,Y} = Point.
{4,5}
5> X.
4
6> {X,_} = Point.
{4,5}
```

TUPLES

- also useful when store 1 data (tagged tuple)

```
10> PreciseTemperature = {celsius, 23.213}.
{celsius,23.213}
11> {kelvin, T} = PreciseTemperature.
** exception error: no match of right hand side value {celsius,
23.213}
```

LISTS

- the most used data
- able to mix more than one type
- show as number only when one of them is can’t be represented as letter

```
3> [97,98,99,4,5,6].
[97,98,99,4,5,6]
4> [233].
"é"
```

LISTS

- ++ , --
  - both right associative

```
5> [1,2,3] ++ [4,5].
[1,2,3,4,5]
6> [1,2,3,4,5] -- [1,2,3].
[4,5]
9> [1,2,3] -- [1,2] -- [3].
[3]
10> [1,2,3] -- [1,2] -- [2].
[2,3]
```

LISTS

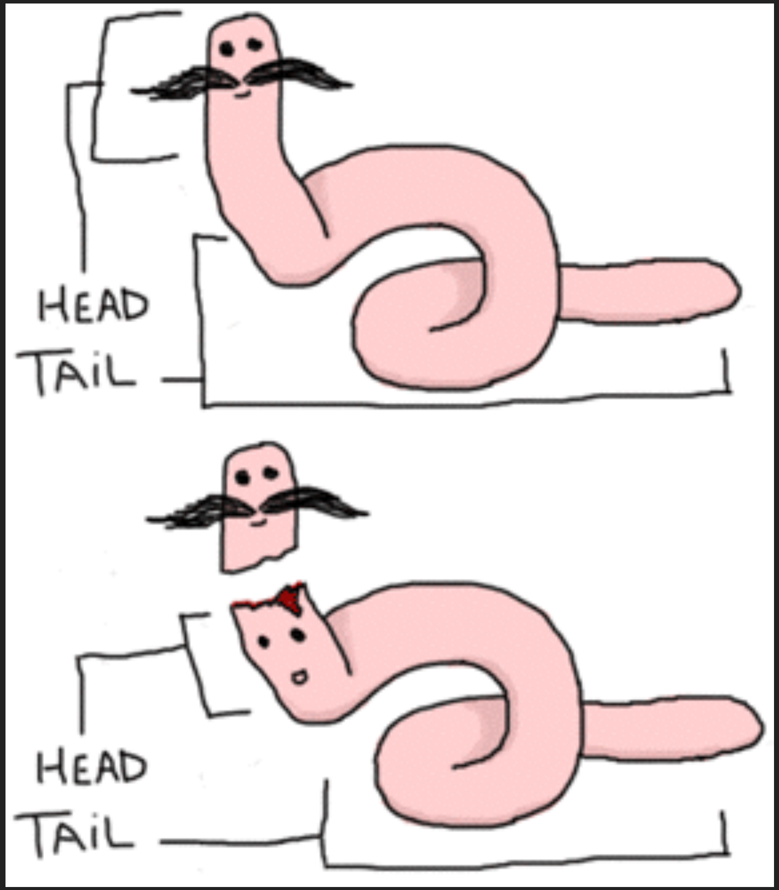
- head, tail
- pattern matching

```
11> hd([1,2,3,4]).
1
12> tl([1,2,3,4]).
[2,3,4]
13> List = [2,3,4].
[2,3,4]
14> NewList = [1|List].
[1,2,3,4]
```

```
15> [Head|Tail] = NewList.
[1,2,3,4]
16> Head.
1
17> Tail.
[2,3,4]
```

LISTS

- familiar with cons
- proper list - terminate with []



```
[a, b, c, d]
[a, b, c, d | []]
[a, b | [c, d]]
[a, b | [c | [d]]]
[a | [b | [c | [d]]]]
[a | [b | [c | [d | [] ]]]]
```

LIST COMPREHENSIONS

- similar with set notation
  - how to build a set by specifying properties its members must satisfy
  - arrow works as '=', with doesn't throwing exception

```
1> [2*N || N <- [1,2,3,4]].
[2,4,6,8]
```

LIST COMPREHENSIONS

- filtering with comma

```
2> [X || X <- [1,2,3,4,5,6,7,8,9,10], X rem 2 =:= 0].
[2,4,6,8,10]
```

- ignore exception

```
6> Weather = [{toronto, rain}, {montreal, storms}, {london, fog},
6>           {paris, sun}, {boston, fog}, {vancouver, snow}].

7> FoggyPlaces = [X || {X, fog} <- Weather].
[london,boston]
```

BIT SYNTAX!

- <<binary>>

```
1> Color = 16#F09A29.
15768105
2> Pixel = <<Color:24>>.
<<240,154,41>>
```

- unpack pattern matching

```
8> <<R:8, Rest/binary>> = Pixels.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
9> R.
213
```

BIT SYNTAX!

- describing way of binary segment

```
Value
Value:Size
Value/TypeSpecifierList
Value:Size/TypeSpecifierList
```

- Types
  - represent kind of binary data used
  - default is integer

```
integer | float | binary | bytes | bitstring | bits | utf8 | utf16 | utf32
```

BIT SYNTAX!

- Signedness
  - only matters for matching type is integer
  - default is unsigned

```
signed | unsigned
```

- endianness
  - only matters when type is integer, utf8~32, float
  - default is big

```
big | little | native
```

BIT SYNTAX!

- Unit
  - size of each segment, in bits
  - allowed range is 1..256
  - usually used to ensure byte-alignment

```
unit:Integer
```

BIT SYNTAX!

- example using binary representations

```
10> <<X1/unsigned>> =  <<-44>>.
<<"Ô">>
11> X1.
212
12> <<X2/signed>> =  <<-44>>.
<<"Ô">>
13> X2.
-44
19> <<Y:4/little-unit:8>> = <<72,0,0,0>>.
<<72,0,0,0>>
20> Y.
72
```

BIT SYNTAX!

- binary operators

```
2#00100 = 2#00010 bsl 1.
2#00001 = 2#00010 bsr 1.
2#10101 = 2#10001 bor 2#00101.
```

- parse TCP segment

```
<<SourcePort:16, DestinationPort:16,
AckNumber:32,
DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,
CheckSum: 16, UrgentPointer:16,
Payload/binary>> = SomeBinary.
```

BIT SYNTAX!

- new notation : bit string
  - pros - space (just array vs linked list)
  - cons - loss simplicity when used pattern mating, manipulation
  - <<“this is bit string!”>>

BINARY COMPREHENSIONS

- since R13B

```
1> [ X || <<X>> <= <<1,2,3,4,5>>, X rem 2 == 0].
[2,4]
```

```
2> Pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
3> RGB = [ {R,G,B} || <<R:8,G:8,B:8>> <= Pixels ].
[{213,45,132},{64,76,32},{76,0,0},{234,32,15}]
```

```
5> << <<Bin>> || Bin <- [<<3,7,5,4,7>>] >>.
** exception error: bad argument
6> << <<Bin/binary>> || Bin <- [<<3,7,5,4,7>>] >>.
<<3,7,5,4,7>>
```

WHAT ARE MODULES

- do not being terrible erlang programmer
- bunch of functions regrouped in a single file, under name
- all functions must be defined in modules
- you already used it! (hd , tl, arithmetic…)

WHAT ARE MODULES

- Using Module

```
Module:Function(Arguments)
```

```
1> erlang:element(2, {a,b,c}).
b
2> element(2, {a,b,c}).
b
3> lists:seq(1,4).
[1,2,3,4]
4> seq(1,4).
** exception error: undefined shell command seq/2
```

WHAT ARE MODULES

- put similar things inside a single module
  - except elrang module

MODULE DECLARATION

- functions
- attribute
    - describe functions
        - name, visibility, author, …
    - hint for complier
    - hint for people

MODULE DECLARATION

- declare module

```
-Name(Attribute)
```

- for compilable, declare this

```
-module(name).
```

- file name should be same with module name

MODULE DECLARATION

- for export function, declare this

```
-export([Function1/Arity, Function2/Arity, ..., FunctionN/Arity]).
```

- define function

```
Name(args) -> Body.
```

  - Body : expressions separated by comma

MODULE DECLARATION

- import module, function

```
-import(Module, [Function1/Arity, ..., FunctionN/Arity]).
```

- instead import, prefer modulename:func
- macro
  - declare macro

```
define(MACRO, some_value).
```

  - use macro

```
?MACRO
```

MODULE DECLARATION

```
-module(useless).
-export([add/2, hello/0, greet_and_add_two/1]).

add(A,B) ->
A + B.

%% Shows greetings.
%% io:format/1 is the standard function used to output text.
hello() ->
io:format("Hello, world!~n").

greet_and_add_two(X) ->
hello(),
add(X,2).
```

COMPILING THE CODE

- complie
  - command line

```
$ erlc flags file.erl
```

  - shell / in module

```
compile:file(FileName)
```

  - shell

```
c()
```

COMPILING THE CODE

- compile

```
1> cd("/path/to/where/you/saved/the-module/").
"Path Name to the directory you are in"
ok
```

```
2> c(useless).
{ok,useless}
```

```
3> useless:add(7,2).
9
4> useless:hello().
Hello, world!
ok
```

COMPILING THE CODE

- compile flags

```
-debug_info
-{outdir,Dir}
-export_all
-{d,Macro} or {d,Macro,Value}
```

```
7> compile:file(useless, [debug_info, export_all]).
{ok,useless}
8> c(useless, [debug_info, export_all]).
{ok,useless}
```

```
-compile([debug_info, export_all]).
```

COMPILING THE CODE

- compile to native code
  - 20% faster?
  - doesn't work for every platform / os ..

```
hipe:c(Module,OptionsList).

c(Module,[native]).
```

MORE ABOUT MODULES

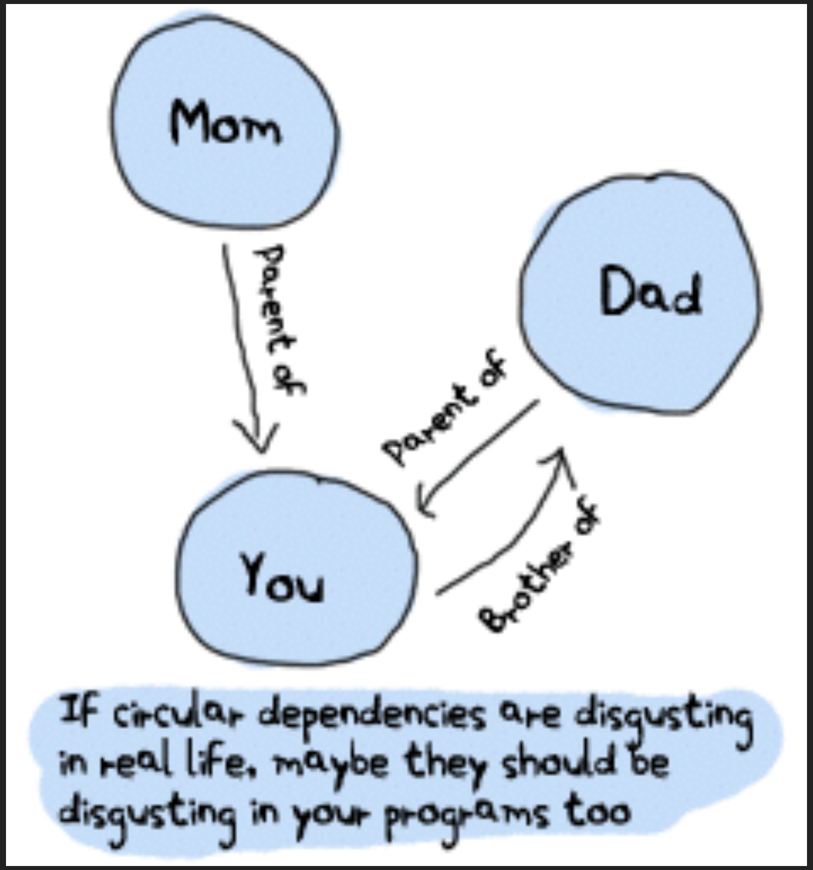
- where is the meta data ?
  - compiler stored.
  - invoke module_name:module_info().

MORE ABOUT MODULES

```
9> useless:module_info().
[{exports,[{add,2},
{hello,0},
{greet_and_add_two,1},
{module_info,0},
{module_info,1}]},
{imports,[]},
{attributes,[{vsn,[174839656007867314473085021121413256129]}]},
{compile,[{options,[]},
{version,"4.6.2"},
{time,{2009,9,9,22,15,50}},
{source,"/home/ferd/learn-you-some-erlang/useless.erl"}]}]
10> useless:module_info(attributes).
[{vsn,[174839656007867314473085021121413256129]}]
```

MORE ABOUT MODULES

- think
  - circular dependency
  - do not depends too many module
  - regroup functions, have similar roles


Mom
Parent of
Dad
Parent of
You
Brother of
If circular dependencies are disgusting in real life, maybe they should be disgusting in your programs too