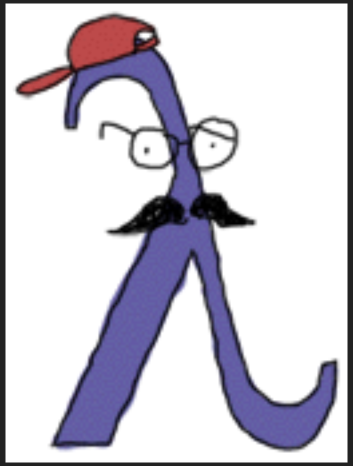
WEEK3 - PRESENTED BY NOLLEH

LEARN U ERLANG

LET'S GET FUNCTIONAL

- in functional programming language…
  - ability to take function
  - used like variable within function
  - THIS IS HIGHER ORDER FUNCTION!
- lambda calculus
  - it is originated from mathematics
  - system, see everything as function, even numbers



LET'S GET FUNCTIONAL

- higher order function - add/2

```
-module(hhfuns).
-compile(export_all).

one() -> 1.
two() -> 2.

add(X,Y) -> X() + Y().
```

LET'S GET FUNCTIONAL

- higher order function - add/2

```
4> hhfuns:add(fun hhfuns:one/0, fun hhfuns:two/0).
3
```

fun Module:Function/Arity

LET'S GET FUNCTIONAL

- where comes from it's strength?

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

decrement([]) -> [];
decrement([H|T]) -> [H-1|decrement(T)].
```

- we will change these functions a bit generally.

LET'S GET FUNCTIONAL

- look at that!
- very smart abstraction!

```
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].

incr(X) -> X + 1.
decr(X) -> X - 1.
```

```
5> hhfuns:map(fun hhfuns:incr/1,  [1,2,3,4,5]).
  [2,3,4,5,6]
6> hhfuns:map(fun hhfuns:decr/1,  [1,2,3,4,5]).
  [0,1,2,3,4]
```

ANONYMOUS FUNCTIONS

- function inline

```
fun(Args1) ->
      Expression1, Exp2, ..., ExpN;
    (Args2) ->
      Expression1, Exp2, ..., ExpN;
    (Args3) ->
      Expression1, Exp2, ..., ExpN
end
```

ANONYMOUS FUNCTIONS

- map with lambda

```
9> hhfuns:map(fun(X) -> X + 1 end, L).
  [2,3,4,5,6]
10> hhfuns:map(fun(X) -> X - 1 end, L).
  [0,1,2,3,4]
```

- the loop logic is can be ignored. instead, focused on what will be done to elements

ANONYMOUS FUNCTIONS

- closure

```
11> PrepareAlarm = fun(Room) ->
11>                     io:format("Alarm set in ~s.~n",[Room]),
11>                     fun() -> io:format("Alarm tripped in ~s! Call Batman!
                                                ~n",[Room]) end
11>                   end.
  #Fun<erl_eval.20.67289768>
12> AlarmReady = PrepareAlarm("bathroom").
  Alarm set in bathroom.
  #Fun<erl_eval.6.13229925>
13> AlarmReady().
  Alarm tripped in bathroom! Call Batman!
  ok
```

ANONYMOUS FUNCTIONS

- understand scope

```
base(A) ->
  B = A + 1,
  F = fun() -> A * B end,
  F().
```

- F inherits base/1's scope

ANONYMOUS FUNCTIONS

- a parent cannot inherits it’s decent’s scope

```
base(A) ->
  B = A + 1,
  F = fun() -> C = A * B end,
  F(),
  C.
```

ANONYMOUS FUNCTIONS

- carry it’s context

```
a() ->
  Secret = "pony",
  fun() -> Secret end.


b(F) ->
  "a/0's password is "++F().
```

```
14> c(hhfuns).
  {ok, hhfuns}
15> hhfuns:b(hhfuns:a()).
  "a/0's password is pony"
```

ANONYMOUS FUNCTIONS

- shadowing

```
base() ->
  A = 1,
  (fun() -> A = 2 end)().
```

```
base() ->
  A = 1,
  (fun(A) -> A = 2 end)(2).
```

MAPS, FILTERS, FOLDS AND MORE

- like map, you can abstract functions like even, old_man..

```
even(L) -> lists:reverse(even(L,[])).
even([], Acc) -> Acc;
even([H|T], Acc) when H rem 2 == 0 ->
even(T, [H|Acc]);
even([_|T], Acc) -> even(T, Acc).

old_men(L) -> lists:reverse(old_men(L,[])).
old_men([], Acc) -> Acc;
old_men([Person = {male, Age}|People], Acc) when Age > 60 ->
old_men(People, [Person|Acc]);
old_men([_|People], Acc) -> old_men(People, Acc).
```

MAPS, FILTERS, FOLDS AND MORE

- extract the common parts

```
filter(Pred, L) -> lists:reverse(filter(Pred, L,[])).

filter(_, [], Acc) -> Acc;
filter(Pred, [H|T], Acc) ->
   case Pred(H) of
   true  -> filter(Pred, T, [H|Acc]);
   false -> filter(Pred, T, Acc)
   end.
```

- try to get rid of what's always the same and let the programmer supply in the parts that change.

MAPS, FILTERS, FOLDS AND MORE

- use it!

```
1> c(hhfuns).
2> Numbers = lists:seq(1,10).
3> hhfuns:filter(fun(X) -> X rem 2 == 0 end, Numbers).
[2,4,6,8,10]
4> People = [{male,45},{female,67},{male,66},{female,12},{unknown,
                174},{male,74}].
[{male,45},{female,67},{male,66},{female,12},{unknown,174},{male,74}]
5> hhfuns:filter(fun({Gender,Age}) -> Gender == male andalso Age >
                60 end, People).
[{male,66},{male,74}]
```

MAPS, FILTERS, FOLDS AND MORE

- fold - it reduces to one value

```
%% find the maximum of a list
max([H|T]) -> max2(T, H).

max2([], Max) -> Max;
max2([H|T], Max) when H > Max -> max2(T, H);
max2([_|T], Max) -> max2(T, Max).
```

MAPS, FILTERS, FOLDS AND MORE

- fold - it reduces to one value

```
%% find the minimum of a list
min([H|T]) -> min2(T,H).

min2([], Min) -> Min;
min2([H|T], Min) when H < Min -> min2(T,H);
min2([_|T], Min) -> min2(T, Min).
```

MAPS, FILTERS, FOLDS AND MORE

- fold - it reduces to one value

```
%% sum of all the elements of a list
sum(L) -> sum(L,0).

sum([], Sum) -> Sum;
sum([H|T], Sum) -> sum(T, H+Sum).
```

MAPS, FILTERS, FOLDS AND MORE

```
max([H|T]) -> max2(T, H).
max2([], Max) -> Max;
max2([H|T], Max) when H > Max -> max2(T, H);
max2([_|T], Max) -> max2(T, Max).

min([H|T]) -> min2(T,H).
min2([], Min) -> Min;
min2([H|T], Min) when H < Min -> min2(T,H);
min2([_|T], Min) -> min2(T, Min).

sum(L) -> sum(L,0).
sum([], Sum) -> Sum;
sum([H|T], Sum) -> sum(T, H+Sum).
```

MAPS, FILTERS, FOLDS AND MORE

- fold - it reduces to one value

```
fold(_, Start, []) -> Start;
fold(F, Start, [H|T]) -> fold(F, F(H,Start), T).
```

MAPS, FILTERS, FOLDS AND MORE

- use it!

```
6> c(hhfuns).
{ok, hhfuns}
7> [H|T] = [1,7,3,5,9,0,2,3].
[1,7,3,5,9,0,2,3]
8> hhfuns:fold(fun(A,B) when A > B -> A; (_,B) -> B end, H, T).
9
9> hhfuns:fold(fun(A,B) when A < B -> A; (_,B) -> B end, H, T).
0
10> hhfuns:fold(fun(A,B) -> A + B end, 0, lists:seq(1,6)).
21
```

MAPS, FILTERS, FOLDS AND MORE

```
reverse(L) ->
fold(fun(X,Acc) -> [X|Acc] end, [], L).

map2(F,L) ->
reverse(fold(fun(X,Acc) -> [F(X)|Acc] end, [], L)).

filter2(Pred, L) ->
   F = fun(X,Acc) ->
      case Pred(X) of
         true  -> [X|Acc];
         false -> Acc
      end
end,
reverse(fold(F, [], L)).
```

MAPS, FILTERS, FOLDS AND MORE

- U can using it from Standard library

```
list:map/2, list:filter/2, list:foldl/3, list:foldr/3
all/2, any/2
dropwhile/2, takewhile/2,
partition/2,
flatten/1, flatlength/1, flatmap/2, merge/1, nth/2, nthtail/2, split/2
```

NOT SO FAST!

- you are probably running into errors
- can’t explain all error handling, now
  - erlang has 2 paradigm
    - functional / concurrent
- even thought, keep it mind that… let it crash~

A COMPILATION OF ERRORS

- syntactic mistake
  - module.beam:module name does not match file name ‘module’
  - ./module.erl:2: Warning: function some_function/0 is unused
    - not export, used with wrong name..
  - ./module.erl:2: function some_function/1 undefined
    - wrong name, arity, function could not be compiled..(forgot period)

A COMPILATION OF ERRORS

- ./module.erl:5: syntax error before: ‘SomeCharacterOrWord'
  - unclosed parenthesis, wrong expression termination, encoding…
- ./module.erl:5: syntax error before:
  - line termination is not correct (specific case of previous error)
- ./module.erl:5: Warning: this expression will fail with a 'badarith' exception
  - llama + 5
- ./module.erl:5: Warning: variable 'Var' is unused

A COMPILATION OF ERRORS

- ./module.erl:5: head mismatch
  - function has more than one head, which is different, i.e. arty
- ./module.erl:5: Warning: this clause cannot match because a previous clause at line 4 always matches
- ./module.erl:9: variable 'A' unsafe in 'case' (line 5)
  - using a variable declared within branch of case … of
  - if you want to use, MyVar = case … of…

NO! YOUR LOGIC IS WRONG!

- using debug tool
  - test framework, tracing module...
- It's easier to focus on those that make your programs crash
  - won't bubble up 50 levels from now

RUNTIME ERRORS

- crash your code
- function_clause

```
1> lists:sort([3,2,1]).
[1,2,3]
2> lists:sort(fffffff).
** exception error: no function clause matching lists:sort(fffffff)
```

- case_clause

```
3> case "Unexpected Value" of
3>    expected_value -> ok;
3>    other_expected_value -> 'also ok'
3> end.
```

RUNTIME ERRORS

- if_clause

```
4> if 2 > 4 -> ok;
4>    0 > 1 -> ok
4> end.
** exception error: no true branch found when evaluating an if
 expression
```

- badmatch

```
5> [X,Y] = {4,5}.
** exception error: no match of right hand side value {4,5}
```

RUNTIME ERRORS

- badarg

```
6> erlang:binary_to_list("heh, already a list").
** exception error: bad argument
in function  binary_to_list/1
called as binary_to_list("heh, already a list")
```

- undef

```
7> lists:random([1,2,3]).
** exception error: undefined function lists:random/1
```

RUNTIME ERRORS

- badarith

```
8> 5 + llama.
** exception error: bad argument in an arithmetic expression
   in operator  +/2
      called as 5 + llama
```

- badfun

```
9> hhfuns:add(one,two).
** exception error: bad function one
in function  hhfuns:add/2
```

RUNTIME ERRORS

- badarity

```
10> F = fun(_) -> ok end.
#Fun<erl_eval.6.13229925>
11> F(a,b).
** exception error: interpreted function with arity 1 called with two
arguments
```

- system_limit
  - atom too large, too many arguments, # of atom is too large…
  - http://www.erlang.org/doc/efficiency_guide/advanced.html#id2265856

RAISING EXCEPTIONS

- let it crash!
- 3 kind of exceptions
  - errors
  - exits
  - throws

RAISING EXCEPTIONS

- errors

```
erlang:error(Reason)
```

  - end execution of current process
  - include stacktrace
  - the error what you've seen in previous clause.
  - do not use where user must take care (tree lookup)
    - {ok, value} / undefined…

RAISING EXCEPTIONS

- define you’re own kind of error

```
1> erlang:error(badarith).
** exception error: bad argument in an arithmetic expression
2> erlang:error(custom_error).
** exception error: custom_error
```

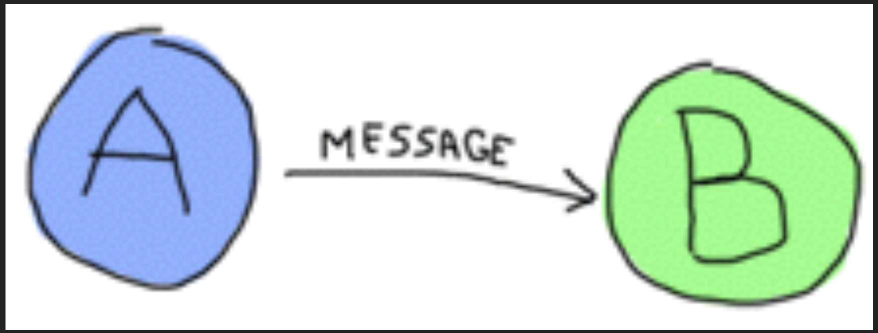
  - can be handled in same manner

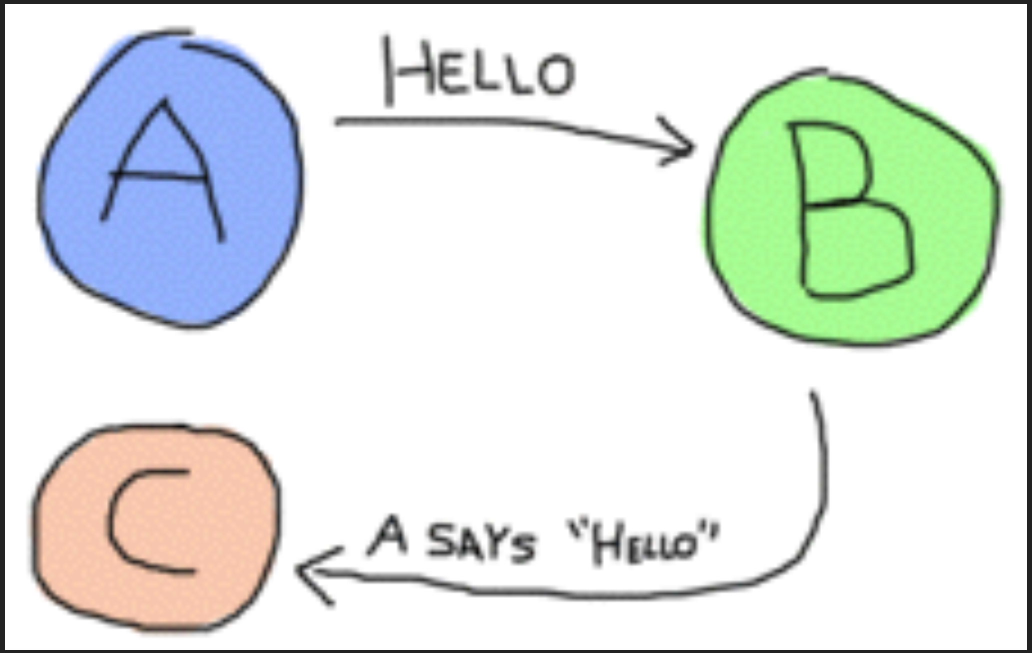
RAISING EXCEPTIONS

- exits
  - internal (exit/1) / external (exit/2)
- roughly, has same use case with error
- what to use ?

RAISING EXCEPTIONS

- understand concept of actors and processes

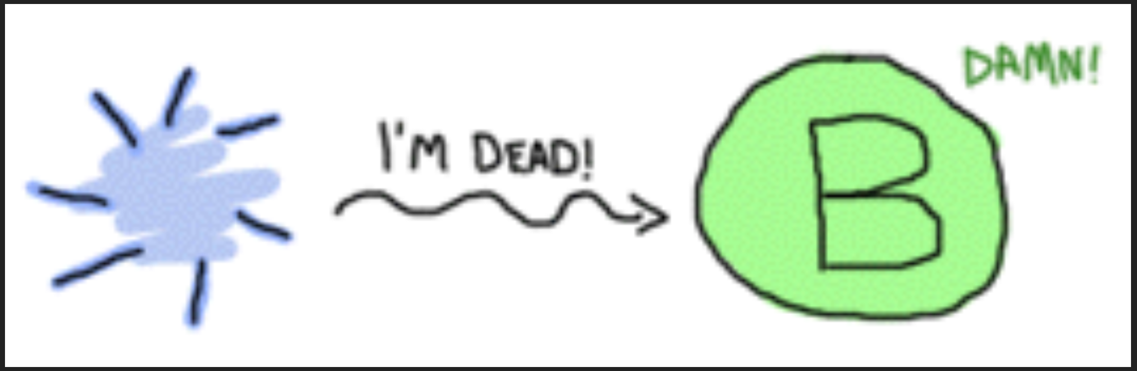




  - listen, pass msg

RAISING EXCEPTIONS

- understand concept of actors and processes



  - communicate exceptions
- the real deference is intent (simply error or terminate?)
- errors - contain stack trace / exit - not contain stack trace

RAISING EXCEPTIONS

- throws
  - programmer can be expected to handle
  - recommend documenting them

```
1> throw(permission_denied).
** exception throw: permission_denied
```

- used for non-local returns in deep recursion

DEALING WITH EXCEPTIONS

- try…catch

```
try Expression of
  SuccessfulPattern1 [Guards] ->
    Expression1;
  SuccessfulPattern2 [Guards] ->
    Expression2
  catch
    TypeOfError:ExceptionPattern1 ->
      Expression3;
    TypeOfError:ExceptionPattern2 ->
      Expression4
end.
```

DEALING WITH EXCEPTIONS

- protected
  - any exception happening within the call will be caught
- replace TypeOfError error, exit, throw..
  - default is throw

DEALING WITH EXCEPTIONS

```
sword(1) -> throw(slice);
sword(2) -> erlang:error(cut_arm);
sword(3) -> exit(cut_leg);
sword(4) -> throw(punch);
sword(5) -> exit(cross_bridge).

black_knight(Attack) when is_function(Attack, 0) ->
  try Attack() of
    _ -> "None shall pass."
  catch
    throw:slice -> "It is but a scratch.";
    error:cut_arm -> "I've had worse.";
    exit:cut_leg -> "Come on you pansy!";
    _:_ -> "Just a flesh wound."
  end.
```

DEALING WITH EXCEPTIONS

- example

```
9> exceptions:black_knight(fun exceptions:talk/0).
  "None shall pass."
10> exceptions:black_knight(fun() -> exceptions:sword(1) end).
  "It is but a scratch."
11> exceptions:black_knight(fun() -> exceptions:sword(2) end).
  "I've had worse."
12> exceptions:black_knight(fun() -> exceptions:sword(3) end).
  "Come on you pansy!"
13> exceptions:black_knight(fun() -> exceptions:sword(4) end).
  "Just a flesh wound."
14> exceptions:black_knight(fun() -> exceptions:sword(5) end).
  "Just a flesh wound."
```

DEALING WITH EXCEPTIONS

- _:_ patttern catch all
- finally - after

```
try Expr of
  Pattern -> Expr1
catch
  Type:Exception -> Expr2
after % this always gets executed
  Expr3
end
```

  - cannot get return value from this

DEALING WITH EXCEPTIONS

- multiple protection

```
whoa() ->
  try
    talk(),
    _Knight = "None shall Pass!",
    _Doubles = [N*2 || N <- lists:seq(1,100)],
    throw(up),
    _WillReturnThis = tequila
  of
    tequila -> "hey this worked!"
  catch
    Exception:Reason -> {caught, Exception, Reason}
  end.
```

DEALING WITH EXCEPTIONS

- when return value is not useful, remove of part

```
im_impressed() ->
  try
    talk(),
    _Knight = "None shall Pass!",
    _Doubles = [N*2 || N <- lists:seq(1,100)],
    throw(up),
    _WillReturnThis = tequila
  catch
    Exception:Reason -> {caught, Exception, Reason}
  end.
```

DEALING WITH EXCEPTIONS

- protected part cannot be tail recursive
- of - catch space can be tail recursive

WAIT, THERE'S MORE!

- keyword catch
    - basically captures all types of exceptions on top of the good results.

```
1> catch throw(whoa).
  whoa
2> catch exit(die).
  {'EXIT',die}
```

WAIT, THERE'S MORE!

- both ‘exit’ - backward compatibility

```
3> catch 1/0.
  {'EXIT',{badarith,[{erlang,'/',[1,0]},
              {erl_eval,do_apply,5},
              {erl_eval,expr,5},
              {shell,exprs,6},
              {shell,eval_exprs,6},
              {shell,eval_loop,3}]}}
4> catch 2+2.
  4
```

WAIT, THERE'S MORE!

```
5> catch doesnt:exist(a,4).
  {'EXIT',{undef,[{doesnt,exist,[a,4]},
  {erl_eval,do_apply,5},
  {erl_eval,expr,5},
  {shell,exprs,6},
  {shell,eval_exprs,6},
  {shell,eval_loop,3}]}}
```

- undef:type of error
- stack trace {Module, Function, Arguments}
- stack trace {Module, Function, Arity}.
- erlang:get_stacktrace/0

WAIT, THERE'S MORE!

- common manner using catch

```
catcher(X,Y) ->
  case catch X/Y of
    {'EXIT', {badarith,_}} -> "uh oh";
    N -> N
  end.
```

WAIT, THERE'S MORE!

- common manner using catch

```
7> exceptions:catcher(3,3).
1.0
8> exceptions:catcher(6,3).
2.0
9> exceptions:catcher(6,0).
"uh oh"
```

WAIT, THERE'S MORE!

- problem 1. parentheses when using with assign

```
10> X = catch 4+2.
* 1: syntax error before: 'catch'
10> X = (catch 4+2).
6
```

WAIT, THERE'S MORE!

- problem 2. difficult to distinct

```
11> catch erlang:boat().

12> catch exit({undef, [{erlang,boat,[]}, {erl_eval,do_apply,5},
{erl_eval,expr,5}, {shell,exprs,6}, {shell,eval_exprs,6}, {shell,eval_loop
,3}]}).

{'EXIT',{undef,[{erlang,boat,[]},
{erl_eval,do_apply,5},
{erl_eval,expr,5},
{shell,exprs,6},
{shell,eval_exprs,6},
{shell,eval_loop,3}]}}
```

WAIT, THERE'S MORE!

- problem 3. is it error or…?

```
one_or_two(1) -> return;
one_or_two(2) -> throw(return).
```

```
13> c(exceptions).
  {ok,exceptions}
14> catch exceptions:one_or_two(1).
  return
15> catch exceptions:one_or_two(2).
  return
```

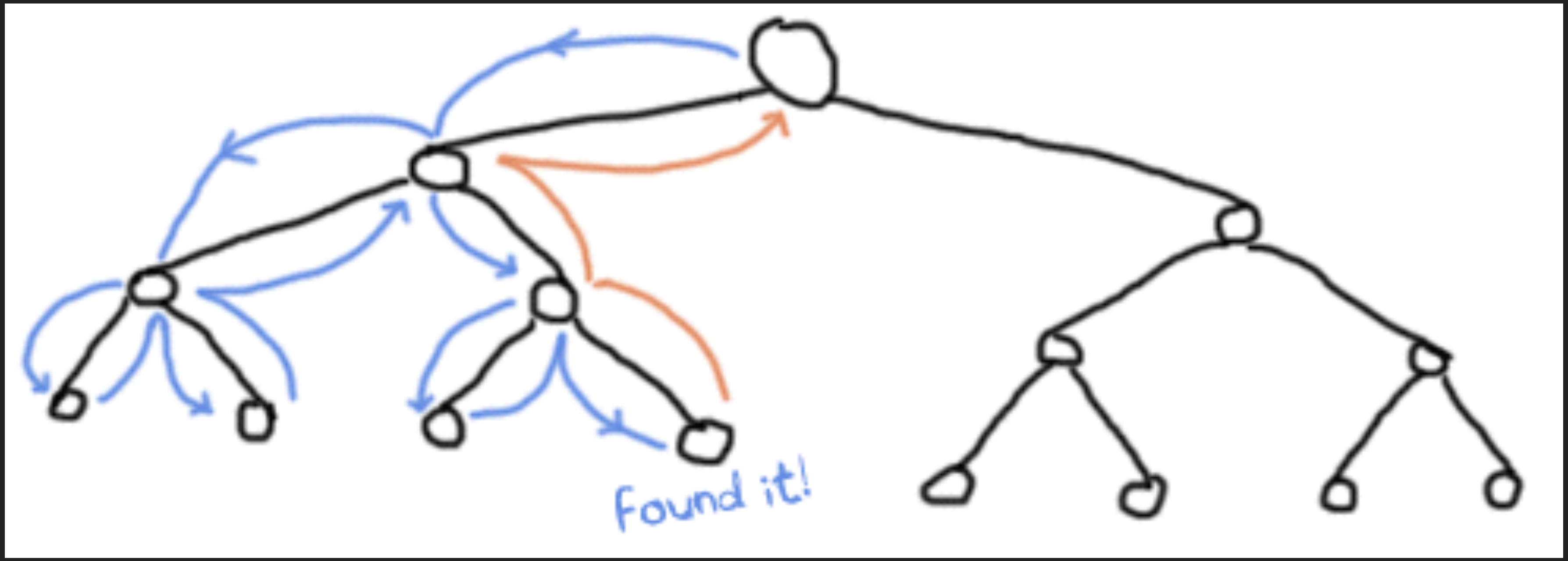
TRY A TRY IN A TREE

- lookup/2

```
%% looks for a given value 'Val' in the tree.
has_value(_, {node, 'nil'}) ->
  false;
has_value(Val, {node, {_, Val, _, _}}) ->
  true;
has_value(Val, {node, {_, _, Left, Right}}) ->
  case has_value(Val, Left) of
    true -> true;
    false -> has_value(Val, Right)
  end.
```

TRY A TRY IN A TREE

- lookup/2



TRY A TRY IN A TREE

- lookup/2 - less annoying

```
has_value(Val, Tree) ->
  try has_value1(Val, Tree) of
    false -> false
  catch
    true -> true
  end.
has_value1(_, {node, 'nil'}) -> false;
has_value1(Val, {node, {_, Val, _, _}}) -> throw(true);
has_value1(Val, {node, {_, _, Left, Right}}) ->
  has_value1(Val, Left),
  has_value1(Val, Right).
```

TRY A TRY IN A TREE

- lookup/2 - less annoying