WEEK-2 PRESENTED BY NOLLEH

# LEARN U ERLANG

# PATTERN MATCHING

- greet without patten matching

```
function greet(Gender,Name)
  if Gender == male then
    print("Hello, Mr. %s!", Name)
  else if Gender == female then
    print("Hello, Mrs. %s!", Name)
  else
    print("Hello, %s!", Name)
end
```

# PATTERN MATCHING

- greet with patten matching

```
greet(male, Name) ->
  io:format("Hello, Mr. ~s!", [Name]);
greet(female, Name) ->
  io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
  io:format("Hello, ~s!", [Name]).
```

# PATTERN MATCHING

- disusing / using

```
function(Args)
  if X then
    Expression
  else if Y then
    Expression
  else
    Expression
```

```
function(X) ->
  Expression;
function(Y) ->
  Expression;
function(_) ->
  Expression.
```

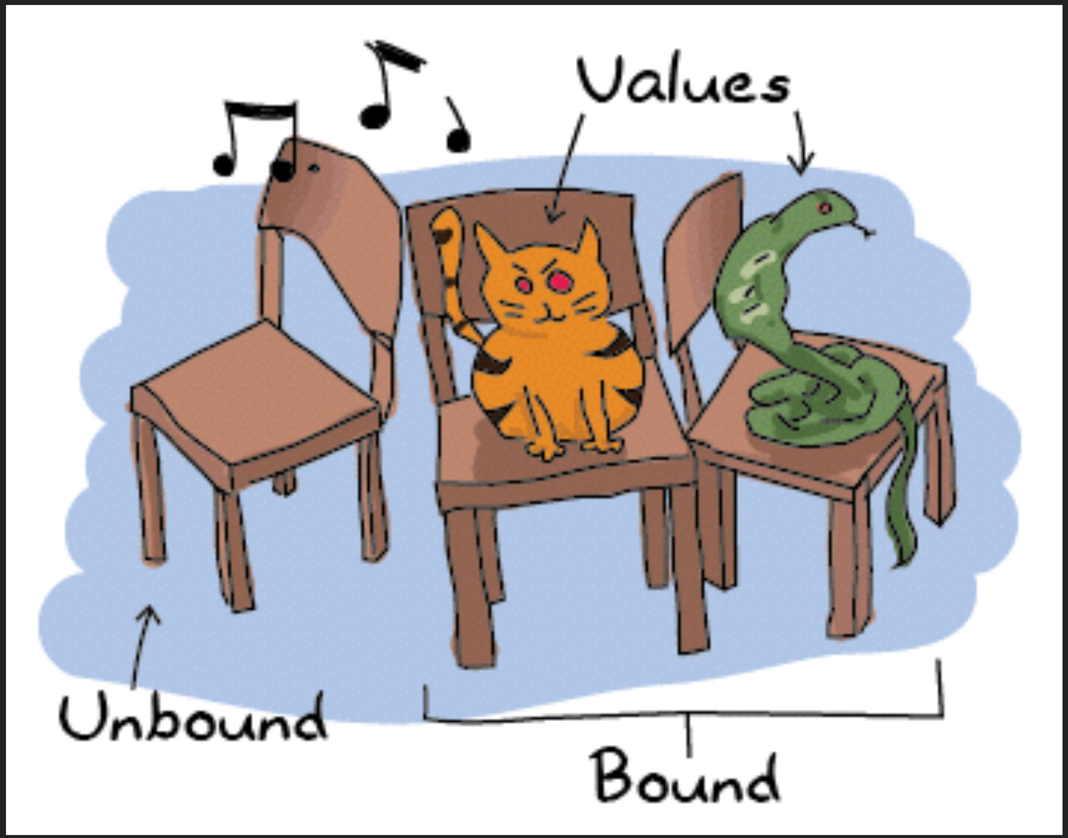
# PATTERN MATCHING

- example

```
head([H|_]) -> H.
second([_,X|_]) -> X.
same(X,X) ->
    true;
same(_,_) ->
    false.
```

# PATTERN MATCHING

- how's it works?
  - error occurs unless the new value is the same as the old one



# PATTERN MATCHING

- example2

```
valid_time({Date = {Y,M,D}, Time = {H,Min,S}}) ->
  io:format("The Date tuple (~p) says today is: ~p/~p/~p,~n”,
                  [Date,Y,M,D]),
  io:format("The time tuple (~p) indicates: ~p:~p:~p.~n”,
                  [Time,H,Min,S]);
valid_time(_) ->
  io:format("Stop feeding me wrong data!~n").
```

- functions head’s ‘=’ operator

# PATTERN MATCHING

- example2

```
4> c(functions).
  {ok, functions}
5> functions:valid_time({{2011,09,06},{09,04,43}}).
  The Date tuple ({2011,9,6}) says today is: 2011/9/6,
  The time tuple ({9,4,43}) indicates: 9:4:43.
  ok
6> functions:valid_time({{2011,09,06},{09,04}}).
  Stop feeding me wrong data!
```

- prob: It also recv just tuple!  and too precise sometimes.

# GUARDS,GUARDS!

- needs expressive way on sometimes…
  - range of value
  - not limited as certain types of data

# GUARDS,GUARDS!

- impractical vs practical

```
old_enough(0) -> false;
old_enough(1) -> false;
old_enough(2) -> false;
...
old_enough(14) -> false;
old_enough(15) -> false;
old_enough(_) -> true.
ok
```

```
old_enough(X) when X >= 16 -> true;
old_enough(_) -> false.
```

# GUARDS,GUARDS!

- simillar with andalso (little diff aspect on exceptions)

```
right_age(X) when X >= 16, X =< 104 ->
  true;
right_age(_) ->
  false.
```

- orelse

```
wrong_age(X) when X < 16; X > 104 ->
  true;
wrong_age(_) ->
  false.
```

# GUARDS,GUARDS!

- in guard, You will be
  - able to use functions like
    - A*B/C >= 0
    - is_integer/1, is_atom/1 ...
  - unable to use user defined function
    - because of side-effect

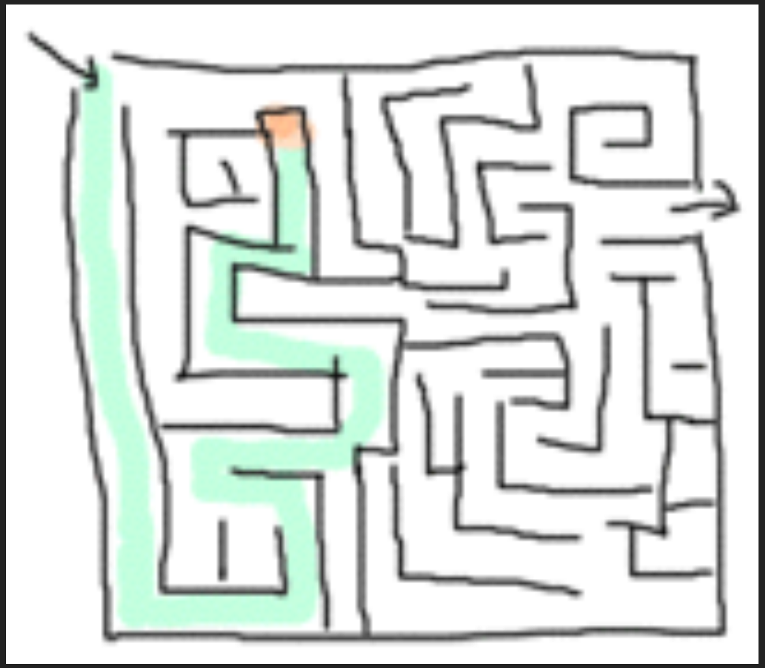
# WHAT THE IF!?

- similiar with guard but outside of function clauses head
- different from other language

# WHAT THE IF!?

```
-module(what_the_if).
-export([heh_fine/0]).

heh_fine() ->
  if 1 =:= 1 ->
    works
  end,
  if 1 =:= 2; 1 =:= 1 ->
    works
  end,
  if 1 =:= 2, 1 =:= 1 ->
    fails
  end.
```



# WHAT THE IF!?

```
1> c(what_the_if).
  ./what_the_if.erl:12: Warning: no clause will ever match
  ./what_the_if.erl:12: Warning: the guard for this clause evaluates to
  'false'
  {ok,what_the_if}
2> what_the_if:heh_fine().
  ** exception error: no true branch found when evaluating an if
  expression
  in function  what_the_if:heh_fine/0
```

# WHAT THE IF!?

- true branch

```
oh_god(N) ->
  if N =:= 2 -> might_succeed;
  true -> always_does  %% this is Erlang's if's 'else!'
end.
```

```
4> what_the_if:oh_god(2).
  might_succeed
5> what_the_if:oh_god(3).
  always_does
```

# WHAT IS IF!?

- why not else ?
  - both branch should be avoided
    - if is usually easier
- guard has only limited set

# IN CASE ... OF

- example

```
insert(X,[]) ->
  [X];
insert(X,Set) ->
  case lists:member(X,Set) of
    true  -> Set;
    false -> [X|Set]
  end.
```

# IN CASE ... OF

- pattern matching + guard

```
beach(Temperature) ->
  case Temperature of
    {celsius, N} when N >= 20, N =< 45 ->
      'favorable';
    {kelvin, N} when N >= 293, N =< 318 ->
      'scientifically favorable';
    {fahrenheit, N} when N >= 68, N =< 113 ->
      'favorable in the US’;
    _ ->
      'avoid beach'
  end.
```

# IN CASE ... OF

- instead, we can replace with bunch of functions.

```
beachf({celsius, N}) when N >= 20, N =< 45 ->
  'favorable';
  ...
beachf(_) ->
  'avoid beach'.
```

# WHICH TO USE?

- function call vs case of
  - same way at a lower level
  - only one difference when arg is more than one

```
case {A,B} of
  Pattern Guards -> ...
end.
```

# WHICH TO USE?

- function call vs case of
  - arguably cleaner

```
insert(X,[]) ->
  [X];
insert(X,Set) ->
  case lists:member(X,Set) of
    true  -> Set;
    false -> [X|Set]
  end.
```

# WHICH TO USE?

- if vs if through guard?
  - use where doesn't need whole pattern matching
- personal preference

# DYNAMITE-STRONG TYPING

- as you’ve seen, no need to type Type!
- elang is dynamically typed
    - compiler won't always yell at you
- statically typed language is safer..?
    - elang is reported as nine nine (99.999 % available)
- strongly typed

```
1> 6 + "1".
```

# DYNAMITE-STRONG TYPING

- type conversion

<type>_to_<type>

```
1> erlang:list_to_integer("54").
54
2> erlang:integer_to_list(54).
"54"
3> erlang:list_to_integer("54.32").
** exception error: bad argument in function  list_to_integer/1
called as list_to_integer("54.32")
```

# TYPE CONVERSION

- type conversion

```
<type>_to_<type>
```

```
4> erlang:list_to_float("54.32").
54.32
5> erlang:atom_to_list(true).
"true"
6> erlang:list_to_bitstring("hi there").
<<"hi there">>
7> erlang:bitstring_to_list(<<"hi there">>).
"hi there"
```

# TO GUARD A DATA TYPE

- type check

| | | |
|---|---|---|
| is_atom/1 | is_binary/1 | |
| is_bitstring/1 | is_boolean/1 | is_builtin/3 |
| is_float/1 | is_function/1 | is_function/2 |
| is_integer/1 | is_list/1 | is_number/1 |
| is_pid/1 | is_port/1 | is_record/2 |
| is_record/3 | is_reference/1 | is_tuple/1 |

- why not typeof
  - force user to make program that surely knowing the effect
- can used in guard expression

# FOR TYPE JUNKIES

- briefly describe tools used to do static type analysis
  - first try in 1997


- success type
  - will not exact type every expression
  - type it infers are right, type errors it finds are really error

# FOR TYPE JUNKIES

- success type

```
and(false, _) -> false;
and(_, false) -> false;
and(true,true) -> true.
```

```
and(_,_) -> bool()
```

# FOR TYPE JUNKIES

- if you interested

```
$ typer --help
```

```
$ dialyzer --help
```

# HELLO RECURSION!

- functional programming do not offer loop

# HELLO RECURSION!

- factorial

```
-module(recursive).
-export([fac/1]).

fac(N) when N == 0 -> 1;
fac(N) when N > 0  -> N*fac(N-1).
```

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n((n-1)!) & \text{if } n > 0 \end{cases}$$

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

# HELLO RECURSION!

- recursion
  - function that calls itself
  - need to have stopping condition (base case)

# LENGTH

- we need

a base case;
a function that calls itself;
a list to try our function on.

- simplest - empty list

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

# LENGTH

- list is recursively

```
[1 | [2| ... [n | []]]].
```

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

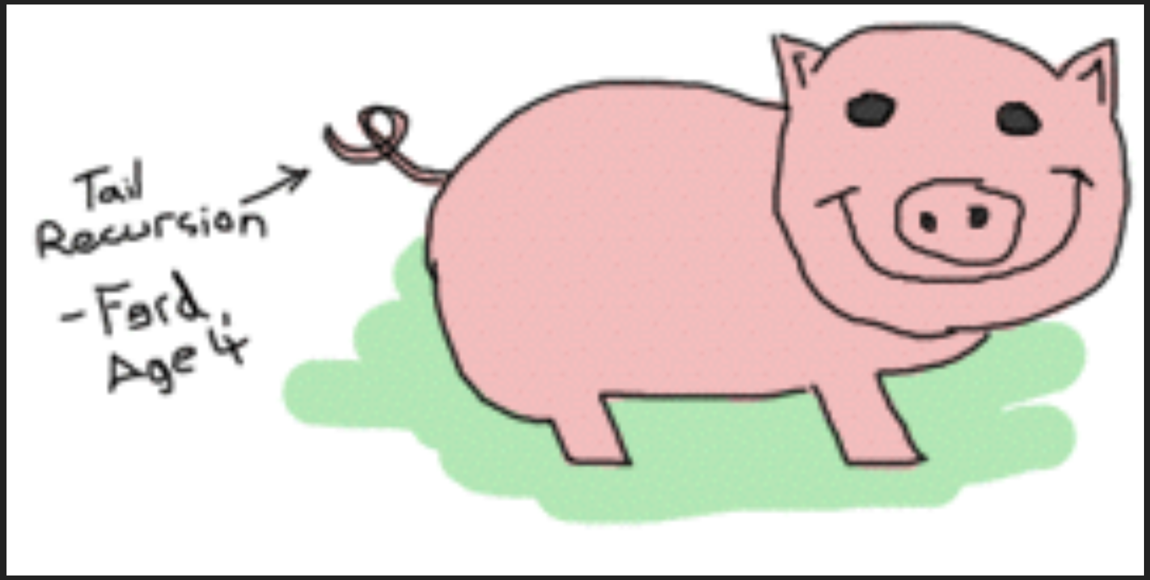
# LENGTH

- how it works?

```
len([1,2,3,4]) = len([1 | [2,3,4])
               = 1 + len([2 | [3,4]])
               = 1 + 1 + len([3 | [4]])
               = 1 + 1 + 1 + len([4 | []])
               = 1 + 1 + 1 + 1 + len([])
               = 1 + 1 + 1 + 1 + 0
               = 1 + 1 + 1 + 1
               = 1 + 1 + 2
               = 1 + 3
               = 4
```

# LENGTH OF A TAIL RECURSION

- it is problematic
  - keep millions of numbers in memory for such a simple calculation.
- let's tail recursion



# LENGTH OF A TAIL RECURSION

- linear -> iterative one
- need to be alone
- additional is stacked
- accumulator
  - need to extra variable to hold the intermediate result

# LENGTH OF A TAIL RECURSION

- using accumulator

```
tail_fac(N) -> tail_fac(N,1).

tail_fac(0,Acc) -> Acc;
tail_fac(N,Acc) when N > 0 -> tail_fac(N-1,N*Acc).
```

```
tail_fac(4)    = tail_fac(4,1)
tail_fac(4,1)  = tail_fac(4-1, 4*1)
tail_fac(3,4)  = tail_fac(3-1, 3*4)
tail_fac(2,12) = tail_fac(2-1, 2*12)
tail_fac(1,24) = tail_fac(1-1, 1*24)
tail_fac(0,24) = 24
```

# LENGTH OF A TAIL RECURSION

- length tail recursion

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

```
tail_len(L) -> tail_len(L,0).

tail_len([], Acc) -> Acc;
tail_len([_|T], Acc) -> tail_len(T,Acc+1).
```

# MORE RECURSIVE FUNCTIONS

- After all, recursion being the only looping construct that exists in Erlang
  - except list comprehension
- duplicate

```
duplicate(0,_) ->
   [];
duplicate(N,Term) when N > 0 ->
   [Term|duplicate(N-1,Term)].
```

# MORE RECURSIVE FUNCTIONS

- duplicate

```
tail_duplicate(N,Term) ->
tail_duplicate(N,Term,[]).

tail_duplicate(0,_,List) ->
  List;
tail_duplicate(N,Term,List) when N > 0 ->
  tail_duplicate(N-1, Term, [Term|List]).
```

```
function(N, Term) ->
  while N > 0 ->
    List = [Term|List],
    N = N-1
  end,
  List.
```

# MORE RECURSIVE FUNCTIONS

- true nightmare which is not tail recursion

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

```
reverse([1,2,3,4]) = [4]++[3]++[2]++[1]
                     = [4,3]++[2]++[1]
                     = [4,3,2]++[1]
                     = [4,3,2,1]
```

# MORE RECURSIVE FUNCTIONS

- let’s rescue

```
tail_reverse(L) -> tail_reverse(L,[]).
tail_reverse([],Acc) -> Acc;
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).
```

```
tail_reverse([1,2,3,4]) = tail_reverse([2,3,4], [1])
                        = tail_reverse([3,4], [2,1])
                        = tail_reverse([4], [3,2,1])
                        = tail_reverse([], [4,3,2,1])
                        = [4,3,2,1]
```

# MORE RECURSIVE FUNCTIONS

- sublist/2
  - a little different

```
sublist(_,0) -> [];
sublist([],_) -> [];
sublist([H|T],N) when N > 0 -> [H|sublist(T,N-1)].
```

```
tail_sublist(L, N) -> tail_sublist(L, N, []).

tail_sublist(_, 0, SubList) -> SubList;
tail_sublist([], _, SubList) -> SubList;
tail_sublist([H|T], N, SubList) when N > 0 ->
  tail_sublist(T, N-1, [H|SubList]).
```

# MORE RECURSIVE FUNCTIONS

- problems..

```
sublist([1,2,3,4,5,6],3)
```

- solve

```
tail_sublist(L, N) -> reverse(tail_sublist(L, N, [])).
```

# MORE RECURSIVE FUNCTIONS

- zip/2

```
1> recursive:zip([a,b,c],[1,2,3]).
[{a,1},{b,2},{c,3}]
```

- implements

```
zip([],[]) -> [];
zip([X|Xs],[Y|Ys]) -> [{X,Y}|zip(Xs,Ys)].
```

```
lenient_zip([],_) -> [];
lenient_zip(_,[]) -> [];
lenient_zip([X|Xs],[Y|Ys]) -> [{X,Y}|lenient_zip(Xs,Ys)].
```

# MORE RECURSIVE FUNCTIONS

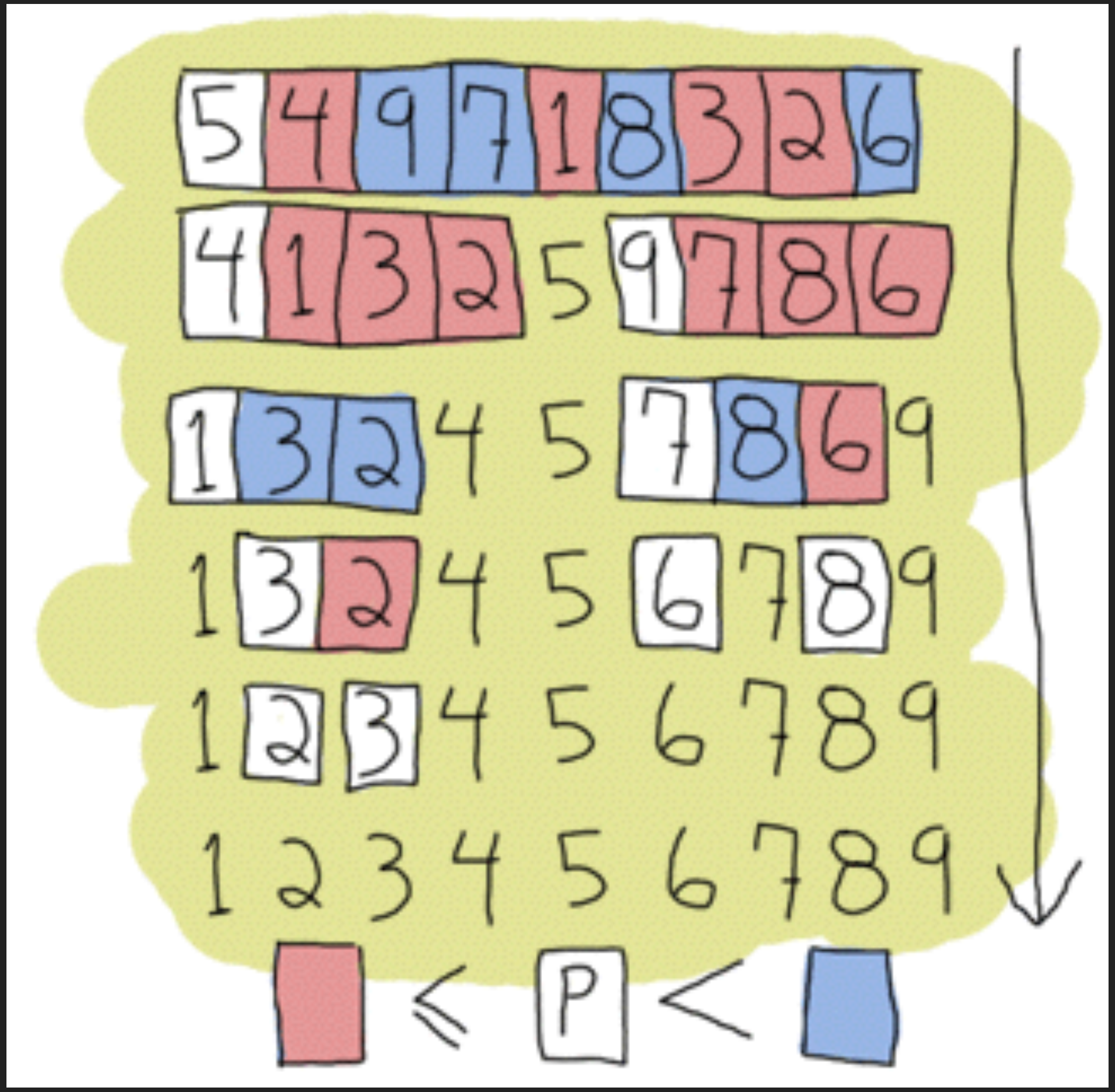
- TCO (tail call optimization)
  - vm does eliminate current stack frame
- LCO (last call optimization)
  - more general

# QUICK, SORT!

- naive version
  - pivot
  - smallers; equals | lagers
  - until empty list to sort

# QUICK, SORT!

- naive version



# QUICK, SORT!

- as two parts
  - partioning
  - apply the partitioning to each parts, and glue

```
quicksort([]) -> [];
quicksort([Pivot|Rest]) ->
  {Smaller, Larger} = partition(Pivot,Rest,[],[]),
  quicksort(Smaller) ++ [Pivot] ++ quicksort(Larger).
```

# QUICK, SORT!

- partitioning

```
partition(_,[], Smaller, Larger) -> {Smaller, Larger};
partition(Pivot, [H|T], Smaller, Larger) ->
  if H =< Pivot -> partition(Pivot, T, [H|Smaller], Larger);
    H >  Pivot -> partition(Pivot, T, Smaller, [H|Larger])
  end.
```

```
lc_quicksort([]) -> [];
lc_quicksort([Pivot|Rest]) ->
  lc_quicksort([Smaller || Smaller <- Rest, Smaller =< Pivot])
  ++ [Pivot] ++
  lc_quicksort([Larger || Larger <- Rest, Larger > Pivot]).
```

# MORE THAN LISTS

- tree
  - key / two other node (smaller, larger)
  - also able to contain empty node

# MORE THAN LISTS

- lets choice tuple!

```
{node, {Key, Value, Smaller, Larger}}
```

```
{node, nil}
```

- empty

```
-module(tree).
-export([empty/0, insert/3, lookup/2]).

empty() -> {node, 'nil'}.
```

# MORE THAN LISTS

- base case is empty node
  - where to put content
- compare, larger / smaller

# MORE THAN LISTS

- compare, larger / smaller

```
insert(Key, Val, {node, 'nil'}) ->
  {node, {Key, Val, {node, 'nil'}, {node, 'nil'}}};
insert(NewKey, NewVal, {node, {Key, Val, Smaller, Larger}})
  when NewKey < Key ->
    {node, {Key, Val, insert(NewKey, NewVal, Smaller), Larger}};
insert(NewKey, NewVal, {node, {Key, Val, Smaller, Larger}})
  when NewKey > Key ->
    {node, {Key, Val, Smaller, insert(NewKey, NewVal, Larger)}};
insert(Key, Val, {node, {Key, _, Smaller, Larger}}) ->
  {node, {Key, Val, Smaller, Larger}}.
```

# MORE THAN LISTS

- returns completely new tree
  - sometimes shared by vm
- look up

```
lookup(_, {node, 'nil'}) ->
  undefined;
lookup(Key, {node, {Key, Val, _, _}}) ->
  {ok, Val};
lookup(Key, {node, {NodeKey, _, Smaller, _}}) when Key < NodeKey ->
  lookup(Key, Smaller);
lookup(Key, {node, {_, _, _, Larger}}) -> lookup(Key, Larger).
```

# MORE THAN LISTS

- using example

```
1> T1 = tree:insert("Jim Woodland", "jim.woodland@gmail.com",
  tree:empty()).
2> T2 = tree:insert("Mark Anderson", "i.am.a@hotmail.com", T1).
3> Addresses = tree:insert("Anita Bath", “abath@someuni.edu",
 tree:insert("Kevin Robert", "myfairy@yahoo.com",
 tree:insert(“Wilson Longbrow", "longwil@gmail.com", T2))).
```

# MORE THAN LISTS

```
{node,{"Jim Woodland","jim.woodland@gmail.com",
    {node,{"Anita Bath","abath@someuni.edu",
        {node,nil},
        {node,nil}}},
    {node,{"Mark Anderson","i.am.a@hotmail.com",
        {node,{"Kevin Robert","myfairy@yahoo.com",
            {node,nil},
            {node,nil}}},
        {node,{"Wilson Longbrow","longwil@gmail.com",
            {node,nil},
            {node,nil}}}}}}}
```

# MORE THAN LISTS

- using example

```
4> tree:lookup("Anita Bath", Addresses).
{ok, "abath@someuni.edu"}
5> tree:lookup("Jacques Requin", Addresses).
undefined
```

# THINKING RECURSIVELY

- our approach is more declarative
- consise algorithm easy to understand
  - divide - and - conquer!
- you will learn how to abstract, next time.