Erlang programming language

Part 1

Kiyoshi Nitta
Yahoo Japan Research
<knitta@yahoo-corp.jp>

Literature

Joe Armstrong. 2010. Erlang. Commun. ACM 53, 9 (September 2010), 68-75. DOI=10.1145/1810891.1810910 http://doi.acm.org/10.1145/1810891.1810910

Joe Armstrong. 2013. *Programming Erlang: Software for a Concurrent World* (2nd edition). Pragmatic Bookshelf. http://pragprog.com/book/jaerlang2/programming-erlang

Objectives

- Get the overall image of Erlang.
- Become familiar with Erlang codes.

What is Erlang?

Functional language

- Pure functional language
  - All expressions return values after evaluations.
  - Functions are one of data that can be took as arguments and can be returned as values.
- No variables (like in imperative programming languages)
  - One variable can be bound once.
- No arrays
  - Array structures are not included in the core part of Erlang. They can be used from one of standard libraries.
- No types
  - No explicit type checking is performed.
  - However, some functions for identifying types of data can be used in guard part of function definitions.

Philosophy

- Shared nothing
  - The system would have to be constructed from physically isolated components communicating through well-defined “pure” protocols
- Erlang View of the World
  - Everything is a process that lacks shared memory and influences one another only by exchanging asynchronous messages.
- Erlang View of errors
  - Let failing processes crash and other processes detect the crashes and fix them.

Sequential programming

Terms

- Numbers
  - Integers, 24 bits (123, -34567, ...)
  - Floats, conventional representation
  - Examples: 12.345, -27.45e-05, 16#ffff, $A = 65
- Atoms
  - Constants with names
  - Begin with a lower-case letter (a..z) and are terminated by a non-alphanumeric character
- Examples:
  - friday unquoted_atoms_cannot_contain_blanks,
  - ’A quoted atom which contains several blanks’,
  - ’hello \n my friend’

Tuples

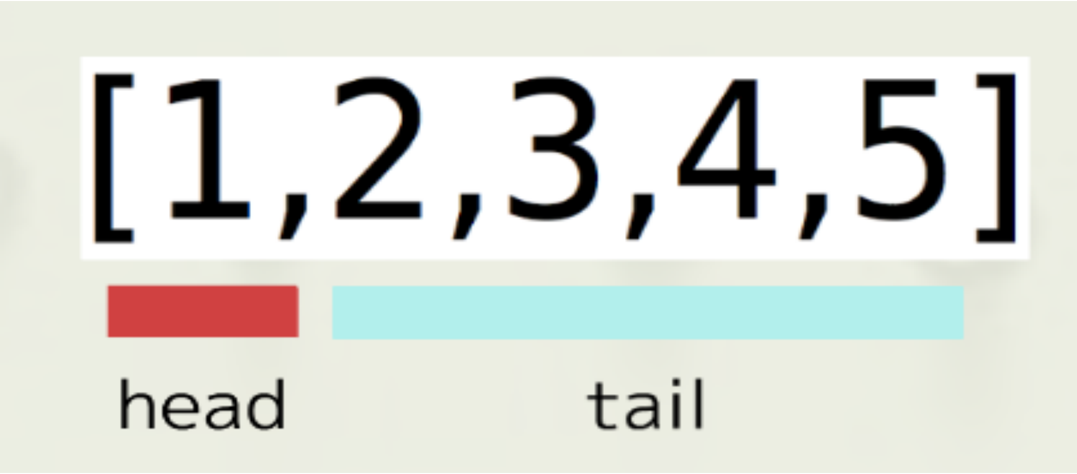
- Terms separated by commas and enclosed in curly brackets are called tuples.
- Tuple {E1,E2,...,En}, where $n \geq 0$, is said to have size n.
- Examples:
  - {a, 12, 'hello'}
  - {1, 2, {3, 4}, {a, {b, c}}}
  - {}

Lists

- Terms separated by commas and enclosed in square brackets are called lists.
- List [E1,E2,...,En], where $n \geq 0$, is said to have length n.
- Examples:
  - [1, abc, [12], ’foo bar’]
  - []
  - [a,b,c]
  - "abcd"

Lists

- "abc" = [97,98,99]
- head and tail of list



Lists

If T is a list, then [H|T] is also a list with head H and tail T. The vertical bar (|) separates the head of a list from its tail. [ ] is the empty list.

```
3> ThingsToBuy = [{apples,10},{pears,6},{milk,3}].
{apples,10},{pears,6},{milk,3}]
4> ThingsToBuy1 = [{oranges,4},{newspaper,1}|ThingsToBuy].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Lists

If we have the nonempty list L, then the expression [X|Y] = L, where X and Y are unbound variables, will extract the head of the list into X and the tail of the list into Y.

```
5> [Buy1|ThingsToBuy2] = ThingsToBuy1.
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
6> [Buy2,Buy3|ThingsToBuy3] = ThingsToBuy2.
[{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Pattern matching

- Patterns have the same structure as terms, with the addition that they can include variables.
- Variables start with an upper-case letter.
- Examples:

```
{A, a, 12, [12,34|{a}]}
{A, B, 23}
{x, {X_1}, 12, My_cats_age}
[]
```

- A, B, X_1, and My_cats_age are variables in Erlang.

Pattern matching

- Pattern matching provides the basic mechanism by which values become assigned to variables.
  - bound / unbound variables
  - assigning a value to a variable is called binding
- Pattern matches with term:
  - They are structurally isomorphic
  - Whenever an atomic data type is encountered in the pattern, the same atomic data type is encountered at the same position in the corresponding term
  - If pattern contains an unbound variable, the variable is bound to the corresponding element in the term

Pattern matching

Pattern	=	*Term*	*Result*
{X,abc}	=	{123,abc}	*Succeeds* with X = 123
{X,Y,Z}	=	{222,def,"cat"}	*Succeeds* with X = 222, Y = def, and Z = "cat"
{X,Y}	=	{333,ghi,"cat"}	*Fails*—the tuples have different shapes
X	=	true	*Succeeds* with X = true
{X,Y,X}	=	{{abc,12},42,{abc,12}}	*Succeeds* with X = {abc,12} and Y = 42
{X,Y,X}	=	{{abc,12},42,true}	*Fails*—X cannot be both {abc,12} and true
[H|T]	=	[1,2,3,4,5]	*Succeeds* with H = 1 and T = [2,3,4,5]
[H|T]	=	"cat"	*Succeeds* with H = 99 and T = "at"
[A,B,C|T]	=	[a,b,c,d,e,f]	*Succeeds* with A = a, B = b, C = c, and T = [d,e,f]

Modules

- Erlang has a module system which allows us to divide a large program into a set of modules.
  - Each module has its own name space
  - Import or export declaration in the module

```
-module(lists1).
-export([reverse/1]).

reverse(L) ->
        reverse(L, []).

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

Modules

Calling functions in other modules

```
-module(sort1).
-export([reverse_sort/1, sort/1]).

reverse_sort(L) ->
        lists1:reverse(sort(L)).

sort(L) ->
        lists:sort(L).
```

Modules

Terminology:

- module definition
- attributes
- blank lines
- comments
- exported function
- local function

```
-module(lists2).

-export([flat_length/1]).

%% flat_length(List)
%%  Calculate the length of a list of lists.

flat_length(List) ->
    flat_length(List, 0).

flat_length([H|T], N) when list(H) ->
    flat_length(H, flat_length(T, N));
flat_length([H|T], N) ->
    flat_length(T, N + 1);
flat_length([], N) ->
    N.
```

Functions

- Syntax and semantics of Erlang functions
  - Structure of clauses:
  - Head of clause
  - Clause guards
  - Guard tests
  - Body of clause

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

Functions

Clause guards:

```
foo(X, Y, Z) when integer(X), integer(Y), integer(Z), X == Y + Z ->
foo(X, Y, Z) when list(X), hd(X) == {Y, length(Z)}  ->
foo(X, Y, Z) when {X, Y, size(Z)} == {a, 12, X} ->
foo(X) when list(X), hd(X) == c1, hd(tl(X)) == c2 ->
```

Guard	Succeeds if
`atom(X)`	X is an atom
`constant(X)`	X is not a list or tuple
`float(X)`	X is a float
`integer(X)`	X is an integer
`list(X)`	X is a list or `[]`
`number(X)`	X is an integer or float
`pid(X)`	X is a process identifier
`port(X)`	X is a port
`reference(X)`	X is a reference
`tuple(X)`	X is a tuple
`binary(X)`	X is a binary

Operator	Description	Type
`X > Y`	X greater than Y	coerce
`X < Y`	X less than Y	coerce
`X =< Y`	X equal to or less than Y	coerce
`X >= Y`	X greater than or equal to Y	coerce
`X == Y`	X equal to Y	coerce
`X /= Y`	X not equal to Y	coerce
`X =:= Y`	X equal to Y	exact
`X =/= Y`	X not equal to Y	exact

Functions

- Clause body:
  - Consists of a sequence of one or more expressions which are separated by commas.
  - Last expression is value of body (function).

```
factorial(N) when N > 0 ->
    N1 = N - 1,
    F1 = factorial(N1),
    N * F1.
```

Examples

```
factorial(0) -> 1;
factorial(N) -> N * factorial(N - 1).
```

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

```
factorial(N) ->
    if
        N == 0 -> 1;
        N >  0 -> N * factorial(N - 1)
    end.
```

```
factorial(N) ->
    case N of
        0 -> 1;
        N when  N > 0 ->
            N * factorial(N - 1)
    end.
```

```
factorial(0) ->
    1;
factorial(N) when N > 0 ->
    N1 = N - 1,
    F1 = factorial(N1),
    N * F1.
```

Programming with Lists

Programming with Lists
List Processing BIFs

Several built-in functions are available for conversion between lists and other data types.

`atom_to_list(A)`
: Converts the atom `A` to a list of ASCII character codes.
  Example: `atom_to_list(hello)` $\Longrightarrow$ `[104,101,108,108,111]`.[1]

`float_to_list(F)`
: Converts the floating point number `F` to a list of ASCII characters.
  Example: `float_to_list(1.5)` $\Longrightarrow$ `[49,46,53,48,48,...,48]`.

`integer_to_list(I)`
: Converts the integer `I` to a list of ASCII characters.
  Example: `integer_to_list(1245)` $\Longrightarrow$ `[49,50,52,53]`.

`list_to_atom(L)`
: Converts the list of ASCII characters in `L` to an atom.
  Example: `list_to_atom([119,111,114,108,100])` $\Longrightarrow$ `world`.

`list_to_float(L)`
: Converts the list of ASCII characters in `L` to a floating point number.
  Example: `list_to_float([51,46,49,52,49,53,57])` $\Longrightarrow$ `3.14159`.

`list_to_integer(L)`
: Converts the list of ASCII characters in `L` to an integer.
  Example: `list_to_integer([49,50,51,52])` $\Longrightarrow$ `1234`.

`hd(L)`
: Returns the first element in the list `L`.
  Example: `hd([a,b,c,d])` $\Longrightarrow$ `a`.

`tl(L)`
: Returns the tail of the list `L`
  Example: `tl([a,b,c,d])` $\Longrightarrow$ `[b,c,d]`.

`length(L)`
: Returns the length of the list `L`
  Example: `length([a,b,c,d])` $\Longrightarrow$ `4`.

Programming with Lists
Some Common List Processing Functions

member(X, L) returns true if X is an element of the list L, otherwise false.

```
member(X, [X|_]) -> true;
member(X, [_|T]) -> member(X, T);
member(X, [])    -> false.
```

```
> lists:member(a,[1,2,a,b,c]).
(0)lists:member(a,[1,2,a,b,c])
(1).lists:member(a, [2,a,b,c])
(2)..lists:member(a,[a,b,c])
(2)..true
(1).true
(0)true
true
> lists:member(a,[1,2,3,4]).
(0)lists:member(a, [1,2,3,4])
(1).lists:member(a, [2,3,4])
(2)..lists:member(a, [3,4])
(3)...lists:member(a, [4])
(4)....lists:member(a, [])
(4)....false
(3)...false
(2)..false
(1).false
(0)false
false
```

Programming with Lists
Some Common List Processing Functions

append(A,B) concatenates the two lists A and B.

```
append([H|L1], L2) -> [H|append(L1, L2)];
append([], L) -> L.
```

```
append([a,b,c], [d,e,f])

[a | append([b,c], [d,e,f])]
```

```
> lists:append([a,b,c],[d,e,f]).
(0)lists:append([a,b,c],[d,e,f])
(1).lists:append([b,c], [d,e,f])
(2)..lists:append([c],[d,e,f])
(3)...lists:append([], [d,e,f])
(3)...[d,e,f]
(2)..[c,d,e,f]
(1).[b,c,d,e,f]
(0)[a,b,c,d,e,f]
[a,b,c,d,e,f]
```

Programming with Lists
Some Common List Processing Functions

reverse(L) reverses the order of the elements in the list L.

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

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

```
> lists:reverse([a,b,c,d]).
(0)lists:reverse([a,b,c,d])
(1).lists:reverse([a,b,c,d], [])
(2)..lists:reverse([b,c,d], [a])
(3)...lists:reverse([c,d], [b,a])
(4)....lists:reverse([d], [c,b,a])
(5).....lists:reverse([], [d,c,b,a])
(5).....[d,c,b,a]
(4)....[d,c,b,a]
(3)...[d,c,b,a]
(2)..[d,c,b,a]
(1).[d,c,b,a]
(0)[d,c,b,a]
[d,c,b,a]
```

Programming with Lists
Examples

sort(X) returns a sorted list of the elements of the list X.

```
-module(sort).
-export([sort/1]).

sort([]) -> [];
sort([Pivot|Rest]) ->
    {Smaller, Bigger} = split(Pivot, Rest),
    lists:append(sort(Smaller), [Pivot|sort(Bigger)]).

split(Pivot, L) ->
    split(Pivot, L, [], []).

split(Pivot, [], Smaller, Bigger) ->
    {Smaller,Bigger};
split(Pivot, [H|T], Smaller, Bigger) when H < Pivot ->
    split(Pivot, T, [H|Smaller], Bigger);
split(Pivot, [H|T], Smaller, Bigger) when H >= Pivot ->
    split(Pivot, T, Smaller, [H|Bigger]).

 > lists:split(7,[2,1,4,23,6,8,43,9,3]).
{[3,6,4,1,2],[9,43,8,23]}

 > append([1,2,3,4,6], [7 | [8,9,23,43]]).
 [1,2,3,4,6,7,8,9,23,43]
```

Programming with Lists
Examples

qsort(X) returns a sorted list of the elements of the list X.

```
qsort(X) ->
        qsort(X, []).

%% qsort(A,B)
%%   Inputs:
%%      A = unsorted List
%%      B = sorted list where all elements in B
%%          are greater than any element in A
%%   Returns
%%      sort(A) appended to B

qsort([Pivot|Rest], Tail) ->
    {Smaller,Bigger} = split(Pivot, Rest),
    qsort(Smaller, [Pivot|qsort(Bigger,Tail)]);
qsort([], Tail) ->
    Tail.
```

Funs: The Basic Unit of Abstraction

- Functions that manipulate functions are called *higher-order functions*, and the data type that represents a function in Erlang is called a *fun*.
- funs are "anonymous" functions. They are called this because they have no name. You might see them referred to as *lambda abstractions* in other programming languages.

```
1> Double = fun(X) -> 2*X end.
#Fun<erl_eval.6.56006484>
2> Double(2).
4
```

Funs: The Basic Unit of Abstraction

```
3> Hypot = fun(X, Y) -> math:sqrt(X*X + Y*Y) end.
#Fun<erl_eval.12.115169474>
4> Hypot(3,4).
5.0

5> Hypot(3).
** exception error: interpreted function with arity 2 called with one argument
```

```
6> TempConvert = fun({c,C}) -> {f, 32 + C*9/5};
6>                  ({f,F}) -> {c, (F-32)*5/9}
6>               end.
#Fun<erl_eval.6.56006484>
7> TempConvert({c,100}).
{f,212.0}
8> TempConvert({f,212}).
{c,100.0}
9> TempConvert({c,0}).
{f,32.0}
```

List Comprehensions

List comprehensions are expressions that create lists without having to use funs, maps, or filters.

```
1> L = [1,2,3,4,5].
[1,2,3,4,5]

2> lists:map(fun(X) -> 2*X end, L).
[2,4,6,8,10]
```

```
4> [2*X || X <- L ].
[2,4,6,8,10]
```

```
1> Buy=[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
2> [{Name, 2*Number} || {Name, Number} <- Buy].
[{oranges,8},{newspaper,2},{apples,20},{pears,12},{milk,6}]
```

Note that the generator part of a list comprehension works like a filter.

```
1> [ X || {a, X} <- [{a,1},{b,2},{c,3},{a,4},hello,"wow"]].
[1,4]
```

Quicksort

lib_misc.erl

```
qsort([]) -> [];
qsort([Pivot|T]) ->
        qsort([X || X <- T, X < Pivot])
        ++ [Pivot] ++
        qsort([X || X <- T, X >= Pivot]).
```

```
1> L=[23,6,2,9,27,400,78,45,61,82,14].
[23,6,2,9,27,400,78,45,61,82,14]
2> lib_misc:qsort(L).
[2,6,9,14,23,27,45,61,78,82,400]

3> [Pivot|T] = L.
[23,6,2,9,27,400,78,45,61,82,14]
```

```
4> Smaller = [X || X <- T, X < Pivot].
[6,2,9,14]
5> Bigger  = [X || X <- T, X >= Pivot].
[27,400,78,45,61,82]
```

```
qsort( [6,2,9,14] ) ++ [23] ++ qsort( [27,400,78,45,61,82] )
= [2,6,9,14] ++ [23] ++ [27,45,61,78,82,400]
= [2,6,9,14,23,27,45,61,78,82,400]
```

Programming with Tuples

Unbalanced Binary Trees

- Internal nodes of the tree are represented by {Key,Value,Smaller,Bigger}.
  - Value is the value of some object which has been stored at some node in the tree with key Key.
  - Smaller is a subtree where all the keys at the nodes in the tree are smaller than Key, and
  - Bigger is a subtree where all the keys at the nodes in the tree are greater than or equal to Key.
  - Leaves in the tree are represented by the atom nil.

Unbalanced Binary Trees

Function lookup(Key,Tree) searches Tree to see if an entry associated with Key has been stored in the tree.

```
lookup(Key, nil) ->
    not_found;
lookup(Key, {Key,Value,_,_}) ->
    {found,Value};
lookup(Key, {Key1,_,Smaller,_}) when Key < Key1 ->
    lookup(Key, Smaller);
lookup(Key, {Key1,_,_,Bigger}) when Key > Key1 ->
    lookup(Key, Bigger).
```

Unbalanced Binary Trees

Function insert(Key,Value,OldTree) inserts new data into the tree. It returns a new tree.

```
insert(Key, Value, nil) ->
    {Key,Value,nil,nil};
insert(Key, Value, {Key,_,Smaller,Bigger}) ->
    {Key,Value,Smaller,Bigger};
insert(Key, Value, {Key1,V,Smaller,Bigger}) when Key < Key1 ->
    {Key1,V,insert(Key, Value, Smaller),Bigger};
insert(Key, Value, {Key1,V,Smaller,Bigger}) when Key > Key1 ->
    {Key1,V,Smaller,insert(Key, Value, Bigger)}.
```

Unbalanced Binary Trees

Function write_tree(Tree) displays it in a way which reflects its structure.

```
write_tree(T) ->
    write_tree(0, T).

write_tree(D, nil) ->
    io:tab(D),
    io:format('nil', []);
write_tree(D, {Key,Value,Smaller,Bigger}) ->
    D1 = D + 4,
    write_tree(D1, Bigger),
    io:format('~n', []),
    io:tab(D),
    io:format('~w ===> ~w~n', [Key,Value]),
    write_tree(D1, Smaller).
```

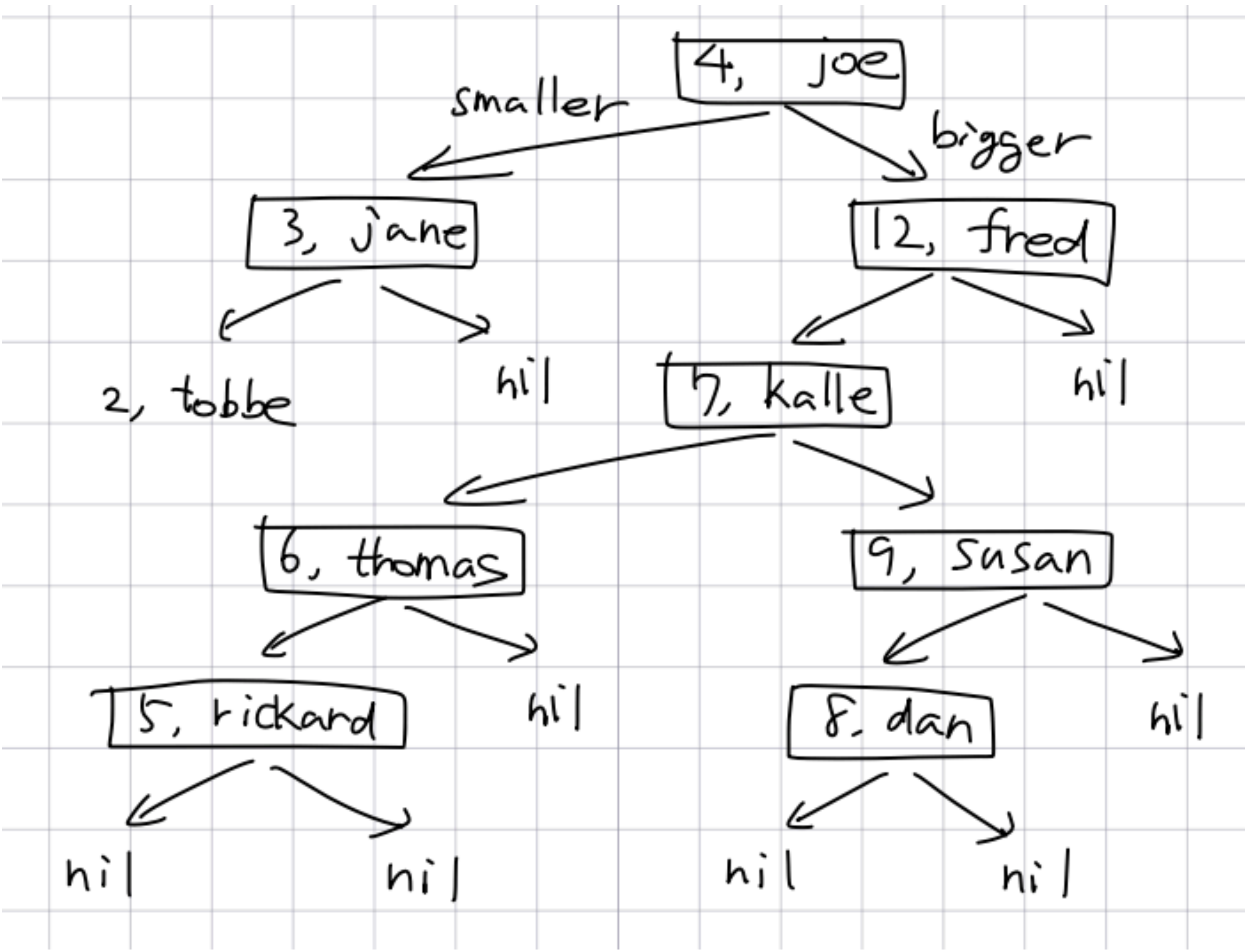
Unbalanced Binary Trees

Function test1() insert data into a tree and print it .

```
test1() ->
    S1 = nil,
    S2 = insert(4,joe,S1),
    S3 = insert(12,fred,S2),
    S4 = insert(3,jane,S3),
    S5 = insert(7,kalle,S4),
    S6 = insert(6,thomas,S5),
    S7 = insert(5,rickard,S6),
    S8 = insert(9,susan,S7),
    S9 = insert(2,tobbe,S8),
    S10 = insert(8,dan,S9),
    write_tree(S10).
```

```
        nil
    12 ===> fred
                nil
            9 ===> susan
                    nil
                8 ===> dan
                    nil
        7 ===> kalle
                nil
            6 ===> thomas
                    nil
                5 ===> rickard
                    nil
4 ===> joe
        nil
    3 ===> jane
            nil
        2 ===> tobbe
            nil
```

Unbalanced Binary Trees



Unbalanced Binary Trees

Function delete(Key, Value) deletes elements from a binary tree.

```
1: delete(Key, nil) ->
       nil;
2: delete(Key, {Key,_,nil,nil}) ->
       nil;
3: delete(Key, {Key,_,Smaller,nil}) ->
       Smaller;
4: delete(Key, {Key,_,nil,Bigger}) ->
       Bigger;
5: delete(Key, {Key1,_,Smaller,Bigger}) when Key == Key1 ->
       {K2,V2,Smaller2} = deletesp(Smaller),
       {K2,V2,Smaller2,Bigger};
6: delete(Key, {Key1,V,Smaller,Bigger}) when Key < Key1 ->
       {Key1,V,delete(Key, Smaller),Bigger};
7: delete(Key, {Key1,V,Smaller,Bigger}) when Key > Key1 ->
       {Key1,V,Smaller,delete(Key, Bigger)}.
```

Unbalanced Binary Trees

In clause 5 the node to be deleted has been located, but this node is an internal node in the tree (i.e. the node has both a Smaller and Bigger subtree. In this case the node having the largest key in the Smaller subtree is located and the tree rebuilt from this node

```
deletesp({Key,Value,nil,nil}) ->
    {Key,Value,nil};
deletesp({Key,Value,Smaller,nil}) ->
    {Key,Value,Smaller};
deletesp({Key,Value,Smaller,Bigger}) ->
    {K2,V2,Bigger2} = deletesp(Bigger),
    {K2,V2,{Key,Value,Smaller,Bigger2}}.
```

Concurrent Programming (Basics)

Process Creation

- The BIF spawn/3 creates and starts the execution of a new process.

```
Pid = spawn(Module, FunctionName, ArgumentList)
```

- The call to spawn/3 returns immediately when the new process has been created and does not wait for the given function to evaluate.
- A process will automatically terminate when the evaluation of the function given in the call to spawn has been completed.

Inter-process Communication

- In Erlang the only form of communication between processes is by message passing.

```
Pid ! Message
```

- Sending a message is an asynchronous operation so the send call will not wait for the message either to arrive at the destination or to be received.
- Messages are always delivered to the recipient, and always delivered in the same order they were sent.

Inter-process Communication

- The primitive receive is used to receive messages.

```
receive
    Message1 [when Guard1] ->
        Actions1 ;
    Message2 [when Guard2] ->
        Actions2 ;
    ...
end
```

- Each process has a mailbox and all messages which are sent to the process are stored in the mailbox in the same order as they arrive.
- Message1 and Message2 are patterns which are matched against messages that are in the process's mailbox.
- The process evaluating receive will be suspended until a message is matched.

Inter-process Communication

- Example: Here is a module which creates processes containing counters which can be incremented.

```
-module(counter).
-export([start/0,loop/1]).

start() ->
    spawn(counter, loop, [0]).

loop(Val) ->
    receive
        increment ->
            loop(Val + 1)
    end.
```

start a new counter process

a perpetual process which is suspended when waiting for input

selective message reception

loop is a tail recursive function

Inter-process Communication

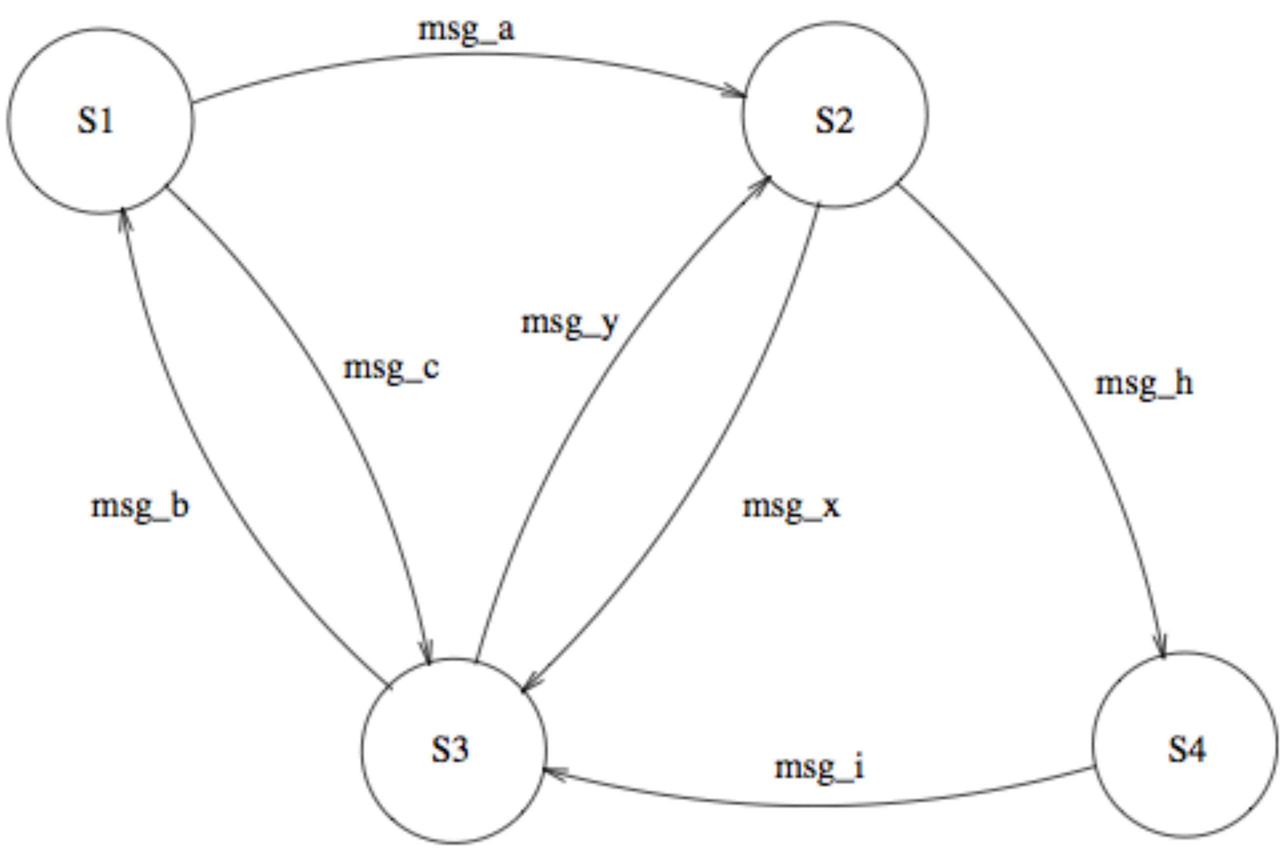
- Example: Here is an improved module counter which allows us to increment counters, access their values and also stop them.

```
-module(counter).
-export([start/0,loop/1,increment/1,value/1,stop/1]).

%% First the interface functions.
start() ->
    spawn(counter, loop, [0]).

increment(Counter) ->
    Counter ! increment.

value(Counter) ->
    Counter ! {self(),value},
    receive
        {Counter,Value} ->
            Value
    end.
```

```
%% The counter loop.
loop(Val) ->
    receive
        increment ->
            loop(Val + 1);
        {From,value} ->
            From ! {self(),Val},
            loop(Val);
        stop ->                   % No recursive call here
            true;
        Other ->                  % All other messages
            loop(Val)
    end.
```

Inter-process Communication

- Example: finite state machine (FSM).



Inter-process Communication

- Example: finite state machine (FSM).

```
s1() ->
    receive
        msg_a ->
            s2();
        msg_c ->
            s3()
    end.

s2() ->
    receive
        msg_x ->
            s3();
        msg_h ->
            s4()
    end.

s3() ->
    receive
        msg_b ->
            s1();
        msg_y ->
            s2()
    end.

s4() ->
    receive
        msg_i ->
            s3()
    end.
```

Objectives were…

- Get the overall image of Erlang.
- Become familiar with Erlang codes.

The topics of the next lecture will be

- Understanding the concept of concurrency.
- Writing concurrent codes in Erlang.
- Developing concurrent and robust applications in Erlang using OTP framework.

Thank you for your attention!