

# Erlang

Спецкурс по функциональному языку  
программирования erlang.



# Функциональность

Erlang – язык с функциональной парадигмой. Базовым объектом такого языка является функция.

Erlang был целенаправленно разработан для применения в распределённых, отказоустойчивых, параллельных системах реального времени.

Поддерживает горячую замену кода.

# Оболочка. Базовые понятия

## Выход из оболочки.

Ctrl-C (unix) Ctrl-Break (win)  
q().

## Арифметические операции

$2 + 2.$   
 $(2 + 3) * 4.$   
 $2 + 3 * 4.$

## Целочисленная арифметика

1234567890 \*  
987654321 \*  
1122334455667788.

Термы – часть данных любого типа.

% – однострочный комментарий

## Действительные числа

$5/3$   
 $2 * 1.5$

# Сопоставление по образцу (Pattern matching)

Переменные не изменяются.  $X = 6.$

Связанные и несвязанные.  $X = Y.$

Область видимости, внутри  $X = Z.$

функции, нет глобальных  
или приватных `** exception error: no match of  
right hand side value 7`

Сопоставление по образцу  
Переменная – это ссылка на ее  
значение.

$\% L = R.$

$X = (2+4).$

$Y = 6.$

$Z = 7.$

# Атомы, кортежи

Атомы (atoms, константы),  
должны начинаться с  
прописной буквы

monday, tuesday

red, december, cat,  
joe@somehost, a\_long\_name

'Monday', 'Tuesday', '+',  
'\*', 'an atom with spaces'

Кортежи (tuples)

{2,3}

{blue,255}

FirstName = {first\_name,  
ivan}.

Person =  
{sidorov,FirstName,  
{age,34}}.

element(1, {a, b}) % == a

Анонимная переменная

{\_,{\_,Name},\_} = Person.

# Списки

`[2, 3, 5, 7, 11, 13, 17]`

`[2, atom, 2.5]`

Голова и хвост списка

`[1 | [2, 3, 4, 5]]`

Выделение элементов из списка

`[E1,E2|T] = [1,2,3,4,5]`

Библиотека `lists`

`lists:nth(1, [1, 2])`

Строка – список

коды символов Latin-1  
(ISO-8859-1)

`[83,117,114,112,114,105,115,101]`.

Долларовый синтаксис

`I = $s. %` целое число,  
которое представляет `s`

# Модули, функции

## Модуль

`-module(some_module)`

## Функции

`имя(арг1, арг2, ...) ->`  
`тело.`

## Клаузы (clause)

`area({rectangle, Width, Ht}) ->`  
`Width * Ht;`

`area({circle, R}) ->`  
`3.14159 * R * R.`

## Компиляция модуля

`1> c(geometry).`

## Вызов функции

`2> geometry:area({rectangle,`  
`10, 5}).`

`3> geometry:area({circle,`  
`1.4}).`

# Арность, лямбда-функции, функции высших порядков

## Арность

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

## Функции высших порядков

```
lists:map(Fun, List).  
lists:map(Z, [1, 2, 3]).  
lists:filter(  
fun(X) -> (X rem 2) == 0 end,  
[1, 2, 3, 4, 5, 6]).
```

## Лямбда

```
Z = fun(X) -> 2*X end.  
Z(2).
```

## List Comprehensions

```
[X || X <- [1,2,a,3,4,b,5,6], X >  
3].  
[{X, Y} || X <- [1,2,3], Y <-  
[a,b]].  
[{1,a},{1,b},{2,a},{2,b},{3,a},  
{3,b}]
```



# Фабрики функций

```
MakeTest = fun(L) -> (fun(X) -> lists:member(X, L) end) end.  
IsFruit = MakeTest([apple,pear,orange]).  
lists:filter(IsFruit, [dog,orange,cat,apple,bear]).
```

# Guard expressions (охранннные выражения)

```
function(arg1, arg2, ...)  
  when guard11, guard12, ... ;  
      guard21, guard22, ... ->  
    ... .
```

Пример

```
f(X,Y) when  
  is_integer(X), X > Y; abs(Y) < 23  
  -> value.
```

Следующие синтаксические формы языка Эрланг допустимы в `guard` выражениях:

- атом `true` (истина), остальные атомы заменяются на `false`
- вызовы контрольных предикатов и встроенные функции (BIF)
- арифметические выражения
- логические выражения
- сравнения термов
- упрощенные (или укороченные) логические выражения

# Операция сравнения

Сравнение термов одного типа.

|            |                                       |
|------------|---------------------------------------|
| $X > Y$    | $X$ больше, чем $Y$                   |
| $X < Y$    | $X$ меньше, чем $Y$                   |
| $X \leq Y$ | $X$ равен, либо меньше, чем $Y$       |
| $X \geq Y$ | $X$ равен, либо больше, чем $Y$       |
| $X == Y$   | $X$ равен $Y$ (с приведением типа)    |
| $X /= Y$   | $X$ не равен $Y$ (с приведением типа) |
| $X ::= Y$  | $X$ идентичен $Y$                     |
| $X \neq Y$ | $X$ не идентичен $Y$                  |

Сравнения термов разных типов

`number < atom < reference < fun < port < pid < tuple < list < binary`

# Debug output

```
debug(Element) ->  
    io:format("~w~n", [elem1, elem2, ...]).
```

~n – Перевод строки.

~p – Структурная распечатка аргумента

~s – Аргумент является строкой

~f – Аргумент является вещественным число

```
% io:format("~.15f~n", [math:pi()]).
```

~w – Вывод данных со стандартным синтаксисом. Используется для вывода термов Эрланга

...

# Упрощенные логические операторы

`andalso(false, _) → false;    orelse(true, _) → true;`  
`andalso(true, E2) → E2.        orelse(false, E2) → E2.`

`Expr1 orelse Expr2`

`Expr1 andalso Expr2`

`A >= -1.0 andalso A+1 > B`

`is_atom(L) orelse (is_list(L) andalso length(L) > 2)`

# Record (записи)

% Записи - кортежи с именованными полями

```
-record(name, {  
    %% the next two keys have default values  
    key1 = Default1,  
    key2 = Default2,  
    ...  
    %% The next line is equivalent to  
    %% key3 = undefined  
    key3,  
    ...  
}).
```

Можно объявлять только в файле.

Подключение.

Из консоли:

```
> rr("records.hrl").
```

В файле:

```
-include(Filename).
```

Объявление переменных

```
X1 = #todo{status=urgent,  
text="Fix errata in book"}.
```

## Record (продолжение)

### Объявление

```
> X1 = #todo{status=urgent,  
text="Fix errata in book"}.  
  
#todo{status = urgent,who =  
joe,text = "Fix errata in  
book"}
```

### Копирование записи с переобозначением полей

```
> X2 = X1#todo{status=done}.  
  
#todo{status = done,who =  
joe,text = "Fix errata in  
book"}
```

### Извлечение значений полей из записей

```
> #todo{who=W, text=Txt} =  
X2.  
  
> W.  
  
> Txt.  
  
> X2#todo.text.
```

# Case, if

```
case Expression of
  Pattern1 [when Guard1] ->
    Expr_seq1;
  Pattern2 [when Guard2] ->
    Expr_seq2;
  ...
  LastPattern [when LastGuard] ->
    Expr_seq_last
end
```

```
if
  Guard1 ->
    Expr_seq1;
  Guard2 ->
    Expr_seq2;
  ...
  LastGuard ->
    Expr_seq_last
end
```



# Exceptions (исключения)

`exit(Why).`

Когда вы действительно хотите  
терминировать данный процесс

Сообщение вида `{'EXIT',Pid,Why}`  
будет послано всем процессам,  
которые связаны с данным

`throw(Why)`

Используется для запуска  
исключения, которое вызывающий,  
возможно захочет перехватить

`erlang:error(Why)`

Вспользуется чтобы обозначить  
"критическую" ошибку в  
программе

Это эквивалентно  
сгенерированной внутренней  
ошибке

# Exceptions syntax

```
try FuncOrExpressionSequence of  
  Pattern1 [when Guard1] -> Expressions1;  
  Pattern2 [when Guard2] -> Expressions2;  
  ...
```

```
catch  
  ExceptionType: ExPattern1 [when ExGuard1] -> ExExpressions1;  
  ExceptionType: ExPattern2 [when ExGuard2] -> ExExpressions2;
```

```
...  
after  
  AfterExpressions  
end
```

```
demo2() ->
```

```
[{I, (catch gen_exceptions:generate_exception(I))} || I <- [1,2,3,4,5]].
```

# Атрибуты

## Атрибуты модуля

```
-import(Mod, [Name1/Arity1,  
Name2/Arity2, ...]).  
-import(lists, [map/2]).  
-compile(export_all).  
-vsn(Version).
```

## Пользовательские атрибуты

```
-SomeTag(Value).  
-author({wad, shandrinov}).  
-purpose("example of attributes").  
attrs:module_info().  
attrs:module_info(attributes).
```

## Макросы

```
-define(Constant, Replacement).  
-define(Func(Var1, Var2, ..., Var), Replacement).
```

## Вызов

?MacroName

# Процессы, сообщения

```
Pid = spawn(Fun)

%% Создаёт новый параллельный процесс,
%% который вычисляет (evaluates) Fun

Pid ! Message

%% Посылает (асинхронно) сообщение
Message процессу с идентификатором Pid.

Receive
  Pattern1 [when Guard1] ->
    Expressions1;
  Pattern2 [when Guard2] ->
    Expressions2;
  ...
  after Time ->
    Expressions
End

%% Принимает сообщение, которое было
%% послано процессу.
```

```
%% Клиент-сервер

%% будем отсылать свой адрес (PID)
серверу

Pid ! {self(),{rectangle, 6, 10}}

loop() ->
  receive
    {From, {rectangle, Width, Ht}} ->
      From ! Width * Ht,
      loop();
  ...
```

# Регистрация процессов

## Приоритетный прием

```
priority_receive() ->  
  receive  
    {hi_priority, X} ->  
      {hi_priority, X}  
  after 0 ->  
    receive  
      Any ->  
        Any  
    end  
end.
```

%% Зарегистрированные процессы

register(AnAtom, Pid)

%% зарегистрировать процесс Pid с именем AnAtom .  
Регистрация не успешна, если

%% AnAtom уже был использован для регистрации процесса.

unregister(AnAtom)

%% удалить любые регистрации, связанные с AnAtom .

whereis(AnAtom) -> Pid | undefined

%% находит, где зарегистрирован AnAtom . Возвращает  
идентификатор процесса Pid ,

%% либо возвращает атом undefined , если никакой процесс  
не связан с AnAtom .

registered() -> [AnAtom::atom()]

%% возвращает список зарегистрированных процессов в  
системе.

# Связанные процессы

`link(Pid).`

`%%` После установления связи оба процесса неявно следят друг за другом. Если

`%%` умрёт А, то В получит сигнал выхода (`exit signal`) . И наоборот – если умрёт В,

`%%` то такой сигнал получит А.

`%%` Обработчик выхода

`utils:on_exit(Pid, Fun).`

`F = fun() ->`

`receive`

`X -> list_to_atom(X)`

`end`

`end.`

`Pid = spawn(F).`

`utils:on_exit(Pid, fun(Why) ->  
io:format(" ~p died with:~p~n",  
[Pid, Why]) end).`

`Pid ! 42.`

# Способы связи и обработки ошибок

**%% Подход 1: Меня не волнует, если процесс, который я создал, падает**  
**Pid = spawn(fun() -> ... end)**

**%% Подход 2: Я хочу умереть, если процесс, который я создал, падает**  
**Pid = spawn\_link(fun()  
-> ... end)**

**%% Подход 3: Я хочу обработать ошибки, если процесс, который я создал, падает**  
**...**  
**process\_flag(trap\_exit, true),**  
**Pid = spawn\_link(fun() -> ... end),**  
**...**  
**loop(...).**  
**loop(State) ->**  
**receive**  
**{'EXIT', SomePid, Reason} ->**  
**%% do something with the error**  
**loop(State1);**  
**...**  
**End.**

# Nodes

%% Запуск ноды с именем

```
erl -sname ttt
```

%% Имя узла имеет вид  
Name@Host.

%% Один сервер, несколько нод.

%% запускаем на одной ноде сервер

```
erl -sname one
```

```
lesson:multiplier().
```

%% на другой ноде

```
erl -sname two
```

%% делаем запрос

%% необходимо воспользоваться стандартной функцией

```
rpc:call(one@localhost, lesson, multiply,  
[3, 4]).
```

%% Несколько серверов с нодами в  
одной локальной сети

%% запускаем на другом сервере  
ноду (в одной подсети)

```
erl -sname one -setcookie abc  
one@vasya 1>
```

%% запускаем на своем сервере ноду  
(в одной подсети)

```
erl -sname two -setcookie abc
```

```
net_adm:ping(Node)
```

```
rpc:call(one@vasya, lesson,  
multiply, [3, 4]).
```



# Некоторые функции

**@spec spawn(Node, Mod, Func, ArgList) -> Pid**

%% создаёт новый процесс, который вычисляет `apply(Mod, Func, Args)` . Он возвращает PID нового процесса.

%% Замечание: эта форма порождения более надёжна, чем `spawn(Node, Fun)` .

%% `spawn(Node, Fun)` может сломаться, если на распределённых узлах работают хотя бы на малость отличающиеся версии соответствующего модуля.

**@spec disconnect\_node(Node) -> bool() | ignored**

**@spec monitor\_node(Node, Flag) -> true**

Если Flag имеет значение `true`, то включается мониторинг. Если Flag имеет

значение `false` , то мониторинг выключается. При включенном мониторинге процессу,

который выполнил эту функцию, посылаются сообщения `{nodeup, Node}` и

`{nodedown, Node}` в случае, когда узел Node присоединяется или покидает набор

подключенных узлов Эрланга.

**@spec node() -> Node**

Возвращает имя локального узла. Если узел не является распределённым, то возвращается `nonode@nohost` .

**@spec node(Arg) -> Node**

Возвращает узел, где находится Arg . Arg может быть PID, ссылка или порт. Если узел не является распределённым, то возвращается `nonode@nohost` .

**@spec nodes() -> [Node]**

Возвращает список всех других узлов в сети, с которыми мы соединены.

**@spec is\_alive() -> bool()**

Возвращает `true` , если локальный узел жив и может быть частью распределённой системы. В противном случае возвращает `false` .

**{RegName, Node} ! Msg**

Посылает сообщение Msg к зарегистрированному на узле Node процессу RegName .



Спасибо за внимание

