

Erlang

Wprowadzenie do języka

Programowanie w języku Erlang

Informatyka
WIEiT, AGH 2019

Plan

- Wprowadzenie
- Typy proste
- Operatory
- Typy złożone
- Zmienne
- Dopasowywanie wzorców
- Funkcje i moduły
- Funkcje wbudowane
- Instrukcje warunkowe
- Rekurencja
- Strażnicy

- 1987 - poszukiwania technologii do tworzenia systemów telekomunikacyjnych
 - 1990 - Haskell
- 1991 - implementacja maszyny wirtualnej Erlanga w C
- 1993 - pierwsza książka o Erlangu
 - 1995 - Java 1.0
- 1998 - wersja Open Source
- 1999 - 36 000 odwiedzin erlang.org
 - 2000 - C#, 2002 - F#
- 2005 - obsługa wielu rdzeni
- 2006 - 1 000 000 odwiedzin erlang.org
- 2012 - język Elixir

- Język i technologia powstały by rozwiązać konkretne problemy
 - *problem => technologia*
- Problemy były związane z programowaniem systemów telekomunikacyjnych
 - centrale telefoniczne
 - switchy telekomunikacyjne
- Wymagania:
 - dostępność,
 - obsługa błędów,
 - współbieżność i komunikacja,
 - czas rzeczywisty,
 - łatwość utrzymania.

- Wysokopoziomowy
- Funkcyjny
- Dynamicznie typowany - typy są sprawdzane w trakcie działania programu
- Silnie typowany - brak niejawnych konwersji
- Oparty o niezmiennie zmienne
- Komunikacja międzyprocesowa wbudowana w język

- Źródła są kompilowane do kodu bajtowego
- Programy są uruchamiane w maszynie wirtualnej (beam)
- Lekkie procesy, planista, wywłaszczanie
- Maszyny wirtualne mogą łączyć heterogeniczne komputery
- Brak pamięci współdzielonej (prawie)
- Garbage Collector - osobny dla każdego procesu, łatwa i skuteczna realizacja

- Nadzorowanie procesów przez inne procesy
- Podejście „happy case”
 - Implementacja dla przypadku optymistycznego
 - Jeśli coś poszło nie tak, odtworzenie procesu

Dostępność i utrzymanie systemów

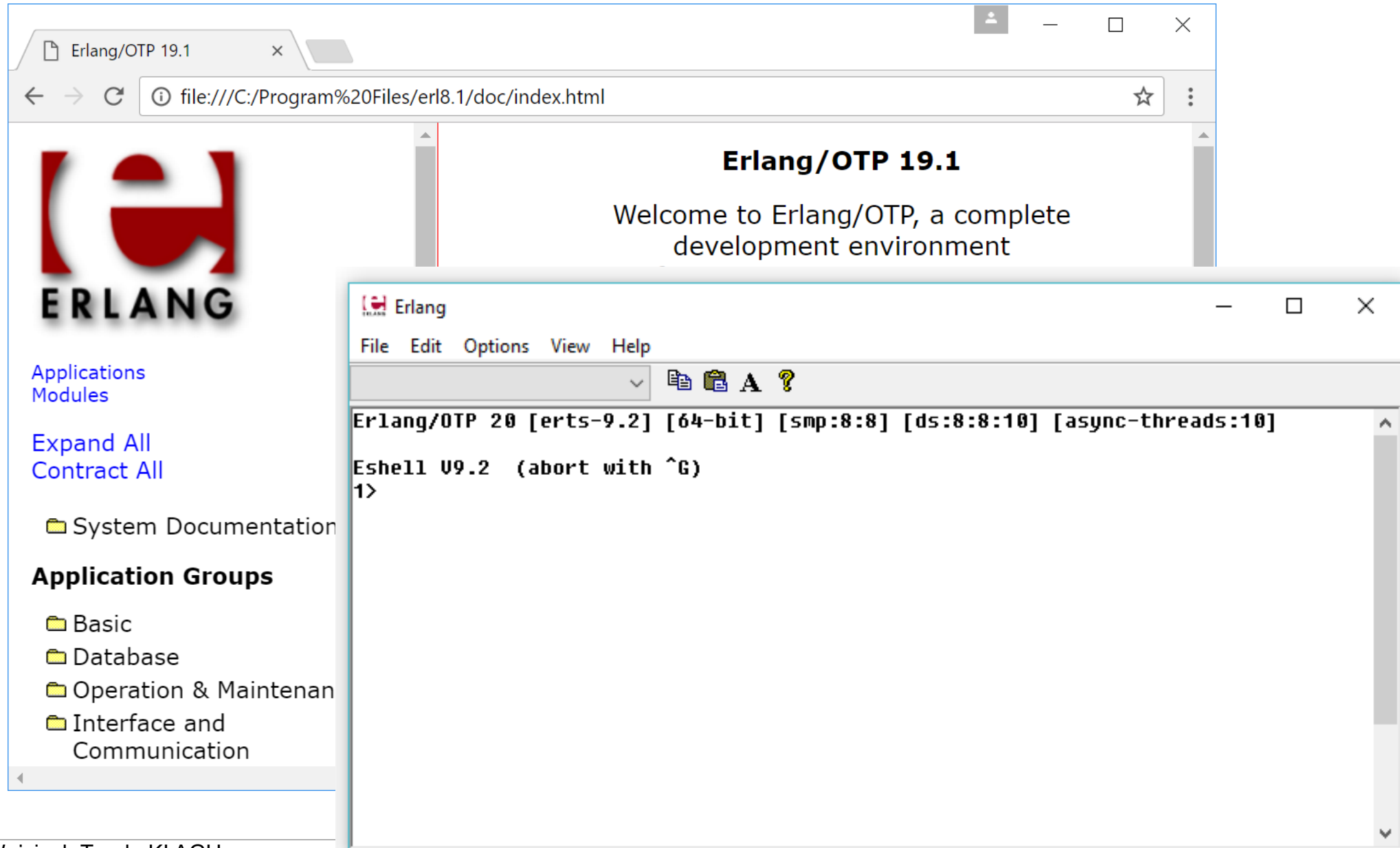
- Kod podzielony na moduły
- Dodawanie i usuwanie modułów w trakcie działania maszyny wirtualnej
- Zmiana wersji modułów w trakcie działania

- TAK:
 - Systemy rozproszone
 - Systemy masywnie współbieżne
 - Systemy udostępniania usług
 - Bazy danych
- Nie za bardzo:
 - GUI
 - Grafika
 - Obliczenia

Jak zacząć

- Erlang OTP: <http://erlang.org>
 - Maszyna wirtualna, narzędzia
 - Dokumentacja bibliotek
- Edycja kodu
 - Emacs - rozszerzenie
 - IntelliJ - wtyczka intellij-erlang
 - Eclipse - wtyczka erlIDE
 - Vim, gedit, Notepad++, TextMate, ...
- Książki
 - Programming Erlang, Joe Armstrong
 - Erlang Programming, F. Cesarini, S. Thompson
 - <http://learnyousomeerlang.com/>
- <http://www.erlangcentral.org/>

Erlang Shell



The image shows a web browser window displaying the Erlang/OTP 19.1 documentation page. The page features the Erlang logo, a welcome message, and a sidebar with navigation links. Overlaid on the bottom right is the Erlang Shell window, which shows the system configuration and the shell prompt.

Erlang/OTP 19.1
Welcome to Erlang/OTP, a complete development environment

ERLANG

Applications
Modules

Expand All
Contract All

System Documentation

Application Groups

- Basic
- Database
- Operation & Maintenance
- Interface and Communication

Erlang

File Edit Options View Help

Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10]
Eshell V9.2 (abort with ^G)
1>

Typy proste

Liczby całkowite

- Zakres ograniczony jedynie wielkością pamięci
- Małe liczby trzymane jako słowa, za duże jako bignum
- 16#val - system szesnastkowy
- \$c - kod znaku wg ASCII

```
0
10
-256
40000001
1232435435663451253413463456786345435345625679535846576
16#56AF
7#426
$a
$A
$\n
is_integer(3)
```

Liczby zmiennoprzecinkowe

- 64bit, floating point
- IEEE 754
- ~~Niezbyt wydajnie zaimplementowane~~

```
0.0  
17.456  
-34.054  
1.25E-9  
is_float(1.1)
```

- Atomy to **stałe literały**
- Zaczynają się małą literą, składają się z liter, cyfr i [`@_`]
- lub są dowolnym ciągiem zamkniętym w `' '`
- Nie można stosować słów kluczowych

```
ala
maKota
aKotMa2Ale
true
false
'Marek'
'Lech, Czech i Rus'
'cokolwiek między { $#@ \n ddd
  Eee'
'http://www.agh.edu.pl/'
is_atom('Ala ma kota')
```

Boolean

- Brak osobnego typu *bool*
- Wykorzystywane są atomy `true` i `false`

```
1 == 1
( 1 == 1 ) == true
1 /= 3
1 == 1.0
1 == 1
( 1 == 1.0 ) == false
1 /= 1.0
1 < 2
abc < xyz

is_boolean ( false )
is_boolean ( 1 == 2 )

is_atom(false)
is_atom(1 == 3)
```


Operator

Operatory logiczne

- and gorliwy
- andalso leniwy
- or gorliwy
- orelse leniwy
- xor
- not

```
not ( false )  
not ( (1 < 4) and (2 > 6) )  
  
1 = 4      => exception  
true or (1 = 4)  => exception  
true orelse (1 = 4)  => true
```

Operatory arytmetyczne

+

-

*

/

div

rem

- Jeśli oba argument to Integer, wynik to Integer
- ale dzielenie zawsze zwraca Float
- *div* i *rem* mogą być stosowane tylko dla Integer

```
1 + 1      = 2
1.0 + 1    = 2.0
17 rem 3    = 2
1 / 3      = 0.33333333333333
5.0 rem 3   = exception
trunc(5.0) rem 3.
```

Operatory bitowe

band

bor

bxor

bnot

bsl

bsr

- Można stosować dla liczb całkowitych

```
123 band 345 = 89
```

Typy złożone

Krotki (tuple)

- Złożony typ danych o stałej liczbie elementów
- Kolejność elementów jest ustalona
- Elementami mogą być dowolne poprawne wyrażenia

```
{a, b, c}
{1, 123, 3.45, zong}
{}
{'Ala ma kota', 3, 4}

{person, 'Ala', 'Nowak'}

{2+3, (a > c), is_integer(false)} = {5, false, false}

is_tuple({za, ra, za})
```

- Złożony typ danych o zmiennej liczbie elementów
- Wielkość jest ustalana dynamicznie
- Elementami mogą być dowolne poprawne wyrażenia

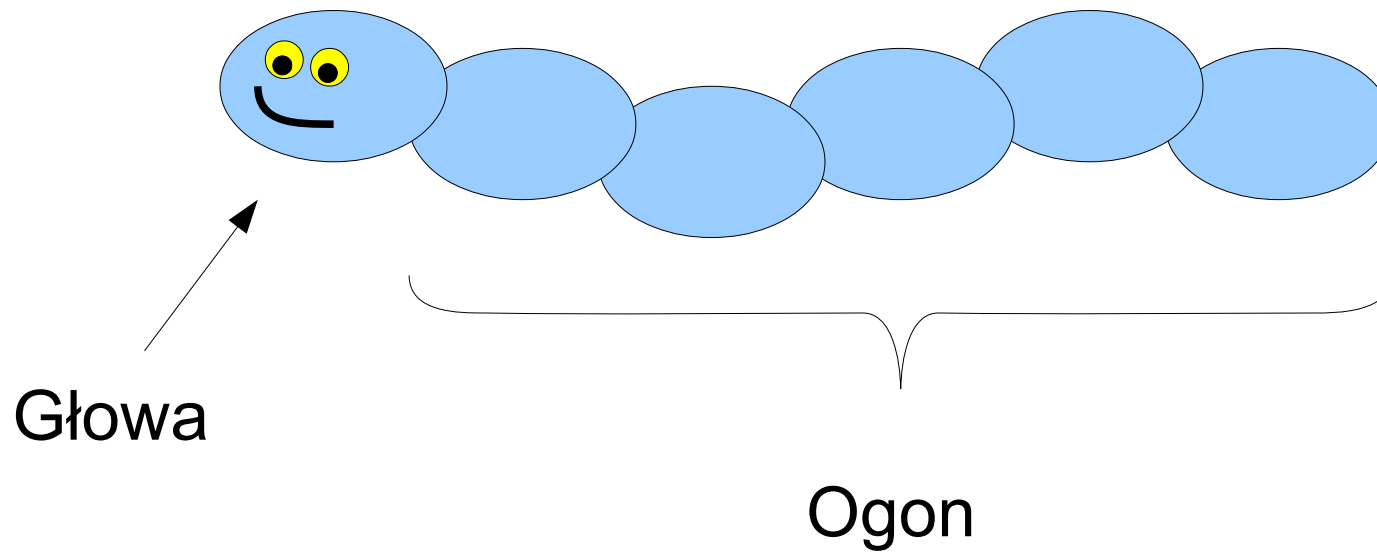
```
[a, b, c]
[1, 123, 3.45, zong]
[]

['Ala ma kota', 3, 4]

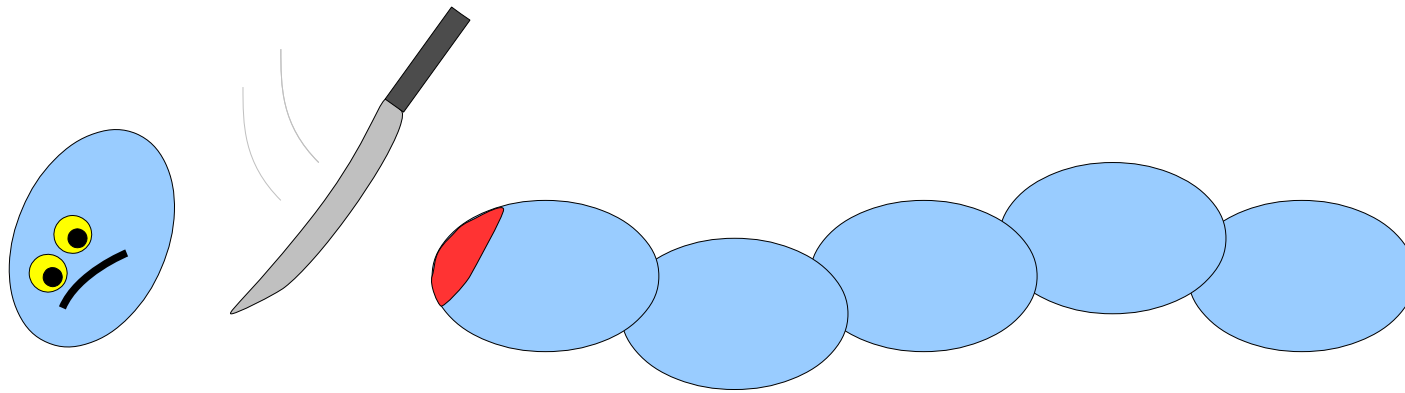
[{a,b,c}, ala, makota, 3]
[1, 2, [a, b, c, {aa, bb}, e, f], {x, y}]

[{person, 'Ala', 'Nowak'},
 {person, 'Jan', 'Kowalski'},
 {person, 'Zenon', 'Robot'} ]
```

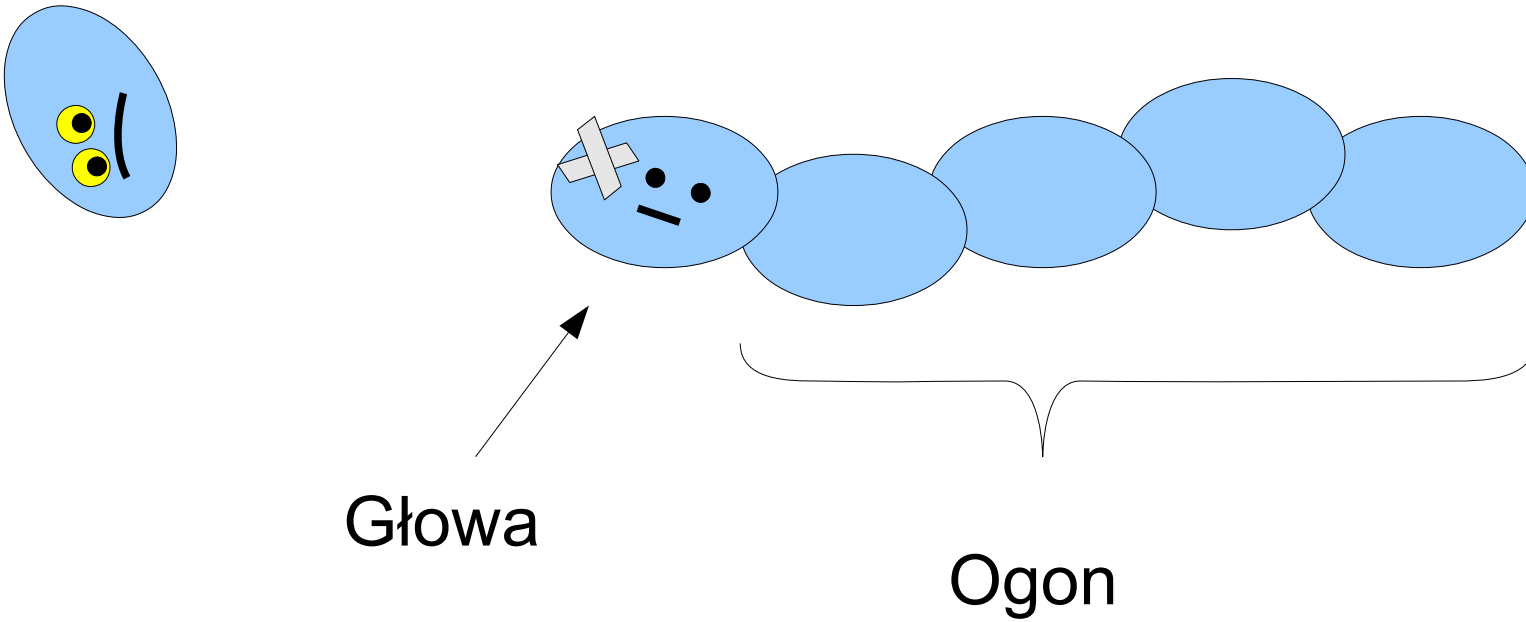
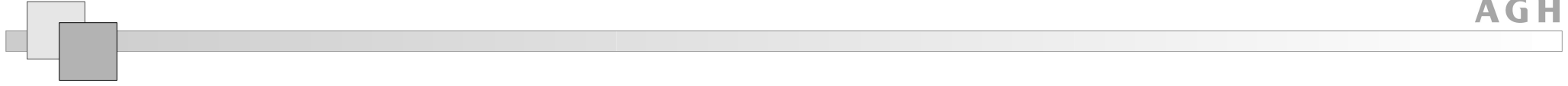
Listy – głowa i ogon



Listy – głowa i ogon



Listy – głowa i ogon



Listy – definicja rekurencyjna

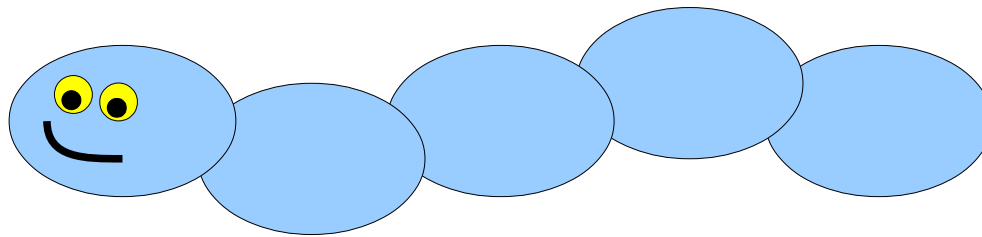
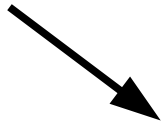
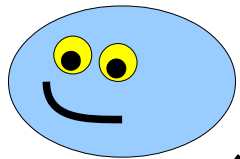
- Lista składa się z głowy i ogona lub jest pusta
- Głowa to dowolne poprawne wyrażenie
- Ogon to lista

[Head | Tail]

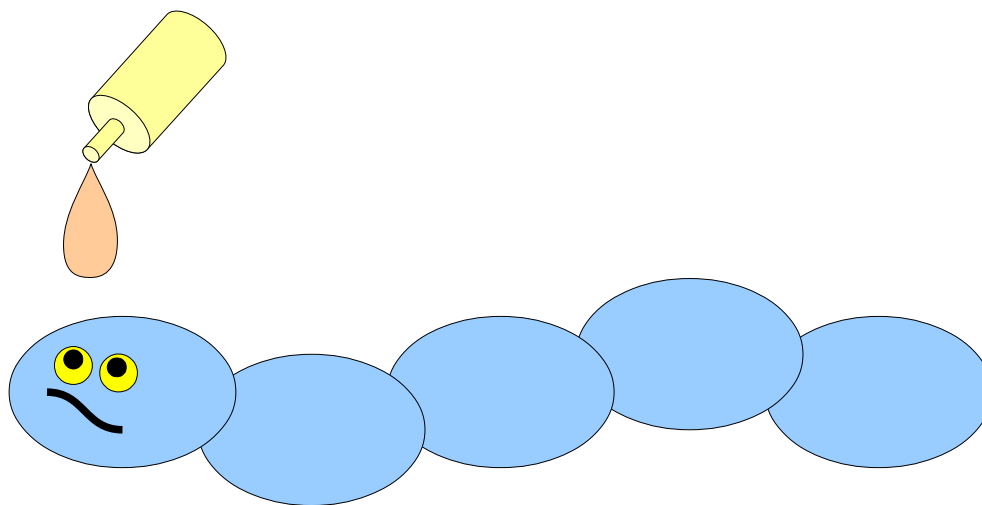
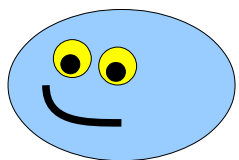
```
[a, b, c]
[a | [b, c] ]
[a | [b | [c] ] ]
[a | [b | [c | [] ] ] ]

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

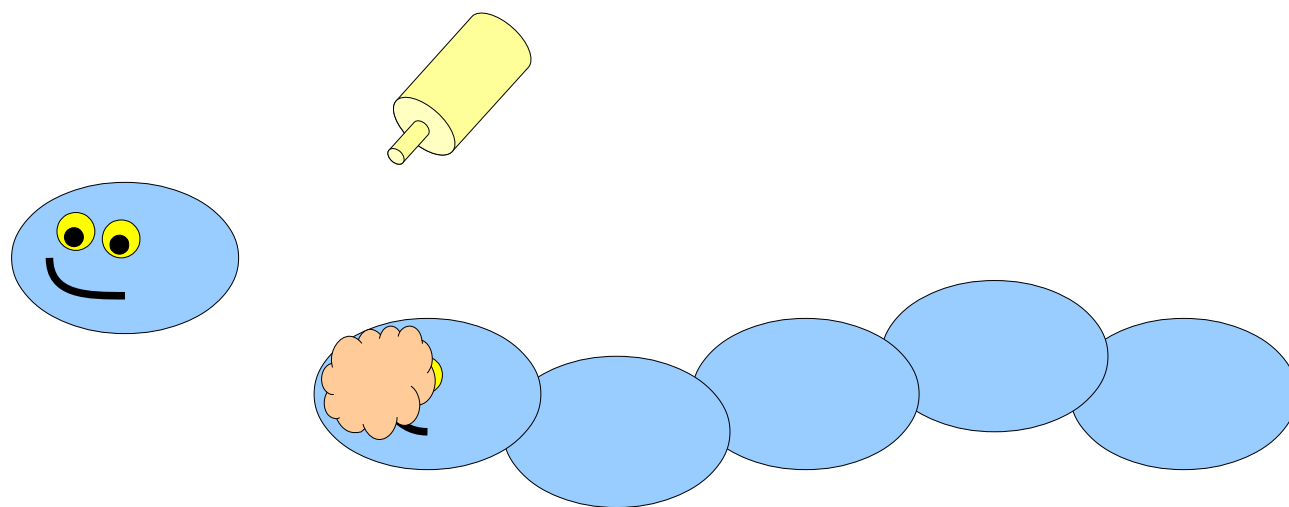
Listy – konstruowanie list



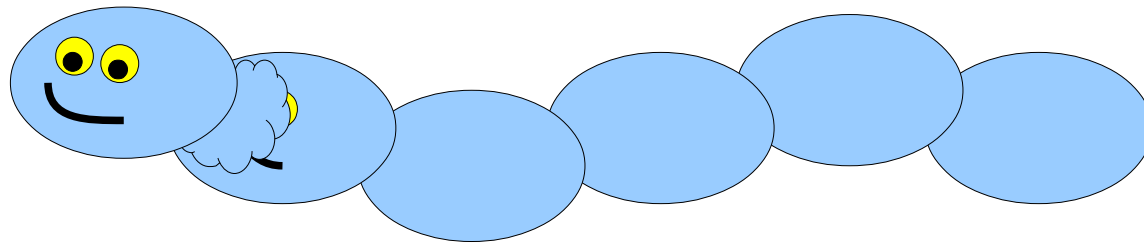
Listy – konstruowanie list



Listy – konstruowanie list



Listy – konstruowanie list



- Listy i krotki można zagnieżdżać

```
[{person, "Ala", "Nowak",  
  [  
    {telephone, "1242523541"},  
    {age, 19}  
  ]},  
{person, "Jan", "Kowalski",  
  [  
    {telephone, "506244644"},  
    {age, 24}  
  ]}  
]
```


Operatory dla list

- Konkatenacja: Lista1 ++ Lista2
- Odejmnowanie: Lista1 -- Lista2
dla każdego elementu Lista2 z Lista1 jest usuwane jego pierwsze wystąpienie

```
> [1,2,3]++[4,5].
```

```
> [1,2,3,2,1,2]--[2,1,2].
```

- Brak osobnego typu *string*
- Ciągi znaków są definiowane jako listy wartości ASCII

```
"Hello World!"  
[$H,$e,$l,$l,$o,$ , $W,$o,$r,$l,$d,$!]  
  
[72, 101, 108, 108, 111]  
  
is_list("Ala ma kota")
```

Zmienne

- Nazwy zmiennych zaczynają się Wielką Literą
- i muszą się składać z cyfr, liter i [`_@`]
- `_zmienna` - której wartości nie wykorzystujemy
- `_` - ignorowana zmienna (joker)

```
Zmienna  
InnaZmienna_zOpisemWartosci  
JeszczeInna234
```

```
_zmiennaBezZnaczenia
```

```
_
```

Przypisywanie wartości do zmiennych

- Zmienne można powiązać z danymi lub strukturami danych... można im przypisać wartość
- Do wiązania zmiennych z wartościami służy operator **=**
- Zmienną można powiązać z wartością tylko **raz**
- Czyli zmienne są niezienne... a dokładniej zmienne dzielimy na **związane** i **niezwiązane**

```
Ala = makota
Kot = ma_mysz

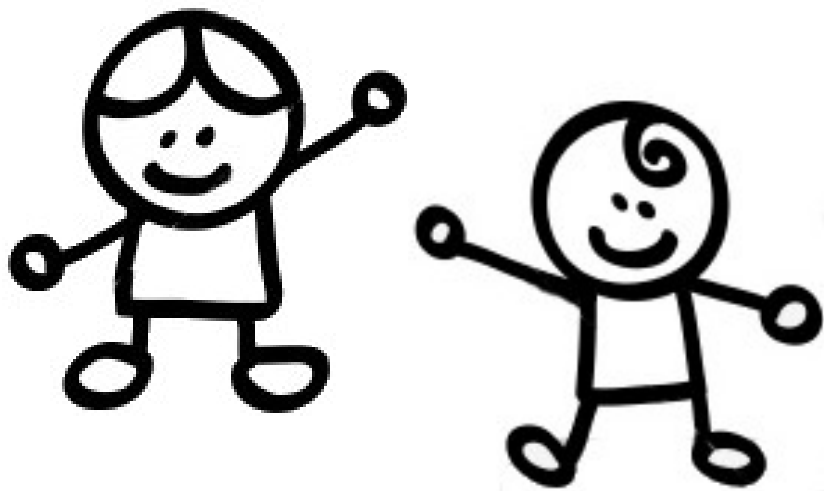
Ala = mapsa      => exception

Makot = {aaa, [b, c, d], zzz}

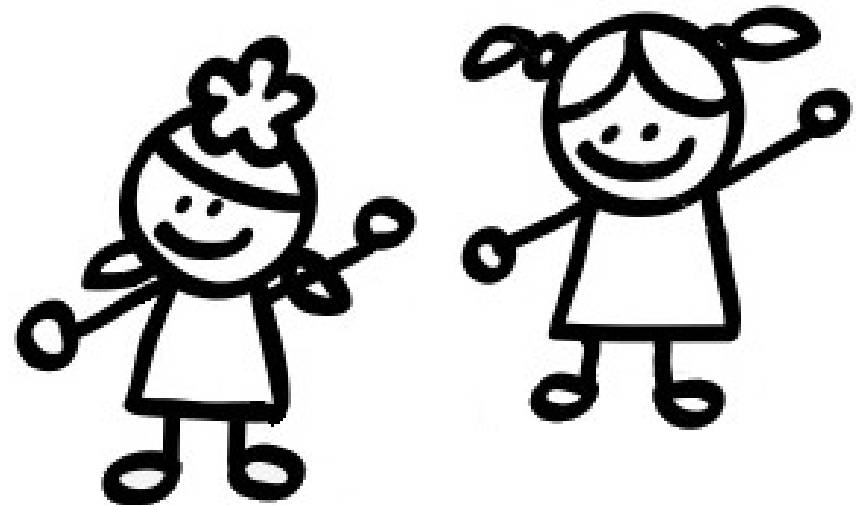
_ala = asd

_ = 123.456
```

Wiązanie zmiennych

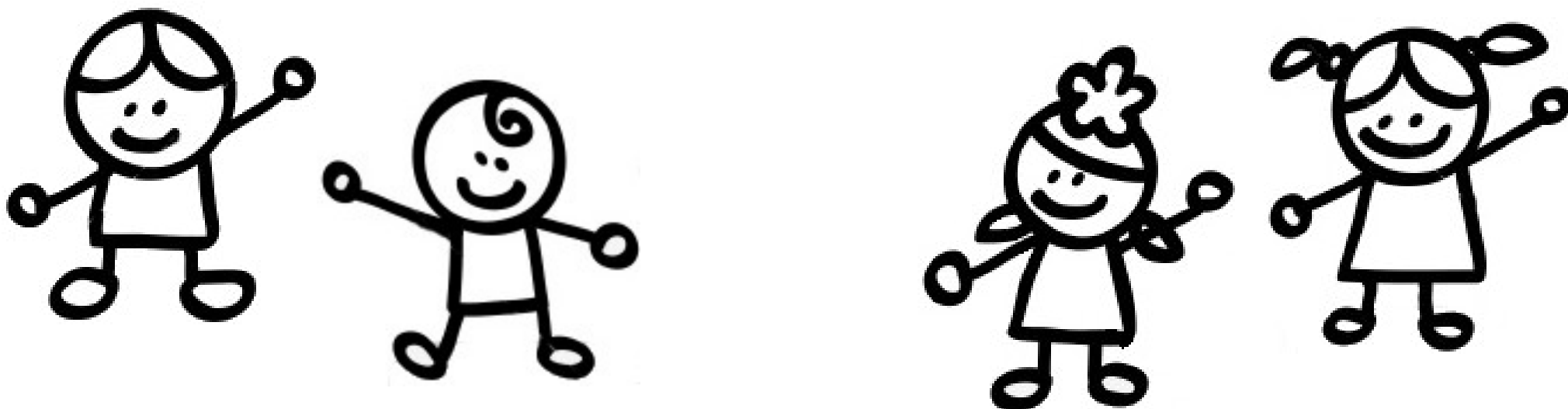


Zmienne niezwiązane

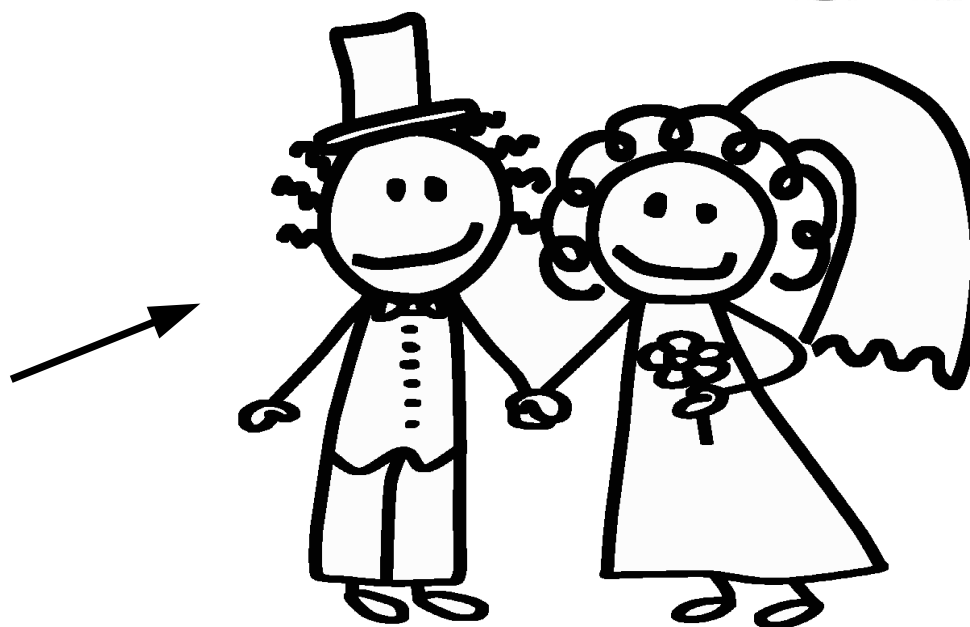


Wartości

Wiązanie zmiennych



Zmienna
związana



Pattern Matching

Wzorzec = wyrażenie

- Ale... *Zmienna = wartosc* ???
- Przypisanie wartości do zmiennej to dopasowanie wzorca
- Dowolne wyrażenie pasuje do zmiennej, która nie jest związana
- Wzorce mogą być złożone

```
Zmienna = wartosc
```

```
A = 1
```

```
A = 1
```

```
{B, C} = {ala, makota}
```

```
[X, Y, Z] = [1, 3, 7]
```

```
[H | T] = [3, 5, 7, 8]
```

Dopasowywanie wzorców – kontrolowanie wykonania kodu

```
if (v == 'image')  
    display(v);  
else if (v == 'music')  
    play(v);  
else if (v == 'text')  
    edit(v);  
else  
    ask(v);
```

```
v of  
    'image' -> display(v)  
    'music' -> play(v)  
    'text'  -> edit(v)  
    _       -> ask(v)
```



Dopasowywanie wzorców – wydobywanie danych

- Złożone wzorce pozwalają na pobranie poszczególnych elementów krotek, list, ...

```
A = 1
{B, C} = {ala, makota}

{A, A, B} = {9, 19, 123}    => exception
{A, A, B} = {9, 9, 123}

{person, Imie, _, _} = {person, "Ala", "Nowak",
                        [
                            {telephone, "1242523541"},
                            {age, 19}
                        ]}
```

Three overlapping squares of different shades of gray are positioned in the top left corner of the slide.

Funkcje i moduły

- Programy w erlangu składają się z funkcji
- Funkcje są definiowane wewnątrz modułów
- Nazwy modułów i funkcji muszą być atomami
- Funkcja jest jednoznacznie zdefiniowana przez:
 - Nazwę modułu
 - Nazwę funkcji
 - Liczbę argumentów
- Moduły mają płaską strukturę

```
io:format("Hello World!")  
  
myFunction(123)
```

Definiowanie funkcji

- Ciało funkcji składa się z klauzul
- Każda klauzula używa innego wzorca parametrów
- Pattern Matching: tylko jedna klauzula jest uruchamiana
- Funkcja zwraca wartość ostatniego wykonanego wyrażenia

```
-module(volumeCalc).  
  
-export([volume/2]).  
  
volume(qube, Edge)    ->  
    Edge * Edge * Edge;  
volume(cuboid, {Edge1, Edge2, Edge3}) ->  
    Edge1 * Edge2 * Edge3;  
volume(_, _) ->  
    io:format("Error in volume/2!"),  
    {error, cannot_calculate}.
```

Definiowanie funkcji

Deklaracja nazwy modułu

Deklaracja eksportowanych funkcji

```
-module(volumeCalc).  
  
-export([volume/2]).  
  
volume(qube, Edge)    ->  
    Edge * Edge * Edge;  
volume(cuboid, {Edge1, Edge2, Edge3}) ->  
    Edge1 * Edge2 * Edge3;  
volume(_, _) ->  
    io:format("Error in volume/2!"),  
    {error, cannot_calculate}.
```

Klauzule funkcji *volume*

Definiowanie funkcji

```
-module(volumeCalc).  
  
-export([volume/2]).  
  
volume(qube, Edge) ->  
    Edge * Edge * Edge;  
volume(cuboid, {Edge1, Edge2, Edge3}) ->  
    Edge1 * Edge2 * Edge3;  
volume(_, _) ->  
    io:format("Error in volume/2!"),  
    {error, cannot_calculate}.
```


Definiowanie funkcji

- Separatory wyrażeń: , ; .

```
-module(volumeCalc).  
  
-export([volume/2]).  
  
volume(cube, Edge) ->  
    Edge * Edge * Edge;  
volume(cuboid, {Edge1, Edge2, Edge3}) ->  
    Edge1 * Edge2 * Edge3;  
volume(_, _) ->  
    io:format("Error in volume/2!"),  
    {error, cannot_calculate}.
```

- Flagowy przykład Erlanga

```
-module(factorialCalc) .  
  
-export([factorial/1]) .  
  
factorial(0)    ->    1;  
factorial(N)    ->    N * factorial(N-1) .
```

Funkcje wbudowane

Funkcja wbudowana

- *built-in function*, czyli **BIF**
- Podstawowe funkcje dostępne z każdego miejsca programu
- Dostarczają podstawowych operacji matematycznych, służą do konwersji typów, pozwalają manipulować danymi itp.
- BIFy są wykonywane jako operacje atomowe!

```
abs(-3.33) .  
  
is_atom(Term) .  
  
list_to_tuple(List) .  
  
module_loaded(theModule) .  
  
erlang:is_integer(2.0) .
```

Three overlapping squares of different shades of gray are positioned in the top left corner of the slide.

Wyrażenia warunkowe

- Wynik wyrażenia dopasowany do wzorców
- Wyrażenie może być wołaniem funkcji
- Jedna z opcji musi zadziałać, bezpiecznie stosować _
- Zwracana jest wartość ostatniego wykonanego wyrażenia

```
factorial(N)  ->

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

Case

```
factorial(N)  ->
```

```
  case N of
```

```
    0 -> 1;
```

```
    _ -> N * factorial(N-1)
```

```
end.
```

- Definiuje listę warunków logicznych
- Wyrażenie po pierwszej prawdziwej jest wykonywane
- Jedna z opcji musi zadziałać, bezpiecznie stosować *true*
- Zwracana jest wartość ostatniego wykonanego wyrażenia

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


If

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

- Czy to jest taka sama funkcja *factorial*?

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

Rekurencja

Iterowanie po listach

- Bardzo typowy przypadek: zrobić „coś” z każdym elementem listy
- Nie ma pętli *for* :(

```
sumlist([]) ->  
    0;  
sumlist([H|T]) ->  
    H + sumlist(T).
```

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

Stack overflow

- Rekurencja jest bardzo wygodna, ale....
- ma problemy z przetwarzaniem bardzo długich (głębokich) zadań

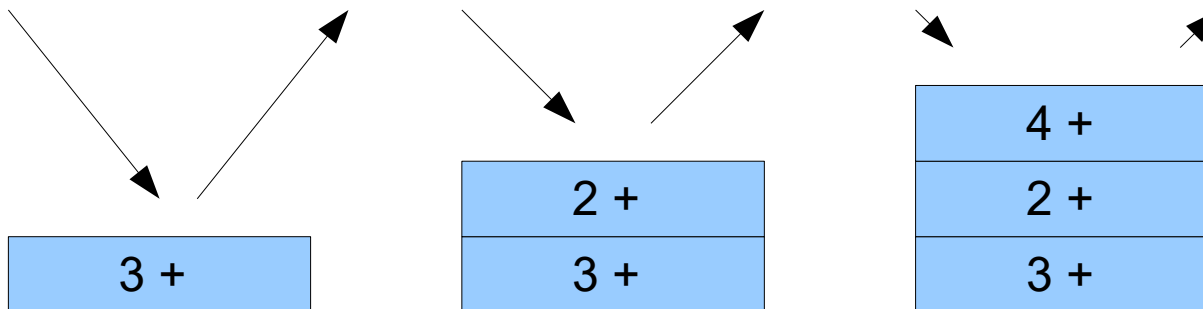
```
sumlist([]) ->  
    0;  
sumlist([H|T]) ->  
    H + sumlist(T).
```

sumlist([3,2,4]).

sumlist([2,4]).

sumlist([4]).

sumlist([]) -> 0



Wywołania ogonowe

- Wywołanie ogonowe – jeśli wywołanie funkcji f jest ostatnią ewaluacją w funkcji g , to do funkcji g nie ma już po co wracać
- Rekurencja z akumulatorem:

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

Strażnicy

- Słowo kluczowe *when* pozwana na zdefiniowanie dodatkowych warunków dopasowywania wzorców
- Guards, czyli strażnicy mogą być stosowani w:
 - Nagłówkach funkcji
 - Opcjach *case*
 - Warunkach *if*

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


Co może być strażnikiem

- Atomy, w szczególności *true* i *false*
- Porównania termów
- Wyrażenia arytmetyczne i logiczne
- Niektóre BIFy

```
is_atom/1  
is_binary/1  
is_boolean/1  
is_float/1  
is_function/1  
is_integer/1  
is_list/1  
is_number/1  
is_pid/1  
is_tuple/1
```

```
abs(Number)  
element(N, Tuple)  
float(Term)  
length(List)  
round(Number)  
self()  
tuple_size(Tuple)
```

In the top left corner, there are three overlapping squares of different shades of gray, with a horizontal line extending to the right from the bottom-right square.

Ale czy ktoś już polubił tą składnię...?

7,237

questions tagged

[erlang](#)

3,593

questions tagged

[elixir](#)

[about »](#)

63,224

questions tagged

[scala](#)

31,709

questions tagged

[haskell](#)

[about »](#)

1,221,427

questions tagged

[java](#)

[about »](#)

7,907

questions tagged

[erlang](#)

[about »](#)

5,440

questions tagged

[elixir](#)

[about »](#)

76,560

questions tagged

[scala](#)

[about »](#)

36,302

questions tagged

[haskell](#)

[about »](#)

1,385,931

questions tagged

[java](#)

[about »](#)

8395

6654

87079

40500

1519971

erlang / otp	24,533 commits		13 branches		106 releases		385 contributors		Apache-2.0	
	28,480 commits		14 branches		147 releases		433 contributors		Apache-2.0	
	32,716 commits		15 branches		215 releases		483 contributors		Apache-2.0	
elixir-lang / elixir	12,426 commits		7 branches		68 releases		528 contributors			
	14,165 commits		9 branches		86 releases		719 contributors		Apache-2.0	
	15,417 commits		11 branches		101 releases		823 contributors		Apache-2.0	
scala / scala	26,960 commits		8 branches		110 releases		311 contributors		BSD-3-Clause	
	28,277 commits		9 branches		121 releases		338 contributors			
	32,353 commits		9 branches		129 releases		428 contributors		Apache-2.0	

- Mr. Business Angel, 2017:

„I am investing in small companies and startups using Erlang. Why? Either they don't know what they are doing at all, which I will find out, or they really know what they are doing and why.”