JAKUB CZARNIECKI

# FUNCTIONAL PROGRAMMING

## A SHORT NOTE ABOUT

# FUNCTIONAL PROGRAMMING

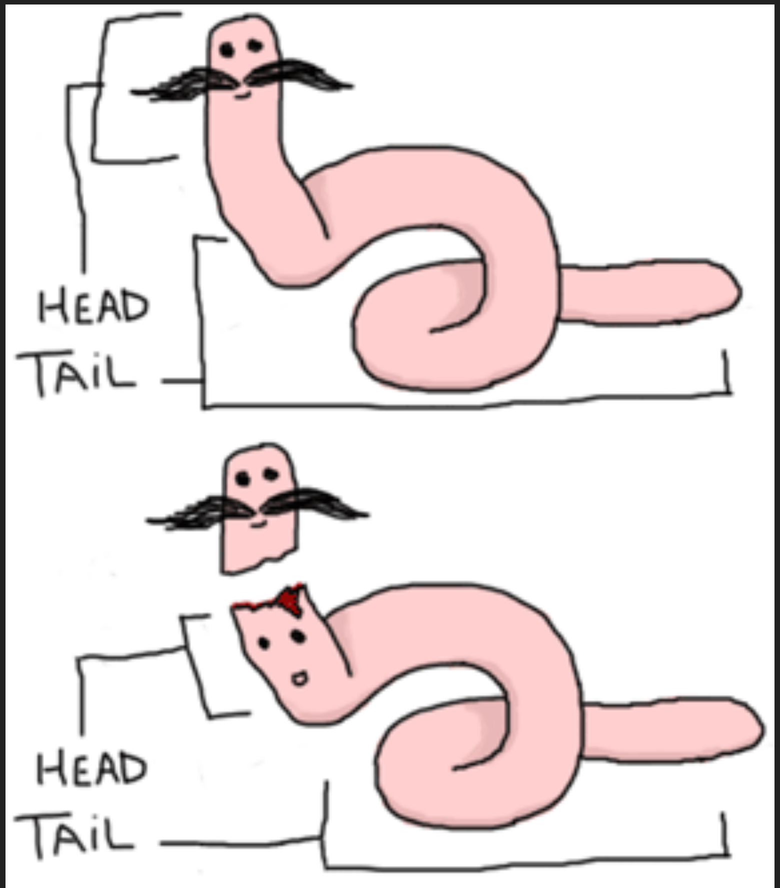
THE BASICS

# BREAD AND BUTTER

# FUNCTIONAL PROGRAMMING IS ALL ABOUT: LISTS, FUNCTIONS, RECURSION AND PATTERN MATCHING

# LISTS

```
12> [1 | [2 | []]] == [1,2].
true
```




```
>>> [X**2 for X in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
1> [X * X || X <- lists:seq(1, 10)].
[1,4,9,16,25,36,49,64,81,100]
```

# SUM ELEMENTS OF THE LIST (ERLANG)

```
16> SumElements = fun(Elements) ->
16>     fun F(Sum, []) ->
16>             Sum;
16>         F(Sum, [Element | Rest]) ->
16>             F(Sum + Element, Rest)
16>     end(0, Elements)
16> end.
#Fun<erl_eval.6.90072148>
17> SumElements([]).
0
18> SumElements([1,2,3,4]).
10
```

# SUM ELEMENTS OF THE LIST (PYTHON)

```
>>> def SumElements(Elements):
...     def F(Sum, Elements):
...         if Elements == []:
...             return Sum
...         else:
...             Element, Rest = Elements[0], Elements[1:]
...             return F(Sum + Element, Rest)
...     return F(0, Elements)
...
>>> SumElements([])
0
>>> SumElements([1,2,3,4])
10
```

WHAT ARE

# PURE FUNCTIONS

# EXAMPLE OF PURE FUNCTION (ERLANG)

```erlang
fibonacci(0) -> 0;
fibonacci(1) -> 1;
fibonacci(N) -> fibonacci(N - 1) + fibonacci(N - 2).


fibonacci2(0) -> 0;
fibonacci2(1) -> 1;
fibonacci2(N) -> fibonacci2(N - 2, 0, 1).

fibonacci2(0, N2, N1) -> N1 + N2;
fibonacci2(Left, N2, N1) -> fibonacci2(Left - 1, N1, N1 + N2).
```

# EXAMPLE OF PURE FUNCTION (PYTHON)

```python
def fibonacci(N):
    if N == 0:
        return 0
    elif N == 1:
        return 1
    else:
        return fibonacci(N - 1) + fibonacci(N - 2)

def fibonacci2(N):
    if N == 0:
        return 0
    elif N == 1:
        return 1
    else:
        def F(Left, N2, N1):
            if Left == 0:
                return N2 + N1
            else:
                return F(Left - 1, N1, N1 + N2)
        return F(N - 2, 0, 1)
```

THE IMPORTANCE OF THE

# TAIL CALL OPTIMISATION

# WHAT IS A TAIL CALL AND WHY IT'S IMPORTANT?

- Tail call is when the last expression in a function is the function call.
- Function which calls itself in the last expression is said to be *tail-recursive*.
- Erlang does optimise tail-recursive functions by replacing function arguments and jumping to the beginning of the function drastically reducing recursion overhead.
- No stack overflows.

## Naive version took almost 1 minute for fib(45)! (Erlang)

```
fibonacci(0) -> 0;
fibonacci(1) -> 1;
fibonacci(N) -> fibonacci(N - 1) + fibonacci(N - 2).

2> {Time, Result} = timer:tc(fib, fibonacci, [45]).
{54274744,1134903170}
3> Time div 1000000.
54
```

# Optimised version took 1 microsecond for fib(45) (Erlang)

```
fibonacci2(0) -> 0;
fibonacci2(1) -> 1;
fibonacci2(N) -> fibonacci2(N - 2, 0, 1).

fibonacci2(0, N2, N1) -> N1 + N2;
fibonacci2(Left, N2, N1) -> fibonacci2(Left - 1, N1, N1 + N2).

2> {Time, Result} = timer:tc(fib, fibonacci2, [45]).
{1,1134903170}
3> Time.
1
```

## … And only 10 seconds for fib(1000000)! (Erlang)

```
fibonacci2(0) -> 0;
fibonacci2(1) -> 1;
fibonacci2(N) -> fibonacci2(N - 2, 0, 1).

fibonacci2(0, N2, N1) -> N1 + N2;
fibonacci2(Left, N2, N1) -> fibonacci2(Left - 1, N1, N1 + N2).

2> {Time, Result} = timer:tc(fib, fibonacci2, [1000000]).
{10252150,_}
3> Time div 1000000.
10
```

# … But failed for N > 999 (Python)

```
>>> fibonacci2(999)
268638100244853593861467272021429239676166093189869523401231759976179817
>>> fibonacci2(1000)
RuntimeError: maximum recursion depth exceeded
>>>
```

Python does not have tail call optimisation (unfortunately)

WHAT DOES IT MEAN TO HAVE

# FIRST-CLASS FUNCTIONS

# PASSING FUNCTIONS AS ARGUMENTS TO OTHER FUNCTIONS (ERLANG)

```
7> First20 = lists:map(fun fib:fibonacci2/1, lists:seq(1, 20)).
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,
 4181,6765]
8> IsEven = fun(N) -> N rem 2 == 0 end.
#Fun<erl_eval.6.90072148>
9> EvenOnly = lists:filter(IsEven, First20).
[2,8,34,144,610,2584]
```

# PASSING FUNCTIONS AS ARGUMENTS TO OTHER FUNCTIONS (PYTHON)

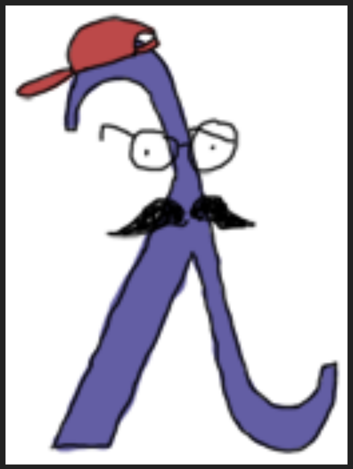
```
>>> First20 = map(fibonacci2, range(1, 21))
>>> First20
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
>>> IsEven = lambda x: 1 if x % 2 == 0 else 0
>>> EvenOnly = filter(IsEven, First20)
>>> EvenOnly
[2, 8, 34, 144, 610, 2584]
```

# RETURN FUNCTIONS AS VALUES FROM OTHER FUNCTIONS (ERLANG)

```
19> GenerateFibs = fun(N) ->
19>     fun() ->
19>         lists:map(fun fib:fibonacci2/1, lists:seq(1, N))
19>     end
19> end.
#Fun<erl_eval.6.90072148>
20> GenerateFirst100 = GenerateFibs(100).
#Fun<erl_eval.20.90072148>
21> GenerateFirst100().
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,
 4181,6765,10946,17711,28657,46368,75025,121393,196418,
 317811,514229|...]
```

# RETURN FUNCTIONS AS VALUES FROM OTHER FUNCTIONS (PYTHON)

```
>>> GenerateFibs = lambda N: lambda: map(fibonacci2, range(1, N + 1))
>>> GenerateFirst100 = GenerateFibs(100)
>>> GenerateFirst100()
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
```

AND WHAT ARE 

# HIGHER-ORDER FUNCTIONS

AN IMPORTANT PART OF ALL FUNCTIONAL PROGRAMMING LANGUAGES IS THE ABILITY TO TAKE A FUNCTION YOU DEFINED AND THEN PASS IT AS A PARAMETER TO ANOTHER FUNCTION. THIS IN TURN BINDS THAT FUNCTION PARAMETER TO A VARIABLE WHICH CAN BE USED LIKE ANY OTHER VARIABLE WITHIN THE FUNCTION. A FUNCTION THAT CAN ACCEPT OTHER FUNCTIONS TRANSPORTED AROUND THAT WAY IS NAMED A HIGHER ORDER FUNCTION. HIGHER ORDER FUNCTIONS ARE A POWERFUL MEANS OF ABSTRACTION AND ONE OF THE BEST TOOLS TO MASTER IN ERLANG.

# EXAMPLE (ERLANG)

```
29> lists:map(fun(X) -> {X, fib:fibonacci2(X)} end, lists:seq(0, 20)).
[{0,0},
 {1,1},
 {2,1},
 {3,2},
 {4,3},
 {5,5},
 {6,8},
 {7,13},
 {8,21},
 {9,34},
 {10,55},
 {11,89},
 {12,144},
 {13,233},
 {14,377},
 {15,610},
 {16,987},
 {17,1597},
 {18,2584},
 {19,4181},
 {20,6765}]
```

# EXAMPLE (PYTHON)

```
>>> map(lambda X: (X, fibonacci2(X)), range(0, 21))
[(0, 0),
 (1, 1),
 (2, 1),
 (3, 2),
 (4, 3),
 (5, 5),
 (6, 8),
 (7, 13),
 (8, 21),
 (9, 34),
 (10, 55),
 (11, 89),
 (12, 144),
 (13, 233),
 (14, 377),
 (15, 610),
 (16, 987),
 (17, 1597),
 (18, 2584),
 (19, 4181),
 (20, 6765)]
```


Side
Effects

# WHY IT IS GOOD TO LIMIT SIDE EFFECTS?

- Code is easier to test – the pure function always returns the same output for the same input.
- Code is easier to read and maintain - there are no hidden dependencies.

# PATTERN MATCHING!

# BASIC EXAMPLES (ERLANG)

```
1> Humidity = {percent, 90}.
{percent,90}
2> {percent, P} = Humidity.
{percent,90}
3> P.
90
```

```
1> Elements = [first, second, third].
[first,second,third]
2> [F | [S | [T | _]]] = Elements.
[first,second,third]
3> {F, S, T}.
{first,second,third}
```

# NOT AN EXAMPLE (PYTHON) + PEP 3132 (PYTHON 3+)

```
>>> Humidity = ('percent', 90)
>>> Humidity
('percent', 90)
>>> P = Humidity[1]
>>> P
90
```

```
>>> Elements = ['first', 'second', 'third']
>>> Elements
['first', 'second', 'third']
>>> F, S, T = Elements
>>> (F, S, T)
('first', 'second', 'third')
```

# PATTERN MATCHING IN FUNCTIONS (ERLANG)

```
1> HTTPError = fun
1>      (bad_request) -> 400;
1>      (not_found) -> 404;
1>      (internal_server_error) -> 500
1> end.
#Fun<erl_eval.6.90072148>
2>
2> HTTPError(not_found).
404
3> HTTPError(forbidden).
** exception error: no function clause matching
                    erl_eval:'-inside-an-interpreted-fun-'(forbidden)
```

# COUNTEREXAMPLE (PYTHON)

```
def HTTPError(Error):
    if Error == 'bad_request':
        return 400
    elif Error == 'not_found':
        return 404
    elif Error == 'internal_server_error':
        return 500
    else:
        raise ValueError('not supported')

>>> HTTPError('not_found')
404
>>> HTTPError('forbidden')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in HTTPError
ValueError: not supported
```

# MATCH IPV4 HEADER IN ONE LINE (ERLANG)

```
1> Packet = <<16#45, 16#00, 16#00, 16#44, 16#ad, 16#0b, 16#00, 16#00, 16#40,
16#11, 16#72, 16#72, 16#ac, 16#14, 16#02, 16#fd, 16#ac, 16#14, 16#00, 16#06>>.
<<69,0,0,68,173,11,0,0,64,17,114,114,172,20,2,253,172,20, 0,6>>

2> <<Version:4, IHL:4, TypeOfService:8, TotalLength:16,  Identification:16,
FlagX:1, FlagD:1, FlagM:1,  FragmentOffset:13, TTL:8, Protocol:8,
HeaderCheckSum:16, SourceAddress:32, DestinationAddress:32, Rest/binary>> = Packet.
<<69,0,0,68,173,11,0,0,64,17,114,114,172,20,2,253,172,20, 0,6>>

3> Version.
4
```

FUNCTIONS

# IDEMPOTENCE.

# WHAT IS IDEMPOTENCE?

- Function applied twice for the same value gives the same result as if it was applied only once.
  $f(f(x)) \equiv f(x)$
- This term also applies to wider topic like composition of functions - every single function in the chain can be idempotent but the composition as a whole may not be idempotent.

NOT MODIFYING THE DATA AT ALL

# NULLIPOTENT FUNCTIONS

# WHY BOTHER?

- Functional code is easier to scale (ex. AWS Lambda, Hadoop) because of lower overhead than OOP.
- Functional code is easier to test, read and maintain.
- It’s geeky! :D

# QUESTIONS?

THANK YOU

CONTACT ME ON
[JAKUB.CZARNIECKI@GMAIL.COM](mailto:jakub.czarniecki@gmail.com)