Ponořme se do světa funkcionálního programování

Patrik Valkovič

30.6.2017

Trochu historie

1958 – LISP
1970 – Scheme
1972 – C
1983 – C++
1990 – Haskell
1991 – Python
1995 – Java, PHP, JavaScript
2000 – C#
2009 - Clojure

Co je to funkcionální programování?

- Paradigma nebo styl programování, ve kterých je základním kamenem funkce
- Jazyky podporující funkcionální paradigma vs. funkcionální jazyky

„Všechno je funkce“

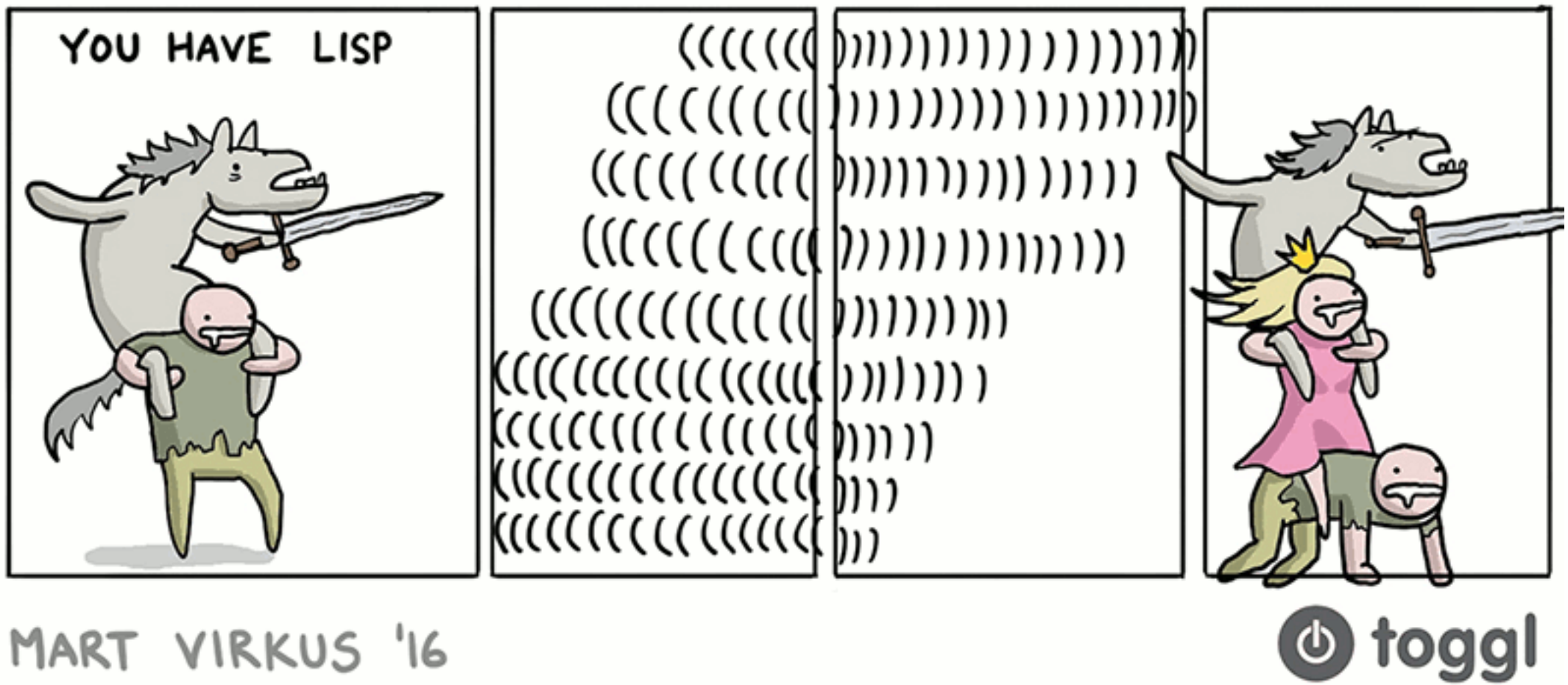
Syntax

Prefix notace

5+6	5+6×3	(5+6) ×3
+ 5 6	+ 5×6 3	× + 5 6 3

Přidáme závorky a máme hotovo

```
Give me some Clojure:
> (+ 5 6)
11
> (+ 2 (* 6 3))
20
> (* (+ 5 6) 3)
33
> (+ 5 6 (* 3 (- 8 2) (/ 10 2 )))
101
>
```


YOU HAVE LISP
MART VIRKUS '16
toggl

Say goodbye to priority

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++` `a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`.` `->`	Member access	
3	`++a` `--a`	Prefix increment and decrement	Right-to-left
	`+a` `-a`	Unary plus and minus	
	`!` `~`	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`new` `new[]`	Dynamic memory allocation	
	`delete` `delete[]`	Dynamic memory deallocation	
4	`.*` `->*`	Pointer-to-member	Left-to-right
5	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	
6	`a+b` `a-b`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	
8	`<` `<=`	For relational operators < and ≤ respectively	
	`>` `>=`	For relational operators > and ≥ respectively	
9	`==` `!=`	For relational operators = and ≠ respectively	
10	`a&b`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	

Co nám poskytuje imperativní programování?

Proměnné
Příkazy
Bloky
Podmínky
Cykly
Funkce
Třídy
Polymorfismus

Co nám poskytuje imperativní programování?

~~*Proměnné*~~

~~*Příkazy*~~

~~*Bloky*~~

Podmínky

~~*Cykly*~~

Funkce

~~*Třídy*~~

~~*Polymorfismus*~~

Funkce

Parametry

$$g(a,b,c) = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Návratová hodnota

Definice funkce

```
1 (defn sum [a b]
2       (+ a b))
3
4 (sum 1 2) ; 3
5 (sum 1 1) ; 2
```

Podmínky

```
1 (if condition true false)
2
3 (defn max [a b]
4     (if (> a b) a b))
```

Výsledek funkce musí záviset pouze na parametrech

Neexistuje stav $\Rightarrow$ *nepotřebujeme proměnné*

„Funkce je orákulum“

Free of side-effects

Funkce nesmí změnit vnější stav

```
1 void movePoint(Point p){
2       p.x += 1;
3       p.y += 1;
4 }
```

```
1 Point movePoint(Point p){
2     Point n = new Point(p)
3     n.x += 1
4     n.y += 1
5     return n
6 }
```

```
1 Point movePoint(Point p){
2     return new Point(
3         p.x+1,
4         p.y+1);
5 }
```

Return nemusíme uvádět

```
1 (defn sum [a b]
2       (+ a b))
3
4 (sum 1 2) ; 3
5 (sum 1 1) ; 2
```

Parametry musí být immutable

```
1 Point movePoint(Point p){
2     Point p1=p.setX(p.getX()+1);
3     Point p2=p.setY(p1.getY()+1);
4     return p2;
5 }
```

Funkce bez side-effects a s immutable parametry

Existuje pouze globální konstantní stav, který se nemění

Datové typy

```
Give me some Clojure:
> 128
128
> "Hello World!"
"Hello World!"
> (list 1 2 3)
(1 2 3)
> (vector 1 2 3)
[1 2 3]
> (set [1 2 3 1 2])
#{1 2 3}
> (hash-map :key1 1, :key2 2)
{:key2 2, :key1 1}
>
```

Operace nad čísly

```
Give me some Clojure:
> (+ 1 2 3)
6
> (/ 10 3)
10/3
> (float (/ 10 3))
3.3333333
> (zero? 1)
false
> (zero? 0)
true
> (odd? 2)
false
>
```

Operace nad řetězci

```
Give me some Clojure:
> (count "Hello World!")
12
> (subs "Hello World!" 5)
" World!"
> (clojure.string/trim "  Hello World!   ")
"Hello World!"
> (clojure.string/join "_" ["Hello " "World!" "!"])
"Hello _World!_!"
>
```

Operace nad listem a vectorem

```
Give me some Clojure:
> (conj '(1 2 3) 0)
(0 1 2 3)
> (conj [1 2 3] 4)
[1 2 3 4]
> (cons 0 '(1 2 3))
(0 1 2 3)
> (cons 0 [1 2 3])
(0 1 2 3)
> (nth [1 2 3] 1)
2
> (first '(1 2 3))
1
> (rest '(1 2 3))
(2 3)
>
```

Operace nad množinou

```
Give me some Clojure:
> (conj #{1 2 3} 4)
#{1 2 3 4}
> (conj {:a 1 ,:b 2 ,:c 3} {:d 4})
{:d 4, :a 1, :c 3, :b 2}
> (contains? #{1 2 3} 1)
true
> (clojure.set/union #{1 2 3} #{3 4 5})
#{1 2 3 4 5}
> (clojure.set/subset? #{1 2} #{1 2 3})
true
```

Operace slovníkem

```
> (:a {:a 1,:b 2})
1
> (keys {:a 1,:b 2,:c 1})
(:a :c :b)
> (vals {:a 1,:b 2,:c 1})
(1 1 2)
> (merge {:a 1,:b 2} {:b 3,:c 4})
{:c 4, :a 1, :b 3}
> (assoc {:a 1,:b 2} :c 3)
{:c 3, :a 1, :b 2}
>
```

Jak na proměnné?

Proměnná může uložit pouze výpočet z parametrů

⇒

Hodnotu jsme vždy schopni dopočítat

Originální kód v Javě

```
1 int myAbsExp(x){
2     int result = x*x+2*x-6;
3     if(result < 0)
4         result *= -1;
5     return result;
6 }
```

Chytřejší kód v Javě

```
1 int myAbsExp(int x){
2     return Math.abs(
3         x*x+2*x-6);
4 }
```

Kód v Clojure

```clojure
(defn abs [result]
    (if (pos? result)
        result
        (* -1 result)))

(defn myExp [x]
    (+ (* x x) (* 2 x) -6))

(defn myAbsExp [x]
    (abs (myExp x)))
```

Chytřejší kód v Clojure

```clojure
(defn myExp [x]
    (+ (* x x) (* 2 x) -6))

(def myAbsExp [x]
    (let [result (myExp x)]
        (if (pos? result)
            result
            (* -1 result))))
```

Všechno přece nemůže být funkcionální!!

Některé operace (vstupně/výstupní) musí nutně měnit globální stav.

Jsou tedy v rozporu s paradigmatem

Print

Vypíše text do konzole a nemá návratovou hodnotu!

```
1 (print 5)
2 (print (+ 5 6))
```

Read-line

Čte vstup z konzole, ale porušuje funkcionální paradigma, protože se nechová jako orákulum

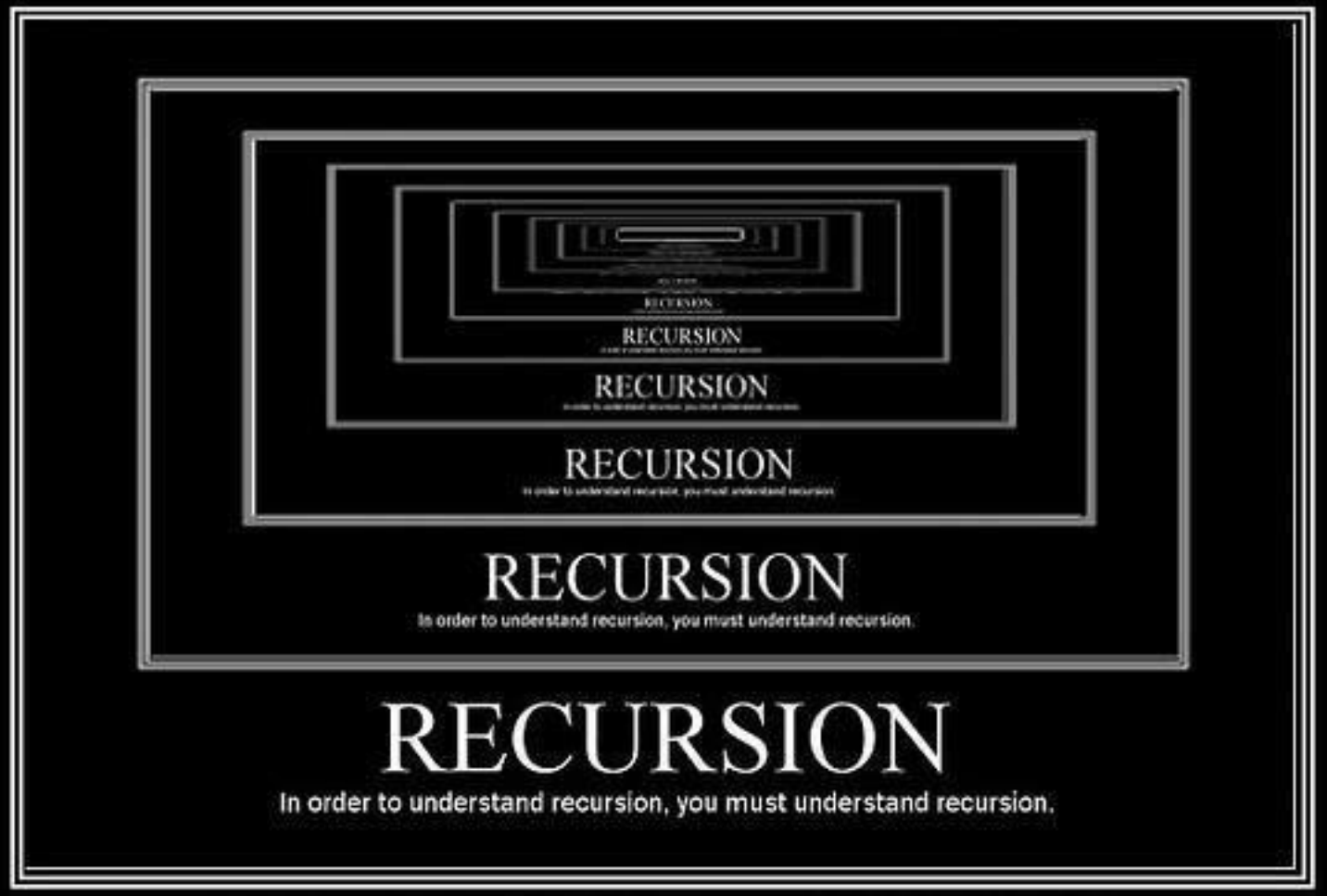
```
1 (defn print_input []
2     (print
3         (Integer/parseInt
4             (str (read-line))))
```

Do

Sekvenčně spustí sérii příkazů a vrátí výsledek posledního

```
1 (defn print_plus3 [x]
2     (do
3         (print x)
4         (print (+ 1 x))
5         (print (+ 2 x))))
```

A co cykly?



RECURSION

In order to understand recursion, you must understand recursion.

```
1 for(int i=0;i<10;i++)
2     print(i);
```

```
1 (defn oneToTen [cur end]
2     (if (< cur end)
3         (do
4             (print cur)
5             (oneToTen (inc cur) end))
6         nil))
```

```java
int[] arr = new int[] {1,2,3,4,5,6,7,8,9,0};
for(int i=0;i<arr.length;i++)
    System.out.print(arr[i]);
```

```clojure
(defn listPrint [l]
    (if (empty? l)
        nil
        (do
            (print (first l))
            (listPrint (rest l)))))
```

```
for(int i = 0; i < data.length / 2; i++)
{
    int temp = data[i];
    data[i] = data[data.length - i - 1];
    data[data.length - i - 1] = temp;
}
```

```
(defn implementation [new old]
  (if (empty? old)
        new
        (implementation (conj new (first old)) (rest old))))

(defn reverse [l]
    (implementation '() l))
```

Bubble-sort v Clojure

```clojure
(defn bubble-step [coll was-changed less?]
  (if (second coll)
    (let [[x1 x2 & xs] coll
          is-less-than (less? x1 x2)
          smaller (if is-less-than x1 x2)
          larger (if is-less-than x2 x1)
          is-changed (or was-changed (not is-less-than))
          [sorted is-sorted] (bubble-step (cons larger xs) is-changed less?)]
      [(cons smaller sorted) is-sorted])
    [coll (not was-changed)]))

(defn bubble-sort [coll less?]
  (loop [[coll is-sorted] [coll false]]
    (if is-sorted
      coll
      (recur (bubble-step coll is-sorted less?)))))
```

Typické rysy funkcionálních jazyků

Funkce je first-class citizen

Funkce je na stejné úrovni jako datové typy

Funkce jako parametr (heigher-order functions)

```
1 (defn operator [op param1 param2]
2     (op param1 param2))
3
4 (defn my-max [f s]
5   (if (< f s) s f))
6
7 (operator + 1 1)
8 (operator my-max 2 3)
```

Anonymní funkce

```clojure
(defn operator [op param1 param2]
    (op param1 param2))

(operator (fn [f s] (max f s)) 2 3)
```

Closures

Co zobrazí tento kód v JavaScriptu?

```
1 for(var a=0;a<10;a++)
2     setTimeout(function(){
3         console.log(a)
4     },100)
```

Closures

```
1 for(var a=0;a<10;a++)
2     (function(a){
3         setTimeout(function(){
4             console.log(a)
5         },100)
6     })(a)
```

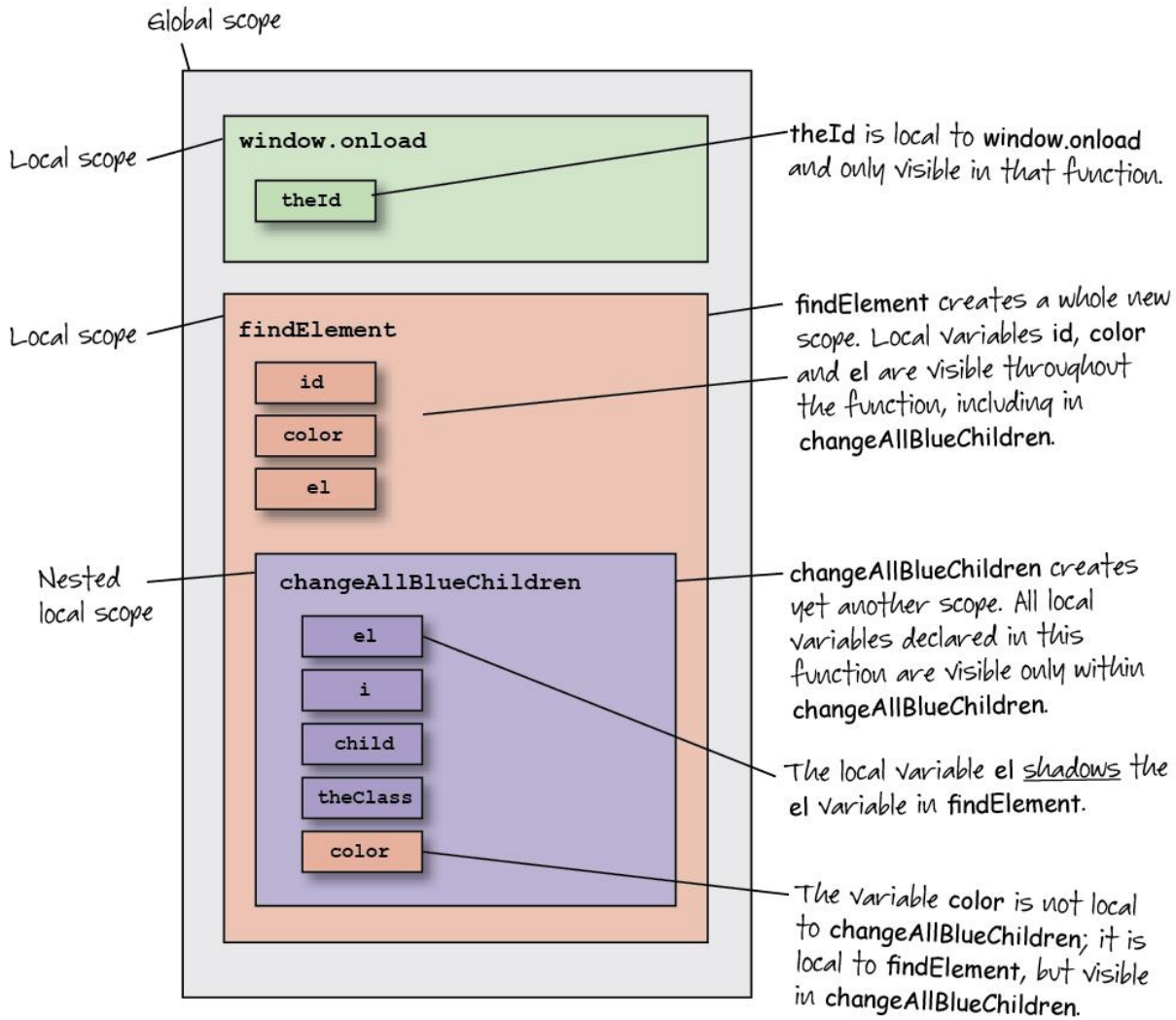
Částečná specializace

```
1 (defn create-plus [increase]
2     (fn [l r]
3         (+ increase l r)))
4
5 ((create-plus 5) 1 1)  ;return 7
```

Dynamický scope

Jiný pohled na proměnné

Lexikální scope



Dynamický scope

```
 1 (def ^:dynamic first 0)
 2 (def ^:dynamic second 0)
 3
 4 (defn plus []
 5     (+ first second))
 6
 7 (plus)                    ; 0
 8 (binding [first 1]        ; 1
 9     (plus))
10 (binding [first 1         ; 2
11           second 1]
12     (plus))
```

Dynamický scope

Hodnota proměnné není určena její pozicí, ale provedenými příkazy

Dynamický scope

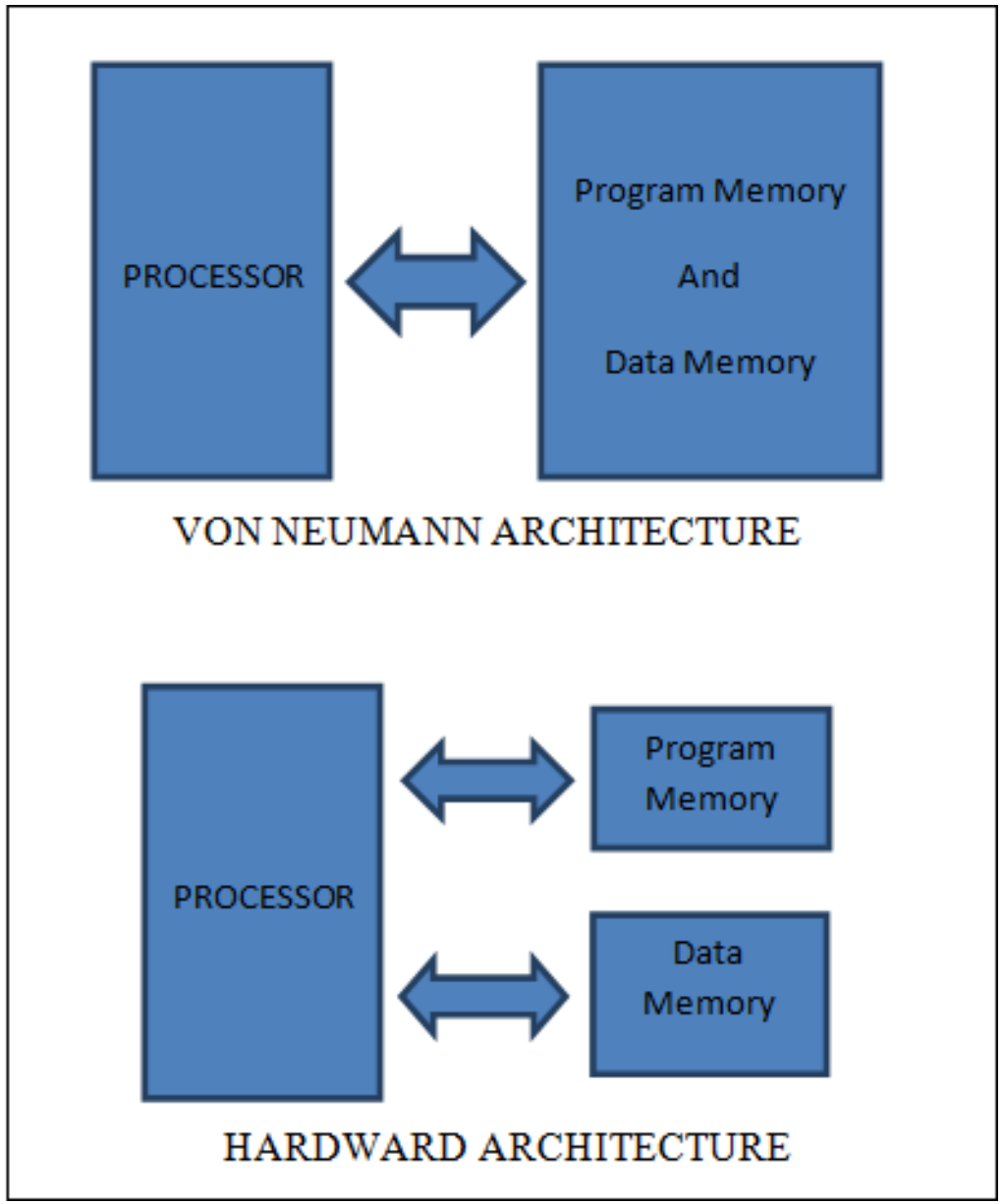
```
 1 int b = 5;
 2 int foo()
 3 {
 4     int a = b + 5;
 5     return a;
 6 }
 7
 8 int bar()
 9 {
10     int b = 2;
11     return foo();
12 }
```

```
14 int main()
15 {
16     foo(); // 10
17     bar(); // 10/7
18     return 0;
19 }
```

Dynamický scope v JavaScriptu?

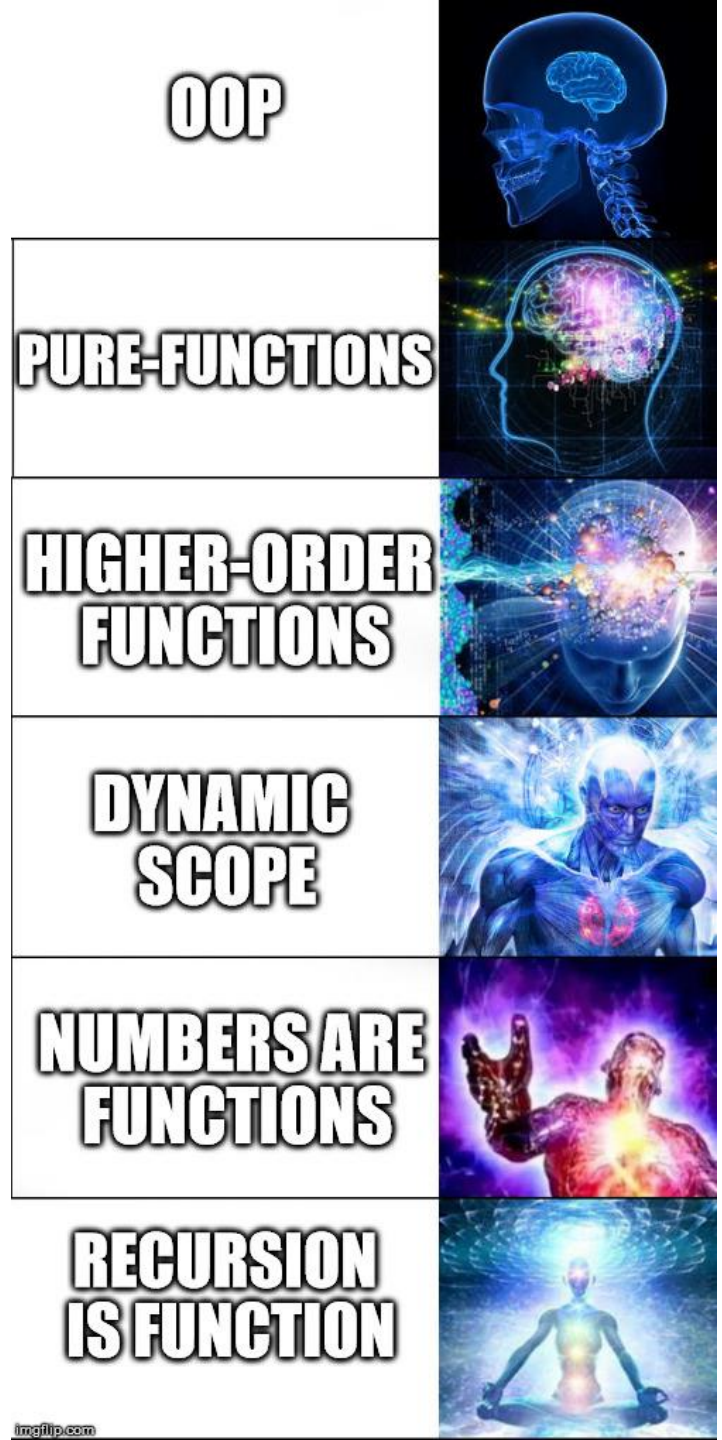
```
 1 class Point{
 2     constructor(x,y){
 3         this.x = x
 4         this.y = y
 5     }
 6
 7     print(){
 8         console.log(this.x,this.y)
 9     }
10 }
11
12 let p = new Point(1,1)
13 p.print() // 1 1
14 p.print.call({x:2,y:2}) // 2 2
15 Point.prototype.print.call({x:3,y:3}) // 3 3
```

Pro	Proti
Snadno testovatelné	Svět není neměnný
Snadno paralelizovatelné	Výkonově pokulhává


PROCESSOR
Program Memory
And
Data Memory
VON NEUMANN ARCHITECTURE
PROCESSOR
Program
Memory
Data
Memory
HARDWARD ARCHITECTURE

Funkcionální programování je skvělé pro implementaci algoritmů

Programy se ale skládají i z dat


OOP
PURE-FUNCTIONS
HIGHER-ORDER FUNCTIONS
DYNAMIC SCOPE
NUMBERS ARE FUNCTIONS
RECURSION IS FUNCTION
imgflip.com

Jak reprezentovat čísla?

```
ccccs
```

```
(defn three [c s] (c (c (c s))))
```

```
(three (fn [& args] (print "1")) 0)
```

Increment

```
cccs => ccccs
```

```
(defn increment [num c s]
  (c (num c s)))
```

```
(increment three (fn [& args] (print "1")) 0)
```

Plus

```
cccs + cccs => ccccccs
```

```
(defn plus [left right c s]
  (left c (right c s)))
```

```
(plus three three (fn [& args] (print "1")) 0)
```

Krát

```
cccs + cccs => ccccccccccs
```

```
(defn multiple [left right c s]
  (left (fn [s] (right c s)) s))
```

True False

```
(defn True [t f] t)
(defn False [t f] f)
```

```
((False (fn [] (print "True")) (fn [] (print "False")) ))
((True (fn [] (print "True")) (fn [] (print "False")) ))
```

And Or

```
(defn And [l r t f]
  (r (l t f) f))

(defn Or [l r t f]
  (r t (l t f)))
```

Not

```
(defn Not [ex t f] (ex f t))
```

```
((Not True printTrue printFalse))
((Not False printTrue printFalse))
```

isZero

```
(defn isZero [num] (num (fn [a] False) True))
```

```
(defn three [c s] (c (c (c s))))
(defn zero [c s] s)
(((isZero zero) printTrue printFalse))
(((isZero zero) printTrue printFalse))
```

Alonzo Church

Rekurze

mindfuck²

Funkcionální rekurze

```
1  (defn Y [f]
2    ((fn [x] (x x))
3     (fn [x]
4       (f (fn [& args]
5            (apply (x x) args))))))
6
7  (defn fac [f]
8          (fn [n]
9            (if (zero? n) 1 (* n (f (dec n))))))
10
11 ((Y fac) 5)
```

Děkuji za pozornost

Patrik Valkovič

30.6.2017

LISP

```lisp
(defun bubble-sort-vector (vector predicate &aux (len (1- (length vector))))
  (do ((swapped t)) ((not swapped) vector)
    (setf swapped nil)
    (do ((i (min 0 len) (1+ i))) ((eql i len))
      (when (funcall predicate (aref vector (1+ i)) (aref vector i))
        (rotatef (aref vector i) (aref vector (1+ i)))
        (setf swapped t)))))

(defun bubble-sort-list (list predicate)
  (do ((swapped t)) ((not swapped) list)
    (setf swapped nil)
    (do ((list list (rest list))) ((endp (rest list)))
      (when (funcall predicate (second list) (first list))
        (rotatef (first list) (second list))
        (setf swapped t)))))

(defun bubble-sort (sequence predicate)
  (etypecase sequence
    (list (bubble-sort-list sequence predicate))
    (vector (bubble-sort-vector sequence predicate))))
```

Scheme

```
(define (bubble-sort x gt?)
  (letrec
    ((fix (lambda (f i)
       (if (equal? i (f i))
           i
           (fix f (f i)))))

     (sort-step (lambda (l)
        (if (or (null? l) (null? (cdr l)))
            l
            (if (gt? (car l) (cadr l))
                (cons (cadr l) (sort-step (cons (car l) (cddr l))))
                (cons (car  l) (sort-step (cdr l))))))))

  (fix sort-step x)))
```

Clojure

```clojure
(defn- bubble-step
  "was-changed: whether any elements prior to the current first element
  were swapped;
  returns a two-element vector [partially-sorted-sequence is-sorted]"
 [less? xs was-changed]
  (if (< (count xs) 2)
    [xs (not was-changed)]
    (let [[x1 x2 & xr] xs
          first-is-smaller   (less? x1 x2)
          is-changed         (or was-changed (not first-is-smaller))
          [smaller larger]   (if first-is-smaller [x1 x2] [x2 x1])
          [result is-sorted] (bubble-step
                              less? (cons larger xr) is-changed)]
      [(cons smaller result) is-sorted])))

(defn bubble-sort
  "Takes an optional less-than predicate and a sequence.
  Returns the sorted sequence.
  Very inefficient (O(n²))"
  ([xs] (bubble-sort <= xs))
  ([less? xs]
     (let [[result is-sorted] (bubble-step less? xs false)]
       (if is-sorted
         result
         (recur less? result)))))

(println (bubble-sort [10 9 8 7 6 5 4 3 2 1]))
```

Haskell

```haskell
bsort :: Ord a => [a] -> [a]
bsort s = case _bsort s of
               t | t == s    -> t
                 | otherwise -> bsort t
  where _bsort (x:x2:xs) | x > x2     = x2:(_bsort (x:xs))
                         | otherwise = x:(_bsort (x2:xs))
        _bsort s = s
```

Datové typy

Číslo – Řetězec
List – Vector
Množina – Slovník

```
Give me some Clojure:
> 128
128
> "Hello World!"
"Hello World!"
> (list 1 2 3)
(1 2 3)
> (vector 1 2 3)
[1 2 3]
> (set [1 2 3 1 2])
#{1 2 3}
> (hash-map :key1 1, :key2 2)
{:key2 2, :key1 1}
>
```

Operace nad čísly

```
Give me some Clojure:
> (+ 1 2 3)
6
> (/ 10 3)
10/3
> (float (/ 10 3))
3.3333333
> (zero? 1)
false
> (zero? 0)
true
> (odd? 2)
false
>
```

Operace nad řetězci

```
Give me some Clojure:
> (count "Hello World!")
12
> (subs "Hello World!" 5)
" World!"
> (clojure.string/trim "  Hello World!   ")
"Hello World!"
> (clojure.string/join "_" ["Hello " "World!" "!"])
"Hello _World!_!"
>
```

Operace nad listem a vectorem

```
Give me some Clojure:
> (conj '(1 2 3) 0)
(0 1 2 3)
> (conj [1 2 3] 4)
[1 2 3 4]
> (cons 0 '(1 2 3))
(0 1 2 3)
> (cons 0 [1 2 3])
(0 1 2 3)
> (nth [1 2 3] 1)
2
> (first '(1 2 3))
1
> (rest '(1 2 3))
(2 3)
>
```

Operace nad množinou a slovníkem

```
Give me some Clojure:
> (conj #{1 2 3} 4)
#{1 2 3 4}
> (conj {:a 1 ,:b 2 ,:c 3} {:d 4})
{:d 4, :a 1, :c 3, :b 2}
> (contains? #{1 2 3} 1)
true
> (clojure.set/union #{1 2 3} #{3 4 5})
#{1 2 3 4 5}
> (clojure.set/subset? #{1 2} #{1 2 3})
true
> (:a {:a 1,:b 2})
1
> (keys {:a 1,:b 2,:c 1})
(:a :c :b)
> (vals {:a 1,:b 2,:c 1})
(1 1 2)
> (merge {:a 1,:b 2} {:b 3,:c 4})
{:c 4, :a 1, :b 3}
> (assoc {:a 1,:b 2} :c 3)
{:c 3, :a 1, :b 2}
>
```