FP & CLOJURE

КАК ПРОЖИТЬ БЕЗ ПЕРЕМЕННЫХ

Дмитрий Цепелев / @dmitrytsepelev

AnjLab

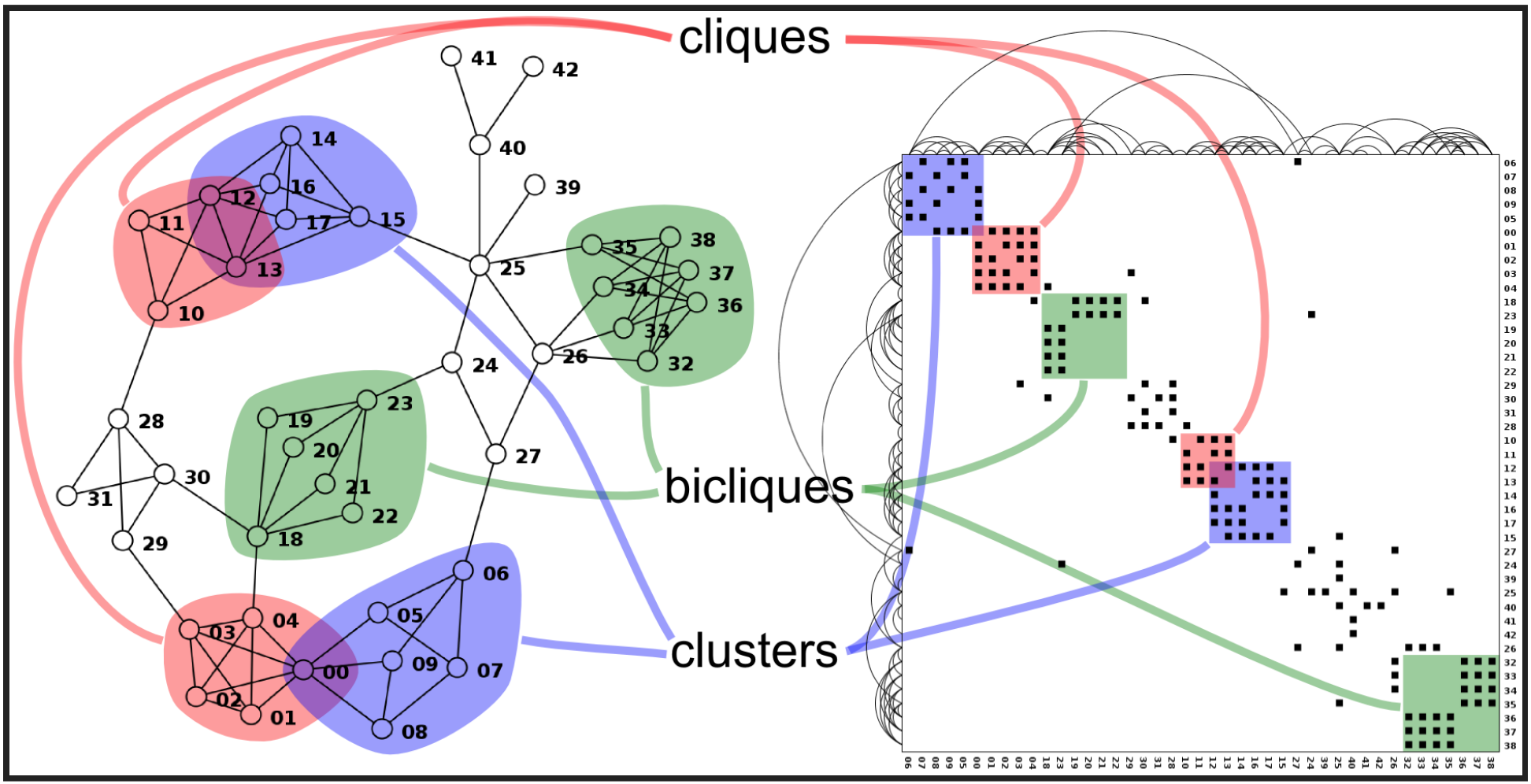
г. Владимир

СУТЬ ФП

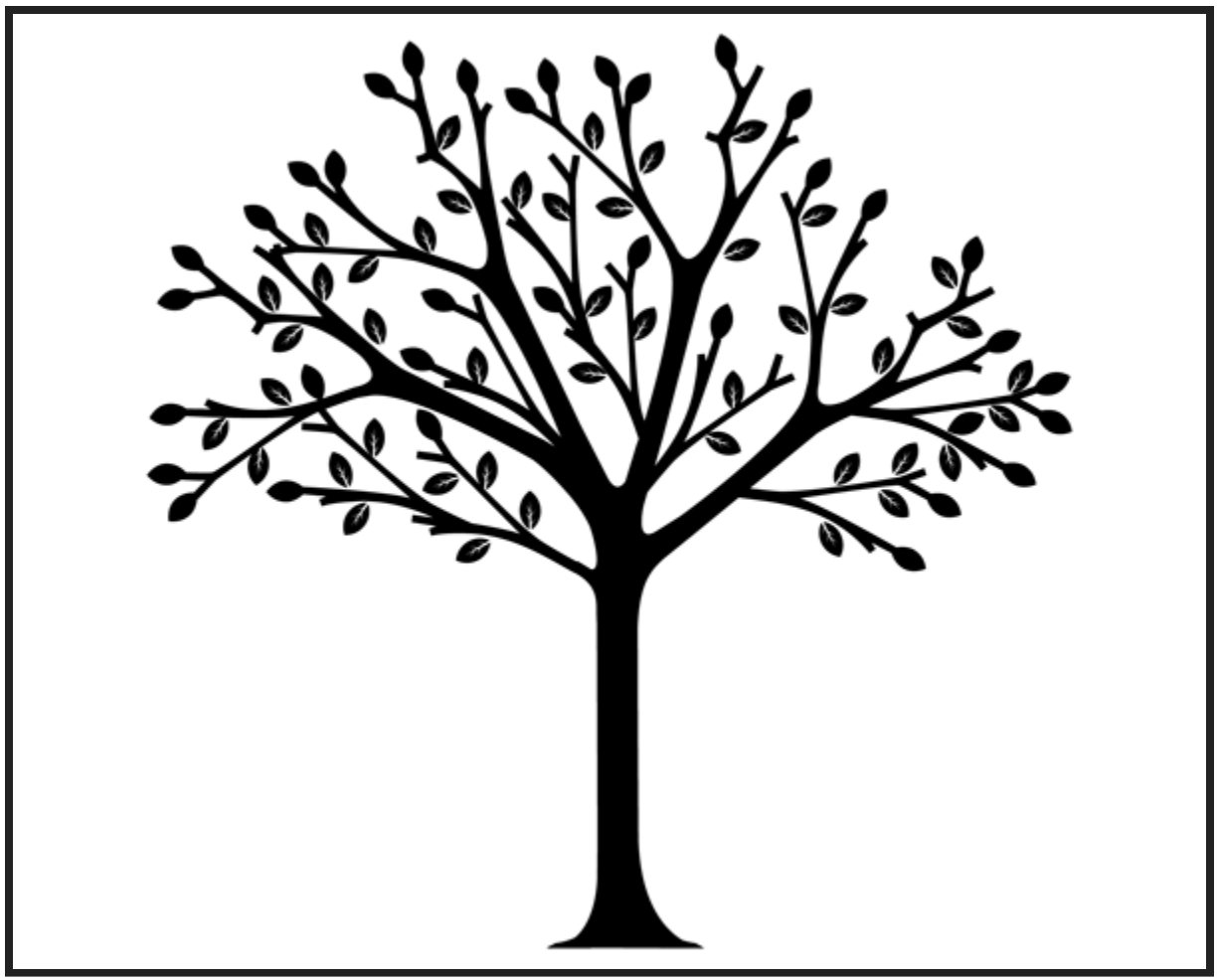
Функция - это алгоритм, который применяется к ***данным***. Данные не могут изменяться, только создаваться.

Переменных нет совсем, но есть **binding**. И состояния тоже нет!

ООП: ПРОГРАММА - ГРАФ



ФП: ПРОГРАММА - ДЕРЕВО



ФУНКЦИОНАЛЬНЫЕ ЯЗЫКИ

Чистые:

- Haskell
- Elm

Смешанные:

- Erlang
- Elixir
- Clojure

CLOJURE

- функциональный язык
- JVM
- LISP (все есть список!)

ЧИСТЫЕ ФУНКЦИИ

Функция может принимать аргументы и ***читать только*** их, а также вернуть ***новое*** значение

```
(def numbers '())
(concat numbers 3) ; => (3)
numbers ; => ()
```

Всегда один и тот же результат!

ФУНКЦИИ ВЫСШИХ ПОРЯДКОВ

Функция, которая может принимать функции как аргументы и возвращать функцию как результат называется **функцией высшего порядка**

Функция **map** принимает на вход функцию *f* и коллекцию *a*, результатом ее работы будет некоторая коллекция *b*, полученная в результате применения функции *f* к каждому из элементов коллекции *a*:

```
(map inc [0 1 2 3]) ; => (1 2 3 4)
```

Открываем фабрику инкрементаторов:

```
(defn inc-maker [inc-by] #(+ % inc-by))
(def inc3 (inc-maker 3))
(inc3 7) ; => 10
```

ДЕКОРИРОВАНИЕ

Каррирование - частичный вызов функции:

```
(def sum (partial reduce +))
```

Реализация функции partial для функции с двумя агрументами:

```
(defn my-partial [fun first]
  (fn [arg] (fun first arg)))

(def inc-all
  (my-partial map inc))

(inc-all '(1 2 3)) ; => (2 3 4)
```

Кэширование:

```
(defn memoize [f]
  (let [mem (atom {})]
    (fn [& args]
      (if-let [e (find @mem args)]
        (val e)
        (let [ret (apply f args)]
          (swap! mem assoc args ret)
          ret)))))

(def inc-cached (memoize inc))
(inc-cached 2) ; => 3
```

КОМПОЗИЦИЯ

Составление из простых функций сложной функции, которая передает свои аргументы одной из предоставленных функций, а каждый последующий результат передает в виде агрумента следующей функции, вызывая их в обратном порядке.

```clojure
(defn square [n] (* n n))

(defn square-sum [list]
  (reduce + (map #(square %) list))

(def square-sum
  (comp
    (partial reduce +)
    (partial map #(square %))))

(square-sum '(1 2 3)) ; => 14
```

ДЕСТРУКТУРИЗАЦИЯ

Установка соответствия между именами и элементами коллекции

```
(defn my-first [[first-thing]] first-thing)

(my-first ["oven" "bike" "war-axe"]) ; => "oven"
```

```clojure
(defn chooser [[first-choice second-choice & unimportant-choices]]
        (println (str "First choice: " first-choice))
        (println (str "Second choice: " second-choice))
        (println (str "Rest of choices: "
                      (clojure.string/join ", " unimportant-choices))))

(chooser ["Marmalade", "Handsome Jack", "Pigpen", "Aquaman"])
; => First choice: Marmalade
; => Second choice: Handsome Jack
; => Rest of choices: Pigpen, Aquaman
```

ЦИКЛЫ

```
(defn sum
  ([vals]
     (sum vals 0))
  ([vals accumulating-total]
     (if (empty? vals)
       accumulating-total
       (recur (rest vals) (+ (first vals) accumulating-total)))))
```

МУЛЬТИМЕТОДЫ

Разные реализации одного метода в зависимости от входных параметров

```
(defmulti full-moon-behavior
        (fn [were-creature] (:were-type were-creature)))

(defmethod full-moon-behavior :wolf
  [were-creature]
  (str (:name were-creature) " will howl and murder"))

(defmethod full-moon-behavior :simmons
  [were-creature]
  (str (:name were-creature) " will encourage people"))

(full-moon-behavior {:were-type :wolf :name "Rachel from next door"})
; => "Rachel from next door will howl and murder"
```

МАКРОСЫ

Макросы служат для метопрограммирования и манипуляций с кодом. Альтернативная реализация сложения двух чисел:

```
(defmacro infix [infixed]
  (list (second infixed) (first infixed) (last infixed)))

(infix (1 + 1)) ; => 2
```

ПРЕИМУЩЕСТВА ФП

- тестирование
- отладка
- многопоточность
- горячее обновление
- доказательные вычисления и оптимизация

КУДА ПОЙТИ ДАЛЬШЕ?

- Clojure for the brave and true
- 4clojure

Q?