Functional Programming Patterns

Abhinav Sarkar

Oct 2019

Flipkart

nilenso

Capillary

FICO

Complexity

Credit: flic.kr/p/d2xjKN

Complexity

- Complexity is the degree of entanglement in software.
- Complex programs are hard to understand and hence hard to change.

Essential Complexity

- Required for solving the problem at hand.
  - Like implementing Breadth-first search algorithm for finding the shortest path.
  - It is unavoidable regardless of which tools and frameworks you use.

Accidental Complexity

» Not inherent to the problem; brought while writing the program.

Accidental Complexity

- Not inherent to the problem; brought while writing the program.
  - Choose to use *C*? Now you have to manage memory too.

Accidental Complexity

- Not inherent to the problem; brought while writing the program.
  - Choose to use *C*? Now you have to manage memory too.
  - Choose to use *Java*? Now you have to model everything as objects.

Accidental Complexity

- Not inherent to the problem; brought while writing the program.
  - Choose to use *C*? Now you have to manage memory too.
  - Choose to use *Java*? Now you have to model everything as objects.
  - Choose to use *node*? Now you have to deal with 3 million dependencies.

Rube Goldberg Machine

Functional Programming

Credit: flic.kr/p/5V7C8S

Functional Programming

> Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

FP: Salient Aspects

» pure functions

FP: Salient Aspects

- pure functions
- first-class and higher-order functions

FP: Salient Aspects

- pure functions
- first-class and higher-order functions
- no/constrained mutable state

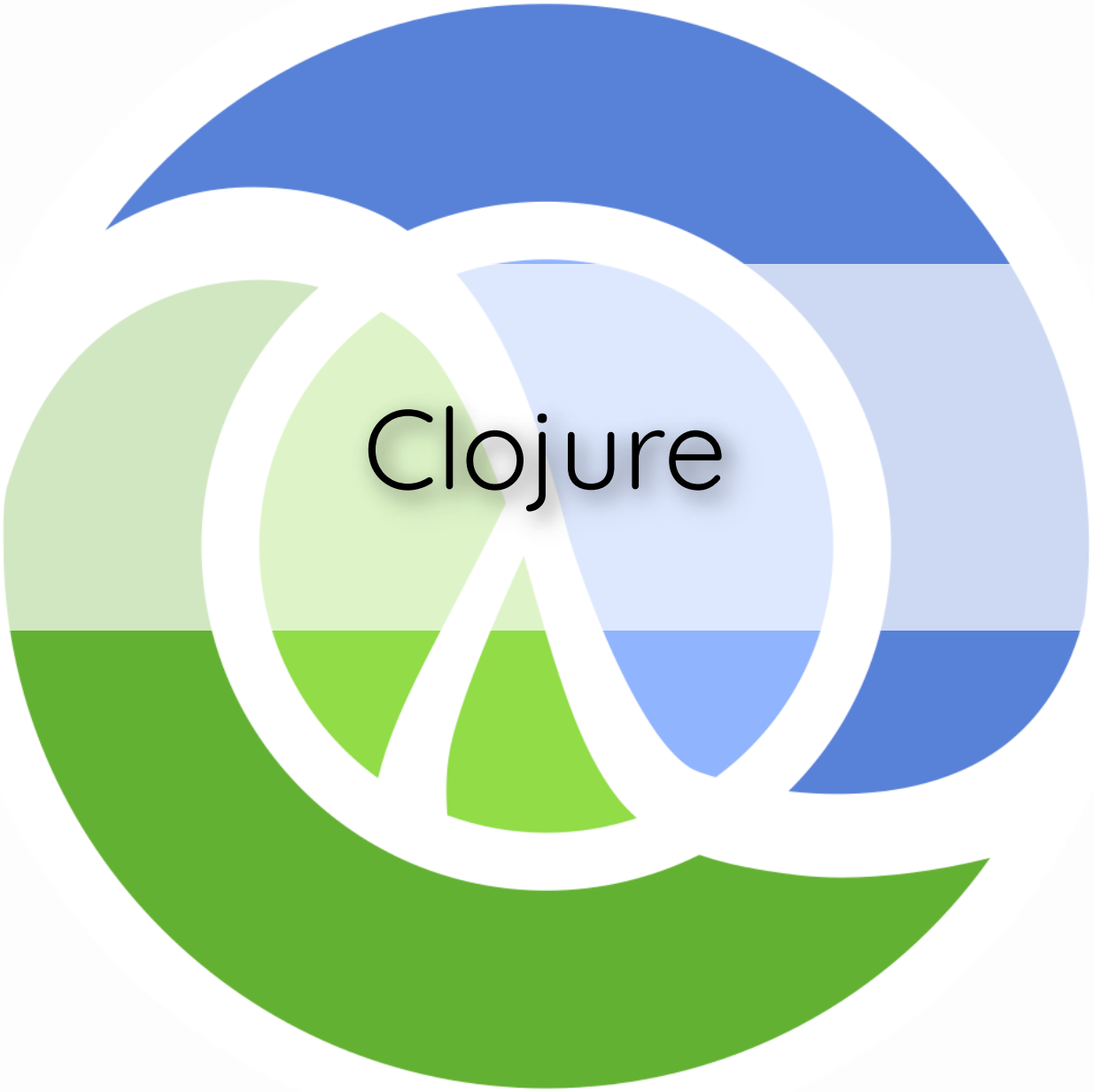
FP: Salient Aspects

- pure functions
- first-class and higher-order functions
- no/constrained mutable state
- no/constrained side-effects

FP: Salient Aspects

- pure functions
- first-class and higher-order functions
- no/constrained mutable state
- no/constrained side-effects
- optionally, lazy evaluation and static type system

```
let meetups =
  [{name: 'JS', isActive: true, members: 700},
   {name: 'Clojure', isActive: true, members: 900},
   {name: 'Java', isActive: false, members: 600},
   {name: 'Go', isActive: true, members: 500}];

let totalMembers = meetups
  .filter(m => m.isActive)
  .map(m => m.members)
  .reduce((acc, m) => acc + m, 0);
```


Clojure

Clojure

Clojure

» dynamically typed

Clojure

- dynamically typed
- functional programming language

Clojure

- dynamically typed
- functional programming language
- from the Lisp family

Clojure

- dynamically typed
- functional programming language
- from the Lisp family
- runs on JVM

Clojure

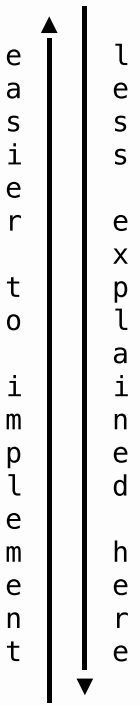
```clojure
(def meetups
  [{:name "JS" :is-active true :members 700}
   {:name "Clojure" :is-active true :members 900}
   {:name "Java" :is-active false :members 600}
   {:name "Go" :is-active true :members 500}])

(def total-members
  (->> meetups
    (filter :is-active)
    (map :members)
    (reduce (fn [acc m] (+ acc m)) 0)))
```

Functional Programming Patterns

Credit: flic.kr/p/3erjC9

Functional Programming Patterns

1. Immutable Data; Explicit State
2. Dataflow Programming
3. Explicit over Implicit
4. Data-oriented Programming
5. Functional Core; Imperative Shell



Immutable Data; Explicit State

Immutable Data

- Objects which once created cannot be changed.
- All domain objects should be immutable:
  - User, Account, Transaction, Event etc.

What?

Immutable Data

- No sneaky action-at-distance.
- Thread safety.
- Easier to reason about.
- GC is quite fast now. Creating objects is cheap.

Why?

Immutable Data

- In Clojure, just use built-in data structures like vector and map.
- In Java, use Google `@AutoValue` or Lombok `@Value` annotations.
- Clojure data structures share memory so they are more memory efficient.

How?

Immutable Data: Java

```java
import lombok.Value;
import lombok.With;

@Value
public class Person {
  private String name;
  @With private int age;
  private double score;
}

Person abhinav = new Person("abhinav", 30, 100.0);
String name = abhinav.getName();
Person olderAbhinav = abhinav.withAge(31);
```

Immutable Data: Clojure

```
=> (def abhinav {:name "Abhinav" :age 30 :score 100.0})
{:name "Abhinav", :age 30, :score 100.0}

=> (def name (:name abhinav))
"Abhinav"

=> (def older-abhinav (assoc abhinav :age 31))
{:name "Abhinav", :age 31, :score 100.0}
```

Explicit State

- Everything is immutable by default.
- Mutable state is marked explicitly so.
- Examples are thread pools, connection pools, dynamic configs, caches etc.

What?

Explicit State

- Mutability is constrained. You know the possible sources of sneaky action-at-distance.
- Proper thread safety implemented for these specific mutability constructs.
- Easier to reason about.

Why?

Explicit State

- In Clojure, use `atom`, `agent` or `ref`.
- In Java, use `AtomicReference` with immutable value, `Quasar` or `Akka` actors with inaccessible mutable state.
- VAVR library provides persistent DS like Clojure for Java.

How?

Explicit State: Clojure

```
=> (def abhinav (atom {:hair-color :black :age 33}))
#'user/abhinav
=> (swap! abhinav pass-time)
{:hair-color :gray, :age 34}
=> (reset! abhinav {:hair-color :none :age 33})
{:hair-color :none, :age 33}
=> @abhinav
{:hair-color :none, :age 33}
```

Explicit State: Clojure

```
=> (def abhinav (atom {:hair-color :black :age 33}))
#'user/abhinav
=> (swap! abhinav pass-time)
{:hair-color :gray, :age 34}
=> (reset! abhinav {:hair-color :none :age 33})
{:hair-color :none, :age 33}
=> @abhinav
{:hair-color :none, :age 33}
```

```
(defn pass-time [person]
   (-> person
      (assoc :hair-color :gray)
      (update :age inc)))
```

Explicit State: Java

```
class Person {
  private String hairColor;
  private int age;
  public Person(String hairColor, int age) {
    this.hairColor = hairColor;
    this.age = age;
  }
  public Person withAge(int age) {
    return new Person(this.hairColor, age);
  }
  public Person withHairColor(String hairColor) {
    return new Person(hairColor, this.age);
  }
  public Person passTime() {
    return this.withHairColor("Gray").withAge(this.age + 1);
  }
}
```

Explicit State: Java

```
AtomicReference<Person> abhinav =
      new AtomicReference(new Person("Black", 33));
abhinav.updateAndGet(Person::passTime);
abhinav.set(new Person(null, 33));
abhinav.get();
```

Dataflow Programming

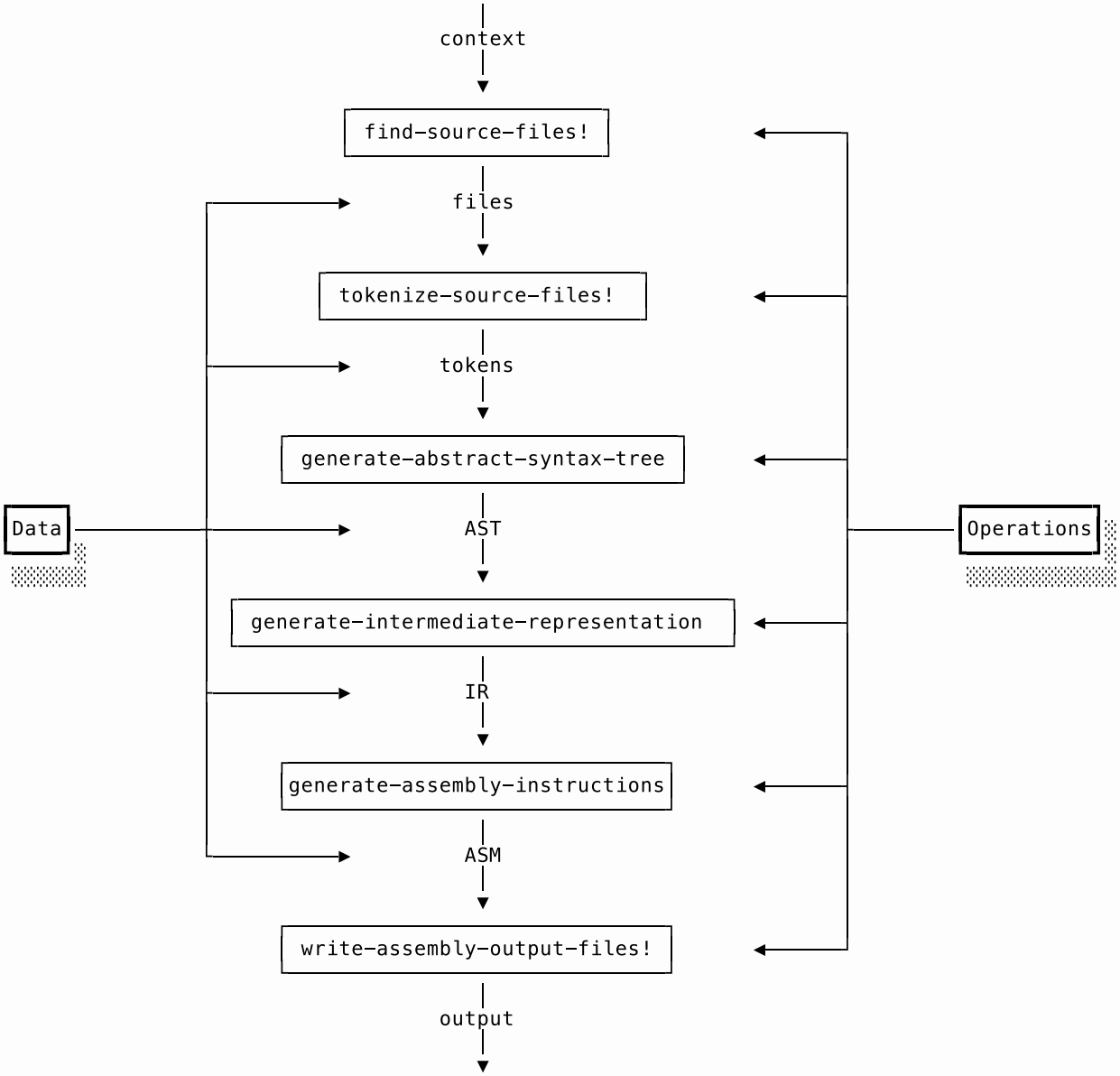
Dataflow Programming

» A programming paradigm that models a program as a directed graph of the data flowing between operations.

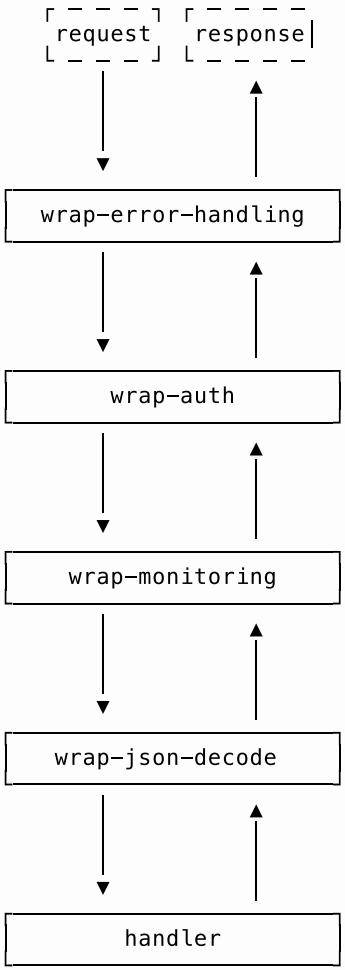
Dataflow Programming

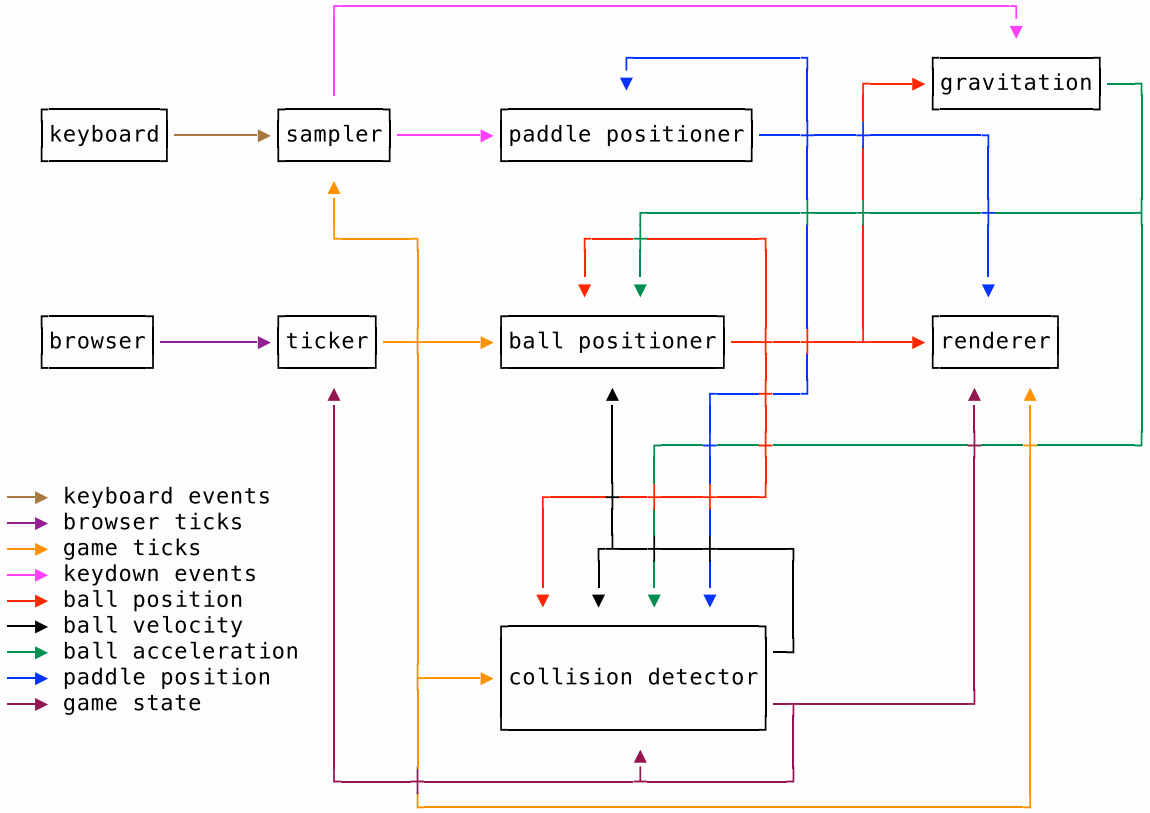
» A programming paradigm that models a program as a directed graph of the data flowing between operations.

```
public class Compiler {
  public void compile() {
    List<File> files = findSourceFiles();
    List<Token> tokens = tokenizeSourceFiles(files);
    AST ast = generateAbstractSyntaxTree(tokens);
    IR ir = generateIntermediateRepresentation(ast);
    ASM asm = generateAssemblyInstructions(ir);
    writeAssemblyOutputFiles(asm);
  }
}
```


context
find-source-files!
files
tokenize-source-files!
tokens
generate-abstract-syntax-tree
Data
AST
Operations
generate-intermediate-representation
IR
generate-assembly-instructions
ASM
write-assembly-output-files!
output

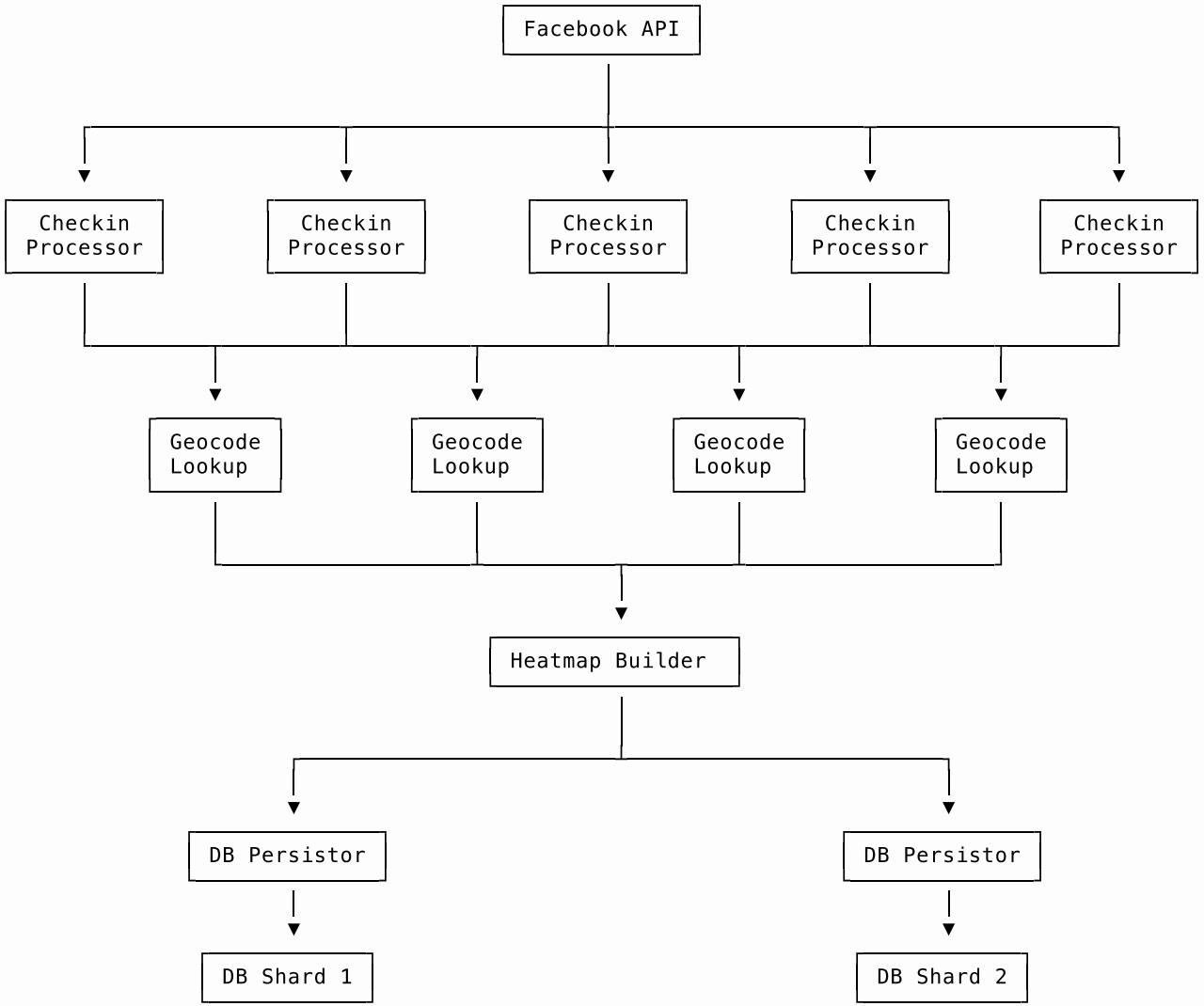
```
(defn compile! []
  (-> (find-source-files!)
      (tokenize-source-files!)
      (generate-abstract-syntax-tree)
      (generate-intermediate-representation)
      (generate-assembly-instructions)
      (write-assembly-output-files!)))
```


request
response
wrap-error-handling
wrap-auth
wrap-monitoring
wrap-json-decode
handler



frpong game dataflow diagram

Source: frpong


Facebook API
Checkin Processor
Checkin Processor
Checkin Processor
Checkin Processor
Checkin Processor
Geocode Lookup
Geocode Lookup
Geocode Lookup
Geocode Lookup
Heatmap Builder
DB Persistor
DB Persistor
DB Shard 1
DB Shard 2

Dataflow Programming

- In Java, use an Actor framework like Akka.
- Or roll your own using `ArrayBlockingQueue` and `ExecutorService`.
- In Clojure, use core.async.

Explicit over Implicit

Explicit over Implicit

- ❌ Dependency Injection
- ❌ Aspect Oriented Programming
- ❌ Hidden Code Generation
- ❌ Threadlocals
- ❌ Hidden Middlewares

Explicit over Implicit

- ❌ Dependency Injection
- ❌ Aspect Oriented Programming
- ❌ Hidden Code Generation
- ❌ Threadlocals
- ❌ Hidden Middlewares

- ✓ Higher-order functions
- ✓ Macros
- ✓ Explicit stateful components
- ✓ Passing context as function parameter

Explicit over Implicit

Replace

```
@Transactional
void transferMoney(Account from, Account to, Amount amount) {
  accountDAO.subtract(from, amount);
  accountDAO.add(to, amount);
}
```

Explicit over Implicit

Replace

```
@Transactional
void transferMoney(Account from, Account to, Amount amount) {
  accountDAO.subtract(from, amount);
  accountDAO.add(to, amount);
}
```

with

```
(defn transfer-money [from to amount]
  (java.jdbc/with-db-transaction [txn db-conn-pool]
    (account-dao/subtract txn from amount)
    (account-dao/add txn to amount)))
```

Explicit over Implicit

Replace

```
@Timed(.. some setting ...)
void importantBusinessFunction() { .. }
```

Explicit over Implicit

Replace

```
@Timed(.. some setting ...)
void importantBusinessFunction() { .. }
```

with

```
(defn important-business-function []
  (timed
    {:some-settings ...}
    (do-something-important)))
```

Explicit over Implicit

Replace

```
@RequestMapping(value = "/{org_nr}/devices")
@Authorization(
        resource = @TargetResource(
          type = ORGANIZATION, identifiedBy = "org_nr"),
        requiresPermission = ANY_ROLE,
        requiresAuthenticationLevel = TWO_FACTOR
)
List<Devices> getDevices(@RequestParam("org_nr") String orgNr) { .. }
```

Explicit over Implicit

Replace

```
@RequestMapping(value = "/{org_nr}/devices")
@Authorization(
        resource = @TargetResource(
          type = ORGANIZATION, identifiedBy = "org_nr"),
        requiresPermission = ANY_ROLE,
        requiresAuthenticationLevel = TWO_FACTOR
)
List<Devices> getDevices(@RequestParam("org_nr") String orgNr) { .. }
```

with

```
(GET "/:org_nr/devices" [org-nr :as req]
  (check-auth
    {:target [:organization org-nr]
     :role :any
     :level :two-factor}
     req
    #(get-devices org-nr))))
```

Explicit over Implicit

```
(def handler
  (-> routes
    (make-handler)
    (wrap-swagger-ui "/swagger-ui")
    (wrap-defaults api-defaults)
    (wrap-json-params)
    (wrap-json-response)
    (wrap-monitoring)
    (wrap-errors)))

(run-jetty handler {:port 3000})
```

Data-oriented Programming

Data-oriented Programming

- Data is easy to manipulate.
- Data is easy to extend.
- Data can be interpreted in different ways.
- Data can be saved into DB/disk or sent over network.

Why?

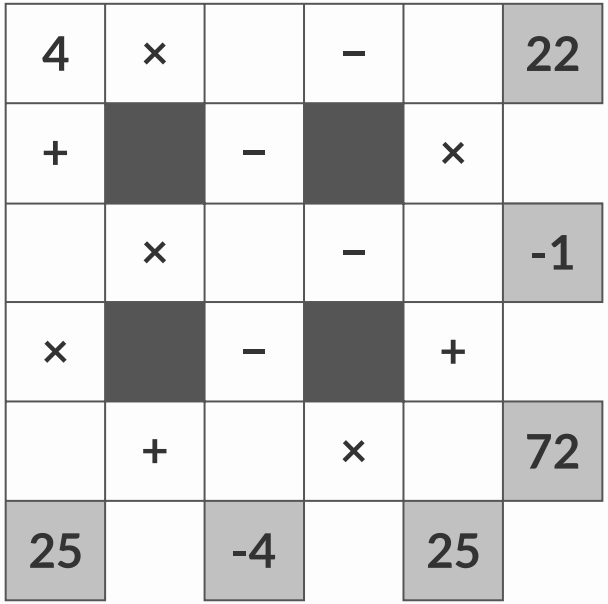
Data as Control Flow

» Dispatch table

```
(defn routes []
  ["/" [["api/" [["v1/" {"ping"  ping
                         "orgs"  {["/" :id] {:get org/get}}
                         "users" {"/me" {:get    user/me
                                         :delete user/logout}}}]
                 [true not-found-handler]]]
        ["oauth/" {[:subdomain ""]          oauth/launch
                   [:subdomain "/callback"] oauth/callback}]
        [true default-site-handler]]])
```

Data as Control Flow

» Logic Programming



```clojure
(ns logic.core
  (:refer-clojure :exclude [==])
  (:require [clojure.core.logic :refer :all])
  (:require [clojure.core.logic.fd :as fd]))

;; Use run* to retrieve all possible solutions
(run* [q]
  ;; Create some new logic vars (lvars) for us to use in our rules
  (fresh [a0 a1 a2  ;; Top row
          b0 b1 b2  ;; Middle row
          c0 c1 c2] ;; Bottom row
    ;; Unify q with our lvars in the output format we want
    (== q [[a0 a1 a2]
           [b0 b1 b2]
           [c0 c1 c2]])
    ;; State that every one of our lvars should be in the range 1-9
    (fd/in a0 a1 a2 b0 b1 b2 c0 c1 c2 (fd/interval 1 9))
    ;; State that each of our lvars should be unique
    (fd/distinct [a0 a1 a2 b0 b1 b2 c0 c1 c2])
    ;; fd/eq is just a helper to allow us to use standard Clojure
    ;; operators like + instead of fd/+
    (fd/eq
      ;; Horizontal conditions for the puzzle
      (= (- (* a0 a1) a2) 22)
      (= (- (* b0 b1) b2) -1)
      (= (+ (* c0 c1) c2) 72)
      ;; Vertical conditions for the puzzle
      (= (* (+ a0 b0) c0) 25)
      (= (- (- a1 b1) c1) -4)
      (= (+ (* a2 b2) c2) 25)
      ;; And finally, in the puzzle we are told that the top left
      ;; number (a0) is 4.
      (= a0 4))))
```

Source: [Using Clojure's core.logic to Solve Simple Number Puzzles]

Data as Instructions

» Rule engines

Data as Instructions

» Interpreters

Raga

Has a name

Rule to ascend

Rule to descend

ot necessarily symmetric

Not necessarily linear

Grouped into families

```
{{:name :suryakantam,
  :num 17,
  :arohanam [:s :r1 :g3 :m1 :p :d2 :n3 :s.],
  :avarohanam (:s. :n3 :d2 :p :m1 :g3 :r1 :s)}
 {:sahuli
  {:arohanam (:s :g3 :m1 :p :n3 :s.),
   :avarohanam (:s. :n3 :d2 :p :m1 :g3 :r1 :s)}
  :shuddhalalita
  {:arohanam (:s :r1 :g3 :m1 :d2 :n3 :s.),
   :avarohanam (:s. :n3 :d2 :m1 :g3 :r1 :s)},
  :shuddhagandharvam
  {:arohanam (:s :r1 :g3 :m1 :p :d2 :n3 :s.),
   :avarohanam (:s. :d2 :p :m1 :r1 :s)},
  :kanakacala
  {:arohanam (:s :g3 :p :n3 :s.),
   :avarohanam (:s. :d2 :m1 :r1 :s)},
  :kanthiravam
  {:arohanam (:s :r1 :g3 :p :d2 :n3 :s.),
   :avarohanam (:s. :n3 :d2 :p :m1 :g3 :r1 :s)}
  :kannal
  {:arohanam (:s :r1 :g3 :m1 :p :d2 :n3 :s.),
   :avarohanam (:s. :d2 :p :m1 :r1 :s)},
  ...
```

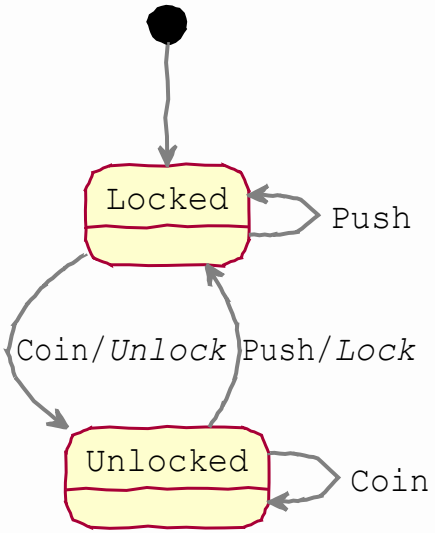
Source: "Making machines that make music" by Srihari Sriraman

Data as Instructions

» Codegen from data

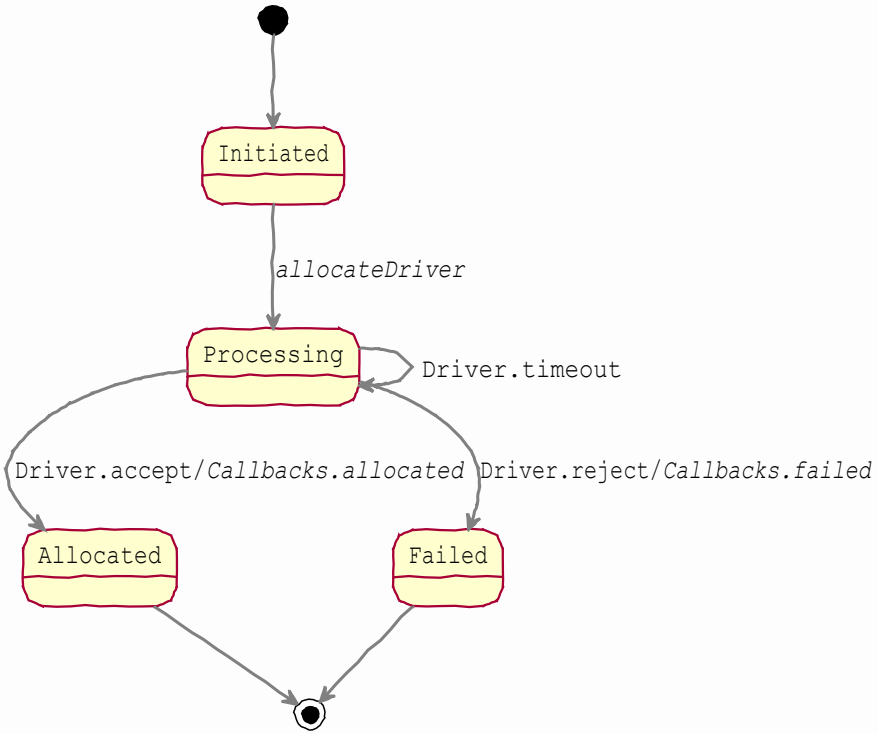
```
(def q-sqlmap {:select [:foo/a :foo/b :foo/c]
               :from   [:foo]
               :where  [:= :foo/a "baz"]})
(sql/format q-sqlmap :namespace-as-table? true)
=> ["SELECT foo.a, foo.b, foo.c FROM foo WHERE foo.a = ?" "baz"]
```

Data as Finite State Machine



A turnstile's state-machine

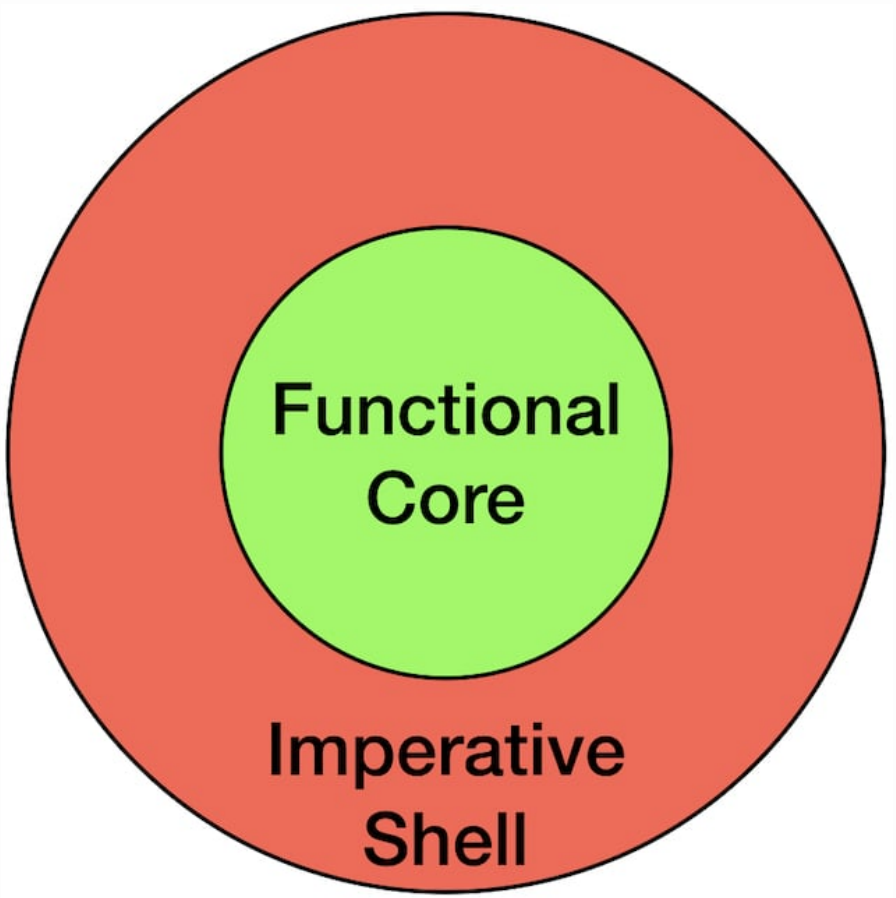
Data as Finite State Machine



Driver allocation state machine

Functional Core; Imperative Shell

Functional Core; Imperative Shell

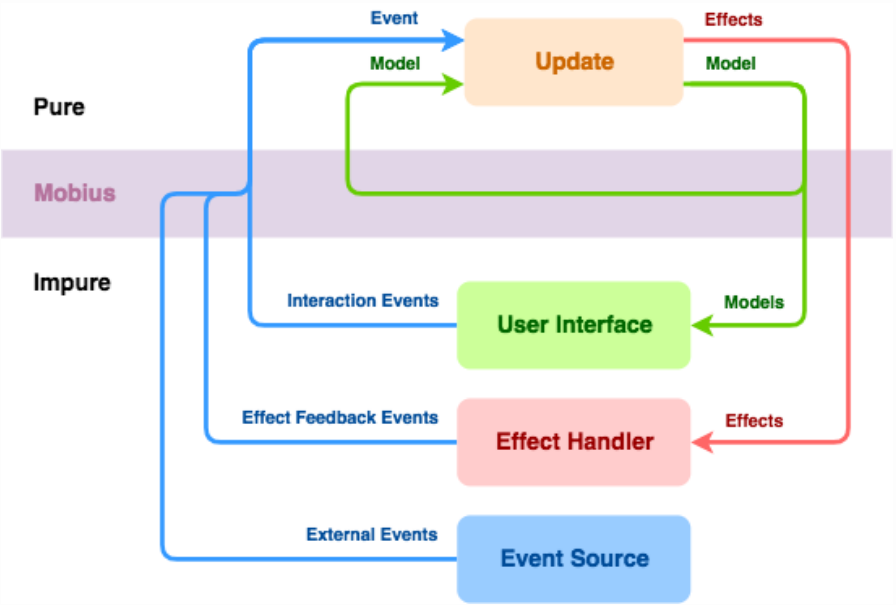


Functional Core

- only pure functions
- no side-effects
- as many decisions as possible
- easier to test and to change
- make as big as possible

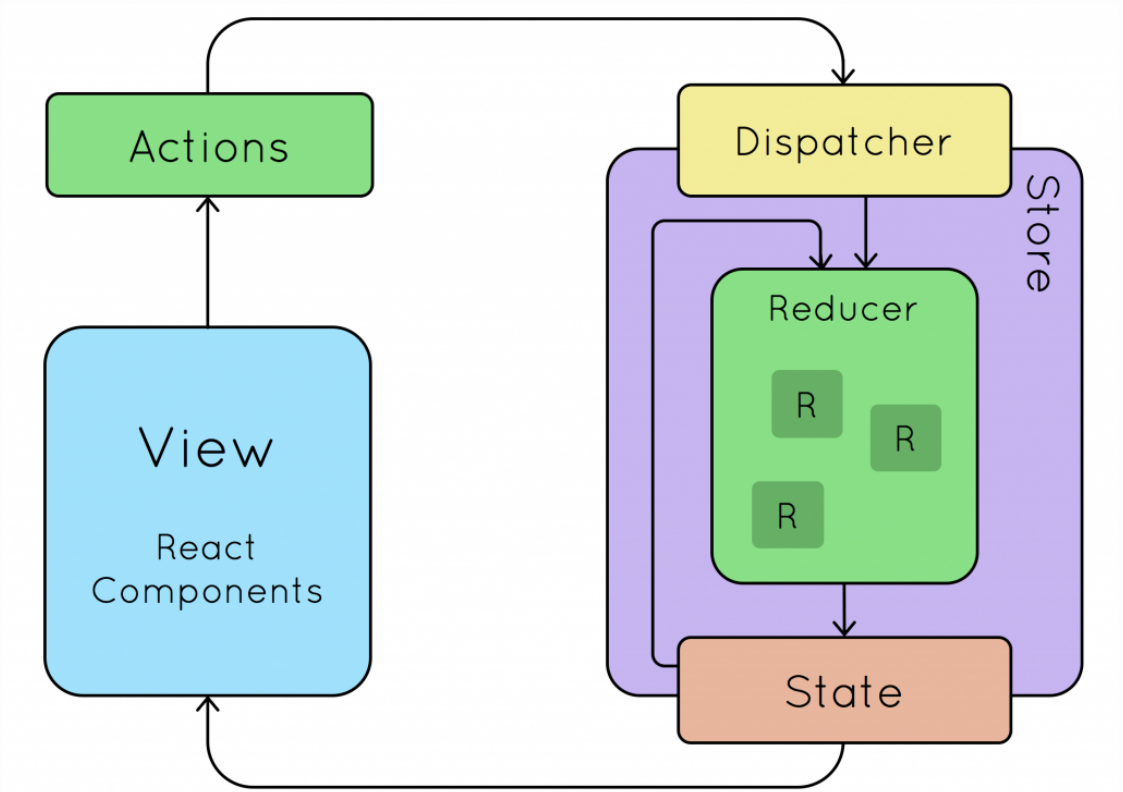
Imperative Shell

- only side effects
  - error handling
  - state
  - IO
- as few conditionals or decisions as possible
- harder to test
- make as thin as possible



Mobius Architecture

Source: [Mobius Wiki]



Redux Architecture

Source: Building modern web apps

Simple Made Easy

Rich Hickey

References

- No Silver Bullet —Essence and Accident in Software Engineering
- Why Functional Programming Matters
- Translating an Enterprise Spring Webapp to Clojure
- Java with a Clojure Mindset
- Functional Core; Imperative Shell
- Simple Made Easy

Thanks

@abhin4v

abnv.me/fpp