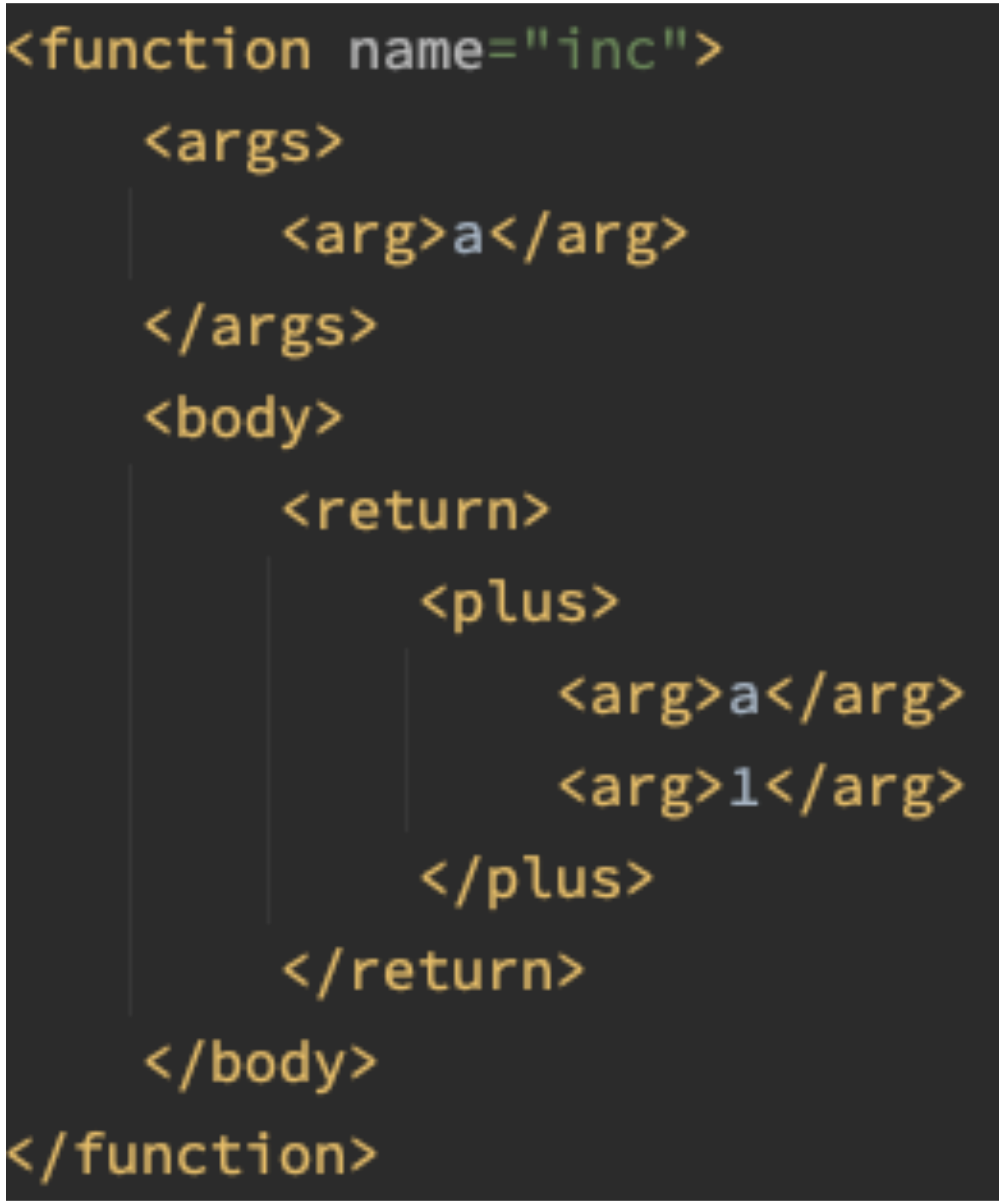
Functional Programming in Java vs. Clojure

Get rid of the state

Lukáš Rychtecký 28.04.2022

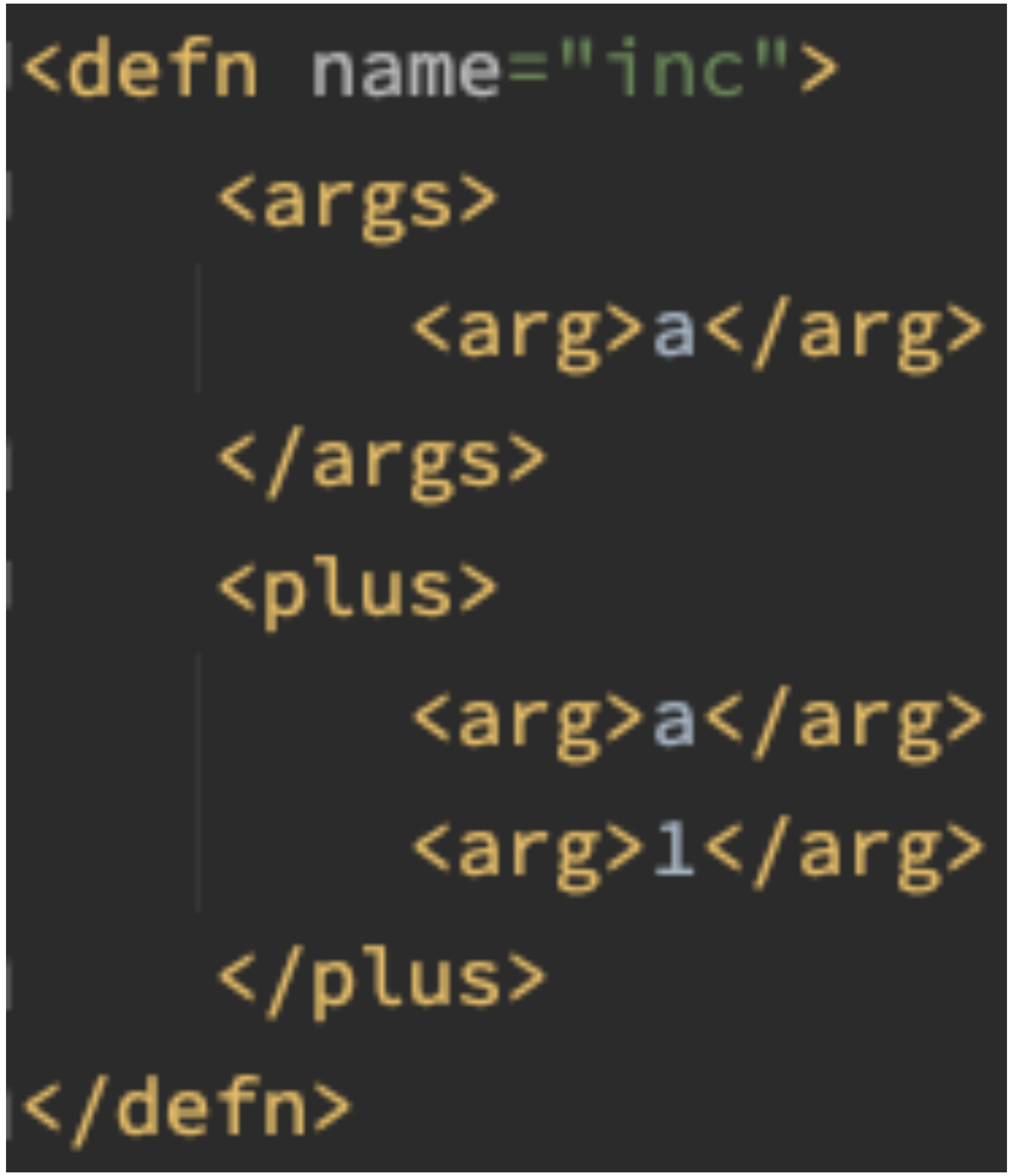
Applifting

Good old XML!



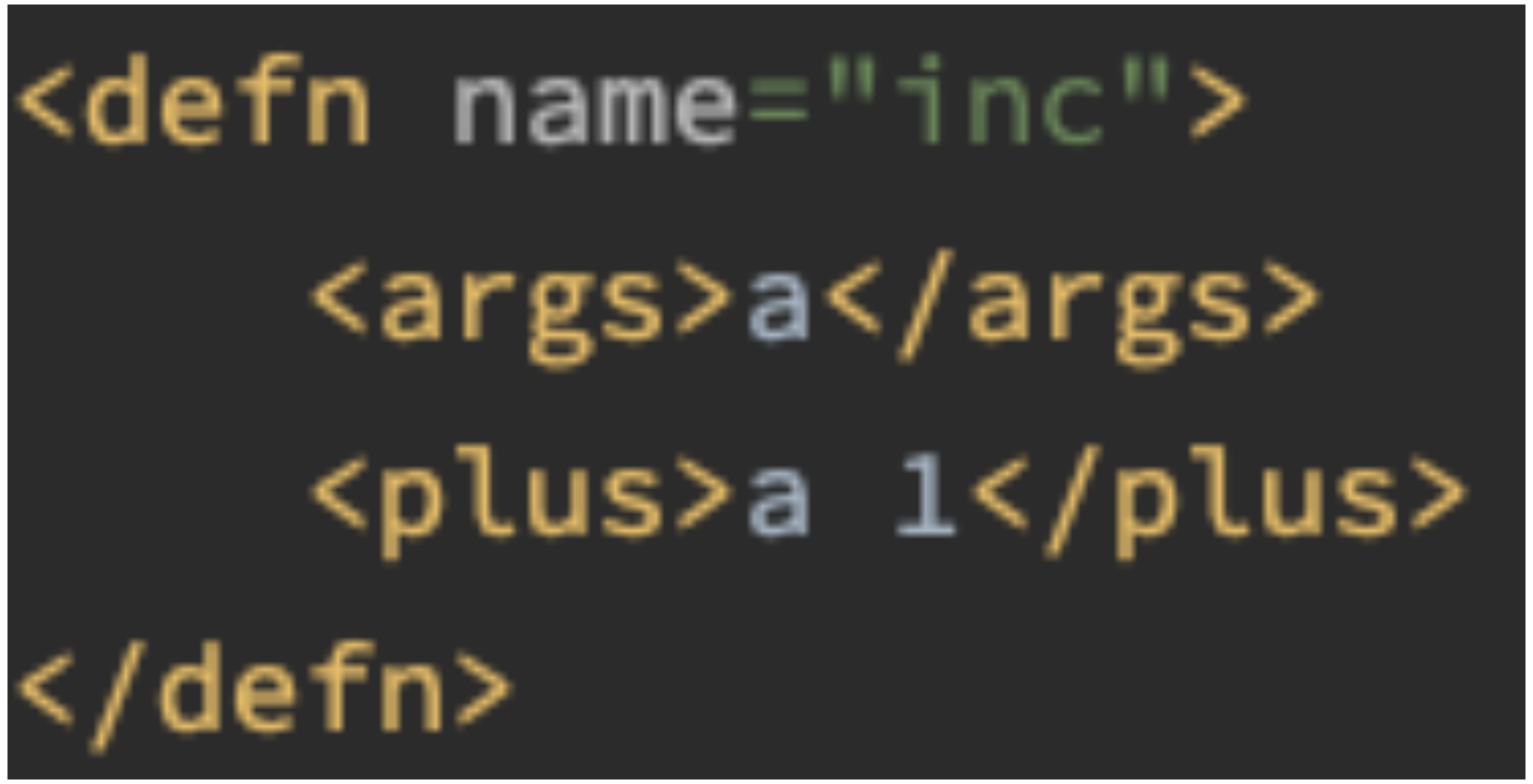
```
<function name="inc">
    <args>
        <arg>a</arg>
    </args>
    <body>
        <return>
            <plus>
                <arg>a</arg>
                <arg>1</arg>
            </plus>
        </return>
    </body>
</function>
```

Let’s reduce some noise



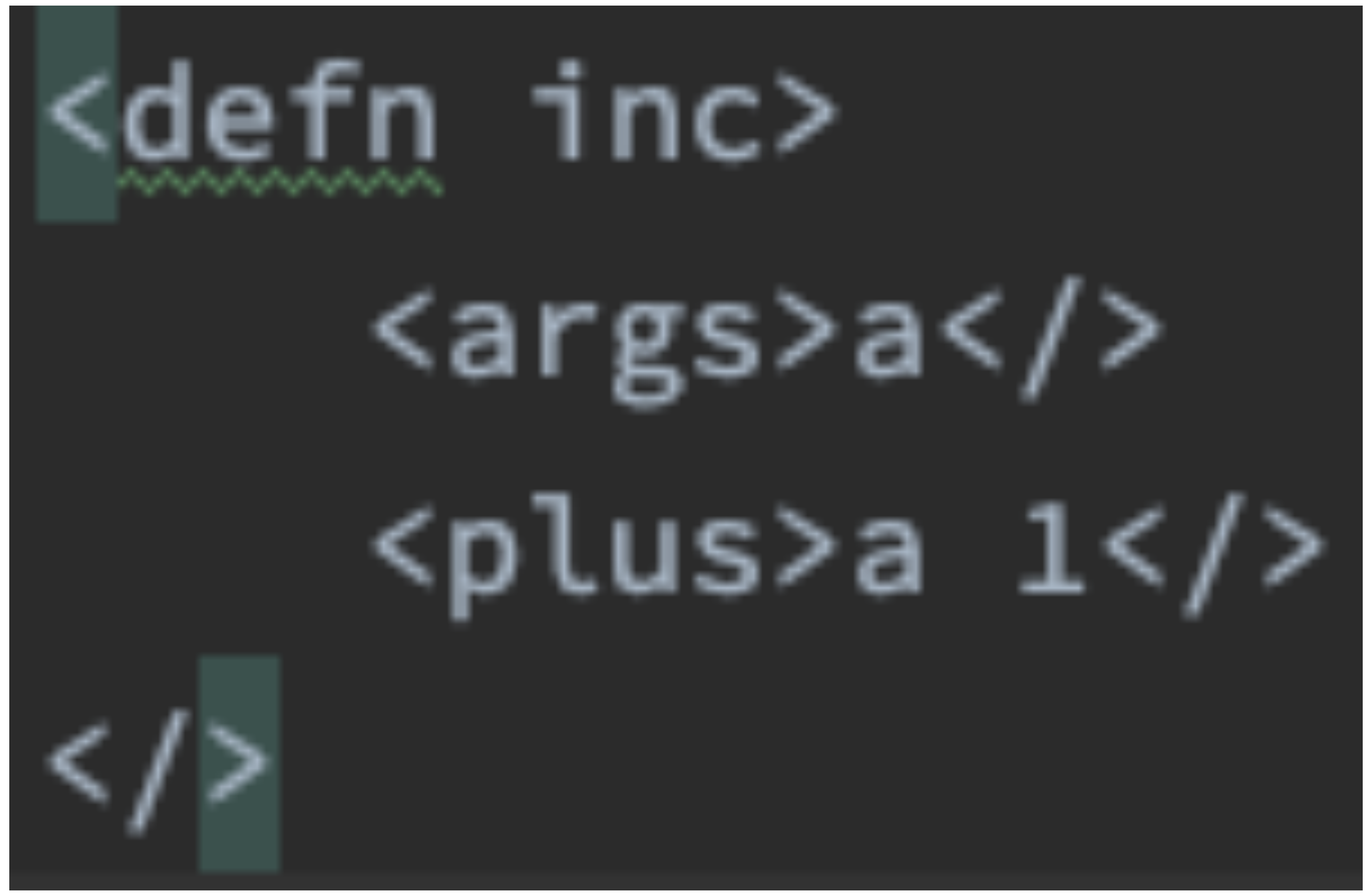
```
<defn name="inc">
    <args>
        <arg>a</arg>
    </args>
    <plus>
        <arg>a</arg>
        <arg>1</arg>
    </plus>
</defn>
```

… and more



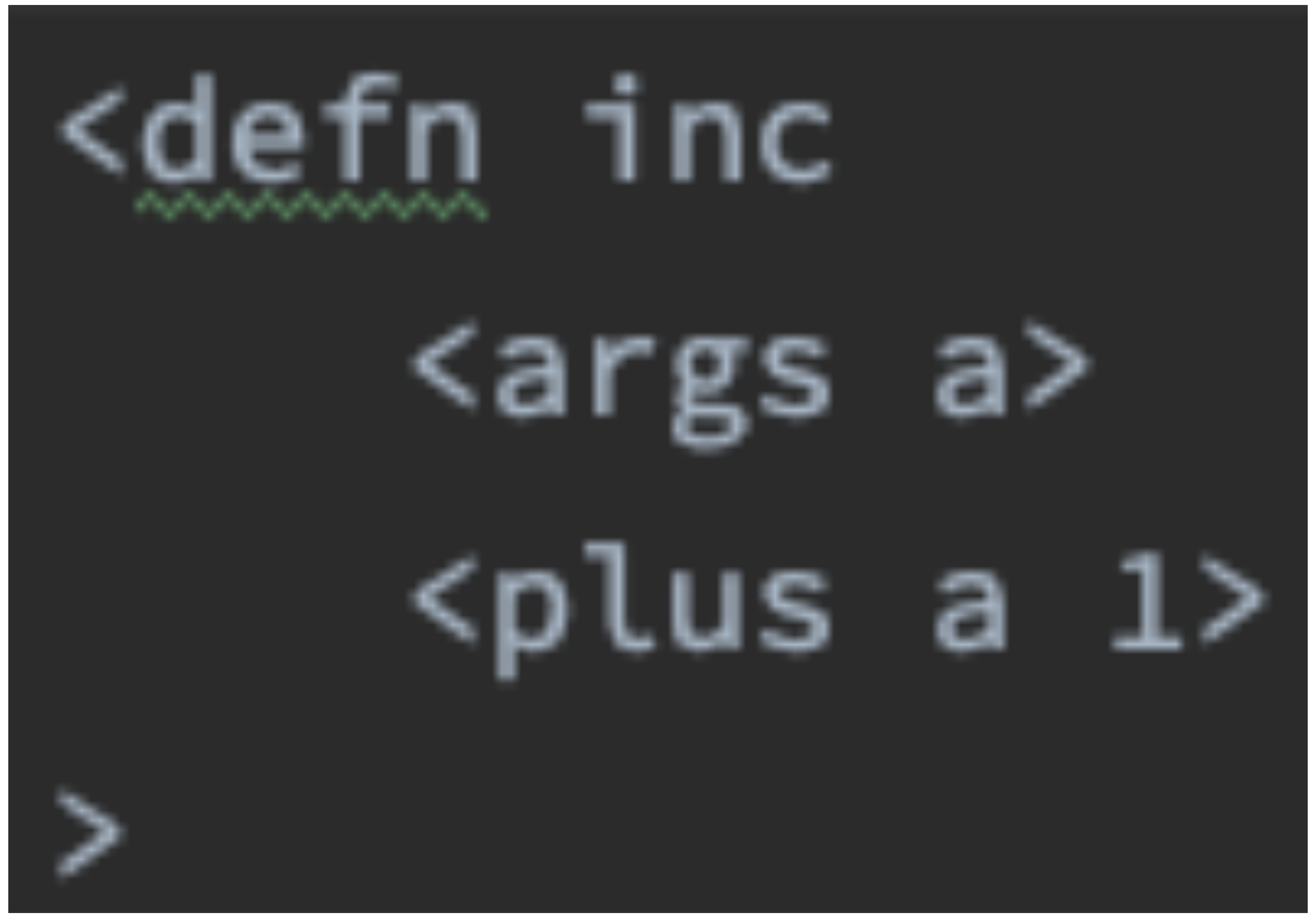
```
<defn name="inc">
    <args>a</args>
    <plus>a 1</plus>
</defn>
```

… even more



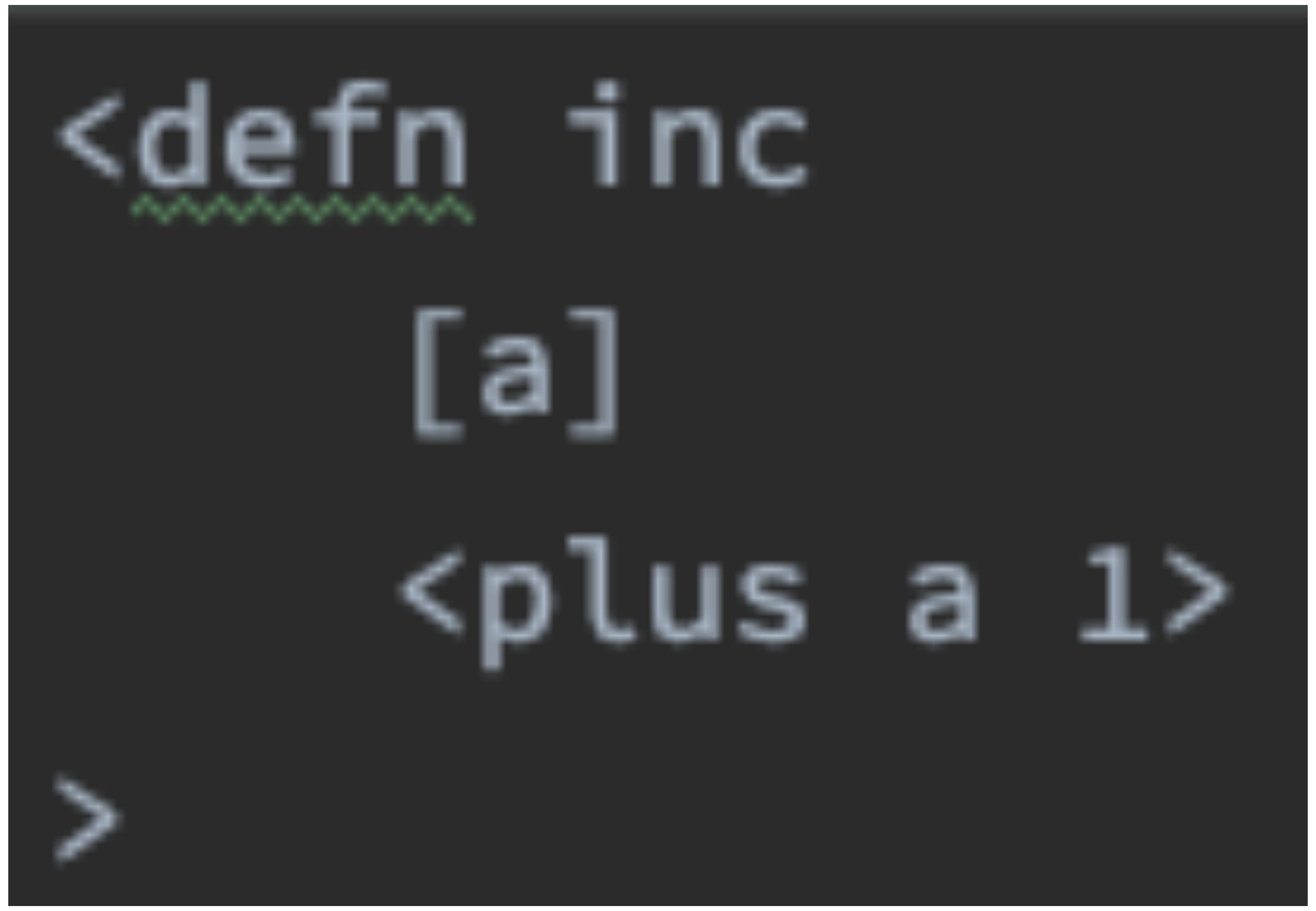
```
<defn inc>
    <args>a</>
    <plus>a 1</>
</>
```

... even even more



```
<defn inc
    <args a>
    <plus a 1>
>
```

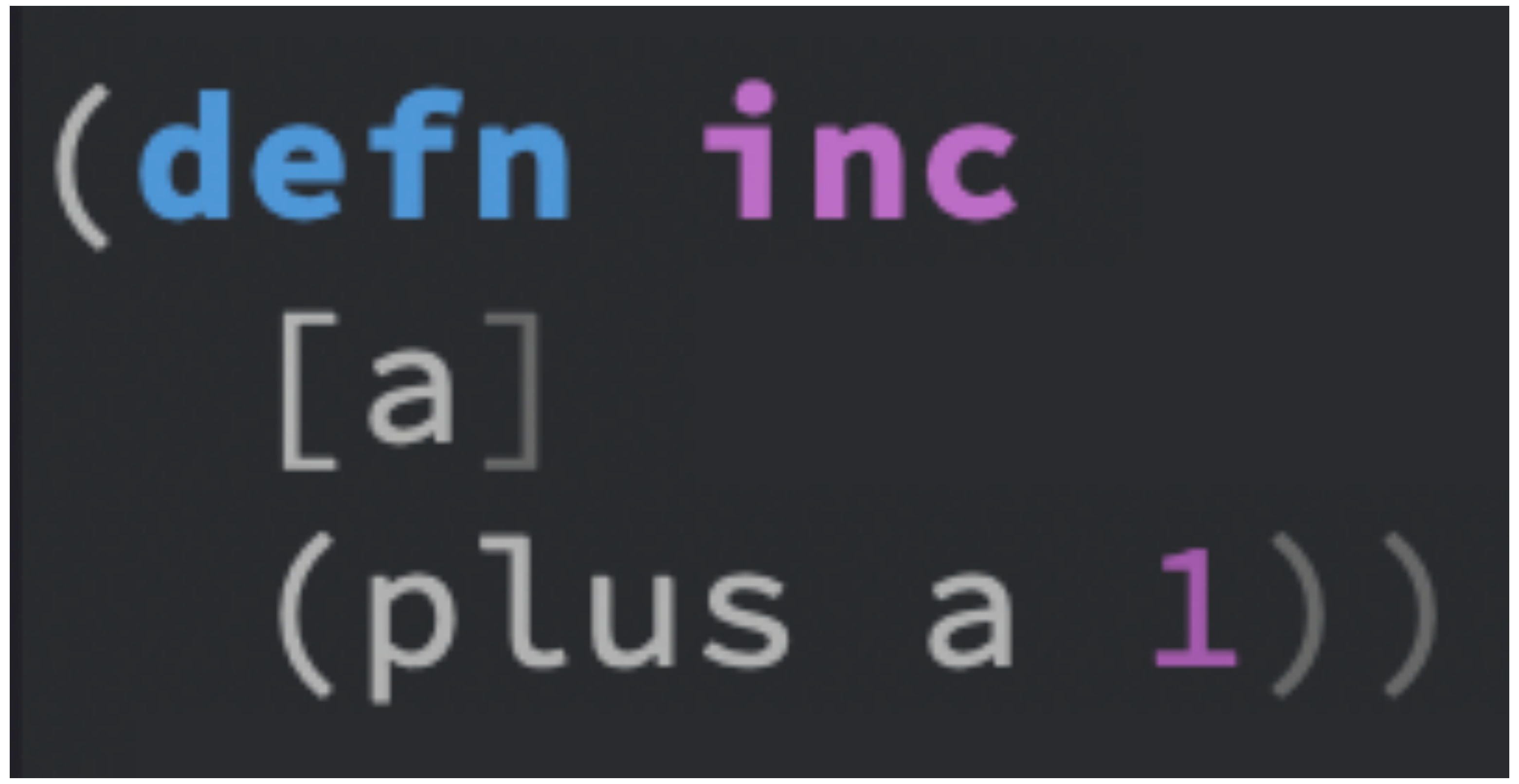
... brace your self, parens coming



```
<defn inc
    [a]
    <plus a 1>
>
```

A voilà!

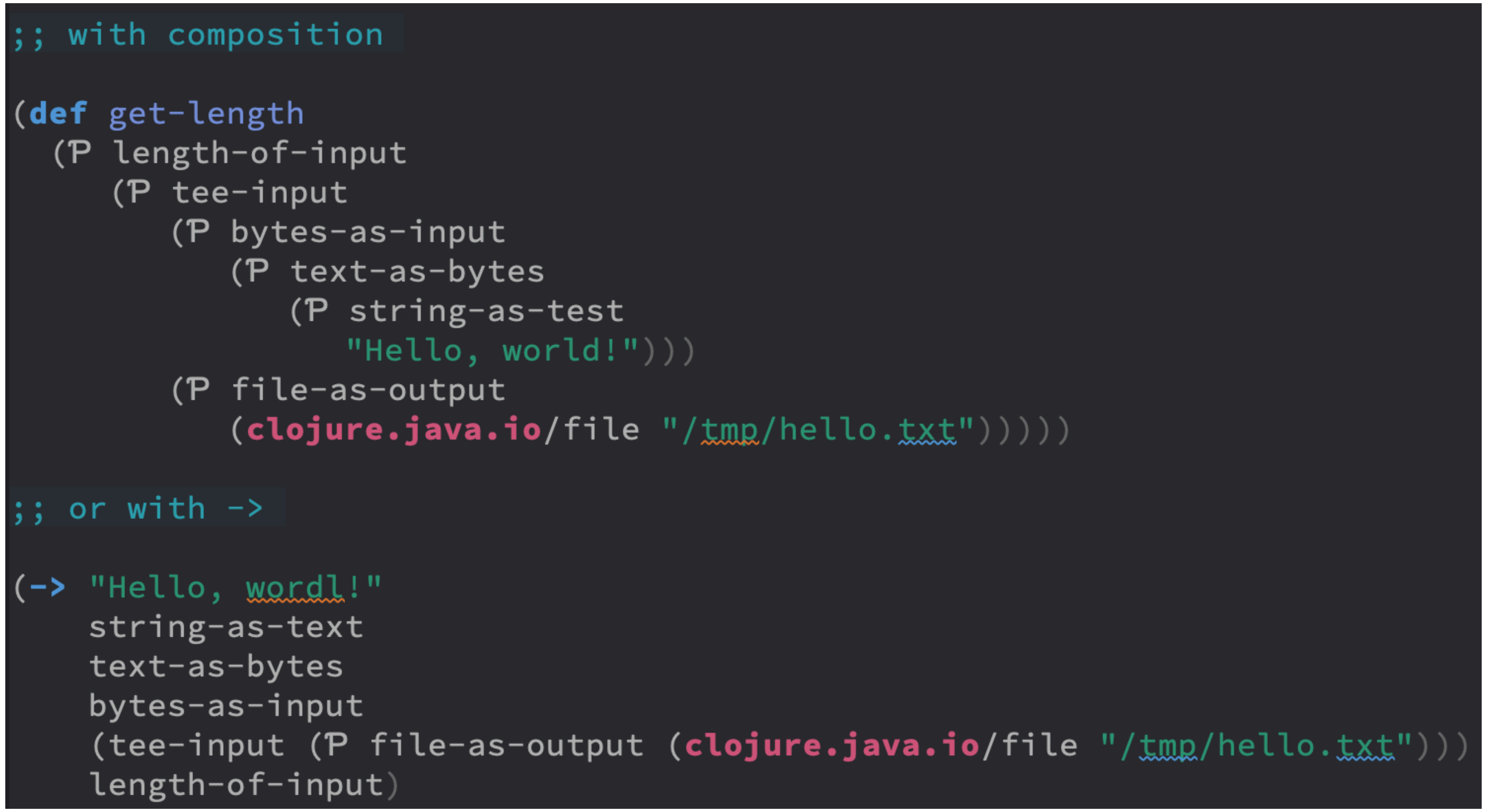
Pure gold



```
(defn inc
  [a]
  (plus a 1))
```

Single Responsibility Principle

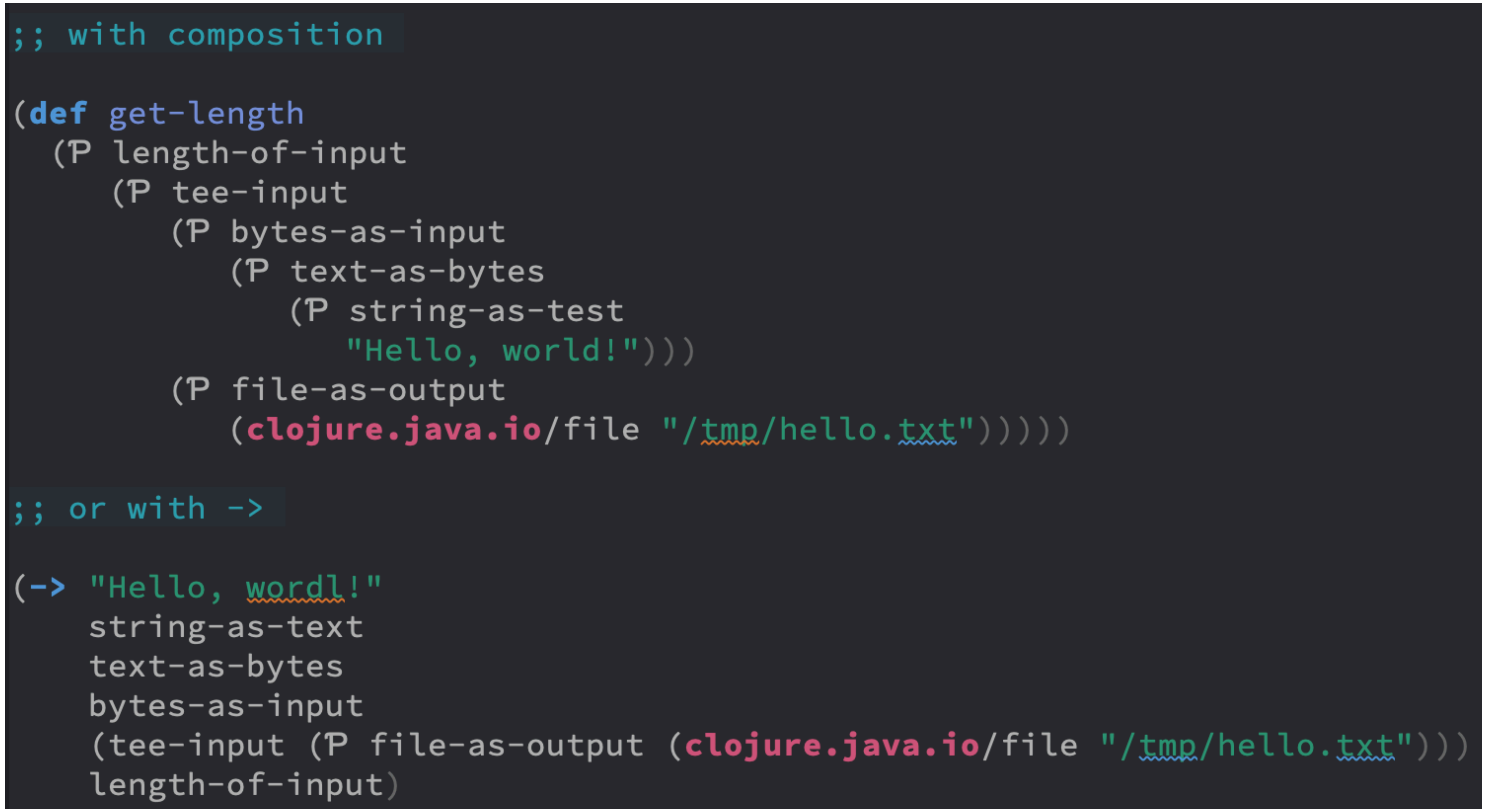
```
new LengthOfInput(
  new TeeInput(
    new BytesAsInput(
      new TextAsBytes(
        new StringAsText(
          "Hello, world!"
        )
      )
    ),
    new FileAsOutput(
      new File("/tmp/hello.txt")
    )
  )
)
```



```
;; with composition

(def get-length
  (𝒫 length-of-input
     (𝒫 tee-input
        (𝒫 bytes-as-input
           (𝒫 text-as-bytes
              (𝒫 string-as-test
                 "Hello, world!")))
        (𝒫 file-as-output
           (clojure.java.io/file "/tmp/hello.txt")))))

;; or with ->

(-> "Hello, wordl!"
    string-as-text
    text-as-bytes
    bytes-as-input
    (tee-input (𝒫 file-as-output (clojure.java.io/file "/tmp/hello.txt")))
    length-of-input)
```

Open-Closed Principle

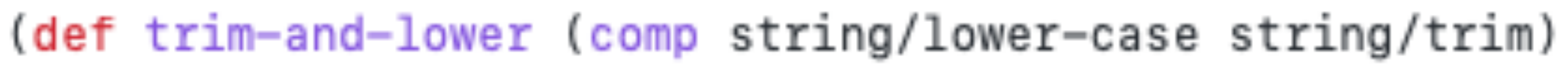
```
new LengthOfInput(
  new TeeInput(
    new BytesAsInput(
      new TextAsBytes(
        new StringAsText(
          "Hello, world!"
        )
      )
    ),
    new FileAsOutput(
      new File("/tmp/hello.txt")
    )
  )
)
```



```
;; with composition

(def get-length
  (Ƥ length-of-input
     (Ƥ tee-input
        (Ƥ bytes-as-input
           (Ƥ text-as-bytes
              (Ƥ string-as-test
                 "Hello, world!")))
        (Ƥ file-as-output
           (clojure.java.io/file "/tmp/hello.txt")))))

;; or with ->

(-> "Hello, wordl!"
    string-as-text
    text-as-bytes
    bytes-as-input
    (tee-input (Ƥ file-as-output (clojure.java.io/file "/tmp/hello.txt")))
    length-of-input)
```

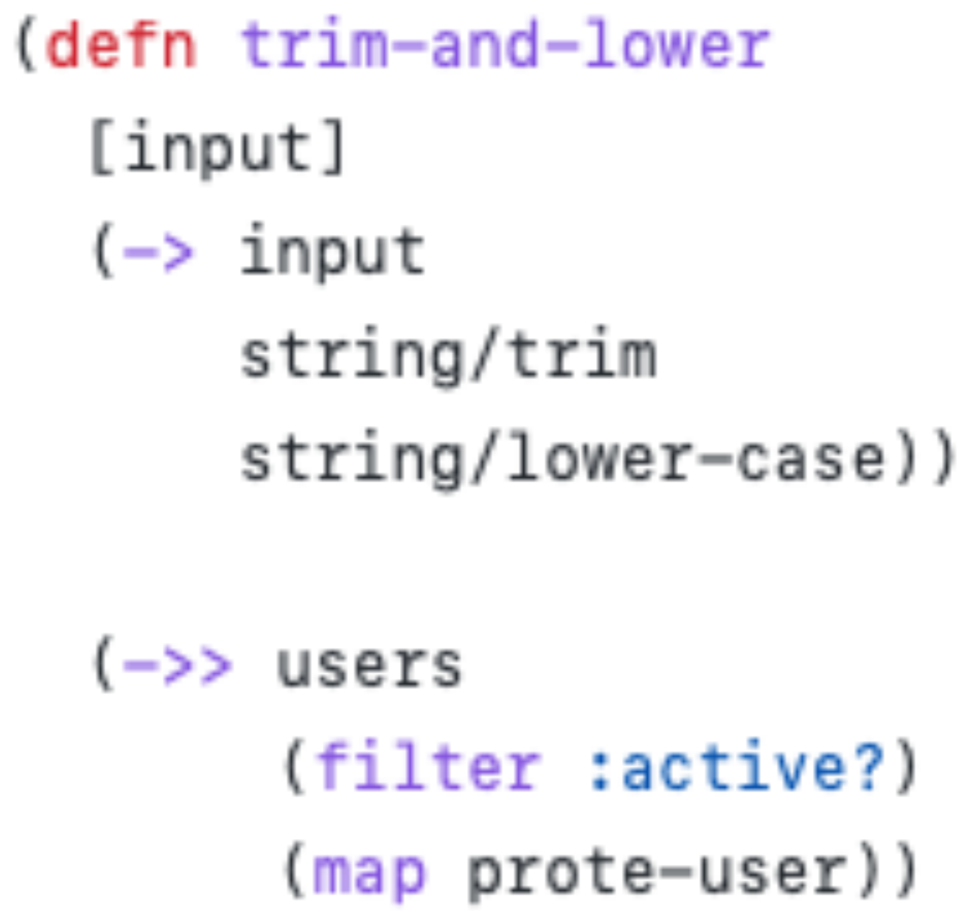
Tools for composition

- comp



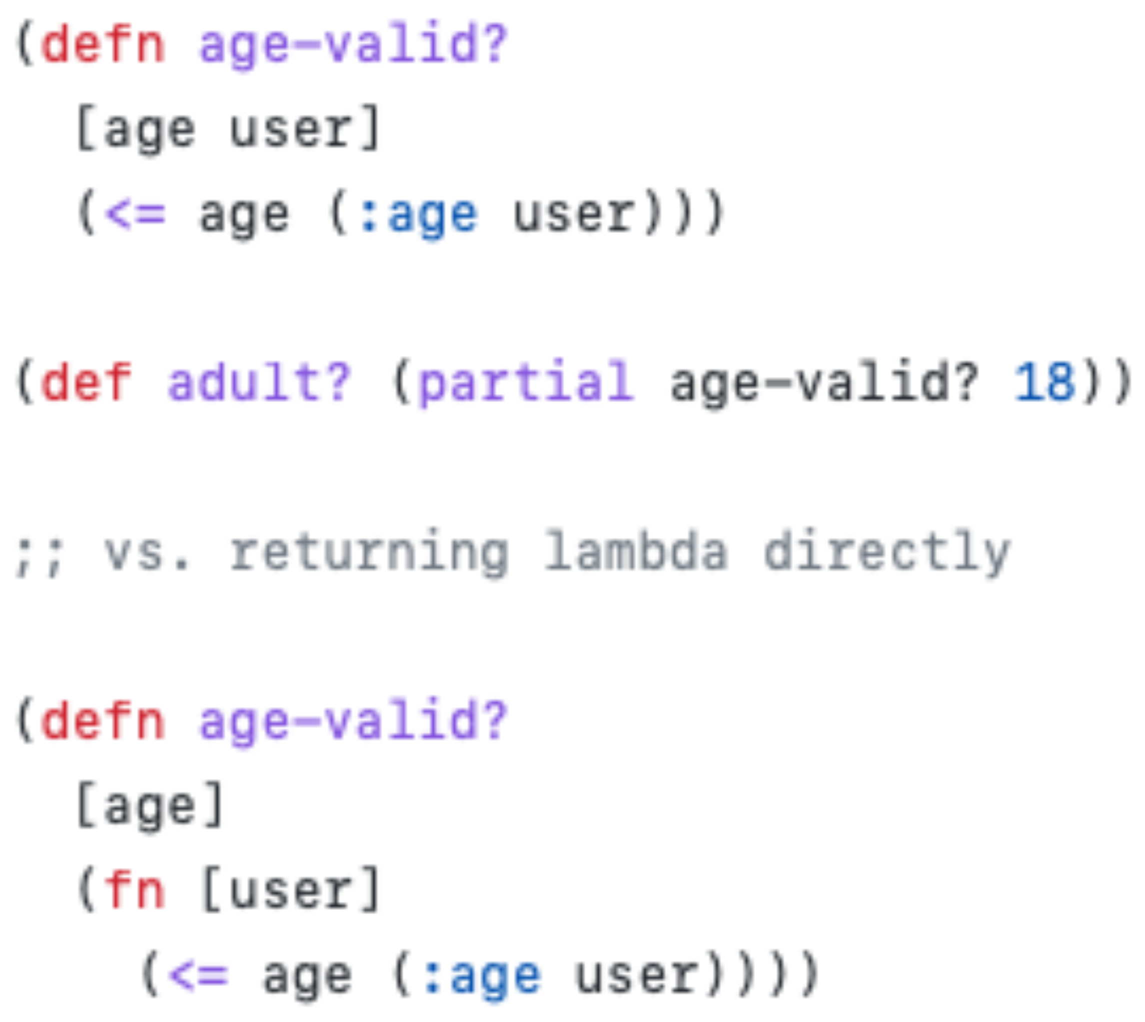
```
(def trim-and-lower (comp string/lower-case string/trim))
```

- ->, ->>



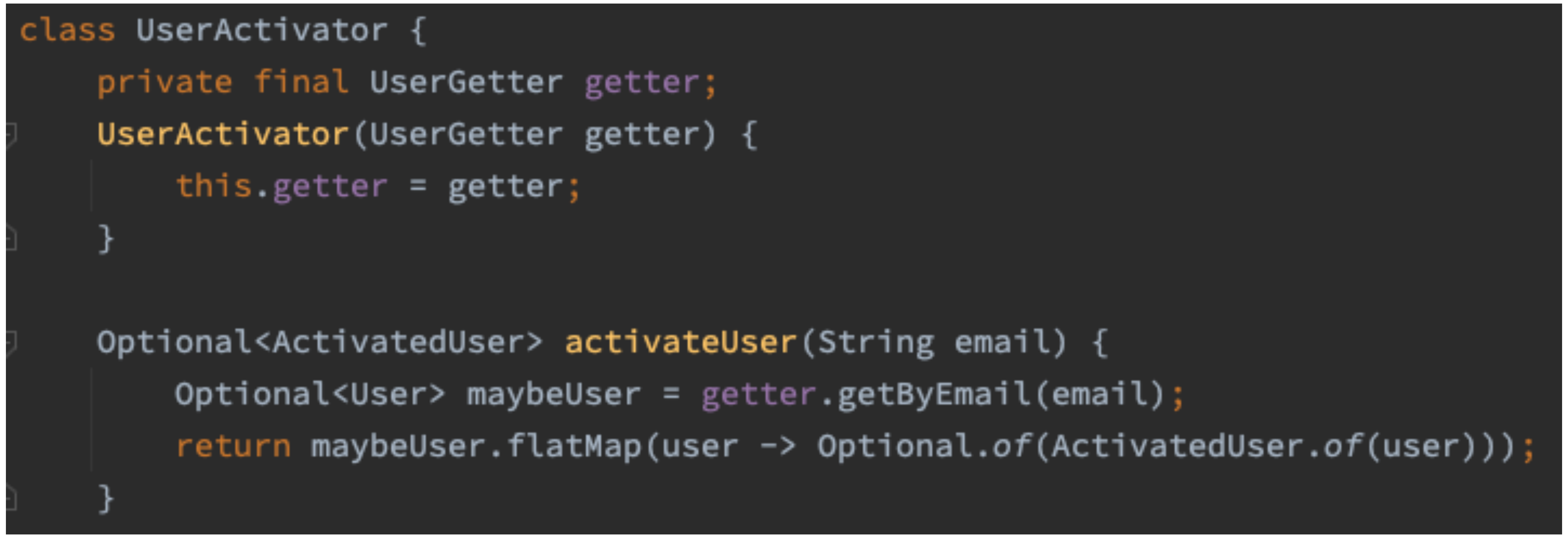
```
(defn trim-and-lower
  [input]
  (-> input
      string/trim
      string/lower-case))

  (->> users
       (filter :active?)
       (map prote-user))
```

- partial

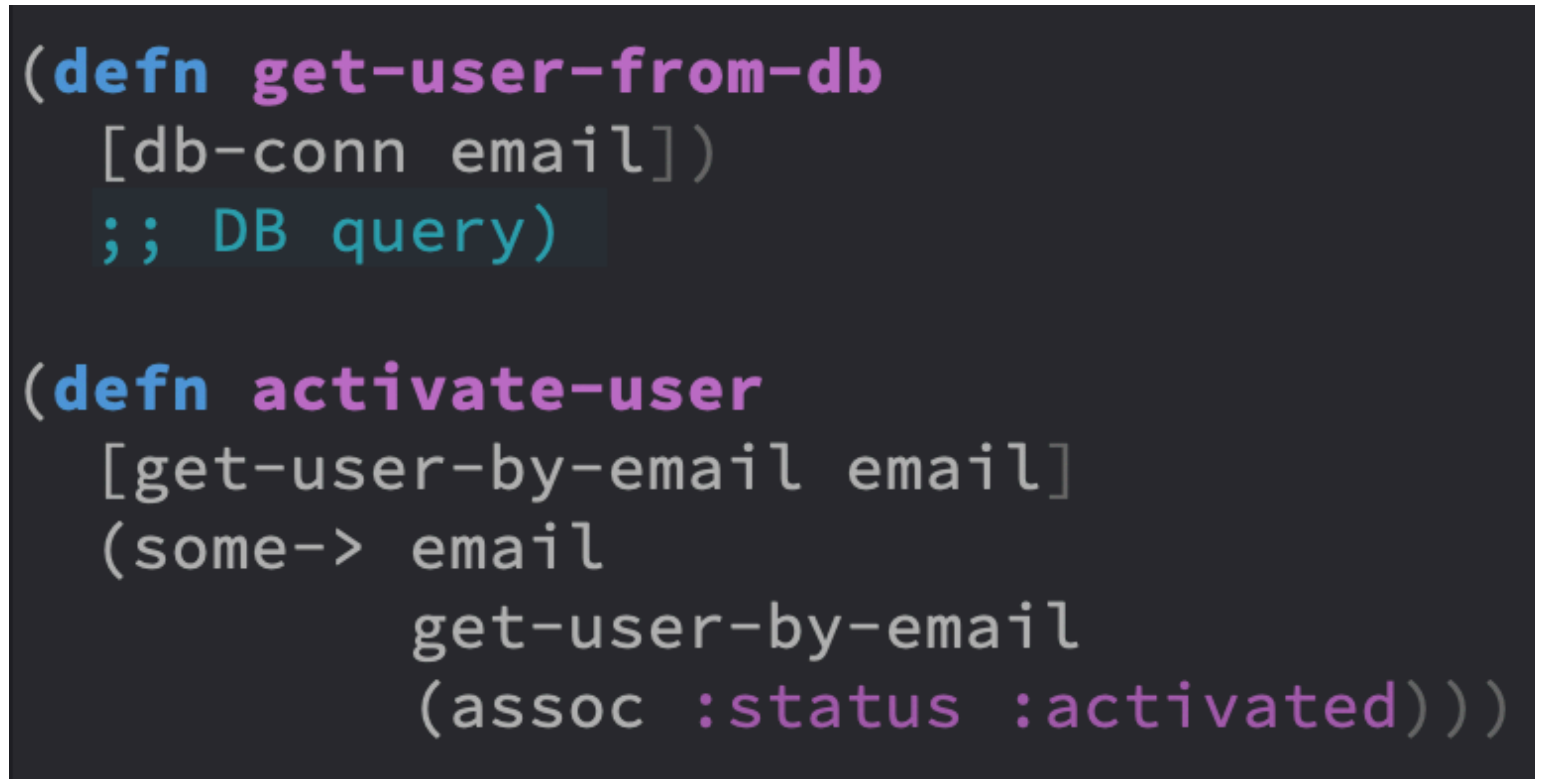


```
(defn age-valid?
  [age user]
  (<= age (:age user)))

(def adult? (partial age-valid? 18))

;; vs. returning lambda directly

(defn age-valid?
  [age]
  (fn [user]
    (<= age (:age user))))
```

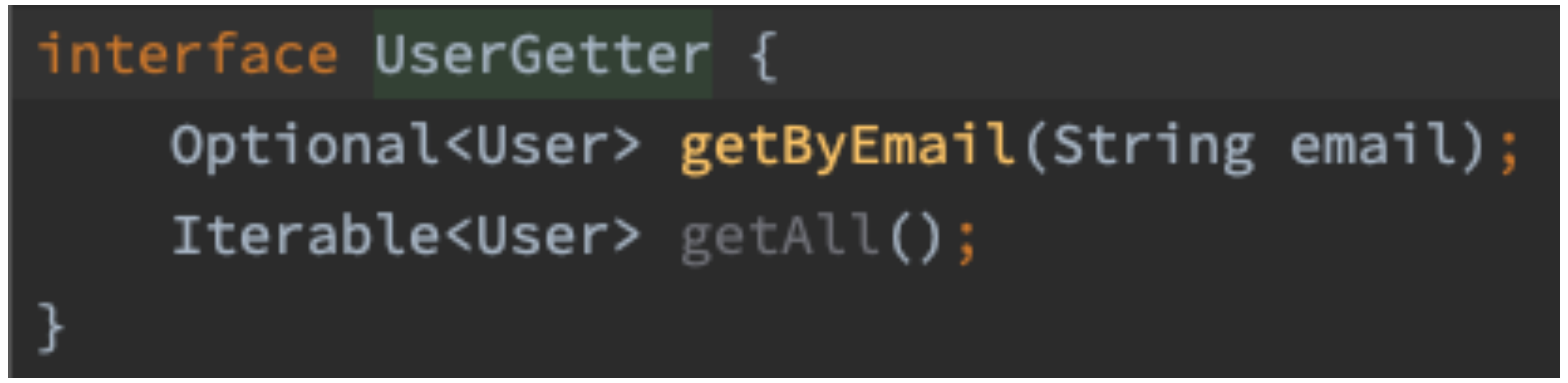
Interface Segregation Principle



```java
class UserActivator {
    private final UserGetter getter;
    UserActivator(UserGetter getter) {
        this.getter = getter;
    }

    Optional<ActivatedUser> activateUser(String email) {
        Optional<User> maybeUser = getter.getByEmail(email);
        return maybeUser.flatMap(user -> Optional.of(ActivatedUser.of(user)));
    }
```

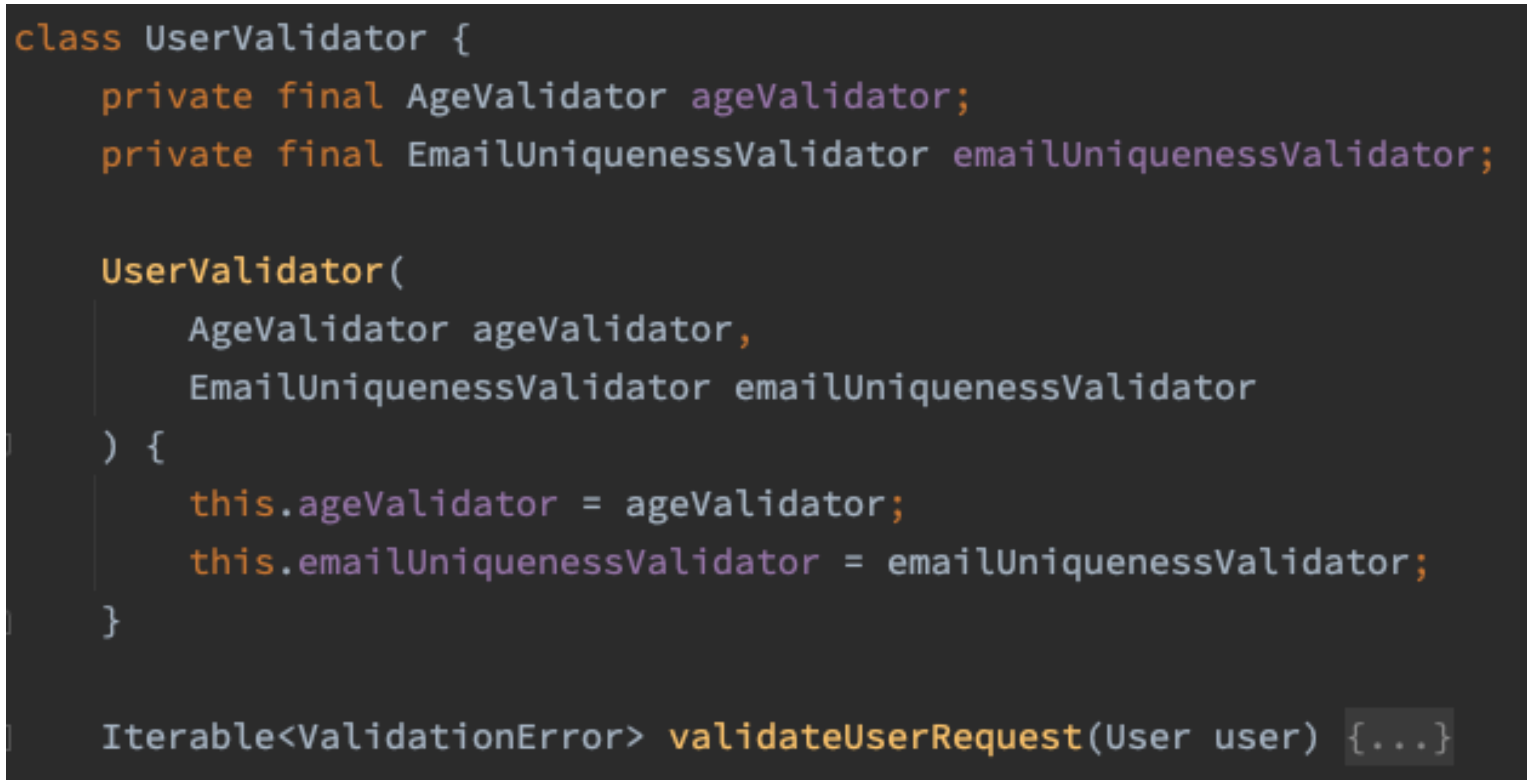


```clojure
(defn get-user-from-db
  [db-conn email])
  ;; DB query)

(defn activate-user
  [get-user-by-email email]
  (some-> email
          get-user-by-email
          (assoc :status :activated)))
```

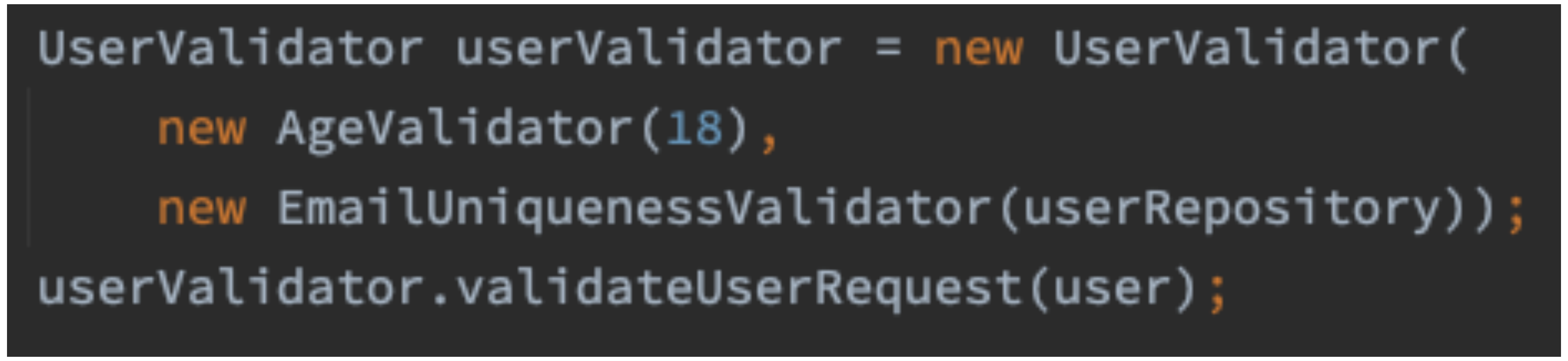


```java
interface UserGetter {
    Optional<User> getByEmail(String email);
    Iterable<User> getAll();
}
```

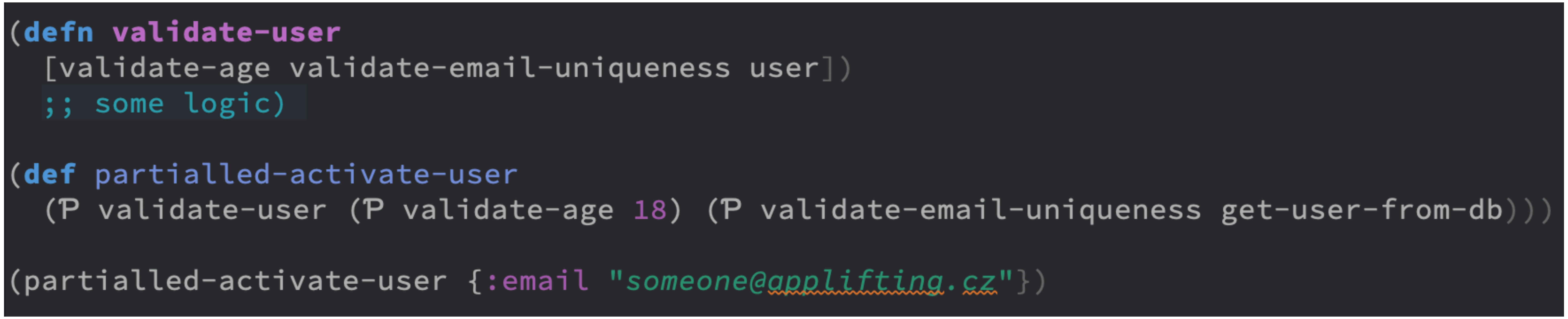
Dependency Inversion Principle



```java
class UserValidator {
    private final AgeValidator ageValidator;
    private final EmailUniquenessValidator emailUniquenessValidator;

    UserValidator(
        AgeValidator ageValidator,
        EmailUniquenessValidator emailUniquenessValidator
    ) {
        this.ageValidator = ageValidator;
        this.emailUniquenessValidator = emailUniquenessValidator;
    }

    Iterable<ValidationError> validateUserRequest(User user) {...}
```



```java
UserValidator userValidator = new UserValidator(
    new AgeValidator(18),
    new EmailUniquenessValidator(userRepository));
userValidator.validateUserRequest(user);
```

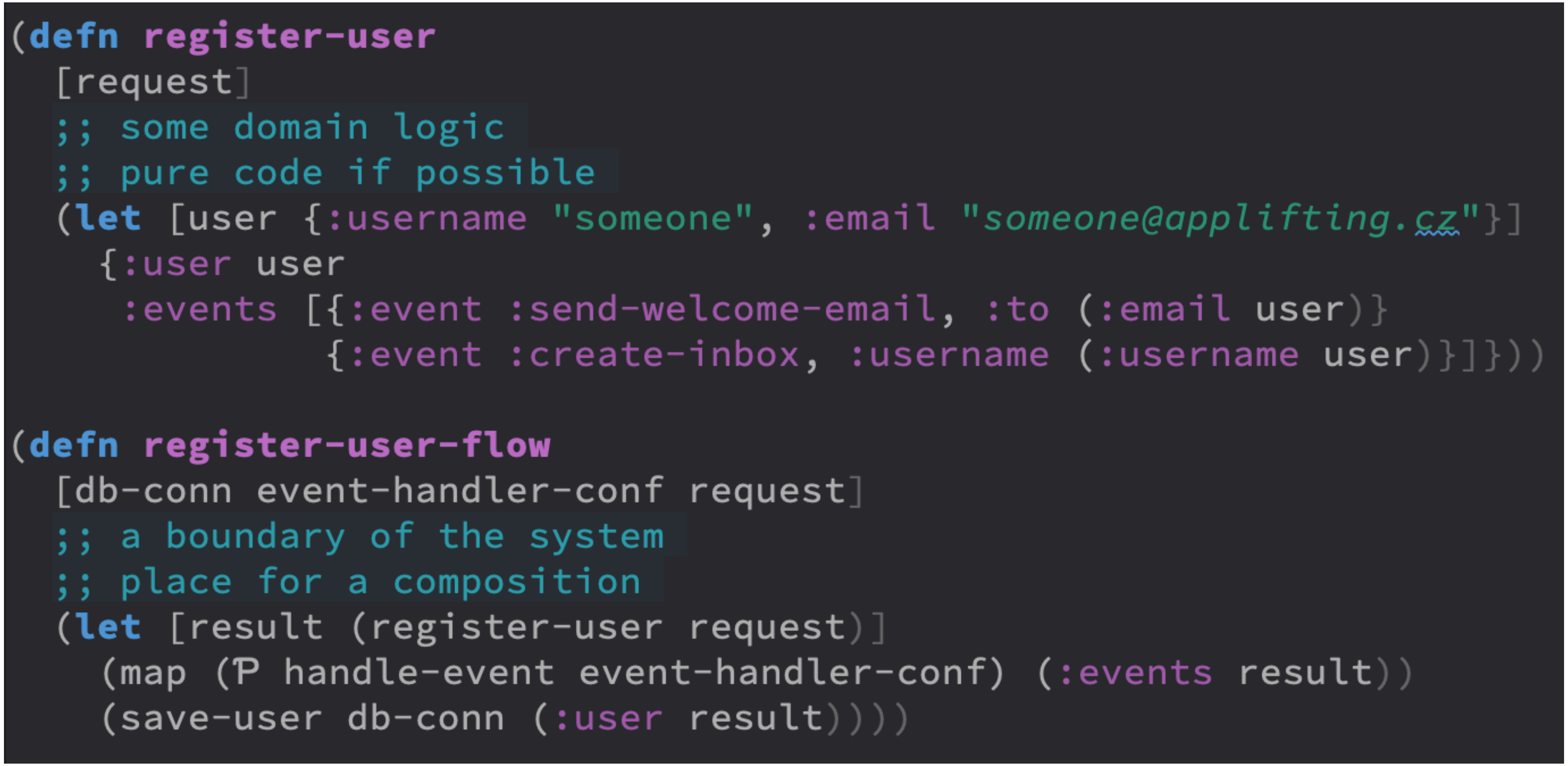


```clojure
(defn validate-user
  [validate-age validate-email-uniqueness user])
  ;; some logic)

(def partialled-activate-user
  (Ƥ validate-user (Ƥ validate-age 18) (Ƥ validate-email-uniqueness get-user-from-db)))

(partialled-activate-user {:email "someone@applifting.cz"})
```

Push side effects to the boundaries of the system

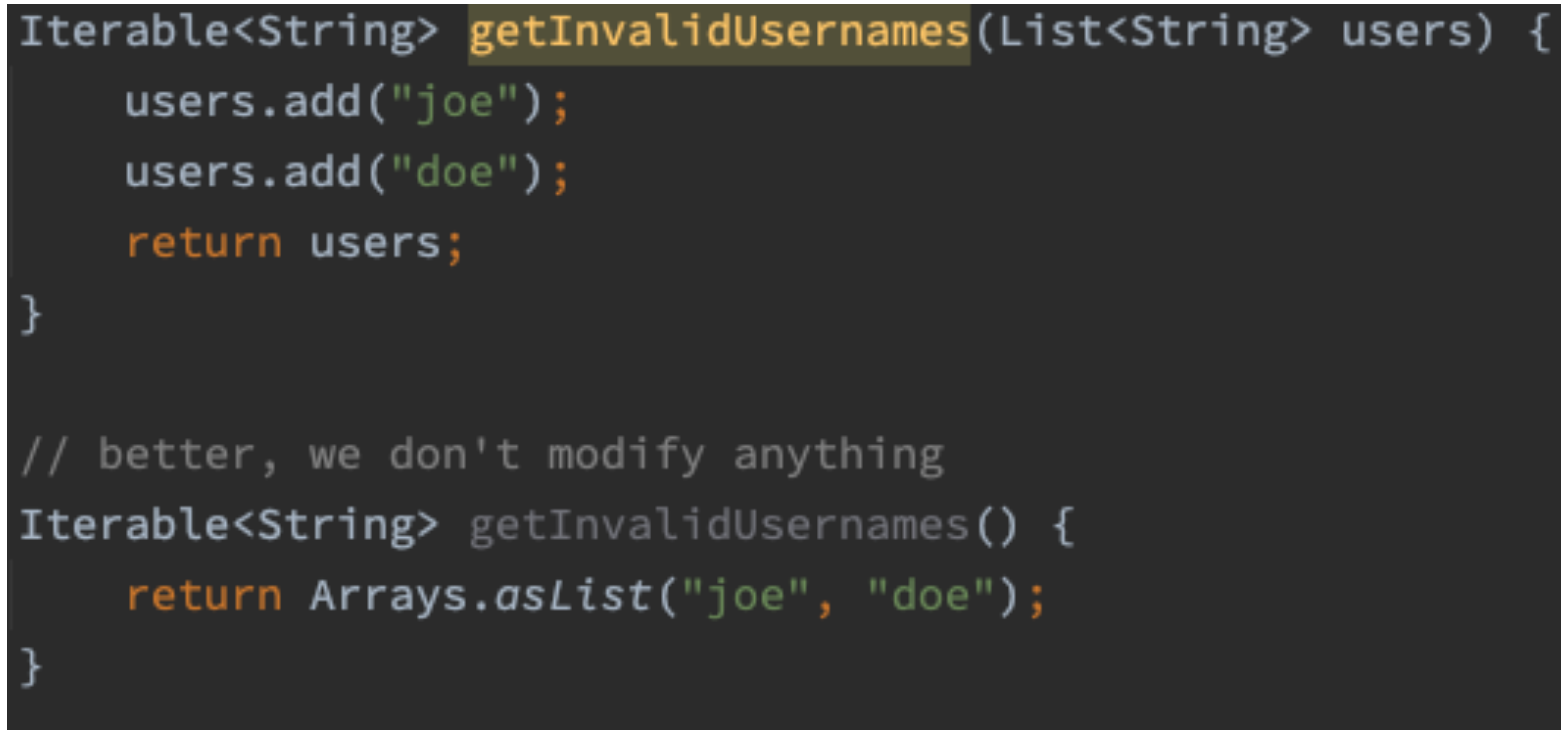
Model side effects as data

Side effects as data

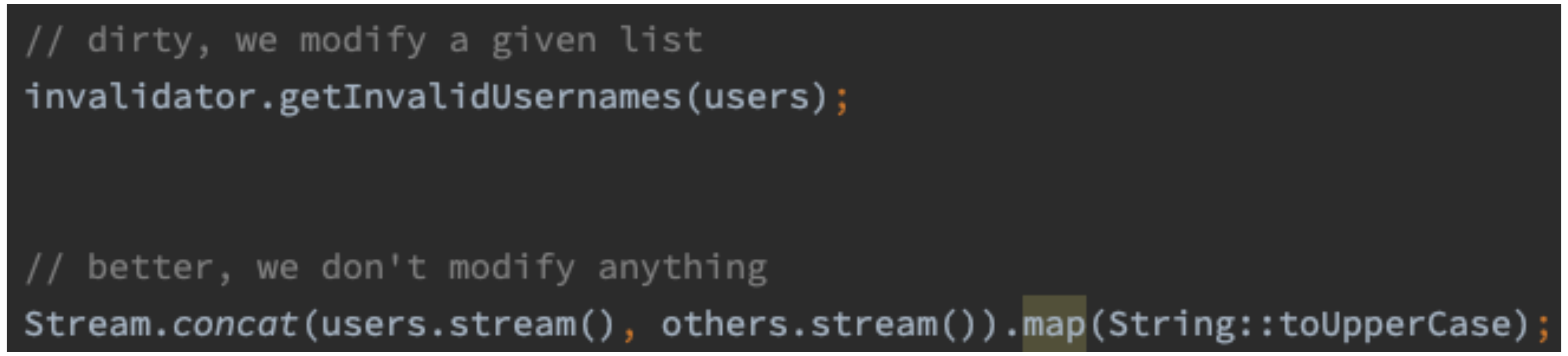


```
(defn register-user
  [request]
  ;; some domain logic
  ;; pure code if possible
  (let [user {:username "someone", :email "someone@applifting.cz"}]
    {:user user
     :events [{:event :send-welcome-email, :to (:email user)}
              {:event :create-inbox, :username (:username user)}]}))

(defn register-user-flow
  [db-conn event-handler-conf request]
  ;; a boundary of the system
  ;; place for a composition
  (let [result (register-user request)]
    (map (Ƥ handle-event event-handler-conf) (:events result))
    (save-user db-conn (:user result))))
```

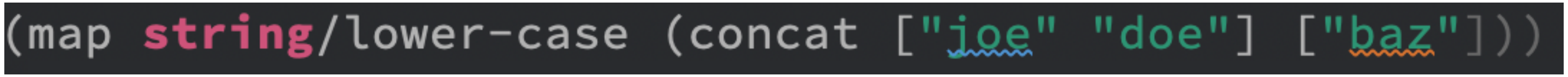
Bonus: Immutability



```java
Iterable<String> getInvalidUsernames(List<String> users) {
    users.add("joe");
    users.add("doe");
    return users;
}

// better, we don't modify anything
Iterable<String> getInvalidUsernames() {
    return Arrays.asList("joe", "doe");
}
```



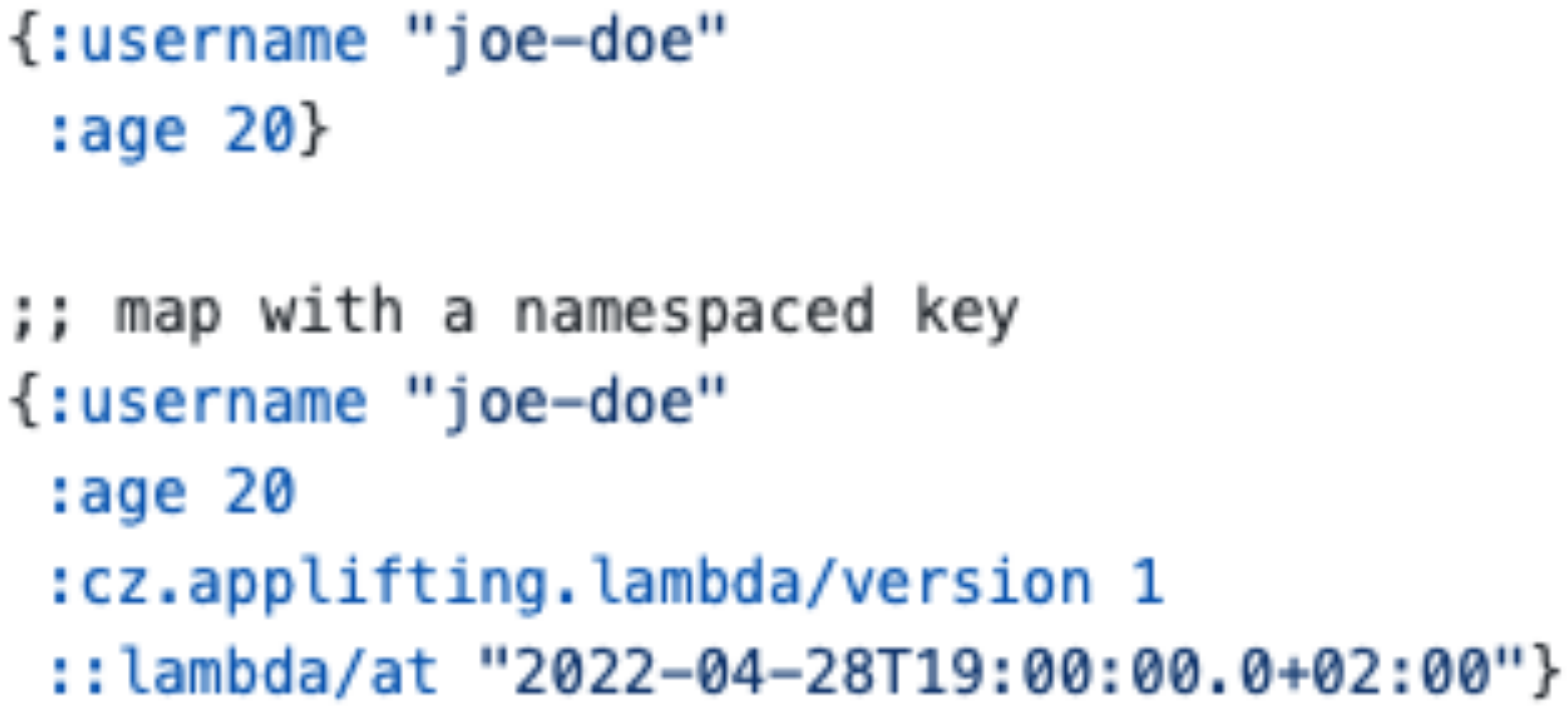
```java
// dirty, we modify a given list
invalidator.getInvalidUsernames(users);


// better, we don't modify anything
Stream.concat(users.stream(), others.stream()).map(String::toUpperCase);
```



```clojure
(map string/lower-case (concat ["joe" "doe"] ["baz"]))
```

Bonus: Open Data structures

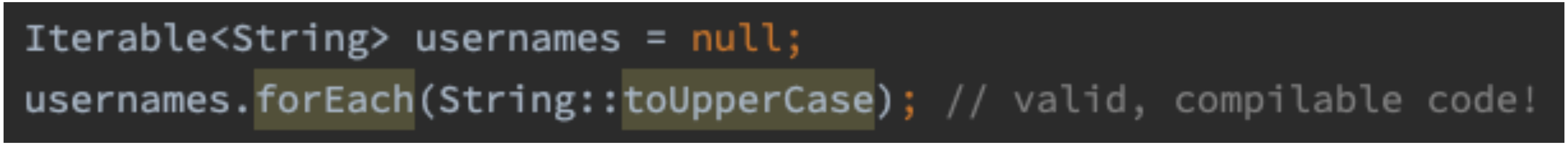
- Maps with keys/values over types
- Keys can be full-namespaced (like Java package)
- Data in, data out



```
{:username "joe-doe"
 :age 20}

;; map with a namespaced key
{:username "joe-doe"
 :age 20
 :cz.applifting.lambda/version 1
 ::lambda/at "2022-04-28T19:00:00.0+02:00"}
```

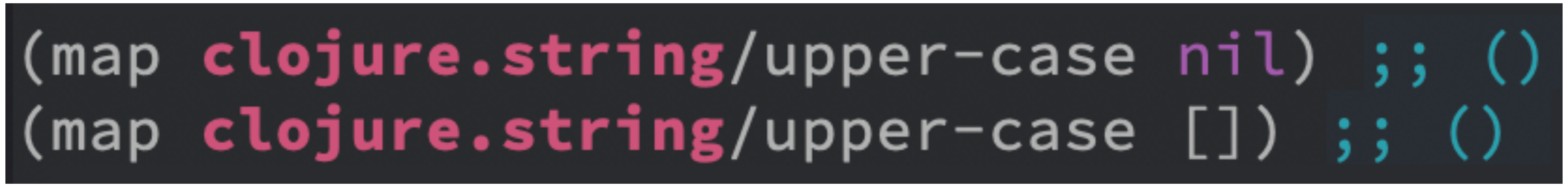
Bonus: Handling null pointers

Nil punning!



```
Iterable<String> usernames = null;
usernames.forEach(String::toUpperCase); // valid, compilable code!
```

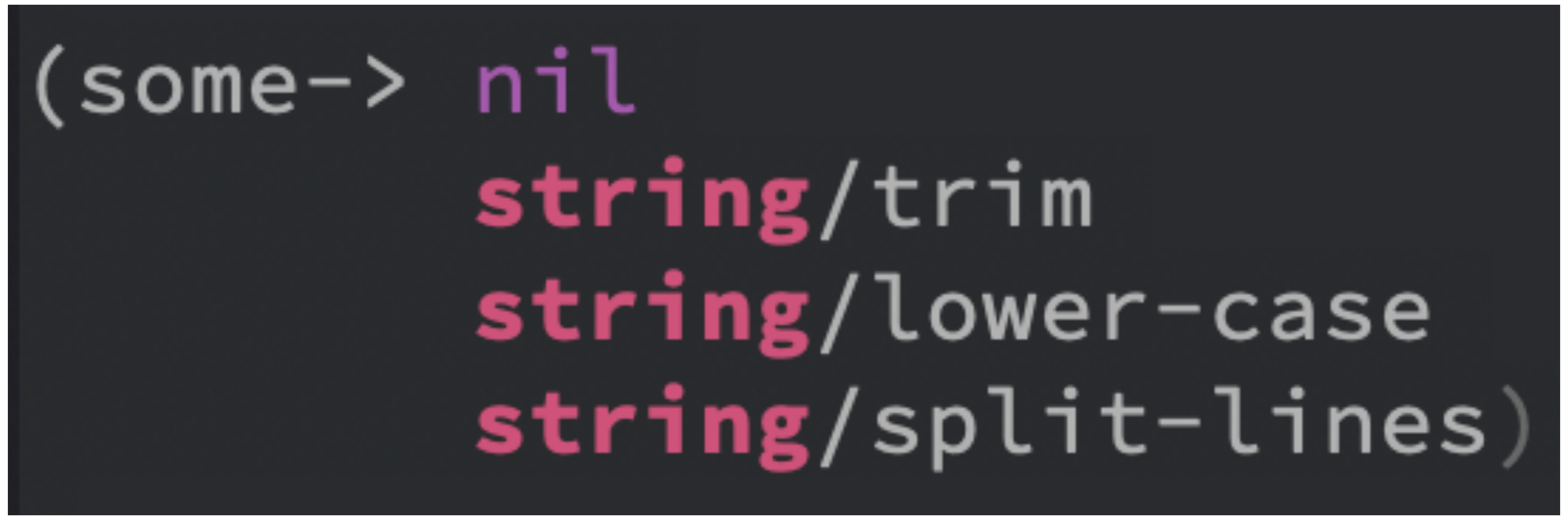
- Nil a is value in Clojure



```
(map clojure.string/upper-case nil) ;; ()
(map clojure.string/upper-case []) ;; ()
```

Bonus: Handling nilable strings

- -> works like a pipe on Unix shell
- some-> stops when there is a nil during flow
- Rich Hickey - Maybe not
- https://youtu.be/YR5WdGrpoug



```
(some-> nil
        string/trim
        string/lower-case
        string/split-lines)
```