```clojure
(ns acceptance-test
  (:require [clojure.test :refer :all]
            [data-processor :refer :all]))

(def rules '((define-counter "email-count" []
               true)
             (define-counter "spam-count" []
               (current "spam"))
             (define-signal {"spam-fraction" (/ (counter-value "spam-count" [])
                                                (counter-value "email-count" []))}
               true)
             (define-counter "spam-important-table" [(current "spam")
                                                     (current "important")]
               true)))

(defn process-data-dropping-signals [state new-data]
  (first (process-data state new-data)))

(deftest initial-state-test
  (testing "Query counter from initial state"
    (is (= 0
           (query-counter (initialize-processor rules) "spam" [])))))

(deftest unconditional-counter-test
  (let [st0 (initialize-processor rules)
        st1 (process-data-dropping-signals st0 {"spam" true})
        st2 (process-data-dropping-signals st1 {"spam" true})]
    (is (= 2
           (query-counter st2 "email-count" [])))))

(deftest conditional-counter-test
  (testing "Count incoming data by current condition"
    (testing "when repeated"
      (let [st0 (initialize-processor rules)
            st1 (process-data-dropping-signals st0 {"spam" true})
            st2 (process-data-dropping-signals st1 {"spam" true})
            st3 (process-data-dropping-signals st2 {"spam" true})]
        (is (= 3
               (query-counter st3 "spam-count" [])))))
    (testing "when ignored field varies"
      (let [st0 (initialize-processor rules)
            st1 (process-data-dropping-signals st0 {"spam" true, "noise" 1})
            st2 (process-data-dropping-signals st1 {"spam" true, "noise" 2})
            st3 (process-data-dropping-signals st2 {"spam" true, "noise" 3})]
        (is (= 3
               (query-counter st3 "spam-count" [])))))
    (testing "when considered field varies"
      (let [st0 (initialize-processor rules)
            st1 (process-data-dropping-signals st0 {"spam" true})
            st2 (process-data-dropping-signals st1 {"spam" false})

            st3 (process-data-dropping-signals st2 {"spam" true})]
        (is (= 2
               (query-counter st3 "spam-count" [])))))))

(deftest contingency-table-counter-test
  (let [st0 (initialize-processor rules)
        st1 (process-data-dropping-signals st0 {"spam" true, "important" true})
```

```
          st2 (process-data-dropping-signals st1 {"spam" true, "important" false})
          st3 (process-data-dropping-signals st2 {"spam" true, "important" false})
          st4 (process-data-dropping-signals st3 {"spam" false, "important" true})
          st5 (process-data-dropping-signals st4 {"spam" false, "important" true})
          st6 (process-data-dropping-signals st5 {"spam" false, "important" true})
          st7 (process-data-dropping-signals st6 {"spam" false, "important" false})
          st8 (process-data-dropping-signals st7 {"spam" false, "important" false})
          st9 (process-data-dropping-signals st8 {"spam" false, "important" false})
          end-state (process-data-dropping-signals st9 {"spam" false, "important" false})]
      (is (= 1
             (query-counter end-state "spam-important-table" [true true])))
      (is (= 2
             (query-counter end-state "spam-important-table" [true false])))
      (is (= 3
             (query-counter end-state "spam-important-table" [false true])))
      (is (= 4
             (query-counter end-state "spam-important-table" [false false])))))


(deftest signal-skip-on-error-test
  (let [st0 (initialize-processor rules)
        [st1 sg1] (process-data st0 {})]
    (is (= '() sg1))))

(deftest signal-launch-test
  (let [st0 (initialize-processor rules)
        [st1 sg1] (process-data st0 {"spam" true})
        [st2 sg2] (process-data st1 {"spam" false})
        [st3 sg3] (process-data st2 {})]
    (is (= 0
           (count sg1)))
    (is (= 1
           (count sg2)))
    (is (= 1
           (get (first sg2) "spam-fraction")))
    (is (= 1
           (count sg3)))
    (is (< 0.49
           (get (first sg3) "spam-fraction")
           0.51))))

(deftest past-value-test
  (let [st0 (initialize-processor '((define-signal {"repeated" (current "value")}
                                      (= (current "value") (past "value")))))
        [st1 sg1] (process-data st0 {"value" 1})
        [st2 sg2] (process-data st1 {"value" 2})
        [st3 sg3] (process-data st2 {"value" 1})
        [st4 sg4] (process-data st3 {"value" 1})
        [st5 sg5] (process-data st4 {"value" 2})]
    (is (= 0
           (count sg1)))
    (is (= 0
           (count sg2)))
    (is (= '({"repeated" 1})
           sg3))
    (is (= '({"repeated" 1})
           sg4))
    (is (= '({"repeated" 2})
```

```
                                    sg5))))
```