ДЕКЛАРАТИВНОЕ ПРОГРАММИРОВАНИЕ

Лекция 9. Пример программирования в Clojure.

Вспоминаем классику: игра «Жизнь»

Игра «Жизнь», придуманная английским математиком Джоном Конвеем (John Conway) (https://ru.wikipedia.org/wiki/Жизнь_(игра)) является алгоритмом, для реализации которого, кажется, лучше всего подойдут массивы. Мы реализуем правила этой игры: сначала традиционным способом – в качестве игрового поля будет использоваться вектор векторов, каждый элемент которого будет иметь либо значение `:on`, либо `nil`, а затем в более идиоматичной для языка Clojure манере – без сложностей (и ограничений), присущих индексам.

```
(defn empty-board
  "Создает прямоугольное игровое поле с указанной шириной и высотой."
  [w h]
  (vec (repeat w (vec (repeat h nil)))))
```

Теперь у нас появилась возможность создавать пустое игровое поле и нам необходимо реализовать добавление живых клеток в него:

```
(defn populate
  "Включает значение :on в ячейках, определяемых координатами [y, x]."
  [board living-cells]
  (reduce (fn [board coordinates]
            (assoc-in board coordinates :on))
          board
          living-cells))

(def glider (populate (empty-board 6 6) #{[2 0] [2 1] [2 2] [1 2] [0 1]}))


(pprint glider)
; [[nil :on nil nil nil nil]
;  [nil nil :on nil nil nil]
;  [:on :on :on nil nil nil]
;  [nil nil nil nil nil nil]
;  [nil nil nil nil nil nil]
;  [nil nil nil nil nil nil]]
```

Теперь самое основное: функция `indexed-step`, принимающая состояние игрового поля и возвращающая следующее его состояние в соответствии с правилами игры:

Пример 3.6. Реализация вспомогательных функций для indexed-step

```
(defn neighbours
  [[x y]]


    (for [dx [-1 0 1] dy [-1 0 1] :when (not= 0 dx dy)]
      [(+ dx x) (+ dy y)]))

  (defn count-neighbours
    [board loc]
    (count (filter #(get-in board %) (neighbours loc))))    ❶
```

```
(defn indexed-step
  "Возвращает следующее состояние игрового поля, используя индексы для
   определения координат ячеек, соседних с живыми клетками."
  [board]
  (let [w (count board)
        h (count (first board))]
    (loop [new-board board x 0 y 0]
      (cond
        (>= x w) new-board
        (>= y h) (recur new-board (inc x) 0)
        :else
          (let [new-liveness
                  (case (count-neighbours board [x y])
                    2 (get-in board [x y])
                    3 :on
                    nil)]
            (recur (assoc-in new-board [x y] new-liveness) x (inc y)))))))
```

❶ Обратите внимание: из-за того, что `count-neighbours` использует `get-in` – реализованную поверх `get`, которая, как мы знаем, возвращает `nil` для неизвестных индексов, она не генерирует ошибки при выходе за пределы игрового поля.

Посмотрим, как все это работает:

```
(-> (iterate indexed-step glider) (nth 8) pprint)
; [[nil nil nil nil nil nil]
;  [nil nil nil nil nil nil]
;  [nil nil nil :on nil nil]
;  [nil nil nil nil :on nil]
;  [nil nil :on :on :on nil]
;  [nil nil nil nil nil nil]]
```

Теперь у нас есть действующая реализация игры «Жизнь» с колонией клеток!

Посмотрим, как можно переделать это решение, чтобы избавиться от индексов. Для начала уберем итерации, выполняемые вручную. Любой вызов `loop` можно заменить вызовом `reduce` для диапазона:

```
(defn indexed-step2
  [board]
  (let [w (count board)
        h (count (first board))]
    (reduce
      (fn [new-board x]
        (reduce
          (fn [new-board y]
            (let [new-liveness
                    (case (count-neighbours board [x y])
                      2 (get-in board [x y])
                      3 :on
                      nil)]
              (assoc-in new-board [x y] new-liveness)))
          new-board (range h)))
      board (range w))))
```

Вложенные операции свертки всегда можно сократить:

```
(defn indexed-step3
  [board]
  (let [w (count board)
        h (count (first board))]
    (reduce
      (fn [new-board [x y]]
        (let [new-liveness
                (case (count-neighbours board [x y])
                  2 (get-in board [x y])
                  3 :on
                  nil)]
          (assoc-in new-board [x y] new-liveness)))
      board (for [x (range h) y (range w)] [x y]))))
```

Мы получили версию без цикла `loop`, в которой, однако, все еще используются индексы.

Как уже говорилось, индексы можно заменить последовательностями, но наши функции `count-neighbours` и `neighbours` тесно связаны с индексами, используя их для определения соседних ячеек и доступа к ним. Как можно выразить понятие «соседа» с помощью последовательности и без использования индексов?

При использовании одномерного массива сделать это очень просто, достаточно воспользоваться функцией `partition`:

```
(partition 3 1 (range 5))
;= ((0 1 2) (1 2 3) (2 3 4))
```

Результат, возвращаемый функцией `partition`, здесь можно интерпретировать как последовательность элементов 1, 2 и 3 с их соседями. Единственная проблема состоит в том, что этот код создает «окна» только для элементов, имеющих соседей с обеих сторон: элементы 0 и 4 с их соседями отсутствуют! Эту проблему можно исправить, дополнив исходную коллекцию:

```
(partition 3 1 (concat [nil] (range 5) [nil]))
;= ((nil 0 1) (0 1 2) (1 2 3) (2 3 4) (3 4 nil))
```

Оформим эту операцию в виде функции `window`:

```
(defn window
  "Возвращает ленивую последовательность окон с тремя элементами в каждом,
   центрами в которых является элемент coll."
  [coll]
  (partition 3 1 (concat [nil] coll [nil])))
```

Но как теперь перейти к двум измерениям? Вся хитрость в том, что когда мы будем применять функцию `window` к коллекции из `n` строк, мы будем получать `n` троек из трех строк, а каждая такая тройка (длиной `m`) может быть преобразована в последовательность из `m` троек. Формально эта операция называется *транспонированием* (transposition). Повторно применив `window` к такой последовательности, мы получим последовательность троек из троек и сможем создать окно 3 на 3 вокруг каждого элемента.

Взгляните на код:

```
(defn cell-block
  "Создает последовательность окон 3x3 на основе тройки из трех
   последовательностей."
  [[left mid right]]
  (window (map vector
               (or left (repeat nil)) mid (or right (repeat nil)))))
```

Две формы or должны заменить дополнительные значения nil, сгенерированные функцией window повторяющимися последовательностями из nil, потому что map остановится, как только один из ее аргументов окажется пустым[1]. Этот код можно упростить, добавив в функцию window необязательный аргумент pad:

```
(defn window
  "Возвращает ленивую последовательность окон с тремя элементами в каждом,
   центрами в которых является элемент coll, и дополненную значением
   pad или nil, если необходимо."
  ([coll] (window nil coll))
  ([pad coll]
    (partition 3 1 (concat [pad] coll [pad]))))

(defn cell-block
  "Создает последовательности окон 3x3 из троек последовательностей по 3
   элемента в каждой."
  [[left mid right]]
  (window (map vector left mid right)))
```

Нам необходимо определить, является ли ячейка в центре блока живой клеткой. Эту операцию так же желательно оформить в виде отдельной функции, но на сей раз воспользуемся механизмом деструктуризации, чтобы сжато разделить блок ячеек на составляющие:

```
(defn liveness
  "Возвращает признак наличия живой клетки (nil или :on) в центральной
   ячейке для выполнения следующего шага."
  [block]
  (let [[_ [_ center _] _] block]
    (case (- (count (filter #{:on} (apply concat block)))
             (if (= :on center) 1 0))
      2 center
      3 :on
      nil)))
```

Теперь можно переделать функцию `indexed-step`, задействовав в ней вспомогательные функции, не зависящие от индексов:

```
(defn- step-row
  "Возвращает следующее состояние центра строки."
  [rows-triple]

(defn index-free-step
  "Возвращает следующее состояние игрового поля."
  [board]
  (vec (map step-row (window (repeat nil) board))))
```

Даже при том, что `index-free-step` опирается на вспомогательную функцию, не зависящую от индексов, она эквивалентна функции `indexed-step`:

```
(= (nth (iterate indexed-step glider) 8)
   (nth (iterate index-free-step glider) 8))
;= true
```

Каждый шаг в проделанном нами пути достаточно прост, но весь путь от «императивного» решения к «последовательному» может показаться слишком заумным, пока вы не пройдете по нему пару раз.

Переход на следующий уровень. Проблема текущей реализации состоит в том, что она остается близкой по духу к *первоначальной реализации*. Однако можно попробовать найти более изящное решение. Для этого нужно глубоко вдохнуть, отступить на шаг назад, и детально исследовать правила игры «Жизнь».

На каждом шаге выполняются следующие переходы:

- любая живая клетка, имеющая по соседству менее двух живых клеток, погибает из-за малонаселенности;
- любая живая клетка, имеющая по соседству две или три живые клетки, продолжает жить в следующем поколении;
- любая живая клетка, имеющая по соседству более трех живых клеток, погибает из-за перенаселенности;
- в любой пустой ячейке, по соседству с которой имеется точно три живые клетки, появляется новая живая клетка в результате размножения.

В этих правилах не упоминаются ни строки, ни столбцы, ни индексы. В них говорится только о соседних ячейках и о наличии живых клеток в них. Точнее, в них говорится о ячейках с живыми клетками, о соседних ячейках и о пустых ячейках рядом с живыми клетками. Таким образом, двумя основными понятиями являются живые клетки и соседние ячейки, а понятие пустых ячеек (соседние пустые ячейки) выводится из понятий *соседние ячейки* и *живые клетки*.

Если придерживаться этих двух понятий, *единственным состоянием мира является множество живых клеток*. Чтобы сгенерировать каждое последующее состояние, достаточно сначала определить всех живых соседей ячейки и затем подсчитать, сколько раз появляется данная «соседняя ячейка» (количество появлений будет соответствовать количеству живых соседей).

Если попытаться переложить это на язык Clojure, в конечном итоге получится:

Пример 3.7. Изящная реализация игры «Жизнь»

```
(defn step
  "Возвращает следующее состояние мира"
  [cells]
  (set (for [[loc n] (frequencies (mapcat neighbours cells))
             :when (or (= n 3) (and (= n 2) (cells loc)))]
         loc)))
```

Это все! Нам потребовалась единственная вспомогательная функция `neighbours`; здесь не нужны индексы, игровое поле в виде вектора векторов, отсутствуют ограничения на размер игрового поля, и игровой мир имеет разреженное представление – явно представлены только координаты ячеек с живыми клетками[1].

Опробуем эту новую функцию `step` с нашим начальным игровым полем; Состояние мира больше не является игровым полем – оно хранит только координаты с живыми ячейками – но мы можем повторно использовать функцию `populate` для создания ограниченного игрового поля, которое проще визуализировать:

```
(->> (iterate step #{[2 0] [2 1] [2 2] [1 2] [0 1]})   ❶
  (drop 8)
  first
  (populate (empty-board 6 6))
  pprint)
; [[nil nil nil nil nil nil]
;  [nil nil nil nil nil nil]
;  [nil nil nil :on nil nil]
;  [nil nil nil nil :on nil]
;  [nil nil :on :on :on nil]
;  [nil nil nil nil nil nil]]
```

❶ Начиная с того же исходного расположения живых клеток, мы можем видеть, что колония перемещается на одну ячейку через каждые четыре шага.

Интересно отметить, что функция `step` в примере 3.7 использует функцию `neighbours`, реализованную для императивной функции `indexed-step` в примере 3.6. Однако, в контексте данного решения, `neighbours` имеет дело *не* с числовыми индексами в конкретной структуре данных, а с координатами: пары `[x y]` являются очевидными идентификаторами в отношении функции `step`, тогда как в исходной императивной функции `indexed-step` эти же пары определялись из индексов самой функцией `indexed-step`.

Таким образом, `neighbours` теперь единственная часть данного алгоритма, где требуется определять содержимое идентификаторов ячеек. Фактически она определяет топологию решетки. Немного изменив `neighbours`, этот код сможет поддерживать конечные решетки, тороидальные решетки, гексагональные решетки, N-мерные решетки и так далее, без изменения функции `step`. Уйдя от императивного мышления, мы получили более ясное разделение проблем и более удачное решение, очень близкое к универсальному.

Из функции `step` мы легко можем сделать нечто действительно универсальное: функцию высшего порядка, `stepper`, действующую как фабричную, производящую функции `step`.

```
(defn stepper
  "Возвращает функцию step для реализации клеточного автомата.
   neighbours принимает координаты и возвращает упорядоченную коллекцию
   координат. Функции survive? и birth? - это предикаты, проверяющие
   число живых соседей."
  [neighbours birth? survive?]
  (fn [cells]
    (set (for [[loc n] (frequencies (mapcat neighbours cells))
               :when (if (cells loc) (survive? n) (birth? n))]
           loc))))
```

Наша реализация функции `step` эквивалентна функции, возвращаемой вызовом `(stepper neighbours #{3} #{2 3})`. Эта функция высшего порядка `stepper` может использовать разные правила определения условий продолжения/зарождения/прекращения жизни и разные топологии, упоминавшиеся выше (гексагональная, трехмерная, ко-

нечная, сферическая, тороидальная, в виде ленты Мебиуса, и так далее). Например, клеточный автомат H.B2/S34 (гексагональная решетка, зарождение жизни при наличии двух живых соседей, продолжение жизни при наличии трех или четырех живых соседей) реализовать достаточно просто, как показано ниже:

```
(defn hex-neighbours
  [[x y]]
  (for [dx [-1 0 1] dy (if (zero? dx) [-2 2] [-1 1])]
    [(+ dx x) (+ dy y)]))

(def hex-step (stepper hex-neighbours #{2} #{3 4}))

;= ; эта конфигурация определяет осциллятор с периодом в 4 шага
(hex-step #{[0 0] [1 1] [1 3] [0 4]})
;= #{[1 -1] [2 2] [1 5]}
(hex-step *1)
;= #{[1 1] [2 4] [1 3] [2 0]}
(hex-step *1)
;= #{[1 -1] [0 2] [1 5]}
(hex-step *1)
;= #{[0 0] [1 1] [1 3] [0 4]}
```

Итак, четырехстрочная функция `stepper` является универсальной фабрикой для произвольных клеточных автоматов. Это оказалось возможным не благодаря отказу от индексов (потому что наша реализация на основе последовательности оказалась не универсальной), а за счет глубокого изменения структур данных, и перехода к использованию множеств, естественных идентификаторов и ассоциативных массивов (для отображения частот). Именно из-за того, что данное решение опирается на множества и естественные идентификаторы, его можно назвать «реляционным».

Литература

1. **Эмерик Ч., Карпер Б., Гранд К. "Программирование на Clojure" (2013)**

Эта книга продемонстрирует вам гибкость Clojure в решении типичных задач, таких как разработка веб-приложений и взаимодействие с базами данных. Вы быстро поймете, что этот язык помогает устранить ненужные сложности в своей практике и открывает новые пути решения сложных проблем, включая многопоточное программирование.
Издание предназначено для программистов, желающих освоить всю мощь и гибкость функционального программирования.