Мини-проект на языке Racket

1.1. Псевдослучайные числа

Реализуйте функции генерации псевдослучайных чисел, используя линейный конгруэнтный метод.

Упражнение 1.1. Реализуйте основные функции для генерации псевдослучайных чисел:

1. `(make-rng n)` — инициализировать состояние генератора[1] из стартового целого значения `n`

```
(make-rng 123)
; #<rng>
```

2. `(next-rng g)` — получите следующее состояние генератора, одним шагом линейного конгруэнтного метода:

$$G_{n+1} = (aG_n + c) \bmod m$$

Константы a, c и m можете взять из таблицы стандартных значений параметров[2].

```
(next-rng (make-rng 123))
; #<rng>
```

Упражнение 1.2. Реализуйте функцию `random-integer`, принимающую границы диапазона и возвращающую случайное число, представленное функцией, принимающей состояние генератора, и возвращающей случайное число в диапазоне от i до j, а также новое состояние генератора:

```
((random-integer 7 77) (make-rng 709))
; '(72 . #<rng>)
((random-integer 7 77) (make-rng 709))
; '(72 . #<rng>)  -- должно совпадать с результатом выше!
```

Упражнение 1.3. Реализуйте функцию высшего порядка `stream-random`, которая принимает функцию генерации случайных значений и начальное состояние генератора, и порождает бесконечный поток псевдослучайных значений. Функция генерации, подающаяся на вход, принимает состояние генератора и возвращает пару из псевдослучайного значения и нового состояния генератора.

```
(stream->list (stream-take (stream-random (random-integer 1 10) (make-rng 709)) 5))
; '(1 5 7 3 8)
(stream->list (stream-take (stream-random (random-integer 1 10) (make-rng 709)) 10))
; '(1 5 7 3 8 5 4 8 5 1) -- первые 5 чисел должны совпадать!
```

Упражнение 1.4. Реализуйте функцию высшего порядка `sample`, которая принимает функцию генерации случайных значений, а также опциональные параметры[3] для количества генерируемых значений и начального состояния генератора, и порождает список из случайных зачений:

```
(sample (random-integer 1 10))
; '(1 5 7 3 8)
(sample (random-integer 1 10) 10)
; '(1 5 7 3 8 5 4 8 5 1)
(sample (random-integer 1 10) 10 (make-rng 705))
; '(4 1 9 6 1 2 5 2 3 3)
```

[1] Используйте пользовательские структуры: §5 Programmer-Defined Datatypes.

[2] https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use

[3] См. https://docs.racket-lang.org/guide/contracts-general-functions.html#%28part._contracts-optional%29

1.2. Обработка случайных значений

Упражнение 1.5. Реализуйте функцию `map-random`, применяющую чистую функцию к случайному значению, чтобы получить другое случайное значение:

```
((random-integer 1 100) (make-rng 709))
; '(17 . #<rng>)
((map-random even? (random-integer 1 100)) (make-rng 709))
; '(#f . #<rng>)    -- потому что 17 — НЕ чётное

(sample (random-integer 1 100))
; '(17 30 3 46 82)
(sample (map-random even? (random-integer 1 100)))
; '(#f #t #f #t #t) -- соответствует числам выше
```

Упражнение 1.6. Реализуйте функцию `random-bool`, порождающую случайное булево значение:

```
(sample random-bool 10)
; '(#f #f #t #t #f #t #f #t #f #t)
```

Упражнение 1.7. Реализуйте функцию `random-coin-flip`, порождающую случайный результат подбрасывания монеты:

```
(sample random-coin-flip 10)
; '(орёл орёл решка решка орёл решка орёл решка орёл решка)
```

Упражнение 1.8. Реализуйте функцию `random-dice-roll`, порождающую случайный результат броска шестигранной кости:

```
(sample random-dice-roll 20)
; '(4 5 4 3 2 5 4 5 2 1 4 3 2 5 2 5 6 3 2 1)
```

Упражнение 1.9. Реализуйте функцию `frequency-count`, подсчитывающую количество элементов каждого значения в списке:

```
(frequency-count (sample random-coin-flip 1000))
; '((орёл . 506) (решка . 494))

(sort
  (frequency-count (sample random-dice-roll 600))
  (lambda (l r) (< (car l) (car r)))) ; сортировка по первой компоненте
; '((1 . 107) (2 . 100) (3 . 103) (4 . 93) (5 . 90) (6 . 107))
```

1.3. Комбинация случайных значений

Упражнение 1.10. Реализуйте функцию высшего порядка `random-cons`, принимающую два случайных значения, и возвращающую случайную пару значений:

```
(sample (random-cons (random-integer 1 10) (random-integer 1 10)))
; '((1 . 5) (7 . 3) (8 . 5) (4 . 8) (5 . 1))
```

Упражнение 1.11. Реализуйте функцию высшего порядка `random-if`, принимающую случайное булево значение и два случайных значения, и возвращающую первое, если булево значение `#t`, и второе — иначе. Важно, что состояние генератора случайных чисел используется только для генерации булево значения и одного из двух случайных значений (но не обоих!):

```
(sample (random-if random-bool random-dice-roll random-coin-flip))
; '(орёл 3 решка решка решка)
```

Упражнение 1.12. Реализуйте функцию высшего порядка `random-constant`, принимающую значение и возвращающее случайное значение, которое всегда равно заданному. Важно, что состояние генератора случайных чисел не используется:

```
(sample (random-constant 'ё) 10)
; '(ё ё ё ё ё ё ё ё ё ё)
```

Упражнение 1.13. Реализуйте функцию высшего порядка `random-choice`, принимающую два случайных значения, и возвращающую **случайным образом** либо первое, либо второе значение. Важно, что состояние генератора случайных чисел используется только для генерации одного из двух случайных значений (но не обоих!):

```
(sample (random-choice random-dice-roll random-coin-flip))
; '(орёл 3 решка решка решка)
```

Упражнение 1.14. Реализуйте функцию `random-item`, принимающую непустой список значений и возвращающую **случайным образом** одно из значений в списке.

```
(sample (random-item '(a b c)) 10)
; '(c a a b b b a c a c)
```

Упражнение 1.15. Реализуйте функцию высшего порядка `random-list-of-size`, порождающую случайные списки заданного размера, где каждое значение генерируется заданной случайным значением, переданным в качестве аргумента:

```
(sample (random-list-of-size random-coin-flip 2))
; '((орёл орёл) (решка решка) (орёл решка) (орёл решка) (орёл решка))

(frequency-count (sample (random-list-of-size random-coin-flip 2) 1000))
; '(((орёл орёл) . 240)
;   ((решка решка) . 235)
;   ((орёл решка) . 268)
;   ((решка орёл) . 257))
```

1.4. Связывание случайных значений

Упражнение 1.16. Реализуйте функцию высшего порядка `random-bind`, принимающую случайное значение и функцию, которая возвращает случайное значение, и применяет функцию к случайному значению, чтобы получить случайный результат:

```
(sample
  ; бросаем шестигранную кость
  (random-bind random-dice-roll
     ; и подбрасываем монетку столько раз, сколько выпало на кости
    (lambda (n) (random-list-of-size random-coin-flip n))))
; '((орёл решка решка орёл)
;   (орёл решка орёл решка решка)
;   (орёл решка орёл)
;   (орёл орёл орёл решка решка)
;   (орёл))
```

Упражнение 1.17. Реализуйте функцию высшего порядка `random-list`, порождающую случайные списки случайного размера, где каждое значение генерируется заданной случайным значением, переданным в качестве аргумента:

```
(sample (random-list random-bool) 1)
; '((#t #f #t #t #t #f #f #f #t))
```

Упражнение 1.18. Реализуйте **макрос** `let*/random`, позволяющий объявлять случайные значения аналогично `let*`:

```
(sample
  (let*/random ([n random-dice-roll]
                [lst (random-list-of-size random-coin-flip n)])
    (length lst)))
; '(4 5 3 5 1)
```

1.5. Вычисление числа π методом Монте-Карло

Упражнение 1.19. Реализуйте функцию `random-real`, порождающую случайное вещественное число в диапазоне:

```
(sample (random-real 0 1))
; '(0.709 0.596 0.823 0.054 0.489)
```

Упражнение 1.20. Реализуйте функцию `random-point-in-unit-square`, порождающую случайную точку в единичном квадрате $-1 \leq x, y \leq 1$:

```
(sample random-point-in-unit-square)
; '((0.53 . 0.19199999999999995)
;   (0.6459999999999999 . -0.892)
;   (-0.02200000000000002 . 0.23199999999999998)
;   (-0.44999999999999996 . -0.9319999999999999)
;   (-0.44599999999999995 . 0.768))
```

Упражнение 1.21. Реализуйте функцию `monte-carlo-pi`, вычисляющую число π методом Монте-Карло. А именно, вам необходимо сгененрировать заданное число точек в единичном квадрате и вычислить приближение числа π на основе доли точек, которые попали внутрь единичного диска ($x^2 + y^2 \leq 1$):

```
(monte-carlo-pi 100)      ; 3.24
(monte-carlo-pi 1000)     ; 3.176
(monte-carlo-pi 10000)    ; 3.14
```

1.6. Проверка свойств

Рассмотрим свойства — как структуры, состоящие из генератора случайных значений, и предиката, определённого на этих значениях:

```
(struct property (random-value predicate))
```

Например, мы можем определить свойство, утверждающее, что для любых списков `lst` верно, что `reverse (reverse lst) = lst`:

```
(define reverse-reverse-is-identity
  (property
    (random-list (random-integer 0 100))
    (lambda (xs) (equal? xs (reverse (reverse xs))))))
```

Упражнение 1.22. Реализуйте свойство `real-addition-is-associative`, утверждающее, что сложение действительных чисел ассоциативно:

$$\forall x, y, z. x + (y + z) = (x + y) + z$$

```
(define real-addition-is-associative
  ...)
```

Упражнение 1.23. Реализуйте функцию `find-counterexample`, находящую не более одного контрпримера для свойства.

```
(find-counterexample property-reverse-reverse)
; '()   — не нашлись контрпримеры

(find-counterexample real-addition-is-associative)
; '((0.546 0.357 0.148))  — есть как минимум один контрпример
```

Упражнение 1.24. Реализуйте **макрос** `forall/property`, позволяющий строить свойства аналогично конструкциям `let` и `for/list`. Порождённые случайные значения должны быть представлены списком, где каждый элемент — список вида `(id = value)`:

```
(find-counterexample
  (forall/property ([x (random-real 0 1)]
                    [y (random-real 0 1)]
                    [z (random-real 0 1)])
    (= (+ x (+ y z))
       (+ (+ x y) z))))
; '(((z = 0.546) (y = 0.357) (x = 0.148)))
```

Упражнение 1.25. Реализуйте **макрос** `forall*/property`, позволяющий строить свойства аналогично конструкциям `let*` и `for*/list`. Порождённые случайные значения должны быть представлены списком, где каждый элемент — список вида `(id = value)`:

```
(find-counterexample
  (forall*/property ([x (random-real 0 1)]
                     [y (random-real 0 x)]
                     [z (random-real 0 y)])
    (> x (+ y z))))
; '(((z = 0.1988745) (y = 0.530332) (x = 0.709)))
```

1.7. Случайные функции

Упражнение 1.26 (**+20 доп. баллов**). Реализуйте возможность генерировать случайные функции над целыми числами, и проверьте следующие свойства известных функций:

1. `(apply + (map f lst)) = (foldl (lambda (x acc) (+ (f x) acc)) 0 lst)`
2. `(length (map f lst)) = (length lst)`
3. `(map f (reverse lst)) = (reverse (map f lst))`
4. `(map f (map g lst)) = (map (lambda (x) (f (g x))) lst)`
5. `(reverse (append xs ys)) = (append (reverse ys) (reverse xs))`
6. `(map f (append xs ys)) = (append (map f xs) (map f ys))`

Также реализуйте возможность генерировать случайные потоки целых чисел, и проверьте следующие свойства для потоков:

1. `(stream-take (stream-take s n) m) = (stream-take s (min n m))`
2. `(stream-drop (stream-drop s n) m) = (stream-drop s (+ n m))`