Lista zagadnień nr 10

Zadanie 18.

Zmodyfikuj interpreter w pliku `nondeterminism-composition.rkt` albo `nondeterminism-monadic.rkt` tak, by nie zwracał listy możliwych rezultatów, ale **dyskretny rozkład prawdopodobieństwa** wyrażony jako lista **par** zawierających prawdopowodbieństwo i wartość. Dodaj procedury wbudowane:

- `(flip p x y)` – wybiera (czyli tworzy rozkład prawdopodobieństwa) wartość `x` z prawdopodobieństwem `p` i wartość `y` z prawdopodobieństwem `(- 1 p)`.
- `(uniform xs)` – wybiera element z list `xs`, każdy z takim samym prawdopodobieństwem.

Przykładowo:

```
> (eval (parse '(uniform (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6
    null)))))))))
'((1/6 . 1) (1/6 . 2) (1/6 . 3) (1/6 . 4) (1/6 . 5) (1/6 . 6))

> (eval (parse '(if (flip 0.3 true false) "wygrana" "przegrana")))
'((0.3 . "wygrana") (0.7 . "przegrana"))

> (eval (parse '(if (flip 0.3 true false)
                    (if (flip 0.3 true false)
                        "wygrana"
                        "przegrana")
                    "przegrana")))
'((0.09 . "wygrana") (0.21 . "przegrana") (0.7 . "przegrana"))
```

Zwróć uwagę, że wartość `przegrana` pojawia się w wyniku dwa razy. I to jest Ok, bo czasem zwyczajnie trudno stwierdzić czy dwie wartości są takie same[1], np. w przypadku procedur wbudowanych.

```
> (eval (parse '(flip 0.5 (lambda (x) (+ 1 x)) (lambda (x)
(+ 1 x))))) '((0.5 . #<clo>) (0.5 . #<clo>))
```

[1] ...albo, co to w ogóle znaczy być takim samym. Czy procedury `(lambda (x) (+ 1 x))` i `(lambda (x) (+ x 1))` są takie same?

Można natomiast uporządkować wyniki już po interpretacji. Zdefiniuj procedurę `norm`, która „skleja" w rozkładzie takie same wartości sumując prawdopodobieństwa, np. rozważmy program

```
(define TWO-DICE
  '(letrec [from-to (lambda (x n)
                (cons x
                      (if (= x n)
                          null
                          (from-to (+ 1 x) n))))]
   (let    [dice1 (uniform (from-to 1 6))]
   (let    [dice2 (uniform (from-to 1 6))]
   (+ dice1 dice2)))))
```

Wówczas:

```
> (eval (parse TWO-DICE))
'((1/36 . 2) (1/36 . 3) (1/36 . 4) (1/36 . 5)  (1/36 . 6) (1/36 . 7)
    (1/36 . 3) (1/36 . 4) (1/36 . 5) (1/36 . 6) (1/36 . 7) (1/36 . 8)
    (1/36 . 4) (1/36 . 5) (1/36 . 6) (1/36 . 7) (1/36 . 8) (1/36 . 9)
    (1/36 . 5) (1/36 . 6) (1/36 . 7) (1/36 . 8) (1/36 . 9) (1/36 . 10)
    (1/36 . 6) (1/36 . 7) (1/36 . 8) (1/36 . 9) (1/36 . 10)  (1/36 .
    11) (1/36 . 7) (1/36 . 8) (1/36 . 9) (1/36 . 10) (1/36 . 11) (1/36
    . 12))
```

Ale:

```
> (norm (eval (parse TWO-DICE)))
'((1/36 . 12) (1/18 . 11) (1/12 . 10) (1/9 . 9) (5/36 . 8) (1/6 . 7)
    (5/36 . 6) (1/9 . 5) (1/12 . 4) (1/18 . 3) (1/36 . 2))
```

Kolejnym przykładem niech będzie program, w którym najpierw rzucamy monetą, a potem jeśli wypadnie orzeł to raz kostką, a jak reszka to dwa razy kostką sumując wyniki (zwracamy uwagę, żę w naszym języku nie ma procedur jednoargumentowych, więc procedura `dice` bierze argument, z którego potem nie korzysta):

```
(define COIN-DICE
  '(letrec [from-to (lambda (x n)
                (cons x
                      (if (= x n)
                          null
                          (from-to (+ 1 x) n))))]
   (let    [dice (lambda (x) (uniform (from-to 1 6)))]
   (if (flip 0.5 true false)
       (dice false)
       (+ (dice false) (dice false))))))
```

Na koniec, jako przykład, zdefiniuj w swoim języku program `DICE-MANY`, który:

- Rzuca kostką. Wynik rzutu oznaczmy jako n.
- Wykonuje n rzutów kostką. Końcową wartością programu jest suma oczek tych n rzutów.

Jego wynik to:

```
> (norm (eval (parse DICE-MANY)))
'((1/279936 . 36) (1/46656 . 35) … (7/216 . 2) (1/36 . 1)))
```

Podsumowaując, w tym zadaniu należy:

- Rozbudować nasz język tak, by obliczeniami były rozkłady prawdopodobieństwa,
- Zdefiniować procedurę `norm`, która skleja w rozkładzie takie same wyniki,
- Napisać program `DICE-MANY`.