# Задание 3. Лямбда-функции, каррирование, сечения и оператор композиции

Функция `integrate` *приближенно* (с помощью метода трапеций) вычисляет определённый интеграл функции $f$ на промежутке $[a, b]$:

```
integrate :: (Double -> Double) -> Double -> Double -> Double
integrate f a b = integrate' f a b 1000 where
  integrate' f a b 0 = 0
  integrate' f a b n = (f (a) + f (a + h)) / 2 * h + integrate' f (a +
        h) b (n - 1)
    where h = (b - a) / n
```

Необходимо, используя функцию `integrate`, вычислить определённые интегралы следующих функций:

1. $f(x) = 2 \cdot x^2 + 3 \cdot x - 1, x \in [-3, 1]$;
2. $f(x) = 2 \cdot x^2, x \in [1, 2]$;
3. $f(x) = 1/(cos\ x)^2, x \in [-\pi/4, 0]$;
4. $f(x) = e^{(-x^2)}, x \in [-1000, 1000]$;
5. $f(x) = Si\ x, (0, 1]$ ($Si\ x$ – интегральный синус, интеграл функции $sin\ t/t$ при $t \in (0, x]$).

- Запрещается создание именованных функций.
- В примерах (2)-(4) подынтегральная функция должна быть создана, используя оператор композиции `(.)`, каррирование и/или сечения операторов.
- Интеграл 5 не должен вычисляться в `NaN`.

## Бесточечное программирование

Частичное применение и композиция функций позволяют определять функции без указания их фактических параметров.

В качестве примера рассмотрим функцию, вычисляющую сумму первых 10 элементов списка чисел:

```
sumOfFirst10 xs = sum (take 10 xs)

*Main> sumOfFirst10 [1..]
55
```

Используя каррирование и оператор композиции мы можем переписать функцию следующим образом:

```
sumOfFirst10' = sum . take 10
```

Напомним, что оператор композиции `f . g ≡ \x -> f(g(x))`.

То есть, `sumOfFirst10'` сначала вычисляет частично примененную `take 10`, а затем передает полученный список в функцию `sum`.

Такой способ записи функций называется бесточечным (англ. — pointfree) стилем программирования.

Плюсом такого подхода является компактность получаемых функций. Бесточечный стиль также позволяет конструировать неименованные функции без использования синтаксиса лямбда-функций:

```
rootsLambda = map (\x -> sqrt (abs x)) [-10..0]
```

vs

```
rootsPointfree = map (sqrt . abs) [-10..0]
```

Злоупотребление бесточечным стилем, однако, может навредить читаемости кода.

Подробнее о бесточечном программировании: Haskell Wiki

# Сечения операторов

Haskell имеет специальный синтаксис частичного применения операторов, называемый сечениями. Различают левые и правые сечения:

- при левом сечении фиксируется аргумент, располагающийся **слева** от оператора: `(2^) ≡ (^) 2 ≡ \x -> 2 ^ x`;
- при правом фиксируется аргумент, располагающийся **справа** от оператора: `(^2) ≡ flip (^) 2 ≡ \x -> x ^ 2`.

Использование сечений возможно для всех операторов (в т.ч. пользовательских), **кроме** оператора вычитания:

- Левое сечение возможно: `(10-) ≡ (-) 10 ≡ \x -> 10 - x`.

- Правое сечение невозможно, выражения вида `-e` интерпретируются как унарный минус. Для задания функции, аналогичной правому сечению, можно использовать функцию `subtract`: `(subtract e)` ≡ `flip (-) 10` ≡ `\x -> x - 10`.

Подробнее о сечениях: Haskell Wiki.