Лямбды

Мотивация

Напишем функцию, фильтрующую список по некоторому условию.

```
fun <T> filter(list: List<T>) : List<T> {
    val result = mutableListOf<T>()
    for (el in list) if (/*condition*/) result.add(el)
    return result
}
```

Довольно часто пригождающаяся штука — например...

- оставить только целые числа
- оставить только непустые строки
- только товары до тысячи рублей
- только пользователей старше 18...

Остаётся вопрос — а как *условие* передать в функцию извне? Один такой механизм мы уже знаем — он широко используется в графике, так передаются обработчики событий.

Потребуем передать *что-то*, у чего есть метод, принимающий `T` и возвращающий `Boolean`. Для этого надо сделать интерфейс.

```
interface Predicate<T> {
    fun test(value: T): Boolean
}
```

Теперь можем завершить наш фильтр:

```
fun <T> filter(list: List<T>, predicate: Predicate<T>) : List<T> {
    val result = mutableListOf<T>()
    for (el in list) if (predicate.test(el)) result.add(el)
    return result
}
```

Выглядит неплохо. А что на вызывающей стороне?

```
class IsEvenPredicate : Predicate<Int> {
    override fun test(value: Int) = value % 2 == 0
}

val list = listOf(1, 2, 3, 4, 5, 6)
filter(list, IsEvenPredicate())
```

```
=> : List<Int> = [2, 4, 6]
```

Оу... *Многословно*. На то, чтобы создать и передать предикат, мы тратим три строчки кода. Это конечно, оправдано, если мы передаём небольшой кусок кода в огромную сложную функцию.

Например, запуском потоков занимается *очень* непростой код, а то, что в этом потоке исполнять, передаётся в наследнике Runnable:

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

Или, переводя на Kotlin,

```
fun interface Runnable {
    fun run()
}
```

... но для таких маленьких задач это, конечно, совершенно неоправданно.

Примеры

Но давайте пока работать с тем, что есть. Обобщим понятие функции, для начала. У функции может быть разное количество аргументов, и, к сожалению, сделать произвольное количество параметров типа мы не сможем.

```
interface Function0<R> {
    fun invoke(): R
}

interface Function1<T, R> {
    fun invoke(arg: T): R
}

interface Function2<T1, T2, R> {
    fun invoke(arg1: T1, arg2: T2): R
}

interface Function3<T1, T2, T3, R> {
    fun invoke(arg1: T1, arg2: T2, arg3: T3): R
}
```

... и так далее. Собственно, набора таких интерфейсов достаточно, чтобы описать всё, что нам нужно:

```
fun <T> List<T>.filter(predicate: Function1<T, Boolean>): List<T> {
    val result = mutableListOf<T>()
    for (element in this) if (predicate(element)) result.add(element)
    return result
}

fun <T, R> List<T>.map(transform: Function1<T, R>): List<R> {
    val result = mutableListOf<R>()
    for (element in this) result.add(transform(element))
    return result
}

fun <T> createList(size:Int, element:Function1<Int, T>) : List<T> {
    val result = mutableListOf<T>()
    for (i in 0 until size) result.add(element(i))
    return result
}

fun <T, R> List<T>.fold(initial: R, folder: Function2<R, T, R>): R {
    var result = initial
    for (element in this) result = folder(result, element)
    return result
}
```

Окей, а какие самые естественные операции над функциями? Взять обратную у нас, к сожалению, не получится, а вот композицию — вполне.

```
class Composition<A, B, C>(
    val f: Function1<A, B>,
    val g: Function1<B, C>
) : Function1<A, C> {
    override fun invoke(arg: A): C = g(f(arg))
}

fun <A, B, C> comp(
    f: Function1<A, B>, g: Function1<B, C>
): Function1<A, C> = Composition(f, g)
```

Можно было, собственно говоря, и не выносить отдельную функцию, но это здесь не просто так. Мы создаём новый класс, но используем его только в одном месте. Мало того, что ему нужно — о, ужас! — придумать имя, так это имя потом загрязняет общее пространство имён. Поэтому есть возможность создать так называемый *анонимный класс*:

```
val isEven = object : Function1<Int, Boolean> {
    override fun invoke(arg: Int): Boolean = arg % 2 == 0
}
listOf(1, 2, 3, 4, 5, 6).filter(
    object : Function1<Int, Boolean> {
        override fun invoke(arg: Int): Boolean = arg % 2 == 0
    }
)
```

Более того, компилятор Kotlin достаточно умён, чтобы обнаружить все переменные снаружи, которые нужно встроить в конструктор этого класса:

```
fun <A, B, C> composeAnon(
    f: Function1<A, B>,
    g: Function1<B, C>
) = object : Function1<A, C> {
    override fun invoke(arg: A): C = g(f(arg))
}
```

> Вообще говоря, если разбирать байт-код, то мы увидим, что создался отдельный класс с именем `FunctionsGeneralizedKt$composeAnon$1` и двумя полями: `f` и `g`. При вызове функции composeAnon создаётся объект этого класса с соответствующими параметарами.

Собственно, лямбды

На этом можно было бы и закончить — в конце концов, анонимных классов вполне достаточно для функционального программирования. Но мы пойдём дальше.

Представьте, что вы пользуетесь тремя библиотеками, и в каждой из них свои интерфейсы, предназначенные для этих целей? (Кстати, интерфейсы с одним методом обычно называют *функциональными*). А ведь, памятуя о сложном устройстве JVM, для эффективности нужно держать и примитивные специализации: например, предикат, поскольку он возвращает примитивный тип, хорошо бы иметь отдельным интерфейсом, а не общий с генериками.

Собственно говоря, именно так и поступает Java Development Kit: в нём есть *сорок три* различных интерфейса, предназначенных для этого:

`BiConsumer`, `BiFunction`, `BinaryOperator`, `BiPredicate`, `BooleanSupplier`, `Consumer`, `DoubleBinaryOperator`, `DoubleConsumer`, `DoubleFunction`, `DoublePredicate`, `DoubleSupplier`, `DoubleToIntFunction`, `DoubleToLongFunction`, `DoubleUnaryOperator`, `Function`, `IntBinaryOperator`, `IntConsumer`, `IntFunction`, `IntPredicate`, `IntSupplier`, `IntToDoubleFunction`, `IntToLongFunction`, `IntUnaryOperator`, `LongBinaryOperator`, `LongConsumer`, `LongFunction`, `LongPredicate`, `LongSupplier`, `LongToDoubleFunction`, `LongToIntFunction`, `LongUnaryOperator`, `ObjDoubleConsumer`, `ObjIntConsumer`, `ObjLongConsumer`, `Predicate`, `Supplier`, `ToDoubleBiFunction`, `ToDoubleFunction`, `ToIntBiFunction`, `ToIntFunction`, `ToLongBiFunction`, `ToLongFunction`, `UnaryOperator`.

Да их все запомнить — уже нелёгкая задача... А ведь все эти “функции” принимают не более двух аргументов!

Поэтому в Kotlin сделаны специальные *функциональные типы*, буквально делающие то, что нам нужно: например, сложение чисел, так как это функция из двух `Int` в `Int`, описывается типом `(Int, Int) -> Int`. И передавать туда можно довольно-таки много, что. Но давайте по порядку, приведём наши функции в порядок:

```
fun <T> List<T>.filter(predicate: (T) -> Boolean): List<T> {
    val result = mutableListOf<T>()
    for (element in this) if (predicate(element)) result.add(element)
    return result
}

fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
    val result = mutableListOf<R>()
    for (element in this) result.add(transform(element))
    return result
}

fun <T> createList(size: Int, element: (Int) -> T): List<T> {
    val result = mutableListOf<T>()
    for (i in 0 until size) result.add(element(i))
    return result
}

fun <T, R> List<T>.fold(initial: R, folder: (R, T) -> R): R {
    var result = initial
    for (element in this) result = folder(result, element)
    return result
}

//fun <A, B, C> compose(f: (A) -> B, g: (B) -> C): (A) -> C = ...
```

Так, а как мы *передаём* эти значения? Что нам возвращать из `compose`? Здесь есть несколько вариантов.

1. Lambda literal. Выглядит это так:

   ```
   fun <A, B, C> compose(f: (A) -> B, g: (B) -> C): (A) -> C = { arg: A -> g(f(arg)) }
   ```

   Здесь есть несколько нюансов. Во-первых, здесь тип `arg` — `A` — известен из контекста, так как возвращаемый тип `compose` — `(A) -> C`, значит, это функция, принимающая один аргумент типа `A`. Во-вторых, если у лямбды ровно один аргумент, то можно не писать его имя — по умолчанию оно будет `it`.

   ```
   fun <A, B, C> compose(f: (A) -> B, g: (B) -> C): (A) -> C = { g(f(it)) }
   ```

2. Ссылка на функцию

   ```
   fun isEven(x: Int) = x % 2 == 0

   filter(listOf(1, 2, 3, 4, 5), ::isEven)

   => : List<Int> = [2, 4]
   ```

3. Ссылка на метод

   Например, сложение чисел описывается методом:

   ```
   operator fun plus(other: Int): Int
   ```

   , объявленным внутри класса `Int`. Как мы понимаем, на самом деле здесь два аргумента: явный `other` и неявный `this`. Поэтому функция `Int::plus` имеет тип `(Int, Int) -> Int`.

   ```
   listOf(1, 2, 3, 4, 5).fold(0, Int::plus)

   => : Int = 15
   ```

4. Ссылка на метод объекта

   Только что же было? Погодите. Это почти то же самое, только вместо имени класса слева от `::` пишем объект этого класса. Например,

   ```
   createList(5, "abc"::plus)

   => : List<String> = [abc0, abc1, abc2, abc3, abc4]
   ```

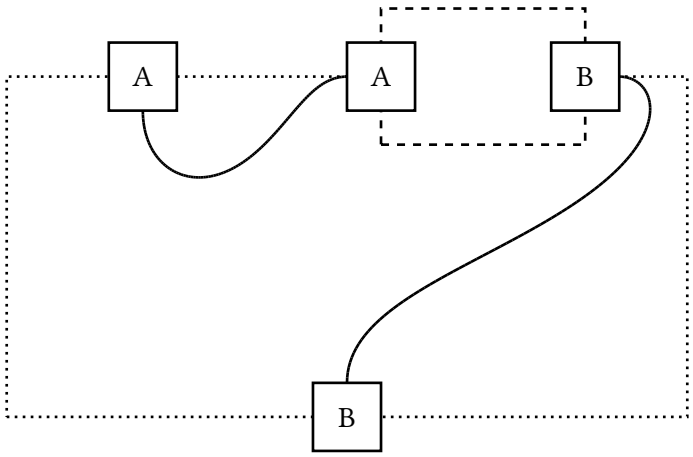
Итак, с этим разобрались. Что дальше?

Давайте рассмотрим следующие функциональные типы. Что они из себя представляют? Какие функции с подобной семантикой можно написать?

- `(A, (A) -> B) -> B`

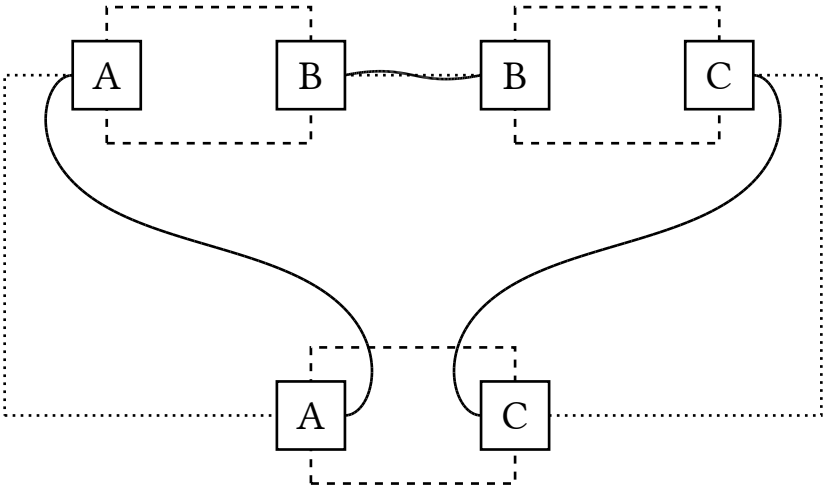
  У нас есть значение типа `A` и функция, переводящая `A` в `B`. Хм, что бы это могло быть! Конечно, это применение функции к значению:

  ```
  fun <A, B> apply(value: A, func: (A) -> B): B = func(value)
  ```



- `((A) -> B, (B) -> C) -> ((A) -> C)`

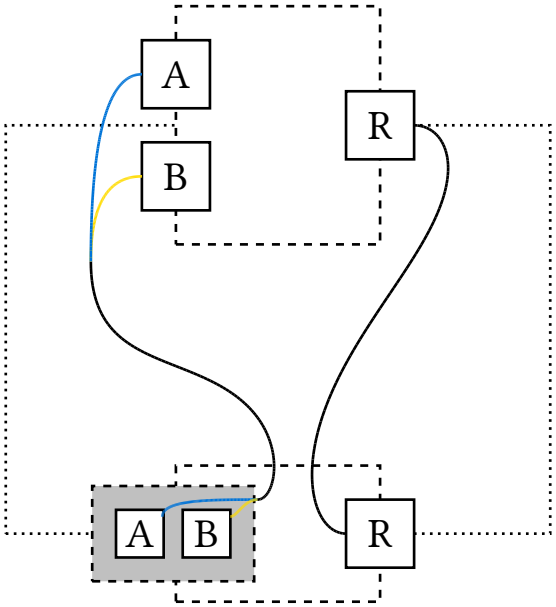
  Это же уже знакомая нам композиция.



- `((A, B) -> R) -> ((Pair<A, B>) -> R)`

  У нас была функция, принимающая `A` и `B`, теперь же это функция, принимающая пару из `A` и `B`. Ну, например мы можем разобрать пару на запчасти и скормить функции по частям.

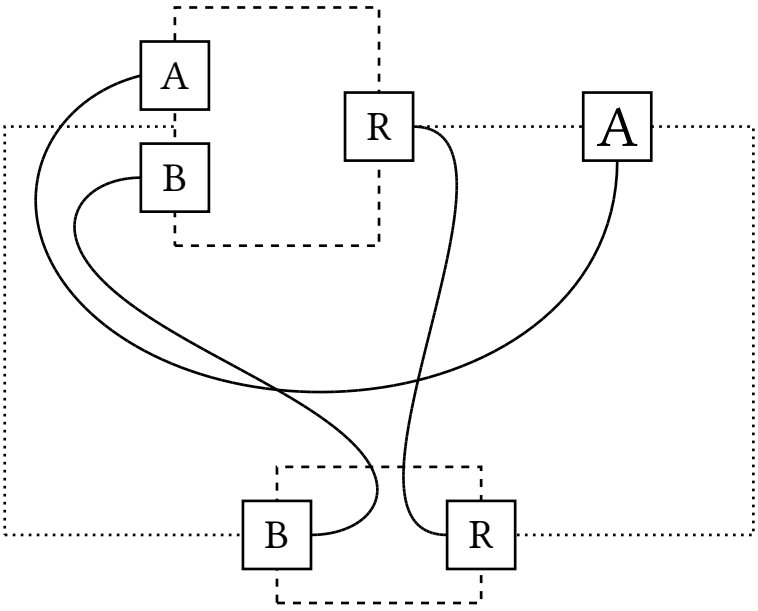
```
fun <A, B, R> decompose(func: (A, B) -> R): (Pair<A, B>) -> R =
        { func(it.first, it.second) }
```



- `((A, B) -> R, A) -> ((B) -> R)`

  Вот, а это уже более интересно. Почему? Фактически, мы подставляем значение в функцию от двух переменных, получая функцию от одной.

```
fun <A, B, R> substitute(func: (A, B) -> R, a: A): (B) -> R = { func(a, it) }
```



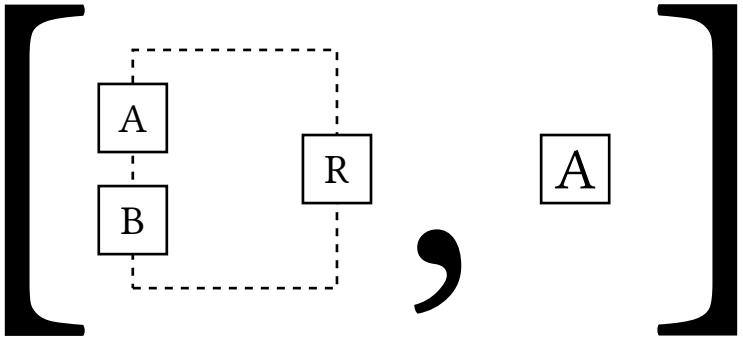
А что в этом такого необычного?

На самом деле, подстановка настолькое естественное действие, что им пользуются для того, чтобы хранить замыкания. Замыкания? А, помните, анонимные объекты, которые

автомагически захватывали переменные из контекста? Как, например, в `composeAnon`. Вот в случае лямбд это и называется замыканиями. Это функция, ссылающаяся на внешние данные.

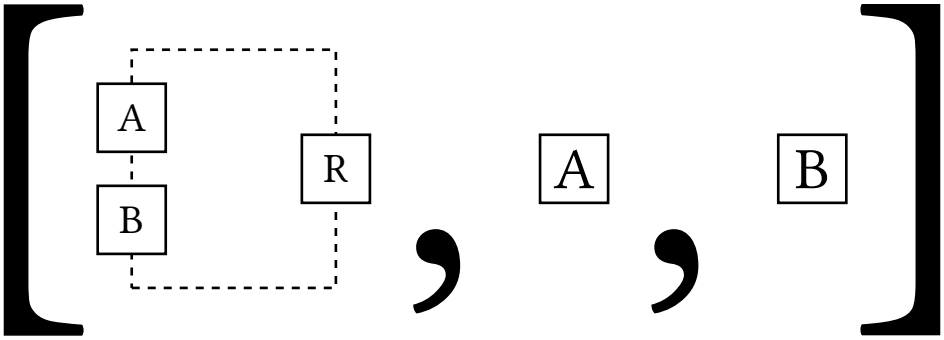
Так что там про хранение? Так вот, один из вариантов — анонимные объекты, которые мы рассматривали раньше. А другой, более простой — просто хранить список из указателя на функцию, и части аргументов!

Так что просто вот так хранить:



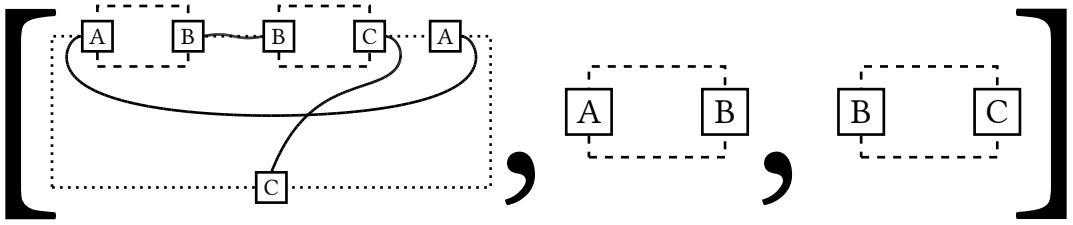
...и понимать это как функцию `(B) -> R`. Кстати говоря, функция `(A, B) -> R` тоже может быть представлена таким вот списком. И так далее.

А что бы означал вот такой список?



Так это просто функция, ничего не принимающая и возвращающая `R`. Причём, если нулевой элемент — честная, ничего не трогающая в реальном мире, функция, то весь лист — функция, всегда возвращающая одно и то же значение. А это почти то же самое, что просто значение типа `R`... В общем говоря, порядок вычислений можно попросту так и записывать — вложенными листами.

И так можно хранить и другие лямбды из тех, что мы уже рассмотрели. `apply` возвращает не лямбду, не интересно, следующий. `compose` возвращает функцию `(A) -> C`. Как бы это понять?



Как бы тут уже не вывихнуть мозг, конечно. Ну да ладно, это не очень нужно понимать, чтобы уметь этим пользоваться... Так, кстати, мы приходим к идее, что `(A) -> ((B) -> R)` и `(A, B) -> R` — это, по сути, одно и то же. И скобки в первой записи принято опускать: `(A) -> (B) -> R`.