The road to Svartalfheim

Ляпин Д.Р., a.k.a. LDemetrios

2024

Что это такое?

Это статья/книга о том, как я создаю свой язык программирования, который называется Svart (шв. Тёмный). Это не столько пошаговый гайд, сколько дневник разработки. Я публикую его в самом начале написания и буду обновлять по мере продвижения.

Что это такое? 2
Язык Svart 3
Конкретнее о концепции 4
- Основной синтаксис 4
- Классы, их наследование и интерфейсы. 4
- Алгебраические типы данных 5
- Типы-функции, типы-массивы и типы-кортежи. 5
- Ассоциированные типы. 6
- Self и final типы-параметры. 7
- Типы-функции, массивы и кортежи… снова. 8
- Операции над типами 8
- Ограничения в generic параметрах 10
- Перегрузка операторов 11
- Внешние перегрузки интерфейсов 13
- Объекты 13
- Наследование ассоциированных типов 14
- Varargs 15

Язык Svart

Этот язык совмещает в себе преимущества разных известных мне языков. Основной вклад внесли Kotlin и Rust, но также немного влияния имели C++, JavaScript, Prolog и другие языки. Конкретнее говоря:

- Kotlin:
  - Общий вид синтаксиса.
  - Компилируемость под JVM, и, соответственно, сборка мусора из коробки.
  - Оболочка над Java reflection, представляющая каноничные типы.
- Rust:
  - Интерфейсы скорее похожи на трейты. В частности, есть трейты для операторов.
  - Кортежи как полноценные типы.
  - Ассоциированные типы.
  - Полноценные алгебраические типы.
  - Дженерики (не как в Java или Kotlin, без стирания).
- JavaScript:
  - toString для замыканий выдаёт осмысленную информацию.
- Prolog:
  - Общий подход к паттерн-матчингу.
- C++:
  - Типы могут параметризоваться числами, с некоторыми возможностями compile-time вычислений. К сожалению, это потенциально означает, что компиляция может никогда не завершиться…

Конкретнее о концепции

В первую очередь, хочется написать как можно больше различного, наполненного разнообразными фичами, кода, чтобы понять, что должно компилироваться и как работать, а что — не должно компилироваться или падать с ошибкой.

Основной синтаксис

… позаимствуем из Котлина. Придётся подождать пару страниц, пока я объясняю его для непосвящённых. Входной точкой является функция `main()`. Вывод осуществляется встроенной функцией `println`, строковые литералы ограничиваются двойными кавычками. Точка с запятой не требуется.

```
fun main() {
    println("Hello, world!")
}
```

А вот прежде чем говорить про ввод, сначала придётся поговорить про слишком много всего.

А пока — переменные и постоянные, циклы и условия, комментарии и аннотации типов — всё без изменений.

```
fun main() {
    val delta : Int = 2
    var n = 9 // Auto inferred type Int
    var fact = 1L // Auto inferred type Long
    while (n > 0) {
        fact *= n
        n -= delta
    }
    println(fact) // Prints 945
}
```

Мы сосчитали 9!! = 945. Отлично.

Ещё у нас есть…

Классы, их наследование и интерфейсы.

```
interface Animal {
    fun speech() : String
}

class Dog : Animal {
    override fun speech() = "Bark!"
}

class Cat : Animal {
    override fun speech() = "Meow!"
}
```

Это всё бывает generic:

```
interface List<+T> {
    val size : Int

    fun get(index: Long) : T
}
```

Они, конечно же, поддерживают declaration-site variance. Ну и всё в таком же духе. В отличие от Котлина, мы можем объявить *абстрактные* `static` методы. Это поможет нам потом, с generics. При этом у каждого `static` метода автоматически появится ещё и не статическая перегрузка:

```
abstract class Base {
    static fun name() : String = "Base"
}

class A : Base {
    override static fun name() = "A"
}

class B : Base {
    override static fun name() = "B"
}
```

Зачем это? Затем, что мы можем вызывать методы, не имея инстанса. В частности, на параметрах типа метода, в том числе и выведенных автоматически:

```
fun <C : Base, X : C, Y : C> commonName(x : X, y : Y) = C.name()
```

Позже мы научимся это делать и без таких махинаций с дженериками, но сработает это так:

```
val a = A()
val b = B()
println(commonName(a, a)) // X = A, Y = A, C = A, prints "A"
println(commonName(a, b)) // X = A, Y = B, C = Base, prints "Base"
```

С другой стороны, имея инстанс, мы сможем вызвать его собственный метод.

Алгебраические типы данных

Собственно, что написано на упаковке:

```
enum Result<T, E> {
    Success(T), Error(E)
}
```

Типы-функции, типы-массивы и типы-кортежи.

В том числе, именованные кортежи. Вот тут-то и начнутся нововведения, а потом и расхождения. Давайте пока условимся, что у нас есть функции `listOf()` и `mutableListOf()`, как в Котлине, ибо мне пока лень придумывать названия для стандартной библиотеки. Может быть, впоследствии это поменяется.

```
fun <T> asList(arr: T[]) : List<T> {
    val result = mutableListOf<T>()
    for (el in arr) {
        result.add(el)
    }
    return result
}
```

Хоть мне это и не очень нравится, но определить параметры типов надо до имени функции, так как у нас есть extension functions.

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

Хотелось бы целиком разделить фазы парсинга и вывода типов, и это налагает ограничения на код. Например, мы здесь на уровне парсинга увидим, что `transform` — это параметр, а не функция, а значит, будем пытаться вызывать на ней операторный метод `invoke`. И, если вдруг у нас есть такая ситуация:

```
fun method(x: Int) : Int = x

fun main() {
    val method = "abc"
    println(method(1))
}
```

В принципе, оно могло бы и скомпилироваться: вызвать метод, объявленный выше. Но нет, в данном скоупе `method` — это строка.

Да, и именованные кортежи:

```
fun <T> List<T>.findIndexed(condition: T => Boolean) : (index: Int, value: T)? {
    for i in 0 .. this.size() {
        if (condition(this[i])) {
            return (index: i, value: this[i])
        }
    }
    return null
}
```

Во-первых, есть nullable типы, которые нужно явно маркировать. Во-вторых, при конструировании результата нужно явно прописать имена аргументов. Зачем? Затем, что тип `(index: Int, value: T)` — подтип `(Int, T)`, и если мы напишем `(i, this[i])` — мы сконструировали второй тип.

Также замечу, что скобки в объявлении `for` не обязательны, оператор `..` предполагает, что конец — исключительно, и в обычном форматировании его стоит выделять пробелами.

Ассоциированные типы.

Их удобство не так уж и очевидно в простых программах, поэтому я постараюсь это описать гораздо позже. Когда мы попытаемся компилятор Svart написать на Svart же. Пока же скажем так — это своего рода постоянная, которую можно запросить на наследнике типа так же, как обычно запрашивают просто переменную на инстансе типа, но в compile-time.

```
interface Common {
    type Associated
    fun paramInstance() : Self::Associated
}

class A : Common {
    override type Associated = Int
    override fun paramInstance() = 1
}

class B : Common {
    override type Associated = String
    override fun paramInstance() = "1"
}

fun <G : Common> genericMethod(something: G) {
    val param = something.paramInstance()
    // Auto inferred G::Associated
}
```

Self и final типы-параметры.

Да, у нас есть тип `Self`, который означает “тип, которому принадлежит `this`”. Очевидно, если его использовать в качестве возвращаемого значения, проблем не возникнет, а вот в качестве параметра...

```
interface Negatable {
    fun negate() : Self
}

interface Monoid {
    static fun one() : Self
    fun mul(another: Self) : Self
}
```

Теперь, допустим, у нас `Int` и `Double` оба реализуют эти два интерфейса.

```
fun checkReversability(x: Negatable) : Boolean {
    val negX = x.negate() // Type is Negatable
    return x == negX.negate()
}
fun checkNeutrality(x: Monoid) : Boolean {
    val one = x.one() // Type is Monoid, despite the fact `one` is static
    val x1 = x.mul(one) // Oops, Compilation Error!
    return x == x1
}
```

Почему же ошибка компиляции? Потому что интерфейс требует реализовать метод `mul(Self)`, а не `mul(Monoid)`. Соответственно, `Int` реализует `mul(Int)`, `Double` реализует `mul(Double)`. Но мы хотим, чтобы так работало?

```
fun <T : Monoid> checkNeutrality(x: T) : Boolean {
    val one = x.one() // Type is T
    val x1 = x.mul(one) // Still Compilation Error!
    return x == x1
}
```

Снова проблема. Потому что никто не запрещает подставить `T = Monoid`, а это приводит к уже известным проблемам… Мы как-то хотим разрешить подставлять только те типы, у которых нет наследников.

```
fun <final T : Monoid> checkNeutrality(x: T) : Boolean {
    val one = x.one() // Type is T
    val x1 = x.mul(one) // OK now
    return x == x1
}
```

Типы-функции, массивы и кортежи… снова.

… да, они у нас есть, мы это уже выяснили. Но во имя операций над типами, у них есть “длинный” синтаксис, консистентный с остальными:

- `T[]` это `Array<T>`. Ничего интересного, на самом деле.
- `(T, U) => R` это `Function<(T, U)>`. Аргумент типа — это кортеж типов аргументов функции. А где тип результата? Он нигде не появляется в сигнатуре функций, поэтому реализовывать две версии интерфейса с разными аргументами для одного класса не очень осмысленно. Поэтому тип результата — это ассоциированный тип.

  ```
  class Something : (Int, Double) => String {
      override operator fun invoke(arg0: Int, arg1: Double) : String = "abc"
  }
  ```

  — это то же самое, что…

  ```
  class Something : Function<(Int, Double)> {
      override type Result = String
      override operator fun invoke(arg0: Int, arg1: Double) : String = "abc"
  }
  ```

- Кортежи… А тут сложно. Это своего рода лист типов, используемый в компайл-тайме. Поэтому, во-первых, у нас есть синтетические типы, обозначающие кортежи длины n: `(A, B, C, D, E, F)` это `Hexad<A, B, C, D, E, F>`. Во-вторых, у нас есть специальный тип `Cons`: `(T, U, R)` — это `Cons<T, Cons<U, Cons<R, Nullad>>>`. И наконец, у нас есть общий тип `Tuple`, от которого они все наследуются. А у `Tuple` компиляторно определено ассоциированный тип `Reduce`, позволяющий совершать операции над типами.

Операции над типами

Для понимания этой главы рекомендуется сначала преисполниться лямбда-исчислением.

Первым делом надо заметить, что ассоциированы с типом могут быть не только единичные типы, но и семейства типов:

```
class Sample {
    type <T> Associated = Comparable<(T, T)>
}
```

Тогда `Sample::Associated<Int>` это то же самое, что `Comparable<(Int, Int)>`. Этот же синтаксис мы можем использовать для задания top-level псевдонимов:

```
type <T> Predicate = T => Boolean
```

Теперь давайте посмотрим, что же мы хотим иметь. Давайте научимся добавлять элемент в конец списка. Для тех, кто не знаком с тем, что такое `reduce`:

```
Reduce([], Func, Acc) = Acc
Reduce(Cons(Head, Tail), Func, Acc) = Func(Head, Reduce(Tail, Func, Acc))
```

Так, например, сумма списка — это `reduce(list, +, 0)`. Хорошо, у нас есть `(A, B, C)`. Нам нужна какая-то функция и какой-то аккумулятор, которые удовлетворяет следующим “функциональным уравнениям”:

```
Func(C, Acc) = X
Func(B, X) = Y
Func(A, Y) = (A, B, C, D)
```

Хочется сразу сказать, что пусть `Func = Cons`. Тогда сразу:

```
Cons(C, Acc) = (C, D)
Cons(B, (C, D)) = (B, C, D)
Cons(A, (B, C, D)) = (A, B, C, D)
```

Отсюда вывод: `Acc = (D,)`.

Тогда

```
type <List : Tuple, Last> Append = List::Reduce<Cons<*, *>, (Last,)>
```

Звёздочки здесь — указание на то, что мы передаём `Cons` как функцию над типами, а не тип. Аналогичным образом давайте развернём список.

```
Func(C, Acc) = X
Func(B, X) = Y
Func(A, Y) = (C, B, A)
```

Понятно, что здесь должно быть `Func`, равное только что написанному `Append`:

```
Append(C, ()) = (C,)
Append(B, (C,)) = (C, B)
Append(A, (C, B)) = (C, B, A)
```

```
type <List : Tuple> Reverse = List::Reduce<Append<*, *>, ()>
```

А теперь хотим написать функцию высшего порядка. Как бы это сделать? Как принять семейство типов в качестве аргумента? Сделаем это так: пусть все функции над типами — синтетические типы, наследники

```
interface TypeFunction<Bounds : Tuple> {
    type Result = +Any?
    abstract type <T: Bounds> Invoke : Self::Result
}
```

Соответственно, например, `Append<*, *>` — это синтетический тип

```
class `Append<*, *>` : TypeFunction<(Tuple, Any?)> {
    override type <T: (Tuple, Any?)> Invoke = Append<T::First, T::Second>
}
```

Итак, мы хотим написать фильтр. Поступим в лучших традициях лямбда-исчисления:

```
sealed interface TypeBoolean : TypeFunction<(Any?, Any?)>

class TypeTrue : TypeBoolean {
    type <T: (Any?, Any?)> Invoke = T::First
}

class TypeFalse : TypeBoolean {
    type <T: (Any?, Any?)> Invoke = T::Second
}

type <T> TypePredicate = TypeFunction<(T,), Result = +TypeBoolean>

type <Value, List : Tuple, Pred : TypePredicate<(Value,)>> CondCons =
    Pred::Invoke<(Value,)>::Invoke<(Cons<Value, List>, List)>

type <List : Tuple, Pred : TypePredicate<Common<List>>> Filter =
    List::Reduce<CondAppend<*, *, Pred>, ()>
```

Как вы могли заметить, здесь при передаче `CondAppend` мы не все параметры пометили `*`. Это `CondAppend`, в который заранее подставили третий аргумент, равный `Pred`. Также есть `type <T : Tuple> Common` — “наиболее узкий общий тип”, встроенная в компилятор функция.

И да, конечно же, как у нас поддерживаются extension functions, так поддержим и extension types!

```
type <A, T : Tuple> Cons<A, T>::First = A
type <A, B, T : Tuple> Cons<A, Cons<B, T>>::Second = B
```

Ограничения в generic параметрах

Давайте придумаем generic класс.

```
class OrderedEntry<N : Number, T : Comparable<T>>(val num : N, val value : T)
```

Теперь мы хотим написать какой-нибудь метод, который его принимает.

```
fun doSomething(param: OrderedEntry<...>, ...)
```

Ага, нам придётся ввести соответствующие переменные.

```
fun <N : Number, T : Comparable<T>> doSomething(param: OrderedEntry<N, T>, other: T)
```

… я бы хотел ввести немного сахара для этого дела. Здесь обе переменные имеют *ровно* такие ограничения, которые требуются для того, чтобы использовать их как параметры `OrderedEntry`. Введём обозначение с вопросительным знаком для того, чтобы вводить такие переменные:

```
fun <T : Comparable<T>> doSomething(param: OrderedEntry<?N, T>, other: T)
```

И даже, если у нас эта переменная используется в объявлении в другом месте, разрешим использовать `?` не более одного раза.

```
fun doSomething(param: OrderedEntry<?N, ?T>, other: T)
```

Так, например, теперь можем написать `First` и `Second` по-другому, короче:

```
type Cons<?A, ?T>::First = A
type Cons<?A, Cons<?B, ?T>>::Second = B
```

Заодно сделаем так: постановка `?` без последующего имени означает то же самое, что и переменная с уникальным именем. В общем, как в Прологе:

```
type Cons<?A, ?>::First = A
type Cons<?, Cons<?B, ?>>::Second = B
```

Перегрузка операторов

Конечно, куда же без неё?

С одной стороны, в Котлине это делается лаконично, ключевым словом `operator`, а в Расте — длинным (не менее, чем в шесть строчек) `impl Trait`. С другой, в Расте знание о том, что класс реализует оператор, получаемо через информацию о реализации соответствующего трейта, и это можно использовать для написания красивого обобщённого кода.

```
impl <U, T : Add<U>> Add<Vector<U>> for Vector<T> {
    type Output = Vector::<<T as Add<U>>::Output>;

    fn add(self, rhs: Vector<U>) -> Self::Output {
        ...
    }
}
```

Так вот. Совместим это. Написание `operator fun` для класса автоматически добавляет соответствующий интерфейс к предкам этого класса. В частности, это означает, что *возвращаемый тип операторной функции нужно специфицировать явно*. Потому что я хочу разделить этапы вывода типов. Ну да об этом позже, когда начнём его писать…

Назовём возвращаемый тип `Result` для всех операторов ниже.

Унарные операторы:

Оператор	Сахар для
`-a`	`a.negate()`
`!a`	`a.not()`
`~a`	`a.inv()`

Здесь нет унарного плюса… Может быть, добавлю позже.

Бинарные операторы:

Оператор	Сахар для
`a + b`	`a.add(b)`
`a - b`	`a.sub(b)`
`a * b`	`a.mul(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`
`a & b`	`a.bitAnd(b)`
`a | b`	`a.bitOr(b)`
`a ^ b`	`a.xor(b)`
`a && b`	`a.bitAnd { b }`
`a || b`	`a.bitOr { b }`

Заметим, что `&&` и `||` принимают правым аргументом функцию, возвращающую нужное значение. Это нужно для возможности ленивых вычислений, как это происходит с настоящими булевыми значениями.

Также у нас есть интересные операторы `?.`, `?:` и `!!` для обеспечения null-safety. А в Rust был интересный `enum Result`, у которого есть методы `map`, `unwrap_or_else`, `unwrap`. В общем… Это ровно то, что нам нужно.

Оператор	Сахар для
`a?.b()`	`a.safeCall { it.b() }`
`a ?: b`	`a.orElse { b }`
`a!!`	`a.orElseThrow()`

Например, для `Result` можем сделать так:

```
operator fun <T, R, E> Result<T, E>.safeCall(func : T => R) : Result<R, E> =
    match (this) {
        Success(?x) => Success(func(x))
        Error(?e) => Error(e)
    }
```

М-м-м… мы не поговорили про паттерн-матчинг пока? Ну, вы поймёте. Похоже на Rust, но не очень…

```
operator fun <T, R> Result<T, ?>.orElse(another : () => R) : Common<(T, R)> =
match(this) {
    Success(?x) => x
    Error(?) => another()
}
operator fun <T> Result<T, ?>.orElseThrow() : T = match(this) {
    Success(?x) => x
    Error(?e) => throw AssertionError(e )
}
```

Таким же образом, кстати, можно обрабатывать умные ссылки!

Так, теперь… инкремент и декремент:

Оператор	Сахар для
`a++`	`a.postInc(b)`
`a--`	`a.postDec(b)`
`++a`	`a.preInc(b)`
`--a`	`a.preDec(b)`

В отличие от Котлина, здесь это разные методы. Иначе это слишком неудобно для мутабельных классов…

Операторы с присваиванием (`+=`, `-=` и так далее) делаем так: сначала ищем метод с соответствующим именем с суффиксом (`addAssign`, `subAssign`, и так далее), а, если не находит, преобразуем в присваивание с применением (`a = a + b`). Если есть и то, и другое — warning (не ошибка).

Операторы индексирования и вызова:

Оператор	Сахар для
`a()`	`a.invoke()`
`a(b)`	`a.invoke(b)`
`a(b, c)`	`a.invoke(b, c)`
`a[b]`	`a.get(b)`
`a[b, c]`	`a.get(b, c)`
`a[b] = x`	`a.set(b, x)`
`a[b, c] = x`	`a.set(b, c, x)`

Стоит лишь заметить, что `invoke` как раз отвечает за интерфейс `Function`, который является отражением функциональных типов (`(T, U) => R` и так далее).

Ещё есть `range`, отвечающий за `...` Операторы `==` и `!=`, остаются за методом `equals`, `===` и `!==` — встроенная в компилятор проверка на идентичность ссылок. И из интересного остались только операторы сравнения.

Здесь у нас есть один интерфейс `Ordered`, с одним же методом `compareTo`. В отличие от привычного `Comparable`, он будет возвращать один из `enum Order { Less, Equal, Greater }`. И есть ещё интерфейс `PartialOrder`, метод которого может также вернуть `Unknown`. Всё это преобразуется понятным образом, я тут пока не документацию пишу, в самом-то деле…

Внешние перегрузки интерфейсов

Во-первых, заметим, что у нас нет стирания, а значит, нам никто не мешает перегружать интерфейс с разными параметрами. И разный набор интерфейсов для по-разному параметризованного типа. Например, `Vector<String>` реализует `Add<Vector<String>>`, а `Vector<Int>` реализует и `Add<Vector<Int>>`, и `Sub<Vector<Int>>`. С другой стороны, мы не хотим, чтобы реализации конфликтовали. Поэтому позаимствуем *orphan rule*: мы можем определить реализацию ~~трейта~~ интерфейса для ~~структуры~~ класса, только если мы определили одно или другое. Соответственно, синтаксис пусть будет такой же:

```
impl <T> Add<T> for List<T> {
    ...
}
```

Хм… ладно, я думал, у меня есть что ещё сказать по этому поводу…

Объекты

Это Singleton классы… ничего особо интересного. Разве что, вместо аннотации `@JvmStatic` сделаем то же самое ключевым словом `static`.

```
object Sample {
    fun a() = 1
    static fun b() = "abc"
}
```

Компилируется в (обойдёмся без байт-кода, просто аналогичным Java кодом):

```
public final class Sample {
    private Sample() {}

    public static final Sample INSTANCE = new Sample();

    public int a() {
        return 1;
    }

    public static String b() {
        return "abc";
    }
}
```

а тем временем

```
static object Sample {
    fun a() = 1
    static fun b() = "abc" // Warning: unnessessary `static`
}
```

Компилируется в:

```
public final class Sample {
    private Sample() {}

    public static int a() {
        return 1;
    }

    public static String b() {
        return "abc";
    }
}
```

Наследование ассоциированных типов

Сделаем следующим образом: если тип объявлен явно, перегрузить его нельзя. Но можно поставить соответствующий знак вариантности, обещая использовать его только в нужно вариантностью.

```
abstract class Base {
    type In = Number
    type +Co = Number
    type -Contra = Number

    abstract fun usingCo() : Self::Co
    abstract fun usingContra(x: Self::Contra)
    abstract fun incorrectUsage() : Self::Contra // Error: can't use in covariant
position
}
```

```
class Derive : Base() {
    // can't override type In
    override type Co = Int
    override type Contra = Any

    override fun usingCo() : Int
    override fun usingContra(x: Any)
}
```

Varargs

По определённым причинам хотелось бы, чтобы можно было параметризовать функцию кортежем. И чтобы он (этот кортеж) был “выпрямлен”. Например, так:

```
fun <T : Tuple> sample(x: Int, y: String, zs: ...T)
```

Тогда:

```
sample(1, "abc", 3) // T is (Int,)
sample(2, "def", 5, "ghi") // T is (Int, String)
```

Но так же должна быть возможность и массив потребовать. Поэтому, для консистентности, нужно будет указать *и* что это массив, *и* что это vararg.

```
fun sample(xs: ...Any[])
```