

Санкт-Петербургский государственный университет

КЫЛЬЧИК Иван Викторович

Выпускная квалификационная работа

Разработка и исследование механизмов
интерпретации мультиплатформенных
Kotlin-функций во время компиляции

Уровень образования: магистратура

Направление *09.04.04 «Программная инженерия»*

Основная образовательная программа *ВМ.5666.2019 «Программная инженерия»*

Научный руководитель:
к.ф.-м.н., Д.Ю. Булычев

Рецензент:
Исследователь “JetBrains Research“, к.ф.-м.н. А.В. Подкопаев

Санкт-Петербург
2021

Saint Petersburg State University

Ivan Cilcic

Master Thesis

Research and development of multiplatform Kotlin-functions interpreting techniques at compile time

Education level: master

Speciality *09.04.04 «Software Engineering»*

Programme *BM.5666.2019 «Software Engineering»*

Scientific supervisor:
Ph.D. D.Yu. Boulytchev

Reviewer:
Researcher at “JetBrains Research“, Ph.D. A.V. Podkopaev

Saint Petersburg
2021

Оглавление

1. Введение	4
2. Постановка задачи	6
3. Обзор предметной области	7
3.1. Основные понятия	7
3.2. Применение механизма выполнения функций на этапе компиляции в Kotlin	8
3.3. Обзор механизма выполнения функций на этапе компи- ляции в других языках программирования	20
3.4. Выводы и анализ результатов	29
4. Реализация прототипа	31
4.1. Требования, выдвигаемые к прототипу	31
4.2. Способы реализации прототипа	32
4.3. Архитектура прототипа	35
4.4. Особенности реализации	38
4.5. Основные ограничения и правила	42
4.6. Архитектура оптимизатора языка	43
5. Апробация	46
5.1. Демонстрация результатов	46
5.2. Проверка корректности	51
5.3. Тестирование производительности	52
6. Заключение	54
Список литературы	55

1. Введение

Компиляторы для современных языков программирования должны уметь эффективно использовать ресурсы компьютера при генерации программ, выполнять сложные оптимизации и учитывать архитектуру процессора, для которого выполняется трансляция. Основные показатели, которые стремятся оптимизировать компилятор, – это размер итоговой программы и время ее выполнения. В общем случае задача генерации оптимального целевого кода из исходной программы является неразрешимой [1], следовательно, при разработке компилятора необходимо вводить дополнительные эвристики, которые позволят получить эффективный код.

Язык Kotlin — это относительно молодой язык программирования, разрабатываемый компанией JetBrains. Из основных особенностей языка можно выделить его простоту, лаконичность и полную совместимость с Java [8]. Kotlin стремительно развивается, и одно из направлений его развития — это улучшение компилятора.

Компилятор извлекает различную информацию о программе, например, размеры массивов, имена файлов, имена классов, точное место в коде где была вызвана функция и т.д. Эту информацию можно эффективно использовать для оптимизации итоговой программы. Например, часть функций (или даже отдельных выражений) можно выполнить еще на этапе компиляции, тем самым на этапе выполнения программы результат уже будет известен, и программа будет исполняться быстрее, а её код будет компактнее.

Язык Kotlin позволяет помечать свойства (properties [12]) модификатором `const`, который означает, что значение этого свойства должно быть вычислено еще на этапе компиляции. Таким образом язык Kotlin уже частично поддерживает механизм интерпретации функций во время компиляции, но в данный момент возможности такого механизма сильно ограничены. Во-первых, накладываются сильные ограничения на инициализатор свойства. Разрешены только базовые операции над примитивными типами и строками [10]. Во-вторых, даже с такими силь-

ными ограничениями пользователи не всегда могут заранее сказать будет ли корректным использование той или иной функции в инициализаторе, что иногда вводит их в заблуждение о том как пользоваться модификатором `const` [7, 9]. И, в-третьих, у пользователей нет никакой возможности самостоятельно определить функции, которые могут быть выполнены на этапе компиляции.

В данной работе предлагается расширить существующий механизм интерпретации, позволив выполнять на этапе компиляции большее количество функций. Такая модификация позволит смягчить существующие ограничения на использование модификатора `const`, а также позволит применять извлеченную информацию о программе более грамотно.

2. Постановка задачи

Целью данной работы является создание нового механизма интерпретации функций на этапе компиляции, который позволит оптимизировать программы, созданные существующим компилятором языка Kotlin. Для её достижения были поставлены следующие задачи:

- обосновать необходимость добавления механизма вычисления функций на этапе компиляции в язык Kotlin;
- провести анализ существующих языков, использующих данный механизм;
- исследовать возможные способы реализации вычисления на этапе компиляции в Kotlin;
- разработать прототип, способный производить вычисления функций на этапе компиляции;
- провести апробацию прототипа.

3. Обзор предметной области

В этой главе вводятся основные понятия и аббревиатуры, которые будут использованы в работе. Приводятся примеры из языка Kotlin, на основе которых будет показана необходимость расширения механизма вычисления функций на этапе компиляции. Также в этой главе представлен краткий обзор языков программирования, в которых уже имеется поддержка такого механизма.

3.1. Основные понятия

Вычисление функций во время компиляции, – это один из видов оптимизаций, выполняемых компилятором. Такие оптимизации применяются уже давно. Выделим их основные виды [2]:

1. Constant folding (CF) – свертка констант. Пожалуй, это один из первых и простейших видов оптимизации. CF подразумевает вычисление простейших арифметических или логических операций над литералами. Например, выражение “10 * 20” будет заменено значением “200”. Так или иначе CF реализован в большинстве современных компиляторов, включая компилятор языка Kotlin.
2. Compile time function execution (CTFE) – выполнение функций на этапе компиляции. Такая оптимизация подразумевает замену вызова функции на вычисленный результат, если известны все необходимые аргументы и функция является корректной с точки зрения выполнения на этапе компиляции. Чаще всего внутри этих функций разрешается использование ветвлений, циклов и прочих возможностей языка.
3. Compile time evaluation (CTE) – исполнение на этапе компиляции. Этот вид оптимизации очень похож на предыдущий, отличием является то, что мы не ограничены вызовами функции. Если при известных входных аргументах часть кода может быть вычислена

на этапе компиляции, то этот код будет заменен результатом выполнения. В качестве примера можно привести следующий кусок кода на языке C:

```
1 int x = 2 * 7;  
2 int y = 15 - x / 2;
```

В этом примере, при использовании CTE оптимизации, инициализация переменной “y” заменится выражением “8”. Если переменная “x” больше нигде не используется, то ее объявление будет удалено.

Основная цель, которую преследуют такого рода оптимизации — это ускорение работы программы за счет частичного вычисления результатов выполнения кода на этапе компиляции. В качестве дополнительной цели можно указать уменьшение размеров исполняемой программы за счет устранения неиспользуемых участков кода.

3.2. Применение механизма выполнения функций на этапе компиляции в Kotlin

Покажем основные плюсы от введения в Kotlin механизма CTFE.

1. Улучшение модификатора const.

Как было сказано, Kotlin поддерживает модификатор `const`, который позволяет производить вычисления на этапе компиляции в виде свертки констант, т.е. вычислимыми считаются только простейшие арифметические и логические операции над примитивными типами или операции преобразования. Например:

```
1 const val answer = 2 * 21  
2 const val msg = 'Hello World!'  
3 const val calculated = answer + 45
```

Полный список допустимых операций указан в спецификации [10]. И все же пользователь не всегда понимает причину того, почему одна функция может быть выполнена на этапе компиляции, а

другая нет. Например, следующий код выполнится без ошибок, и вычисления пройдут на этапе компиляции [7]:

```
1 const val x = 1.toByte()
```

В то же время следующий код не может быть скомпилирован:

```
1 const val y = '1'.toByte()
```

Такое поведение объясняется тем, что метод “toByte” класса Int является непосредственно методом этого класса, а метод “toByte” для класса String — это extension-функция, свертка констант для которой не предусмотрена. С точки зрения пользователя такое поведение является необычным.

Это приводит к мысли, что есть необходимость явно помечать функции, которые можно использовать для вычислений на этапе компиляции. Например, введя некий модификатор или аннотацию, которая бы указывала на то, что функция является корректной для CTFE.

2. Вычисление функции на этапе компиляции.

Самым простейшим применением CTFE является вычисление функции один раз, на этапе компиляции. Такое применение очевидно исходя из самого определения CTFE, тем не менее это одна из мотиваций введения такого механизма в Kotlin. Например:

```
1 fun fib(n: Int) : Int {  
2     if (n ≤ 1) return n  
3     return fib(n - 1) + fib(n - 2)  
4 }  
5  
6 const val n2 = fib(2)  
7 const val n10 = fib(10)  
8 const val n20 = fib(20)
```

Компилятор может посчитать значение функции “fib” один раз на этапе компиляции, тем самым предотвращая повторное вычисление на этапе выполнения программы.

3. Подстановка отдельных ветвей кода.

Язык Kotlin поддерживает модификатор `inline`, суть которого заключается в том, что функция помеченная этим модификатором будет “вставлена” на место ее вызова. К сожалению, поддержка `inline` в Kotlin является очень ограниченной и в данный момент она эффективна для lambda-функций, но не для значений. Например, на примере ниже функция “`toNanoSeconds`” помечена как `inline`, но никакого выигрыша в производительности не будет просто потому что текущий компилятор не знает какие можно провести оптимизации.

```
1 inline fun Long.toNanoSeconds(from : TimeUnit): Long = when (from) {  
2     TimeUnit.NANOSECONDS → this  
3     TimeUnit.MICROSECONDS → this * 1_000  
4     TimeUnit.MILLISECONDS → this * 1_000_000  
5     TimeUnit.SECONDS → this * 1_000_000_000  
6     TimeUnit.MINUTES → this * 1_000_000_000 * 60  
7     TimeUnit.HOURS → this * 1_000_000_000 * 60 * 60  
8     TimeUnit.DAYS → this * 1_000_000_000 * 60 * 60 * 24  
9 }  
10  
11 val threeSecondsInNanos =  
12     getNonConstValue().toNanoSeconds(from = TimeUnit.SECONDS)
```

Данная функция производит преобразование единиц времени в наносекунды. В качестве параметра в эту функцию передается число, отображающее единицы времени, а также значение enum-класса, описывающее, что это за единицы времени. Последний параметр является известным на этапе компиляции, а единицы времени нет. Но даже несмотря на это, используя оптимизации на этапе компиляции мы можем упростить вызов этой функции до выражения вида

```
1 val threeSecondsInNanos = getNonConstValue() * 1_000_000_000
```

4. Улучшенные встроенные функции (“power” inline).

Приведем еще один пример, который показывает возможность повышения производительности программы за счет inline-модификаций. Объявим класс *Bar* с тремя полями типа *Int*. Предположим, что пользователь хочет разрешить сравнения таких объектов, для этого надо реализовать интерфейс *Comparable* по этим полям. В простейшем случае надо описать логику сравнения трех полей:

```
1 class Bar(val a: Int, val b: Int, val c: Int) : Comparable<Bar> {
2
3     override fun compareTo(other: Bar): Int {
4         val aCompare = a.compareTo(other.a)
5         if (aCompare != 0) return aCompare
6
7         val bCompare = b.compareTo(other.b)
8         if (bCompare != 0) return bCompare
9
10        val cCompare = c.compareTo(other.c)
11        if (cCompare != 0) return cCompare
12
13        return 0
14    }
15 }
```

Стандартная библиотека Kotlin содержит функцию, которая может реализовать объект типа *Comparator* по указанным полям. Выглядеть это будет следующим образом:

```
1 class Foo(val a: Int, val b: Int, val c: Int) : Comparable<Foo> {
2
3     private val comparator = compareBy<Foo>(Foo::a, Foo::b, Foo::c)
4
5     override fun compareTo(other: Foo) = comparator.compare(this, other)
6 }
```

Минус такого вызова — это производительность. Покажем это, написав небольшой тест на производительность с использованием фреймворка JMH.

```
1 @BenchmarkMode(Mode.AverageTime)
```

```

2 @OutputTimeUnit(TimeUnit.NANOSECONDS)
3 @State(Scope.Thread)
4 @Warmup(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
5 @Measurement(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
6 open class CompareToBenchmark {
7
8     @Benchmark
9     fun benchmarkBar(bh: Blackhole) {
10         val a = Bar(Random.nextInt(), Random.nextInt(), Random.nextInt()
11         ↪ )
12         val b = Bar(Random.nextInt(), Random.nextInt(), Random.nextInt()
13         ↪ )
14         bh.consume(a.compareTo(b))
15         bh.consume(a)
16         bh.consume(b)
17     }
18
19     @Benchmark
20     fun benchmarkFoo(bh: Blackhole) {
21         val a = Foo(Random.nextInt(), Random.nextInt(), Random.nextInt()
22         ↪ )
23         val b = Foo(Random.nextInt(), Random.nextInt(), Random.nextInt()
24         ↪ )
25         bh.consume(a.compareTo(b))
26         bh.consume(a)
27         bh.consume(b)
28     }
29
30     @Benchmark
31     fun benchmarkBarCreation(bh: Blackhole) {
32         bh.consume(Bar(Random.nextInt(), Random.nextInt(), Random.
33         ↪ nextInt()))
34         bh.consume(Bar(Random.nextInt(), Random.nextInt(), Random.
35         ↪ nextInt()))
36     }
37
38     @Benchmark
39     fun benchmarkFooCreation(bh: Blackhole) {
40         bh.consume(Foo(Random.nextInt(), Random.nextInt(), Random.
41         ↪ nextInt()))

```

```
35         bh.consume(Foo(Random.nextInt(), Random.nextInt(), Random.  
36         ↪ nextInt()))  
37     }
```

Здесь присутствуют четыре теста:

- (a) измерение времени работы метода “compareTo” для случая компаратора, написанного самостоятельно;
- (b) измерение времени работы метода “compareTo” для случая создания компаратора средствами стандартной библиотеки;
- (c) измерение времени создания объекта Bar;
- (d) измерение времени создания объекта Foo.

Последние два теста нужны, чтобы исключить время на создание объектов.

Все рассматриваемые здесь и далее тесты выполнялись на рабочей станции с ОС “macOS Big Sur“, с процессором “Intel Core i9“ частотой 2.3 ГГц и с 32 Гб оперативной памяти.

Результаты измерений представлены ниже.

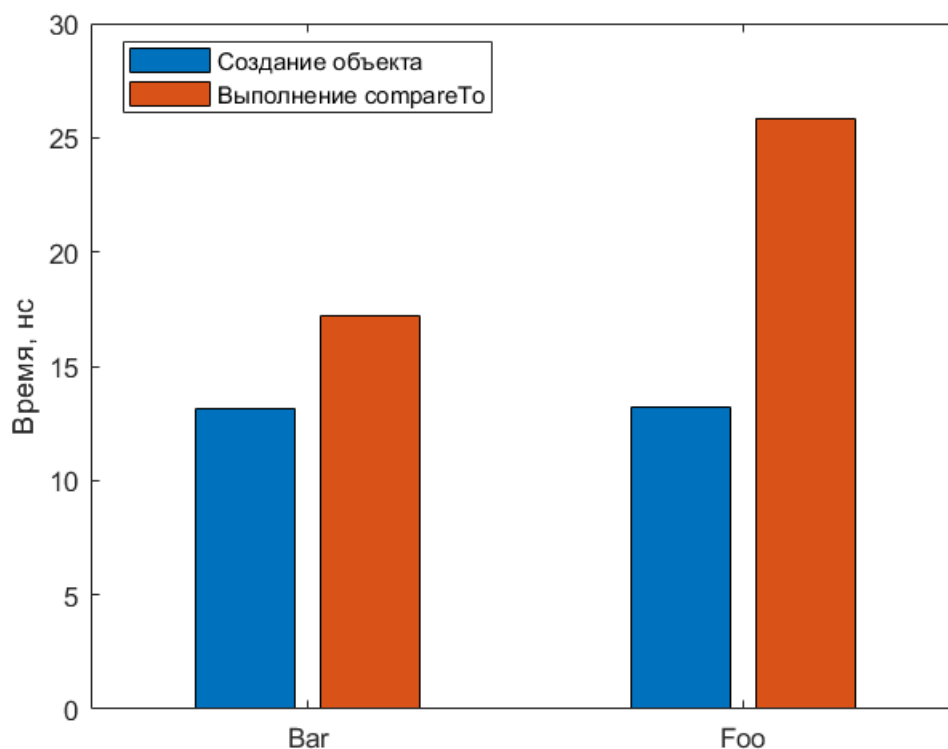


Рис. 1: Результаты выполнения теста на производительность для метода compareTo

Итоговое время работы времени работы:

- ≈ 4 нс на вызов для компаратора, написанного самостоятельно;
- ≈ 13 нс на вызов для компаратора, сделанного средствами стандартной библиотеки.

Таким образом использование метода “compareTo” для создания компаратора замедляет производительность примерно в 3 раза. Как уже было сказано, такое поведение объясняется тем, что “compareTo” не является встраиваемой (inline) функцией.

Попробуем описать то, каким именно образом может происходить вот такая inline-оптимизация и почему она является настолько эффективной. Сперва произойдет процесс “встраивания” функций. Результат такого действия будет примерно следующим:

```

1 private val comparator: Comparator<Foo> = run {
2     val selectors = arrayOf(Foo::a, Foo::b, Foo::c)
3     require(selectors.size > 0)
4     Comparator<Foo> { a, b →
5         for (fn in selectors) {
6             val v1 = fn(a)
7             val v2 = fn(b)
8             val diff = (v1 as Comparable<Any>).compareTo(v2)
9             if (diff != 0) return@Comparator diff
10        }
11        return@Comparator 0
12    }
13 }

```

Компилятор смог избавиться от лишних вызовов, но в этом коде все еще присутствуют места, которые можно улучшить. Например, можно развернуть цикл и избавиться от ненужного массива. Уже после этой оптимизации можно заменить операции со ссылками на свойства (знак “::”) на непосредственный вызов этого свойства.

```

1 private val comparator: Comparator<Foo> =
2     Comparator<Foo> { a, b →
3         val diff_a = a.a.compareTo(b.a)
4         if (diff_a != 0) return@Comparator diff_a
5         val diff_b = a.b.compareTo(b.b)
6         if (diff_b != 0) return@Comparator diff_a
7         val diff_c = a.c.compareTo(b.c)
8         if (diff_c != 0) return@Comparator diff_c
9         return@Comparator 0
10    }

```

Производительность вот такого кода будет такой же, как и у компаратора, написанного самостоятельно.

5. Псевдо-функции.

Еще один интересный пример это введение псевдо-функций. Например, можно создать вот такую функцию:

```
1 inline fun SourceLocation(): String = TODO('Intrinsic')
```

На этапе компиляции вызов такой функции будет заменен строкой содержащей название файла и номер строки, где был произведен вызов (возможен вариант возвращения `data` класса). Основное применение такой функции – это быстрое протоколирование (logging). За счет того, что вычисления произойдут на этапе компиляции, во время выполнения программы не будет никаких дополнительных затрат и такое протоколирование потенциально будет очень быстрым.

6. Рефлексия на этапе компиляции.

На этапе компиляции известно очень много информации о классах и их свойствах. Этим можно воспользоваться для реализации набора `Compile-Time Reflection` функций и методов. Они будут работать аналогично обычным методам рефлексии, но за счет того что функции будут выполняться на этапе компиляции, они будут более легковесны и вычислены один раз.

В качестве базовых примеров можно привести следующие функции:

- Функция, возвращающая имя переданного свойства.
- Функция, возвращающая имя текущего файла (наподобие `SourceLocation`).
- По заданному объекту получить набор его полей, функций, конструкторов.
- По заданной аннотации получить все функции, помеченные этой аннотацией.

Этот список не полный и в процессе разработки будет пополняться. Последний пример особенно важен и имеет реальное практическое применение в бес серверных вычислениях (например, `Amazon AWS`). Суть таких вычислений заключается в том, что для них не

выделяется отдельный сервер, вычисления производятся в момент запроса [4]. Одним из главных преимуществ такого сервиса является “оплата по факту”: мы платим только за то время, которое было использовано для исполнения нашего кода.

Если использовать тяжеловесные методы reflection, то мы очевидно будем переплачивать, тем не менее время работы программы может быть оптимизировано за счет Compile-Time Reflection методов.

7. Делегирование отдельных методов

Язык Kotlin вводит такое понятие как “data class”. Это обычный класс, но для него по умолчанию генерируется ряд полезных методов, например, геттеры, сеттеры, методы copy, hashCode, equals и toString.

Пример такого класса, а также результат вызова метода toString, приведены ниже.

```
1 data class Data(val a: Int, val b: Double, val c: String)
2 fun main() {
3     println(Data(1, 2.0, 'Three')) // Data(a=1, b=2.0, c=Three)
4 }
```

Не всегда пользователь хочет создавать data класс, чаще всего ему нужен обычный класс. При этом, если пользователь хочет воспользоваться реализацией метода toString как у data класса, ему надо писать это вручную. В стандартной библиотеке могла бы быть какая-нибудь функция вида:

```
1 fun Any.toStringAsDataClass(): String {
2     val kClass = this::class
3     val properties = kClass.members.filterIsInstance<KProperty1<Any,
4     ↪ *>>()
5     val propertiesFormatted = properties.joinToString { "it.name ={it.
6     ↪ invoke(this)}" }
7     return "kClass.simpleName(propertiesFormatted)"
8 }
```

Проблема такой реализации это опять же производительность. Показать это можно используя фреймворк JMH.

```
1 @BenchmarkMode(Mode.AverageTime)
2 @OutputTimeUnit(TimeUnit.NANOSECONDS)
3 @State(Scope.Thread)
4 @Warmup(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
5 @Measurement(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
6 @Suppress('UNUSED')
7 open class ToStringBenchmark {
8     @Benchmark
9     fun benchmarkDataToString(bh: Blackhole) {
10         val a = Random.nextInt()
11         val b = Random.nextDouble()
12         val c = String(Random.nextBytes(10))
13         val data = Data(a, b, c)
14         bh.consume(data.toString())
15     }
16
17     @Benchmark
18     fun benchmarkNotDataToString(bh: Blackhole) {
19         val a = Random.nextInt()
20         val b = Random.nextDouble()
21         val c = String(Random.nextBytes(10))
22         val notData = NotData(a, b, c)
23         bh.consume(notData.toString())
24     }
25
26     @Benchmark
27     fun benchmarkDataCreation(bh: Blackhole) {
28         val a = Random.nextInt()
29         val b = Random.nextDouble()
30         val c = String(Random.nextBytes(10))
31         val data = Data(a, b, c)
32         bh.consume(data)
33     }
34
35     @Benchmark
36     fun benchmarkNotDataCreation(bh: Blackhole) {
37         val a = Random.nextInt()
```

```

38     val b = Random.nextDouble()
39     val c = String(Random.nextBytes(10))
40     val notData = NotData(a, b, c)
41     bh.consume(notData)
42 }
43 }
44
45 data class Data(val a: Int, val b: Double, val c: String)
46 class NotData(val a: Int, val b: Double, val c: String) {
47     override fun toString(): String = toStringAsDataClass()
48 }

```

Суть теста абсолютно аналогична тому что было в тесте для “compareTo”.

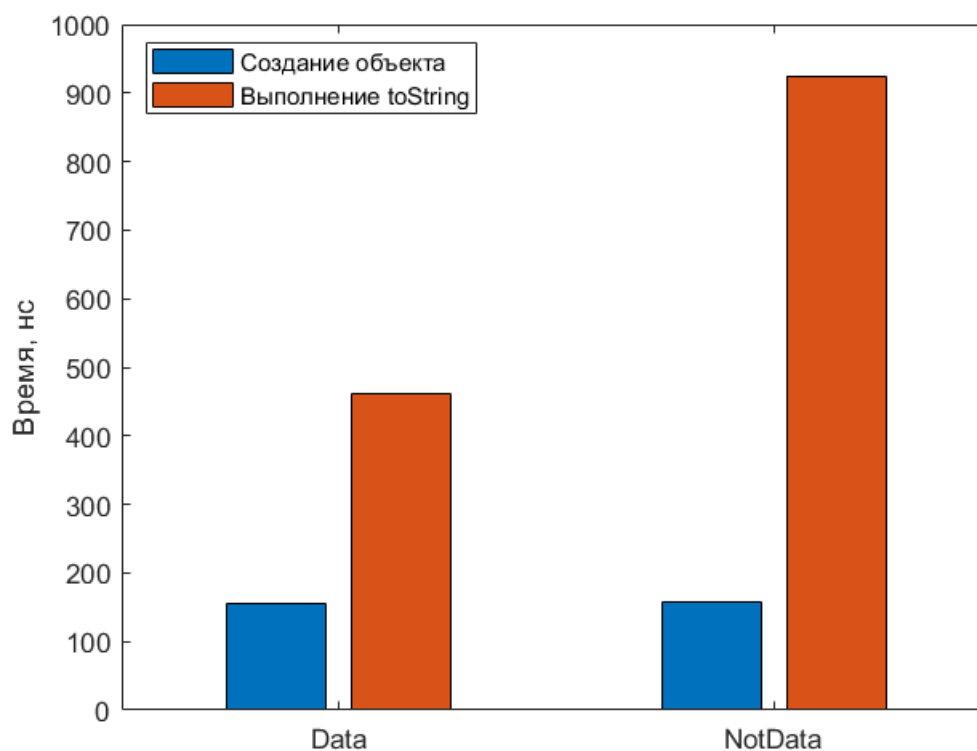


Рис. 2: Результаты выполнения теста на производительность для метода toString

Итоговое время работы время работы:

- ≈ 300 нс на вызов, для метода toString из data-класса;
- ≈ 760 нс на вызов, для метода toString сделанного средствами стандартной библиотеки.

Таким образом видно, что написанный вручную toString уступает по скорости примерно в 2.5 раза.

Наличие таких примеров подтверждает, что в языке Kotlin не хватает механизма CTFE.

3.3. Обзор механизма выполнения функций на этапе компиляции в других языках программирования

Рассмотрим примеры того, как и в каких языках применяются вычисления на этапе компиляции.

1. C++

В ранних версиях языка C++ для CTFE применялось метапрограммирование на шаблонах [14]:

```

1 template <int N>
2 struct Factorial {
3     enum { value = N * Factorial<N - 1>::value };
4 };
5
6 template <>
7 struct Factorial<0> {
8     enum { value = 1 };
9 };
10
11 void Foo() {
12     int x = Factorial<0>::value; // == 1
13     int y = Factorial<4>::value; // == 24
14 }
```

В стандарте языка C++ 11 был введен модификатор “constexpr”, который позволил писать более простой и понятный код:

```
1 #include <stdio>
2
3 constexpr int Factorial(int n) { return n ? (n * Factorial(n - 1)) : 1;
   ↪ }
4
5 constexpr int f10 = Factorial(10);
6
7 int main() {
8     printf("%d\n", f10);
9     return 0;
10 }
```

Данный модификатор обозначает, что если на вход константного выражения подаются константы или другие константные выражения, то вычисления могут быть выполнены на этапе компиляции. Из предыдущего примера видно, что этот модификатор может быть использован в разных контекстах:

- **Constexpr-переменная.** В данном случае модификатор означает, что значение переменной будет вычислено на этапе компиляции. Его использование также неявно помечает переменную как `const`. Constexpr-переменная должна удовлетворять следующим требованиям:
 - Должна иметь литеральный тип [5].
 - Должна быть сразу инициализирована.
 - Выражение для инициализации должно быть вычисляемо на этапе компиляции.
- **Constexpr-функция.** Модификатор указывает на то, что функция может быть вычислена на этапе компиляции. Вычисления произойдут только если все переданные аргументы являются значениями периода компиляции. Изначально на constexpr-функции были наложены жесткие ограничения:
 - Она не может быть виртуальной.

- Она должна возвращать литеральный тип, т.е. `void` вернуть нельзя.
- Все параметры должны иметь литеральный тип.
- Тело функции должно содержать только `static_assert`, `typedef`, `using` и ровно один `return`, который может содержать только литералы или `constexpr`-переменные и `constexpr`-функции.

В версии языка C++ 14 эти ограничения сильно смягчили. Этим стандартом разрешается использовать в телах `constexpr`-функций любые выражения, кроме:

- Ассемблерных вставок.
- Оператора `goto`.
- Объявление переменных не литерального типа.
- Объявление `static` переменных.
- Объявление `thread_safe` переменных.
- **Constexpr-конструктор.** Позволяет компилятору инициализировать объект на этапе компиляции. На него накладываются те же ограничения, что и на `constexpr`-функции, но с добавлением одного условия: все нестатические члены класса и члены базовых классов должны быть инициализированы каким-либо образом.

В редакции стандарта C++17 все лямбда-функции, которые удовлетворяют условиям для `constexpr`-функций, неявным образом заносятся в класс `constexpr`-выражений [11].

Стандарт языка C++20 расширяет возможности константных вычислений еще больше:

- Разрешается использование `constexpr`-деструкторов.
- Добавляются два новых модификатора:
 - `constexpr`. Применяется к функциям и указывает на то, что вызов каждой такой функции обязан быть вычислен

на этапе компиляции. Если по какой-то причине это сделать не удастся, то это считается ошибкой компиляции.

- `Constinit`. Объявляет переменную со статической или потоковой длительностью хранения. Это означает, что если для создания такой переменной потребуется динамическая инициализация, то программа считается некорректной.

Подводя итоги можно сказать только то, что константные вычисления языка C++ — это узконаправленный, но мощный инструмент. Правильное использование модификатора `constexpr` позволяет ускорить работу программы.

2. D

Другим хорошим примером языка, который способен производить вычисления на этапе компиляции, является язык D.

Главным отличием CTFE в языке D от C++ является то, что такие вычисления выполняются неявно. Нет необходимости указывать модификатор `constexpr`, компилятор сам разберется что можно, а что нельзя вычислить. Приведем пример CTFE для вычисления факториала:

```
1 int factorial(int n) {  
2     if (n == 0)  
3         return 1;  
4     return n * factorial(n - 1);  
5 }  
6  
7 // computed at compile time  
8 enum y = factorial(0); // == 1  
9 enum x = factorial(4); // == 24
```

Использование ключевого слова “enum” говорит компилятору, что вычисления должны быть выполнены на этапе компиляции. Для того чтобы такие вычисления были корректными, аргументы

функции тоже должны быть вычислимы. Существуют и другие контексты, на которых происходит CTFE:

- Инициализация статической переменной или manifest-константы (примером такой константы является `enum`).
- Статическая инициализация полей структуры или класса.
- Статические массивы.
- Аргумент в шаблоне.
- В таких конструкциях как: `static if`, `static foreach`, `static assert`.

Еще одним отличием от CTFE в C++ является наличие псевдопеременной `__ctfe` типа *boolean*. Ее значение принимает `true` тогда и только тогда, когда вычисление значения данной переменной происходит в процессе компиляции.

Компиляцию программы в языке D можно разбить на примерные 4 фазы [6]:

- (a) Лексический анализ и парсинг. Это стандартный этап для большинства компиляторов. Происходит преобразование текста кода в абстрактное синтаксическое дерево AST (abstract syntax tree). AST представляет собой структуру программы как ее видит компилятор и содержит все необходимое для дальнейшего преобразования программы в исполняемый машинный код.
- (b) Манипулирование AST. На этом этапе происходит структурное изменение AST. Примером может послужить подстановка конкретных типов в шаблоны.
- (c) Семантический анализ. Происходит сопоставление узлов AST и функция языка D, например, сопоставление идентификатора и конкретного типа.
- (d) Генерация исполняемого кода. Семантически проанализированный код преобразуется в машинный.

Проблема CTFE заключается в том, что этот механизм должен происходить между 2 и 3 фазами. Если перейти к 3 фазе, то изменить AST уже не получится.

Попробуем на простом примере рассмотреть, как в языке D реализован подобный механизм.

```
1 int ctfeFunc(bool b)
2 {
3     if (b)
4         return 1;
5     else
6         return 0;
7 }
8
9 // the enum forces the compiler to evaluate the function in CTFE
10 enum myInt = ctfeFunc(true);
```

Сперва происходит построение AST. На этом этапе нет ни входных аргументов, ни ввода, ни вывода, нет никакой информации, чем на самом деле являются идентификаторы. Для того чтобы получить эти данные, необходимо перейти на следующую фазу – семантического анализа. Но, как уже было сказано, перейдя к следующей фазе, уже не получится изменить AST. В примере выше, переменная “myInt” помечена как “enum“, а это значит ее значение должно быть вычислено на этапе компиляции. Но пока мы не перешли на этап семантического анализа, мы не можем даже говорить о каких-то вычислениях. Что делать? Стоит заметить, что AST для функции “ctfeFunc” полностью построено и для него можно выполнить этап анализа.

На этом этапе произойдет частичный семантический анализ. Компилятор продолжит процесс CTFE и выполнит семантический анализ функции “ctfeFunc” независимо от изначального AST. Теперь остается только передать результат этой операции и необходимые аргументы в систему CTFE, отвечающую непосредственно за интерпретацию. Результат “1” передается абстрактному дереву

для завершения компиляции.

Подведем итог. CTFE в языке D ничем не уступает таковому в C++. Можно даже сказать, что с какой-то стороны такой подход даже лучше. Из основных преимуществ можно выделить следующие:

- Пользователь пишет функцию только 1 раз и не задумывается о том, будет ли необходимость в CTFE.
- Компилятор гораздо лучше сможет определить, где надо вычислять функцию, а где нет, без использования подсказки в виде модификатора `constexpr`.
- Без нагромождения модификаторов код выглядит чище и понятнее.

Однако есть и ограничения. Самым сильным из них является невозможность создать объект или выделить память (создание статических массивов является корректным).

3. Scala

Еще один интересный подход к CTFE представлен в языке Scala в виде макросов. Макросом можно назвать специальную функцию, которая вызывается компилятором в процессе компиляции. Используя эти функции, мы имеем доступ к API компилятора и к AST программы. Макросы предоставляют нам следующие возможности:

- Анализ типов объектов, включая generic типы (анализ AST).
- Создание новых объектов (добавление нового узла к AST).
- Полный доступ к методам объекта (доступ к дочерним узлам AST).

Разберем принцип работы макросов на примере программы, вычисляющей значение факториала:

- (a) Сперва необходимо определить обычный метод, который мы бы использовали для вычисления факториала:

```
1 def regularFactorial(n: Int): Int =  
2   if (n == 0) 1  
3   else n * regularFactorial(n - 1)
```

- (b) Объявим макрос используя следующий шаблон:

```
1 def m(x: T): R = macro implRef
```

- m - имя макроса
- x – параметр макроса
- R – возвращаемый тип
- macro – ключевое слово
- implRef – другой метод, который предоставляет реализацию макроса.

В нашем примере макроопределением будет:

```
1 import scala.language.experimental.macros  
2 def factorial(n: Int): Int = macro factorial_impl
```

- (c) Следующий шаг – реализация макроса, где будет описана вся его функциональность. Отличие от реализации обычного метода заключается в том, что макрос принимает и возвращает AST. Во всем остальном это обычный метод. Чтобы реализовать вычисление факториала в процессе компиляции необходимо развернуть AST, применить написанный ранее метод “regularFactorial” и сформировать новую структуру AST. Следующий код представляет реализацию макроса factorial:

```
1 import scala.reflect.macros.blackbox.Context  
2 def factorial_impl(c: Context)(n: c.Expr[Int]): c.Expr[Int] = {  
3   import c.universe._  
4   n match {  
5     case Expr(Literal(Constant(nValue: Int))) =>  
6       // если в качестве аргумента была передана константа  
7       val result = regularFactorial(nValue)
```

```

8      c.Expr(Literal(Constant(result)))
9      case _ =>
10         // если в качестве аргумента было передано что-либо еще
11         println("Not_supported!")
12         throw new NotImplementedError("wrong "AST)
13     }
14 }

```

Стоит отметить, что этот код будет выполнен в процессе компиляции и даже сообщение “Not supported! “ будет выведено.

Этот пример наглядно показывает, как можно применять макросы в Scala для CTFE, хотя и очевидно, что макросы можно применять не только для этого. Тем не менее это один из подходов для реализации вычислений в процессе компиляции.

4. Dart

В языке Dart нет CTFE в том виде, как в C++ или Scala. Здесь нет возможности произвести вычисления функций на этапе компиляции. Тем не менее Dart поддерживает модификатор `const`, который используется для инициализации переменных в процессе компиляции. Объекты, помеченные как `const`, являются неизменяемыми (`immutable`). Константой считается одно из следующих выражений:

- Значение примитивного типа.
- Значение примитивного типа, полученное в результате основных арифметических или логических операций.
- Константный конструктор.

В качестве примера можно привести следующий код:

```

1 main() {
2     const double primitive_double = 1.0; //primitive double
3     const List speed = const ['slow', 'medium', 'fast']; //const list
  object
4     radar_longitude += 2; //Error - can't modify

```

```
5 | radar_modes.add('Crazy_Fast'); //Error - cannot add to an  
   | immutable list  
6 | }
```

Dart поддерживает также модификатор `final`. Его основное отличие заключается в том, что значения `final` переменных присваиваются в процессе выполнения программы.

Основное назначение модификатора `const` – ускорение работы программы за счет кэширования объектов в процессе компиляции. Основные принципы работы константных объектов:

- (a) Все константные объекты вычисляются в процессе компиляции.
- (b) От объектов сохраняется их отпечаток (snapshot).
- (c) При загрузке приложения константные объекты загружаются в память.
- (d) Здесь не может произойти ленивой инициализации.
- (e) Эти объекты не собираются сборщиком мусора.

Исследование языка Dart предоставляет еще один взгляд на то как может быть реализовано СТФЕ и более сложная операция – сохранение объектов в процессе компиляции.

3.4. Выводы и анализ результатов

1. Приведенные примеры из языка Kotlin наглядно показывают, что расширение механизма СТФЕ способно значительно улучшить качество целевого кода, генерируемого компилятором. Дальнейший обзор других языков, использующих СТФЕ, показывает, что подобная функциональность бывает полезна и применяется во многих современных языках программирования.
2. Во всех приведенных примерах из других языков, реализация СТФЕ зависит от внутренней структуры компилятора. Таким образом можно сделать вывод, что для реализации СТФЕ в Kotlin

потребуется исследовать конкретно компилятор Kotlin, а не компиляторы других языков и уже на основе этого анализа принимать решение об архитектуре прототипа.

3. Также в этих примерах видно, что есть два основных способа реализовать СТФЕ на уровне языка: явное вычисление по запросу, например с использованием модификатора “constexpr” и неявное вычисление, где решение принимает компилятор как в языке D. Таким образом возникает вопрос: какой вариант подходит языку Kotlin?

На основе данного обзора можно сформулировать требования, которым должен соответствовать прототип.

4. Реализация прототипа

Результат данной работы представлен в виде реализованного интерпретатора языка. В данной главе формируются требования, выдвигаемые к прототипу, описывается его архитектура, основные особенности и ограничения. Также здесь будет показано то, как используя написанный интерпретатор применить его для целей оптимизации, описанных в разделе №3.2.

4.1. Требования, выдвигаемые к прототипу

На основе проведенного обзора можно выдвинуть основные требования, которым должен удовлетворять прототип, способный выполнять вычисления на этапе компиляции в Kotlin.

1. Независимость от платформы.

Как уже было сказано, Kotlin — это мультиплатформенный язык. Это означает, что исходный код может быть скомпилирован под несколько платформ, например, JVM, JS или Native. В то же время в описанных выше примерах не было никакого платформенно-специфичного кода. Все потому что такая функциональность должна работать одинаково, независимо от платформы.

2. Вычисление функций по явному запросу.

Следуя примеру языка D, можно сделать все вычисления на этапе компиляции неявными. Очевидными плюсами является то, что компилятор может лучше пользователя понять где и что можно вычислить на этапе компиляции. В то же время очевидными минусами являются понижение скорости компиляции (как следствие того, что каждую функцию надо анализировать на предмет выполнимости) и трудности для пользователя, связанные с проблемами осознания что может быть вычислено, а что нет. Вторая проблема особенно важна, так как ее игнорирование противоречит тому, что было сказано при анализе языка Kotlin в примере

№1.

3. Поддержка широкого репертуара исполняемых функций.

В простейшем случае прототип может работать только со встроенными функциями. Очевидно, что так будет проще написать и поддерживать прототип, но такой вариант не подходит. Во-первых, Kotlin уже умеет вычислять встроенные функции, так как это необходимое условие для инициализации свойств с модификатором `const`. Во-вторых, все описанные ранее оптимизации предполагают более сложные вычисления и если мы действительно хотим удовлетворить все потребности, то нужно расширить круг вычисляемых функций.

4. Возможность расширения прототипа.

В отличие от остальных, данное требование является скорее дополнительным. Описанные ранее примеры не являются исчерпывающим списком. Можно придумать еще множество таких оптимизаций, но всех их будет связывать тот факт, что для реализации требуется некий механизм способный вычислять функции на этапе компиляции. Поэтому очевидным требованием к прототипу может служить наличие удобного и гибкого API, позволяющего разработчикам компилятора (или даже пользователям) писать свои оптимизаторы.

4.2. Способы реализации прототипа

Сформулировал основные требования можно рассмотреть варианты реализации прототипа.

1. Текущая реализация вычислений на этапе компиляции в Kotlin.

Как уже было сказано, компилятор Kotlin поддерживает простейшие вычисления. Возникает вопрос можно ли переиспользовать или дописать текущую реализацию? Для того чтобы это понять

надо подробнее рассмотреть процесс компиляции. Этапы работы компилятора представлены на схеме ниже.

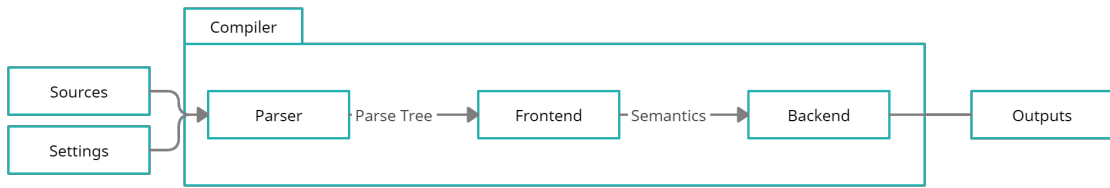


Рис. 3: Условная схема работы компилятора Kotlin

Рассмотрим каждый из них вкратце.

- **Parser.** В качестве входных параметров компилятор принимает набор аргументов и список файлов (исходный код). Задача парсера проанализировать эти файлы и сформировать синтаксическое дерево программы – Parse Tree.
- **Frontend.** На этом этапе происходит более глубокий анализ программы. Из дерева, построенного на предыдущем шаге, извлекается семантическая информация, например типы переменных или сигнатуры функций. Здесь же формируются диагностические сообщения об ошибках в программе.
- **Backend.** Последним шагом происходят последовательные оптимизации кода и в конечном итоге генерация кода для заданной платформы.

В текущей реализации, процесс вычисления функций происходит на этапе frontend. Это в первую очередь связано с тем, что этот этап компиляции не зависит от платформы, а во-вторых, здесь присутствует вся необходимая информация, которая может быть использована для вычисления встроенных функций.

К сожалению, переиспользовать данную реализацию не получится просто потому что информации на этапе frontend недостаточно для вычисления пользовательских функций. Самым важным

ограничением является отсутствие тел функций взятых, например, из стандартной библиотеки. Для анализа кода это не является проблемой, а вот вычислить такую функцию мы естественно не сможем.

2. Скриптинг

Язык Kotlin умеет работать со скриптами. Скрипт — это исходный файл с расширением “.kts” и исполняемым кодом верхнего уровня [13]. Потенциально скриптинг может быть использован для наших целей, т.к. используя скрипты можно вычислить любое выражение на языке Kotlin. Но скриптинг не удовлетворяет требованию о мультиплатформенности. В данный момент скрипты работают только для платформы JVM. Это связано с особенностью их исполнения. Скрипт выполняется в два этапа:

- **Компиляция.** Очевидно, что сначала нужно скомпилировать скрипт. Нет необходимости проходить весь процесс компиляции, достаточно этапа frontend. Результат этого этапа – семантическая информация о программе, используя которую можно сформировать объект типа “KClass”, описывающий скрипт.
- **Исполнение.** Объект типа “KClass” является прямым аналогом “Class” в Java. Используя эту информацию можно динамически создать такой объект и выполнить код прямо в процессе компиляции.

Именно использование “KClass” ограничивает применение скриптинга.

3. Интерпретатор языка

Остается еще один вариант – написать интерпретатор языка, который будет удовлетворять всем поставленным требованиям.

- **Требование №1** – мультиплатформенность.

Для того чтобы не пришлось реализовывать интерпретатор под каждую платформу, очевидно, что он должен интерпретировать код до этапа кодогенерации. В то же время, из пунктов описанных выше следует, что он должен работать после этапа frontend. Значит необходимо реализовать интерпретатор, который умеет работать с промежуточным представлением программы, переданное на backend.

- **Требование №3** – широкий репертуар.

Еще одним плюсом такого места для интерпретатора является тот факт, что на этапе backend есть возможность получить тела функций и методов, а это в свою очередь дает возможность реализовать интерпретацию практически любого Kotlin-кода.

Доступ к телам функций происходит через загрузку данных из klib-файла. Если описывать эту структуру вкратце, то это чем-то напоминает jar. Там хранится дерево компиляции, которое может быть загружено при необходимости. Такая структура сейчас является экспериментальной, но для целей прототипа она нам подходит.

- Оставшиеся требования (**№2** и **№4**) будут поддержаны прямо в реализации прототипа.

4.3. Архитектура прототипа

Если попытаться описать алгоритм работы некоего интерпретатора, то получится примерно следующий набор действий:

1. прочитать инструкцию;
2. проанализировать инструкцию;
3. выполнить действия, соответствующие этой инструкции;
4. перейти к шагу 1, если остались инструкции.

В этом плане интерпретатор языка Kotlin будет мало чем отличаться от любого другого интерпретатора.

На основе диаграммы, представленной на рисунке ниже, попробуем разобрать архитектуру полученного прототипа.

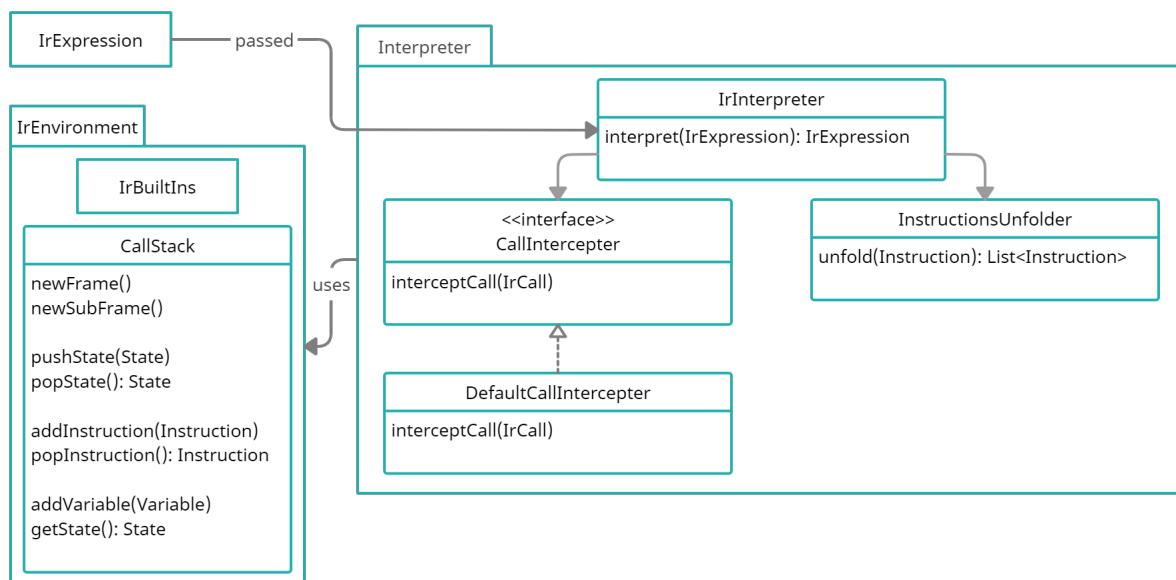


Рис. 4: Архитектура интерпретатора языка Kotlin

Рассмотрим каждую компоненту по отдельности.

1. IrInterpreter.

Этот класс является центром всего прототипа. В качестве публичного API предоставляется функция “interpret“, которая принимает на вход узел дерева, имеющий тип **IrExpression**. После интерпретации возвращается либо новый узел, описывающий некую константу, либо узел описывающий ошибку, возникшую на этапе интерпретации.

2. IrEnvironment

Для создания интерпретатора необходимо сперва сконструировать его окружение. Данный класс содержит:

- стек вызовов – **CallStack**;

- класс контейнер – **IrBuiltIns**, описывающий встроенные функции и классы языка;
- различные конфигурации, например, максимальный размер стека.

По большей части окружение это просто контейнер, который хранит информацию, используемую всеми объектами из блока “Interpreter”.

3. CallStack

Стек вызовов представляет собой набор фреймов, каждый из которых хранит информацию о части программы, за которую они ответственны. Новый фрейм создается каждый раз при вызове функции и значит каждый фрейм ответственен за хранение информации внутри этой функции.

Условно стек вызовов хранит три типа информации:

- набор инструкций, доступ к которым осуществляется через методы “addInstruction” и “popInstruction”;
- стек данных, доступ через методы “pushState” и “popState”;
- память, доступ через методы “addVariable” и “getState”.

На стеке данных и в памяти хранятся объекты типа “State” — интерфейс описывающий внутреннее представление данных. Подробное его описание будет дано в следующем разделе.

4. InstructionsUnfolder

Данный класс реализует шаг №2 в алгоритме работы интерпретатора, т.е. здесь происходит анализ новой инструкции и формируется набор действий, который необходимо выполнить интерпретатору.

Основное преимущество “разворачивания” инструкций заключается в том, что это исключает ошибки вида “StackOverflowError”

в интерпретаторе из-за глубокой вложенности пользовательского кода.

5. CallInterceptor

Сам по себе интерпретатор никак не обрабатывает вызовы функций. Для этой задачи применяется “CallInterceptor”. Реализацию данного интерфейса можно описать как черный ящик, который принимает описание вызова и возвращает результат на стеке.

Основная необходимость такого интерфейса, возникает в момент когда речь идет о возможности расширения интерпретатора. Например, если какой-то разработчик компилятора захочет написать свой оптимизатор с использованием новой псевдо-функции, то для того чтобы интерпретатор смог ее распознать, ему достаточно определить свою реализацию для “CallInterceptor”, а не искать место в коде, где обрабатываются псевдо-функции и исправлять этот кусок.

На диаграмме видно, что у этого интерфейса есть одна реализация – “DefaultCallInterceptor”. Она представляет собой обработку вызовов по умолчанию, т.е. те функции которые имеют тела, будут проинтерпретированы, а остальные будут обработаны согласно неким заданным в коде правилам. Подробнее это будет описано в следующем разделе.

4.4. Особенности реализации

Рассмотрим подробнее некоторые особенности работы интерпретатора.

4.4.1. Интерфейс State

Данный интерфейс применяется для описания объектов внутри интерпретатора. Для его реализации необходимо определить два поля:

- поле **irClass** содержит информацию о классе, который описывает данный “State“, в терминах промежуточного представления (intermediate representation, - IR).
- поле **fields** содержит значения полей соответствующего объекта.

На рисунке ниже представлены все реализации интерфейса “State“.

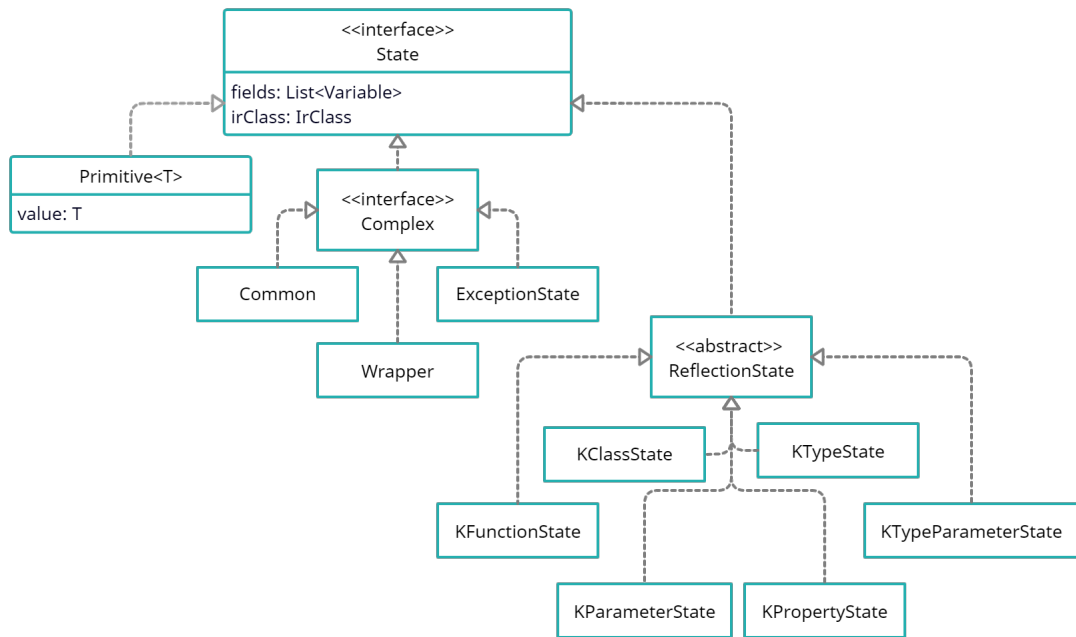


Рис. 5: Реализации интерфейса State

Рассмотрим каждый из них вкратце.

- **Primitive.** Описывает значения примитивного типа, строковый литерал, а также массивы. Содержит дополнительное поле “value“, которое хранит в себе это значение в виде объекта Kotlin. Мы можем так делать, потому что интерпретатор написан на языке Kotlin, а значит такие примитивы мы можем хранить как есть.
- **Complex.** Описывает объекты, которые не получается сохранить в том виде, как это было бы при исполнении кода. Вместо этого их надо описывать как набор полей (**fields**) и описание соответствующего класса (**irClass**). Имеет три реализации:

- **Common.** Описывает объект, который можно создать путем интерпретации кода.
- **Wrapper.** Описывает объект, полученный путем выполнения платформенно-зависимого кода. Текущая реализация интерпретатора умеет работать только с кодом на языке Kotlin, но есть ряд функций и классов, реализация которых зависит от конкретной платформы и которые могли бы быть полезны. Например, “ArrayList” или “HashMap”. Такие объекты создаются путем исполнения кода, а не интерпретации. Подробнее об этом механизме рассказано в следующем разделе.
- **ExceptionState.** Описывает исключение, возникшее в результате интерпретации пользовательского кода.
- **ReflectionState.** Данный класс (и все его реализации) описывают объекты, необходимые для применения рефлексии.

4.4.2. Особенности работы с платформенно-зависимым кодом.

Для разработки мультиплатформенных приложений Kotlin предоставляет механизм “expect” и “actual” деклараций. “Expect” декларации применяются в модуле с общим кодом, “actual” декларации соответственно описываются в платформенном модуле и реализуют необходимое поведение в терминах конкретной платформы.

Стандартная библиотека языка также применяет данный механизм. Можно предположить, что таких деклараций немного и будут они использоваться не часто, но это не так. Например, при вызове функции “listOf” происходит создание списка и возвращение некоего объекта реализующего “List”, но сама реализация этого интерфейса является платформенно-зависимой. В интерпретаторе обязательно должна быть возможность обработки такого кода.

Компилятор языка использует для своей работы платформу JVM, значит в интерпретаторе можно воспользоваться неким API из Java для выполнения отдельных фрагментов кода динамически. Например, можно воспользоваться “Reflection API”. Это очень мощный механизм,

позволяющий динамически создавать объекты, получать к ним доступ и даже изменять их по ходу выполнения, но все это работает очень медленно. К счастью, нам не нужны все возможности такого API, а значит можно воспользоваться более легковесной его версией – “MethodHandles API”.

“MethodHandles” — это низкоуровневый механизм для поиска, адаптации и вызова методов. Таким API сложнее пользоваться (так как оно низкоуровневое), но для наших целей главное это производительность.

В результате выполнения метода через “MethodHandles API” возвращается некий Java объект. Для применения такого результата в интерпретаторе необходимо преобразовать его в “State”. Именно для этих целей и применяется класс “Wrapper”. Он содержит в себе Java объект и по запросу из интерпретатора предоставляет информацию об этом объекте.

4.4.3. Проксирование функций

Попробуем на конкретном примере рассмотреть одну особенность использования “MethodHandles API” в интерпретаторе. Допустим пользователь написал следующий код:

```
1 class MyCharSequence(val str: String): CharSequence {  
2     override val length: Int = str.length  
3     override fun get(index: Int) = str[index]  
4     override fun subSequence(startIndex: Int, endIndex: Int) = str.subSequence  
      ↪ (startIndex, endIndex)  
5 }  
6  
7 const val sbSize = StringBuilder(MyCharSequence('MyString')).length
```

Здесь определяется новый класс “MyCharSequence”, реализующий интерфейс “CharSequence”. После этого объект такого класса передается в “StringBuilder” и запрашивается длина полученной строки.

С точки зрения прототипа, такой код корректен, его можно проинтерпретировать и записать результат. Однако класс “StringBuilder” является expect/actual декларацией, а значит все действия с этим классом

должны быть выполнены, а не проинтерпретированы.

Для того чтобы создать такой объект, нужно воспользоваться “MethodHandles API” и передать аргумент в виде Java объекта. Но создать полноценный объект “MyCharSequence” в интерпретаторе мы не можем, как минимум потому что мы еще не закончили компиляцию. Для решения этой проблемы поможет “java.lang.reflect.Proxy”. Этот класс позволяет динамически создать объект реализующий заданный интерфейс. Все вызовы к такому объекту будут обработаны по заданным в коде правилам, что позволяет интерпретировать логику методов в момент выполнения кода через “MethodHandles API”.

4.5. Основные ограничения и правила

1. Для упрощения реализации, было принято решение ввести в язык Kotlin аннотацию *CompileTimeCalculation*, что является прямым аналогом модификатора “constexpr” в C++. Ее основное применение – указать интерпретатору что может быть проинтерпретировано, а что нет. Такой аннотацией можно пометить функцию, метод, конструктор, класс или интерфейс.
2. Если функция помечена этой аннотацией, это означает, что данная функция может быть выполнена на этапе компиляции (если в момент вызова ее аргументы тоже могут быть выполнены).
3. Внутри тел compile-time функций и конструкторов разрешается использование любых конструкций языка Kotlin таких, как циклы, ветвления, создание объектов и даже возбуждение исключений. Все вызовы других функций должны быть выполнимы в compile-time.
4. Если класс или интерфейс целиком помечены аннотацией *CompileTimeCalculation*, то все поля, методы, конструкторы, внутренние и вложенные классы считаются выполнимыми на этапе компиляции. Это правило не распространится только на *object* и *companion object*.

5. Для того чтобы, метод объявленный внутри *object* или *companion object*, мог быть выполнен он должен быть помечен отдельно. Это сделано по причине того, что *object* — это синглтон в Kotlin, поэтому на этапе компиляции мы не имеем права как-либо инициализировать его. Однако если метод не зависит от внутреннего не статического состояния, то его можно выполнить.
6. Разрешается наследование и переопределение *compile-time*-методов. Здесь фигурирует важное правило: если класс объявлен как *compile-time*, то при переопределении другого *compile-time*-метода он обязан быть помечен аннотацией *CompileTimeCalculation*. Такой подход позволяет вызывать *compile-time*-методы, не опасаясь за то, что при переопределении пользователь намеренно или случайно уберёт аннотацию и будет в теле метода использовать невыполнимый код.
7. Тип возвращаемого значения из *compile-time*-функции никак не ограничивается, но сохраняется только значение примитивных типов и *String*. Это означает, что если итоговым результатом будет какой-либо объект, то результат просто отбрасывается. Чтобы лучше понять это правило рассмотрим пример:

```
1 const val constStringBuilder = StringBuilder('Hello')    // Error
2 val runtimeStringBuilder = StringBuilder('Hello')      // No error, did
   compile time evaluation, but skipped the result
```

В данном случае такое решение сделано для простоты дизайна. Хочется иметь возможность передавать результаты в виде объектов между разными *compile time*-функциями, но в то же время сохранить такой результат пока возможности нет.

4.6. Архитектура оптимизатора языка

Покажем теперь как на основе интерпретатора языка можно писать оптимизатор, способный ускорить выполнение программы за счет

анализа compile-time информации и более сложной обработки inline-функций.

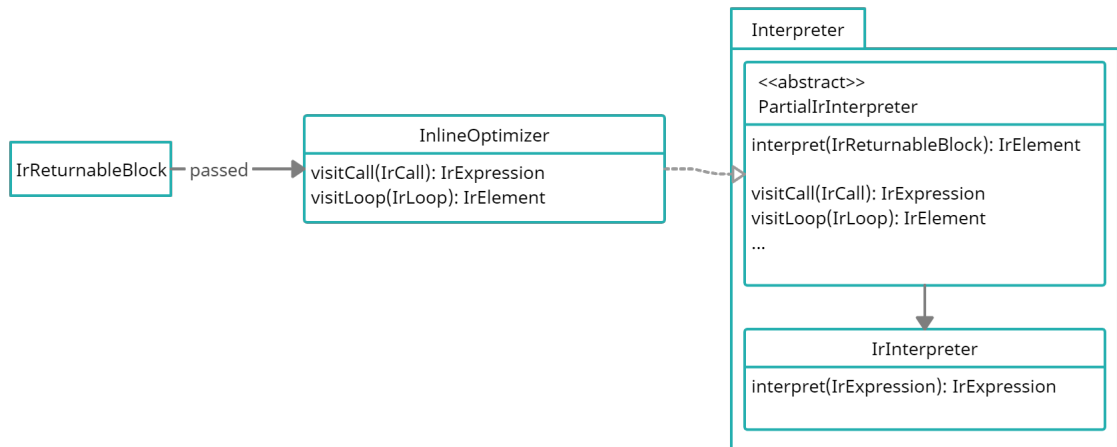


Рис. 6: Архитектура inline-оптимизатора языка Kotlin

Рассматриваемый оптимизатор выполняет следующий набор действий.

1. Сперва происходит встраивание функций в место вызова, т.е. будут обработаны все функции помеченные модификатором `inline`. После этого место где было вызвана такая функция будет описываться узлом дерева – `IrReturnableBlock`.
2. Далее этот узел передается как входной параметр оптимизатору. Необходимо шаг за шагом проинтерпретировать выражения в данном узле и собрать compile-time информацию. Такая логика будет являться общей для любого оптимизатора, поэтому для таких целей применяется еще один слой абстракции – `PartialIrInterpreter`. Данный класс реализует паттерн `transformer` и позволяет проходить по узлам IR дерева, параллельно изменяя их.
3. Для реализации конкретного оптимизатора необходимо унаследовать класс `PartialIrInterpreter` и переопределить необходимые методы. Для случая inline-оптимизаций нас интересуют метод об-

хода вызовов (“visitCall”), применяемый для устранения reflection-вызовов, и метод обхода циклов (“visitLoop”), применяемый для разворачивания этих циклов в случае когда это необходимо.

5. Апробация

В этом разделе приводится демонстрация полученных результатов. Сперва приводятся несколько примеров применения интерпретатора и в целом compile-time вычислений. Далее будет показано, что все вычисления являются корректными, а также что использование inline-оптимизатора действительно ускоряет работу программы.

5.1. Демонстрация результатов

Для целей тестирования был собран полноценный компилятор и плагин для среды IntelliJ IDEA. Это позволяет прямо в среде разработки воспользоваться созданным интерпретатором и увидеть результат выполнения compile-time-функции в окне сообщений.

5.1.1. Вычисление корректных функций

1. Простейшим примером является вычисление суммы двух чисел.

```
1 @CompileTimeCalculation
2 fun sum(a: Int, b: Int) = a + b
3
4 const val getSum = sum(1, 2)
```

Результаты вычислений можно увидеть в окне сообщений в среде разработки IntelliJ IDEA:

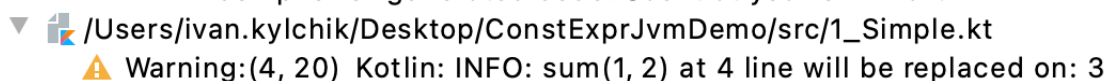


Рис. 7: Окно сообщений с результатом вычисления функции “sum”

2. Из более сложных примеров можно привести вычисление регулярных выражений:

```
1 const val regexReplace1 = Regex('0').replace('There_are_0_apples', 'n')
2 const val regexReplace2 = Regex('red|green|blue').replace('Roses_are_red
   ↪ ,_Violets_are_blue') { it.value + '!' }
```

```

3 const val regexReplace3 = Regex('red|green|blue').replaceFirst('Roses_
    ↪ are_red,_Violets_are_blue', 'REPLACED')
4 const val regexReplace2 = Regex('\\W+').split('Roses_are_red,_Violets_
    ↪ are_blue').size

```

Результатом выполнения является:

```

▼ /Users/ivan.kylchik/Desktop/ConstExprJvmDemo/src/15_Regex.kt
▲ Warning:(2, 10) Kotlin: INFO: Regex("0")
    .replace("There are 0 apples", "n") at 2 line will be replaced on: There are n apples
▲ Warning:(4, 10) Kotlin: INFO: Regex("(red|green|blue)")
    .replace("Roses are red, Violets are blue") { it.value + "!" } at 4 line will be replaced on: Roses are red!, Violets are blue!
▲ Warning:(6, 10) Kotlin: INFO: Regex("(red|green|blue)")
    .replaceFirst("Roses are red, Violets are blue", "REPLACED") at 6 line will be replaced on: Roses are REPLACED, Violets are blue
▲ Warning:(8, 51) Kotlin: INFO: Regex("\\W+")
    .split("Roses are red, Violets are blue").size at 8 line will be replaced on: 6

```

Рис. 8: Результат интерпретации регулярных выражений

3. Была реализована описанная ранее псевдо-функция “sourceLocation”.

Особенность использования псевдо-функций заключается в том, что в момент компиляции все вызовы таких функции будут заменены на результат ее вычисления. Для случая с функцией “sourceLocation”, вызов заменяется строкой вида “FileName:CallLine”.

Для того чтобы эту функцию было удобно использовать была написана отдельная вспомогательная функция “log”, которая выводит в консоль переданное сообщение вместе с местом ее вызова.

```

1 import kotlin.experimental.sourceLocation
2
3 @CompileTimeCalculation
4 inline fun log(
5     info: String, fileNameAndLine: String = sourceLocation()
6 ): String {
7     return info + 'at' + fileNameAndLine
8 }

```

Для тестирования возможностей функции “sourceLocation” был написан следующий код, расположенный в файле “16_SourceLocation.kt”:

```

1 import kotlin.experimental.sourceLocation
2
3 const val thisFileInfo = sourceLocation()
4
5 fun main() {
6     println(log('Some_data'))
7     println(sourceLocation())
8     println(thisFileInfo)
9 }

```

Результатом выполнения программы является:

```

Some data at 16_SourceLocation.kt:6
16_SourceLocation.kt:7
16_SourceLocation.kt:3

Process finished with exit code 0

```

Рис. 9: Результат выполнения кода содержащего функцию “sourceLocation”

- Следующий пример демонстрирует применение reflection-методов на уровне компиляции. Для демонстрации базовых возможностей был написан следующий код:

```

1 @CompileTimeCalculation
2 class ToReflect(val somePropertyWithLongName: Int)
3
4 const val propName = ToReflect::somePropertyWithLongName.name
5 const val className = ToReflect::class.qualifiedName!!

```

В результате компиляции мы получаем следующее сообщение:

```

▼ /Users/ivan.kylchik/Desktop/ConstExprJvmDemo/src/17_Reflection.kt
⚠ Warning:(4, 58) Kotlin: INFO: ToReflect::somePropertyWithLongName.name at 4 line will be replaced on: somePropertyWithLongName
⚠ Warning:(5, 53) Kotlin: INFO: ToReflect::class.qualifiedName!! at 5 line will be replaced on: ToReflect

```

Рис. 10: Результат компиляции файла с кодом, содержащий compile-time reflection методы

5.1.2. Обработка ошибок

1. Для плагина также реализовано отображение ошибок, если какие-то вычисления невозможны на этапе компиляции.

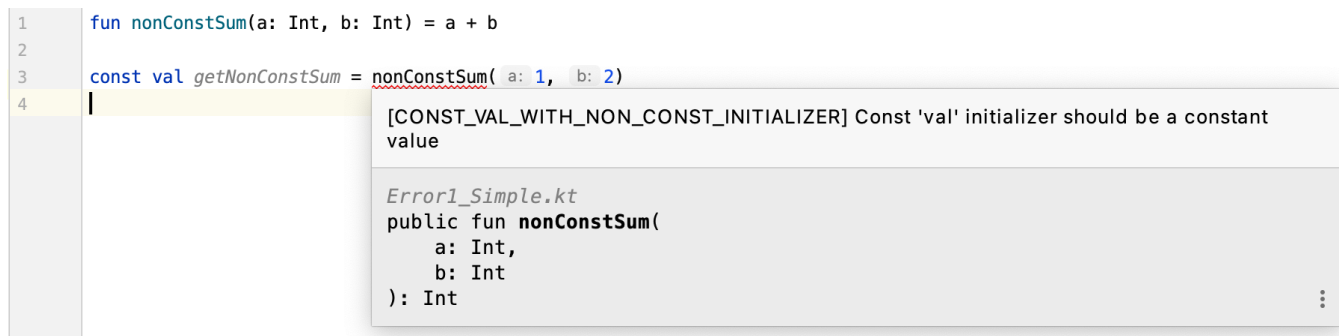


Рис. 11: Пример ошибки компиляции, обозначающей невозможность присвоения не константного значения в переменную с модификатором const



Рис. 12: Пример ошибки компиляции при использовании вызова не константной функции внутри другой функции, помеченной как compile time



Рис. 13: Пример ошибки компиляции, указывающей на то, что в классе, первичный конструктор которого является compile-time, есть метод из compile-time-интерфейса, который не помечен аннотацией CompileTimeCalculation

- Ниже приведен пример возбуждения исключения в момент интерпретации.

```
1 const val divideByZero = 1 / 0
```

Интерпретатор отлавливает ошибку и предупреждает об этом:

```

▼ /Users/ivan.kylchik/Desktop/ConstExprJvmDemo/src/9_Exceptions.kt
  ⚠ Warning:(1, 26) Kotlin: Division by zero
  ⚠ Warning:(1, 26) Kotlin: INFO: 1 / 0 at 1 line will be replaced on:
    Exception java.lang.ArithmeticException: / by zero
    at 9_ExceptionsKt.<clinit>(9_Exceptions.kt:1)
  
```

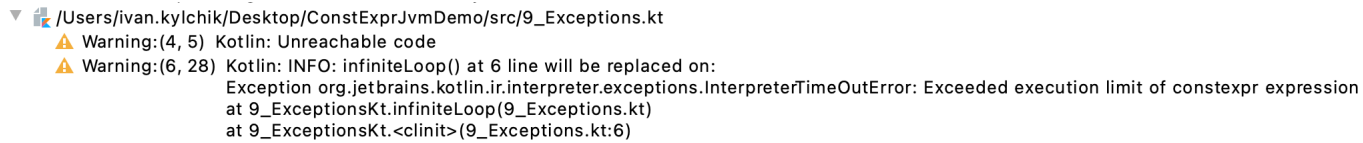
Рис. 14: Предупреждение компилятора о делении на 0

- Интерпретатор умеет корректно обрабатывать случаи когда пользователь напишет код который никогда не завершится, например бесконечный цикл.

```

1 @CompileTimeCalculation
2 fun infiniteLoop(): Int {
3     while(true) {}
4     return 0
5 }
6 const val executionLimit = infiniteLoop()
  
```

Интерпретатор останавливает вычисление и предупреждает, что для данной функции был превышен лимит команд:



The screenshot shows two warning messages in an IDE. The first warning is at line 4, column 5, with the message 'Kotlin: Unreachable code'. The second warning is at line 6, column 28, with the message 'Kotlin: INFO: infiniteLoop() at 6 line will be replaced on: Exception org.jetbrains.kotlin.ir.interpreter.exceptions.InterpreterTimeOutError: Exceeded execution limit of constexpr expression at 9_ExceptionsKt.infiniteLoop(9_Exceptions.kt) at 9_ExceptionsKt.<clinit>(9_Exceptions.kt:6)'. The file path is '/Users/ivan.kylchik/Desktop/ConstExprJvmDemo/src/9_Exceptions.kt'.

```
▼ /Users/ivan.kylchik/Desktop/ConstExprJvmDemo/src/9_Exceptions.kt
⚠ Warning:(4, 5) Kotlin: Unreachable code
⚠ Warning:(6, 28) Kotlin: INFO: infiniteLoop() at 6 line will be replaced on:
Exception org.jetbrains.kotlin.ir.interpreter.exceptions.InterpreterTimeOutError: Exceeded execution limit of constexpr expression
at 9_ExceptionsKt.infiniteLoop(9_Exceptions.kt)
at 9_ExceptionsKt.<clinit>(9_Exceptions.kt:6)
```

Рис. 15: Предупреждение о превышении лимита количества операций

5.2. Проверка корректности

Для проверки корректной работы интерпретатора использовались уже существующие тесты в проекте Kotlin. Для наших целей отлично подходят black box тесты. Тестовым файлом для таких тестов является программа на языке Kotlin. Тестирующая система собирает компилятор, с помощью которого происходит компиляция, а затем непосредственно запуск программы. По возвращаемому результату происходит проверка корректности работы компилятора.

Была создана отдельная тестирующая система, которая вместо запуска программы начинает ее интерпретацию. Также была написана система проверки файлов, которая автоматически отсеивает тесты содержащие заранее не интерпретируемый код, например, тесты содержащие код на Java.

Результат тестирования приведен ниже.

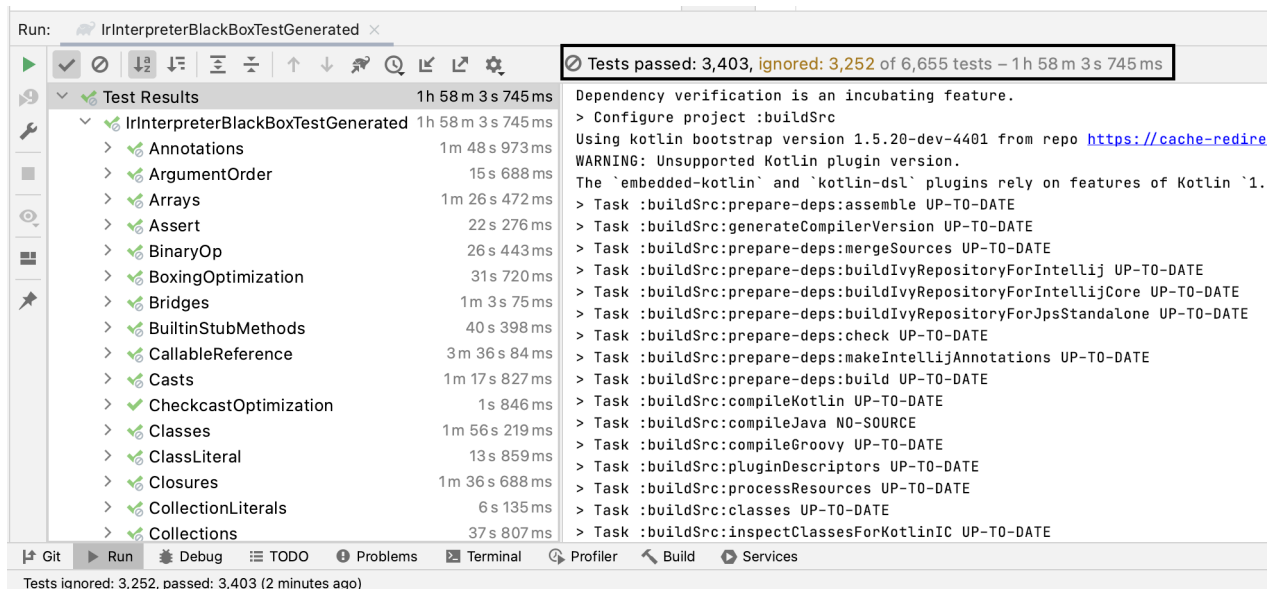


Рис. 16: Результаты тестирования интерпретатора на корректность

Как видно почти половина тестов была проигнорирована т.к. интерпретатор не умеет работать с некоторыми выражениями. Тем не менее, оставшаяся половина тестов прошла успешно.

5.3. Тестирование производительности

Демонстрация полученного выигрыша производительности при использовании inline-оптимизатора будет производиться на примере с методом “compareBy”. Для этих целей будет применяться абсолютно тот же тест, что был показан в разделе 3.1. Единственная разница в том что ранее количество полей в тестируемых классах было равно трем, теперь это переменное значение.

Тестируются три разновидности компаратора: написанный вручную; созданный методом “compareBy”, но не оптимизированный; созданный методом “compareBy” и оптимизированный. Результат можно увидеть на графике.

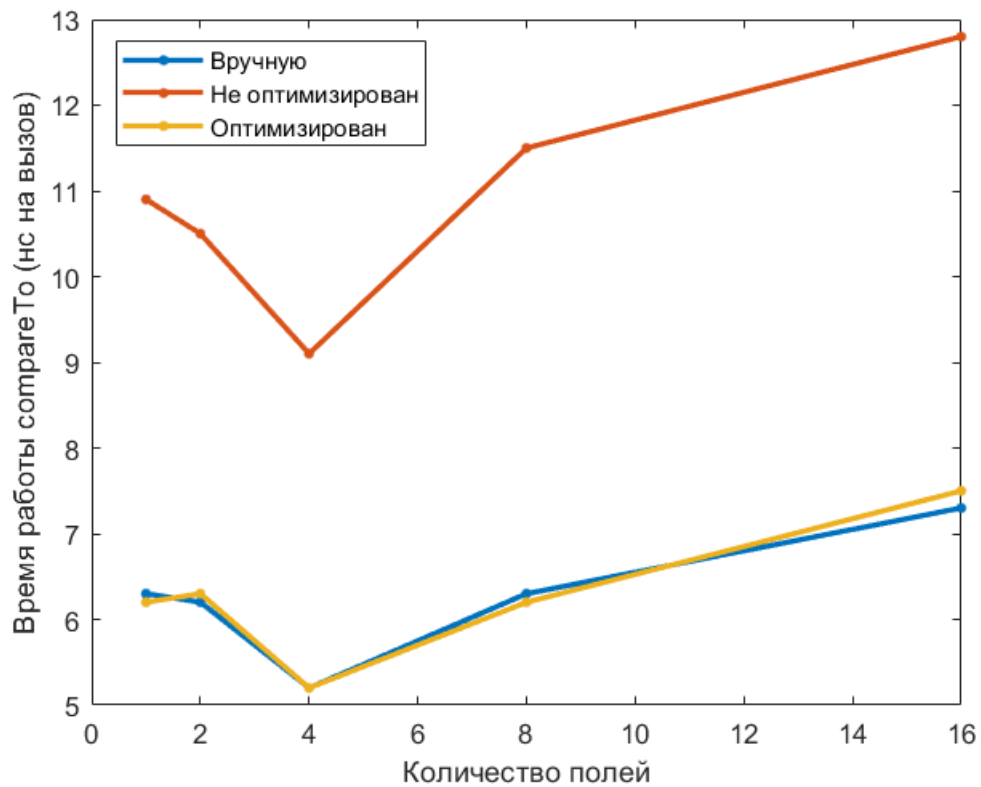


Рис. 17: Результаты тестирования времени работы метода “compareTo”

Отлично видно, что inline-оптимизация ускорила работу компаратора созданного методом “compareBy” настолько, что время работы теперь сравнимо с временем работы метода “compareTo” написанного вручную.

6. Заключение

В рамках данной работы были получены следующие результаты.

- Был произведен анализ существующих способов вычисления функций на этапе компиляции. На основании этого анализа было предложено ввести в язык аннотацию, которое будет явно указывать на функцию, для которой возможно вычисление на этапе компиляции.
- Была показана необходимость вычисления функций на этапе компиляции в языке Kotlin. На основе приведенных примеров был создан список требований, которым должен удовлетворять разрабатываемый прототип.
- Были исследованы различные способы выполнения функций на этапе компиляции. Для реализации поставленных целей и удовлетворению всем требованиям, было принято решения написать интерпретатор языка Kotlin.
- Был реализован прототип в виде интерпретатора [3], продемонстрирована корректная работа интерпретатора с использованием плагина для среды IntelliJ IDEA. Также было показано, что использование интерпретатора действительно ускоряет работу программ.

Список литературы

- [1] Ахо Альфред, Сети Р, Ульман Дж. Компиляторы // Принципы, технологии, инструменты. М.: Вильямс. — 2003.
- [2] Евгений Клименков, Constexpr — большое благо, выраженное в неправильной идее. — URL: <http://0x1.tv/20191205AG> (дата обращения: 15.01.2020).
- [3] Исходный код разработанного интерпретатора языка Kotlin. — URL: <https://github.com/JetBrains/kotlin/tree/master/compiler/ir/ir.interpreter> (дата обращения: 24.05.2021).
- [4] AWS Lambda, бессерверные вычисления. — URL: <https://aws.amazon.com/ru/lambda/?c=ser&sec=srv> (дата обращения: 07.05.2021).
- [5] C++ named requirements, LiteralType. — URL: https://en.cppreference.com/w/cpp/named_req/LiteralType (online; accessed: 15.01.2020).
- [6] D lang, Compile time evaluation. — URL: https://wiki.dlang.org/User:Quickfur/Compile-time_vs._compile-time (online; accessed: 15.01.2020).
- [7] Introduce constexpr modifier, prerequisites for creating a constexpr modifier in Kotlin. — URL: <https://youtrack.jetbrains.com/issue/KT-14652> (online; accessed: 07.05.2021).
- [8] Jemerov Dmitry, Isakova Svetlana. Kotlin in action. — Manning Publications Company, 2017.
- [9] Kotlin const misunderstanding, an example of a misunderstanding of how the const modifier works in the Kotlin language. — URL: <https://stackoverflow.com/questions/44921746/what-can-be-expressed-in-a-compile-time-constant-const-val> (online; accessed: 07.05.2021).

- [10] Kotlin const modifier specification. — URL: <https://kotlinlang.org/spec/declarations.html#constant-properties> (online; accessed: 07.05.2021).
- [11] Olsson Mikael. Handbook of C++ Syntax: A Reference to the C++ Programming Language. — Siforia, 2011.
- [12] Properties in Kotlin, short description. — URL: <https://kotlinlang.org/docs/properties.html> (online; accessed: 17.04.2021).
- [13] Scripting in Kotlin, short description. — URL: <https://kotlinlang.org/docs/command-line.html#run-scripts> (online; accessed: 17.04.2021).
- [14] Vandevorode Daveed. Reflective metaprogramming in C++ // N1471. — 2003. — P. 03–0054.