Санкт-Петербургский государственный университет

Системное программирование

Группа 19Б.10-мм

***Порсев Егор Валерьевич***

# Реализация и внедрение системы тестирования в экспериментальный язык программирования

Отчёт по учебной практике

Научный руководитель:
старший преподаватель каф. Инф., к.ф.-м.н. С.И. Салищев

Санкт-Петербург
2022

# Оглавление

# 1. Введение

Этап тестирования считается важной частью цикла разработки продукта, так как чем дольше ошибка в программе остается незамеченной, тем больше возможные потери в случае программного сбоя [10]. При разработке программных продуктов разработчики пишут код на различных языках программирования. Для того, чтобы программа приобрела вид, который понимает компьютер, почти всегда используются компиляторы — инструменты, которые переводят код из языка, который понимает человек, в машинный код. Сбои в работе самого компилятора могут привести к ошибочной программе, несмотря на то, что написанный разработчиками код мог быть корректен. Поэтому компилятор должен проходить особенно строгие этапы тестирования.

Современные языки программирования проектируются с учетом необходимости тестирования программ, стараясь оптимизировать этот процесс. Все самые популярные языки поддерживают возможность тестирования программ, написанных на этих языках [1, 7, 8, 11]. В этой работе рассматривается экспериментальный язык программирования, который находится на этапе разработки. Для того, чтобы компилятор экспериментального языка и пользовательские программы, написанные на этом языке, можно было тестировать, необходимо, чтобы язык имел поддержку тестирования пользовательского кода. При отсутствии этой возможности компилятор экспериментального языка невозможно будет эффективно проверять на корректность, что будет вызывать ошибки и недоверие пользователей, а пользовательские программы невозможно будет эффективно тестировать, что сделает язык неконкурентоспособным.

Целью этой работы является реализация и внедрение системы тестирования в экспериментальный язык. Для этого рассматриваются требования к тестирующей системе в экспериментальном языке. На основе требований выбирается набор ортогональных критериев, по которым сравниваются 6 систем тестирования в других языках программирования.

Результатом сравнения систем тестирования по набору критериев является предложение для тестирующей системы в экспериментальном языке, которое включает синтаксис и описание реализации. Полученное предложение удовлетворяет всему набору критериев и является комбинацией подходов других языков программирования.

В рамках данной работы была реализована система, позволяющая писать тесты на экспериментальном языке программирования. Архитектура тестовой системы позволяет проверять корректность всех требуемых библиотек, даже саму тестовую систему, что математически обосновано (3.8).

В ходе внедрения реализованной системы стандартные и внутренние библиотеки экспериментального языка были покрыты тестами, что проверяет на корректность не только библиотеки, но и компилятор.

## 1.1. Цель и задачи

Целью является разработка системы, позволяющей тестировать на правильность программы, написанные на экспериментальном языке программирования и отдельные компоненты этих программ, и дальнейшее внедрение этой системы для тестирования компилятора, стандартной библиотеки языка и пользовательских библиотек.

Для выполнения данной цели были поставлены следующие задачи:

1. Выбрать набор ортогональных критериев для сравнения подходов к тестированию;

2. Изучить существующие подходы к тестированию и их реализации в других языках программирования и библиотеках по выбранным критериям;

3. Изучить особенности экспериментального языка, оказывающие влияние на выбор тестирующего подхода;

4. Определить подход к тестированию программ для экспериментального языка, который удовлетворяет приведенному набору критериев;

5. Спроектировать и реализовать систему, позволяющую писать и запускать тесты на экспериментальном языке программирования;

6. Внедрить и апробировать тестирующую систему в компилятор и стандартную библиотеку;

# 2. Обзор

## 2.1. Ключевые особенности экспериментального языка

Экспериментальный язык является компилируемым статически типизированным языком общего применения. Он поддерживает классы, интерфейсы, структуры, а также функции высшего порядка. Типовая система одновременно учитывает именную и структурную типизацию, алгебраические типы и перечисления.

Система модулей опирается на понятие пакетов — набора файлов, схожих по содержанию. Символы , определенные в пределах одного пакета, по-умолчанию видны коду в любом файле в этом же пакете. Для того, чтобы символ был виден за пределами пакета предусмотрен знак экспортирования.

Единицей компиляции может быть либо пакет, либо входной файл, содержащий входную точку пользовательского приложения. Единицы компиляции могут импортировать другие пакеты и использовать символы определенные в этих пакетах. В экспериментальном языке запрещены циклы в графе зависимостей.

Идеология экспериментального языка, ключевая для целей этой работы, состоит из следующих позиций:

- Экспериментальный язык старается позволять пользователю кратко излагать свою мысль;
- В экспериментальном языке не должно существовать несколько языковых конструкций, выражающих одну идею;
- Экспериментальный язык не должен иметь большое количество сложных правил;

На момент проектирования тестовой системы в языке отсутствовала поддержка рефлексии и восстановления после исключений.

## 2.2. Архитектура компилятора экспериментального языка

Компилятор экспериментального языка сначала разбирает исходный код, получая промежуточное представление в виде дерева (далее – *AST*). После этого он выполняет более 5 проходов по этому дереву, дополняя его и в конце генерируя целевой код. Во время разбора исходного кода или проходов по *AST* компилятор может обнаружить допущенные пользователем ошибки, для чего предусмотрена система вывода ошибок.

Рассмотрим проходы компилятора, релевантные для данной работы:

**Load**. На этапе загрузки компилятор получает из исходного кода начальное представление в виде *AST*. Этот проход определяется рекурсивно:

- сначала компилятор разбирает исходный код данной единицы компиляции;
- в *AST* единицы компиляции находятся все ссылки на другие пакеты;
- выполняется поиск каждого такого пакета в файловой системе, и, если он еще не загружен он загружается рекурсивно.

При загрузке пакетов каждый файл с кодом разбирается отдельно, после чего происходит объединение этих файлов в единый *AST*-узел. Например, при слиянии собирается общий список деклараций и импортированных пакетов.

**Нормализация**. Далее следуют несколько проходов: *lookup*, *promote*. На проходе *lookup* разрешаются имена, проводятся все ребра в графе промежуточного представления между применениями имен и их декларациями. Проход *promote* производит проверку типов.

**Система вывода ошибок**. Объекты типа `Error` содержат тип, сообщение ошибки, название файла, где произошла ошибка, номер строки в этом файле и номер столбца. В качестве списка ошибок

выступает объект типа `ErrorAdapter`, он имеет методы `AddError` и `PrintAllErrors`, позволяющие добавлять ошибки в список и выводить их на экран.

**Расширение файлов**. Все файлы, содержащие код на экспериментальном языке, должны иметь фиксированное расширение, для примера `.code`.

## 2.3. Виды тестирования

Для классификации тестирования программного обеспечения можно представить следующие категории:

- внутренние и внешние тесты;
- тотальные и выборочные тесты;
- один или разные языки для написания кода и тестов.

Внутренние тесты позволяют компонентам программы проверять целостность своих инвариантов при любом взаимодействии с этими компонентами. Внешние тесты проверяют правильное поведение компонентов программ по отношению к другим компонентам, которые могут с ними взаимодействовать. Таким образом, внутренние и внешние тесты дополняю друг друга, поэтому для полноты тестовой системы необходимо рассмотреть оба вида тестов.

Тотальные тесты доказывают корректность алгоритмов для всех наборов данных, а выборочные рассматривают только подмножество всех возможных данных для проверки корректности. Частным случаем тотального тестирования является статическая проверка типов. Выборочное тестирование является наиболее простым с точки зрения реализации, поэтому программы часто сначала подвергают выборочному тестированию перед тем, как применять другие виды тестирования. Поскольку тотальное тестирование во многих практических случаях приводит к алгоритмически неразрешимым задачам, выборочное тестирование не

может быть заменено тотальным и должно присутствовать в экспериментальном языке. Далее эта работа рассматривает только выборочное тестирование, а тотальное выходящее за рамки проверки типов будет рассмотрено в следующей работе.

У подхода, при котором язык тестирования и язык программы совпадают, есть ряд преимуществ над другим подходом:

1. можно более эффективно тестировать локальные символы;
2. быстрее и проще процесс тестирования, потому что тестирование не требует установки и настройки другого языка и подготовки программы к тестированию на другом языке;
3. при совпадении языков можно использовать при тестировании все функции и библиотеки из программы;
4. при освоении языка легче учиться одному языку, чем двум;
5. при использовании экспериментального языка для тестирования будет найдено больше ошибок в реализации самого языка. Более того все популярные языки реализуют способ тестировать программы, поэтому наличие такой функциональности необходимо, чтобы экспериментальный язык был успешным.

Подход с использованием двух языков также имеет преимущества, если для тестирования используется язык более выразительный язык. Например использования для тотального тестирования кода на языке `C` языков `Haskell, Coq, Isabelle` или авто-генерация (мета-программирование) тестов для компилируемых языков на языке `Python`. Но для написания выборочных тестов языков достаточно высокого уровня эти преимущества не являются критическими. Одной из целей данной работы является исследование возможности авто-генерации тестов на том же языке.

По этим причинам в дальнейшей работе будут рассмотрены только подходы, где язык тестирования совпадает с языком, на котором написаны программы.

## 2.4. Критерии сравнения подходов

Для сравнения других подходов к тестированию используется набор ортогональных критериев. Выбор каждого критерия будет далее обоснован в этом разделе. Далее на основе этих критериев и информации об экспериментальном языке для нашей системы тестирования будет сформулирован набор требований.

**Критерии внешнего тестирования**

(1) Возможность программно во время исполнения программы создавать тесты (авто-генерация тестов), которые учитываются тестовой системой: результаты запуска этих тестов выводятся в диагностической информации, и эти тесты имеют тот же самый API для взаимодействия с тестовой системой;

(2) Поддержка подготовки и разрушения окружений для тестов;

(3) Система находит более одного пользовательского теста на этапе компиляции;

(4) Не используется рефлексия;

(5) Использование тестов как примеров в документации или, наоборот, примеров из документации как тестов;

(6) Возможность тестирования не экспортированных символов. Например, приватных функций;

(7) Возможность написания вспомогательных функций, видных только тестирующему коду;

Критерии являются ортогональными, что будет частично доказано далее при рассмотрении существующих подходов.

**(1) Программное создание тестов.** *Предполагается, что тестовые системы, удовлетворяющие этому критерию, являются расширяемыми.* Во многих случаях пользователи расширяют функциональности языка своими библиотеками, и тестовая система не является

исключением. Всегда можно написать любую надстройку над тестовой системой, используя тестовую систему только как запускающий механизм. Но при расширении системы таким образом тесты в расширенной системе не будут использовать API тестирующей системы и информация о запусках тестов не будет выведена тестовой системой автоматически. Такие надстройки мы не будем считать за расширения.

**(2) Окружение для тестов.** Нередко какие-то наборы тестов требуют общее окружение. В таком случае, если каждый тест будет сам создавать и разрушать окружение, то возникнет проблема копирования кода. Этот критерий требует возможность вынесения логики, связанной с тестовым окружением, за пределы каждого теста.

**(3) Поиск тестов на этапе компиляции.** Поиск тестов на этапе компиляции ускоряет время исполнения программы. Если же система умеет находить только один пользовательский тест, то можно считать этот тест точкой входа, что не представляет интереса. Поэтому мы исключаем этот случай.

**(4) Без рефлексии.** Рефлексия замедляет исполнение программы, поэтому ее использование желательно минимизировать в новом языке. Также на момент проектирования тестовой системы поддержка рефлексии в языке отсутствовала.

**(5) Интеграция с документацией.** Инструменты генерации документации могут добавлять тесты в качестве примеров использования API, или для проверки правильности примеров в документации эти примеры могут компилироваться и запускаться как тесты.

**(6) Тестирование не экспортированных символов.** Произвольная часть программы (компонент, класс, файл и т.д.) могут определять набор символов, но не экспортировать их. Возможность тестирования таких символов позволяет избежать ошибок реализации и реализации.

**(7) Вспомогательные функции.** Чтобы избежать копирования кода, общий код выносят в отдельные функции. Когда тестовый код проектируется подобным образом, отдельные функции нередко нужны только для тестирования, поэтому не должны быть видны основному

коду.

**Критерии внутреннего тестирования**

(1) Поддержка высказываний, проверяющих предположения о состоянии работающей программы;

(2) Возможность отключения проверяющих высказываний при публикации программы;

(3) Поддержка условной компиляции блоков высказываний;

Важно, чтобы тестирующий код не замедлял работу основной программы при публикации. Поэтому важна возможность компилятора не включать тестирующий код в сборку программы.

**(1) Поддержка проверяющих высказываний (assert).** Проверяющие высказывания нужны для возможности проверки на правильность инвариантов или результатов промежуточных операций. Такие проверки должны быть выделены синтаксически с возможностью их удаления в определенных сборках программы, что дает им особенную семантику.

**(2) Возможность отключения проверяющих высказываний.** Как уже было сказано, тестирующий код не должен замедлять программу, когда она находится в руках у пользователя. Поэтому тестирующие высказывания не должны включаться в конечную сборку программы.

**(3) Условная компиляция блоков высказываний.** Тестирующий код может содержать более одного высказывания и требовать промежуточных вычислений. В этом случае полезны блоки кода, которые будут скомпилированы только в тестовой сборке. Также для расширяемости системы внутреннего тестирования важна поддержка пользовательских расширений проверяющих высказываний, которые компилятор сможет удалять из конечных сборок программы. Для этого пользователь может написать свои собственные проверяющие функции и вызывать их внутри условно компилирующихся блоков.

## 2.5. Выбор тестовых систем для анализа

В список языков программирования, рассматриваемых в данной работе входят Java, Kotlin, Rust, Go, Dlang, Swift и F#.

Набор языков программирования для сравнения составлялся таким образом, чтобы он обладал следующими свойствами:

- состоит из компилируемых языков;
- содержит и старшие (Java, Dlang, F#), и более молодые языки (Kotlin, Rust, Go, Swift);
- содержит объектно-ориентированные языки (Java, Kotlin, Dlang), функциональные (F#) и другие (Rust, Go);
- содержит языки, идеологически близкие к экспериментальному языку (Go, Swift).

## 2.6. Сравнение подходов к тестированию в других языках программирования

Сравним по приведенным критериям подходы к тестированию в Java, Kotlin, Rust, Go, Dlang, Swift и F#. Также опишем такие особенности языков, не вписывающихся в рамки выбранных критериев.

### 2.6.1. Kotlin Java JUnit

Языки Kotlin и Java рассматриваются вместе с точки зрения тестового фреймворка JUnit, одного из самых популярных в экосистеме Java.

**Синтаксис.** [7] Функция `add`, складывающая два числа, изображена на рис. 1a. Чтобы с помощью `JUnit` проверять эту функцию на правильность требуется написать тестирующий класс, содержащий как минимум один тестирующий метод, помеченный аннотацией `@Test`, изображенный на рис. 1b.

Класс, обособляющий тестовые методы, виден только для тестового кода. Следовательно, вспомогательные функции написанные в тестовых классах будут видны только тестовому коду. Также вспомогательные функции можно писать в любых классах, и далее добавлять этот класс только в тестовую сборку программы.

**Тестирование не экспортированных символов.** В Java существует режим видимости *package-private*[1], при котором символы видны только в пределах пакета, в котором они определены. Таким образом можно тестировать эти символы, если расположить тестирующий код в этом же пакете.

```kotlin
// Math.kt

class Math {
    fun add(x: Int, y: Int): Int {
        return x + y
    }
}
```

(a) Тестируемая функция на языке *Kotlin*

```kotlin
// MathTest.kt

class MathTest {
    private fun helperFunction() { ... }

    @Test
    fun testAdd(x: Int, y: Int) {
        assertEquals(10, Math.add(6, 4))
    }
}
```

(b) Unit-тест на языке *Kotlin*

Рис. 1: *JUnit Kotlin*

**Запуск тестов.** Процесс запуска тестов будет рассмотрен для системы сборки Gradle , при использовании других подходов идейных

[1] Controlling Access to Members of a Class, Oracle Java Documentation

различий не будет. После вызова пользователем команды запуска тестов:

1. Gradle соберет программу пользователя, состоящую из основных, тестирующих и дополнительных Gradle классов;

2. Gradle запустит программу, указав один из дополнительных классов как входную точку;

3. Запущенный класс с помощью рефлексии найдет все пользовательские тестовые классы [2];

4. Каждый из найденных классов будет запущен указанным классом, предназначенным для запуска тестов и реализующим интерфейс `Runner`[3] в *JUnit*;

5. `Runner`-класс, получив управление, обычно находит все тестовые методы с помощью рефлексии и запускает их [6].

**Дополнительные возможности.** Для подготовки и разрушения тестового окружения предназначены аннотации `BeforeEach`, `AfterEach` и `BeforeAll`, `AfterAll`, которыми помечаются методы, запускающиеся перед каждым, после каждого, перед всеми и после всеми тестами соответственно (рис. 2a). Она реализована стандартным `Runner`-классом, который перед и после запуска тестов находит рефлексией все методы для подготовки и разрушения окружения и запускает их.

Поддержка параметризованных тестов (рис. 2b) реализована аналогично в стандартном `Runner`-классе.

**Расширение системы**. Если пользователь хочет расширить тестовую систему самостоятельно, он должен определить класс, реализующий интерфейс `Runner`. Этот класс будет отвечать за запуск тестов в пределе одного тестирующего класса. Весь процесс запуска тестов может быть переопределен пользователем[4].

[2] Gradle 7.4.2. Test detection

[3] JUnit 4.13. `Runner` interface

[4] Пример класса, переопределяющего процесс запуска тестов

```kotlin
@BeforeAll
fun setUp() { ... }

@BeforeEach
fun prepare() { ... }

@AfterEach
fun sweep() { ... }

@AfterAll
fun tearDown() { ... }
```

(a) Поддержка `BeforeEach` и других методов жизненного цикла в *JUnit Kotlin*

```kotlin
internal class A {
    companion object {
        @JvmStatic
        fun params(): Stream<Arguments> {
            ...
        }
    }

    @ParameterizedTest
    @MethodSource("params")
    fun `should be parsed correctly`(param: String) {
        ...
    }
}
```

(b) Параметризованные тесты в *JUnit Kotlin*

Рис. 2: Возможности *JUnit*

**Обнаружение ошибок.** Тесты считаются провальными, если они вызвали ошибку или исключение. Для того, чтобы определить бросил ли тест исключение *JUnit* оборачивает [5] вызов тестирующей функции в *try-catch* блок. Отрывок исходного кода изображен на 3.

**Внутренние тесты.** В Java `assert`-высказывание (рис. 4a) проверяет условие на правдивость и бросает `AssertError`, если условие ложно. Эти высказывания можно исключить из сборки проекта соответ-

[5] Исходный код метода, проверяющего полученное из теста исключение в *JUnit*

```
try {
    executable.execute();
} catch (Throwable actualException) {
    if (expectedType.isInstance(actualException)) {
        // success
    } else {
        // failure
        throw new AssertionFailedError(...);
    }
}
```

Рис. 3: Отрывок кода, проверяющего исключение, которое бросил тест в *JUnit*

```
assert i == 1;
```

(a) assert в *Java*

```
// эта переменная определяется переменной окружения
// она равна true, если приложение компилируется
// в тестовой сборке
private static final boolean debug = ...;

// ...
if (debug) {
  // Этот блок не компилируется, если
  // debug == false
}
```

(b) Условная компиляция блоков высказываний в *Java*

Рис. 4: Внутреннее тестирование в *Java*

ствующей настройкой компилятора. Для того, чтобы исключить определенный блок высказываний из конечной сборки программы, можно обособить этот блок условной конструкцией, условие которой вычисляется на этапе компиляции и зависит от сборки программы (рис. 4b).

### 2.6.2. Rust

**Синтаксис.** [9] Код, содержащий тестируемую функцию `add` и тестирующий код изображен на рис. 5. В Rust тесты представляют из себя функции, помеченные макросом #[test], которые расположены в помеченном макросом `#[cfg(test)]` тестовом модуле. Тестовый модуль часто располагают в одном файле с тестируемым кодом.

```rust
// add.rs

pub fn add(x: i32, y: i32) -> i32 {
    x + y
}

#[cfg(test)]
mod tests {
    fn helper_function() { ... }

    #[test]
    fn test_add() {
        // we can call helper_function here
        assert_eq!(add(1, 2), 3);
    }
}
```

Рис. 5: Функция `add` и тестирующий ее код

Макрос `#[cfg(test)]` помечает функции, структуры и модули, которые будут включены только в тестовую сборку приложения. Поэтому функции, помеченные этим макросом, и функции, написанные в помеченных этим макросом модулях, будут добавлены только в тестовую сборку приложения, а значит не видны для основного кода.

Rust позволяет проверять на правильность примеры из документации к коду (рис. 6). По-умолчанию каждый пример рассматривается как отдельный тест и запускается для проверки.

```rust
// Returns the sum of two numbers
//  ```
//  let sum = crate::add(5, 10);
//  assert_eq!(sum, 15);
//  ```
pub fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

Рис. 6: Документация в *Rust*, которая используется как юнит-тесты

**Тестирование не экспортированных символов.** Непубличные символы в Rust видны в модуле, в котором они определены, и в его внутренних модулях. Поэтому тестовые модули, которые располагаются в одном модуле с тестируемым кодом, могут тестировать непубличные символы.

**Запуск тестов.** Рассмотрим пример запуска тестов без проверки примеров из документации. При запуске тестов:

1. Компилятор, добавив в сборку все конструкции, помеченные макросом `#[cfg(test)]`, найдет все функции, помеченные макросом `#[test]`;

2. Компилятор добавит к собираемой программе входную точку, имеющую информацию о всех тестовых функциях, и соберет запускаемый файл;

3. Запускаемый файл запустит все тестовые функции параллельно.

Для проверки примеров из документации каждый пример проходит следующую предобработку:

1. Добавляется импорт проекта, к которому относится данный пример;

2. Код примера оборачивается в `fn main()` и становится входной точкой;

**Расширение системы.** Все тесты система находит на этапе компиляции, и нет возможности добавления тестов в процессе работы программы. Поэтому для расширения тестовой системы на Rust часто используют макросы, которые добавляют тесты на этапе компиляции. Пользователь, пишет тестовый код с расширенными возможностями, который при помощи макросов превращается в набор стандартных Rust тестов. Более подробно будет описано далее на примере библиотек *stainless*[6] и *rstest*[7].

**Дополнительные возможности.** Rust не имеет дополнительных способов для подготовки и разрушения тестового окружения, а также для параметризованных тестов. Но данные возможности реализуют библиотеки с помощью макросов.

На рис 7a. показано, как библиотека *stainless* позволяет пользователю подготавливать и разрушать тестовое окружение `before_each`,и `after_each` блоках, а на рис. 7b изображено как раскрываются данные макросы на этапе компиляции, в результате чего получаются обычные тесты, которые поддерживает *Rust*.

Библиотека *rstest* реализует поддержку параметризованных тестов с помощью макроса `#[case]` (рис. 8). *rstest* для каждого параметра параметризованного теста генерирует отдельный unit-тест.

**Обнаружение ошибок.** В Rust провальными считаются те тесты, которые вызвали панику, а успешными, которые вернули значение. Для того, чтобы определить вызвал ли тест панику, вызов каждого теста системой оборачивается[8] в `catch_unwind`, который обнаруживает процесс паники и останавливает его подъем по стеку вызовов. Примерный отрывок кода, реализующего это, изображен на рис. 9.

**Внутренние тесты.** Макрос `debug_assert!` в *Rust* проверят условие и поднимает панику, если это условие ложно (рис. 10a). Вызовы этих макросов остаются только в отладочной сборке программы и удаляются из конечной сборки. Для того, чтобы исключать блоки выска-

[6] Библиотека stainless

[7] Библиотека rstest

[8] Исходный код функции, запускающей тесты в *Rust*

```
describe! stainless {
    before_each {
        // Start up a test.
        let mut stainless = true;
    }
    it ''makes organizing tests easy'' {
        // Do the test.
        assert!(stainless);
    }
    after_each {
        // End the test.
        stainless = false;
    }
}
```

(a) Поддержка `before_each` `after_each` в библиотеке *stainless*

```
mod stainless {
    #[test]
    fn makes_organizing_tests_easy() {
        let mut stainless = true;
        assert!(stainless);
        stainless = false;
    }
}
```

(b) Код, который получается после расширения макросов их *stainless*

Рис. 7: *stainless* для *Rust*

зываний из конечной сборки программы, можно обособлять эти блоки условной конструкцией, где условие вычисляется из версии текущей сборки `cfg!` (рис. 10b).

### 2.6.3. Go

**Синтаксис.** [5] Юнит-тест на языке *Go* для функции, изображенной на рис. 11a, написан на рис. 11b. В *Go* юнит-тесты должны быть расположены в отдельном файле, имя которого имеет суффикс `_test.go`, каждый юнит тест является функцией, имя которой имеет префикс `Test`, эта функция должна принимать ровно один аргумент типа ×

```rust
#[rstest]
#[case(0)]
#[case(1)]
fn fibonacci_test(#[case] input: u32) {
    ...
}
```

Рис. 8: Использование библиотеки *rstest* на *Rust*

```rust
let start = report_time.then(Instant::now);
let result = catch_unwind(AssertUnwindSafe(testfn));
let exec_time = ...
```

Рис. 9: Отрывок кода, проверяющего панику, которую вызвал тест в *Rust*

```rust
    debug_assert!(i == 0);
```

(a) debug_assert в *Rust*

```rust
// получим тип сборки программы
if cfg!(test) {
    // этот блок будет добавлен только в
    // сборку программы "test"
}
```

(b) Условная компиляция блоков высказываний в *Rust*

Рис. 10: Внутреннее тестирование в *Rust*

`testing.T`.

Файлы с суффиксом `_test.go` включаются только в тестовую сборку приложение, поэтому вспомогательные функции определенные в этих файлах не видны для основного кода.

**Тестирование не экспортированных символов.** Область видимости не экспортированных символов совпадает с пакетом, в котором они определены [4]. Поэтому тестовые функции, которые находятся в определенном пакете, могут проверять не экспортированные символы этого пакета.

**Запуск тестов.** Процесс запуска тестов аналогичен Rust. При за-

```go
// add.go

func Add(x, y int) int {
    return x + y
}
```

(a) Тестируемая функция на *Go*

```go
// add_test.go

func helperFunction() { ... }

func TestAdd(t *testing.T) {
    total := Add(1, 5);
    if total != 6 {
        t.Fail()
    }
}
```

(b) Юнит тест на *Go*

Рис. 11: Внешнее тестирование в *Go*

пуске тестов:

1. В сборку добавляется каждый пакет, вместе с файлами, имеющими суффикс названия *_test.go*;
2. Для каждого пакета находятся все тестовые функции;
3. Каждый пакет программы компилируется в отдельный бинарный файл, в который добавляется входная точка, у которой есть информация о всех найденных тестовых функциях;
4. Входная точка запускает все найденные тесты;

**Обнаружение ошибок.** Тесты считаются провальными, если пользователь вызвал метод `t.Fail()`, или была вызвана паника. В *Go* можно с помощью высказывания `defer` и функции `recover()` отловить панику и не дать ей подняться по стеку вызовов. Именно это и используется[9] тестовой системой для обнаружения наличии паники и для ее

[9]Исходный код функции из библиотеки *Gomega*, отлавливающей паники

нейтрализации, что примерно изображено на рис. 12.

```
defer func() {
    if e := recover(); e != nil {
        // recover from error
    }
}()
// execute test
```

Рис. 12: Отрывок кода, который вызывает тест и восстанавливается после паники на *Go*

**Дополнительные возможности.** Несмотря на то, что тестовые функции в *Go* обнаруживаются на этапе компиляции, в *Go* есть возможность создания подтестов во время исполнения программы. Таким образом можно запускать параметризованные тесты, в цикле создавая подтест для каждого набора данных (рис. 13).

```
func TestFibonacci(t *testing.T) {
    for _, n := range numbers {
        t.Run(n, func(t *testing.T) {
            // test fibonacci for n
        })
    }
}
```

Рис. 13: Запуск параметризованных тестов с помощью подтестов в *Go*

**Расширение системы.** Поддержка подготовки и разрушения тестового окружения в *Go* отсутствует, но ее реализует библиотека *Ginkgo*. *Ginkgo* расширяет тестовую систему *Go*, используя ее как входную точку. В коде на рис. 14 когда происходит запуск тестов, *Go* запускает тестовую функцию `TestBooks`, которая передает управление *Ginkgo*, которая далее запускает все определенные с помощью *GinkGo* тесты.

При создании переменной `_` на рис. 14 вызывается функция `Describe`, которая записывает данные о создавшемся тестовом блоке в глобальную переменную *Ginkgo*.

*Ginkgo* поддерживает подготовку и разрушение тестового окружения за счет замыканий. Внутри блока `Describe` можно создать переменную и далее инициализировать ее внутри `BeforeEach` блока перед запуском каждого теста.

```
// запускает ginkgo с помощью тестовой системы Go
func TestBooks(t *testing.T) {
    RegisterFailHandler(Fail)
    // запустить GinkGo
    RunSpecs(t, ''Books Suite'')
}

var _ = Describe(''Book'', func() {
    // храним тестовое окружение в переменной
    var x: Book
    BeforeEach(func() {
        // подготовливаем тестовое окружение
        // засчет того, что переменная x
        // захватывается этим замыканием
    })
    It(''should be a novel'', func() {
        // тест №1
    })
    It(''should be a short story'', func() {
        // тест №2
    })
})
```

Рис. 14: Использование библиотеки *GinkGo* на *Go*

**Внутренние тесты.** В Go проверочные высказывания не поддерживаются, для цели проверок используют условные конструкции (рис. 15a), но в конечной сборке программы эти проверки остаются. Для того, чтобы эти проверки присутствовали только в тестовой сборке программы, их можно обособить блоками, которые компилируются условно. Эти блоки представляют из себя условные конструкции, условия которых зависят от переменных, чьи значения определяются на этапе компиляции. В *Go* это реализуется на уровне файлов: файлы, имя которых заканчивается на *_debug* включаются только в отладочную сборку, а

те имя которых заканчивается на *_release* – в конечную сборку. Таким образом, определив константу разными значениями в паре таких файлов, можно получить константу, значение которой зависит от версии сборки (рис 15b).

```
if i == 0 {
    panic(...)
}
```

(a) Проверки в *Go*

```
// файл config_debug.go
package main
const debug = true
// файл config_release.go
package main
const debug = false
// файл main.go
package main
if debug {
    // если константа debug равна false
    // то этот блок не будет добавлен в сборку
    // константа debug определяется на этапе компиляции
}
```

(b) Условная компиляция на *Go*

Рис. 15: Внутреннее тестирование в *Go*

### 2.6.4. Dlang

**Синтаксис.** [2] В *D* юнит-тесты пишутся рядом с тестируемым кодом, что изображено на рис. 16. Каждый юнит-тест представляет из себя блок, помеченный ключевым словом `unittest`. Эти блоки принято писать внутри классов рядом с тестируемыми методами и также под тестируемыми классами. Поскольку в этом подходе тестирующий код не расположен в отдельной обособляющей области видимости от основного кода, то возникает вопрос, где можно располагать вспомогательные функции, которые были бы видны только для тестов. Для

этого в языке существуют блоки, помеченные `version(unittest)`, в которых могут содержаться символы, которые будут видны только для тестирующего кода.

Блоки `version(unittest)` обособляют участки кода, компилируемые только в режиме `unittest`, а в остальных режимах отбрасываются.

Короткие юнит-тесты бывают полезны в роли примеров использования классов или функций. Поэтому *Dlang* по-умолчанию добавляет юнит-тесты к классам и функциям в документацию в качестве примеров. При этом важно, чтобы программист придерживался конвенции, что юнит-тесты к классу или функции должны следовать сразу под определением класса или функции.

```d
// Math.d

class Math {
    static int add(int x, int y) {
        return x + y;
    }

    unittest {
        assert(add(2, 2) == 4);
    }
}

version(unittest) {
    import std.stdio;
    void helperFunction() { ... }
}

unittest {
    auto math = new Math();
    assert(math.add(1, 5) == 6);
}
```

Рис. 16: Функция `add` и тестирующий ее код на *D*

**Тестирование не экспортированных символов.** В Dlang при-

ватные функции, классы и переменные видны в пределах модуля (файла), в котором они определены. Поэтому `unittest` блоки могут ссылаться на символы, определенные в этом же файле.

**Запуск тестов.** Процесс аналогичен *Rust* и *Go*: компилятор находит все тесты и собирает бинарный файл, который запускает эти тесты.

**Обнаружение ошибок.** После запуска бинарного файла тестовая система запускает каждый тест отдельно. Тесты считаются провальными, если они вызывают ошибки. Вызов каждого теста оборачивается в *try-catch* блок, отлавливающий все ошибки, которые могут быть вызваны в тесте.

**Дополнительные возможности.** В *Dlang* существует поддержка параметризованных тестов с помощью макросов, аналогично *Rust*. На каждый набор параметров макросы генерируют отдельный тест (рис. 17). Каждому тесту можно дать имя, которое будет отображено в отладочной информации. Таким образом у параметризованных тестах имя теста может зависеть от параметров.

```
static foreach (i; 0 .. 2)
    static foreach (j; 0 .. 2)
        @(format!"%s + %s == 1"(i, j))
        unittest
        {
            (i + j).should.eq(1);
        }
```

Рис. 17: Параметризованные тесты в D

**Расширение системы.** *D* не позволяет создавать тесты во время исполнения программы, не поддерживает подготовку и разрушение тестового окружения. Для расширения тестовой системы надо использовать `unittest` блок как стартовую точку, которая будет вызывать тестовый фреймворк.

**Внутренние тесты.** В *D* есть 3 вида контрактов, предназначенных для внутреннего тестирования, которые исключаются из конечной сборки программы: `assert`-высказывания (рис. 18), функциональные

контракты (рис. 19) и проверки инвариантов классов и структур (рис. 20). Функциональные контракты проверяют параметры, получаемые функцией, и возвращаемое значение функции, они могут бросать исключения, если значения некорректны. Если класс или структура имеет метод `invariant`, то он будет вызван при создании и уничтожении объекта, а также перед и после вызова каждого метода.

```
assert(sqrt(4) == 2);
```

Рис. 18: `assert` в D

```
// функция, считающая квадратный корень
long square_root(long x)
in {
    // проверяем, что аргумент >= 0
    assert(x >= 0);
} out (result) {
    // проверяем, что возвращаемое значение
    // близко к корню
    assert((result * result) <= x
        && (result+1) * (result+1) > x);
} do {
    // реализация функции
    return cast(long)std.math.sqrt(cast(real)x);
}
```

Рис. 19: Функциональные контракты в D

### 2.6.5. F#

Будут рассмотрены фреймворки *FsUnit* и *NUnit*. *NUnit* представляет из себя тестовый фреймворк на платформе .NET со всей логикой, реализующей поиск, запуск и вывод результатов тестов. *FsUnit* — обертка для использования нижележащего фреймворка на *F#*.

**Синтаксис.** [3] Для тестирования функции, изображенной на рис. 21a, требуется написать тестирующий класс, содержащий тестирующие

```
struct Date {
    private {
        int year;
        int month;
        int day;
    }
    // проверка инвариантов
    invariant() {
        assert(year >= 1900);
        assert(month >= 1 && month <= 12);
        assert(day >= 1 && day <= 31);
    }
}
```

Рис. 20: Проверки инвариантов в D

методы (рис 21b). Каждая функция-юнит-тест и тип, обособляющий эти функции, должны быть помечены специальными аннотациями.

```
// Program.fs

let add x y = x + y
```

(a) Тестируемая функция на *F#*

```
// Test.fs

[<TestFixture>]
type MyTests () =
    let helper x = ...

    [<Test>]
    let ``Sum of 4 and 6 should be 10`` () =
        add 6 4 |> should equal 10
```

(b) Юнит тест на *F#*

**Тестирование не экспортированных символов.** Один из модификаторов видимости *F#* (`internal`) делает символ доступным только в пределах текущего проекта. Тестирующий код может ссылаться на `internal` символы.

**Запуск тестов.** Процесс запуска тестов в *NUnit* анологичен *JUnit*.Программа компилируется вместе с тестовыми классами, а далее во время исполнения программы эти классы находятся с помощью рефлексии, и тестовые методы в них находятся с помощью рефлексии. Нахождение и запуск тестов производит класс, реализующий интерфейс `IProjectLoader`.

**Обнаружение ошибок.** Аналогично другим языкам, тесты считаются провальными, если они бросили исключение. Вызовы тестов оборачиваются в *try-catch* блоки, которые обрабатывают исключения.

**Дополнительные возможности.** Подготовка и разрушение тестового окружение поддерживается аналогично *JUnit*. Пользователь должен определить методы у тестового класса и пометить их аннотациями `[<SetUp>]` и `[<TearDown>]`. Параметризованные тесты также реализованы аналогично *JUnit*.

**Расширение системы.** Для дальнейшего расширения тестовой системы пользователь должен создать свой класс, переопределяющий поведение одного из встроенных классов в *NUnit* (аналогично *JUnit*). Например, пользователь может написать свой класс, реализующий интерфейс `IProjectLoader`, который будет отвечать за поиск и запуск тестов.

**Внутренние тесты.** Поддерживаются `assert` высказывания, которые удаляются из конечной сборки программы. Для условной компиляции существуют директивы компилятора, которые позволяют пропускать и включать код в сборку программы (рис. 22).

```
#if VERSION1
let function1 x y = ...
#else
let function1 x y = ...
#endif

let result = function1 10 20
```

Рис. 22: Условная компиляция в F#

### 2.6.6. Swift

**Синтаксис.** Функция `add` (рис. 23a) складывает два числа. Чтобы протестировать эту функцию, мы должны создать в отдельном тестовом модуле тестовый класс (рис. 23b), наследующийся от `XCTestCase` и написать в нем тестовый метод, имя которого начинается с `test`. Тестовый модуль включается только в тестовую сборку программу, поэтому все символы, определенные в нем, не видны для основного кода, но видны для тестов.

```swift
internal func add(x: Int, y: Int) -> Int {
    return x + y
}
```

(a) Тестируемая функция на *Swift*

```swift
@testable import mymodule

func add(x: Int, y: Int) -> Int {
    return x + y
}
```

(b) Юнит тест на *Swift*

**Тестирование не экспортированных символов.** В *Swift* символы, имеющие модификатор видимости `internal`, имеют область видимости, совпадающие с модулем, где они определены. Модулем считается набор пакетов, все приложение может состоять только из 1 модуля. Тестовый модуль может импортировать модуль с основным кодом (рис. 23a), добавив аннотацию `testable` (рис. 23b), что сделает `internal` символы видными для тестового модуля.

**Запуск тестов.** Тесты находятся на этапе компиляции, а далее список тестов передается определенным классам, которые запускают данные тесты. При запуске тестов:

1. Компилятор находит все тестовые классы, наследующиеся от `XCTestCase`, и все тестовые методы в них, начинающиеся с *test*;

2. Формируется список всех найденных тестовых классов, где каждый тестовый метод представляется в виде замыкания;

3. Каждый тестовый класс создает `TestSuite`, куда по-умолчанию добавляются все тестовые методы этого класса;

4. Набор полученных `TestSuite` запускается;

**Дополнительные возможности.** Класс `XCTestCase` имеет методы `setUp` и `tearDown`, которые позволяют определить процесс создания и разрушения тестового окружения (рис. 24).

```
class AddTests: XCTestCase {
    override func setUp() { ... }
    override func tearDown() { ... }
}
```

Рис. 24: Подготовка и разрушения тестового окружения в *Swift*

**Расширение.** Тестовая система *XCTest* позволяет в процессе исполнения программы определять тесты. Для этого достаточно переопределить процесс создания `TestSuite` в классе `XCTestCase`. В созданный `TestSuite` можно добавить произвольное количество тестов. Таким образом можно реализовать поддержку параметризованных тестов, которой нет изначально.

**Обнаружение ошибок.** Каждый тест запускается в *do-catch* блоке, который отлавливает и обрабатывает все полученные исключения.

**Внутренние тесты.** Поддерживаются `assert` высказывания (рис. 25a), которые включаются только в отладочную версию программы. Для условной компиляции используются директивы компилятора (25b).

```
assert(i == 0)
```

(a) `assert` в *Swift*

```
#ifdef DEBUG
    // Компилируется только в отладочной
    // сборке программы
#endif
```

(b) Условная компиляция на Swift

## 2.7. Итоги сравнения

### 2.7.1. Внешнее тестирование

| . | JUnit | Rust | Go | D | F# | Swift |
|---|---|---|---|---|---|---|
| Программное создание тестов | + | - | + | - | + | + |
| Подготовка и разрушение окружения для тестов | + | - | - | - | + | + |
| Поиск тестов на этапе компиляции | - | + | + | + | - | + |
| Без рефлексии | - | + | + | + | - | + |
| Интеграция с документацией | - | + | - | + | - | - |
| Тестирование приватных символов | + | + | + | + | + | + |
| Возможность написания вспомогательных функций | + | + | + | + | + | + |

Все рассматриваемые системы поддерживают тестирование символов с ограниченной областью видимости и возможность написания вспомогательных функций.

На примере Rust можно определить, что генерация тестов на этапе компиляции является неподходящим для экспериментального языка подходом. При использовании макросов в Rust ухудшаются диагностические сообщения, которые видит пользователь, а компилятор внутри макросов не производит все проверки типовой совместимости.

Одновременно возможность программного создания тестов и поиска

тестов на этапе компиляции поддерживают только 2 из рассматриваемых тестовых системы: Go и Swift. Go, в отличие от Swift, не имеет поддержки подготовки и разрушения окружения для тестов.

Поскольку в экспериментальном языке правила видимости не экспортированных символов совпадают с правилами видимости не экспортированных символов в Go, подход Go является самым оптимальным для экспериментального языка при тестировании внутренних символов.

Интеграцию с документацией поддерживают только Rust и D, и причем совершенно разными способами: Rust считает документацию тестами, а D тесты считает документацией. Подход D ближе к идеологии экспериментального языка, чем Rust, потому что в Rust требуются отдельные системы, извлекающие код из комментариев, а комментарии в экспериментальном языке отбрасываются на этапе компиляции, обработка комментариев усложнила бы набор правил в экспериментальном языке.

Таким образом, сочетая подходы Swift и D можно получить желаемый результат.

### 2.7.2. Внутреннее тестирование

| . | Java | Rust | Go | D | F# | Swift |
|---|---|---|---|---|---|---|
| Поддержка `assert` | + | + | - | + | + | + |
| Возможность отключения `assert` | + | + | - | + | + | + |
| Условная компиляция блоков высказываний | + | + | + | + | + | + |

Все рассматриваемые языки, кроме *Go*, имеют поддержку `assert` высказываний, которые можно исключать из конечной сборки приложения. Поддержка условной компиляции реализована во всех языках, хотя стоит заметить, что Java, Rust и Go для нее используют условные блоки, не создавая новой языковой конструкции, в отличие от F#, D и

Swift, которые используют директивы компилятора. Если экспериментальный язык будет следовать подходам F#, D и Swift, то в нем будет два способа выражать условную компиляцию: условными блоками и директивами компилятора. Поэтому подходы Java, Rust и Go ближе к идеологии экспериментального языка, в котором не должно существовать несколько языковых конструкций для одной идеи. Также в Go и в Java пользователю придется сделать больше действий для определения переменной, хранящей в себе информацию о типе сборки, в отличие от остальных языков. Поскольку экспериментальный язык старается позволять пользователю кратко излагать свою мысль, подход Rust к условной компиляции является предпочтительным.

У D самый большой набор возможностей, связанных с внутренним тестированием: кроме `assert` Dlang поддерживает функциональные контракты и проверки инвариантов объектов.

Таким образом сочетая условную компиляцию из Rust и контракты из D можно получить систему внутреннего тестирования, наиболее подходящую для экспериментального языка.

# 3. Реализация

## 3.1. Предложение по реализации тестовой системы

Исходя из проведенного анализа формулируем оптимальный подход к тестированию для экспериментального языка.

### 3.1.1. Внешнее тестирование

В экспериментальном языке должна быть возможность написания простых unit-тестов, которые будут использоваться в качестве примеров в документации, как в *Dlang* (2.7.1). Для этого в языке должна быть возможность написания блоков, которые содержат примеры кода, под функциями, структурами или классами, для которых написан пример кода (рис. 26). Все примеры будут найдены компилятором и запущены на этапе тестирования, компилятор также будет удалять примеры из конечной сборки программы.

Примеры к коду, в отличие от многих других языков программирования, располагаются под основным кодов в блоках, а не в комментариях, что упрощает анализ кода и написание этих примеров для пользователя.

```
function add(x: число, y: число) {
    возвращает число x + y
}

пример {
    результат = add(1, 2)
    проверяем, что результат равен 3
}
```

Рис. 26: Интеграция с документацией в экспериментальном языке

Другие тесты в экспериментальном языке должны располагаться в файлах, имеющих суффикс названия *_test*. Эти файлы могут располагаться в одном пакете с тестируемым кодом, за счет чего не экспортированные символы основного кода (рис. 27а) будут видны тестирующему

коду.

Тесты должны представлять из себя набор тестирующих классов и тестирующих методов в них. В каждом тестовом классе пользователь может переопределить методы подготовки и разрушения тестового окружения (рис. 27b).

```
// файл add
пакет abc

function add(x, y) {
    возвращает сумму чисел x + y
}
```

(a) Тестируемая функция на экспериментальном языке

```
// файл add_test
пакет abc

тестирующий класс, наследующийся от класса TestCase {
    подготовить окружение {
        // ...
    }
    разрушить окружение {
        // ...
    }
    // не обязательная функция,
    // позволяющая добавлять и убирать тесты из набора
    // во время исполнения программы
    функция, возвращающая набор тестов {
        // ...
    }
    тестовый метод 1 {
        // ...
    }
    тестовый метод 2 {
        // ...
    }
}
```

(b) Тестовый класс на экспериментальном языке

При запуске тестов должен выполняться следующий алгоритм, схо-

жий с алгоритмом запуска тестов в Swift:

1. Компилятор находит все тестирующие классы и тестирующие методы в них;

2. У каждого тестового класса вызывается метод, возвращающий набор тестов `TestSuite`;

3. Каждый полученный набор тестов запускается, с учетом подготовки и разрушения тестового окружения;

Данный алгоритм поддерживает создание и разрушение тестового окружения, и за счет того, что для создания тестового набора вызывается метод тестового класса, пользователь может переопределить тестовый набор, добавив или удалив оттуда тесты.

### 3.1.2. Внутреннее тестирование

Как и в большинстве рассмотренных языков, в экспериментальном языке должно быть проверочное высказывание, которое проверяет условие на правдивость и поднимает панику, если условие ложно (рис. 28a). Это высказывание должно присутствовать только в отладочной сборке программы.

Для того, чтобы пользователь смог разрабатывать расширения `assert` высказываний и мог использовать более сложную логику во внутренних тестах, экспериментальный язык должен поддерживать условную компиляцию блоков высказываний. Условная компиляция блоков должна быть реализована на основе условных конструкций (рис. 28b), условие которых вычисляется на этапе компиляции и зависит от типа текущей сборки программы, иначе в языке будет несколько языковых конструкций, выражающих одну мысль (2.7.2).

Также в экспериментальный язык следует внести систему контрактов, схожую с системой в D (2.7.2). Для проверки входных параметров и возвращаемого значения функции в экспериментальном языке должны быть соответствующие блоки, которые пользователь может написать в

определении любой функции (рис. 28c). Для проверки внутренних инвариантов классов и структур в экспериментальном языке должен быть соответствующий блок, который пользователь может написать внутри определения класса или структуры (рис. 28d).

```
assert(condition)
```

(a) `assert` в экспериментальном языке

```
if DEBUG {
    // этот блок будет добавлен только в
    // отладочную сборку программы
}
```

(b) Условная компиляция в экспериментальном языке

```
function f(x)
проверка параметра x {
    // ...
}
проверка возвращаемого значения y {
    // ...
}
реализация функции {
    // ...
}
```

(c) Проверка входных и возвращаемых данных из функции

```
структура или класс {
    блок invariant {
        // проверяет соблюдение
        // внутренних инвариантов
    }
}
```

(d) Проверка инвариантов

## 3.2. План реализации тестовой системы

Поскольку обсуждается реализация тестовой системы для экспериментального языка программирования, следует учитывать нестабильность языка программирования как программного продукта. В про-

цессе реализации тестовой системы язык развивается, получая новые языковые возможности и теряя другие. Исходя из этого было принято решение разрабатывать тестовую систему с помощью итеративного подхода. В данной работе будет показана первая итерация разработки тестовой системы.

Цель первой итерации тестовой системы – проверка наличия необходимых языковых средств в языке и проверка корректности стандартных библиотек языка с помощью разработанного тестового покрытия. Поэтому из приведенного выше предложения (3.1) было выбрано минимальное множество функциональных требований для внешнего тестирования, которые требуется реализовать. Реализованная тестовая система в следующих итерациях может быть расширена до тестовой системы, описанной в предложении.

## 3.3. Интерфейс командной строки тестовой системы

Запуск юнит-тестирования осуществляется через командную строку (рис. 29). Существует 2 режима запуска тестовой системы: тестирование одного пакета, тестирования множества пакетов. При запуске в режиме тестирования одного пакета пользователь должен передать компилятору путь до этого пакета. В режиме тестирования множества пакетов пользователь передает путь до папки, содержащей пакеты.

При переданном флаге *-json* последняя строка вывода тестовой системы содержит данные о количестве упавших и пройденных тестов в *json* формате (Рис. 30).

```
compiler.exe test <path> [-repo] [-json]

Примеры:
compiler.exe test path.to.package
compiler.exe test myproject -repo
```

Рис. 29: Команда запуска юнит-тестирования

**Пакеты тестируются отдельно**. Поскольку в языке нет методов

восстановления после исключений, любой тест может остановить работу всей тестовой системы, бросив исключение. Поэтому было принято решение запускать тесты для каждого пакета отдельно, чтобы исключение могло остановить работу тестовой системы только в пределах одного пакета.

**Вывод тестовой системы**. Тестовая система выводит имя каждого тестируемого пакета и набор тестов в этом пакете (рис. 30). Выводится статус каждого теста и *stdout*, если тест в него что-то напечатал. Тесты формируют иерархическую структуру, вложенность тестов отражена в количестве отступов слева. В конце тестирования выводится краткая информация о количестве пройденных и упавших тестов, при тестировании множества пакетов (рис. 31) также выводится информация о количестве удачно запущенных пакетов и имена пакетов, содержащие провальные тесты.

```
...
== TEST test1
-- PASS
== TEST test2
-- PASS
== TEST test3
-- PASS

TESTS 23 | SUCCESS 23 | FAILED 0
{''passed'': 23, ''failed'': 0}
```

Рис. 30: Вывод при тестировании одного пакета с *JSON*

## 3.4. Написание юнит-тестов

Пользователь может юнит-тестировать любой пакет, для этого он должен создать поддиректорию *unit-tests* в пакете и добавить в нее файлы с кодом. Несмотря на то, что файлы лежат в поддиректории, они будут принадлежать пакету, поэтому в этих файлах можно писать произвольный код.

```
...
==== TEST for z
---- PASS
-- PASS

TESTS 13 | SUCCESS 13 | FAILED 0


##### SUMMARY #####

There are failed tests in packages:

 - math (failed 1 of 8)
 - fmt (failed 1 of 13)

PACKAGES: TOTAL 15 | EXECUTED 15 | CRASHED 0
TESTS: TOTAL 130 | SUCCESS 128 | FAILED 2
```

Рис. 31: Вывод при тестировании множества пакетов

В файлах, лежащих в папке, будут найдены все экспортированные функции. Эти функции считаются тестовыми, каждая из них будет запущена как тест. Каждая тестовая функция обязана принимать только один аргумент параметра `testing.T` и ничего не возвращать. Тип `testing.T` объявлен в тестовой библиотеке.

## 3.5. Программный интерфейс тестовой библиотеки

Тестовая библиотека позволяет получать статус теста, помечать тесты как провальные и запускать подтесты. Взаимодействие с тестовой библиотекой происходит через объект, реализующий интерфейс `testing.T` (рис. 33). Объект этого типа получает каждая тестовая функция.

- `fail()`. Помечает текущий тест как провальный.
- `assertEquals(a, b)`. Проверяет на равенство оба аргумента $a$, $b$ и помечает тест как провальный, если аргументы не равны. Используется проверка на глубокое равенство с помощью рефлексии.

```
// файл math/add
пакет math

function add(x, y) {
    возвращает сумму чисел x + y
}
```

(a) Тестируемая функция на экспериментальном языке

```
// файл math/add_test
пакет math

импорт типа testing.T из тестовой библиотеки

public function test_add(t: testing.T) {
    t.assertEquals(add(1, 3), 4)
}
```

(b) Тестовая функция на экспериментальном языке

Рис. 32: Функция и тест к ней на экспериментальном языке

- `assertTrue(condition)`. Помечает текущий тест как провальный, если `condition == false`.

- `run(name, test)`. Создает для данного теста подтест с именем `name` и тестовой функцией `test`, возвращает `true`, если подтест успешно завершился. Тесты формируют древовидную структуру, если хотя бы один дочерний тест провалился, то считается, что тест-родитель тоже провальный. Этот метод полезен, когда есть набор параметров, для которых надо запустить одинаковый тестовый код. Для каждого тестового параметра можно в цикле создать отдельный тест, который захватывает этот параметр как замыкание. На рис. 34 изображен тест, проверяющий корректность функции `isLetter` на *ASCII* буквах.

```
interface testing.T {
    fail()
    assertEquals(a, b)
    assertTrue(condition: логическое значение)

    run(name: строка, test: тестовая функция)
}
```

Рис. 33: Интерфейс `testing.T`

```
exported function f(t: testing.T) {
    // i -- ascii символ
    // перебираем ascii буквы нижнего регистра
    for i from 97 to 122 {
        t.run(..., function(t: testing.T) {
            // функция isLetter должна возвращать true,
            // если переданный символ -- буква
            t.assertTrue(isLetter(i))
        })
    }
}
```

Рис. 34: Параметризованные тесты через `testing.T`

## 3.6. Реализация поиска тестовых функций

**Load**. На этапе загрузки (2.2) была добавлена логика, обрабатывающая тестовые папки *unit-test*. При загрузке в список файлов пакета добавляются файлы из папки *unit-test*, если она существует. Таким образом эти файлы становятся частью пакета.

После разбора каждого файла и перед слиянием совершается проход по тестовым файлам, где помечаются все экспортированные функции. Было добавлено поле `IsTestFunc` в тип *AST*-узла функции, на данном проходе у всех найденных функций выставляется значение этого флага *true*.

## 3.7. Входная точка в тестовую систему

Тестовая система генерирует запускаемый файл, который запускает все тесты и выводит результаты. Для этого тестовая система автоматически создает входную точку в приложение, содержащую всю информацию о всех найденных тестах. При запуске приложения входная точка запускает все найденные тесты.

**Load**. Для этого после загрузки тестируемого пакета создается *AST*-узел для входной точки приложения. Этот узел импортирует тестируемый пакет и тестовую библиотеку (3.5). При загрузке этого файла загружается тестовая библиотека и все ее зависимости.

**Lookup**. После того, как компилятор разрешил все имена, собирается информация о всех объявленных тестовых функциях, используя флаг `IsTestFunc`, описанный выше. В автоматически созданную входную точку приложения добавляются ссылки на тестовые функции.

Входная точка приложения, содержащего изображенные на рис. 35a тестовые функции, представлена на псевдокоде на рис. 35b. Она содержит вызов функции `testMain`, реализующей логику запуска тестовой системы. В функции `testMain` передается массив найденных тестовых функций.

## 3.8. Реализация тестовой библиотеки и циклические зависимости

**testing.T**. Класс `T`, реализующий интерфейс `testing.T` (рис. 33), изображен на рис. 36. При создании нового теста создается новый экземпляр этого класса, сохраняющий ссылку на тест-родитель (`parent`). Таким образом экземпляры класса `testing.T` формируют древовидную структуру.

Экземпляры класса `T` хранят статус тестов в поле `failed`. Метод `fail()` устанавливает значение `true` в поле `failed` и рекурсивно вызывает метод `fail()` родительского теста. Все `assert` вызывают метод `fail()` при ложных условиях.

```
// файл math/add_test
пакет math

импорт типа testing.T из тестовой библиотеки

exported function test_add(t: testing.T) { ... }

exported function test_add2(t: testing.T) { ... }
```

(a) 2 тестовые функции

```
// входная точка
импорт функции testMain из тестовой библиотеки
импорт функций test_add и test_add2 из math

входная точка {
    testMain([test_add, test_add2])
}
```

(b) Созданная входная точка тестирующей системы

Рис. 35: Входная точка тестовой системы

Метод `run(name, testfn)` создает новый экземпляр класса `T`, передавая ему текущий тест в качестве родителя, запускает тестовую функцию и выводит диагностическую информацию по поводу данного теста в *stdout*.

**Входная точка testMain**. Входная точка тестовой системы получает набор найденных тестовых функций. Для каждой тестовой функции она создает новый экземпляр класса `T` и вызывает метод `run`, передавая ему тестовую функцию, что производит запуск каждого теста. Если запуск теста был успешен, то увеличивается счетчик успешных тестов. В конце выводится диагностическая информация о результатах тестирования. Реализация функции `testMain` изображена на рис. 37.

**Устранение циклических зависимостей**. Тестовая система должна позволять тестировать все стандартные библиотеки, включая саму себя. При тестировании некоторых стандартных библиотек могут возникнуть запрещенные в экспериментальном языке (2.1) циклические зависимости, если тестовая библиотека сама использует эти библио-

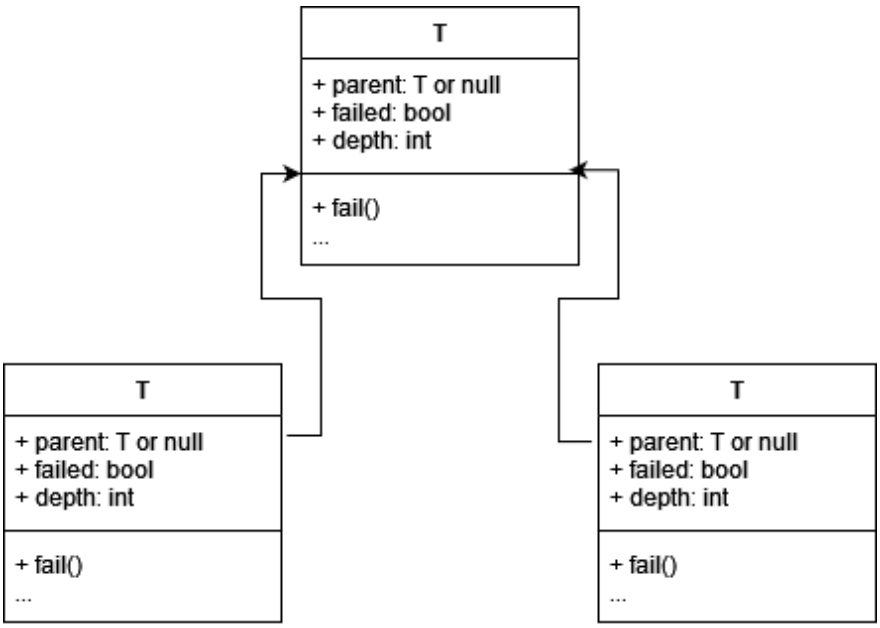


Рис. 36: Класс `T`

теки. Для устранения циклических зависимостей тестовая библиотека была спроектирована так, чтобы при тестировании пользовательского кода в графе зависимостей не было циклов, если графе зависимостей пользовательского кода нет циклов. Покажем архитектуру тестовой системы в виде графа зависимостей (рис. 38):

`testing.T` – описанный выше (рис. 33) интерфейс, не содержащий реализации.

`user package` – тестируемый пользовательский пакет (рис. 35a). Зависит от `testing.T`, так как тестовые функции принимают этот тип как аргумент и транзитивно использует библиотеки `libs1`.

`T implementation` – описанная выше (рис. 36) реализация интерфейса `T` и входная точка `testMain` в тестовую систему (рис. 37). Реализует `testing.T`, транзитивно использует библиотеки `libs2` и может зависеть от `user package`, если тестируемый пакет является стандартной библиотекой.

`entry` – автоматически созданная входная точка (рис. 35b). Создает объект класса `T` из `T implementation`, ссылается на тестовые функции из `user package`.

**Лемма.** *Пусть*

```
function testMain(testFns: массив тестовых функций) {
    passedCount = 0

    for testFn in testFns {
        t = new object of class T
        passed = t.run(name, testFn)
        if passed {
            passedCount++
        }
    }

    println(...)
}
```

Рис. 37: Реализация `testMain` на псевдокоде

1. *в графе зависимостей пользовательского кода* ***`user package`*** *и используемых библиотек* ***`libs1`*** *нет циклов;*

2. *в графе зависимостей* ***`T implementation`***, ***`testing.T`*** *и* ***`libs2`*** *нет циклов;*

3. *пользовательсий код* ***`user package`*** *не может (транзитивно) зависеть от* ***`T implementation`*** *(но эти вершины могут совпадать), то есть* ***`T implementation`*** $\notin$ ***`libs1`***.

*Тогда в графе* G, *полученном из этих двух графов объединением множеств вершин (множества* ***`lib1`*** *и* ***`lib2`*** *могут пересекаться, или вершины* ***`T implementation`*** *и* ***`user package`*** *могут совпадать, или* ***`user package`*** *может содержаться в* ***`libs2`***) *и добавлением вершины* ***`entry`*** *(описанные выше зависимости), не будет циклов.*

*Доказательство.* Пусть $G_1$ – первый граф из `user package` и `libs1`, а $G_2$ – второй из `T implementation`, `testing.T` и `libs2`. Если в новом графе существует цикл, то он должен содержать вершины из обоих графов, иначе он полностью принадлежит одному ациклическому графу.

Заметим, что вершины `testing.T`, `entry` не принадлежат никакому циклу.

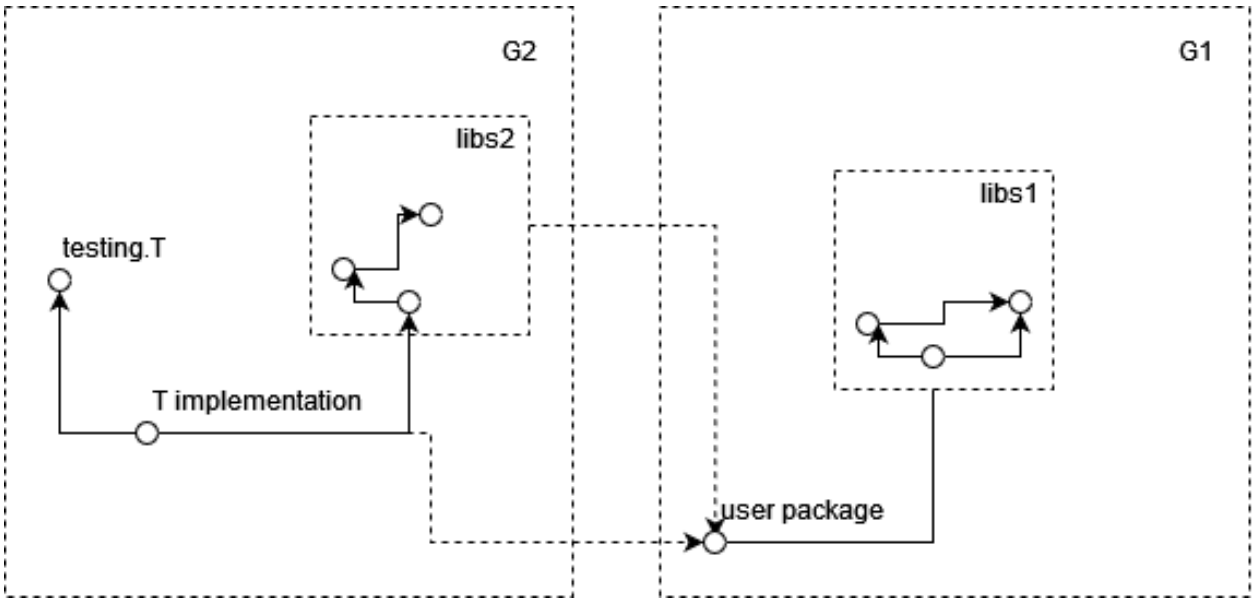


Рис. 38: Граф зависимостей

Заметим, что при объединении `libs1` и `libs2` не может возникнуть циклов, так как если существует цикл, то он полностью лежит в графе $G_1$ или $G_2$ по определению `libs1` и `libs2`.

Заметим, что ни в каком цикле не могут одновременно содержаться вершины `T implementation` и `user package`, так как иначе `user package` транзитивно зависит от `T implementation`.

Пусть существует цикл в графе $G$, тогда по предыдущим замечаниям он проходит через вершину `T implementation` или `user package`, причем только через одну из них.

Тогда не существует цикла, содержащего `T implementation`, так как иначе этот цикл бы проходил только по вершинам `libs1`, `libs2` и `T implementation`, то есть лежал бы полностью в $G_2$. Аналогично, не существует цикла, содержащего `user package`.

Следовательно, в графе $G$ не существует циклов.

□

**Следствие.** *Если пакеты* ***`user package`*** *и* ***`T implementation`*** *можно скомпилировать, и ни одна библиотека не зависит от* ***`T implementation`****, то при тестировании библиотеки* ***`user package`*** *не возникнет циклических зависимостей.*

*Доказательство.* Поскольку в экспериментальном языке запреще-

ны циклические зависимости, то возможность скомпилировать `user package` и `T implementation` влечет предположения 1 и 2 леммы. Из того, что ни одна библиотека не зависит от `T implementation`, следует предположение 3. Выполнены условия леммы. □

Теорему можно усилить, добавив ребра в граф.

## 3.9. Тестирования множества пакетов

Режим множественного тестирования пакетов получает на вход папку, в которой лежат пакеты. Сначала в данной папке рекурсивно находятся все папки, которые являются пакетами на экспериментальном языке программирования. Алгоритм считает, что папка является пакетом, если она содержит файлы с расширением `.code` (2.2), содержит папку *unit-tests* с файлами `.code`, имя папки состоит из латинских букв и символа подчеркивания.

Для каждого найденного пакета компилятор запускает режим тестирования одного пакета и собирает запускаемый файл. Далее все запускаемые файлы запускаются последовательно и собирается общая диагностическая информация.

## 3.10. Выдача диагностической информации

**Режим тестирования одного пакета**. При тестировании одного пакета запоминается общее количество тестов и количество провальных тестов. После завершения тестирования на экран печатается информация, изображенная на рис. .

**Режим тестирования множества пакетов**. При тестировании нескольких пакетов для каждого пакета включается выдача диагностики в формате *JSON*. После тестирования каждого пакета *JSON* диагностическая информация собирается из *stdout*, разбирается и добавляется в общую диагностическую информацию.

## 3.11. Отображение ошибок компиляции

Все тестовые функции должны иметь фиксированную сигнатуру: принимать 1 аргумент типа `testing.T` и ничего не возвращать. Для каждой тестовой функции с неверной сигнатурой компилятор выводит ошибку компиляции.

При реализации проверки сигнатур функций была переиспользована логика компилятора без написания дополнительных проходов. Когда программа компилируется в режиме тестирования, если пользователь допустил ошибку в декларации тестовой функции, компилятор замечает ошибку типов во входной точке тестовой системы `testMain` (в массиве присутствует функций с неправильной сигнатурой). Перед началом компиляции в режиме тестирования текущая реализация `ErrorAdapter` подменяется объектом типа `UnittestErrorAdapter`. `UnittestErrorAdapter` осуществляет отображение ошибок компиляции из автоматически созданного файла входной точки в пользовательский код, в результате чего пользователю выводятся легко читаемые ошибки, которые указывают на функции с некорректными сигнатурами.

# 4. Тестирование

В процессе апробации тестовой системы были разработаны тесты стандартной библиотеки, библиотеки среды исполнения языка, самой тестовой системы и компилятора.

**Стандартные библиотеки**. Суммарно было написано 130 тестов, покрывающих 15 стандартных библиотек (всего реализовано 22 стандартные библиотеки). В список покрытых тестами библиотек входят контейнеры, математический пакет, форматный вывод и рефлексия.

**Среда исполнения**. Экспериментальный язык для своей работы на каждой программно-аппаратной платформе требует набор низкоуровневых библиотек. В список таких библиотек, покрытых тестами, входят низкоуровневая реализация рефлексии и лексикографические сравнения строк.

**Тестовая система**. Входная точка тестовой системы и логика запуска тестов представлены в виде стандартной библиотеки, которая, как и другие библиотеки, покрыта тестами. Это возможно благодаря тому, что архитектура тестовой системы позволяет тестовой системе проверять на корректность саму себя (3.8).

**Компилятор**. Компилятор экспериментального языка программирования проверяется на корректность всеми тестами, упомянутыми выше. Каждый тест стандартной библиотеки и среды исполнения можно рассматривать как исполняемую документацию для компилятора экспериментального языка программирования. Таким образом все упомянутые выше тесты проверяют на правильность логику компилятора. Например, правильность представления каждого типа в памяти проверяют тесты на библиотеку рефлексии.

# 5. Заключение

В ходе данной работы была разработана система, позволяющая тестировать на правильность программы, написанные на экспериментальном языке программирования и отдельные компоненты этих программ. Данная система была внедрена для тестирования компилятора, стандартной библиотеки языка и пользовательских библиотек.

Были выполнены следующие задачи:

- Выбран ортогональный набор критериев для сравнения подходов к тестированию;
- Проанализированы существующие подходы к тестированию и их реализации в других языках и библиотеках по выбранному набору критериев;
- Изучены ключевые особенности экспериментального языка и их влияние на выбор тестирующего подхода;
- Сделано предложение, которое включает в себя синтаксис тестирующей системы, и ее реализацию;
- Спроектирована и реализована система, позволяющая писать и запускать тесты на экспериментальном языке программирования;
- Тестирующая система внедрена в стандартные библиотеки экспериментального языка и библиотеки среды исполнения;

Исходный код проекта закрыт.

# Список литературы


[1] CUnit. CUnit. — 2021. — URL: http://cunit.sourceforge.net/.

[2] D documentation: Unit Tests. — 2021. — URL: https://dlang.org/spec/unittest.html.

[3] FsUnit Official Documentation. — 2021. — URL: http://fsprojects.github.io/FsUnit/.

[4] Go Doc: Names. — 2021. — URL: https://go.dev/doc/effective_go#names.

[5] Go Tutorial: Add a Test. — 2021. — URL: https://go.dev/doc/tutorial/add-a-test.

[6] Gradle Doc: Test Detection. — 2021. — URL: https://docs.gradle.org/current/userguide/java_testing.html#sec:test_detection.

[7] JUnit User Guide. — 2021. — URL: https://junit.org/junit5/docs/current/user-guide/.

[8] Pytest. pytest: helps you write better programs. — 2021. — URL: https://docs.pytest.org/en/6.2.x/#module-pytest.

[9] Rust Docs: How to Write Tests. — 2021. — URL: https://doc.rust-lang.org/book/ch11-01-writing-tests.html.

[10] Sanket. The exponential cost of fixing bugs. — 2019. — URL: https://deepsource.io/blog/exponential-cost-of-fixing-bugs/.

[11] Tiobe. TIOBE Index for December 2021. — 2019. — URL: https://www.tiobe.com/tiobe-index/.