Функциональное программирование

Конспект лекций

Aleksandr Penskoi

current

Оглавление

1 Example. Header 1

1.1 Example. Header 2

1.1.1 Example. Header 3

1.1.1.1 Example. Header 4

1.1.1.1.1 Example. Header 5

Цитата или предупреждение

Some text with **bold** and footnote[1]. Arrow: $\rightarrow$. Math example: $\frac{1}{x^y}$. And correct em dash — bla.

```
class Database {
    void setLogin(String login);
    void setPassword(String password);
    Connection connect() throws InvalidCredentialsException;
}
```

2 Введение. Обзор курса. Особенности ФП

2.1 Рационализация

Низкоуровневые ЯП (машинные), почти не используются:

Если говорить о сегодняшнем времени, то низкоуровневое программирование достаточно редко используется и его почти нигде нет. Однако, оно все еще актуально в некоторых специфических областях:

- Разработка операционных систем.
- Написание драйверов и микроконтроллеров.
- Решение performance critical задач (например, браузеры).

Языки, используемые для этих целей, и их недостатки:

- C — не предоставляет большой свободы действий, то есть нет большого выбора различных вариантов реализаций.
- C++ — в современном мире его сложно назвать низкоуровневым языком. Он обладает множеством библиотек, большой гибкостью, поддержкой ООП, возможностью управления памятью и автоматизацией. Тем не менее он позволяет обращаться к внутренним частям компьютера, что делает его низкоуровневым.

Также существуют относительно новые низкоуровневые языки, которые используют принципы функциональных языков:

- Rust — система типов сильно напоминает язык программирования Haskell. Rust использует конструкции для описания поведения типов, подобные Typeclass в Haskell, называемые Traits. Rust поддерживает вывод типов и выражения на уровне системы, что делает его мощным инструментом для разработки системного ПО.
- Zig — позволяет использовать вычисления на уровне компиляции (compile time) и во время выполнения (runtime), что позволяет создавать более эффективный и гибкий код. Compile time вычисления полезны для проверки свойств типов заранее, а если результат зависит от входных параметров, можно воспользоваться runtime вычислениями. Их можно комбинировать, заранее подготавливая данные на уровне компиляции и работать с ними на уровне выполнения программы.

Современные ЯП — структурные. Эффективность — доказана:

Сегодня структурное программирование является доминирующей парадигмой.

Что такое структурное программирование?

Структурное программирование состоит из последовательных блоков инструкций, каждый из которых имеет единственную точку входа и выхода. Это самый популярный стиль, используемый почти во всех современных языках программирования.

[1] Footnote example.

Недостатки структурного программирования:

- Накладывает ограничения, затрудняющие написание тонко оптимизированного кода.

Однако современные компиляторы отлично справляются с задачами оптимизации. Кроме того, читаемость кода в структурном стиле значительно выше, что делает этот подход предпочтительным.

Nota Bene: Почти все, а возможно и все, функциональные языки программирования являются структурными.

ООП получило широкое распространение, хотя:

- Имеет концептуальные проблемы.
- Заявлена “интуитивная понятность”, но книги по Design Patterns.
- Эффективность не доказана.
- Получило распространение вместе с GC и полиморфизмом.

Объектно-ориентированное программирование (ООП) — это одна из наиболее популярных концепций в программировании. Многие разработчики знакомы с ней и успешно применяют её на практике, поскольку основные идеи ООП интуитивно понятны: окружающий нас мир состоит из объектов, которые можно моделировать и эффективно использовать в программных решениях. Однако это не всегда оказывается столь очевидным.

Доказательства того, почему ООП не является таким эффективным и понятным, как может показаться на первый взгляд:

- Множество книг по Design Patterns, которые пытаются объяснить, как правильно использовать ООП, хотя оно должно быть интуитивно понятным. Количество таких книг говорит об обратном.
- ORM (Object-Relational Mapping), которое позволяет отображать объекты в концепциях баз данных. Если ООП настолько хорошая и удобная концепция, то не пришлось бы прибегать к таким сложным инструментам.
- Немного философии. Сам подход к моделированию окружающей нас реальности является не очень состоятельным (пример про изобретение Аристотеля — субстанция, которое работает плохо).

Несмотря на то, что большинство крупных систем сейчас создано при помощи ООП, это не доказывает его эффективности. ООП массового взлетело на языке C++, Java, а также на языках с динамической интерпретацией — Python, Ruby и подобных.

Из важного в C++ и Java появились встроенные инструменты полиморфизма, которые могут работать с данными, пришедшими из runtime. Также важная концепция, пришедшая в Java — это Garbage Collector, работающий в фоновом режиме. Он помог решить проблемы с ручным управлением памятью, что значительно облегчило жизнь программистам.

Небольшой вывод по эффективности ООП:

Современное ООП, скорее всего, не доказало свою эффективность напрямую, но функции, такие как полиморфизм и Garbage Collector, сильно популяризировали эту концепцию. Но так ли необходимо само ООП? Поэтому можно рассмотреть молодые языки программирования.

Молодые ЯП часто далеки от ООП.

Из активно развивающихся языков программирования можно рассмотреть Rust и Go. Оба языка не реализуют объектно-ориентированную парадигму. Они в большей

степени возвращают нас к классическому структурному программированию, но при этом сохраняют Garbage Collector и полиморфизм. Современные языки программирования и функциональный подход коррелируют, так как почти все из них реализуют анонимные лямбда-функции и функции высшего порядка (пример про Golang, в котором до появления дженериков реализация функции sort выглядела достаточно интересно).

Изменение вычислительных платформ:

- много ядер, параллелизм — компьютеров с одним ядром сейчас почти не найти. Современные компьютеры запускают свои программы параллельно, множество процессов и много ядер, так как задачи требуют процессов.
- облака, распределённые вычисления — концепции, в которых принципы работы с параллелизмом становятся еще более актуальными и строгими.

Рефакторинг, верификация, оптимизация, поддержка, автоматизация этих процессов.

И последний пункт, который также намекает на принципы функционального программирования. За счёт формальных свойств языков в них гораздо проще делать рефакторинг, верификацию, оптимизацию, поддержку кода и автоматизацию всех этих процессов.

2.2 Обзор функциональных языков программирования

Для выбора языка программирования можно опираться на следующие критерии:

- Динамические/Статические.
- Минималистичные/Богатые(обширные, сложные языки):

  Минималистичные языки поддерживают небольшое множество базовых инструментов и механизмов, на основе которых программист может строить сложные конструкции. Плюсом минималистичных языков является их простота, гибкость и легкость в изучении. Минусом — как раз из-за отсутствия широкого набора конструкций может потребоваться больше кода и более сложная логика в реализации продвинутых функций, что также повлечет за собой увеличение времени разработки. Одним из таких языков является Go.

  Обширные языки предлагают широкий спектр встроенных функций, синтаксических конструкций и возможностей. Примеров таких языков могут быть: C++, Rust. Однако, стоит заметить, что, несмотря на большие возможности, которые предоставляют обширные языки, пользоваться ими стоит с осторожностью, чтобы не перегружать код, делая его медленным при исполнении и сложным для чтения.

- Практические/Теоретические:

  В том или ином виде все языки, о которых пойдет речь дальше, являются практическими так как они использовались для решения каких-то определенных практических задач. Однако, есть языки, которые больше тяготеют к теоретической части. Что мы подразумеваем под теорией ? Теоретические языки — это языки, разработанные главным образом для исследований в области теории вычислений, формальных систем и математических основ программирования. Они служат инструментами для изучения новых концепций, парадигм и моделей вычислений. Такие языки могут не иметь развитых экосистем библиотек и инструментов для разработки реальных приложений.

  Хорошим примером практического языка является Erlang, а в качестве теоретических языков можно взять Agda. На стыке двух видов находится Haskell, который изначально был разработан как чисто теоретический функциональный язык

программирования, и благодаря своим возможностям как система типов, ленивые вычисления и чистота функций активно используется в академической среде. Со временем Haskell обрел развитую экосистему, включая множество библиотек и инструментов, что позволило использовать его в промышленности.

- Область применения.

2.2.1 Семейство языков Lisp

Lisp считается одним из первым языков программирования высокого уровня, написанный в 1958 году Джоном Маккарти. Первая реализация была завершена в MIT в 1960 году. Некоторые характеристик языка Lisp:

- Работая с Lisp можно достаточно редко встретить if-statements. Зачастую используются другие более выразительные и удобные конструкции для управления потоком выполнения. К примеру cond, when или unelss, которые выглядят следующим образом:

```
(cond ((condition1) expr1)
      ((condition2) expr2)
      (t default-expr))
```

```
(when (condition)
  expr1
  expr2
  expr3)
```

```
(unless (condition)
  expr1
  expr2)
```

- Поддержка функций первого класса. Функции первого класса означают, что функции в языке программирования могут быть обработаны как любые другие данные: в качестве аргументов, возвращены из другой функции и присвоены переменным.

  Пример программы вычисления квадрата числа, в которой мы передаем функцию как аргумент в другой функции:

```
(defun apply-function (f x)
    (funcall f x))

(let ((result (apply-function #'(lambda (y) (* y y)) 5)))
    (format t "~a" result)) ; => 25
```

- Поддержка рекурсии.
- Динамическая типизация.
- Наличие сборщика мусора.
- В Lisp нет разделения между выражениями (expressions) и инструкциями (statements). Все конструкции в Lisp являются выражениями, то есть функции, макросы и так далее всегда возвращают какой-то результат, который можно использовать в дальнейших вычислениях отчего сама структура кода может сильно меняться и быть непривычной.
- Символы (тип данных symbol) и строки (тип данных string) — это разные типы. Символы — атомарный объект, используемый главным образом как идентификатор: имя переменной, функции, ключевое слово и так далее. Символы неизменяемые

и сохраняются в глобальной таблице символов. Название символа уникально, а символы с одним и тем же названием будут указывать на одну и ту же область в памяти. Чтобы удостовериться, посмотрим на вывод этой программы:

```
(eq 'hello 'hello)   ; => T
```

- Строки — простая последовательность символов для представления текстовых данных. Строки в отличие от символов могут быть как изменяемыми, так и неизменяемыми в зависимости от реализации Lisp.

  ```
  (eq "foo" "foo")     ; => NIL
  (equal "foo" "foo")  ; => T
  ```

  Во второй строчке возвращается T, потому что мы сравниваем содержимое, а не ссылки.

- Гомоиконичность. Гомоиконичность говорит о том, что программы, написанные на Lisp, представлены обычной структурой данных, присуще Lisp, а именно списком. Это позволяет достаточно просто манипулировать программным кодом, так как его можно обрабатывать, изменять и генерировать, как обычные данные. К тому же такое свойство как гомоиконичность делает Lisp идеальным языком для метапрограммирования и создания макросов.

- Все возможности языка можно использовать во время загрузки, компиляции и выполнения кода.

  - Разработчик имеет полный доступ к полному инструментарию языка на каждом из этапов.
  - Во время загрузки программы можно выполнять произвольный Lisp-код. Можно определять функции, изменять глобальные переменные, выполнять вычисления. Это позволяет настраивать окружение или подготавливать данные непосредственно во время загрузки программы.
  - Во время компиляции также доступны все возможности языка. Макросы используют эту возможность: они позволяют генерировать и преобразовывать код во время компиляции, создавая эффективный машинный код либо новые функции.
  - Во время выполнения программа может динамически создавать и выполнять код, изменять свое поведение на основе входных данных или окружения.
  - Это также одна из особенностей языка, которая делает его мощным инструментом для создания DSL и средств автоматизации.

Итого: Lisp предоставляет непрерывную и унифицированную среду, где программный код является данными, что упрощает реализацию сложных задач вроде адаптивного и метапрограммирования.

2.2.1.1 Scheme

Один из диалекторв языка Lisp, созданный в 1975 году Гаем Стилом и Джеральдом Сусманом. Scheme разрабатывался как крайней упрощенный и более чистый вариант Lisp, фокусирующийся на минимализме и теоретической строгости.

Базовые конструкции языка основаны на лямбда-исчислении, что позволяет создать язык с минималистичным и компактным ядром. Это достигается за счёт включения в язык лишь небольшого числа фундаментальных конструкций, необходимых для выражения вычислений, аналогичных тем, что определяются в лямбда-исчислении.

На данный момент самая популярная реализация Scheme — Racket

Несколько особенностей языка Scheme:

- В основном Scheme и его реализации используются в образовательных целях для обучения базовым понтиями функционального программирования и теории вычислений.
- Scheme имеет несколько стандартов (например, R5RS, R6RS, R7RS), каждый из которых определяет различные аспекты языка, обеспечивая гибкость и адаптивность.
- В традиционных стандартах языка Scheme таких, как R5RS, используется единое пространство имен для функций и переменных. Это означает, что одно и то же имя не может одновременно представлять как функцию, так и переменную. Однако, начиная с более современных стандартов, таких как R6RS и R7RS, в Scheme были введены системы модулей, которые позволяют создавать раздельные пространства имен внутри модулей.
- Scheme обязательно оптимизирует хвостовую рекурсию. То есть нет необходимости где-то отдельно указывать, что мы хотим использовать оптимизацию хвостовой рекурсии.
- Из-за минимализма языка многие общие процедуры и синтаксические формы не определены в стандарте, поэтому в сообществе Scheme принят процесс “Scheme Request for Implemnetation” или SRFI. Многие SRFI (более 60 штук) поддерживаются всеми или большинство реализаций Scheme.

2.2.1.2 Common Lisp Мощный и универсальный диалект языка Lisp, стандартизированный ANSI (American natioanl standards institute) в 1994 году. Язык появился в 80-х годах, как попытка объединить многочисленные разрозненные реализации концепций Lisp и предоставления богатого набора функций для разработки сложных и производительных приложений. На практике, вероятно всего, на сегодняшний момент данный диалект является наиболее используемым на практике.

Особенности и характеристики Common Lisp:

- Мультипарадигменность:

  Подразумевает возможность в рамках одного языка использовать и безболезненно сочетать разные стили и подходы — например, ООП и функциональный.

- Динамичность:
  - Динамическая типизация с возможностью явного объявления типов.
  - Представление программы в виде живого образа (live image), в котором любой объект, за исключением 25 базовых операторов языка, может быть переопределен в ходе исполнения теми же средствами, что и в момент его создания.
  - Поддержка полноценных возможностей интроспекции.
- Использование раздельных пространств имен для переменных, функций, типов и так далее.
- Common Lisp Object System.

  Мощная и гибкая система ООП, встроенная в Common Lisp, которая достаточно сильно отличается от традиционного ООП.

  Немного свойств CLOS:
  - Множественное наследование.
  - Дженерик (generic) функции.

    * В CLOS поведение описывается через дженерик функции, объединяющиеся в связанные методы. Эти функции не принадлежать конкретному классу, а определяются отдельно и работают с разными типами объектов. К примеру: В классическом ООП метод speak будет частью класса Dog. В CLOS speak — дженерик функция, которая включает методы для Dog, Cat, и других типов.
  - Множественная диспетчеризация.
    * Выбор метода зависит от всех типов аргументов, а не только от первого (как в традиционном ООП).
  - Динамичность.
    * Классы, объекты и методы можно изменять во время исполнения, что дает CLOS невероятную гибкость.
  - Комбинация методов.
    * CLOS предоставляет механизм комбинирования методов, который определяет порядок вызова методов для различных классов и типов аргументов (через ключевые слова :before, :after, :around).

- Conditioanl System — система работы с исключениями.

  Альтернативный гибкий и мощный механизм для обработки исключений, отличающийся от традиционной системы `try-catch` в других языках программирования.

  Основные компоненты Condition System:

  - Условия (conditions). Условия представляют собой объекты, описывающие различные ситуации, возникающие во время выполнения программы, такие как ошибки, предупреждения или информационные сообщения. Условия создаются с помощью функций `signal`, `error`, `warn`.
  - Обработчики (handlers). Обработчики определяют, как программа должна реагировать на сигнализируемые условия. Они устанавливаются с использованием макроса `handler-bind`. Обработчики могут иметь также различные уровни приоритета, определяющие порядок срабатывания.
  - Перезапуски (restarters). Перезапуски предоставляют механизмы для восстановления программы после возникновения условий. Они позволяют определить точки, в которых программа может продолжить выполнение после обработки условия. Перезапуски создаются с помощью макроса `restart-bind`.

  Такой подход позволяет отделить обработку ошибок от основного кода и поддержать гибкое восстановление после ошибок без необходимости перезапуска программы.

Интересный факт: грамматическое ядро популярного сервиса Grammarly написано на Common Lisp (Running Lisp in production).

2.2.1.3 Clojure Lisp-подобный язык общего назначения, разработанный для JVM, вышедший в 2007 году. Автор языка Ричард Хикки. Несмотря на некоторую схожесть Clojure с Common Lisp и Scheme, он на 100 процентов не совместим ни с одним из них, но позаимствовал многие идеи из этих языков, добавив новые вещи, о которых речь пойдет чуть дальше.

Особенности Clojure:

- Неизменяемые структуры данных. Clojure максимально пытается заставить разработчика не использовать изменяемое состояние, поэтому большинство структур данных неизменяемые по умолчанию.
- Clojure полностью совместим с существующими библиотеками и фреймворками Java, а также с системы сборки (Maven, Gradle) и IDE. Пример использования Spring в Clojure.

```
(import 'org.springframework.context.support ClassPathXmlApplicationContext)

(def ctx (ClassPathXmlApplicationContext. "applicationContext.xml"))
(def bean (.getBean ctx "myBean"))
```

- От Lisp'а Clojure «унаследовал» макросы, мультиметоды и интерактивный стиль разработки, а JVM дает переносимость и доступ к большому набору библиотек, созданных для этой платформы.
- Поддержка ленивых коллекций.
- Clojure хорошо подходит для создания многопоточных приложений.
  - Благодаря комбинации неизменяемых структур данных, мощной системы управления состоянием через STM (Software Transactioanl Memory), высокоуровневых абстракций для параллелизма и интеграции с JVM, Clojure предоставляет разработчикам мощные инструменты для создания эффективных и безопасных многопоточных приложений.
  - Благодаря STM в Clojure все-таки можно изменять состояние, если это правда необходимо.

На официальном сайте есть полный список отличий Common Lisp от Clojure, Differnces with other Lisps.

Также есть `Clojure Script` — это диалект языка программирования Clojure, который компилируется в JavaScript. Он позволяет разработчикам использовать функциональные возможности Clojure для создания веб-приложений и других проектов, работающих в среде JavaScript.

Почему столько реализаций диалектов Lisp? Это можно объяснить старостью языка. Раньше, когда компиляторы были платные, университеты или независимые лица писали свои реализации языка, брав вдохновение из конференций или подобных мероприятий.

Scheme, Common Lisp и Clojure, несмотря на то, что мы рассматривали их внутри одной категории Lisp-языков, на самом деле очень разные как на области применения, так и в своих подходах к программированию, особенностях использования и философиях развития.

2.2.2 Erlang/OTP, Elixir

Рассмотрим другую группу динамических языков Erlang и его переиздание Elixir.

Erlang — язык программирования, созданный в 1986 году в Ericsson для разработки отказоустойчивых, масштабируемых и распределённых систем реального времени. Проект начался как исследовательская работа по улучшению телекоммуникаций, а в 1998 году язык стал открытым.

Для создания телекоммуникационных систем, где критически важны надёжность и отказоустойчивость, использовались языки Prolog и C++. Prolog хорошо подходил для написания бизнес-логики, а C++ позволял работать с пакетами, протоколами и другими низкоуровневыми компонентами. Однако за 8 лет разработки проект на этих языках не

был доведён до рабочего состояния. Позже был создан более практичный Erlang/OTP со своей виртуальной машиной, который идеально подходит для решения задачи создания надёжной телекоммуникационной системы, позволяя создавать системы с доступностью 99.9999999% (3 секунды простоя в год). К тому же синтаксис Prolog невероятно сильно напоминает синтаксис Erlang.

Особенности и характеристики Erlang:

- Можно отнести к императивным языкам.
- Динамический функциональный язык программирования. Erlang обязан быть динамическим языком. Иначе без динамичности не получилось бы писать отказоустойчивые и надежные программы, которые во время исполнения могли бы меняться и восстанавлиться в рабочее состояние при возникновении непредвиденных ситуаций.
- Поддержка парадигмы акторной модели.

  Акторная модель — подход к параллельному программированию, где основным строительным блоком являются акторы, обменивающиеся сообщениями между собой. Акторы — это независимые сущности, которые:

  - Обладают собственным состоянием.
  - Обрабатывают сообщения, приходящие извне.
  - Запускаются и работают параллельно.
  - Из-за изолированности каждого процесса у нас пропадает проблема с синхронизацией так как происходит копирование данных из процесса в процесс. Да, это может давать накладные расходы по памяти, однако пропадает еще большая проблема с синхронизацией, гонками и так далее.
- Философия “let it crash” и супервизоры.
  - Концепция “let it crash” подразумевает вместо того, чтобы обрабатывать все возможные ошибки внутри каждого процесса, Erlang поощряет разработку процессов, которые могут завершиться при ошибке, в то время как супервизоры запускают новые процессы взамен упавших.
  - Структуры, которые следят за работой процессов и управляют их перезапуском при сбоях, обеспечивая надёжности системы. Обычно их представляют в виде дерева супервизоров и в итоге приложение на Erlang может выглядеть следующим образом.
- Горячая замена кода (hot code loading and modules).
  - Каждый модуль в Erlang может иметь до двух версий, текущая и предыдущая. Когда загружается новая версия кода во время исполнения, текущая версия помечается как предыдущая, а новая становится текущей. Процессы будут продолжать выполнять старую версию кода, пока не вызовут функцию из нового загруженного модуля.
- Виртуальная машина BEAM.
  - Оптимизирована для обработки тысяч независимых процессов, которые невероятно легковесны.
  - Обеспечение кроссплатформенность.
- OTP (Open Telecom Platform).

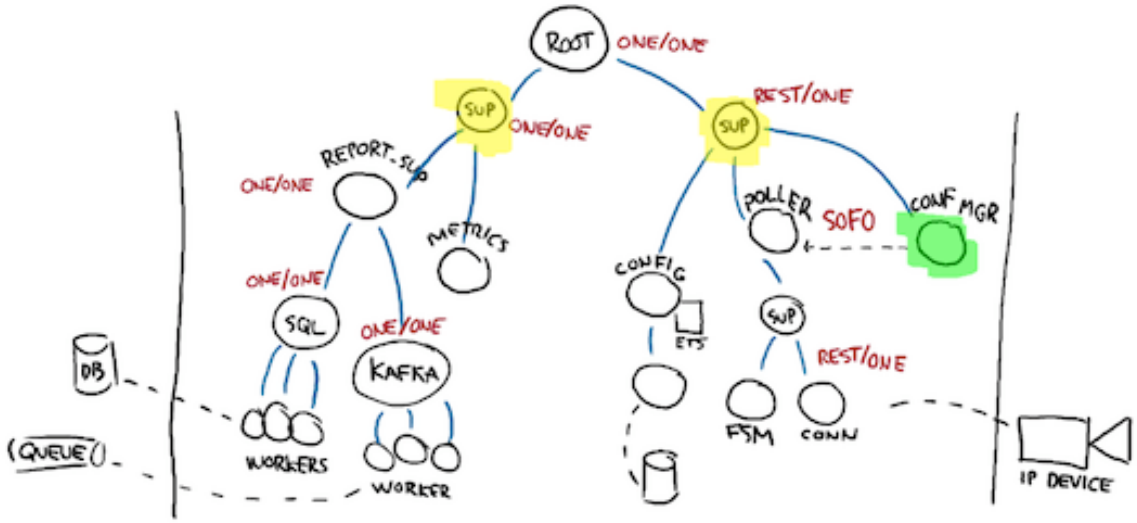


Figure 1: erlang_supervisor_tree

  - Набор библиотек и инструментов, предоставляющий готовые решения для распространённых задач, таких как управление процессами, построение сетевых приложений, хранение состояний и многое другое. OTP обеспечивает стандартизацию и упрощает разработку сложных приложений. Предоставляет менее гибкую, но более шаблонную организацию приложения и простую работу с процессами.

Elixir — современный функциональный язык, созданный в 2011 году Хосе Вальимом. Он построен на виртуальной машине Erlang (BEAM), что обеспечивает ему все преимущества экосистемы Erlang, включая акторную модель, горячую замену кода, легковесные процессы и отказоустойчивость.

Основные особенности Elixir:

- Синтаксис Elixir вдохновлен Ruby.
- Полная совместимость с Erlang и библиотеками OTP.
- Метапрограммирование.
  - Благодаря макросам, Elixir можно легко расширять, добавляя новые функции и конструкции без изменения ядра языка. Это способствует созданию внутренних DSL (domain-specific languages), адаптированных под специфические задачи проекта.
- Полиморфизм через протоколы
  - Протоколы в Elixir —- это механизм, позволяющий определять набор функций, которые могут быть реализованы для различных типов данных. Они обеспечивают ад-хок полиморфизм, позволяя разработчикам расширять функциональность без изменения существующих типов или структур. Рассмотрим пример:

    Определение протокола:

    ```
    defprotocol Describable do
      @doc "Возвращает описание объекта"
    ```

```
    def describe(data)
end
```

Реализация протокола для конкретных типов данных:

```
defimpl Describable, for: String do
  def describe(data), do: "Строка: #{data}"
end

defimpl Describable, for: Integer do
  def describe(data), do: "Целое число: #{data}"
end
```

Использование протокола:

```
Describable.describe("Hello")            # => "Строка: Hello"
Describable.describe(42)                 # => "Целое число: 42"
```

2.2.3 Ocaml

Ocaml (Objective Caml) — функциональный язык программирования общего назначения, разработанный в 1996 году в институте INRIA во Франции как часть семейства языков ML (Meta Language). OCaml сочетает в себе функциональные, императивные и объектно-ориентированные парадигмы, предоставляя средства для разработки надёжных и эффективных программ.

Особенности Ocaml:

- Компилируемый и высокопроизводительный.
  - Ocaml использует компилятор, который генерирует оптимизированный машинный кода, схожий с языком C.
  - Благодаря системе типов и проверки типов во время компиляции получается выявлять и устранять лишние операции.
  - Ocaml использует гибридный сборщик мусора с поколениями (Hybrid generational Garbage Collector).
- Статический.
  - Типы проверяются на этапе компиляции, что предотвращает множество ошибок и повышает надёжность программ.
  - Компилятор способен автоматически выводить типы выражений, что снижает необходимость явного объявления типов и делает код более компактным.
- Поддержка ООП.
  - Несмотря на функциональный фундамент, OCaml поддерживает изменяемые данные для случаев, когда это необходимо.
  - Возможность создания классов и объектов, поддержка инкапсуляции и наследования, что расширяет возможности языка для решения различных задач.
- Неизменяемость данных, функции высшего порядка, сопоставление с образцом (pattern mathcing).
  - Сопоставление с образцом — механизм, используемый в программировании для проверки структуры данных и извлечения информации, соответствующей

определённому шаблону. Это похоже на проверку, подходит ли объект под заранее определённый формат.

- Алгебраические типы данных (ADT).
  - Механизм для определения пользовательских типов, объединяющий в себе возможности суммовых типов (вариантов) и записей. Они позволяют создавать сложные структуры данных, обеспечивая при этом безопасность типов и удобство работы с различными вариантами данных.

```
type shape =
  | Circle of float                    (* Радиус *)
  | Rectangle of float * float         (* Ширина и высота *)
  | Triangle of float * float * float  (* Стороны *)
```

- Параметризованные модули.
  - Параметризованные модули, также известные как модули с аргументами или функторы, позволяют создавать модули, которые принимают другие модули в качестве входных параметров. Такая функциональность очень напоминает шаблоны в C++, дженерики в Java либо инъекцию зависимостей (dependency injection) в ОПП языках.

2.2.4 F-sharp

F# (F-sharp) — это современный функциональный язык программирования, разработанный Microsoft как часть экосистемы .NET. Представленный в 2005 году, F# сочетает в себе мощь функционального программирования с гибкостью объектно-ориентированных и императивных парадигм.

Особенности F-sharp:

- Строгая статическая типизация с выводом типов (type inference).
  - Вывод типов означает то, что компилятор автоматически определяет типы выражений, что позволяет писать более компактный код без явных аннотаций типов.
- Полный доступ к библиотекам и экосистеме .NET. Совместимость с языками C# и VB.NET.
- Поддержка ООП.
- Параллелизм и асинхронность.
  - Асинхронные рабочие процессы в F# предоставляют удобный способ написания асинхронного кода, который выглядит как синхронный. Это достигается с помощью специальных синтаксических конструкций, которые позволяют управлять асинхронными операциями без необходимости использования колбеков или явного управления потоками.
  - F-sharp полностью совместим с .NET, что позволяет использовать возможности Task Parallel Library для управления параллельными задачами. Это включает создание, запуск и управление задачами, а также синхронизацию между ними.
- Программирование с использованием Агентов.
  - Парадигма, основанная на использовании агентов для управления состоянием и асинхронными задачами в многозадачных приложениях. В F-sharp реализовано через тип MailboxProcessor, который предоставляет простой и мощный способ создания агентов для управления состоянием и обработки сообщений.

2.2.5 Haskell

Haskell — чисто функциональный язык программирования общего назначения, известный своей выразительной и мощной системой типов, строгой типизацией. Он был разработан в 1980-х годах с целью создания стандарта для функциональных языков и широко используется для академических исследований и индустриальных приложений, где важны математическая строгость и безопасность типов, а также на данный момент в реальных практических задачах. В Haskell функции первого класса, поддерживаются монады, и используется система типов, обеспечивающая надежность и параллелизм. Ленивость позволяет эффективно работать с бесконечными структурами данных и отложенной вычислительной моделью.

Особенности Haskell:

- Чистота функций.
  - В Haskell функции являются чистыми, то есть они не имеют побочных эффектов и всегда возвращают один и тот же результат для одних и тех же аргументов. Это облегчает анализ и тестирование кода.
- Вывод типов (type infernce).
- Ленивые вычисления.
  - Haskell использует стратегию ленивых вычислений, при которой выражения вычисляются только тогда, когда их значения действительно необходимы. Это позволяет создавать бесконечные структуры данных и оптимизировать производительность за счет избежания ненужных вычислений.
- Конкурентность.
  - Haskell использует легковесные потоки, управляемые рантаймом GHC, что позволяет создавать тысячи потоков без значительных накладных расходов.
  - Наличие STM упрощает управление состоянием в многопоточной среде, обеспечивая атомарные транзакции.
  - Поддержка реализации акторной модели обмена сообщениями с помощью `Chan` и асинхронные вычисления.
- Сильная статическая типизация.
  - В Haskell используется система типов с параметризацией (generic types), полиморфизмом, типами классов (type classes) и выводом типов (type inference). Система типов Haskell позволяет комбинировать различные абстракции, не теряя при этом строгой типовой безопасности.
- Строгий компилятор и структурированность кода.
- Очень богатая библиотека.

  К примеру:
  - `aeson` позволяет просто и эффективно сериализовать и десериализовать данные в JSON.
  - Парсер-комбинатор `megaparsec` — мощный текстовый пасрер-комбинатор для разбора сложных текстовых форматов.
  - Веб-разработка с помощью `servant` или `yesod`.

  И многое другое
- Активное развитие языка.

- Из-за активного развития появляется такая ситуация, что при использовании разных расширений мы можем получать немного разные реализации языка Haskell.

2.2.6 Coq, Gallina

Coq — не язык программирования, а больше интерактивная система доказательства теорем, основанная на языке спецификаций и формальной верификации программ. Она используется для разработки математических доказательств и проверки корректности программного обеспечения.

Gallina — это основной язык спецификаций в Coq. Он используется для определения типов, функций, лемм и теорем. Gallina основана на исчислении конструкций индуктивных типов (Calculus of Inductive Constructions) и поддерживает как функциональное программирование, так и формальную верификацию.

Эти системы уже относятся к более теоретической части, чем предыдущие рассмотренные языки.

Основные особенности:

- Предоставляет средства для выражения математических утверждений и спецификаций программного обеспечения. Это позволяет четко определить требования к программам и теоремы, которые необходимо доказать.
- Позволяет разработчикам пошагово разрабатывать формальные доказательства для заданных теорем. Интерактивный подход облегчает процесс доказательства, предоставляя гибкие инструменты для исследования различных путей доказательства.
- Автоматически проверяет корректность доказательств с помощью относительно небольшого “ядра” сертификации.
- Позволяет извлекать сертифицированные программы на языки программирования, такие как OCaml, Haskell или Scheme. Это обеспечивает механизм перехода от формальных спецификаций и доказательств к реальным программам, сохраняя их корректность.

К примеру на Coq можно реализовать красно-чёрное дерево и формально доказать, что дерево всегда правильно раскрашено, сбалансировано с разницей не более чем плюс/минус один.

Простой пример доказательства теоремы на Coq:

```
Theorem example1 : forall A B : Prop, A ∧ B → B ∀ B.
Proof.
  intros A B H.
  destruct H as [_ HB].
  right. exact HB.
Qed.
```

2.2.7 Agda, Idris

Agda и Idris — функциональные языки программирования с зависимыми типами. В отличие от классических языков программирования, где типы не зависят от значений данных, языки с зависимыми типами позволяют типам зависеть от значений. Это позволяет создавать более точные и специфичные типы, выражающие свойства и ограничения программ на уровне типов, что повышает надежность и безопасность программ.

Рассмотрим каждый из языков более подробно:

2.2.7.1 Agda Agda — это функциональный язык программирования, который объединяет программирование и доказательство теорем. Благодаря мощной системе типов, Agda позволяет выражать свойства программ в самих типах, что обеспечивает проверку корректности на этапе компиляции.

Например, рассмотрим определение списка и функцию вычисления его длины:

```
module ListLength where

open import Agda.Builtin.Nat

-- Определение типа списка
data List (A : Set) : Set where
    []   : List A                    -- Пустой список
    _::_ : A → List A → List A       -- Конструктор, добавляющий элемент в список

-- Функция вычисления длины списка
length : ∀ {A} → List A → Nat
length []        = zero              -- Длина пустого списка равна нулю
length (_ :: xs) = suc (length xs)   -- Длина списка с элементом в голове равна 1 + длина
```

В этом примере определяется тип списка `List`, после чего реализуется рекурсивная функция `length`, вычисляющая его длину. Система типов Agda гарантирует корректность этой функции, проверяя соответствие типов во время компиляции.

2.2.7.2 Idris Idris — это современный функциональный язык программирования, который сочетает практичность и мощные возможности типизации. Он стремится быть языком общего назначения, предоставляя удобные средства для разработки реальных приложений.

Особенности Idris:

- Синтаксис, похожий на Haskell, что облегчает изучение языка для разработчиков, знакомых с функциональным программированием.
- Вывод типов и поддержка интерфейсов (type classes), позволяющие писать обобщённый и переиспользуемый код.
- Интерактивная среда разработки с поддержкой REPL (Read-Eval-Print Loop), что упрощает экспериментирование и прототипирование.
- Возможность взаимодействия с внешним миром через ввод-вывод, работу с файлами и сетью.

Пример программы на Idris с использованием зависимых типов:

```
module Main

-- Определение типа вектора с длиной, закодированной в типе
data Vect : Nat → Type → Type where
    Nil  : Vect 0 a
    (::) : a → Vect n a → Vect (S n) a

-- Функция, которая конкатенирует два вектора
concat : Vect n a → Vect m a → Vect (n + m) a
concat Nil       ys = ys
concat (x :: xs) ys = x :: concat xs ys
```

```
-- Пример использования функции concat
example : Vect 3 Int
example = concat (1 :: 2 :: 3 :: Nil) (4 :: 5 :: Nil)
```

В этом примере определяется тип Vect, который представляет собой вектор с длиной, закодированной в типе. Функция concat конкатенирует два вектора, и тип результата гарантирует, что его длина будет равна сумме длин входных векторов. Это демонстрирует, как зависимые типы могут использоваться для обеспечения дополнительных гарантий на этапе компиляции.

2.3 Идея ФП

2.3.1 Типичный нефункциональный язык

```
procedure inc(var m: integer);
begin
    m := m + 1;
end;

var x: integer = 0;
begin
  inc(x);
  writeln ('Result:', x);
end.
```

На данном примере представлен код на языке Pascal. В процедуру передается ссылка на переменную, внутри которой происходит её инкрементация. Затем переменная объявляется и вызывается процедура, после чего соответствующий результат выводится в консоль. Это пример классического процедурного программирования, где каждая процедура работает с общей глобальной памятью и может её изменять или модифицировать.

2.3.2 Функциональный язык

```
#include<stdio.h>

int inc(int m) {
    return m + 1;
}

int main(void) {
    int x = 0;
    x = inc(x);
    printf("Result:%i", x);
}
```

Далее переходим к функциональным языкам. На данном примере представлен язык программирования C. Он не является исключительно процедурным, а обладает элементами функционального программирования. Сравнивая с примером на Паскале, можно заметить, что в функции инкремента значение передается не по ссылке, а через автоматическую память. На стеке создается область памяти для значения переменной, затем вычисляется сумма данного значения с единицей, и результат возвращается через стек в главную функцию.

Таким образом, язык перестает быть процедурным в том смысле, что он не изменяет внешнее состояние в памяти напрямую. Вместо этого он возвращает новое значение, сохраняя неизменность данных (immutability).

2.4 От процедур к функциям

Функциональное программирование представляет собой переход от процедурного программирования к функциональному. В нём функции рассматриваются как математические объекты. Это означает, что функции не могут изменяться или переопределяться. При передаче значения функция может лишь создать на его основе что-то новое и вернуть результат, но не изменять исходное значение

2.4.1 Процедуры

Рассмотрим классическое определение процедур: процедуры — это подпрограммы, которые вместо возврата одного значения, позволяют получить группу результатов.

Nota Bene: Под “results” обычно подразумеваются побочные эффекты:

- запись в память/чтение из памяти.
- ввод/вывод.

Таким образом, группа результатов может включать запись в память или чтение из неё, а также операции ввода/вывода.

Это неудобно тем, что по функции невозможно предсказать, к каким изменениям она может привести.

2.4.2 Функции (в математике)

В чисто функциональном стиле, в математике, все немного иначе. Мы можем представить любую функцию, как отображение значения области определения (на примере это X) в область значений (на примере это Y).

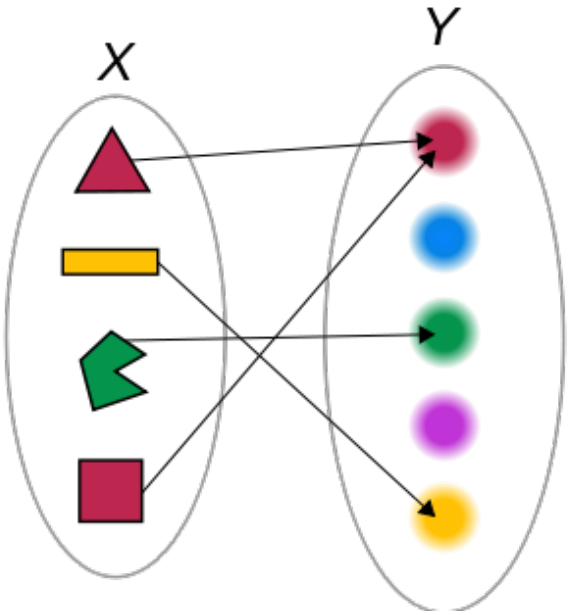


Figure 2: function-in-math

Соответственно мы получаем главные свойства, заложенные в функциях:

- вход однозначно определяет выход (независимо от количества запусков функции с одним и тем же входным значением, результат всегда будет одинаковым).
- отсутствие побочных эффектов (функция изменяет только те данные, которые подаются на вход, и никакие другие — это так называемые чистые функции).
- невыразимость ввода-вывода (операции ввода-вывода являются побочными эффектами, которые изменяют внешнее состояние, что противоречит принципам чистых функций).

2.4.3 Функциональное программирование стремится к математическому пониманию функции

Поскольку функциональное программирование является практической областью, а не теоретической, то оно стремится привести программирование к такому виду, которое напоминало бы математику. Соответственно мы приходим к следующему:

- **чистые функции с точки зрения пользователей**;

  Чистая функция гарантирует, что при одном и том же входе всегда будет один и тот же выход. Она не изменяет глобальные переменные, файлы или что-либо вне своего локального контекста.

  Для пользователя это значит, что функция предсказуема: её поведение полностью определяется входными данными.

- **чистые функции с точки зрения реализации** (позволяют понять, является ли функция изменяющей внешнее окружение или нет);

  Реализация должна быть такой, чтобы программист или анализатор кода мог с уверенностью сказать, что функция не производит побочных эффектов.

- **функции оперируют математическими объектами, а не объектами компьютера**:

  Ссылочная прозрачность (это свойство означает, что выражение можно заменить его значением без изменения поведения программы, что характерно для чистых функций).

  Структуры данных пересобираются, а не изменяются (в функциональном программировании структуры данных являются неизменяемыми, что делает параллельное программирование достаточно простым).

  Функция — математический объект (функции можно передавать как данные: передавать их в качестве аргументов другим функциям, возвращать как результаты или хранить в структурах данных);

- **программа — символьная конструкция** (адресовано к инструментарию компилятора).

  Стремиться $\neq$ являться. Это подчёркивает, что функциональные языки стремятся приблизить программирование к математике, но не являются чистой математикой.

  Например, реальный мир требует работы с вводом-выводом, который нарушает чистоту функций (потому что ввод-вывод изменяет состояние мира). Поэтому используются специальные конструкции, как монады (например, IO в Haskell), чтобы отделить нечистые вычисления от чистых.

Примеры стековых языков программирования, таких как Forth:

- **PostScript** (язык описания страниц, используемый в графических и печатных системах, разработанный Adobe. Основан на стековой модели).

- **Factor** (современный стековый язык программирования с мощными библиотеками, поддержкой объектов и расширенными функциями).
- **ColorForth** (упрощённая и улучшенная версия Forth, разработанная самим Чарльзом Муром. Отличается минималистичной синтаксической моделью, где цвет кода имеет значение).
- **Cat** (функциональный стековый язык программирования, вдохновлённый Haskell).

2.5 Итоги перехода к функциональному программированию из классического программирования

Преимущества:

- локализация проблем, тестирование, отладка (в сложных продуктах легче находить узкие места и проблемы, так как функции всегда явно показывают свои действия и результаты).
- относительная простота параллельного программирования (например, в Erlang благодаря OTP и gen servers каждая задача изолирована в своём процессе, что снижает риск ошибок, связанных с доступом к общим данным).
- больше формальных свойств (возможность анализа и оптимизации вычислений благодаря строгим правилам порядка выполнения и свойствам кода).
- функции высшего порядка (модульность, конфигурируемость), этот подход делает программы более модульными и конфигурируемыми. Пример: parser combinators, которые позволяют эффективно создавать трансляторы, используя комбинации функций высшего порядка.

Недостатки:

- не позволяет программировать алгоритмы и структуры “в лоб” (многие классические алгоритмы, зависящие от изменения состояния, не подходят для прямой реализации в функциональном стиле).
- многие задачи требуют изменяемого состояния (множество реальных задач требует управления изменяющимся состоянием, что противоречит принципам функционального подхода и усложняет поиск элегантного решения).
- вычислительный контекст приходится таскать самому (нельзя использовать глобальные переменные или синглтоны для хранения данных. Контекст нужно явно передавать через параметры или использовать сложные механизмы, такие как монады).
- трудности ввода-вывода и “нелинейность” процессов (ввод-вывод часто связан с изменением состояния, что вызывает проблемы в языках, строго следующих функциональной парадигме).
- на пустом месте: “И как сюда printf воткнуть?”(в функциональном программировании добавление стандартного вызова `printf` или аналогичных функций вывода может быть сложным из-за строгих принципов неизменяемости и отсутствия побочных эффектов, свойственных этой парадигме).
- программисты с математическим скалдом ума (программисты с математическим складом ума могут чрезмерно увлекаться абстрагированием, что усложняет понимание и поддержку кода).

3 Основы ФП. Рекурсия, структуры

В данном разделе мы рассмотрим основы функционального программирования, основные особенности представления данных и отличия от традиционных языков программирования, а также постараемся ответить на вопрос, чем обусловлены эти отличия.

Функциональное программирование основывается на идее использования языка программирования для разбиения кода на отдельные части, называемые **функциями**, которые рассматриваются как **математические функции**.

Функция в математическом смысле — это правило, которое связывает каждое значение из одного множества с ровно одним значением из другого множества.

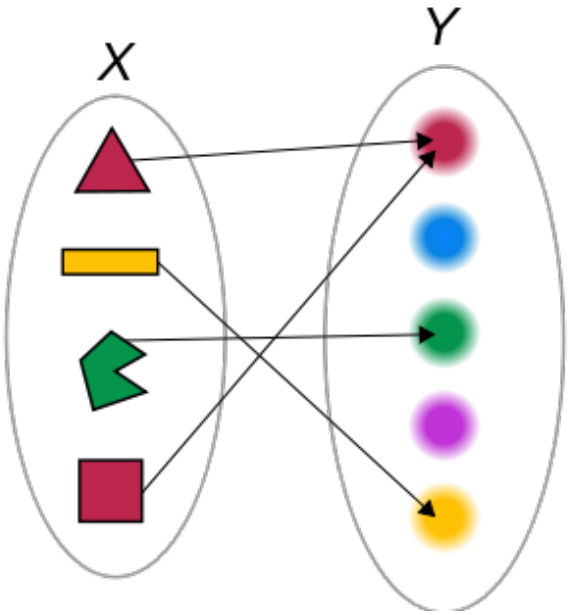


Figure 3: Function-In-Math

3.1 Чистые и грязные функции

Несмотря на то, что мы стараемся рассматривать наши программные функции как математические, но для нас важно понимать то, что программные функции могут зависеть/изменять внешнее состояние, в таком случае принято говорить, что функция имеет **побочные эффекты(side-effects)**.

Функции, имеющие побочные эффекты принято называть **грязными**, по той причине, что они лишены однозначности результатов, в то время как однозначно заданные функции, не имеющие побочных эффектов, принято называть **чистыми**.

Мы стремимся писать чистые функции, потому что тем самым мы получаем предсказуемое поведение и лёгкую отладку.

Пример грязной функции инкремента:

```
package main
```

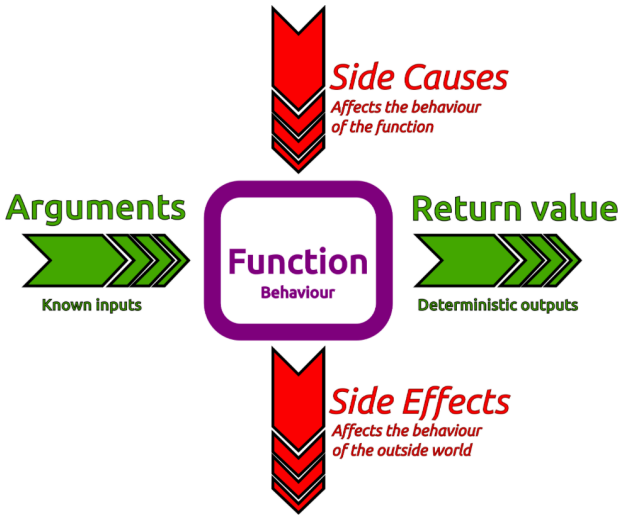


Figure 4: Function-Side-Effects

```
import "fmt"

func inc(m *int) {
    *m = *m + 1
}

func main() {
    x := 0
    inc(&x)
    fmt.Println("Result: ", x)
}
```

Функция `inc` является грязной, потому что она меняет внешнее состояние — передаваемую переменную, которая инкриминируется по указателю.

Пример чистой функции инкремента:

```
package main

import "fmt"

func inc(m int) int {
    m = m + 1
}

func main() {
    x1 := 0
    x2 := inc(x)
    fmt.Println("Result: ", x2)
}
```

Отдельно стоит рассмотреть функцию `main`, которая с виду кажется чистой, но при этом является грязной, так как в ней присутствует вызов `fmt.Println`, который выводит данные в консоль, то есть результат функции зависит от внешней среды.

3.2 Циклы и рекурсии

Задачи, связанные с обработкой последовательностей, коллекций, массивов и списков, а также задачи вычислительной математики, часто требуют применения циклов(итеративной обработки данных). Однако функциональный стиль программирования не предполагает использование циклов в традиционном понимании, где применяется аккумулятор, изменяющийся с каждой итерацией.

В условиях функционального программирования альтернативой циклам является **рекурсия**. Это процесс, при котором функция вызывает саму себя для выполнения определенных действий. **Рекурсия** позволяет итеративно обрабатывать данные без необходимости явного использования циклов и изменения состояния переменных.

Одна из математических функций, легко реализуемых через рекурсию — функция факториала:

$$fac(n) = \begin{cases} 1, & if: n = 0 \\ n * fac(n-1) \end{cases}$$

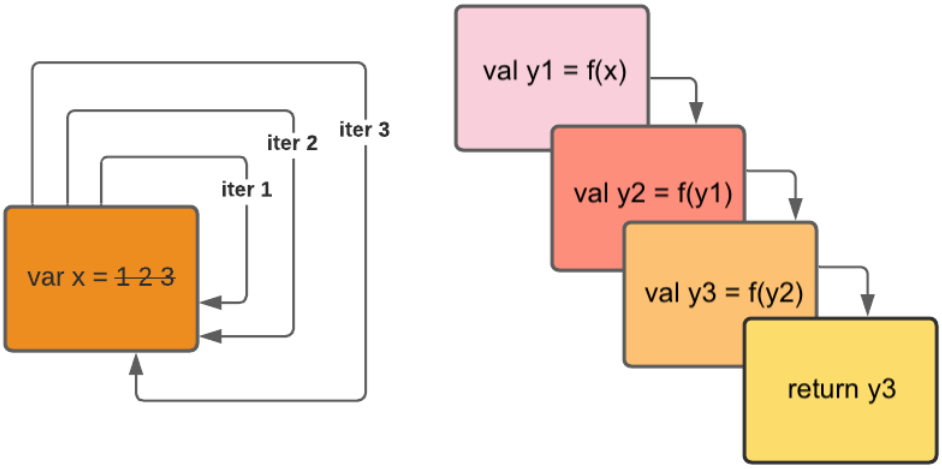


Figure 5: Loops-Vs-Recursion

Пример факториала через цикл:

```
def fac(n):
    assert n >= 0
    acc = 1
    for e in range(n, 1, -1):
        acc *= e
    return acc
```

Пример факториала через рекурсию:

Традиционный ЯП:

```
def fac(n):
    assert n >= 0
    if n == 0: return 1
    return n * fac(n - 1)
```

Функциональный ЯП с использованием сопоставления с образцом:

```
fac(0) -> 1;
fac(n) when n > 0 -> n * fac(n - 1).
```

В чем недостаток такого подхода?

При каждом вызове `fac()` создается новый стековый фрейм. Например, при вызове `fac(3)` происходит следующее:

1. `fac(3)` ожидает результат `fac(2)`.
2. `fac(2)` ожидает результат `fac(1)`.
3. `fac(1)` ожидает результат `fac(0)`.
4. `fac(0)` возвращает 1.

Таким образом, для `fac(3)` создается 4 стековых фрейма. Если `n` велико (например, 10000), это может привести к переполнению стека.

Для избежания такой ситуации такой ситуации можно использовать **хвостовую рекурсию** — особый вид рекурсии, где последний вызов в функции-родителе является вызовом этой же самой функции и с её результатом отсутствуют дальнейшие операции.

В этом случае не нужно сохранять состояние предыдущего вызова, так как его результат не используется после выполнения текущего вызова, что позволяет оптимизировать использование памяти на уровне компилятора используя текущий стековый фрейм для последующих рекурсивных вызовов, вместо создания новых фреймов.

Пример факториала через хвостовую рекурсию:

```
def fac(n):
    assert n >= 0
    return fac_inner(n, 1)

def fac_inner(n, acc):
    if n <= 1: return acc
    return fac_inner(n - 1, acc * n)
```

3.3 Чисто функциональные структуры данных

Чисто функциональная структура данных — это такая структура, которая может быть реализована в чисто функциональном языке программирования (ЯП). Основное отличие между произвольной структурой данных и чисто функциональной заключается в том, что последняя является (строго) неизменяемой.

Но как работать с функциональными структурами данных, если их нельзя изменять? В этом контексте функциональное программирование предлагает несколько подходов:

1. Широкое использование рекурсии обуславливает применение рекурсивных структур данных.
2. Изменения, вносимые в структуру данных, создают новую структуру, не затрагивая исходную.

Рассмотрим, как эти идеи применяются в функциональном программировании.

3.3.1 Односвязный список

Как уже было сказано ранее, применение рекурсии обуславливает применение рекурсивных структур данных. Так и в этом примере, односвязный список можно тоже представить как рекурсивную структуру данных.

Голова списка (Head) — это первый элемент списка, а хвоста списка (`Tail`) — это список, содержащий все остальные элементы.

Figure 6: Head-Tail-Example

Например, в списке `[1, 2, 3, 4]` head будет `1`, а `tail` — список `[2, 3, 4]`.

При этом пустой список не имеет head и `tail`.

Ещё один пример - список `[1]`, состоящий только из одного элемента, в таком случае его head будет `1`, а `tail` будет являться пустым списком.

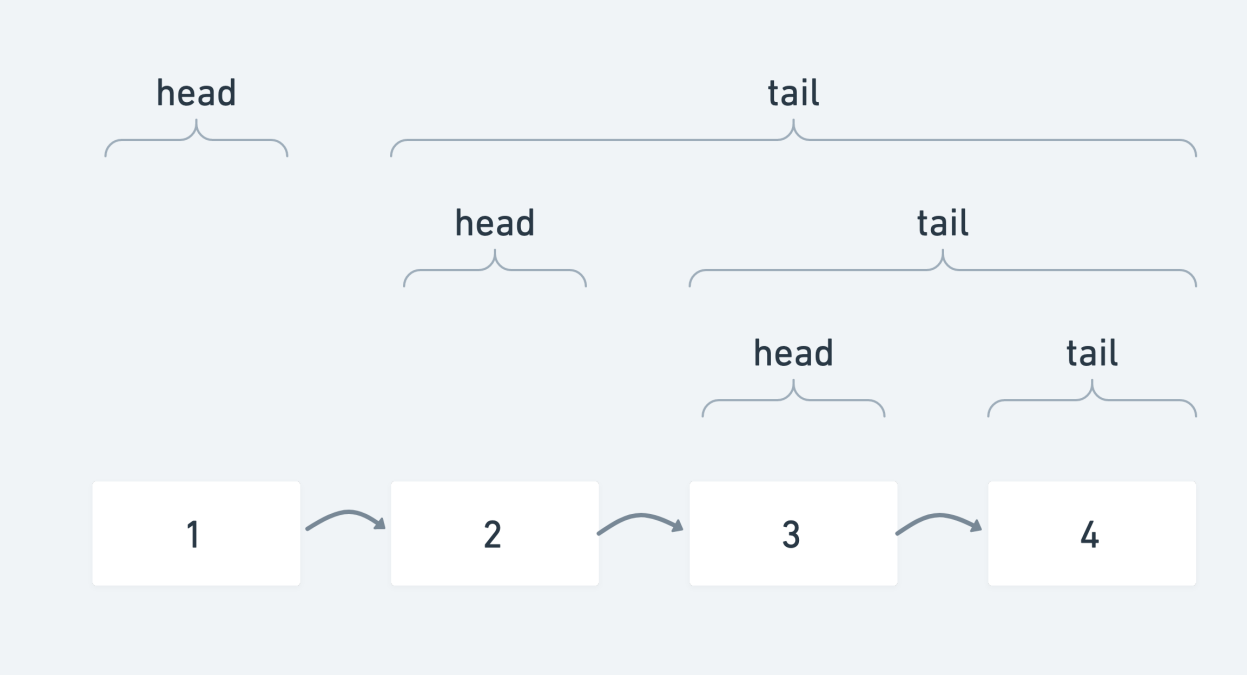


Figure 7: Head-Tail-Example

Как можно заметить, в данном примере, мы начали с головы списка, но хвост этого списка также является списком, который имеет свою собственную голову и хвост. Это также справедливо для хвоста от хвоста, и так далее, вследствие этого, данная структура данных является рекурсивной.

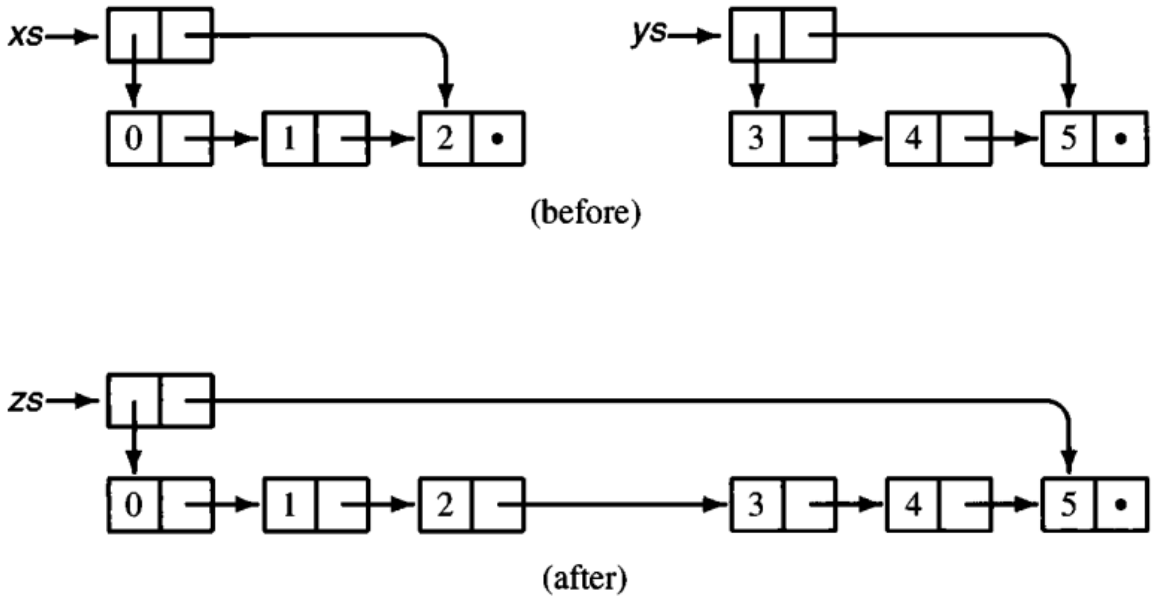


Figure 8: Head-Tail-Example

3.3.2 Нефункциональное объединение списков

На картинке представлено нефункциональное объединение списка X со списком Y в итоговый список Z.

Xs — содержит указатель на первый элемент списка и на последний элемент списка X, при этом последний элемент списка X содержит “*“, которая является указателем на пустой элемент.

После объединения списков происходит изменение указателя в последнем элементе списка с пустого элемента, на первый элемент списка Y.

Тем самым, после объединения этих списков, нарушается инвариант для **xs**. Вследствие этого, если в какой-то части программы осталось использование **xs**, мы столкнуться с неправильным поведением.

3.3.3 Функциональное объединение списков

При функциональном объединении списка не нарушается инвариант **xs** по той причине, что перед объединением списков в список Z, происходит копирование списка X.

Тем самым, функциональное объединение списков не влияет на внешнее состояние.

3.3.4 Вставка в бинарное дерево

Как и в предыдущем примере, вставка в бинарное дерево является копирующей, то есть после вставки элемента **e** фактически происходит создание нового дерева. При этом элементы, которые не изменились, например, ноды **a, b, c, h** используются без изменений, при этом для дерева с корнем **ys** были созданы новые ноды **d, g, f** вместо старых нод в дереве с корнем **xs**, так как были изменены их дочерние ноды вследствие добавления элемента **e**.

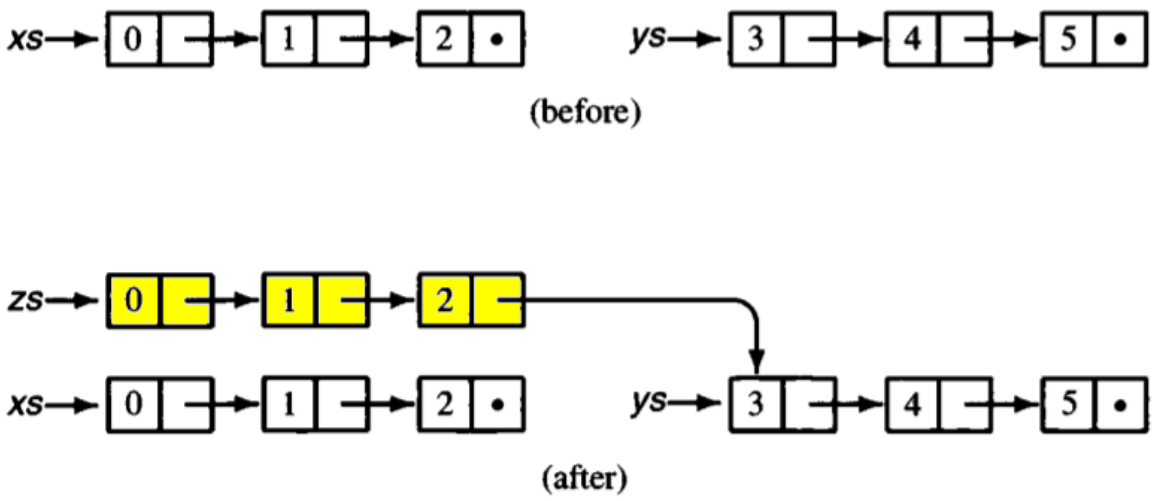


Figure 9: Head-Tail-Example

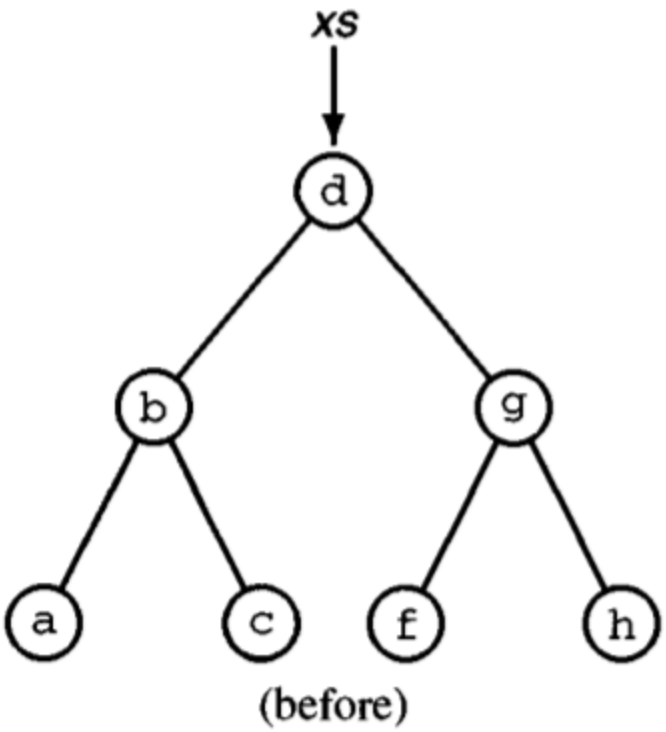


Figure 10: Head-Tail-Example

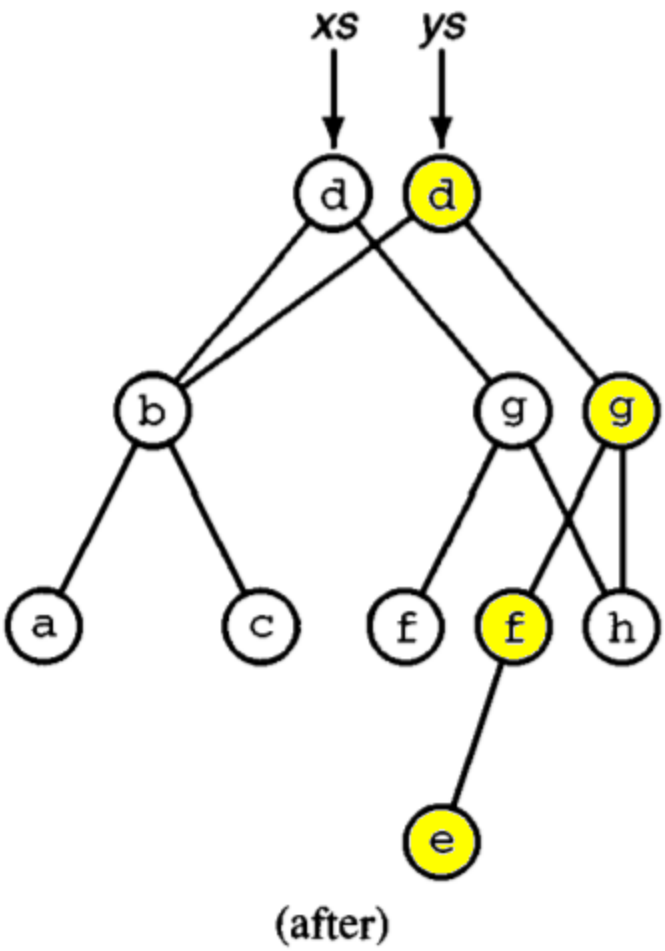


Figure 11: Head-Tail-Example

3.3.5 Бинарное дерево поиска (рекурсивный алгоритм)

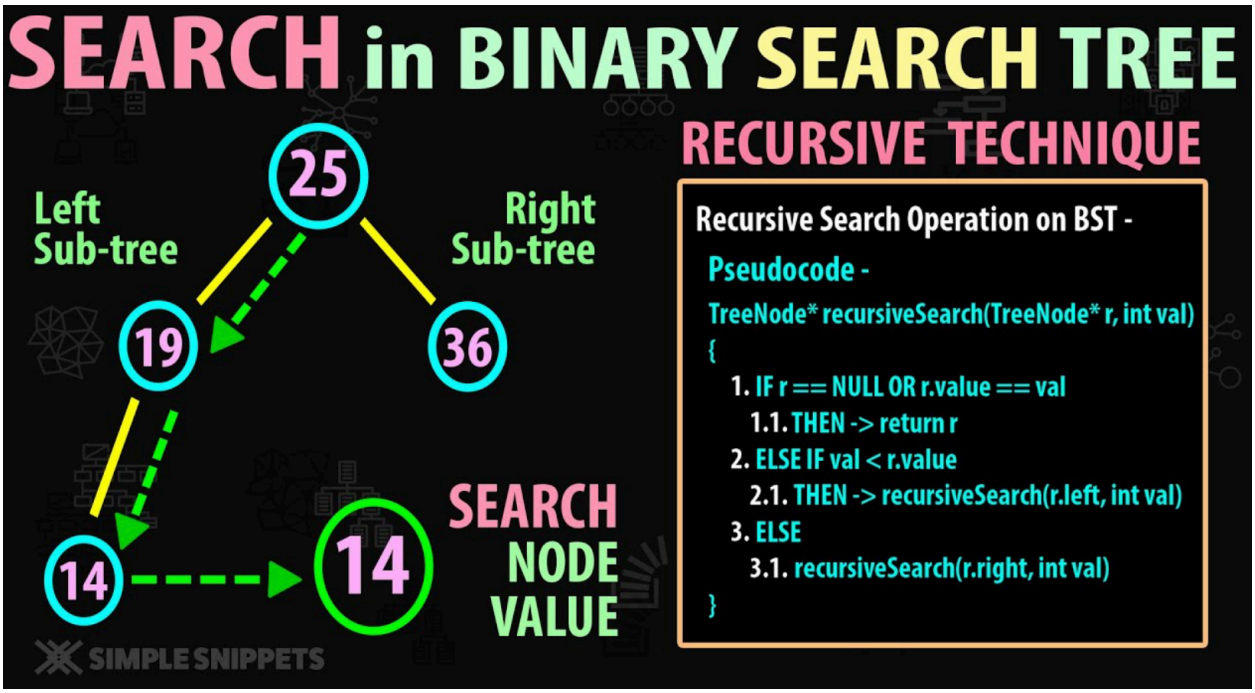


Figure 12: Head-Tail-Example

Рекурсия в функциональных языках программирования является индуктивной. Это означает, что рекурсивная функция определяется через себя, используя **базовый случай** и **индуктивный шаг**.

Базовый случай — это простейший случай, когда функция может быть вычислена напрямую, без рекурсии.

Индуктивный шаг — это шаг, в котором функция вызывает сама себя с меньшим входным значением, пока не достигнет базового случая.

Для нас это значит то, что мы можем продемонстрировать корректность работы алгоритма относительно шага индукции и условия остановки.

В контексте бинарного поиска индуктивная рекурсия работает следующим образом:

- **Базовый случай:** Если список пуст или содержит только один элемент, то элемент найден (или не найден).
- **Индуктивный шаг:** Если список содержит более одного элемента, то функция рекурсивно вызывает себя с левой или правой половиной списка, в зависимости от того, где находится искомый элемент.

Пример элегантного рекурсивного алгоритма (быстрая сортировка):

```
quick_sort([]) -> [];
quick_sort([H | T]) ->
  quick_sort([X || X <- T, X < H]) ++ [H] ++ quick_sort([X || X <- T, X >= H]).
```

Данная реализация не является самой эффективной с точки зрения затрачиваемых ресурсов, но является хорошей демонстрацией применения рекурсии.

Плюсы и минусы:

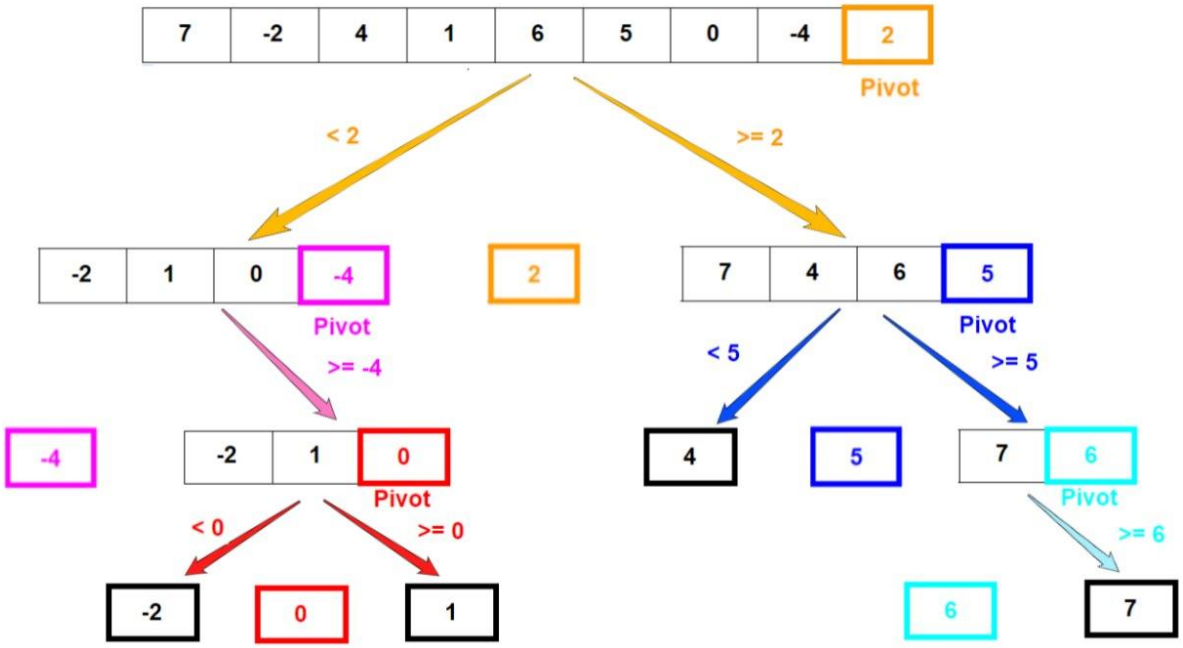


Figure 13: Head-Tail-Example

Является ли использование чисто функциональных структур данных эффективным? Почему они не используются во всех языках?

Вопрос об эффективности использования таких структур данных можно рассматривать с разных сторон. Например, с точки зрения потребления памяти, immutable структуры данных, как правило, менее эффективны, чем mutable структуры. С другой стороны, использование неизменяемых структур данных позволяет упростить и оптимизировать сборку мусора, при этом, фактически, без автоматического управления памятью не обойтись.

Тем самым, использование чисто функциональных структур данных - это tradeoff производительности и безопасности/предсказуемости кода.

3.4 Алгебраический (индуктивный) тип данных

Алгебраическим называют тип данных, состоящий из различных разновидностей (возможно, составных) термов (значений).

Как уже было сказано ранее, в функциональном программировании (ФП) мы стремимся работать с неизменяемыми (immutable) структурами данных. Поэтому было бы удобно организовать нашу программу так, чтобы состояние не могло быть представлено некорректно.

Рассмотрим пример: предположим, у нас есть необходимость реализовать структуру данных — геометрическую форму (одну из `Rectangle`, `Circle`, `Triangle`). В традиционных языках программирования мы бы получили примерно следующее:

```
enum ShapeType { RECTANGLE, CIRCLE, TRIANGLE };
```

Figure 14: Illegal-State-Unrepresentable

```
struct Rectangle {
    Point topLeft, bottomRight;
};

struct Circle {
    Point center;
    double radius;
};

struct Triangle {
    Point a, b, c;
};

union ShapeData {
    struct Rectangle rectangle;
    struct Circle circle;
    struct Triangle triangle;
};

struct Shape {
    enum ShapeType type;
    union ShapeData data;
}
```

Проблема данной реализации заключается в использовании union для хранения в памяти одной из трёх форм. В этом случае тип, хранимый в ShapeData, определяется

во время выполнения (Runtime). Таким образом, если в этот union записана, например, структура Triangle, то при попытке чтения другой структуры, которая также может быть записана в этот union, мы получаем непредсказуемое поведение, что небезопасно. Чтобы избежать этой ситуации, необходимо использовать ещё одну структуру Shape, для хранения типа, который записан в union. Однако, несмотря на все наши усилия по обеспечению безопасности кода, никто не может гарантировать, что структура будет использована правильно и что в неё не будет записан, например, неправильный тип данных.

Как уже было сказано ранее, цель функционального программирования в данном случае — сделать невозможным возникновение неправильного состояния. Рассмотрим это на другом примере:

```haskell
data Shape
    = Rectangle
        { topLeft :: Point
        , bottomRight :: Point
        }
    | Circle
        { center :: Point
        , radius :: Double
        }
    | Triangle
        { a :: Point
        , b :: Point
        , c :: Point
        }

data Point = Point { x :: Double, y :: Double }
```

В данном случае тип Shape может состоять из трёх подтипов: `Rectangle`, `Circle`, `Triangle`.

Теперь предположим, что нам необходимо использовать функцию, которая принимает Shape и выполняет некоторые операции после определения подтипа. Как это сделать? — Использовать сопоставление с образцом (pattern-matching).

```haskell
-- Геттер для получения радиуса из Shape
getRadius :: Shape -> Maybe Double
getRadius (Circle _ r) = Just r
getRadius _            = Nothing
```

Объектно-ориентированная реализация этих структур данных основывалась бы на наследовании. Это имеет один существенный недостаток по сравнению с использованием алгебраических структур данных — невозможность ограничить количество наследников (исключение составляют sealed классы). Таким образом, невозможно гарантировать, что будут учтены все наследники, из-за чего может возникнуть неправильное состояние.

В функциональном программировании JSON-объекты можно представлять с помощью алгебраических типов данных.

Объявим такой тип и функции, для работы с ним:

```haskell
data JSObject = JSNull
  | JSBool Bool
  | JSRational Bool Rational
  | JSString String
```

```
  | JSArray [ JSObject ]
  | JSObject ( String, JSObject )

encode :: JSObject -> String
encode JSNull = "null"
encode (JSBool True) = "true"
encode (JSBool False) = "false"
encode (JSString s) = s
encode (JSArray xs) = show $ map encode xs
```

Тип `JSObject` представляет собой рекурсивный тип данных, который может принимать следующие формы:

- `JSNull`: представляет значение `null`.
- `JSBool Bool`: представляет логическое значение (`true` или `false`).
- `JSString String`: представляет строку.
- `JSArray [JSObject]`: представляет массив, содержащий список объектов `JSObject`.
- `JSObject [(String, JSObject)]`: представляет объект, состоящий из пар "ключ-значение", где значение может быть любым из типов `JSObject`, включая другие объекты.

Функция `encode` преобразует значения типа `JSObject` в строку, представляющую их в формате JSON.

Пример работы с `JSObject`:

```
example = JSArray [ JSBool True, JSString "s" ]
main = putStrLn (encode example) -- => ["true","s"]
```

3.5 Структурная / вполне обоснованная рекурсия

В функциональных языках программирования все итеративные алгоритмы основываются на рекурсии.

Когда мы рассматриваем алгоритм, содержащий цикл, в общем случае невозможно определить, является ли функция конечной или бесконечной (проблема остановки).

Понятие **вполне обоснованная рекурсия** относится к особому виду рекурсии, при котором, анализируя программу, мы можем установить, что аргумент каждого рекурсивного вызова в определённой степени стремится к значению остановки (в упрощённом виде, уменьшается).

Давайте рассмотрим подробнее на примере:

```
data List a = Cons a (List a) | Empty

example = Cons 1 (Cons 2 (Cons 3 Empty))

length Empty = 0
length (Cons x xs) = 1 + length xs
```

В данном примере показан рекурсивный тип `List`, который либо является пустым, либо парой из `Head` (голова листа) и `Tail` (хвост листа).

Функция `length`, в данном случае, является примером вполне обоснованной рекурсии, так для неё можно формальным образом доказать то, что она является конечной, приведем это доказательство:

- Если пуст (`Empty`) — вычисление окончено.
- Если список не пуст (`Cons`), то осуществим индуктивный переход для аргумента `Cons x xs -> xs`, тогда:
  - Если список конечен, то в нём присутствует `Empty` — вычисление окончено.
  - Если список бесконечен, алгоритм не завершится.

Подведем итоги ключевых аспектов, которые отличают рекурсию от циклов:

- В рекурсии зависимость между шагами явно определена.
- Обоснованная рекурсия приводит к конечному алгоритму. Некоторые языки программирования, такие как Coq и Agda, требуют использования обоснованной рекурсии.
- Рекурсия предоставляет возможность естественного использования динамической памяти.

4 Функции высших порядков. Свёртки, отображения. Замыкания. Бесточечный стиль. Ссылочная прозрачность

4.1 Функции высших порядков

Функции высших порядков (Higher-Order Functions) — это функции, которые принимают другие функции в качестве аргументов и/или возвращают функции в качестве результатов.

4.1.1 Применение функций высших порядков

1. **Параметризация обобщённых функций**

   Функции высших порядков позволяют изменять поведение обобщённых функций путём передачи другой функции в качестве параметра. Пример: функция сортировки может принимать компаратор, который определяет порядок сортировки.

2. **Создание декораторов**

   Функции высших порядков могут использоваться для создания декораторов — функций, которые принимают другую функцию и возвращают её модифицированную версию. Это позволяет добавлять дополнительное поведение без изменения исходной функции. Пример: функция-декоратор, которая добавляет логирование вызовов переданной функции.

3. **Комбинация функций**

   Функции высших порядков позволяют комбинировать простые функции для создания более сложных. Пример: создание новой функции, которая сначала выполняет одну операцию над числом, а затем применяет вторую.

4.1.2 Анонимные функции

Анонимные функции (Anonymous functions) — это функции без имени, которые можно использовать непосредственно в месте их объявления. Они удобны для краткосрочных операций и передаются как аргументы другим функциям.

Примеры объявления и использования анонимных функций, инкрементирующих значение, в различных языках программирования:

```python
# Python
lambda e: e + 1 # Создание анонимной функции
```

```
(lambda e: e + 1)(1) # Создание и вызов анонимной функции
# => 2

-- Haskell
\x -> x + 1 -- Создание анонимной функции
(\x -> x + 1) 1 -- Создание и вызов анонимной функции
-- => 2

;; Clojure
(fn [x] (+ 1 x)) ;; Создание анонимной функции. Эквивалентно #(+ 1 %)
((fn [x] (+ 1 x)) 1) ;; Создание и вызов анонимной функции
;; => 2
```

В примерах выше сначала демонстрируется создание анонимной

4.1.3 Каррирование и композиция функций

Каррирование (Currying) — преобразование функции от многих аргументов в набор вложенных функций, каждая из которых является функцией от одного аргумента.

Композиция функций (Function composition) позволяет объединять несколько функций в одну, так что результат одной функции передаётся в другую.

Примеры каррирования и композиции функций в различных языках программирования:

```
# Python
import functools

add = lambda a, b: a + b # Определяем анонимную функцию сложения
inc = functools.partial(add, 1) # Создаем функцию, которая увеличивает число на 1

def comp(g, h): # Определяем композицию функций
    return lambda x: g(h(x))  # Возвращаем функцию, которая применяет g к результату h(x)

inc2 = comp(inc, inc) # Композиция inc и inc, то есть увеличение числа дважды

inc2(1) # Результат применения inc2 к числу 1
# => 3

-- Haskell
inc = (+1) -- Функция увеличивает число на 1
inc2 x = inc (inc x) -- Функция увеличивает число на 2 (два раза применяет inc)

inc2 1 -- Вызов функции inc2
-- => 3

;; Clojure
(def inc (partial + 1))  ;; Функция увеличивает число на 1, используя partial
((comp inc inc) 1)  ;; Композиция двух inc, которая увеличивает число дважды
```

4.1.4 Фильтрация, Отображение и Свёртка

Функции высшего порядка Filter, Map и Reduce являются краеугольными камнями функционального программирования. Они реализуют мощный способ обработки элементов структур данных. Отличительной особенностью этих функций является то, что они скрывают структуру данных, позволяя работать с элементами коллекции без необходимости явно манипулировать её представлением.

4.1.4.1 Фильтрация Функция-фильтр принимает в качестве аргумента функцию, которая используется для определения, какие элементы должны быть включены в результирующую коллекцию.

Примеры использования фильтрации в различных языках программирования:

```
# Python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

def is_even(n):
    return n % 2 == 0

list(filter(is_even, numbers)) # Фильтруем исходный массив чисел, оставляя только чётные
# => [2, 4, 6, 8]

# Python с использованием генератора
def myfilter2(pred, lst):
  for e in lst:
    if pred(e):
      yield e # Возвращаем элемент через генератор
```

Генераторы — это функции в Python, которые возвращают итераторы. Вместо использования `return` они используют ключевое слово `yield`.

Основные преимущества:

1. Экономия памяти: Генераторы создают элементы “на лету”, не занимая память под весь список.
2. Ленивые вычисления: Элементы создаются только при запросе, что удобно для работы с большими или бесконечными последовательностями.
3. Простота реализации: Генераторы упрощают реализацию сложных итераторов, устраняя необходимость управлять состоянием вручную.
4. Повышение производительности: За счет ленивой генерации данных программа работает быстрее в некоторых сценариях, например, при обработке потоков данных.

```
-- Haskell
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

isEven :: Int -> Bool
isEven n = n `mod` 2 == 0

filter isEven numbers -- Фильтруем исходный массив чисел, оставляя только чётные
-- => [2, 4, 6, 8]
```

В Haskell ленивая семантика позволяет вычислять значения только по мере необходимости. Рассмотрим подробно, как это работает на примере:

```
head $ filter even [1..9] =>
head $ filter even (1 : [2..9]) =>
head $ if even 1 then 1 : (filter even [2..9])
                 else     filter even [2..9] =>
head $ if false  then 1 : (filter even [2..9])
                 else     filter even [2..9] =>
head $ filter even (2 : [3..9]) =>
head $ if even 2 then 2 : (filter even [3..9])
                 else     filter even [3..9] =>
head $ if true   then 2 : (filter even [3..9])
```

```
                else      filter even [3..9] =>
head $ 2 : (filter even [3..9]) =>
2
```

Сначала фильтруется список [1..9] на четность с помощью filter even, но элементы списка вычисляются только по мере необходимости. Каждый элемент проверяется по порядку, начиная с головы, и для каждого элемента вычисляется хвост tail. Когда применяется head, выполнение останавливается на первом четном числе — 2, и дальнейшие элементы (хвост) не проверяются.

Ключевые особенности лени:

1. Ленивое вычисление: элементы обрабатываются по мере их запроса.
2. Минимальные вычисления: вычисления останавливаются, как только достигнут нужный результат.
3. Эффективность: разделение списка на вычисления позволяет экономить время и память.

4.1.4.2 Отображение Функция-отображение принимает в качестве аргумента функцию, которая применяется ко всем элементам коллекции и возвращает новую коллекцию с результатами применения этой функции к каждому элементу.

Примеры использования отображения в различных языках программирования:

```python
# Python (Императивные отображения)
def map(f, lst):
    acc = []
    for e in lst:
        acc.append(f(e))
    return acc

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

def square(n):
    return n ** 2

list(map(square, numbers)) # Преобразовываем каждое число из массива, возводя его в квадра
# => [1, 4, 9, 16, 25, 36, 49, 64, 81]

# Python (Декларативные отображения)
def map(f, lst):
    for e in lst:
        yield f(e)
```

В данной реализации обеспечивается экономия памяти благодаря тому, что объединённые операции, такие как map и filter, обрабатывают элементы коллекции по одному, вместо работы с целой структурой данных сразу. Это делает обработку более эффективной. Важно также учитывать семантические аспекты: функции должны быть понятны и предсказуемы в своей работе. Использование чистых функций позволяет не только улучшить анализ кода, но и раскрыть возможности для параллелизации вычислений. Благодаря этому операции могут выполняться на нескольких группах элементов одновременно, что повышает производительность. Среда исполнения может автоматически оптимизировать обработку, учитывая размер и структуру коллекции, что дополнительно ускоряет выполнение задач.

```clojure
;; Clojure
(defn map [f coll] ;; Реализация map
```

```
  (if (empty? coll) '() (cons (f (first coll)) (map f (rest coll)))))

(def numbers [1 2 3 4 5 6 7 8 9])

(defn square [n]
  (* n n))

(map square numbers)
;; => [1 4 9 16 25 36 49 64 81]
```

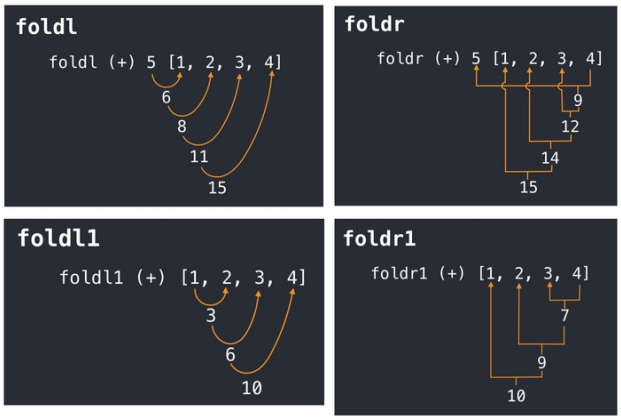
Функция `map` в данном примере рекурсивно применяет функцию `f` к каждому элементу коллекции `coll`, создавая новую коллекцию, где каждый элемент является результатом применения `f`. Если коллекция пуста, возвращается пустой список. Результат строится с использованием функции `cons`, которая добавляет преобразованный элемент к результату обработки оставшейся части коллекции.

Важно отметить, что при передаче пустой коллекции в функцию `map` на выходе будет пустая коллекция, но её тип может отличаться от типа исходной, хотя содержимое останется идентичным.

4.1.4.3 Свёртка Функция-свёртка принимает в качестве аргумента функцию и коллекцию, и используется для последовательного применения этой функции к элементам коллекции с накоплением результата. В результате получается одно значение, которое является итогом свёртки всех элементов коллекции.

Левая свёртка (Left fold) выполняется с использованием первого элемента коллекции и затем применяется к следующему элементу, пока не обработаются все элементы коллекции.

Правая свёртка (Right fold) начинается с последнего элемента коллекции и двигается к начальному, сворачивая элементы с правой стороны.



В функциональных языках важно, с какой стороны начинается свёртка при работе со списками. Если начать с обратной стороны, то необходимо сначала произвести реверс списка, иначе это приведёт к излишнему потреблению ресурсов.

Примеры использования свёртки в различных языках программирования:

```
# Python (Правая свёртка)
def fold(f, init, lst):
    for e in reversed(lst):
        init = f(e, init)
    return init
```

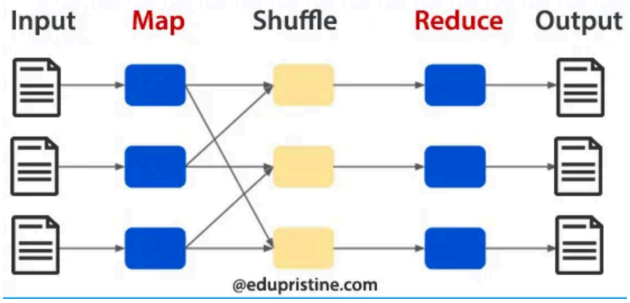
```python
fold(lambda a, b: a + b, 0, [1, 2, 3]) # Находим сумму чисел массива, используя свёртку
# => 6
```

```haskell
-- Haskell (Левая свёртка)
foldl :: (s -> a -> s) -> s -> [a] -> s
foldl _ _ [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs

foldl (+) 0 [1, 2, 3]
-- => 6
```

4.1.4.4 Map-Reduce **Map-Reduce** — это модель распределённых вычислений от компании Google, используемая для параллельных вычислений над очень большими наборами данных.

Принцип работы заключается в разделении задачи на два этапа: на этапе Map данные разделяются на подзадачи, которые обрабатываются параллельно, а на этапе Reduce результаты этих подзадач агрегируются и обрабатываются для получения итогового результата.



Трудности реализации:

1. Обработка ошибок: При работе с большими распределёнными системами важно корректно обрабатывать ошибки, такие как сбои при обработке данных или сети. Перезапуск задач и восстановление состояния после ошибок требуют сложных механизмов для обеспечения устойчивости.
2. Проблема остановки/отмены: Остановка или отмена задач на одном из этапов (например, на этапе Reduce) может привести к нарушению согласованности данных или к потере результатов, если не предусмотрены механизмы для корректного завершения работы с учётом уже обработанных данных.
3. Консистентность: В распределённых системах важно поддерживать консистентность данных, особенно при параллельной обработке. Без надлежащей синхронизации могут возникнуть проблемы с некорректным объединением результатов, что приведёт к ошибкам в итоговых вычислениях.
4. Отказ узлов: Отказ отдельных узлов в распределённой системе может привести к потерям данных или замедлению работы, если не предусмотрены механизмы восстановления или перераспределения задач между оставшимися активными узлами.

4.2 Замыкания

Замыкания (Closures) — это функции, которые “замыкают” в себе контекст, в котором они были созданы, позволяя им использовать переменные из этого контекста даже после того, как выполнение программы вышло из этой области.

Динамический тип функции: тип такой функции не определён заранее.

```python
# Python
def itemSort(items, price):
    return sorted(items, key=lambda x: abs(price - x)) # Замыкаем price внутри анонимной ф
```

Статический тип функции: тип такой функции определён на этапе компиляции. К примеру, в языке Go до добавления generics сортировка была реализовано таким образом, что использовалась функция, которая принимала два индекса элементов (`i` и `j`) и возвращала булево значение, указывающее на то, должен ли элемент с индексом `i` идти перед элементом с индексом `j` в отсортированном срезе. Замыкание сортируемой коллекции позволило реализовать такое элегантное решение в условиях отсутствия generics.

```go
// Go
sort.Slice(items, func(i, j int) bool {
    return math.Abs(items[i]) > math.Abs(items[j]) // Замыкаем items
})
```

4.2.1 Чистые замыкания

Чистые замыкания (Pure closures) — это замыкания, которые работают как чистые функции. Они обладают такими характеристиками:

- Детерминированность: возвращаемое значение функции зависит только от её аргументов, и она всегда возвращает один и тот же результат при одинаковых входных данных.
- Отсутствие побочных эффектов: чистые замыкания не изменяют состояния или переменных вне своей области видимости.
- Иммутабельность окружения: переменные, на которые ссылается замыкание, не изменяются.

Пример чистого замыкания:

```python
# Python
weekdays = {
    0: 'Sunday',
    1: 'Monday',
    2: 'Tuesday',
    3: 'Wednesday',
    4: 'Thursday',
    5: 'Friday',
    6: 'Saturday',
    7: 'Sunday'
}

def n2weekdayName(n):
    return weekdays[n] # Состояние weekdays не изменяется

days = [1, 1, 2, 2, 3, 3]

list(map(n2weekdayName, days))
# => ['Monday', 'Monday', 'Tuesday', 'Tuesday', 'Wednesday', 'Wednesday']
```

4.2.2 Грязные замыкания

Грязные замыкания (Impure closures) — это замыкания, которые могут изменять свое внешнее окружение или содержать побочные эффекты, а также иметь разное поведение при одинаковых входных данных. Такие замыкания снижают предсказуемость и стабильность кода.

Пример грязного замыкания:

```
# Python
def mk_counter(begin=0):
    i = begin
    def counter():
        nonlocal i
        tmp = i
        i += 1 # Изменяется состояние i
        return tmp
    return counter

c1 = mk_counter(0)
c2 = mk_counter(10)

list(c1(), c1(), c2(), c2(), c1())
# => 0 1 10 11 2
```

4.2.3 Замыкания и ООП

Замыкания могут использоваться как способ реализации ООП благодаря способности сохранять состояние и поведение в области видимости функции. Они позволяют создавать структуры, напоминающие объекты, где функции выступают в роли методов, а переменные замыкания — в роли атрибутов. Такой подход обеспечивает инкапсуляцию и управление состоянием, что соответствует основным принципам ООП. В функциональном программировании это позволяет создавать функциональные аналоги объектов, сохраняя состояние и поведение без необходимости классов, что поддерживает чистоту и иммутабельность кода.

Пример реализации ООП без замыканий:

```
# Python
class DogByClass(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def description(self):
        return '{} is {} years old'.format(self.name, self.age)
    def speak(self, sound):
        return '{} says {}'.format(self.name, sound)
    def happy_birth_day(self):
        self.age += 1

d1 = DogByClass("Fluffy", 1)

d1.description() # Fluffy is 1 years old
d1.speak("woof") # Fluffy says woof
d1.happy_birth_day()
d1.description() # Fluffy is 2 years old
```

Пример реализации ООП через замыкания:

```python
# Python
def DogByClosure(name, age):
    def description():
        return '{} is {} years old'.format(name, age)
    def speak(sound):
        return '{} says {}'.format(name, sound)
    def happy_birth_day():
        nonlocal age
        age += 1
    return {
        'description': description,
        'speak': speak,
        'happy_birth_day': happy_birth_day,
    }

d2 = DogByClosure("Buddy", 1)
print(d2['description']())  # Buddy is 1 years old
print(d2['speak']("bark"))  # Buddy says bark
d2['happy_birth_day']()
print(d2['description']())  # Buddy is 2 years old
```

Пример реализации ООП через замыкания на функциональном языке:

```clojure
;; Clojure
(defn dog [name age]
  (let [*age (atom age)]
    {:description #(str name " is " @*age " years old")
     :speak #(str name " says " %)
     :happy-birth-day #(swap! *age + 1)}))

(def d (dog "Skippy" 1))

((:description d))      ; => "Skippy is 1 years old"
((:happy-birth-day d))  ; => 2
((:description d))      ; => "Skippy is 2 years old"
((:speak d) "bork")     ; => "Skippy says bork"

(defmacro => [obj method & args]
  `((~method ~obj) ~@args))

(macroexpand '(=> d :speak "bork")) ; => ((:speak d) "bork")

(=> d :speak "bork")    ; =>"Skippy says bork"
```

atom используется для управления состоянием, обеспечивая атомарность операций. В данном случае он управляет возрастом, позволяя изменять его через замыкания.

Пример реализации ООП через процессы и сообщения на функциональном языке:

```erlang
% Erlang
% Реализация объекта
start(Name, Age) ->
    spawn(fun() -> loop(Name, Age) end).

loop(Name, Age) ->
```

```
  receive
    {description, From} ->
      From ! {self(), io:format("~s is ~p years old", [Name, Age])},
      loop(Name, Age);
    {speak, Sound, From} ->
      From ! {self(), io:format("~s says ~s", [Name, Sound])},
      loop(Name, Age);
    {happy_birth_day, From} ->
      NewAge = Age + 1,
      From ! {self(), NewAge},
      loop(Name, NewAge)
  end.

% Реализация интерфейса для взаимодействия с объектом
description(Dog) ->
    Dog ! {description, self()},
    receive {Dog, Msg} -> Msg end.

speak(Dog, Sound) ->
    Dog ! {speak, Sound, self()},
    receive {Dog, Msg} -> Msg end.

happy_birth_day(Dog) ->
    Dog ! {happy_birth_day, self()},
    receive {Dog, NewAge} -> NewAge end.

% Взаимодействие с объектом
Dog = start("Buddy", 1), % => <0.93.0>
description(Dog),      % => Buddy is 1 years old__ok
speak(Dog, "bark"),    % => Buddy says bark__ok
happy_birth_day(Dog),  % => 2
description(Dog).      % => Buddy is 2 years old__ok
```

В Erlang ООП реализуется через процессы, где каждый объект представляет собой процесс. Эти процессы взаимодействуют через сообщения, что гарантирует изоляцию состояний и асинхронность.

Процесс создается с помощью функции spawn, которая запускает функцию обработки сообщений (в данном случае loop), в которой хранится состояние объекта. Далее объект ожидает сообщений, например, для получения описания или выполнения действий. Каждое полученное сообщение обрабатывается внутри цикла receive, и в ответ отправляется новое сообщение.

4.3 Бесточечный стиль

Бесточечный стиль (Point-free style) — это подход, при котором функции определяются без явного указания аргументов. Вместо этого выражения строятся путём комбинирования функций, фокусируясь на том, что делают функции, а не на данных, которые они получают. Такой стиль позволяет создавать более лаконичные и декларативные функции, избегая лишних переменных и повышая читабельность кода, особенно в цепочках преобразований.

Примеры реализации бесточечного стиля в различных языках программирования:

```
// Java (Stream API)
Stream.of("a1", "a2", "a3")
```

```
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);
```

В Java можно использовать ссылку на метод (Object::method), который будет вызываться для каждого элемента коллекции, при этом избавляясь от необходимости указания аргументов.

```
-- Haskell (Комбинаторы)
foldl (+) 0 . map (\Just x -> x) . filter isJust . map readMaybe
```

В Haskell используются комбинаторы (.), которые передают функции в цепочку обработки данных без явного указания аргументов. Данный код пытается преобразовать строки в числа, затем фильтрует только успешные результаты и выполняет суммирование всех чисел.

```
;; Clojure (Threading Macros)
(defn calculate* []
   (->> (range 10)
        (filter odd?)
        (map #(* % %))
        (reduce +)))
```

В Clojure, макросы -> и ->> используются для упрощения цепочек вызовов функций, улучшая читаемость кода. -> (thread-first): Передает результат текущего выражения как первый аргумент следующей функции. ->> (thread-last): Передает результат текущего выражения как последний аргумент следующей функции.

4.4 Мемоизация

Рассмотрим наивную реализацию функции для расчёта чисел Фибоначчи:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Каждый вызов функции fib для значений n > 1 приводит к множественным вычислениям для одинаковых подзадач. Например, fib(3) вызывает fib(2) и fib(1), а затем fib(2) вызывает снова fib(1) и fib(0). Это приводит к экспоненциальному росту числа вызовов. Для решения проблем такого рода можно использовать **мемоизацию**.

Мемоизация (Memoization) — это метод оптимизации, который сохраняет результаты выполнения функции с определёнными аргументами в кэше, чтобы избежать повторных вычислений. Она применима только к чистым функциям, которые всегда возвращают один и тот же результат при одинаковых входных значениях и не имеют побочных эффектов. Мемоизация широко используется в рекурсивных функциях, существенно снижая затраты времени на выполнение за счёт повторного использования ранее вычисленных результатов.

Пример реализации мемоизации:

```
# Python
def mycache(f): # Декоратор для кеширования
    buf = {} # Кеш результатов
    def foo(x):
        if x in buf: return buf[x]  # Возврат из кеша, если есть
        tmp = f(x)
        buf[x] = tmp # Вычисление и сохранение результата
```

```
        return tmp
    return foo
```

Декоратор mycache кеширует результаты выполнения функции, чтобы избежать повторных вычислений с одинаковыми аргументами. Он сохраняет результаты в словарь buf и при следующем вызове с тем же аргументом возвращает значение из кеша вместо выполнения функции заново.

Примеры использования мемоизации в различных языках программирования:

```
# Python
@mycache # Используем созданный декоратор
def fib(n):
    if n == 0 or n == 1: return 1
    return fib(n-1) + fib(n-2)
```

```
;; Clojure
(defn fib [n]
  (if (<= n 1)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))

(def cached-fib (memoize fib)) ;; memoize - Функция стандартной библиотеки Clojure
```

```
-- Haskell
fib n = fiblist !! n
  where
    fiblist = 1 : 1 : ( zipWith (+) fiblist (tail fiblist) )
```

Функция fib возвращает n-е число Фибоначчи, используя ленивый список fiblist. Оператор !! извлекает элемент из списка по индексу. Список fiblist начинается с двух единиц (1 : 1 : ...). Затем используется zipWith (+) для создания оставшихся чисел Фибоначчи: zipWith (+) объединяет два списка, поэлементно суммируя их значения. Первый список — это сам fiblist, который хранит все числа Фибоначчи. Второй список — это tail fiblist, то есть тот же список, но без первого элемента. Таким образом, каждая пара элементов из fiblist и tail fiblist суммируется, генерируя следующее число Фибоначчи.

4.5 Ссылочная прозрачность

Ссылочная прозрачность (Referential transparency) — это свойство выражения, при котором его результат всегда может быть заменён самим значением выражения без изменения поведения программы. В таких выражениях для каждого набора входных значений результат выполнения всегда идентичен, и отсутствуют побочные эффекты.

4.5.1 Эквивалентность

4.5.1.1 Подходы к определению эквивалентности К определению эквивалентности объектов можно подойти по-разному.

Рассмотрим пример в Common Lisp:

1. = сравнивает только числа, независимо от их типа.
2. eq сравнивает символы. Два объекта считаются eq, если они на самом деле являются одним и тем же объектом в памяти. Не используйте это для чисел и символов.
3. eql сравнивает символы аналогично eq, а также числа (с учетом типа) и символы (с учетом регистра).

4. equal сравнивает более общие объекты. Два объекта считаются equal, если они являются eql, строки — это eql символов, битовые векторы имеют одинаковое содержимое, или списки содержат equal объекты. В остальных случаях используется eq.
5. equalp аналогичен equal, но более продвинут. Сравнение чисел нечувствительно к типу. Сравнение символов и строк нечувствительно к регистру. Списки, хеши, массивы и структуры считаются equalp, если их элементы — это equalp. В остальных случаях используется eq.

Из-за наличия большого количества способов сравнения объектов может возникнуть неочевидное поведение.

Рассмотрим примеры неочевидного поведения из-за работы с объектами по ссылкам в Python:

```python
# Python
x = 0
y = 0
while True:
    x += 1
    y += 1
    print(x, y, x is y, x == y)
    if x is not y:
        break
# => 1 1 True True
# => 2 2 True True
# => ...
# => 256 256 True True
# => 257 257 False True
```

1. is проверяет, указывают ли две переменные на один и тот же объект в памяти.
2. == проверяет, равны ли значения объектов.

Как следствие, в данном примере наблюдается неочевидное поведение из-за внутренних механизмов языка (в данном примере из-за интернирования целых чисел).

```
def create_functions():
    funcs = []
    for i in range(4):
        funcs.append(lambda: i)
    return funcs

funcs = create_functions()

for f in funcs:
    print(f(), end=' ') # => 3 3 3 3
```

В данном примере неочевидность поведения заключается в том, что лямбда-функция захватывают не значение переменной i, а её ссылку. Из-за этого после завершения цикла funcs содержит четыре лямбда-функции с одинаковыми значениями i, а не те значения, которые были на каждом шаге цикла.

4.5.1.2 Примеры эквивалентности при ссылочной прозрачности Ссылочная прозрачность предоставляет возможность заменять выражения их значениями без изменения поведения программы, что позволяет экономить память и улучшает читаемость кода.

```python
# Python
a = [1, 2, 3]
b = [1, 2, 3]

# Эквивалентная реализация, возможная благодаря ссылочной прозрачности

a = [1, 2, 3]
b = a
```

```python
# Python
c = {:a 1 :b 2}
d = {:a 1 :b 2}

# Эквивалентная реализация, возможная благодаря ссылочной прозрачности

c = {:a 1 :b 2}
d = c
```

```python
# Python
a = f(x)
b = f(x)
c = a + b

# Эквивалентная реализация, возможная благодаря ссылочной прозрачности

a = f(x)
c = a + a
```

4.5.1.3 Пример развертывания цикла при ссылочной прозрачности

В примере ниже вместо того чтобы вызывать `delete(x)` для каждого значения по очереди, компилятор может выполнить сразу несколько удалений для последовательных значений в одном шаге, что потенциально улучшает производительность.

```c
// C
for (int x = 0; x < 100; x++) {
    delete(x);
}

// Эквивалентный пример, возможный благодаря ссылочной прозрачности

for (int x = 0; x < 100; x += 5 ) {
    delete(x);
    delete(x + 1);
    delete(x + 2);
    delete(x + 3);
    delete(x + 4);
}
```

4.5.1.4 Пример реорганизации аргументов функции при ссылочной прозрачности

В примере ниже реорганизуются аргументы функции, перемещая вызов `bar(x)` в место, где он фактически используется, вместо того чтобы вычислять его заранее. Это позволяет уменьшить количество вычислений, так как `bar(x)` вычисляется только при необходимости.

```python
# Python
def foo(p, a, b):
    if p(a):
        return b

foo(is_even, x, bar(x))

# Эквивалентная реализация, возможная благодаря ссылочной прозрачности

def foo(p, a, b):
    if p(a):
        return bar(x)

foo(is_even, x, x)
```

4.5.1.5 Суперкомпиляция при ссылочной прозрачности Наличие ссылочной прозрачности в языке теоретически открывает возможность реализации суперкомпиляции, поскольку позволяет компилятору оптимизировать код, например, путем предсказания и развертывания вычислений, что делает программу более эффективной и снижает избыточные вычисления.

4.5.2 Сопоставление с образцом

Сопоставление с образцом (Pattern matching) — это механизм, позволяющий проверить структуру и значения данных на соответствие заданным шаблонам или образцам. Часто применяется в функциональном программировании для удобной декомпозиции сложных структур (например, списков, кортежей), благодаря чему можно задавать разные действия для разных форм данных.

Примеры сопоставления с образцом в разных языках программирования:

```clojure
;; Clojure
(doseq [n (range 1 101)]  ;; Итерируемся по числам от 1 до 100
  (println
    (match [(mod n 3) (mod n 5)]  ;; Сопоставляем остатки от деления на 3 и 5
      [0 0] "FizzBuzz"  ;; Если кратно и 3, и 5, выводим "FizzBuzz"
      [0 _] "Fizz"      ;; Если кратно 3, выводим "Fizz"
      [_ 0] "Buzz"      ;; Если кратно 5, выводим "Buzz"
      :else n)))        ;; В остальных случаях выводим число
```

```fsharp
// F#
for n in 1 .. 100 do
    match (n % 3, n % 5) with
    | (0, 0) -> printfn "FizzBuzz"
    | (0, _) -> printfn "Fizz"
    | (_, 0) -> printfn "Buzz"
    | _ -> printfn "%d" n
```

5 Динамическая верификация. Тестирование

Конспект лекции по функциональному программированию – Динамическая верификация.

5.1 Валидация. Верификация – основные понятия

Валидация и верификация – базовые понятия для компьютерных систем.

Валидация – проверка системы на соответствие неформализованному запросу стейкхолдеров. Отвечает на вопрос “Have we done the **right thing**?”

Верификация – проверка на соответствие спецификации Отвечает на вопрос: “Have we **done** the thing **right**?”.

Таким образом, верификация требует наличия документа с формализованными требованиями, на основе которого будут проводиться проверки.

Следующий пример призван подчеркнуть разницу между валидацией и верификацией: Большая Берта – пушка времён Первой мировой войны. Она соответствовала функциональным требованиям: стреляла с огромной разрушительной силой на большие расстояния, однако использовать её было нецелесообразно. Ввиду веса в 41 тонну, она могла перемещаться только по железной дороге, что делало её использование на линии фронта затруднительным. То есть, она успешно проходила верификационные испытания, но не валидационные.

5.2 Подходы к верификации – основные понятия

Верификация делится на два класса:

- **Динамическая** – осуществляется через эксперимент. Тестирование в привычном понимании этого слова.
- **Статическая** – проверки применяются к артефактам программной системы: код как текст, документация, модели. Запуск программы, и даже само существование готовой системы не требуется.

Эксперимент затрагивает набор конкретных тестовых случаев. Поэтому он не способен продемонстрировать отсутствие ошибок, лишь выявить их с некоторой вероятностью.

В то же самое время, статическая верификация позволяет рассуждать о целых категориях вопросов, касательно поведения системы. Проще говоря, проверяет её свойства.

Это даёт больше гарантий корректности, но подобная работа с общим случаем крайне затруднительна и требует ухищрений. Например, при проверке всех ветвей исполнения программы, возникает path explosion, с которым необходимо бороться, склеивая ветви и разворачивая циклы.

5.3 Динамическая верификация – виды (основные понятия)

Динамическая верификация классифицируется следующим образом:

По аспекту, на котором тестирование фокусируется:

- Функциональное – проверка поведения системы на соответствие ожидаемому поведению
- Нагрузочное – проверка работоспособности системы в условиях критического использования ресурсов
- Стресс–тестирование – проверка возможности восстановления системы. Например, проверка того, что после сбоя не осталось никаких артефактов.
- Стабильности, безопасности, локализации, совместимости и т.д.

По уровню тестирования:

- Модульное – тестирование отдельных компонентов, остальные при этом заменяются mock-моделями.
- Интеграционное – тестирование взаимодействия компонентов.
- Системное – тестирование системы как единого целого.

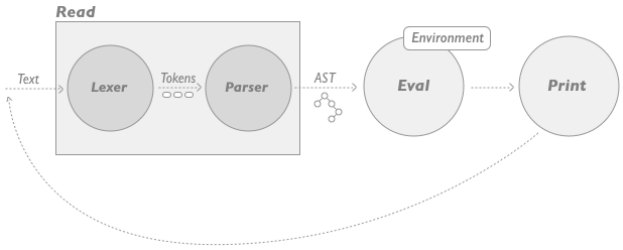
По характеру проведения тестирования:

- Ручное. (В данной лекции рассматривается только REPL)
- Автоматизированное

5.4 REPL – READ EVAL PRINT LOOP

REPL – это один из видов ручного динамического тестирования, который заключается в выполнении следующей последовательности действий:

ПРОЧИТАТЬ → ВЫПОЛНИТЬ КОМАНДУ → ВЫВЕСТИ РЕЗУЛЬТАТ → ПОВТОРИТЬ



REPL может быть реализован на основе интерпретатора, поэтому он в том или ином виде есть у следующих языков: Haskell, Clojure, SmallTalk, Common Lisp, Erlang, Python.

Однако REPL приведенных выше языков сильно отличаются функциональностью. Например, REPL Python предоставляет возможности отладки, но не позволяет сразу же исправить выявленную ошибку:

```
>>> def devide(a: int, b: int):
...     return a / b
...
>>> devide(1, 2)
0.5
>>> devide(1, 0)
Traceback (most recent call last):
  File "<python-input-4>", line 1, in <module>
    devide(1, 0)
    ~~~~~~^^^^^^
  File "<python-input-2>", line 2, in devide
    return a / b
           ~~^~~
ZeroDivisionError: division by zero
```

В данном примере рассматривается ручное тестирование функции деления целых чисел. Тестовый случай, при котором производится деление на 0, приводит к выброшенному исключению. Однако REPL предоставляет только stack trace, показывающий, где именно произошла ошибка. Он не предлагает пользователю никаких действий, которые бы позволили обработать ее на ходу, что делает исправление ошибок более трудоемким.

Подобной проблемой страдает и GHCi Haskell'а. Отметим, что благодаря свой системе типов, Haskell прямо в REPL предоставляет информацию о типах аргументов и результата функции, что помогает при отладке и разработке:

```
ghci> let devide a b = div a b
ghci> :i devide
devide :: Integral a => a -> a -> a
        -- Defined at <interactive>:1:5
ghci> devide 5 2
2
ghci> devide 5 0
*** Exception: divide by zero
```

5.4.1 Требования к REPL

Чтобы REPL было удобно использовать для тестирования и разработки, он должен:

- Позволять вручную изменять состояние нашей программы при проведении его полной инспекции – это условие означает, что хороший REPL должен также быть и отладчиком.
- При возникновении ошибок или нежелательного результата REPL должен позволять **на лету** изменять реализацию функции или инструкции, при этом:
  1. **Не** теряя текущее состояние процесса
  2. **Не** останавливая текущий процесс

  Этот механизм называется “Hot Code Reloading” и является главным отличием REPL от обычных интерпретаторов.
- Позволяет **на лету** реализовывать программу. Иными словами, хороший REPL должен позволять по–настоящему редактировать код приложения.

Примером, который соответствует перечисленным требованиям, служит REPL Small Talk. При возникновении ошибок он допускает следующие способы её обработки:

- Доопределить функцию
- Подменить её вызов другой функцией прямо в Runtime

Существуют и более специфические примеры. Язык Erlang предназначен для создания распределённых вычислительных систем. Благодаря его REPL можно привести запущенный сервер в состояние, в котором возникает ошибка, а затем исправлять код без его отключения.

Схожим образом работает и REPL Clojure. Он позволяет подключиться к запущенному серверу, вызвать ошибку, замкнуть текущее состояние в глобальную переменную для дальнейшего исследования, внести изменения в код и повторно запустить тестовый сценарий.

5.4.2 Разработка приложения с использованием REPL

Стоит отметить, что принцип разработки приложения в REPL сильно отличается от привычного многим программистам способа.

Обычно при разработке приложения вы реализуете его сверху–вниз, от большего к меньшему: декларируете модуль, декларируете класс, декларируете его функционал, реализуете его функционал.

В случае с REPL подход должен быть противоположным – снизу–вверх: Сначала вы реализуете логику в виде некоторых функций, тестируете их, и только в самом конце помещаете их в некоторый класс. К моменту, когда вы его объявите, весь его функционал уже будет релизован.

Однако надо мириться и с некоторыми ограничениями, основное из которых – это область видимости переменных.

Предположим, используя REPL, вы будете реализовывать классы сверху–вниз. Таким образом, любая объявленная вами переменная, которая в будущем будет использоваться в нескольких функциях будет глобальной. Это чревато тем, что поведение функций будет зависеть друг от друга – такого быть не должно, однако и решается такой вопрос довольно легко, вам просто необходимо реализовывать функционал снизу вверх, таким образом вы:

- Объявляете переменные, которые будут использоваться в функции – сейчас они глобальные
- Реализуете функцию
- Далее перед вызовом функции замыкаете переменные, по существу, передавая их скопированные значения в функцию – ура, теперь они локальные

Прочитать подробнее:

1. Small Talk REPL
2. Clojure REPL

5.5 Unit tests vs. Integration tests

Данный пример демонстрирует, что необходимо проводить как unit тесты, так и intgration тесты.

```
def f1(a, b):
    return a / b

def f2(a, b):
    return a * b

def f1_2(a1, b1, a2, b2):
    return f1(a1, b1), f2(a2, b2)
```

Пусть тип входных аргументов это signed byte, числа от -2^7 до $2^7 - 1$.

Тогда для проверки отдельных компонентов f1 и f2 (модульное тестирование), потребуется проверить.

- $2^8 + 2^8 = 2^9$ комбинаций.

Для полной проверки их взаимодействия f1_2 (интеграционное тестирование), потребуется проверить.

- $2^{8+8} = 2^{16}$ комбинаций.

То есть, в интеграционном тестировании ОДЗ представляет собой комбинацию из ОДЗ компонентов. Граничных случаев получается больше, и проверить их сложнее.

При модульном тестировании комбинаций входных аргументов меньше, и добиться высокого покрытия легче. Тем не менее, только им ограничиться не получится, потому что оно не позволяет проверить, корректно ли модули взаимодействуют друг с другом.

Таким образом, эти тесты решают разные задачи, и их нужно комбинировать, а не использовать по–отдельности.

5.6 Test coverage

Метрикой качества динамической верификации служит тестовое покрытие. Чаще всего, под этим понятием подразумевают процентное соотношение строк кода, которые выполняются во время тестов, к общему объёму.

Однако при подобном способе оценки даже 100% покрытие не гарантирует качественного тестирования.

В приведённом ниже примере, абсолютно все строки функции будут хотя бы единожды выполнены в тестах благодаря первым трём `assert`. Тем не менее, оставшаяся комбинация входных аргументов, `True, True`, приводит к возникновению ошибки.

```
def f(a: bool, b: bool):
    lst = [1, 2]
    i = 0
    if a: i+=1
    if b: i+=1
    return lst[i]

assert f(False, False)
assert f(False, True)
assert f(True, False)
# here we have full test coverage for fucntion f
# but:
assert f(True, True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in f
IndexError: list index out of range
```

Выходом из подобной ситуации служит использование другой метрики – покрытие путей исполнения (path coverage).[2] В этом случае, в тестах требуется пройти все возможные пути исполнения программы, которые возникают при ветвлении.

Благодаря продвинутым техникам программного анализа, вроде символьного исполнения, возможно автоматически найти небольшой набор комбинаций входых аргументов, проверка которого обеспечит 100% path coverage.

Но когда дело касается нетривиальных программ, содержащих вложенные циклы, или циклы, число итераций которых зависит от переменных, проверка всех путей не представляется возможной.

Получается, что простые метрики дают недостаточно информации о качестве проведенного тестирования, а сложные – слишком тяжело рассчитать.

Таким образом, ориентироваться исключительно на метрики не стоит. Вместо этого, следует отдать предпочтение вдумчивому написанию тестов, с опорой на структуру исходного кода и анализ предметной области.

Разумеется, это потребует дополнительных усилий и времени. Однако при некачественном тестировании потери всё равно будут больше, поскольку вносить исправления придётся на более поздних этапах работы.

5.7 Doctest

Doctest – один из видов автоматического тестирования основанный на документации. При его использовании, документация содержит:

1. Основное текстовое содержание
2. Примеры использоваия **в виде испольняемых тестов**

Пример на языке Python:

[2] Black Box and White Box Testing Techniques – A Literature Review

```
def factorial(n):
    """Return the factorial of n,
    an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0
.........................................
```

Далее запустим проверку документации нашего файла при помощи утилиты:

```
$ python -m doctest -v example.py
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

Данный вид тестирования необходим, чтобы поддерживать документацию в актуальном состоянии. Предположим, вы написали код с использованием библиотеки. Вы запускаете ее тесты, но некоторые из них не проходят. Вы не понимаете, на чьей стороне проблема – либо ошиблись вы, либо документация устарела.

Однако с doctest'ами устаревшие примеры в документации приводили бы к ошибке при сборке, вынуждая разработчика их обновить.

Прочитать подробнее:

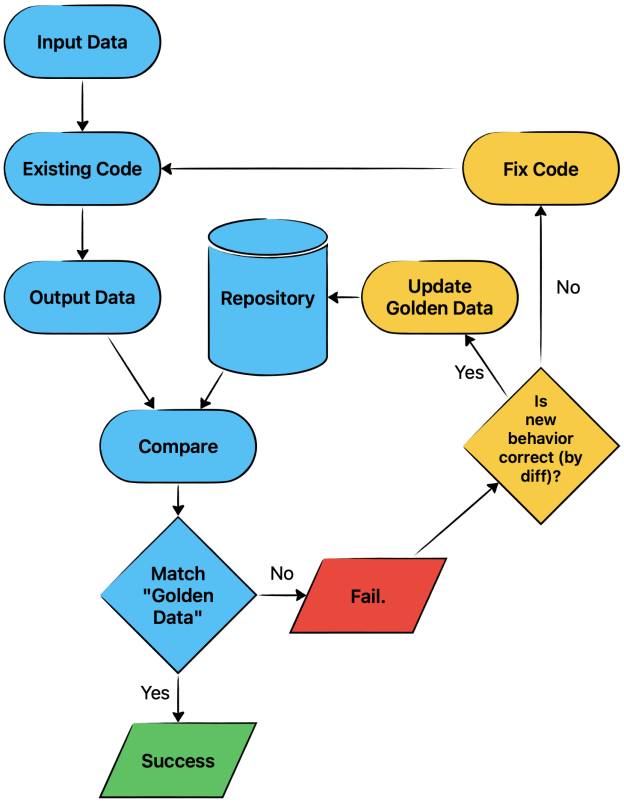
1. Haskell doctest
2. Python doctest

5.8 Golden Master Testing

Golden тестирование чем–то похоже на Doctest. Оно является подвидом регрессионного тестирования: давайте будем сохранять **золотой** вариант работы нашей программы, с которым в будущем мы сможем сравнивать результат работы последующих, измененных версий программ.

Особенности данного метода:

1. Если вывод программы очень большой, будет неудобно вставлять весь этот код в исходники некоторого unit теста. Поэтому вывод сохраняется в отдельный файл
2. Он позволяет работать с выводами программы, реализованными в любых форматах. В частности – с форматами, которые люди не могут прочитать, например, бинарниками



Для работы golden test указываются входные данные и эталонный результат. Конфигурация тестовых случаев обычно задаётся с помощью языков разметки, например, yaml.

Во время тестирования, входные данные из голден тестов подаются на измененную программу. Полученный результат сравнивается с эталонным. Вот, что может произойти дальше:

Выходные данные не совпали:

- Если мы уверены, что они должны были совпасть, тогда в нашей программе, с последнего обновления голден теста присутствует ошибка. Поправим ее.
- Может возникнуть и другая ситуация: эталон стал уже неактуален, тогда как мы уверены, что новая версия программы работает корректно. В этом случае, в файлах обновляется сам эталонный результат.

Выходные данные совпали: это значит, что наша программа до сих пор реализует функционал эталона.

Прочитать подробнее:

1. Python golden tests
2. Haskell golden test – неплохая статья про реализацию Golden test'ов на Haskell.

5.9 Fuzzy and Monkey Testing

Входные данные некоторых систем могут быть крайне вариативны, что вызывает трудности при тестировании.

Взять, например, компилятор. Множество исходных программ, которые он должен обрабатывать, безгранично. Как убедиться в том, что он работает корректно? Какие

тестовые случаи рассматривать?

В подобных ситуациях, когда необходимо протестировать большое количество комбинаций чего–либо, применяется **фаззинг**.

> The term “fuzz” was originally coined by Miller et al. in 1990 to refer to a program that “generates a stream of random characters to be consumed by a target program.

В общем смысле этого слова, фаззинг описывает процесс автоматического тестирования программы «мусорными данными». То есть, приоритет отдаётся объёму и разнообразию тестов, а не их точности.

Легче всего использовать фаззинг для penetration testing. В этом случае, в процессе фаззинга входные данные намеренно искажаются. Они слегка «выходят за рамки» значений, ожидаемых программой. При запуске программы на этих искажённых, невалидных данных, проводится проверка того, что они были успешно отвергнуты программой в результате внутренних проверок и обработки.

Однако можно тестировать и более сложные свойства. Для этого в фаззерах задают «оракул» – модуль, который проверяет их соблюдение.

Generation–based (или model–based) **фаззеры** генерируют входные данные, основываясь на модели, которую им предоставили. Например, для компилятора generation–based фаззер будет генерировать программы, основываясь на грамматике языка (grammar–based input generation), заданной в форме EBNF (Extended Backus-–Naur Form). Зачастую фаззеры предоставляют API, благодаря которому тестировщик может задать требуемую модель.

Mutation–based фаззеры, не требуют модели входных данных. Вместо этого, они меняют уже готовый набор входных данных.

Ограничением random testing является то, что сгенерированные данные лишь с небольшим шансом затронут интересующие нас области программы, поскольку они могут быть отвергнуты в результате внутренних проверок.

Без дополнительного анализа, сложно определить требуемую структуру входных данных. Например, как сгенерировать валидный mp3 файл с нуля? Здесь можно обратиться к принципу локальности: если входные данные валидны, то, изменив их небольшую часть, мы всё равно с большой вероятностью получим валидные данные. Таким образом, специфика mutation–based фаззеров заключается в том, что предоставленные примеры входных данных меняются лишь слегка. Например, инвертируются несколько бит.

Источники:

1. The Art, Science, and Engineering of Fuzzing: A Survey
2. A Survey of Modern Compiler Fuzzing

Monkey Testing – разновидность random testing, которая сильно напоминает фаззинг. Суть заключается в том, что к тестируемой программе применяется последовательность случайных действий, с намерением её сломать.

Можно провести аналогию с обезъяной, которая лазает по тестрируемой системе и беспорядочно жмёт на все кнопки.

Эта техника подходит для распределённых систем. В таком случае, отключаются произвольные куски системы, микросервисы, а затем проверяется её работоспособность.

5.10 Race Condition Detection

Одна из проблем многопоточного программирования – условия гонок – Race Condition. Если несколько потоков выполняются одновременно, мы не можем определить, в каком порядке будут выполнены инструкции на процессоре.

Рассмотрим небольшой пример на языке программирования Go:

```
func main() {
    i := 0
    go func() {
        i++         // main.go:7
    }()
    fmt.Println(i) // main.go:9
}
```

Здесь функция **func()**, которая увеличивает счетчик **i** на 1, вызывается в новом потоке (Goroutines).

Мы не можем точно предугадать, что произойдет раньше: `i++` или `fmt.Println(i)`, из–за чего результат вывода на экран не определен. Посмотрим на **Race Condition detector**, который реализован для языка Go:

Запустим программу с ключом **-race**:

```
go run -race racy.go
```

И получим следующий результат:

```
WARNING: DATA RACE
Write by goroutine 6:
  main.main.func()
      /tmp/main.go:7 +0x44

Previous read by main goroutine:
  main.main()
      /tmp/main.go:9 +0x7e

Goroutine 6 (running) created at:
  main.main()
      /tmp/main.go:8 +0x70
```

Как видите, этот механизм подсказывает нам, что в нашей программе существует **DATA RACE**, в данном случае, внутри **main.main.func1()** значение используется с целью **WRITE**, то есть изменяется, в то время как в **main.main()** значение используется для **READ**, то есть считывается.

Работает данный механизм относительно легко:

- Сначала он проходится по коду и запоминает все ситуации, когда процесс потенциально может повести себя по–разному из–за диспетчеризации потоков – в данном примере он увидит функцию **func()**, которая запускатеся в другом потоке при помощи ключевого слова go.
- Далее он проходится по всем возможным трассам доступа к данным – то есть по всем путям, в которых в зависимости от диспетчеризации, инструкции чтения или записи данных могут быть поменены местами. Для каждой трассы он рассматирвает порядок чтения–записи данных. Если он находит две трассы, в которых результат выполнения одной и той же функции различается, он дает об этом знать.

Данное решение требует больших вычислительных и временных ресурсов для нетривиальных программ. Однако стоит заметить, что от перебора всех возможных случаев отказаться не получится: нельзя пропустить ни одну трассу работы с данными – вдруг именно в ней произойдет Race Condition.

Прочитать подробнее:

1. Introducing the Go Race Detector

5.11 Property–Based Testing

Property–Based Testing развился из идей Fuzzy/Monkey Testing. Как можно понять из названия – это тесты, основанные на свойствах объектов, функций или структур данных. Идея следующая: вместо того, чтобы формально доказать свойство системы, мы эксперементально убедимся в нем на большем наборе тестов.

Предположим, вы написали некоторую функцию, тогда вы можете сформулировать для данной функции свойства – то есть некоторые утверждения, которые в широком смысле слова описывают её поведение. Примеры свойств:

- **Коммутативность сложения чисел**:

Пусть функция `add(x, y)` возвращает сумму двух, поданных на вход чисел, тогда одно из свойств, которое точно должно быть выполнено для корректного сложения – это коммутативность, то есть `add(x, y) == add(y, x)`.

- **Свойства Моноида**:

Пусть `<S, *, e>` – Моноид на множестве S, с заданной бинарной операцией *, и нейтральным элементом = e. Тогда:

1. Для любых a, b, c из S: (a * b) * c = a * (b * c)
2. Для любых a из S: a * e = e * a = a

Примеры моноидов: - `<str, ++, "">` – все возможные строки относительно операции конкатенации и пустой строкой в качестве нейтрального элемента. Свойства довольно очевидны – в какой бы последовательности вы не складывали строки, вы всегда получите одно и то же. Так же, добавление пустой строки к любой другой слева или справа, очевидно, ничего не меняет. - `<set, union, empty set>` – все множества с операцией объединения и пустым множеством в качестве нейтрального элемента.

- **Раскраска и балансировка red–black tree**:

Следующий пример демонстрирует, зачем вообще тестировать свойства: предположим, вы реализуете множество как структуру данных. Конечный пользователь не будет знать, как именно ваша реализация будет работать “под капотом”, однако ему известны:

1. Интерфейс взаимодействия – в множество можно положить элемент, убрать элемент, проверить наличие элемента.
2. Заявленое время выполнения операций – например, если вы реализуете множество на основе красно–черных деревьев, пользователь уверен, что время выполнения операции вставки имеет асимптотику: `O(log n)`, где n – количество элементов в множестве.

Если первое свойство проверяется обычными юнит тестами, то второе требует особого подхода. Так, если вы раскрасите вершины красно–черного дерева неправильно или неправильно будете производить добавление (то есть так, что свойства этого дерева будут нарушены), асимптотика операции вставки не будет соответствовать ожиданиям пользователя. Воспользуемся Property–base тестингом! Введем свойства красно черных

деревьев, сгенерируем деревья и проверим их. Таким образом, мы будем уверены не только в том, что множество поддерживает операции вставки, удаления и поиска, но и в том, что красно–черное дерево лежащее в основе реализации будет сбалансированным.

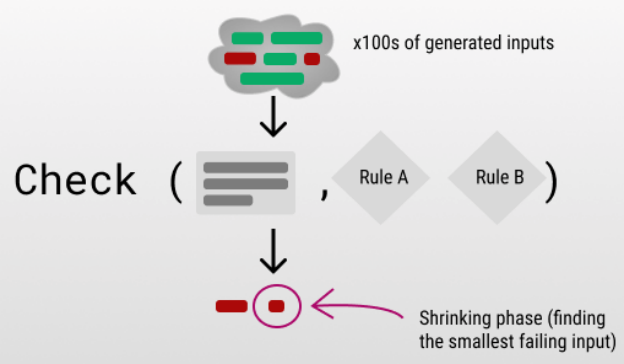
- **Эквивалентность исходной и оптимизированный программы**:

Пусть мы написали два компилятора, один из которых будет компилировать код без оптимизаций, а второй – с оптимизациями. Утвердим свойство – программа, скомпилированная двумя различными компиляторами должна одинаково исполняться – то есть на одни и те же входные данные, она должна выводить одни и те же выходные. Проверка данного свойства обеспечит эквивалентность исходной и оптимизированный программы.

5.11.1 Этапы Property–Based Testing

Данное тестирование имеет три основных этапа:

1. Генерация исходных данных
2. Проверка условий
3. Сжатие входных данных в случае нарушения свойств



Генерация данных должна быть автоматизирована, потому что тестирование должно производиться десятками тысяч тестов.

Рассмотрим **сжатие (shrinking)** входных данных в случае нарушения свойств:

Пусть мы нашли тест, который ломает нашу программу и пусть в нем содержится, например, массив из 10000 целых чисел. Продебажить этот тест, или тем более рассмотреть его вручную – непосильная задача.

Именно поэтому библиотеки property–base testing'а определенным способом "сжимают" этот контр–пример, чтобы система все так же ломалась на нем, но он был меньше по объему. Стоит лишь отметить, что это не всегда возможно.

5.11.2 Примеры property–base тестов на основе QuickCheck (Haskell)

В данной программе проверяется простое свойство функции reverse, которая разворачивает массив:

```
import Test.QuickCheck
import Data.List (intersperse)

prop_revapp :: [Int] -> [Int] -> Bool
prop_revapp xs ys = reverse (xs ++ ys) == reverse ys ++ reverse xs

main = quickCheck prop_revapp
```

В данной программе тестируются свойства split и unsplit:

```
import Test.QuickCheck
import Data.List (intersperse)

split :: Char -> String -> [String]
split c [] = []
split c xs = xs' : if null xs'' then [] else split c (tail xs'')
    where xs' = takeWhile (/=c) xs
          xs''= dropWhile (/=c) xs

unsplit :: Char -> [String] -> String
unsplit c = concat . intersperse [c]

prop_split_inv xs
    = forAll (elements xs) $ \c ->
        unsplit c (split c xs) == xs

main = quickCheck prop_split_inv
```

Запуск тестов реализуется с помощью вызова quickCheck. библиотека QuickCheck автоматические сгенерирует входные данные, протестирует и сожмет входные данные, если найдется непроходимый тест.

Во второй программе QuickCheck корректно найдет ошибку:

```
*** Failed! Falsified (after 2 tests and 1 shrink):
"a"
'a'
```

Она возникает, потому что split и unsplit не совсем корректны:

```
split 'a' "a" = [""]
unsplit 'a' [""] = ""
```

5.11.3 Достаточно ли Property-base тестинга для проверки всей системы?

Ответ: **нет**

Дело в том, что проверка свойств системы проверяет исключительно **свойства**, но не **поведение**, так, свойства могут быть выполнены и у некорректной функции.

Пример на основе **hypothesis (python)**:

```
from hypothesis import given
import hypothesis.strategies as st

def bad_add(a, b):
    return 0

@given(st.integers(), st.integers())
def test_ints_are_commutative(x, y):
    assert bad_add(x, y) == bad_add(y, x)
```

Функция bad_add всегда возвращает 0, хотя должна складывать числа. Однако, как мы можем заметить, свойство коммутативности выполнено:

```
bad_add(x, y) = 0 = bad_add(y, x)
```

Таким образом, использование property–base тестов не исключает необходимости написания unit тестов.

Прочитать подробнее:

1. Моноид
2. QuickCheck (Haskell)

6 Статическая верификация. Типизация

6.1 Предисловие

В рамках данного раздела мы попытаемся рассмотреть системы типов, откуда они появились и в чем их суть, но сразу стоит сказать, что мы рассмотрим их в достаточно абстрактном представлении, так как настоящая система типов – это большое количество математики, математической логики, выводов и прочего. Рассмотрим системы типов в контексте того: что они регулируют и в чем их основная задача?

6.2 Статическая и динамическая верификация

Давайте вспомним, что мы обсуждали ранее: **динамическая верификация**. Фактически, это проверка того, что выдаст система в зависимости от данных, которые мы ей передадим. И в зависимости от этих данных, мы получим либо правильный, либо неправильный ответ.

Но что же представляет из себя статическая верификация кода?

Она объединяет в себе синтаксис и семантику. То есть, такая верификация возможна без компиляции исходного кода и работы с ним в рантайме. По таком принципу работают линтеры, которые позволяют выявлять не только стилистические недочеты, но и возникновение ошибок во время выполнения программы, также к статической верификации относится еще одна технология, без которой пользование современными языками программирования и системами стало невозможным – система типов – о ней и пойдет речь в рамках этого раздела.

6.3 Системы типов в компьютере и их происхождение

6.3.1 Машина Фон-Неймана

В самом начале развития компьютеров, не было понятия типизации данных. Любая информация представляла из себя набор 0 и 1. В простом представлении это выглядело примерно так:

- У нас есть какая-то информация, которая лежит в памяти.
- У нас есть набор функций, который производит операции над значениями, лежащими в памяти и ему абсолютно все равно, что это за значения, он просто берет их и преобразует так, как мы захотели.

В чем удобство такого подхода:

- Язык общения с машиной однозначно говорит о том, что мы можем делать с информацией.
- Позволяет произвести абсолютно любое действие, которое мы захотим: если мы считаем, что так можно сделать, компьютер это сделает.
- Программист избегает лишних шагов при работе с данными.

Но также стоит перечислить и недостатки отсутствия абстракций над данными:

- Может привести к ошибкам их обработки, что напрямую сказывается на консистентности результатов.
- Делает код подверженным различным уязвимостям, таким как: перезапись адресов возврата из функций и прочим.

Самый простой пример ошибки, при работе с данными без абстракций их представления – это складывание числа типа float и числа типа int, что в конечном итоге приводит к неожиданному результату.

6.3.2 Тегированные ЭВМ

Следующим шагом развития способов хранения данных стал уход от идентичности всех данных к добавлению нескольких дополнительных бит для тэга, который указывает на тип данных: `<tag><actual-data>`

Плюсы данного подхода в том, что мы уже ограничиваем возможность протечки абстракций и теперь, для того, чтобы поменять тип данных(поменять тег), мы должны явно это продекларировать.

Минусы: большое количество дополнительных расходов памяти на теги.

Еще стоит помнить о том, что ошибки привидений типов, также могут привести к неожиданному поведению программ.

6.3.3 Интерпретатор, ВМ

По сути, то же самое, что и предыдущий этап. Просто процессор заменен виртуальной машиной.

Как можно заметит, путь, пройденный в сторону типизаций в рантайме, очень большой, но имеет много противоречивых вопросов, так как любая проверка типов в рантайме – дополнительные накладные расходы.

Отсюда вопрос: а нужна ли вообще типизация в реальном времени?

Может, будет лучше, если на уровне процессора будет архитектура Фон-Неймана, а типы и прочее мы будем проверять уже на уровнях выше: в виртуальной машине или умным транслятором?

6.4 Indirection (косвенная адресация)

Поднимаясь по уровням абстракций, мы перестаем производить конкретные действия над данными, накладывая на них абстракции и начиная работать с ними в обобщенном виде. А типизация, в свою очередь, накладывает свои ограничения на те данные, относительно которых осуществляется косвенная адресация (indirection).

Приведем пример кода:

```
a = 5
b = 7
print(a+b)
```

В данном случае, у нас есть 2 переменных: a и b – и когда мы выполняем операцию сложения, на них накладываются ограничения, что они должны иметь конкретные типы, которые можно сложить.

Теперь, когда мы знаем немного больше о косвенной адресации данных, давайте поднимемся еще выше по лестнице абстракций и познакомимся с объектами первого класса.

6.5 Объекты первого класса

Фактически, когда мы начинаем работать с любым языком программирования, мы достаточно быстро приходим к мысли о том, что наши возможности так или иначе ограничены объектами, которые предоставляют базовые операции на уровне языка. Тут появляется такое понятие, как **“объекты первого класса”**, они характеризуют те возможности языка, которые являются базовыми и которые можно использовать.

Что является их особенностями?

- Все объекты первого класса могут быть параметрами функции.
- Все эти объекты могут быть возвращены, как результат работы функции.
- Эти объекты можно использовать в правой части выражения присваивания значения переменной.
- Возможность проверки на эквивалентность двух значений одного типа.

Все объекты, которые не подходят под эти 4 пункта надо выбросить из рассмтрения, так как они ограниченно поддержаны со стороны языка!

Примеры объектов первого класса:

- Значения и ссылки
- Указатели на функции

Не являются объектами первого класса:

- Макросы в C/C++
- Метки в различных языках программирования

6.6 Типизация на уровне языка. Происхождение

Давайте посмотрим на типизацию с точки зрения языка программирования.

Какими способовами можно ее реализовать?

6.6.1 Первый вариант: Специфицирование

Как оно работает?

На самом деле, достаточно примитивно. Мы фиксируем типы на уровне документации, а дальше проверяем, что все вызовы корректны с точки зрения заложенной программистом спецификации.

Такое можно встретить во многих современных динамических языках программирования: python, erlang, elixir и прочих.

6.6.2 Второй вариант: все типы известны

В чем идея? Обогащаем функции более сложными иденитификаторами, либо полной информацией о типах.

Пример такого обогащения в C:

```
int add(int, int);

float add(float, float);
```

6.6.3 Третий вариант: обобщенное программирование

В современных реалиях работать с языком без какого-либо полиморфизма - невозможно.

Примеры обобщенного программирования:

- Массивы в языке C – синтаксический сахар поверх машины Фон-Неймана. По сути, у нас есть кусок памяти и указательна его начало, который определяет шаг для перехода к следующему элементу. В результате чего создается впечатление о полноценной поддержки массивов в языке C. Выглядит это как-то так:

  ```
  int a[5] = {0, 1, 2, 3, 4};

  a[3] => *(a + 3 * sizeof(a)) => 3;
  ```

- Параметрические типы(дженерики)
- Динамическая диспетчеризация, когда мы в рантайме не можем понять тип данных, с которым мы работаем(интерфейсы)

6.6.4 Четвертый вариант: ограничения, вывод типов, поиск подходящих вариантов

Этот подход заключается в том, что мы ставим компилятор логическую задачку, чтобы он сам нашел тот тип, который в должен быть в конкретном месте. Это не делает пользователь, это происходит в рантайме.

6.7 Виды системы типов

Для начала, введем определение **системы типов**.

Система типов – это совокупность правил в языках программирования, назначающих свойства, именуемые типами, различным конструкциями, составляющим программу – такие как переменные, выражения, функции и модули.

Думаю, вы уже поняли, что в языках встречаются различные виды типизации. Но как мы можем сравнивать их между собой?

Для этого обратим вниамание на следующую картинку:

На ней представлены 2 оси координат, в рамках которых представлены: динамическая – статическая, сильная – слабая типизации.

Но что же это значит?

Давайте разберемся на конкретных примерах:

- **Сильная типизация:** если в коде указано, что тип данных Integer, то мы не можем применять операции, которые связаны с Float, Long и прочими типами данных. В рамках данной типизации требуется больше кода от программиста, но ниже риск падения в рантайме из-за неожиданного поведения.
- **Слабая типизация:** яркий пример слабой типизации – JavaScript, когда рантайм пытается привести типы для выполнения операции. С таким видом типизации от программиста требуется значительно меньше кода, но повышаются риски неожиданного поведения.

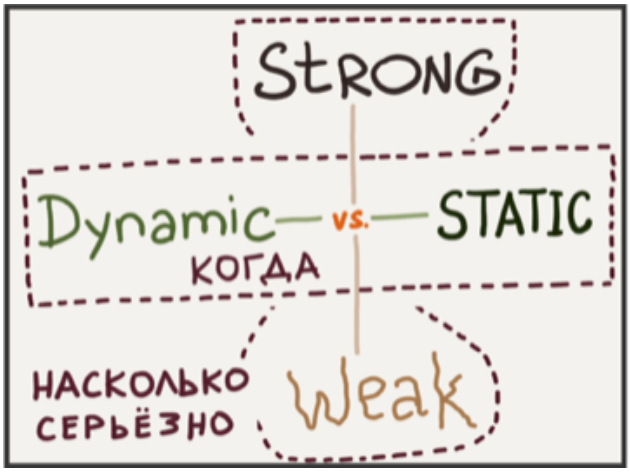


Figure 15: Виды типизаций, как оси координат

- **Динамическая типизация:** все типы узнаются только в рантайме, какие данные попадут в конкретный кусок кода, такие и будут.
- **Статическая типизация:** все типы известны на момент компиляции и не меняются.

Примеры видов типизаций и языков:

- Динамическая сильная – Python
- Динамическая слабая – JavaScript
- Статическая сильная – C, Haskell, Ocaml
- Статическая слабая – Java, C++

6.8 Система типов и множества (поверхностно)

Перед тем, как начать говорить о более предметых вещах, нужно определиться: как взаимодействуют код и система типов? Как представить типы через множества?

- Singletons: $A = \{a\}$, $B = \{b\}$ – те множества, которые могут состоять из одной переменной.
- Booleans: $\{true, false\}$ – имеют перечисленное количество значений.

  ```
  bool flag = true;
  ```

- Natural – натуральные числа зачастую определены на уровне железа, а не языка.
- enum: $\{a, b\}$ – позволяет перечислять значения.

  ```
  enum Status{
      OK,
      VALIDATION_ERROR,
      FORBIDDEN
  };
  ```

- struct: $\{A, B\}$ – с точки зрения теории множеств является декартовым произведением вида: $A \times B = \{(a, b)\}$. То есть это пары всех возможных пар значений из множеств A и B.

```
struct data {
    int a;
    long b;
};
```

- union: $\{A, B\}$ – позволяет использовать все из множества A и все из множества B воедино: $A \cup B = \{a, b\}$. Отличный пример такого типа можно найти в языке C:

  ```
  union data = {
      int a;
      char b[4];
  };
  ```

  То есть мы можем в рамках одного и того же union поместить либо одно значение типа int, либо 4 значения типа char

- Generics: `class Box<T>{T value;}`
  - $\equiv$ `data Box a = Box a`
  - $\equiv box(T) = T \times \{box\}$
- Disjoint union: `data Either a b = Left a | Right b`
  - $\equiv either(A, B) = (A \times \{left\}) \cup (B \times \{right\})$
- Recursive type: `data List a = Cons a (List a) | Nil`
  - $\equiv list(A) = (A \times list(A) \times \{cons\}) \cup \{nil\}$
- `data Maybe a = Just a | Nothing`
  - $\equiv maybe(A) = (A \times \{just\}) \cup \{nothing\}$
  - `fromMaybe` — unbox;
  - `int->int->int` — use.
- `int?`=$N \cup \{nil\}$
  - `int?->int?->int?`
    - run-time exception
    - cast `nil` to `0` or `1`
    - `nil` propagate `nil` (`nil + x = nil = x + nil`)

Тут приведены более сложные примеры.

Generics: первый пример – это java с классом Box, в который мы можем передать значение произвольного типа T. Второй пример написан на Haskell. Слева располагается вызов функции, которая может иметь одно из разных значений, то есть происходит связь системы типов с типом уже на уровне данных. С точки зрения теории множеств, это можно записать как box(T)(некоторая функция), которая является синтаксическим сахаром, обёрткой вокруг объединения любого значения переменной T в пару с box. Фактически получается метка для каждого типа, что это Box T. Это явление называется боксингом и анбоксингом значений. Именно по этому данная операция может просаживать производительность, во многом это относится к динамической типизации, потому что, приходя к типу данных, вы сначала смотрите, промаркирован ли правильным образом данный тип, далее убираете маркировку и только после этого получаете доступ к значению. По большому счету, с точки зрения теории множеств формирование generics – это декартово произведение с тегом, которым вы хотите все пометить.

Disjoint union(размеченное объединение): пример – это either, параметризированный тип, в который можно подать значение как из множества A, так и из множества B. При этом нужно четко отметить, из какого множества эти значения. Классическим

примером может являться код, в котором при отсутствии ошибок будет выбрано множество A, в противном случае – B. Это позволяет в случае корректной работы поочередно, от строки к строке выполнять код и в конце возвращать результат, или в случае ошибки прокидывать первичное сообщение об ошибке до конца и возвращать из данных. С точки зрения теории множеств мы определяем функцию уровня системы типов и говорим о том, что множество `either(A, B)` – это объединение, двух множеств, где A промаркировано тегом `left` , а B – тегом `right`. Это позволяет на уровне системы типов определить, что мы содержим либо первое, либо второе, что в свою очередь это дает большое количество возможностей как для дизайнера языка, так и для программиста. Статически типизированные языки использую Disjoint union в самых разных видах: case-типы, алгебраические типы и т.д.

Recursive type: классическое определение односвязного списка. У нас есть пара, имеющая или не имеющая указателя на следующий элемент и хранящая значение текущего элемента в списке. С точки зрения теории множеств, мы имеем конструктор типа от A, которое формирует объединение множества с элементом `nil` и рекурсивной структуры элемента множества списка (сам элемент A, список элементов A и тег пары). Используя эту нотация можно описать любой тип списка, даже бесконечный.

Maybe: классический пример работы с пустотой в языке программирования. Мир функционального программирования решил эту проблему так, что у нас с одной стороны есть множество значений, а с другой маркер, когда его нет. Далее наш код разбивается на две части: те моменты, где мы работаем с типом Maybe и пытаемся определить, есть значение или его нет, а также на те, где работа идет с уже определенными значениями. С точки зрения теории множеств мы помечаем наши значения тегом `just` либо используем тег пустоты `nothing`. Получается комбинация функции `fromMaybe`, которая извлекает значение, либо падает с ошибкой, либо подставляет какое-то значение по умолчанию, и чистой функции которая работает со всегда готовыми значениями `int`.

`nil`: подход позволяет избежать работы с типами, наличие которых нам неизвестно, и просто проверять наличие значения. В случае наличия возвращается значение, в случае отсутствия – nil (null). Выбор лучшего варианта крайне неоднозначен и дискуссионнен, но с точки зрения теории множеств это совершенно разные истории. Maybe предоставляет нам размеченное множество, которое нужно разобрать, понять что там лежит и только после этого, продолжать работу. В этом же подходе, мы объединяем множества, то есть можем иметь либо число, либо nil. Это означает, что в случае если мы на уровне языка работаем с nilable данными, то мы обязаны все наши функции писать так `int?->int?->int?`, фактически мы должны каждый раз ставить под сомнение те данные, которые нам передали. И здесь возникает вопрос с точки зрения дизайна языка: что делать если пришли странные данные, например, при сложении двух чисел? Первый вариант – это сгенерировать `runtime exeption`. Вариант второй, мы приводим nil к integer значению. Третий вариант, если хотя бы один из элементом является nil, то мы возвращаем nil. Выбор варианта зависит сугубо от требований, которые мы выдвигаем к языку, если нужен гибкий язык, то вариант с опциональными nilable типами будет подходящим, если это статический язык, то, как правило, выбирается Maybe.

6.9 Неявная статическая типизация

В современных языках программирования явное прописывание всех типов - это редкость. Все языки, используя различные средства, могут самостоятельно их определять. **Вывод типов (type inference)** - это возможность компилятора (интерпретатора) самостоятельно логически выводить и указывать тип данных на основе анализа выражения.

Пример 1:

```
func main() {
 var x float32 = 123.78
 y := 23
 fmt.Printf("%T", y + x)
} // invalid operation: y + x (mismatched types int and float32)
```

Примечание: запрещено сложение переменных типа int и float, у переменной y тип `int` по умолчанию.

Пример 2:

```
func main() {
 var x float32 = 123.78
 y := 23 + x
 fmt.Printf("%T", y)
} // float32
```

Примечание: за счет переноса операции суммирования, компилятор способен самостоятельно определить осуществить неявное преобразование переменной из типа `int` во `float`

Пример 3:

```
map f [] = []
map f (x:xs) = f x : map f xs
-- map :: _ -> _ -> _
-- map :: _ -> [_] -> _
-- map :: _ -> [_] -> [_]
-- map :: (_ -> _) -> [_] -> [_]
-- map :: forall a.
--        (a -> _) -> [a] -> [_]
-- map :: forall a b.
--        (a -> b) -> [a] -> [b]
```

В примере 3 написана программа, реализующая отображение, приводящее функцию f к каждому элементу списка.

1. Объявлена функция с двумя аргументами и одним результатом
2. Отмечаем тип второго элемента – список
3. Отмечаем, что результат также является списком
4. Отмечаем, что в качестве первого аргумента используется функция, которая должна получить один аргумент и что-то вернуть
5. Используем переменную типа a, что символизирует собой абстрактный тип, для определения конкретных типов в нашем отображении. Не трудно заметить, то первый элемент списка подается на вход функции f, что помогает заключить, что на второй позиции должен стоять список `[a]`.
6. Добавляем переменную b, которую подставляем в получившееся на 5-ом шаге выражение

Примечание: данный пример являет крайне простым и не касается сложных деталей и нюансов.

6.10 Тотальность функции

Тотальная функция (total function) – это функция, определенная для всех возможных входных данных.

Примеры:

```
(/) :: Int -> Int -> Int
_ / 0 = error "divide on ZERO"
a / b = ...
```

В примере представлена функция деления, которая by-design на целых числах. Фундаментальная проблема, связанная с этой операцией, – это, конечно же, деление чисел на `0`. Сделать в ней мы ничего не может, поэтому приходится всегда проверять значение знаменателя на неравенство `0`.

Есть и более искусственные проблемы. Например, head — взятие головы от списка. Варианты типизации:

```
head1 :: [a] -> a
head2 :: [a] -> Maybe a
head3 :: NonEmpty a -> a
head4 :: [a] -> a + {nil}
```

1. Тип по умолчанию. Его главной проблемой является пустой список. В случае, если мы попытаемся взять голову, будет выброшено исключение, сообщающее о попытке взятия списка от пустого списка.
2. Тип `Maybe`. У данного варианта есть недостатки в виде большого количества boiler-plate кода, который нужен вокруг этого, но он в свою очередь дает гарантии того, что вы получите именно, то что нужно.
3. Тип непустого списка. В отличии 2 варианта мы выворачиваем структуру наоборот, гарантируя возможность взятия головы, но только одну, так как `NonEmpty` – это гарантия наличия лишь одного элемента, но не большего количества. То есть свойство применяется не в выходному, а ко входному списку. Свойства 2 и 3 хороши тем, что вы в компайлтайме знаете где функция применима, а где – нет.
4. Nilable значения. Мы достаем либо значение, либо `nil`. Но у данного варианта имеется проблема, связанная с возможностью нахождения `nil` в списке. Все зависит от конкретной ситуации, инфраструктуры и от того, с кем вы интегрированы. Но ответив на вопрос по типу: А может ли список состоять из одного `nil`? -, вы сможете если не решить проблему, то убедится в необходимости использования более строгих 2 или 3 свойств. То как мы хотим работать с `nil`, зависит от того, делаете вы язык со статической или динамической типизацией.

Maybe Not – Rich Hickey

6.10.1 Пример проблемы интерпретации типов

Из описания формата JSON:

“‘number in JSON number integer fraction exponent integer digit onenine digits ‘-’ digit ‘-’ one-nine digits

digits digit digit digits

digit ‘0’ onenine

```number in JSON
```

```
onenine
    '1' . '9'

fraction
    ""
    '.' digits

exponent
    ""
    'E' sign digits
    'e' sign digits
```

Что такое тип? Тип – это множество допустимых значений. Приведенный выше пример – это выдержка из стандарта JSON, которая говорит о том, что такое `number`.

`number` – это integer(целое число), fraction(составляющие части, которых может не быть или написанные через “.”), а также exponent(для записи чисел в экспоненциальном формате через e или E). Стоит отдельно остановиться на integer.

integer определен так, что начинает с цифры(от 0 до 9), с цифры от 1 до 9 или с минуса.

У данного определения типа `number` присутствует большая проблема: данное определение поддерживает сколько угодно большое значение, поэтому по умолчанию это должен быть bigInt, но практика говорит о том, что int более предпочтителен для этого. bigInt требует сложных вычислений, сложных структур данных и для взаимодействия с памятью тоже требуются более сложные алгоритмы. Но int не является панацеей, так например в C есть integer, но четкого понимания, что он из себя представляет – нет. По-хорошему у нас есть short(половина integer) и long(двойной integer), это связано с размером машинного слова. Но на деле получается так, что только конкретный компилятор может сказать, что же за типы перед вами. В современных языках абстракция над целыми числами протаскивается в систему типов. Сейчас в области криптографии и блокчейна используются числа в 256 бит, что порождает проблему представления их в памяти компьютера, поэтому может появляться протечка абстракции, ошибочно используется один тип вместо другого.

6.10.2 Контекст

- JavaScript — старая экосистема, разработанная для автоматизации веба “на коленке”.
  - Следствие: работа с `number` как с `long` в большинстве решений.
- Haskell — язык, сделанный любителями и профессионалами от математики.
  - Следствие 1: есть `Integer` без ограничения диапазона значений.
  - Следствие 2: библиотека Aeson (стандарт для работы с JSON) интерпретирует `number` как `Integer`.
- Crypto & Blockchain mess — активное использование ключей.
  - Следствие: числа в 256 бит не предел.

Конкретно в данном примере, проблема решилась достаточно просто: число представляется строкой.

7 Элементы истории языков программирования. Лямбда-исчисление

7.1 Элементы неполной и в основном неверной истории языков программирования

Цель раздела: обрисовать историю развития языков программирования и почему эта индустрия пришла к тому, что мы сейчас имеем.

Охват темы истории языков программирования: кратко и субъективно (исходя из собственного опыта работы и насмотренности). Только общая логика.

Фокус: на элементах функционального программирования.

7.1.1 Машина Тюринга

Машина Тьюринга была предложена Аланом Тьюрингом в 1936 году для определения понятия алгоритма. Сама машина является абстрактным исполнителем и стала, пожалуй, тем концептуальным устройством, которое положило начало такой науке, как теория вычислимости. Она представляет собой абстрактную вычислительную модель, основными элементом которой являются бесконечная лента, на которой записываются данные, головка, которая бегает по ленте и имеет возможность считывать и записывать наборы символов, а также алфавит символов.

Головка машины движется по **ленте** вправо или влево, изменяя данные в соответствии с **программой**, которая находится на выполнении в устройстве. Для работы используется заранее определённый **алфавит**, из символов которого формируются записи на ленте.

Одним из важнейших свойств машины Тьюринга является её полнота по Тьюрингу. Это означает, что любой существующий алгоритм может быть реализован с использованием машины Тьюринга. Таким образом, она стала первым базовым инструментом в теории вычислимости и теоретической информатике.

В контексте универсальных языков программирования свойство полноты по Тьюрингу применяется следующим образом: мы считаем язык программирования универсальным, если он полон по Тьюрингу, то есть на нём можно реализовать любой алгоритм. Такое свойство считается важным именно по той причине, что любой алгоритм может быть реализован на Тьюринг-полном языке, то есть, фактически, любая задача, имеющая алгоритм решения, может быть решена с использованием Тьюринг-полного языка.

Но возникает закономерный вопрос: так ли обязательна полнота по Тьюрингу языку, чтобы этот язык был полезен и хорошо применим на практике для решения конкретных, пусть и не всех на свете, задач? Постараемся ответить на него.

Хотя машина Тьюринга обладает свойством полноты, не все языки программирования, которые полезны и эффективны на практике, должны быть полными по Тьюрингу. Примером такого языка является Agda. В этом языке запрещены циклы, а рекурсия допускается только в форме, которая является структурно корректной. Это означает, что каждая функция и алгоритм, написанные на Agda, гарантированно завершатся, что, как оказывается, делает язык удобным и хорошо справляющимся с задачами верификации программ и доказательства теорем.

Машина Тьюринга не может быть полностью реализована на практике из-за необходимости бесконечной ленты. Однако сама её концепция остаётся важной и для практики, потому что она, тем не менее, является моделью, ориентированной на реализацию. То есть, заменив бесконечную ленту на конечную, мы получаем очень простое вычислительное

устройство, которое способно выполнить любой алгоритм, который останавливается за конечное число шагов.

Одной из ключевых проблем, связанных с машиной Тьюринга, является проблема останова – вопрос о том, может ли программа определить, завершится ли другой алгоритм на заданных входных данных. Эта проблема так и остаётся нерешённой для произвольных алгоритмов, но она, по большому счёту, является больше научной проблемой, нежели чем прикладной.

Кроме того, в концепции машины Тьюринга данные и управление чётко отделены. Код программы существует отдельно от данных и загружается в машину отдельно от данных, что, в целом, отличается от современной вычислительной техники, для которой, в большинстве своём, характерна общая память для данных и инструкций (так называемая архитектура фон Неймана, о ней мы ещё поговорим чуть позже).

7.1.2 Машина фон Неймана

Архитектура фон Неймана – это модель организации компьютерной системы, предложенная американским математиком фон Нейманом в сороковые годы двадцатого века. Машина фон Неймана стала следующим важным этапом в развитии вычислительной техники после машины Тьюринга. Несмотря на важность этой модели, её отличия от машины Тьюинга не столь велики, она не претерпела каких-то очень серьёзных изменений, которые бы сделали её на порядок сложнее, наоборот, она сохранила простоту, при этом получив некоторые доработки, оказавшиеся полезными на практике:

1. Бесконечная лента была заменена адресуемой памятью, где каждая ячейка имеет свой адрес. Это позволило хранить данные в памяти, к которой можно легко обращаться для чтения и записи.
2. Инструкции и данные стали храниться в одной памяти. Это изменение обеспечило упрощение внутренней организации вычислительной машины и сделало проще не только вычислительный процесс, но также и построение самих вычислительных машин и работу с ними, так как позволило упростить и унифицировать память и обращение к ней.
3. Всё остальное в устройстве машины фон Неймана в основном унаследовано от машины Тьюринга. Например, принципы вычислений, управление потоком выполнения и логика операций.

Эти изменения сделали концепцию машины фон Неймана основой для современных архитектур компьютеров, позволили создать универсальные вычислительные системы, какими мы их знаем сейчас, разве что ещё добавились регистры, кэши и некоторые другие новшества, не описанные фон Нейманом в своей концепции, но оказавшиеся полезными и эффективными решениями на практике.

Пример программы для машины фон Неймана:

Комментарий: данная программа вычисляет факториал числа.

Address	Mnemonic	Comment
0400	MOV CX, [0500]	CX <- [0500]
0404	MOV AX, 0001	AX <- 0001
0407	MOV DX, 0000	DX <- 0000
040A	MUL CX	DX:AX <- AX * CX
040C	LOOP 040A	Go To [040A]
0410	MOV [0600], AX	[0600] <- AX
0414	MOV [0601], DX	[0601] <- DX

Address	Mnemonic	Comment
0418	HLT	Остановка исполнения программы

Как можно заметить, данная программа представляет собой последовательность инструкций, записанных в памяти. Машина фон Неймана выполняет их последовательно, обращаясь к памяти для чтения и записи данных. При необходимости машина выполняет манипуляции с данными в соответствии с логикой инструкции.

Этот подход обеспечивает программную гибкость: программист может описывать алгоритмы в виде инструкций, хранимых в памяти, которые управляют данными в адресуемых ячейках. Такой подход стал основой для современных ассемблеров и языков программирования низкого уровня.

7.1.3 Начало эволюции языков программирования

Со временем вычислительная техника начала всё быстрее и быстрее развиваться, богатство и разнообразие архитектур вычислительных машин увеличивалось, появлялись всё более интересные, разнообразные и сложные задачи для компьютеров. Соответственно, и инженерам-программистам становилось сложнее взаимодействовать с компьютерами, обладая малым и даже скудным набором интерфейсов для взаимодействия с вычислительной техникой того времени, а также из-за отсутствия высокоуровневых языков программирования. Писать программы на ассемблере и машинном коде стало банально неэффективно и очень сложно, так возникла необходимость в развитии индустрии языков программирования и другого инструментария.

Первым делом стал появляться макроассемблер: ассемблер с автоматической разметкой памяти, метками для процедур и данных, так ассемблер стал удобнее, побогаче человеческими интерфейсами и более ориентированным на программиста.

Непереносимость ассемблерного кода являлась действительно большой проблемой из-за разнообразия архитектур, которой занимались многие специалисты того времени. С одной стороны, портировать (переносить) ассемблерный код вручную с одной архитектуры на другую крайне дорого и неэффективно, с другой стороны, реализовать даже несложную программу-транслятор (без оптимизаций исходного кода и кодогенераций) с языка более высокого уровня в машинный код было крайне трудно даже для ведущих учёных и специалистов в области вычислительной техники того времени: очень сильно не хватало инструментов и опыта для решения этой задачи.

8 Лямбда-исчисление

8.1 Что такое лямбда-исчисление и из чего оно состоит

Лямбда-исчисление – это формальная система, предложенная Алонзо Чёрчем в 1930-х годах, которая используется для определения и анализа вычислений. Оно является концептуальным математическим инструментом, эквивалентным по своей выразительности машине Тьюринга, но более ориентированным на работу с функциями и выражениями. Лямбда-исчисление лежит в основе многих концепций функционального программирования и используется для описания вычислений в чистой математической форме. Лямбда-исчисление также оказалось полезно и в Certidied Programming, для формальной проверки свойств программ, пример – язык Coq.

На какие категории делятся все лямбда-термы:

1. Переменные: $x, y, z \ldots$

2. Константы: $a, b, c...$
3. Комбинации термов (утверждений) или применение функции
    1. s to an argument t: st
4. Абстракция
    1. x - переменная, s - выражение: $\lambda x.s$ означает, что x является переменной в выражении s

8.1.1 Свободные и связанные переменные

Все переменные делятся на два типа: свободные (Free) и связанные (Bound)

1. Свободная переменная – это та переменная, которая **не связана лямбда-оператором** внутри выражения. Пример: $\lambda x.(x + y)$, здесь y является свободной переменной. Свободные переменные определены во внешнем контексте, вне лямбда-функции.
2. Связанная переменная (иногда также называется формальным параметром) – это переменная, которая **определена лямбда-оператором** внутри абстракции (в выражении). Пример: $\lambda x.(x + y)$, здесь x является свободной переменной. Являются параметрами лямбда-функции, то есть когда выражение λx будет применено к значению a будет осуществлена подстановка a в выражение.

8.2 Формальное определение свободных переменных

1. $FV(x) = \{x\}$:
    - Если x – просто переменная, то она считается свободной, так как не находится под лямбда-оператором.
2. $FV(c) =$:
    - Если c – константа, то она не содержит свободных переменных.
3. $FV(st) = FV(s) \cup FV(t)$:
    - Если выражение представляет собой применение st, то множество свободных переменных равно объединению свободных переменных из s и t.
4. $FV(\lambda x.s) = FV(s)\{x\}$:
    - Если выражение – это лямбда-абстракция $\lambda x.s$, то переменная x становится связанной, поэтому её нужно исключить из множества свободных переменных $FV(s)$.

8.3 Формальное определение связанных переменных

1. $BV(x) = \emptyset$:
    - Если x – просто переменная, то она не является связанной.
2. $BV(c) = \emptyset$:
    - Если c – константа, то она не содержит связанных переменных.
3. $BV(s\ t) = BV(s) \cup BV(t)$:
    - Если выражение представляет собой применение $s\ t$, то множество связанных переменных равно объединению связанных переменных из s и t.
4. $BV(\lambda x.s) = BV(s) \cup \{x\}$:

- Если выражение – это лямбда-абстракция $\lambda x.s$, то переменная x становится связанной, поэтому она добавляется в множество связанных переменных $BV(s)$.

Теперь подумаем о том, как и с помощью чего лямбда-исчисление позволяет описывать какую-то математику и какие-то действия. Какие правила помогут смоделировать вычисление функции, описание алгоритма?

8.4 Замена в лямбда-термах (Lambda-term substituting)

Цель: Замена в лямбда-термах используется для вычисления или упрощения выражений. Это позволяет подставить одно выражение вместо переменной и продолжить редукцию, чтобы приблизиться к конечному результату.

1. **Обозначение**:
   $*[t/x]$ – замена переменной x на терм t в произвольном лямбда-выражении.
   - Здесь (t) – терм, который нужно подставить, а (x) – переменная, которую заменяем.
2. **Правила замены:**
   - $x[t/x] = t$
     Если переменная x заменяется на t, то результатом будет сам терм t.
     Комментарий: Простая подстановка: переменная заменяется на терм.
   - $y[t/x] = y$**, если** $x \neq y$
     Если переменная y не совпадает с заменяемой x, она остаётся неизменной.
     Комментарий: Никаких изменений, если переменная не та, которую мы заменяем.
   - $c[t/x] = c$
     Если c – константа, то она не изменяется.
     Комментарий: Константы не затрагиваются заменой.
   - $(s_1\ s_2)[t/x] = (s_1[t/x])\ (s_2[t/x])$
     Для применения $(s_1\ s_2)$ замена выполняется отдельно в каждом подвыражении s_1 и s_2.
     Комментарий: Замена применяется рекурсивно к каждому из термов в применении.
   - $(\lambda x.s)[t/x] = \lambda x.s$
     Если x связана в абстракции $\lambda x.s$, замена не производится. Комментарий: Переменная x в пределах λx уже связана, поэтому замена игнорируется.
   - $(\lambda y.s)[t/x] = \lambda y.(s[t/x])$**, если** $x \neq y$ **и** $y \notin FV(t)$
     Если переменная y не совпадает с x и не входит в свободные переменные терма t, то замена производится в теле s.
     Комментарий: Если замена безопасна, то она выполняется внутри тела функции.
   - $(\lambda y.s)[t/x] = \lambda z.(s[z/y][t/x])$**, иначе, где** $z \notin FV(s) \cup FV(t)$
     Если переменная y потенциально конфликтует (например, входит в $FV(t)$), то происходит **альфа-конверсия**. Мы заменяем y на новую переменную z, чтобы избежать конфликтов, а затем выполняем замену.
     Комментарий: Альфа-конверсия используется для предотвращения ошибок при подстановке.

Обозначения:

- Альфа-конверсия (z) – замена имени переменной для устранения конфликта с существующими свободными переменными.

Этот процесс замены важен для корректного выполнения редукций в лямбда-исчислении, а также для анализа, упрощения и оптимизации выражений.

8.4.1 1. Альфа-конверсия (Alpha conversion)

$\lambda x.s \xrightarrow{\alpha} \lambda y.s[y/x]$ при условии, что $y \notin FV(s)$

- **Описание**: Замена связанной переменной (x) на новую переменную (y), которая не конфликтует со свободными переменными (FV(s)) в теле терма.
- **Пример**: $\lambda u.u\ v \xrightarrow{\alpha} \lambda w.w\ v$
- **Пояснение**: Переменная (u) была заменена на (w) для устранения возможных конфликтов.

8.4.2 2. Бета-конверсия (Beta conversion)

$(\lambda x.s)\, t \xrightarrow{\beta} s[t/x]$

- **Описание**: Применение лямбда-абстракции к аргументу. Переменная (x) в теле функции (s) заменяется на терм (t).
- **Пример**: $(\lambda x.x + 1)\, 2 \xrightarrow{\beta} 2 + 1$
- **Пояснение**: Произошла подстановка аргумента (2) вместо переменной (x), что упрощает выражение.

8.4.3 3. Эта-конверсия (Eta conversion)

$\lambda x.t\ x \xrightarrow{\eta} t$ при условии, что $x \notin FV(t)$

- **Описание**: Упрощение выражения, если функция $\lambda x.t\ x$ просто передаёт свой аргумент в (t). Свободные переменные (t) не должны зависеть от (x).
- **Пример**: $\lambda u.v\, u \xrightarrow{\eta} v$
- **Пояснение**: Функция $(\lambda u.v\, u)$ эквивалентна (v), так как (u) напрямую передаётся в (v).

Все эти операции мы можем применять в любой последовательности. Но когда мы говорим про языки программирования, нам важно понимать, какая именно стратегия вычислений в нём используется. Например, нужно вычислить функцию с переданными в неё сложными аргументами (например, какими-то выражениями, другими функциями). Возникают два варианта: (1) Normal-order reduction - предподсчитать аргументы функции и подать функции на вход уже подсчитанные значения аргументов, (2) Applicative order reduction - передать в функцию не предподсчитанные аргументы, а аргументы в виде выражений, и только потом их вычислить.

8.5 Порядок редукции (Reduction Order)

Порядок редукции определяет, как именно выполняются преобразования (редукции) в лямбда-выражениях. Основные стратегии – это **Normal-order reduction** и **Applicative-order reduction**.

8.5.1 (1) Normal-order reduction (редукция нормального порядка)

- **Описание**:
  Редукция нормального порядка сначала выполняет редукцию в **головной позиции** (внешняя часть выражения), прежде чем переходить к редукциям внутри подвыражений. Это повторяется до тех пор, пока не станет невозможной дальнейшая редукция головного выражения.
- **Алгоритм**:
  1. Выбирается самый внешний терм для редукции (в головной позиции).
  2. Когда редукция невозможна в головной позиции, выполняются редукции в подвыражениях (слева направо).
- **Особенность**:
  Гарантирует, что, если терм имеет нормальную форму (конечное упрощение), он будет достигнут.
- **Пример**:
  Если терм выглядит как:
  $(\lambda x.y)\,((\lambda x.x\,x)\,(\lambda x.x\,x))$.
  Сначала редуцируется внешнее выражение:
  $(\lambda x.y) \to y$.

8.5.2 (2) Applicative-order reduction (редукция аппликативного порядка)

- **Описание**:
  Редукция аппликативного порядка сначала вычисляет **внутренние подвыражения** (аргументы функции) до их нормальной формы, а затем применяется редукция головного выражения.
- **Алгоритм**:
  1. Выполняются редукции во всех подвыражениях.
  2. После завершения внутренних редукций выполняется редукция головного выражения.
- **Особенность**:
  Может не завершиться (не гарантирует, что нормальная форма будет достигнута).
- **Пример**:
  Рассмотрим терм, который имеет нормальную форму:
  $(\lambda x.y)\,((\lambda x.x\,x)\,(\lambda x.x\,x))$.
  В аппликативном порядке сначала редуцируется внутреннее выражение:
  $((\lambda x.x\,x)\,(\lambda x.x\,x)) \to (\lambda x.x\,x)\,(\lambda x.x\,x)$.
  То есть этот процесс с данным порядком редукции может продолжаться бесконечно, даже если выражение обладает нормальной формой (т.е. такой формой, которая больше не может быть редуцирована).

8.5.3 Особенность аппликативного порядка

- Пример, где редукция **не завершается**:
  $(\lambda x.y)\,((\lambda x.x\,x)\,(\lambda x.x\,x))$.
  Постоянная редукция внутреннего выражения:
  $((\lambda x.x\,x)\,(\lambda x.x\,x)) \to (\lambda x.x\,x)\,(\lambda x.x\,x) \to \ldots$

Эти стратегии являются основой для реализации вычислений в различных интерпретаторах функциональных языков программирования.

8.6 Но как же превратить лямбда-исчисление в полноценный язык программирования?

Для этого нам понадобится определить:

1. Булеву алгебру на базе лямбда-исчислений
2. Поддержку пар и кортежей
3. Понятие натуральных чисел
4. Рекурсивные функции

И сразу же после этого мы будем способны реализовать алгоритм расчёта факториала в чистом лямбда-исчислении.

8.7 Булевы значения и условные выражения (Boolean and If Statement)

Булевы значения и условные выражения в лямбда-исчислении описываются с использованием базовых конструкций, представляющих истину и ложь, а также условного выбора.

8.7.1 Булевы значения

1. **Истина (true):**
   true $\equiv \lambda xy.x$
   - Выбирает первый аргумент.
2. **Ложь (false):**
   false $\equiv \lambda xy.y$
   - Выбирает второй аргумент.

8.7.2 Условное выражение (if-then-else)

if E then E_1 else $E_2 \equiv E\,E_1\,E_2$

- **Описание:** Выражение E вычисляется как условие (подразуемваем, что используем Normal-order reduction):
  - Если $E =$ true, возвращается E_1.
  - Если $E =$ false, возвращается E_2.
- **Пример** с true:
  - if true then E_1 else E_2
    Преобразуем:
    $=$ true $E_1\,E_2$
    $= (\lambda xy.x)\,E_1\,E_2$
    $= E_1$.

8.7.3 Базовые логические функции

1. **Отрицание (not):**
   not $p \equiv$ if p then false else true
2. **Конъюнкция (and):**
   p and $q \equiv$ if p then q else false
3. **Дизъюнкция (or):**
   p or $q \equiv$ if p then true else q

8.8 Пары (Pairs)

Пары в лямбда-исчислении реализуются через функции, которые принимают обработчик (функцию) и применяют его к двум элементам. С помощью пар мы можем вернуть несколько значений из функций.

8.8.1 Определение пары

$(E_1, E_2) \equiv \lambda f.f\ E_1\ E_2$

- Пара представляется как функция, которая принимает аргумент f и применяет его к значениям E_1 и E_2.

8.8.2 Операции с парами

1. **Получение первого элемента (fst):**

   fst $p \equiv p$ true

   - Первый элемент пары получается, применяя её к функции true.

2. **Получение второго элемента (snd):**

   snd $p \equiv p$ false

   - Второй элемент пары получается, применяя её к функции false.

8.8.3 Пример получения первого элемента пары (p, q)

1. fst $(p, q) = (p, q)$ true
2. $= (\lambda f.f\ p\ q)$ true
3. $=$ true $p\ q$
4. $= p$

8.9 Натуральные числа (Natural Numbers)

В лямбда-исчислении натуральные числа определяются через так называемые **числа Чёрча**. Это функции, которые принимают два аргумента: функцию f и значение x, и применяют функцию f n раз к x. То есть натуральные числа в лямбда-исчислении описываются через функцию f, повторно применяемую n раз. Арифметические операции реализуются через композицию функций.

8.9.1 Определение чисел

1. **Общее определение:** $n \equiv \lambda fx.f^n\ x$

   - Число n – это функция, которая применяет f ровно n раз.

2. **Примеры чисел Чёрча:**

   - $0 \equiv \lambda fx.x$
     (нулевое применение f, результатом остаётся x).
   - $1 \equiv \lambda fx.f\ x$
     (одно применение f к x).
   - $2 \equiv \lambda fx.f(f\ x) = \lambda fx.f^2\ x$
     (два применения f к x).

8.9.2 Арифметические функции

1. **Следующее число (SUC), эквивалент инкремента:** SUC $\equiv \lambda nfx.f(n\,f\,x)$
   - Увеличивает число n на 1.
2. **Проверка на ноль (ISZERO):** ISZERO $n \equiv n(\lambda x.\text{false})\,\text{true}$
   - Если $n = 0$, возвращает true, иначе false.
3. **Сложение (m + n):**
   $m + n \equiv \lambda fx.m\,f\,(n\,f\,x)$
   - Применяет функцию f $m + n$ раз.
4. **Умножение (m * n):** $m \cdot n \equiv \lambda fx.m\,(n\,f)\,x$
   - Применяет $n\,f$ m раз.

8.9.3 Преобразования для вычисления предшествующего числа

1. **Вспомогательная функция (PREFN):** PREFN $\equiv \lambda fp.(\text{false}, \text{if fst } p \text{ then snd } p \text{ else } f(\text{snd } p))$
2. **Предшествующее число (PRE):** PRE $n \equiv \lambda fx.\text{snd}\ (n(\text{PREFN}\ f)\ (\text{true}, x))$
   - Вычисляет $n - 1$, если $n > 0$, и остаётся 0, если $n = 0$.

8.10 Рекурсивные функции (Recursive Functions)

В лямбда-исчислении для определения рекурсивных функций используется **Y-комбинатор**. Это специальная функция, которая позволяет определять рекурсию даже в рамках функционального программирования без явного самоповторения.

8.10.1 Определение Y-комбинатора

$Y \equiv \lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x))$

- Y – это комбинатор, который принимает функцию f и позволяет ей рекурсивно вызывать саму себя.

8.10.2 Пример применения Y к функции f

1. $Y\,f = (\lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x)))\,f$
2. $= (\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x))$
3. $= f\,((\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x)))$
4. $= f\,(Y\,f)$

8.11 Факториал (Factorial)

Факториал в лямбда-исчислении можно определить рекурсивно с использованием **Y-комбинатора**.

8.11.1 Определение факториала

1. **Первый шаг - стандартное определение:**
   $fact(n) = \text{if } ISZERO\, n \text{ then } 1 \text{ else } n * fact(PRE\, n)$
   - Если $n = 0$, возвращается 1.
   - Иначе: $n * fact(n - 1)$.
2. **Второй шаг - через лямбда-исчисление:**
   $fact = \lambda n.\text{if } ISZERO\, n \text{ then } 1 \text{ else } n * fact(PRE\, n)$
3. **Третий шаг - рекурсивное определение через функции:**
   $fact = (\lambda fn.\text{if } ISZERO\, n \text{ then } 1 \text{ else } n * f(PRE\, n))\, fact$
   - Здесь f представляет функцию, которая вызывается рекурсивно.
4. **Четвёртый шаг - определение вспомогательной функции F:**
   $F = \lambda fn.\text{if } ISZERO\, n \text{ then } 1 \text{ else } n * f(PRE\, n)$
   - Эта функция содержит основную логику факториала, а рекурсия обеспечивается внешним комбинатором.
5. **Пятый шаг - запуск рекурсии с использованием Y-комбинатора:** $fact = Y\, F$
   - Здесь Y – Y-комбинатор, который делает F рекурсивной.

8.12 Вычисление факториала: Итерация 1

А сейчас посмотрим на пример вычисления факториала тройки $fact(3)$, определённого по изложенным выше правилам.

1. Исходное выражение факториала

   $$fact(3) \xrightarrow{\text{fact}} Y\, F\, 3$$

2. Развёртывание Y

   $$Y\, F\, 3 \xrightarrow{\text{Y}} (\lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x)))\, F\, 3$$

3. Применение β-редукции

   $$\xrightarrow{\beta} (\lambda x.F\,(x\,x))(\lambda x.F\,(x\,x))\, 3$$

4. Применение второй β-редукции

   $$\xrightarrow{\beta} F\,((\lambda x.F\,(x\,x))(\lambda x.F\,(x\,x)))\, 3$$

5. Обратное сокращение (backward β)

   $$\xrightarrow{\text{backward } \beta} F\,(Y\, F)\, 3$$

6. Развёртывание F

   $$F\,(Y\, F)\, 3 \xrightarrow{\text{F}} (\lambda fn.\text{if } ISZERO\, n \text{ then } 1 \text{ else } n * f(PRE\, n))\,(Y\, F)\, 3$$

7. Применение β-редукции

   $$\xrightarrow{\beta} \text{if } ISZERO\, 3 \text{ then } 1 \text{ else } 3 * (Y\, F)(PRE\, 3)$$

8. Проверка $ISZERO$ (то, как работает $ISZERO$, описано немного ниже)

   $\xrightarrow{\text{ISZERO 3}}$ if false then 1 else $3 * (Y\ F)(PRE\ 3)$

9. Применение условия (if)

   $\xrightarrow{\text{if}}$ false $1\ (3 * (Y\ F)(PRE\ 3))$

10. Применение false

    $\xrightarrow{\text{false}}$ $(\lambda xy.y)\ 1\ (3 * (Y\ F)(PRE\ 3))$

11. Применение β-редукции

    $\xrightarrow{\beta}$ $3 * (Y\ F)(PRE\ 3)$

12. Применение $PRE3$ и результат (описание работы $PRE3$ также находится ненмого ниже)

    $\xrightarrow{\text{PRE3}}$ $3 * (Y\ F)\ 2$

Затем идут итерации 2 и 3, схожие с итерацией 1, а на 4 итерации мы получаем вызов факториала от нуля, его разберём ниже, после разбора встреченных в первой итерации функций.

8.13 Вычисление $ISZERO\,3$

1. Исходное выражения с $ISZERO$

   $ISZERO\,3 \xrightarrow{\text{ISZERO}} 3\ (\lambda x.\text{false})\ \text{true}$

2. Развёртывание числа 3

   $3 \xrightarrow{\text{ISZERO}} (\lambda fx.f^3\ x)\ (\lambda x.\text{false})\ \text{true}$

3. Применение β-редукции к выражению

   $\xrightarrow{\beta}$ $(\lambda x.\text{false})^3\ \text{true}$

4. Развёртывание степени 3

   $\xrightarrow{\text{number}}$ $(\lambda x.\text{false})\ (\lambda x.\text{false})\ (\lambda x.\text{false})\ \text{true}$

5. Применение β-редукции для каждого шага

   - Сначала: $\xrightarrow{\beta}$ $(\lambda x.\text{false})\ (\lambda x.\text{false})\ \text{true}$
   - Затем: $\xrightarrow{\beta}$ $(\lambda x.\text{false})\ \text{true}$
   - Наконец: $\xrightarrow{\beta}$ false

Итог:

$ISZERO\,3 \to$ false
Функция $ISZERO\ 3$ возвращает $false$, поскольку число 3 не равно 0.

8.14 Вычисление $PRE\,3$

1. Исходное выражение с PRE

   $PRE\,3 \xrightarrow{\text{PRE}} \lambda fx.\text{snd}(3\,(PREFN\,f)\,(\text{true}, x))$

2. Развёртывание числа 3

   $\xrightarrow{\text{number}} \lambda fx.\text{snd}((\lambda fx.f^3\,x)\,(PREFN\,f)\,(\text{true}, x))$

3. Применение β- редукции

   $\xrightarrow{\beta} \lambda fx.\text{snd}((\text{PREFN}\ f)^3\,(\text{true}, x))$

4. Развёртывание $PREFN\,f$ (то, как работает PREFN, немного ниже)

   $\xrightarrow{(PREFNf)^3\ (true,\ x)} \lambda fx.\text{snd}((\text{false}, f^2\,x))$

5. Применение snd

   $\xrightarrow{\text{snd}} \lambda fx.f^2\,x$

6. Числовая интерпретация

   $\xrightarrow{\text{number}} 2$

Итог:
$PRE\,3 \to 2$

8.15 Вычисление $(PREFN\,f)^3\,(\textbf{true}, x)$

1. Исходное выражение

   $(PREFN\,f)^3\,(\text{true}, x) \xrightarrow{\text{power}} PREFN\,f\,((PREFN\,f)^2\,(\text{true}, x))$

2. Развёртывание второй степени

   $\xrightarrow{(PREFNf)^2\ (true,x)} PREFN\,f\,(\text{false}, f\,x)$

3. Развёртывание третьего вызова $PREFN$

   $\xrightarrow{\text{PREFN}} (\lambda f\,p.(\text{false}, \text{if fst}\,p\ \text{then snd}\,p\ \text{else}\ f(\text{snd}\,p)))\,f\,(\text{false}, f\,x)$

4. Применение β-редукции

   $\xrightarrow{\beta}$ (false, if fst(false, $f\ x$) then snd (false, fx) else f (snd (false, $f\ x$)))

5. Развёртывание if-then-else statement и fst, snd

   $\xrightarrow{snd.if.fst} (false, f(fx))$

6. Результат

   $\xrightarrow{power} (false, f^2x)$

Итог:

$(PREFN\ f)^3\ (\text{true}, x) \to (\text{false}, f^2\ x)$

8.16 Вычисление факториала: Итерация 4

Здесь вычисляется последняя итерация вычисления факториала тройки $3{*}2{*}1{*}(Y\ F)\ 0$.

1. Исходное представление итерации

   $(Y\ F)\ 0 \xrightarrow{Y} (\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)))F\ 0$

2. Применение β-редукции

   $\xrightarrow{\beta} (\lambda x.F(x\ x))(\lambda x.F(x\ x))\ 0$

3. Применение второй β-редукции

   $\xrightarrow{\beta} F((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ 0$

4. Обратное свёртывание

   $\xrightarrow{\text{backward } \beta} F(Y\ F)\ 0$

5. Развёртывание F

   $\xrightarrow{F} (\lambda f\ n.\text{if}\ ISZERO\ n\ \text{then}\ 1\ \text{else}\ n * f(PRE\ n))(Y\ F)\ 0$

6. Применение β-редукции

   $\xrightarrow{\beta} (\text{if}\ ISZERO\ 0\ \text{then}\ 1\ \text{else}\ 0 * (Y\ F)(PRE\ 0))$

7. Проверка $ISZERO$

   $\xrightarrow{\text{ISZERO 0}} \text{if true then}\ 1\ \text{else}\ 0 * (Y\ F)(PRE\ 0)$

8. Применение условия (if)

   $\xrightarrow{\text{if}} \text{true}\ 1\ (0 * (Y\ F)(PRE\ 0))$

9. Применение true

   $\xrightarrow{\text{false}} (\lambda x\ y.x)\ 1\ (0 * (Y\ F)(PRE\ 0))$

10. Применение β-редукции

$\xrightarrow{\beta} 1$

Итог:

$3 \cdot 2 \cdot 1 \cdot (Y\ F)\ 0 = 3 \cdot 2 \cdot 1 \cdot 1$

8.17 Вычисление факториала: Fold Stack

Далее показан процесс “свёртки” результата вычисления факториала $fact\ 3$ до окончательного значения. Рассмотрим шаги последовательно.

Исходное выражение перед сверткой:

$fact\,3 \xrightarrow{\text{fact}} 3 * 2 * 1 * 1$

Шаг 1: $3 * 2$

1. $3 * 2 \xrightarrow{*} \lambda f\,x.3(2f)x \xrightarrow{number} \lambda f\,x.(\lambda f\,x.f^3x)(2f)x$
2. $\xrightarrow{\beta} \lambda f\,x.(2f)^3x \xrightarrow{number} \lambda f\,x.((\lambda fx.f^2x)f)^3x$
3. $\xrightarrow{\beta} \lambda f\,x.((\lambda x.f^2x))^3x \xrightarrow{\beta} \lambda f\,x.(f^2)^3x \xrightarrow{\beta} \lambda f\,x.f^6x$
4. $\xrightarrow{\text{number}} 6$

Шаг 2: $6 * 1 * 1$

1. $6 * 1 \xrightarrow{*} \lambda f\,x.6(1f)x \xrightarrow{number} \lambda f\,x.(\lambda f\,x.f^6x)(1f)x$
2. $\xrightarrow{\beta} \lambda f\,x.(1f)^6x \xrightarrow{number} \lambda f\,x.((\lambda fx.fx)f)^6x$
3. $\xrightarrow{number} \lambda f\,x.(\lambda x.fx)^6x \xrightarrow{\beta} \lambda f\,x.f^6x$
4. $\xrightarrow{\text{number}} 6$

Шаг 3: Окончательный результат

1. $\xrightarrow{6*1} = 6 * 1$
2. $\xrightarrow{6*1} 6$

Теперь уже можно сказать, что процесс успешно завершён, и результат факториала $fact\,3$ равен 6.

Но зачем нам это нужно в таком казалось бы сложном виде?

А ответ на этот вопрос состоит в том, что иногда полезно задуматься о том исходном наборе правил, определений и аксиом, который мы принимаем за основание (базу), т.е. такой набор объектов, используя которые можно сымитировать всё что угодно, например, описать алгоритм вычисления факториала числа или доказать какие-либо свойства программы. В то же время, лямбда-исчисление как раз можно взять за такой базовый набор, который, к тому же, как показывает практика, хорошо справляется со своей задачей.

9 Языки программирования высокого уровня

Языки программирования высокого уровня предоставляют в первую очередь следующее:

- Абстракцию от процессора (в частности от его системы команд)
- Структурное программирование
- Декларативное программирование
  - (уровень абстракции позволяет отгородиться от конкретных процессов протекающих за различными операциями)
  - декларативность зависит от стиля кодирования

9.1 Структурное программирование

Его суть заключается в том, чтобы программировать не инструкции и переходы меджу ними, а структурные блоки.

Приемущество данного подхода заключается в определении четких границ программы:

- Четко понятны мементы входа и выхода в определенный блок. Для циклов и условий (if else) всегда наглядно что и когда выполняется
- Можно четко отследить последовательность вызовов и границы подпрограмм

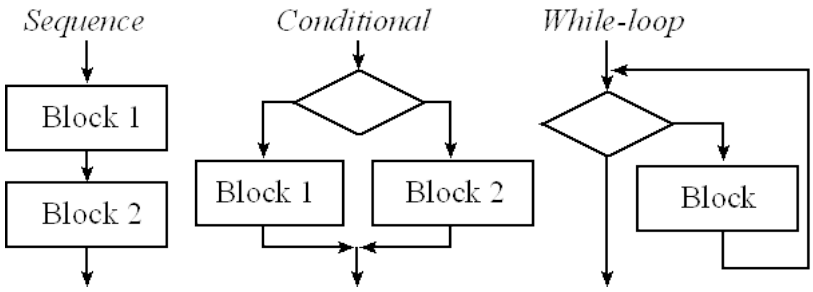


Figure 16: Структурные блоки

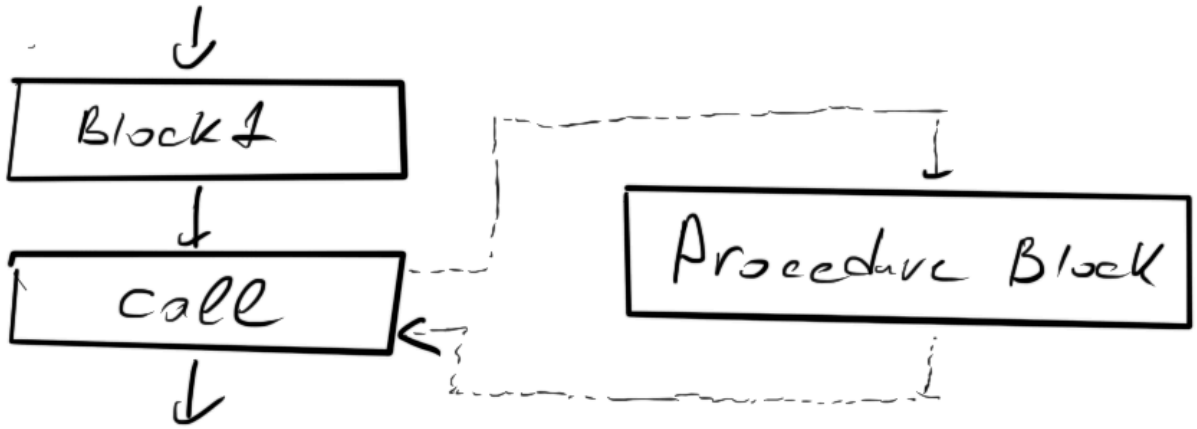


Figure 17: Процедурный блок

9.1.1 Почему Go To признан плохим

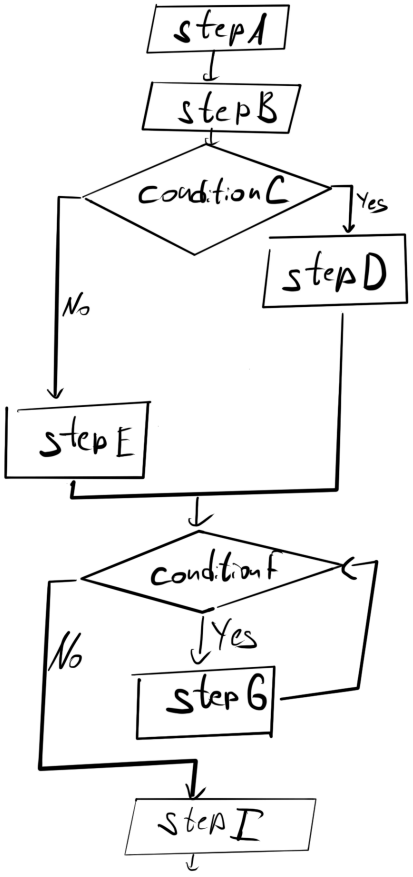
Аргументация Дейкстра в его статье Go To Statement Considered Harmful

В ней, на примере простой программы на псевдоструктурном и псевдонизкоуровневом языке он пытается написать трассу выполнения программы. Главным критерием оценки являлось то, насколько просто написать эту трассу

Структурный вариант vs **Низкоуровневый**

Программы:

Трассировки:




```
1.  stepA()                 | 1.  stepA
                            |
2.  stepB()                 | 2.  stepB
                            |
3.  if conditionC():        | 3.  conditionC
                            | 4.  jz 7
5.      stepD()             | 5.  stepD
                            |
                            |
6.  else:                   | 6.  jmp 8
7.      stepE()             | 7.  stepE
                            |
                            |
8.  while conditionF:       | 8.  conditionF
                            | 9.  jz 12
                            |
10.     stepG()             | 10. stepG
                            | 11. jmp 9
                            |
12. stepI()                 | 12. stepI
```

Figure 18: Сравнение структурного и низкоуровневого вариантов

Структурный	Низкоуровневый
1	1
2	2
3.5	4.5
8	4.8
8[1].10	4.9.10
12	4.9.10.9.10.9.12

При построении трассы сразу видно, что благодаря наличиную логических блоков в структурном подходе мы можем явно определить, в какой момент закончилась та или иная часть программы (видна точка выхода), в то время как в низкоуровневом варианте такой маркер отсутствует (не видна точка выхода).

На сегодняшний день, при повсеместном использовании структурного подхода, goto плох тем, что он ломает ту самую блочную гарантию, создавая новую точку выхода, превращая код в спагетти.

9.1.2 В каких случаях Go To полезен

- Выход из вложенных циклов

```
outer_loop:
for(...) {
   for(...) {
      if (break condition) {
         goto outer_loop;
      }
   }
}
```

Однако во многих языках данная функция реализована на уровне семантики языка

- Освобождение ресурсов

Необходимость подчистить какие-то ресурсы перед выходом из функции в особенности нескольких

```
a = open_resource()
if (!a) return;

b = open_resource()
if (!b) goto close_A;

c = open_resource()
if (!c) goto close_B;

...

close_B:
   close_resource(b);
close_A:
   close_resource(a);
```

- Конечные автоматы

```
goto state2;
...
state1:
   ...
state2:
   ...
   goto state1;
```

10 Изменяемое состояние считать вредным

10.1 Предисловие

> Материал, представленный в этом разделе, хотя и связан с функциональным программированием, вполне применим и к другим подходам разработки программного обеспечения.

Как известно, функциональные языки программирования часто отпугивают новичков не только непривычным синтаксисом, обилием математики и абстракций, но и тем, что в них **нет изменяемого состояния**.

Например, язык программирования Haskell вообще не содержит в своем синтаксисе оператор присваивания. Такое решение обосновывается тем, что **отказ от изменяемых данных существенно повышает корректность работы программы.**

Тем не менее, идея о вреде изменяемого состояния пока не получила широкого распространения за пределами функциональных языков. Почему же в реальной практике мы не можем просто отказаться от изменяемого состояния?

10.2 Причины невозможности отказаться от изменяемых данных

1. Многие подходы, например ленивые вычисления, успешно используемые в функциональном программировании, сложно (или вообще невозможно) реализовать в Java или C++. Это объясняется тем, что их модели памяти и времени выполнения изначально заточены под императивный стиль программирования.
2. Многие реализации из стандартных библиотек традиционных языков программирования напрямую завязаны на использование изменяемого состояния (например, коллекции и операция добавления в коллекцию нового элемента).

   Более того, для многих коллекций и вовсе отсутствуют эффективные аналоги в неизменяемых структурах данных (например, хеш-таблица)
3. Отказ от изменяемого состояния часто приводит к увеличению потребления памяти, а значит нам нужна сборка мусора. Однако в системах с жесткими требованиями к времени отклика использование сборщика мусора становится неприемлемым, и работа с изменяемым состоянием оказывается более практичной
4. Многие предметные области прямо диктуют наличие изменяемого состояния (например, изменяемый банковский счет)

Хотя отказ от изменяемого состояния часто не оправдан, важно понимать, что его использование связано с рядом рисков, связанных с работой и поддержкой такого кода. Давайте рассмотрим эти проблемы поподробнее.

10.3 Опасности изменяемого состояния

10.4 Неожиданные изменения → Нарушение инвариантов

Одна из самых главных проблем изменяемого состояния заключается в том, что оно может быть изменено в любой момент времени (кэп), и часто эти изменения приводят к нарушению внутреннего состояния программы.

Например, поле `size` у объекта связный список должно в точности совпадать с реальным количеством элементов в этом списке. В таком случае говорят, что соблюдается некоторый инвариант.

Код пишется таким образом, чтобы в любой момент времени, когда мы можем наблюдать за состоянием программы, ее инварианты не нарушались. Каждый участок кода начинает работать с предположением о том, что инварианты соблюдаются и по завершении его работы инварианты также не будут нарушены.

Однако на практике сохранение инвариантов представляет собой крайне непростую задачу. Для каждого участка кода, работающего с изменяемым состоянием, нужно не только знать все связанные с объектом инварианты, но и следить за их соблюдением!

Еще сложнее сохранять инварианты без нарушения модульности. Поэтому программисты стремятся делать инварианты охватывающими как можно меньшее число объектов и зависящими от как можно меньшего числа их изменяемых свойств. Ведь иначе мы сами того не подозревая, нарушим целостность нашей программы.

Приведем пример:

Пусть нам дан следующий класс Box на языке Python. Как видно, состояние коробки свободно поддается изменению, с помощью метода `update_value()`:

```
class Box():
    def __init__(self, value):
        self.value = value
    def __eq__(self, other):
        return self.value == other.value
    def __hash__(self):
        return hash(self.value)
    def update_value(self, value):
        self.value = value

a, b = Box(1), Box(2)
a == a          # => True
a != b          # => True
a == Box(1)     # => True
```

Проблема 1 — Эквивалентность и необходимость знания деталей

Следующие два объекта эквивалентны друг другу, но при этом словарь содержит 2 разных значения по “одинаковым” ключам:

```
c, d = Box(1), Box(1)
c == d          # => True
dict = {}
dict[c] = 'c'
dict[d] = 'd'
dict # => { <__main__.Box object at 0x101046f50>: 'c',
    #      <__main__.Box object at 0x100ea0ad0>: 'd' }
```

Как думаете, что будет, если вызвать `dict[Box(1)]`?

При этом если бы мы имели структуру с неизменяемым состоянием (например, число):

```
e, f = 1, 1
e == f          # => True
dictIm = {}
dictIm[e] = 'e'
dictIm[f] = 'f'
dictIm          # => {1: 'f'}
```

Проблема 2 — Утиная типизация и нарушение коммутативности

Допустим, мы определили еще одну коробку:

```
class DoubleBox():
    def __init__(self, value1, value2):
        self.value = value1
        self.second_value = value2
    def __eq__(self, other):
        if type(self) == type(other):
return self.value == other.value \
    and self.second_value == other.second_value
        return False
```

Попытаемся применить сравнение:

```
Box(1) == DoubleBox(1, 2)     # => True
DoubleBox(1, 2) != Box(1)     # => True
```

Коммутативность нарушена. В одном случае используется сравнение, закрепленное за объектом типа `Box`, в другом — за `DoubleBox`.

При этом класс `Box` ничего не знает о `DoubleBox`, теоретически, его можно было бы определить в любом месте программы и проблема будет воспроизводиться в самых неожиданных местах.

Проблема 3 — Нарушение внешнего инварианта

Нам дан словарь:

```
dictMut = {}
g, h = Box(1), Box(2)
dictMut[g] = 'g'
dictMut[h] = 'h'
```

Давайте попробуем обновить состояние `g`:

```
g.update_value(10)
g in dictMut # => False
h in dictMut # => True
g         # => <__main__.Box object at 0x101047790>
h         # => <__main__.Box object at 0x101047dd0>
dictMut   # => {
          #    <__main__.Box object at 0x101047790>: 'g',
          #    <__main__.Box object at 0x101047dd0>: 'h'
          #    }
```

Все, что мы теперь можем сделать с этой структурой данных — выбросить в мусор.

10.5 Кэширование

Изменяемое состояние очень сложно кэшировать.

Допустим, мы кэшировали наше состояние. Но вот в какой-то момент оно поменялось, и наш кэш перестал быть валидным. Из этой проблемы есть 2 выхода, и оба одинаково плохи:

1. Синхронизация кешей/обновление кешей при обновлении данных $\longrightarrow$ Overhead
2. Отказаться от кэширования $\longrightarrow$ потеря производительности

Приведем пример:

Пусть нам дан класс корзины покупателя:

```
class Cart {
    private Customer customer;
    private List<Product> products;
    private int totalPrice = -1;

    private int computeTotalPrice() { ... }
    // It's a hard to compute value...

    public int getTotalPrice() {
        // So we will compute it only by the request...
        if(totalPrice == -1)
            totalPrice = computeTotalPrice();
        return totalPrice;
    }

    // ...And for each changing of the cart
    // we will invalidate cache, like this
    public void addProduct(Product p) {
        products.add(p);
        totalPrice = -1;
    }
    public void removeProduct(Product p) {
        products.remove(p);
        totalPrice = -1;
    }
}
class Product {
    // Let the Product has a mutable state
    public void setPrice(int price) { ... }
}
```

Является ли `totalPrice` корректно кэшируемым значением?

Ответ: Нет, не является.

Сценарий:

- Пользователь собрал корзину и вычислил ее стоимость (`getTotalPrice()`)
- Магазин обновил цену на товар (`setPrice(322)`)
- Итоговая цена в корзине пользователя **осталась прежней**

Выходы из ситуации:

- Отказ от кэширования (потеря производительности, eager-пересчет)
- Накручивание шаблона Observer для класса `Product` (программная оверинженерия)
- Фиксация стоимости для каждого пользователя — кэш пользователя считать окончательным (безопасность, подмена кэша)

10.6 Многопоточное изменяемое состояние

Вкратце, описывается следующим образом:

Подавляющее большинство багов в многопоточных программах связано с изменяемыми данными, а именно — с тем, что две (или более) корректных последовательности изменений, переплетаясь в условиях многопоточности, вместе образуют некорректную.

Приведем пример:

Допустим, нам дан банковский счет:

```
class BankAccount {
    void deposit(int amount) {
        setMoney(getMoney() + amount);
    }
    void withdraw(int amount) {
        if(amount > getMoney()) throw new InsufficientMoneyException();
        setMoney(getMoney() - amount);
    }
}
```

Операции deposit/withdraw могут быть в многопоточной среде, как показано ниже:

Действия Марьи	Действия Ивана	Деньги на счете
deposit(50)	deposit(25)	100

Действия Марьи	Действия Ивана	Деньги на счете
getMoney() → 100		100
	getMoney() → 100	100
setMoney(100+50)		150
	setMoney(100+25)	125

В зависимости от победителя в гонке, деньги одного из участников навсегда будут утеряны.

Вопрос: Допустим, у банка имеется три дата-центра в трех различных городах. Связь с одним из оборвалась. Как избежать ситуации, при которой человек снял деньги в городе с потерянным дата-центром, а после приехал в город и снял эти же деньги повторно?

10.7 Сложный код

С введением изменяемых данных в коде появляется измерение времени как на высоком уровне (взаимодействия компонентов), так и на низком — уровне последовательности строк кода. Вслед за ним приходит дополнительная сложность: необходимо не только решить, что надо сделать, но и в каком порядке. Она проявляется, в основном, в реализациях сложных структур данных и алгоритмов.

Приведем пример:

```
public void add(int index, Object o) {
    Entry e = new Entry(o);
    if (index < size) {
        Entry after = getEntry(index);
        e.next = after;
        e.previous = after.previous;
        if (after.previous == null)
            first = e;
        else
            after.previous.next = e;
        after.previous = e;
    } else if (size == 0) {
        first = last = e;
    } else {
        e.previous = last;
        last.next = e;
        last = e;
    }
    size++;
}
```

Перед вами добавление элемента в двусвязный список из GNU Classpath. Можете ли вы без листа бумаги и карандаша проверить корректность реализованной операции?

10.8 Классификация изменяемого состояния

Перед тем, как мы продолжим, подробно изучим нашего врага.

Следующая классификация изменяемого состояния представляет собой слегка модифицированную классификацию, предложенную Скоттом Джонсоном:

1. Изменяемое состояние полностью отсутствует

2. Изменяемое состояние не видимо для программиста
3. Изменяемое состояние не видимо для клиента
4. Монотонное изменяемое состояние
5. Двухфазный цикл жизни
6. Управляемое изменяемое состояние
7. Инкапсулированное изменяемое состояние
8. Неинкапсулированное изменяемое состояние
9. Многопоточное разделяемое изменяемое состояние

Рассмотрим каждое из состояний подробнее

1. **Отсутствующее**. Наиболее безопасное из классов состояний — достижимо лишь в теории.
2. **Невидимое программисту**. Код, не использующий изменяемое состояние, компилируется в машинный код, использующий изменяемые регистры, стек, память. При правильной реализации компилятора незаметно. С практической точки зрения данное состояние является безопасным.
3. **Невидимое клиенту**. Локальные переменные-счетчики внутри процедур: изменение таких переменных вне процедур не наблюдаемо $\rightarrow$ клиент ничего не знает $\rightarrow$ с точки зрения клиента данное состояние является безопасным.
4. **Монотонное**. Переменные, присваивание которых происходит не более 1 раза — сначала переменная не определена, затем — определена. Всего 2 состояния, одно из которых - не целостно. Перейти из целостного состояния обратно в нецелостное невозможно.
5. **Двухфазное**. Разновидность монотонного изменяемого состояния. Переменная имеет 2 фазы жизни - фазу записи, во время которой нельзя читать, и фазу чтения, во время которой нельзя изменять переменную. Такие ограничения необходимо гарантировать — для этого используется прием “заморозка”.
6. **Управляемое**. Изменение состояния переменной происходит за счёт средств координации, таких как, например, транзакции в СУБД.
7. **Инкапсулированное**. Доступ к переменной происходит только из одного места (`private` поля объектов, обращаться к которым можно только через сеттеры/геттеры). Контроль за целостностью состояния — целиком на реализации объекта. Для контроля инвариантов, охватывающих несколько объектов, необходимы специальные средства; либо же можно инкапсулировать весь контроль за состоянием этих нескольких объектов в другом объекте.
8. **Неинкапсулированное**. Глобальные переменные. Видимы везде, могут быть изменены кем угодно, когда угодно. Страшное зло, которое тяжело (иногда невозможно) контролировать.
9. **Многопоточное разделяемое**. Глобальные переменные в многопоточной среде — как отдельный вид боли. Слишком много различных состояний — при наличии N потоков, каждый из которых проходит K состояний, количество возможных последовательных событий при одновременном выполнении равно K^N.

Техники борьбы с негативными эффектами изменяемого состояния приведены ниже.

10.9 Способы борьбы с изменяемым состоянием

Общие идеи:

1. **Минимизация общего числа состояний**: Чем меньше у системы состояний, тем меньшее количество случаев надо учитывать при взаимодействии с ней. Следует помнить и о том, что чем больше состояний, тем больше и последовательностей состояний, а именно непредусмотренные последовательности состояний зачастую являются причинами багов.
2. **Локализация изменений**: Чем более локальные (обладающие меньшей областью видимости) объекты затрагивает изменение, тем из меньшего числа мест в коде изменение может быть замечено. Чем менее изменения размазаны между несколькими объектами во времени, тем легче логически сгруппировать их в несколько крупных и относительно независимых изменений объектов. К примеру, при сложном изменении узла структуры данных лучше сначала вычислить все новые характеристики этого узла, а затем произвести изменения, нежели производить изменения сразу в самом узле.
3. **Разграничение права на наблюдение и изменение состояния (инкапсуляция)**: Чем меньше клиентов могут изменить состояние, тем меньше действия этих клиентов нужно координировать. Чем меньше клиентов могут прочитать состояние, тем легче его защищать от изменений во время чтения.
4. **Исключение наблюдаемости промежуточных не-целостных состояний**: Если систему невозможно застать в таком состоянии, то снаружи она всегда выглядит целостной и корректно работающей.
5. **Навязывание эквивалентности состояний и их последовательностей**: Если некоторые состояния или их последовательности в каком-то смысле эквивалентны, то клиент избавлен от необходимости учитывать их частные случаи.

Основные техники:

1. Осознание и отказ
2. Инкапсуляция
3. Двухфазный цикл жизни и заморозка
4. Превращение изменяемой настройки в аргумент
5. Концентрация изменений во времени
6. Концентрация изменений в пространстве
7. Многопоточные техники

10.10 Осознание и отказ

Прежде чем внедрять изменяемое состояние, нужно понять, обусловлено ли оно спецификой предметной области:

- Есть ли в реальной модели мира объекты, которые меняются со временем?
- Меняются ли именно те объекты, которые вы планируете сделать изменяемыми?

В геометрии точки и прямоугольники не изменяются. Например, разработчики библиотеки java.awt некорректно сделали геометрические классы изменяемыми. Вместо изменения координат точки корректнее было бы изменить, какая точка представляет собой верхний левый угол окна.

Изменяемым структурам существуют альтернативы. Математические множества и словари изначально не подразумевают изменяемости.Использование чисто функциональных структур данных, которые возвращают новые версии объектов вместо изменения старых, может быть эффективным и более подходящим. Например, операция “добавить элемент” возвращала бы новое множество, а не изменяла бы старое.

Если же предметная область требует наличия изменяемых объектов, необходимо не забывать о них (особенно актуально в многопоточных средах, где данные объекты могут изменяться хаотично и ведут к непредсказуемому поведению программы).

Код в многопоточности — не статическая инструкция, а последовательность событий с ненулевой длительностью, где каждый вызов метода или присваивание занимают время. Понимание природы изменяемого состояния позволяет заметить потенциальные ошибки в коде еще на ранних этапах.

10.11 Инкапсуляция

Изменяемое состояние вызывает сложности, когда:

- слишком многие компоненты могут его изменять
- слишком многие компоненты могут его читать

Это затрудняет координацию изменений, нарушает модульность и усложняет код.

Необходимо ограничить доступ к изменяемому объекту, предоставив клиентам минимально возможный набор методов для его изменения и чтения.

Компоненты, читающие состояние, должны:

- либо быть тесно связаны с компонентами, изменяющими его,
- либо работать с состоянием только тогда, когда оно неизменно,
- либо быть спроектированы с учетом того, что состояние может измениться в любой момент.

10.11.1 Не возвращайте изменяемые объекты из методов для чтения

- Методы, предназначенные для получения свойства объекта (например, даты создания), должны возвращать неизменяемые объекты.
- Если это невозможно, возвращайте копии объектов.

Примеры:

- Возвращать объект класса `Calendar` напрямую недопустимо (это изменяемый объект). Вместо этого используйте `long`, представляющий дату в миллисекундах с 1 января 1970 года.
- При возврате коллекций используйте неизменяемые обертки (`Collections.unmodifiableList()` в Java).
- Убедитесь, что составляющие элементы коллекции также неизменяемы.

10.11.2 Не возвращайте массивы — используйте коллекции

- Массивы не обеспечивают инкапсуляции, так как клиенты могут изменять их элементы.
- Коллекции позволяют:
  - ограничивать доступ через переопределенные методы,
  - гарантировать атомарность операций (в многопоточности).

10.11.3 Предоставляйте интерфейсы, соответствующие предметной области

Методы изменения состояния должны отражать действия, характерные для предметной области. Например, в приведенном выше классе `BankAccount` необходимо использовать операции `deposit` и `withdraw` вместо `getMoney` и `setMoney`. Это позволит сохранить инварианты (например, ненулевой баланс), а также централизовать логику контроля за состоянием внутри объекта.

Также включайте атомарные операции в свой код (такие как `AtomicInteger.addAndGet` и `ConcurrentMap.putIfAbsent`)

10.12 Двухфазный цикл жизни и заморозка

Некоторые объекты проходят две четкие фазы в своем жизненном цикле:

- Фаза записи (накопления): информация записывается в объект, но не читается извне.
- Фаза чтения: объект становится неизменяемым, предоставляя доступ только для чтения.

Важные аспекты:

- Запись и чтение не пересекаются во времени.
- На переходе между фазами можно выполнить подготовку данных (кэширование, построение индексов) для ускорения чтения.

Рассмотрим подходы к реализации двухфазного цикла:

10.12.1 Статический подход

Четкое разделение интерфейсов для фаз записи и чтения

```
public interface ReadFacet {
    Foo getFoo(int fooId);
    Bar getBar();
}

public interface WriteFacet {
    void addQux(Qux qux);
    void setBaz(Baz baz);
    ReadFacet freeze();
}

class Item implements WriteFacet {
    // ...
    ReadFacet freeze() {
        return new FrozenItem(myData);
    }
}

class FrozenItem implements ReadFacet {
    FrozenItem(ItemData data) {
        this.data = data.clone();
        prepareCachesAndBuildIndexes();
    }
    // ...
}
```

- Клиент использует интерфейс WriteFacet для записи данных.
- После завершения записи вызывается метод `freeze()`, возвращающий объект ReadFacet для чтения.
- Гарантируется, что объект для чтения (ReadFacet) независим от объекта записи (`WriteFacet`).

Зачастую организовывать 2 лишних интерфейса неудобно, иногда такой возможности нет. Потому существует второй подход.

10.12.2 Динамический подход

Один интерфейс, но с разным поведением в зависимости от текущей фазы.

```
class Item implements QuxAddableAndFooGettable {
    private boolean isFrozen = false;

    void addQux(Qux qux) {
        if (isFrozen) throw new IllegalStateException();
        // ...
    }

    Foo getFoo(int fooId) {
        if (!isFrozen) throw new IllegalStateException();
        // Efficiently get foo using caches/indexes
    }

    void freeze() {
        prepareCachesAndBuildIndexes();
        isFrozen = true;
    }
}
```

- В фазе записи методы чтения выбрасывают исключение.
- В фазе чтения методы записи становятся недоступными.
- Проверки состояния (`isFrozen`) обеспечивают корректность.

Пример из практики: кластеризация документов.

Сценарий:

- Документы представлены разреженными битовыми векторами.
- На фазе записи векторы заполняются: устанавливаются/сбрасываются биты.
- На фазе чтения производится только расчет попарных расстояний между векторами, для чего строится специальная структура (например, «пирамида»).

Результаты:

- Построение «пирамиды» сразу после записи позволяет ускорить вычисления в два порядка.
- Переход к фазе чтения делает объект неизменяемым, упрощая использование и повышая производительность.

Двухфазный цикл жизни — мощный инструмент для проектирования структурированных и эффективных систем. Даже без использования заморозки четкое разделение на фазы записи и чтения помогает упрощать код и снижать вероятность ошибок.

10.13 Превращение изменяемой настройки в аргумент

Изменяемые данные часто применяются для настройки объекта через методы-сеттеры, после чего объект выполняет свою работу. Это похоже на двухфазный цикл жизни, но:

- Настройка сосредоточена в одном месте.
- Вся последовательность вызовов проходит за короткий промежуток времени.

Однако такой подход может привести к проблемам, особенно в многопоточной среде, где объект может быть разделяемым.

Пример:

```
// Комментарий на русском языке
class Database {
    void setLogin(String login);
    void setPassword(String password);
    Connection connect() throws InvalidCredentialsException;
}
```

Проблемы:

- Если объект `Database` используется несколькими клиентами, вызовы `setLogin`, `setPassword` и `connect` могут запутаться.
- Один пользователь может получить соединение, настроенное чужими данными.

Решение: избавиться от изменяемости. Вместо сеттеров настройки передаются как аргументы:

```
class Database {
    Connection connect(String login, String password)
            throws InvalidCredentialsException;
}
```

Преимущества данного подхода:

- Интерфейс становится более предсказуемым и безопасным.
- Исключается путаница с настройками между клиентами.
- Логика становится проще для тестирования и поддержки.

10.14 Концентрация изменений во времени

Вместо последовательности мелких изменений над объектом выполняется одно крупное изменение.

Цели:

- Устранение промежуточных состояний: предотвращает проблемы, когда объект оказывается в некорректном или неопределенном состоянии между шагами.
- Устранение необходимости протоколов: исключает сложные правила порядка изменений, которые клиент обязан соблюдать.

Проблемы мелких изменений:

- Промежуточные состояния: при внесении изменений объект может находиться в частично настроенном состоянии, что приводит к путанице между клиентами (транзакции передают привет).
- Протоколы последовательности: если изменения взаимозависимы, клиенту приходится учитывать порядок их выполнения. Это неудобно, особенно если изменения вводятся динамически (например, из файлов или других источников).

Пример: библиотека бизнес-правил

```
interface RuleEngine {
    void addRule(String varName, String formula)
            throws ParseException, UndefinedVariableException;
```

```
    double computeValue(String varName);
}
```

Проблемы:

- Клиент должен добавлять правила в порядке топологической сортировки (сначала независимые, затем зависимые).
- Если порядок нарушен, вызывается исключение `UndefinedVariableException`.
- Все вычисления зависят от корректности последовательности, что затрудняет работу с большим числом правил.

Решение: концентрирование изменений. Новый интерфейс:

- Все правила передаются за один вызов, что инкапсулирует сложность добавления зависимостей.
- Вся логика обработки правил вынесена внутрь библиотеки.

```
interface RuleParser {
    RuleSet parseRules(Map<String, String> var2formula)
            throws ParseException, CircularDependencyException;
}

interface RuleSet {
    double computeValue(String varName);
}
```

Конструирование набора правил заключено в `parseRules` — он сам выполняет топологическую сортировку передаваемых ему пар и ищет циклические зависимости.

10.15 Концентрация изменений в пространстве

Необходимо устранить необходимость в поддержании инвариантов, охватывающих несколько объектов (меньше охват → проще сохранять → меньше шансов, что что-то разрушит инвариант, изменив один из составляющих его объектов). Например, сложность реализации двусвязных списков в значительной степени проистекает из того, что каждой операции, затрагивающей конкретный узел, необходимо заботится о его соседях слева и справа (в случае двусвязного списка такая проблема нерешаема).

Пример: артефакты в многопользовательской ролевой игре

```
class Artifact {
        Player getOwner();
        void setOwner(Player player);

        Picture getPicture();

        int getStrengthBoost();
        int getHealthBoost();
        int getDexterityBoost();
        int getManaBoost();
}
class Player {
        List<Artifact> getArtifacts();
        void addArtifact(Artifact a);
        void dropArtifact(Artifact a);
        void passArtifact(Artifact a, Player toWhom);
}
```

У игрока есть лист артефактов, которые у него есть в инвентаре. У артефакта есть хозяин. Получаем инвариант: если `a.getOwner() == p`, то `p.getArtifacts().contains(a)`, и наоборот.

Такой инвариант поддерживать непросто.

Во-первых, если артефакт никем не подобран, то `a.getOwner() == null`. Когда игрок подбирает артефакт, вызываются `a.setPlayer(p)` у артефакта и `p.addArtifact(a)` у игрока.

Во-вторых, для каждого артефакта, присущего на карте мира, нужен отдельный экземпляр `Artifact` с разными полями (использовать один и тот же объект для одинаковых артефактов нельзя — это же разные артефакты, и они могут быть у разных игроков) — это большие затраты памяти.

В-третьих, если в многопользовательской игре есть расы, и каждая раса получает свои бонусы от артефактов (`strength`/`health`/`dexterity`/`mana`), то для получения бонуса артефакту нужны данные о расе игрока $\rightarrow$ укрепляем зависимость между объектами.

Решение: разрыв зависимости между игроком и артефактом

- Убрать из `Artifact` методы `getOwner()` и `setOwner()`
- Хранить все характеристики (`strengthBoost`, `healthBoost` и т.п.) как неизменяемые.
- Сделать промежуточный класс `ArtifactRules`, который позволит получать характеристики с помощью объектов артефакта и игрока (уберем зависимости игрока от артефактов).

Теперь `Artifact` — чисто информационный объект, и это решает все перечисленные выше проблемы.

10.16 Многопоточные техники

Написание корректного многопоточного кода связано с проверкой большого числа трасс выполнения. Рассмотрим следующие техники для уменьшения их количества:

10.16.1 Критические секции

Использование критической секции превращает блок кода из нескольких состояний в единый атомарный переход.

Пример:

```
synchronized(lock) {
    // Critical section code
    sharedResource.update();
}
```

Результат: Число трасс уменьшается, так как действия в критической секции выполняются атомарно. Недостаток: Критические секции блокируют потоки, что снижает параллелизм.

10.16.2 Атомарные операции

Атомарные операции минимизируют необходимость явной синхронизации. Аппаратно реализованные примеры:

- Compare-and-exchange (CAS): Проверяет и изменяет значение только если оно соответствует ожидаемому.

- Get-and-add: Увеличивает значение и возвращает предыдущее.

Важно предоставлять интерфейсы, скрывающие многопоточность и гарантии атомарности.

Примеры:

```
balance -= amount; // Withdraw
balance += amount; // Deposit
```

Использовать:

```
account.transfer(toAccount, amount);
```

Вместо:

```
if (!set.contains(item)) {
    set.add(item);
}
```

Использовать:

```
set.addIfAbsent(item);
```

Пример API с атомарными операциями:

```
class Account {
    public synchronized void transfer(Account to, int amount) {
        this.balance -= amount;
        to.balance += amount;
    }
}
```

10.16.3 Локализация изменяемого состояния

Локализация изменений позволяет минимизировать взаимодействие потоков, так как состояние изменяется локально, а затем применяется атомарно. Пример:

Вместо изменения объекта несколькими потоками:

```
sharedResource.updatePartA();
sharedResource.updatePartB();
```

Сначала выполнить вычисления локально:

```
LocalState newState = computeNewState();
sharedResource.apply(newState);
```

Результат: Уменьшается количество трасс, связанных с изменением глобального состояния.

10.16.4 Навязывание эквивалентности трасс

Стирание различий между трассами уменьшает их количество. Для этого используются свойства операций:

- Идемпотентность: операция, будучи выполненная несколько раз, имеет тот же эффект, что и при однократном выполнении
- Коммутативность: порядок последовательности из нескольких аргументов не имеет значения

Пример: интернет-магазин

```
class Shop {
        void buy(Request request, int productId, int quantity) {
                Session session = request.getSession();
                Order order = new Order(
                        session.getCustomer(), productId, quantity);
                database.saveHistory(order);
                billing.bill(order);
        }
}
```

У пользователя может быть плохой интернет, потому при выполнении функции buy может произойти потеря соединения. Пользователь нажмет кнопку “Купить” еще раз, все заработает, но в итоге он купит 2 одинаковых пылесоса и потратит в 2 раза больше денег :C

Исправим нарушение идемпотентности:

```
class Session {
        int stateToken;

        void advance() {
                ++stateToken;
        }
        int getStateToken() {
                return stateToken;
        }
}
class Shop {
        Page generateOrderPage(Request request) {
                Session session = request.getSession();
                session.advance();
                ...<INPUT type='hidden'
                                value='"+session.getStateToken()+"'>...
        }

        void buy(Request request, int productId, int quantity) {
                Session session = request.getSession();
                if(session.currentStateToken() !=
                   request.getParamAsLong("stateToken"))
                {
                        return;
                }
                Order order = new Order(
                        session.getCustomer(), productId, quantity);
                database.saveHistory(order);
                billing.bill(order);

                session.advance();
        }
}
```

Теперь сделать несколько раз заказ с одной страницы невозможно — счетчик контролирует однократное выполнение функции `buy()`. Проблема, указанная выше, не возникнет.

10.17 Подведем итоги

Итак, сегодня мы узнали о проблемах, связанных с использованием изменяемого состояния в программировании. Непредсказуемость изменяемого состояния усложняет отладку, тестирование и сопровождение кода.

Мы рассмотрели функциональные подходы как хорошую альтернативу: отсутствие изменяемого состояния помогают минимизировать непредсказуемость, улучшая надежность и, зачастую, ещё и читаемость кода.

Борьба с изменяемым состоянием не только улучшает качество программного обеспечения, но и снижает когнитивную нагрузку на разработчиков. Использование концепций из функционального мира делает программы более предсказуемыми, а процесс их создания — более простым и управляемым.

10.18 Литература

1. Евгений Кирпичев, 2009 — Изменяемое состояние: опасности и борьба с ними, Практика функционального программирования: https://www.fprog.ru/2009/issue1/eugene-kirpichov-fighting-mutable-state/

11 Авторы

Данные конспект лекций в значительной степени является результатов совместного труда следующих людей:

- Андриенко Сергей Вячеславович
- Долгих Александр Алексеевич
- Комягин Дмитрий Анатольевич
- Мальков Павел Александрович
- Нигаматуллин Степан Русанович
- Ефремов Марк Андреевич
- Афанасьев Кирилл Александрович
- Сорокин Артём Николаевич
- Глотов Егор Дмитриевич
- Ефимов Арслан Альбертович
- Гайдеров Ярослав Игоревич
- Булко Егор Олегович
- Лянгузов Дмитрий Максимович
- Мартыненко Вадим Андреевич