Практическая работа №7. Типы данных для результатов вычислений.

Теоретическая часть

Последние десятилетия существовал тренд на «ослабление» типов данных целого ряда языков, в числе языков, ворвавшихся в топ языков программирования TIOBE (https://www.tiobe.com/tiobe-index/), росли JavaScript, Python.

Подобные языки позволяют быстро писать код, уменьшают объём «служебных» конструкций и часто оказываются удобны для прототипирования. Однако со временем стало понятно, что отсутствие строгой информации о типах создаёт серьёзные проблемы:

- сложнее выполнять статический анализ программ;
- часть ошибок обнаруживается только во время выполнения и в принципе не может быть обнаружена на стадии компиляции;
- снижается качество подсказок и сообщений об ошибках.

Поэтому сейчас все чаще специалисты стали задумываться о том, что типы для переменных должны определяться заранее, либо хотя бы должна существовать возможность для их аннотации, так как это открывает существенные возможности для статического анализа кода (и, кстати, заметно более удобно для работы с инструментами ИИ, как бы мы к ним не относились).

Одним из наиболее известных примеров является TypeScript — язык, разработанный компанией Microsoft как надстройка над JavaScript. Основное его нововведение — возможность явно указывать типы переменных, параметров функций и возвращаемых значений.

```
function add(a, b) {

    return a + b;

}
```

Из самого кода неясно:

- какие типы ожидаются;
- что произойдёт при передаче строки;
- является ли функция корректной.

В TypeScript:

```
function add(a: number, b: number): number {
    return a + b;
}
```

Похожая тенденция наблюдается и в Python. Хотя сам язык остаётся динамически типизированным, начиная с Python 3 появились аннотации типов:

```
def add(a: int, b: int) -> int:
    return a + b
```

Изначально они носили скорее документирующий характер, однако затем вокруг них начали формироваться полноценные системы статического анализа.

Одним из примеров является basedpyright — строгий анализатор типов и языковой сервер для Python, развивающий идеи проекта Pyright. Он активно использует аннотации типов и может требовать их обязательного указания, обнаруживая множество ошибок ещё до запуска программы.

Таким образом, даже в динамических языках типы постепенно начинают рассматриваться не как «лишняя формальность», а как важный инструмент описания программы.

> **Совет**
>
> Тип функции становится кратким формальным описанием её поведения.

Особенно важны типы результатов функций. Современные языки всё чаще стараются явно описывать:

1) может ли функция не вернуть результат,

2) возможна ли ошибка в ходе ее выполнения,

3) сколько результатов может быть и

4) обязана ли программа обработать возможный отказ.

Тем более, если рассматривать новые языки, такие как Rust, Kotlin, TypeScript, Zig, то как правило (за исключением случаев, когда язык осознанно удерживается относительно простым: Go, C, Lua) подразумевается, что помимо типов, содержащих значения, нужны некие «контейнеры», сигнализирующие о кардинальности результатов. Не отстают и современные «старые» «тяжеловесные» языки, такие как C++ последних версий и Java.

Одиночный тип данных

Наиболее простой случай — функция всегда возвращает одно значение фиксированного типа.

Например:

```
fun length(s: String): Int
```

В подобных случаях программист заранее знает, что результат вычисления существует и всегда принадлежит одному и тому же типу.

Как правило, различают, по крайней мере, целочисленный тип, тип чисел с плавающей точкой, строки. Иногда также различают отдельные символы (C++), а также булевский тип данных.

Перечисление

Во многих задачах функция может возвращать не произвольное значение, а одно из ограниченного набора вариантов.

Например, состояние сетевого соединения:

```
enum class ConnectionState {
    CONNECTING,
    CONNECTED,
    DISCONNECTED,
    TIMEOUT
}
```

Или C++:

```
enum class ConnectionState {
    Connecting,
    Connected,
    Disconnected,
    Timeout
};

ConnectionState connectionState();
```

Подобные типы называются перечислениями (enum). Они позволяют явно ограничить множество допустимых состояний программы.

Технически перечисления всегда можно реализовать «на коленке» из набора целочисленных констант, но это «плохой» путь, в котором компилятор никак не поможет Вам в разработке целостной программы. Так лучше поступать только если Вы имеете дело с простым, ограниченным языком программирования.

Перечисления особенно полезны в случаях, когда заранее известны все возможные варианты результата вычисления.

Структура (запись)
Иногда результат вычисления состоит не из одного значения, а из нескольких связанных полей.

Например, функция может возвращать информацию о погоде в некотором городе:

```
data class WeatherInfo(

    val temperature: Double,

    val humidity: Int,

    val pressure: Int

)

fun currentWeather(city: String): WeatherInfo
```

Пример на C++:

```
struct WeatherInfo {
    double temperature;
    int humidity;
    int pressure;
};

WeatherInfo currentWeather(
    const std::string& city
);
```

Подобные типы называются структурами (или записями). Они позволяют объединять несколько связанных значений в единое значение.

Структуры особенно полезны в случаях, когда разные поля логически относятся к одному результату вычисления.

Кортеж как анонимная структура

Во многих языках программирования существуют кортежи — специальные типы данных, позволяющие объединять несколько значений без создания отдельной структуры.

Например, функция может возвращать координаты точки:

```
fun point(): Pair<Int, Int>
```

Пример на C++:

```
std::tuple<int, int> point();
```

Кортежи удобны для временного объединения значений, особенно если структура используется только в одном месте программы.

Однако в крупных программах структуры обычно предпочтительнее, так как поля получают явный смысл.

Также во многих языках программирования существует операция `zip`, позволяющая объединять несколько последовательностей в последовательность кортежей.

Например, в Python:

```
names = ["Анна", "Борис", "Виктор"]
scores = [5, 4, 5]

pairs = list(zip(names, scores))
```

Результат:

```
[
    ("Анна", 5),
    ("Борис", 4),
    ("Виктор", 5)
]
```

Подобная операция существует и в других языках, например, в Kotlin:

```
val names = listOf("Анна", "Борис")
val scores = listOf(5, 4)

val result = names.zip(scores)
```

или в Rust:

```
names.iter().zip(scores.iter())
```

Операция zip особенно полезна при попарной обработке элементов:

- имён и оценок;
- делимых и делителей;
- координат x и y;
- дат и значений измерений.

При этом результатом zip обычно является список или последовательность кортежей.

Один или ноль

Во многих задачах функция может не найти результат:

- поиск пользователя по логину;
- поиск первого простого числа в диапазоне;

- поиск ближайшей свободной аудитории;
- поиск символа в строке.

В подобных случаях используются специальные типы:
- Optional (C++),
- Option (Rust),
- nullable-значения (Kotlin, Java, C#),
- Maybe (Haskell).

Пример на Kotlin:

```
fun findStudent(id: Int): Student?
```

Знак ? означает, что функция может вернуть либо объект Student, либо null.

Похожая ситуация сложилась и в других языках, например, C# 8.

В C++17 (и новее) эту же функцию выполняет `std::optional`:

```
std::optional<Student> findStudent(int id);
```

В Rust этот контейнер называется Option:

```
fn find_student(id: i32) -> Option<Student>
```

Функция возвращает первое простое число из списка либо null, если простых чисел нет.

```
std::optional<unsigned int> findPrime(
    const std::vector<unsigned int>& vec
);
```

Пример на Rust:

```
fn find_uppercase(text: &str)
    -> Option<char>
```

Важно понимать, что отсутствие результата не обязательно является ошибкой. Например, в строке действительно может не оказаться заглавной буквы.

Подобные типы позволяют явно описывать возможность отсутствия значения и делают поведение функции более понятным.

Значение или какая-то ошибка

Иногда вычисление может завершиться неудачей:
- файл не найден;
- ошибка сети;
- неверный формат данных;
- недостаточно прав доступа;
- деление на ноль.

Во многих современных языках для подобных случаев используются специальные типы результата, содержащие либо успешное значение, либо описание ошибки.

Пример на Rust:

```
fn read_config(
    path: &str
) -> Result<String, IOError>
```

Подобный тип означает:

- либо функция успешно вернула содержимое файла,
- либо произошла ошибка IOError, причем какая конкретно — мы заранее не знаем.

Пример на Zig:

```
fn parsePort(text: []const u8)
    error{InvalidFormat, OutOfRange}!u16
```

Пример на C++23:

```
std::expected<Image, ImageLoadError>
loadImage(const std::string& path);
```

В функциональных языках (Haskell) или функциональных библиотеках языков программирования общего назначения (в частности, в Arrow Kotlin) часто используется тип `Either`.

Подобные типы позволяют: явно описывать причины ошибок, избегать скрытых исключений, заставлять программиста учитывать возможные неудачные варианты выполнения программы.

В отличие от Option, Result позволяет различать различные типы ошибок, избегать скрытых исключений, явно описывать причины некорректной работы программы.

Важно различать отсутствие результата и ошибку выполнения. Отсутствие пользователя в базе данных это нормальная ситуация, а вот невозможность подключиться к базе данных — уже ошибка.

Список

Если у Вас может быть возвращено не одно значение, а несколько значений одинакового типа `T` или вовсе ни одного, в таком случае используется список в широком смысле этого слова. Под «списком» следует понимать любую коллекцию из множества элементов:

- статический массив,
- динамический массив,
- связный список,
- стек,
- очередь,
- и др.

Подобные структуры значительно отличаются:
- по способу хранения элементов;
- по скорости чтения;
- по скорости вставки;
- по стоимости удаления элементов;
- по эффективности доступа по индексу.

Например:
- массив обеспечивает быстрый доступ по индексу;
- связный список удобен для вставки элементов;
- стек позволяет работать по принципу LIFO («последним пришёл — первым вышел»);
- очередь — FIFO («первым пришёл — первым вышел»).

Однако на уровне теории типов данных все подобные структуры объединяет одна идея: возможность хранить произвольное количество элементов одного типа.

С точки зрения теории типов, список часто описывается рекурсивно.

Например, связный список можно описать как:
- текущее значение;
- ссылку на следующий элемент списка.

На языке Kotlin подобная структура может быть описана следующим образом:

```
data class Node<T>(
    val value: T,
    val next: Node<T>?
)
```

Пример на Rust:

```
struct Node<T> {
    value: T,
    next: Option<Box<Node<T>>>,
}
```

Именно благодаря рекурсивности подобные структуры способны хранить:

- 0 элементов;
- 1 элемент;
- произвольное количество элементов.

Антипаттерн «Строка как универсальный тип данных»

Начинающие программисты нередко пытаются использовать строку (`String`) в качестве «универсального» типа данных.

Например:
- хранить дату как строку;
- хранить денежную сумму как строку;
- хранить IP-адрес как строку;
- хранить список значений как одну длинную строку;
- хранить JSON «как текст» без разбора структуры.

Подобный подход кажется удобным, так как строка способна содержать практически любые данные, и, кроме того, будучи последовательностью (списком) символов, она «автоматически» поддерживает:
- отсутствие символов (0 элементов),
- один символ,
- произвольное количество символов (`N`).

Фактически строка сама является контейнером значений типа `Char`, а с помощью парсинга (разбора строки на части и интерпретации значений) может «хранить» и другие значения.

Однако именно эта универсальность нередко приводит к злоупотреблению строками как «универсальным контейнером для любых данных». В большинстве случаев подобный подход считается антипаттерном проектирования.

Грубо говоря, «вы делаете на коленке свой язык», вместо того чтобы пользоваться тем, что уже дано разработчиками и компилятором/интерпретатором/языковым сервером.

Например, следующий код практически ничего не сообщает о смысле данных:

```
data class User(
    val birthDate: String,
    val phone: String,
```

```
    val money: String
)
```

Например, денежные суммы нельзя корректно хранить как строки, так как программа не сможет выполнять над ними арифметические операции без дополнительного разбора текста.

Однако существует и обратная ошибка: попытка представить данные числом там, где число на самом деле является идентификатором.

Например:

- номер телефона;
- почтовый индекс;
- номер дома;
- номер паспорта.

Хотя подобные значения часто состоят из цифр, математического смысла они обычно не имеют, и нередко в них включаются буквы и прочие символы.

Поэтому тип данных должен отражать не внешний вид значения, а его смысл в предметной области программы.

Практическая часть

Решить три варианта задач, составив прототипы (заголовки) функций без реализации самого кода на любом современном языке программирования. Один взять по номеру в группе N, другой по последним цифрам студенческого билета, и третий по желанию.

Совет

В решении оценивается не реализация алгоритма, а выбор типов возвращаемых значений и декомпозиция задачи на функции.

1. Знаки решений квадратного уравнения в вещественных корнях. Параметры a, b, c вводятся с клавиатуры.

   Рекомендуется задачу на независимые функции: решение_квадратного_уравнения, получение_данных, знак_числа.

2. Список товаров по цене «ниже средней». Указание: реализовать прототипы функций «список товаров по цене ниже средней» и функции «средняя цена».

3. При регистрации формы пользователь может не указывать номер дома для традиционной доставки курьером. Напишите, какой тип данных должен хранить это поле в структуре и почему?

4. Собрать статистику, сколько раз упоминаются те или иные числа в каждом игровом никнейме игроков некоей игры

   Уточнение: нужны функции «числа в строке», «список игроков». Допустим, что полный список игроков возвращается через API по сетевому запросу с серверов игры, и никнеймы обязаны иметь все игроки.

5. Функция попарного вычисления остатков от деления для двух списков чисел: первое делимое с первым делителем, второе делимое со вторым делителем, …

6. Функция поиска студента с наибольшим количеством долгов по оценкам (среди оценок в ведомости ставится «2»). Предусмотреть функции, необходимые для получения этих данных.
7. Найти все строки файла, содержащие слово «ERROR». 1) Реализовать прототип функции чтения файла по всем строкам и 2) функции поиска в строках искомой фразы (например, ERROR)
8. Функция проверки корректности IPv4-адреса и ее маски с возвращением его типа (широковещательный, локальный loopback, целая сеть, обычный)

   Уточнение: различать разные типы ошибок составления IPv4-адреса
9. Функция поиска фамилии, имени и отчества (тройкой значений) студента по номеру зачётки.
10. Функция поиска даты рождения клиента магазина по номеру телефона.