Метапрограммирование

Мультипарадигменный подход в инженерии программного обеспечения

© Тимур Шемсединов, Сообщество Метархия

Киев, 2015 — 2022

Аннотация

Все программы - это данные. Одни данные интерпретируются, как значения, а другие - как типы этих значений, третьи - как инструкции по обработке первых двух. Любые парадигмы и техники программирования - это лишь способ формировать метаданные, дающие правила и последовательность потока обработки других данных. Мультипарадигменное программирование берет лучшее из всех парадигм и строит из них синтаксические конструкции, позволяющие более понятно и удобно описать предметную область. Мы связываем высокоуровневые DSL (доменные языки) с низкоуровневыми машинными инструкциями через множество слоев абстракций. Тут важно не фанатично следовать одной парадигме, а наиболее эффективно отображать задачу для исполнения на машинном уровне. Наиболее эффективно - это с меньшим количеством слоев и зависимостей, наиболее удобно для понимания человеком, для поддержки и модификации, обеспечения надежности и тестируемости кода, расширяемости, переиспользования, ясности и гибкости конструкций метаданных на каждом уровне. Мы полагаем, что такой подход позволит получать как быстрые первые результаты в разработке каждой задачи, так и не терять темпов при большом потоке изменений на этапах, когда проект уже достиг высокой зрелости и сложности. Мы постараемся рассмотреть приемы и принципы программирования из разных парадигм через призму метапрограммирования и не столько изменить этим программную инженерию, как расширить ее осмысление новыми поколениями инженеров.

Оглавление

(машины состояний)

11.4. Специализированные языки для предметных областей (DSL)

11.5. Программирование на потоках данных

11.6. Метапрограммирование

11.7. Динамическая интерпретация метамодели

12. Базы данных и постоянное хранение

12.1. История баз данных и навигационные базы данных

12.2. Ключ-значение и другие абстрактные структуры данных

12.3. Реляционная модель данных и ER-диаграммы

12.4. Бессхемные, объектно- и документо-ориентированные базы данных

12.5. Иерархическая модель данных и графовые базы данных

12.6. Колоночные базы данных и in-memory базы данных

12.7. Распределенные базы данных

13. Распределенные системы

13.1. Межпроцессное взаимодействие

13.2. Бесконфликтные реплицированные типы данных (CRDT)

13.3. Согласованность, доступность и распределенность

13.4. Стратегии разрешения конфликтов

13.5. Протоколы консенсуса

13.6. CQRS, EventSourcing

1. Введение

Постоянное переосмысление своей деятельности, даже самой простой, должно сопровождать инженера всю его жизнь. Привычка записывать свои мысли словами и оттачивать формулировки очень помогает в этом. Текст этот появился как мои отрывочные заметки, написанные в разные годы, которые я накапливал и критически вычитывал десятки раз. Часто, я не соглашался с самим собой, перечитывая отрывок после того, как он полежит некоторое время. Поэтому, я доводил текст до того, пока сам не соглашался с написанным после продолжительных периодов выдержки материала. Своей задачей я принял писать как можно более кратко и неоднократно переписывал большие фрагменты, находя, что их можно выразить короче. Структура текста и оглавление начали появляться после первого года преподавания, но на десятом году я решил выложить все материалы не только в виде открытых видеолекций, как делал уже около пяти лет, но и в виде текста. Это позволило участвовать в формировании книги всем желающим из сообщества Метархия, быстро находить опечатки и неточности благодаря читателям, а также многим просто удобнее воспринимать в виде книги. Актуальную версию всегда можно найти на https:// github.com/ HowProgrammingWorks/ Book, она будет дополняться постоянно. Прошу присылать запросы на исправления и дополнения в issues: https:// github.com/ HowProgrammingWorks/ Book/ issues на английском языке, новые идеи в discussions: https:// github.com/ HowProgrammingWorks/ Book/ discussions) на любом языке, а свои дополнения и исправления оформлять в виде pull-request в репозиторий книги.

Программирование — это искусство и инженерия решения задач при помощи вычислительной техники.

Инженерия, потому, что оно призвано извлекать пользу из знаний, а искусство, потому, что знаниями программирование на современном этапе развития, к сожалению, не ограничивается и вынуждено прибегать к интуиции и слабо осмысленному личному опыту. Задача программиста не в нахождении математически верного решения, а в отыскании обобщенного механизма решения, способного нас приводить к нахождению приемлемого решения за ограниченное время в как можно большем классе задач. Другими словами, в нахождении абстрактного класса решений. Не все

парадигмы программирования предполагают решение пошаговое, но физическая реализация вычислительной техники и природа человеческого мышления предполагают пошаговость. Сложность в том, что эти действия далеко не всегда сводятся к машинным операциям и вовлекают внешнее взаимодействие с устройствами ввода/ вывода и датчиками, а через них, с внешним миром и человеком. Это обстоятельство создает большую неопределенность, которая не позволяет математически строго доказать правильность способа решения всех задач и, тем более, строго вывести такое решение из аксиом, как это характерно для точных наук. Однако, отдельные алгоритмы могут и должны быть выведены аналитически, если они сводимы к чистым функциям. То есть, к функциям, которые в любой момент для определенного набора входных данных однозначно дают один и тот же результат. Чистая функция не имеет истории (памяти или состояния) и не обращается ко внешним устройствам (которые могут такое состояние иметь), может обращаться только к другим чистым функциям. От математики программирование унаследовало возможность находить точные решения аналитически, да и сама вычислительная машина функционирует строго в рамках формального математического аппарата. Но процесс написания программного кода не всегда может быть сведен к формальным процедурам, мы вынуждены принимать решения в условиях большой неопределенности и конструировать программы инженерно. Программист ограничен и временем разработки программы, поэтому, мы сокращаем неопределенность благодаря введению конструктивных ограничений, не являющихся при этом строго выводимыми из задачи и основанных на интуиции и опыте конкретного специалиста. Проще говоря, за неимением оптимального алгоритма, программист может решать задачу любым способом, который дает приемлемые результаты за разумное время, и который может быть реализован за такое время, пока задача еще остается актуальной. В таких условиях мы должны принимать во внимание не только меру приближения решения к оптимальному, но и знания программиста, владение инструментарием и другие ресурсы, имеющиеся в наличии. Ведь даже доступ к знаниям уже готовым программным решениям ограничен авторским правом, правами владения исходным кодом и документацией, соответствующими лицензионными ограничениями, не только на программные продукты, но и на книги, видео, статьи, обучающие материалы, и т.д. Все это существенно усложняет и замедляет развитие отрасли, но со временем доступность знаний необратимо растет, они просачиваются в

свободное хождение в сети через популяризаторов, энтузиастов и движение свободного программного обеспечения.

1.1. Подход к изучению программирования

Многие думают, что главный навык программиста — это писать код. На самом деле, программисты чаще читают код и исправляют его. А основные критерии качества кода — понятность, читаемость и простота. Как говорил Гарольд Абельсон: «Программы должны писаться для людей, которые будут их читать, а машины, которые будут эти программы исполнять — второстепенны».

Главные навыки программиста — это чтение и исправление кода.

Каждая тема содержит примеры хорошего кода и плохого кода. Эти примеры собраны из практики программирования и ревью проектов. Специально заготовленные примеры плохого кода будут работоспособны, но полны антипаттернов и проблем, которые нужно выявить и исправить. Даже сама первая практическая работа в курсе будет связана с исправлением кода, повышением его читабельности. Если давать традиционные задания (написать функцию по сигнатуре, алгоритм, класс), то начинающий, очевидно, реализует его не лучшим образом, но будет защищать свой код, потому что это первое, что он написал. А если задача будет "взять пример чужого плохого кода, найти проблемы и исправить", не переписать с нуля, а улучшить в несколько шагов, фиксируя и осознавая эти шаги, то включается критический подход.

Исправление плохого кода — один из самых эффективных способов обучения.

Начинающий получает примеры ревью кода и по аналогии стремится исправить и свое задание. Такие итерации повторяются много раз, не теряя критичного настроя. Очень хорошо, если будет наставник, который наблюдает за улучшениями, и может корректировать и подсказывать. Но наставник ни в коем случае не должен делать работу за новичка, а скорее наталкивать его на то, как нужно думать о программировании и где искать решение.

Наставник — незаменим на любом этапе профессионального роста.

Дальше будут идти задания по написанию своего кода. Очень рекомендуем начинающим обмениваться между собой этими решениями для перекрестного ревью. Конечно, перед этим нужно применить линтеры и форматеры кода, которые проанализируют синтаксис, находя в нем ошибки, и выявят проблемные места по большому количеству шаблонов кода. Нужно добиться того, чтобы коллега понимал выраженную тобой мысль, а не тратил время на синтаксис и форматирование.

Применяйте дружественное ревью кода, перекрестное ревью, линтеры и форматеры.

Переходим к упражнениям на снижение зацепления между несколькими абстракциями, потом между модулями, т.е. сделать так, чтобы нужно было как можно меньше знать про структуры данных одной части программы из другой ее части. Снижение языкового фанатизма достигается параллельным изучением с самого начала нескольких языков программирования и переводами с одного языка на другой. Между **JavaScript** и **Python** переводить очень просто, а **C** посложнее будет, но эти три языка, какие бы они ни были, нельзя не включить в курс.

С первых шагов не допускайте никакого фанатизма: языкового, фреймворкового, парадигменного.

Снижение фреймворкового фанатизма — запрет для начинающих использовать библиотеки и фреймворки, и сосредоточиться на максимально нативном коде, без зависимостей. Снижение парадигмального фанатизма — стараться комбинировать процедурное, функциональное, ООП, реактивное и автоматное программирование. Мы постараемся показать, как эти комбинации позволяют упростить паттерны и принципы из GoF и SOLID.

Следующая важная часть курса — изучение антипаттернов и рефакторинга. Сначала мы дадим обзор, а потом будем практиковаться на реальных примерах кода из живых проектов.

1.2. Примеры на языках JavaScript, Python и C

Примеры кода мы будем писать на разных языках, но предпочтение будет отдаваться не самым лучшим, красивым и быстрым, а тем, без которых нельзя обойтись. Мы возьмем **JavaScript**, как самый распространенный, **Python**, потому что есть области, где без него нельзя и **C**, как язык достаточно близкий к ассемблеру, все еще очень актуальный и оказавший самое большое влияние на современные языки по синтаксису и по заложенным в него идеям. Все три очень далеки от языка моей мечты, но это то, что у нас есть. На первый взгляд **Python** очень отличается от **JavaScript** и других C-подобных языков, хотя это только на первый взгляд, мы покажем, что он очень похож на **JavaScript** из-за того, что система типов, структуры данных, а особенно, встроенные коллекции в них очень похожи. Хоть синтаксически, различие в организации блоков кода при помощи отступов и фигурных скобок **{}** бросается в глаза, но на деле, такое различие не так уж значимо, а между **JavaScript** и **Python** гораздо больше общего, чем у любого из них с языком **C**.

Начнем мы не с изучения синтаксиса, а сразу с чтения плохого кода и поиску в нем ошибок. Давайте посмотрим следующие фрагменты, первый будет на **JavaScript**:

```
let first_num = 2;
let secord_num = 3;
let sum = firstNum + secondNum;
console.log({ sum });
```

Попробуйте понять, что тут написано и в чем могут быть ошибки. А потом сравните этот код с его переводом на **C**.

```
#include <stdio.h>

int main() {
  int first_num = 2;
  int secord_num = 3;
  int sum = firstNum + secondNum;
  printf("%d\n", sum);
}
```

Ошибки тут те же самые, их легко может выявить человек, не знающий даже основ программирования, если будет рассматривать код. А следующий фрагмент кода будет на **Python** , он делает абсолютно то же и содержит те же ошибки.

```
first_num = 2;
secord_num = 3;
sum = firstNum + secondNum;
print({ 'sum': sum  });
```

Дальше мы будем часто сравнивать примеры кода на разных языках, искать и исправлять ошибки, оптимизировать код, улучшая в первую очередь его читаемость и понятность.

1.3. Моделирование: абстракции и повторное использование

В основе любого программирования лежит моделирование, то есть создание модели решения задачи или модели объектов и процессов в памяти машины. Языки программирования предоставляют синтаксисы для конструирования ограничений при создании моделей. Любая конструкция и структура, призванная расширить функциональность и введенная в модель, приводит к дополнительным ограничениям. Повышение же уровня абстракции, наоборот, может снимать часть ограничений и уменьшать сложность модели и кода программы, выражающего эту модель. Мы все время балансируем между расширением функций и сверткой их в более обобщенную модель. Этот процесс может и должен быть многократно итеративным.

Удивительно, но человек способен успешно решать задачи, сложность которых превышает возможности его памяти и мышления, при помощи построения моделей и абстракций. Точность этих моделей определяет их пользу для принятия решений и выработки управляющих воздействий. Модель всегда не точна и отображает только малую часть реальности: одну или несколько ее сторон или аспектов. Однако, в ограниченных условиях использования, модель может быть неотличимой от реального объекта предметной области. Есть физические, математические, имитационные и другие модели, но нас будут интересовать, в первую очередь, информационные и алгоритмические модели.

Абстракция — это способ обобщения, сводящий множество различных, но схожих между собой случаев, к одной модели. Нас интересуют абстракции данных и абстрактные алгоритмы. Самые простые примеры абстракции в алгоритмах — это циклы (итерационное обобщение) и функции (процедуры и подпрограммы). При помощи цикла мы можем описать множество итераций одним блоком команд, предполагая его повторяемость несколько раз, с разными значениями переменных. Функции так же повторяются много раз с разными аргументами. Примеры абстракции данных — это массивы, ассоциативные массивы, списки, множества и т.д. В приложениях абстракции нужно объединять в уровни — слои абстракций. Низкоуровневые абстракции встроены в язык программирования (переменные, функции, массивы, события). Абстракции более высокого уровня содержатся в программных платформах, рантаймах, стандартных библиотеках, и внешних библиотеках или их можно построить самостоятельно из простых абстракций. Абстракции так называются потому, что решают абстрактные обобщенные задачи общего назначения, не связанные с предметной областью.

Построение слоев абстракций — это чуть ли не самая важная задача программирования от удачного решения которой зависят такие характеристики программного решения, как гибкость настройки, простота модификации, способность к интеграции с другими системами и период жизни решения. Все слои, которые не привязаны к предметной области и конкретным прикладным задачам, мы будем называть системными. Над системными слоями программист надстраивает прикладные слои, абстракция которых наоборот снижается, универсальность уменьшается и конкретизируется применение, привязываясь к конкретным задачам.

Абстракции разных уровней могут находиться как в одном адресном пространстве (одном процессе или одном приложении), так и в разных. Отделить их один от другого и осуществить взаимодействие между ними можно при помощи программных интерфейсов, модульности, компонентного подхода и просто усилием воли, избегая прямых вызовов из середины одного программного компонента в середину другого, если язык программирования или используемая платформа не заботятся о предотвращении такой возможности. Так следует поступать даже внутри одного процесса, где можно было бы обращаться к любым функциям, компонентам и модулям из любых других, даже если они

логически относятся к разным слоям. Причина этого в необходимости понизить связанность слоев и программных компонентов, обеспечив их взаимозаменяемость, повторное использование и делая возможной их раздельную разработку. Одновременно нужно повышать связность внутри слоев, компонентов и модулей, что обеспечивает рост производительности кода, простоту его чтения, понимания и модификации. Если же нам удастся избегать связанности между разными уровнями абстракций и при помощи декомпозиции добиться того, чтобы один модуль всегда мог быть полностью охвачен вниманием одного инженера, то процесс разработки становится масштабируемым, управляемым и более предсказуемым. Подобная идея положена в основу архитектуры микросервисов, но более общий принцип применим для любых систем, и не важно, будут ли это независимо запущенные микросервисы или модули, запущенные в одном процессе.

Нужно отметить, что чем лучше система распределена, тем лучше она централизована. Потому, как решения задач в таких системах находятся на адекватных уровнях, где уже достаточно информации для принятия решений, обработки и получения результата, отсутствует жесткая связанность моделей разного уровня абстракции. При таком подходе не происходит излишних эскалаций задачи на верхние уровни, избегаются “перегревы” узлов принятия решений, минимизирована передача данных и повышено оперативное быстродействие.

1.4. Алгоритм, программа, синтаксис, язык

Чем же занимается программист в процессе работы? Чем занимается компьютер? Есть много терминов, связанных с программированием, для определенности нам следует выяснить разницу между ними. Самым старым понятием тут является алгоритм, который мы все помним из школьного курса математики (алгоритм Евклида для нахождения наибольшего общего делителя двух целых чисел).

Алгоритм (Algorithm) — это формальное описание порядка вычислений для определенного класса задач за конечное время.

На **JavaScript** нахождение наибольшего общего делителя (или общей меры) можно написать так:

```
const gcd = (a, b) => {
  if (b === 0) return a;
  return gcd(b, a % b);
};
```

Или даже короче, но менее привычно:

```
const gcd = (a, b) => (b === 0 ? a : gcd(b, a % b));
```

Этот простой алгоритм является рекурсивным, т.е. обращается к самому себе для вычисления следующего шага и предусматривает выход, когда **b** доходит до **0**. Для алгоритмов мы можем определять вычислительную сложность, классифицировать их по ресурсам времени и памяти, необходимым для решения задачи.

Программа (Program) — программный код и данные, объединенные в одно целое для вычислений и управления ЭВМ.

У Никлауса Вирта есть книга «Алгоритмы + Структуры данных = Программы». Ее название схватило очень важную истину, которая глубоко запечатлелась не только в мировоззрении читателей, но и в названиях курсов в ведущих ВУЗах, и даже на собеседованиях, когда испытуемого просят сконцентрироваться именно на этих двух вещах. За первые 50 лет существования индустрии программного обеспечения, оказалось, что структуры данных не менее важны, чем алгоритмы. Более того, многие известные программисты делают на них основную ставку, например, известна цитата Линуса Торвальдса: «Плохие программисты беспокоятся о коде. Хорошие программисты беспокоятся о структурах данных и связях между ними». Дело в том, что выбор структур данных во многом предопределяет то, каким будет алгоритм, ограничивает его в рамках вычислительной сложности и семантики задачи, которую программист понимает через данные, разложенные в памяти, гораздо лучше, чем через последовательность операций.

Эрик Рэймонд выразил это так: «Умные структуры данных и тупой

код работают куда лучше, чем наоборот».

Код позволяет найти общий язык.

Однако, тот же Линус Торвальдс сказал нам еще и «Болтовня ничего не стоит. Покажите мне код». Это совсем не противоречит сказанному выше. Я думаю, что тут он имел в виду то, что программный код не допускает двусмысленности. Это универсальный язык, который позволяет программистам находить общий язык даже тогда, когда естественные языки, из- за своей многозначности не позволяют точно понять друг друга, можно сделать это просто взглянув на код.

Инженерия (Engineering) — извлечение практической пользы из имеющихся ресурсов при помощи науки, техники, различных методик, организационной структуры, приемов и знаний.

Я помню, что в первые годы изучения программирования для меня уже было важно, чтобы код использовался людьми, улучшал их жизнь и сам жил долго. Олимпиадные задачи казались мне неинтересными, учебные задачи слишком надуманными, хотелось сконцентрироваться на том, что люди будут запускать на своих компьютерах каждый день: приложения баз данных, формы и таблицы, сетевые и коммуникационные приложения, программы, управляющие аппаратурой, работающие с датчиками, и множество инструментов для самих программистов.

Так же, как и в других инженерных отраслях, в программировании очень важна польза для человека, а не правильность и или стройность концепции. Инженерия призвана использовать научные достижения, а в тех местах, где научных знаний, имеющихся на сегодня, недостаточно, инженерия применяет интуицию, инженерную культуру, метод проб и ошибок, применение неосознанного опыта и опыта, имеющего недостаточное научное осмысление.

В этом и преимущество инженерии и недостаток. Мы имеем множество разных и противоречивых решений одной задачи, мы не всегда знаем почему что- то не работает, но это еще ладно, мы иногда удивляемся, почему что-то работает. Такой подход приводит

к накоплению плохих практик в проектах и такому переплетению хороших и плохих практик, что разделить их очень сложно и часто усилия тратятся повторно на уже решенные задачи. Никлаус Вирт сказал «Программы становятся медленнее быстрее, чем "железо" становится быстрее» и мы часто сталкиваемся с тем, что написать программу заново проще, чем исправлять в ней ошибки.

Инженерия программного обеспечения (Software engineering) — приложение инженерии к индустрии программного обеспечения. Включает архитектуру, исследование, разработку, тестирование, развертывание и поддержку ПО.

Индустрия программного обеспечения превратилась в мощную отрасль промышленности, обросла вспомогательными технологическими практиками, которые позволяют уменьшить влияние ее недостатков, уже приведенных выше и сделать конечный продукт достаточно надежным, чтобы он приносил прибыль, но недостаточно качественным, чтобы можно было выпускать все новые и новые его версии.

«Большинство программ на сегодняшний день подобны египетским пирамидам из миллиона кирпичиков друг на друге и без конструктивной целостности — они просто построены грубой силой и тысячами рабов» // Алан Кей

Программирование (Programming) — это искусство и инженерия решения задач при помощи вычислительной техники.

Тут важно отметить, что вычислительная техника очень сильно влияет на то, как мы программируем, диктует то, какие парадигмы и подходы будут работать эффективнее и будут давать результат, доступный нам по ресурсам, затраченным на программирование и по вычислительным ресурсам, требуемым для исполнения задачи.

Кодирование (Coding) — написание исходного кода программы при помощи определенного синтаксиса (языка), стиля и парадигмы по готовому ТЗ

(техническому заданию).

Разработка может быть разделена на проектирование и кодирование, и это дает более эффективное приложение сил на долгой дистанции, но часто приходится начинать программировать без ТЗ и без предварительного проектирования. Разработанные таким образом системы называются прототипами, MVP (minimum viable product), пилотными системами или стендами. Их польза заключается в проверке гипотез о полезности для потребителя или экономической эффективности их использования.

Программист не всегда осознает, что он делает, прототип или продукт, и мы получаем прототип, сделанный так добротно, как готовый продукт, или готовый продукт, сделанный как временное решение. Тем не менее, есть энтузиасты, которые любят свою работу, и именно на них держится эта отрасль, противоречивая и полная проблем.

«Большинство хороших программистов делают свою работу не потому, что ожидают оплаты или признания, а потому что получают удовольствие от программирования» // Линус Торвальдс

Разработка программного обеспечения (Software development) — это соединение программирования и кодирования на всех этапах жизненного цикла ПО: проектирования, разработки, тестирования, отладки, поддержки, сопровождения и модификации.

Давайте же стремиться к тому, чтобы наши программы, были простыми и для потребителя и для нас самих, как людей, которые будут их много раз модифицировать и постоянно сталкиваться с теми решениями, которые мы заложили в них при первичной разработке. А если мы ограничены во времени и вынуждены писать неэффективный или малопонятный код, то следует планировать его переработку, рефакторинг и оптимизацию до того, как мы забудем его структуру и у нас выветрятся все идеи по улучшению. Накопление проблем в коде называется "технический долг" и он приводит не только к тому, что программы становятся менее гибкими и понятными, но и к тому, что наши младшие коллеги, подключаясь к проектам, читают и впитывают не лучшие практики и перенимают наш оверинжиниринг. Простота решения сложных

задач, является целью хорошего программиста, скрытие сложности за программными абстракциями — это метод опытного инженера.

«Я всегда мечтал о том, чтобы моим компьютером можно было пользоваться так же легко, как телефоном; моя мечта сбылась: я уже не могу разобраться, как пользоваться моим телефоном» // Бьёрн Страуструп

1.5. Декомпозиция и разделение ответственности

No translation

1.6. Обзор специальности инженер-программист

Вокруг программирования, как и вокруг любой сферы своей деятельности, человек успел построить огромное количество предрассудков и заблуждений. Первейший источник проблем, это терминология, ведь различные парадигмы, языки и экосистемы насаждают свою терминологию, которая не только противоречива между собой, но и нелогична даже внутри отдельного сообщества. Более того, многие программисты самоучки и одиночки, выдумывают самобытную, ни на что не похожую терминологию и концепции, дублирующие друг друга. Оголтелые маркетологи тоже деструктивно влияют на ИТ отрасль вцелом, на формирование мировоззрения и терминологии. Выкручивая очевидные вещи, запутывая и усложняя концепции, они обеспечивают неисчерпаемую лавину проблем, на которой только и держится весь софтверный бизнес. Переманивая пользователей и программистов на свои технологии, гиганты индустрии нередко создают очень заманчивые и правдоподобные концепции, приводящие в итоге к несовместимости, войне стандартов и к явной зависимости от поставщика программной платформы. Группировки, захватывающие внимание людей, десятилетиями паразитирую на их внимании и бюджетах. Ведомые гордыней и тщеславием, некоторые разработчики и сами распространяют сомнительные, а иногда и заведомо тупиковые идеи. Ведь делать программное обеспечение хорошо - совершенно не выгодно для производителя. Ситуация существенно лучше в сфере свободного ПО и открытого кода, но децентрализованные энтузиасты слишком разобщены, чтобы эффективно противодействовать мощной пропаганде

гигантов отрасли.

Как только какая-то технология или экосистема развивается в достаточной степени, чтобы на ней можно было создавать хорошие решения, она обязательно устаревает или производитель прекращает ее поддержку или она становится чересчур сложной. На моей памяти уже сменилось больше пяти таких технологических экосистем.

1.7. Обзор парадигм программирования

Математик рассматривает программу как функцию, которую можно декомпозировать (разделить) на более простые функции так, чтобы программа-функция была их суперпозицией. То есть, грубо говоря, программа является сложной формулой, преобразователем данных, когда на вход подаются условия задачи, а на выходе мы получаем решение. Не всякий программист знаком с этой точкой зрения, хоть она и не идеальная, но полезная для переосмысления своей деятельности. Более распространена противоположная точка зрения, которую легко получить из практики программирования. Заключается она в написании программ исходя из представления пользователя, из рисунков экранов пользовательского интерфейса, и из инструментария, языка, платформы и библиотек. В результате, мы получаем не программу-функцию, а большую систему состояний, в которой происходит комбинаторный взрыв переходов и поведение которой непредсказуемо даже для автора, не то что для пользователя. Но нельзя сразу отметать этот, казалось бы, ужасный подход. В нем есть конструктивное зерно, и заключается оно в том, что не все программы возможно в краткие сроки реализовать в функциональной парадигме как преобразователями данных. Тем более что человеческая деятельность вся состоит из шагов и изменения состояний окружающих нас предметов по принципу пошаговых манипуляций ими, а представить ее в виде функций было бы достаточно неестественным для нашего мышления.

Парадигма задает набор идей и понятий, допущений и ограничений, концепций, принципов, постулатов, приемов и техник программирования для решения задач на вычислительной машине.

В этом разделе мы рассмотрим некоторые из них поверхностно, а

далее в книге будет специальная глава с более подробным обсуждением каждой парадигмы. Есть языки, которые поддерживают одну парадигму, а есть мультипарадигменные языки. Мы обратим внимание на разные языки и отличия реализации парадигм в них.

Для человека естественно представление о любом действии как о наборе шагов или алгоритме, это императивный подход. Шаги эти могут быть или линейными, или принятием решения о переходе к другому шагу плана, вместо того, чтобы исполнять действия последовательно. Принятие решения для машины — это операция сравнения, приводящая к ветвлению алгоритма, дающая варианты (обычно два). Действия можно условно разделить на внутренние и внешние. Во внутренних принимают участие только процессор и память, действие выполняется сразу, без ожидания, и имеет определенный результат, который доступен непосредственно на следующем шаге алгоритма. Внешние действия — это обращения к внешним устройствам ввода- вывода (сеть, диски, другие устройства), и они требуют ожидания реакции от устройства, которая придет за время, обычно неизвестное заранее. Мы отправляем управляющий сигнал периферийным устройствам о том, что им нужно что-то сделать и передаем им необходимые для этого данные. Далее у нас есть опять два варианта: или ожидать результата, и это будет называться блокирующим режимом ввода-вывода, или перейти к следующему шагу алгоритма, не дожидаясь результата, и это будет неблокирующий ввод- вывод. Такое разделение вызвано значительной разницей в длительности внутренних и внешних действий. Большинство внешних действий связаны с физическими операциями над внешней средой. Например, передача данных по беспроводной или проводной сети, запись или чтение с физического носителя, взаимодействие с датчикоом, реле или приводом. Такие операции часто имеют не цифровую, а аналоговую природу, поэтому требуется дополнительное преобразование данных, ожидание переходного процесса, ожидание нужного показания датчика или сигнала от устройства и т.д. Устройства ввода- вывода часто имеют свой контроллер, в котором выполняется отдельный поток операций, а взаимодействие между центральным процессором и устройствами ввода- вывода также требует согласования, что занимает время и может закончиться неудачно.

Парадигма предлагает модель решения задач,

определенный стиль, шаблоны (примеры хороших и плохих решений) применяемых для написания программного кода.

2. Базовые концепты

Нам нужны комментарии для временного предотвращения выполнения или компиляции блока кода, для хранения структурированных аннотаций или метаданных (интерпретируемых специальными инструментами), для добавления TODO или понятных разработчику объяснений.

Комментарий — это последовательность символов в коде, игнорируемая компилятором или интерпретатором.

Комментарии во всех языках семейства **C**, таких как **C++**, **JavaScript**, **Java**, **C#**, **Swift**, **Kotlin**, **Go** и т. д., имеют одинаковый синтаксис.

```
// Single-line comment
```

```
/*
  Multi-line
  comments
*/
```

Не держите в комментариях очевидные вещи, не повторяйте то, что и так понятно из самого кода.

В **bash** (shell-скриптах) и **Python** мы используем знак "номер" (диез или решётку) для комментированиев.

```
# Single-line comment
```

Python использует многострочные строки как многострочных комментариев с выделение тройными кавычками. Но помните, что это строковый литерал, не присвоенный в переменную.

```
"""
  Multi-line
  comments
"""
```

SQL использует два тире, чтобы начать однострочный комментарий до конца строки.

```
select name from PERSON -- comments in sql
```

HTML-комментарии имеют только многострочный синтаксис.

```
<!-- commented block in xml and html -->
```

В ассемблере и множестве диалектов LISP мы используем точку с запятой (или несколько точек с запятой) для различных типов комментариев.

```
; Single-line comment in Assembler and LISP
```

2.1. Значение, идентификатор, переменная и константа, литерал, присвоение

Значение (Value) — величина, записанная в определенное место памяти в определенном формате и представляющая данные, которым может манипулировать программа.

Идентификатор (Identifier) — имена переменных, констант, функций, методов, аргументов, классов, как внутренние, так и импортированные из других модулей и глобальные.

```
const INTERVAL = 500;
let counter = 0;
const MAX_VALUE = 10;
let timer = null;

const event = () => {
  if (counter === MAX_VALUE) {
    console.log('The end');
```

```
    clearInterval(timer);
    return;
  }
  console.dir({ counter, date: new Date() });
  counter++;
};

console.log('Begin');
timer = setInterval(event, INTERVAL);
```

Переменная (Variable) — именованная область памяти (идентификатор), имеющая тип данных, адрес и значение.

Мы можем менять значение переменной в отличие от константы (а для некоторых языков и тип):

```
let cityName = 'Beijing';
```

Константа (Constant) — идентификатор, с которым связано неизменяемое значение и тип

```
const WATCH_TIMEOUT = 5000;
```

```
// Constants

const SALUTATION = 'Ave';

const COLORS = [
  /* 0 */ 'black',
  /* 1 */ 'red',
  /* 2 */ 'green',
  /* 3 */ 'yellow',
  /* 4 */ 'blue',
  /* 5 */ 'magenta',
  /* 6 */ 'cyan',
```

```
  /* 7 */ 'white',
];
```

Литерал (Literal) — запись значения в коде программы.

Например: литералы чисел, логических значений, null и undefined, строк, массивов, объектов, функций. Литералы могут иметь различный синтаксис, от очень простого, для записи чисел, до сложных синтаксических конструкций, для записи объектов.

Присвоение (Assignment) — связывания значения и идентификатора (например переменной).

Операция присвоения во многих языках возвращает присваиваемое значение (имеет поведение выражения).

2.2. Типы данных, скалярные, ссылочные и структурные типы

Тип (Type) - множество значений и операции, которые могут быть произведены над этими значениями.

Например, в **JavaScript** тип **Boolean** предполагает два значения **true** и **false** и логические операции над ними, тип **Null** предполагает одно значение **null** , а тип **Number** множество рациональных чисел с дополнительными ограничениями на минимальное и максимальное значение, а также ограничения на точность и математические операции **+ - * ** / % ++ -- > < >= <= & | ~ ^ << >>**.

Типы данных (Data Types)

```
const values = [5, 'Kiev', true, { size: 10 }, (a) => ++a];

const types = values.map((x) => typeof x);
console.log({ types });
```

Скаляр (Scalar, Primitive, Atomic value) — значение примитивного типа данных.

Скаляр копируется при присвоении и передается в функцию по значению.

Ссылка (Reference) указывает на значение ссылочного типа, т.е. не скалярное значение.

Для **JavaScript** это подтипы **Object**, **Function**, **Array**.

Структурные типы (Composed types) — композитные типы или структуры состоят из нескольких скалярных значений.

Скалярные значения объединяются в одно таким образом, чтоб над этим объединенным значением можно выполнять набор операций. Например: объект, массив, множество, кортеж.

Перечислимый тип (Enumerated type)

Флаг (Flag) — логическое значение, определяющее состояние чего-либо.

Например, признак закрытия соединения, признак завершения поиска по структуре данных и т.д. Например:

```
let flagName = false;
```

Иногда флагами могут называть не логические, а перечислимые типы.

Строка (String) — последовательность символов

В большинстве языков к каждому символу можно обратиться через синтаксис доступа к элементам массива, например, квадратные

скобки.

2.3. Контекст и лексическое окружение

Область видимости (Scope) — часть кода, из которой "виден" идентификатор.

```
const level = 1;

const f = () => {
  const level = 2;
  {
    const level = 3;
    console.log(level); // 3
  }
  console.log(level); // 2
};
```

В современном стандарте **JavaScript** область видимости порождается функцией или любым блоком операторов, имеющим фигурные скобки **{}** или операторами ветвления и циклов, в которых могут быть блоки, но скобки **{}** могут и опускаться. В **Python** scope порождается только функциями. Сравните этот код с предыдущим примером:

```
level = 1

def f():
  level = 2
  if level == 2:
    level = 3
    print(level) // 3
  print(level) // 3

f()
```

Лексический контекст (Lexical environment) — набор

идентификаторов, связанных с определенными значениями в рамках функции или блока кода (в том числе блоков циклов, условий и т.д.).

Лексический контекст или лексическое окружение имеют вложенность, т.е. кроме локальных переменных в блоке, порождающем контекст, есть и вышестоящий блок со своим контекстом. Если идентификатор определен в вышестоящем контексте, то он виден во всех вложенных, если только не происходит перекрытия имен. Перекрытие — это случай, когда во вложенном контексте заново объявлен идентификатор, уже имеющийся во внешнем, тогда значение из внешнего контекста становится недоступно и мы можем доступаться только к внутреннему.

Объектный контекст функции — объект, связанный со служебным идентификатором `this`.

Все функции, кроме стрелочных, могут быть связаны с объектным контекстом. Объект связан с **this**, если функция является методом этого объекта, если функция привязана к нему через **bind** или вызвана через **apply** и **call**.

Глобальный контекст (Global context) — глобальный объект-справочник.

Если идентификатор не находится ни в одном из вложенных лексических контекстов, то будет выполнен его поиск в глобальном контексте (global, window, sandbox).

2.4. Оператор и выражение, блок кода, функция, цикл, условие

Инструкция (Instruction) — один шаг алгоритма вычислений, например инструкция процессора исполняется CPU.

Оператор (Statement) — наименьшая синтаксическая часть языка программирования, исполняемая интерпретатором, средой или компилируемая в машинный код.

Команда (Command) — атомарная задача для командного процессора.

Выражение (Expression) — синтаксическая конструкция языка программирования предназначенная для выполнения вычислений.

Выражение состоит из идентификаторов, значений, операторов и вызова функций. Пример:

```
(len - 1) * f(x, INTERVAL);
```

Блок кода (Code block) — логически связанная группа инструкций или операторов.

Блоки создают область видимости. Блоки могут быть вложенными. Примеры в разных языках: **{}**, **(+ a b)**, **begin end**, в Python блоки выделяются отступами.

Цикл (Loop) — многократное исполнение блока операторов.

```
const MAX_VALUE = 10;

console.log('Begin');
for (let i = 0; i < MAX_VALUE; i++) {
  console.dir({ i, date: new Date() });
}
console.log('The end');
```

Условие (Conditional statements) — синтаксическая конструкция, позволяющая выполнить разные действия или возвращающая разные значения (тернарный оператор) в зависимости от логического выражения (возвращающего true или false).

Рекурсия (Recursion) — задание алгоритма вычисления функции через вызов ее самой (прямой или непрямой) или определение функции, через нее саму.

Косвенная (непрямая) рекурсия — когда функция определена или вызывает себя не напрямую, а через другую или цепочку функций.

Хвостовая рекурсия — частный случай, когда рекурсивный вызов является последней операцией перед возвратом значения, что всегда может быть преобразовано в цикл, даже автоматическим способом. Не хвостовая тоже может быть преобразована в цикл и оптимизирована, но более сложным способом, обычно вручную.

2.5. Процедурная парадигма, вызов, стек и куча

Процедура или подпрограмма (Procedure, Subroutine) — логически связанная группа инструкций или операторов, имеющая имя.

Процедура способствует повторному использованию кода и может быть вызвана из разных частей программы, много раз и с разными аргументами. Процедура не возвращает значений, в отличие от функций, но в некоторых языках (но не в JavaScript) может модифицировать свои аргументы. Во многих языках процедура описывается при помощи синтаксиса функций (например, типа void).

Функция (Function) — абстракция преобразования значений. Функция однозначно отображает одно множество значений в другое множество значений.

Функция может быть задана блоком операторов или выражением. Функция имеет набор аргументов. Функция может быть вызвана по имени или через указатель. Функция способствует повторному использованию кода и может быть вызвана из разных частей программы, много раз и с разными аргументами. В JavaScript функция описывается при помощи function или синтаксиса стрелок (лямбда-функций).

Сигнатура функции (Function signature) включает в себя: имя (идентификатор), количество аргументов и их типы (а иногда и имена аргументов), тип результата.

Метод — функция или процедура, связанная с объектным или классом.

```
{
  a: 10,
  b: 10,
  sum() {
    return this.a + this.b;
  }
}
```

```
const colorer = (s, color) => `\x1b[3${color}m${s}\x1b[0m`;

const colorize = (name) => {
  let res = '';
  const letters = name.split('');
  let color = 0;
  for (const letter of letters) {
    res += colorer(letter, color++);
    if (color > COLORS.length) color = 0;
  }
  return res;
};

const greetings = (name) =>
  name.includes('Augustus')
    ? `${SALUTATION}, ${colorize(name)}!`
```

```
    : `Hello, ${name}!`;
```

Usage

```
const fullName = 'Marcus Aurelius Antoninus Augustus';
console.log(greetings(fullName));

const shortName = 'Marcus Aurelius';
console.log(greetings(shortName));
```

2.6. Функция высшего порядка, чистая функция, побочные эффекты

Объявление функции (Function definition) — в **JavaScript** это способ объявления функции, который виден из любого места в лексическом контексте, в котором объявлена функция, пример:

```
function sum(a, b) {
  return a + b;
}
```

Функциональное выражение (Function expression) — связывание функции с идентификатором при помощи присвоения, при котором значение будет доступно через идентификатор не во всем лексическом контексте, а только после места присвоения. Имеет несколько синтаксических вариантов:

Функциональное выражение с именованной функцией (Named function expression):

```
const max = function max(a, b) {
  return a + b;
};
```

Анонимное функциональное выражение (Anonymous function expression):

```
const max = function (a, b) {
  return a + b;
};
```

Стрелочная или лямбда-функция (Arrow, Lambda function):

```
const max = (a, b) => {
  return a + b;
};
```

Лямбда- выражение, Функция- стрелка с выражением в качестве тела (Lambda expression, Arrow function):

```
const max = (a, b) => a + b;
```

Чистая функция (Pure Function) — детерминированная функция без побочных эффектов.

Чистая функция - функция, вычисляющая результат только на основе аргументов, не имеющая состояния и не обращающаяся к операциям ввода- вывода. Результат такой функции всегда детерминированный и без побочных эффектов (см. побочный эффект).

Замыкание (Closure) — функция, связанная с лексическим окружением в момент своего создания.

Если вернуть функцию **g** из функции **f**, то **g** будет видеть контекст функции **f**, так же, как и свои аргументы. Если **f** возвращает **g**, то говорят, что экземпляр **g** замкнул контекст **f**. Замыкание — это способ, позволяющий связать функцию с контекстом (с данными или переменными контекста). Замыкание позволяет создать эффект, аналогичный состоянию объекта (набору его свойств) в ООП. Свойства связаны с методами через объект, по сути объект в ООП сам является контекстом связывания. Замыкание так же порождает подобный контекст, но на основе функций первого класса и лексического контекста, а не объектного.

При помощи замыкания можно реализовать функциональное

наследование.

Примеры:

```
const add = (a) => (b) => a + b;

const hash =
  (data = {}) =>
  (key, value) => ((data[key] = value), data);
```

Суперпозиция (Superposition) — объединение вызова функций в выражения таким образом, что результат одних функций становится аргументами других функций.

```
const expr2 = add(
  pow(mul(5, 8), 2),
  div(inc(sqrt(20)), log(2, 7))
);
```

Композиция (Composition) — создание новой функции объединением более простых.

```
const compose = (f1, f2) => (x) => f2(f1(x));
```

```
const compose = (...funcs) => (...args) =>
  funcs.reduce((args, fn) => [fn(...args)], args);
```

Частичное применение (Partial application)

```
const partial = (fn, x) => (...args) => fn(x, ...args);
```

Каррирование (Currying)

```
const result = curry((a, b, c) => a + b + c)(1, 2)(3);
```

Побочные эффекты (Side effects)

Функция высшего порядка (Higher-order Function)

1. Если функция передается в другую функцию в качестве аргумента, то это колбек.

2. Если функция возвращается в качестве результата, то это фабрика функций на замыканиях.

3. Если возвращаемая функция имеет ту же семантику, что и получаемая в аргументах, но с дополнительным (расширенным) поведением, то это функция-обертка.

4. Редко бывает, что возвращаемая функция не связана с функцией из аргументов, или связана не прямо, а также имеет другую семантику и функцией-оберткой она не является.

5. Если на выходе класс или функция-конструктор, то это фабрики классов и прототипов соответственно.

Функция-обертка (Wrapper)

Функция, которая оборачивает другую функцию (иногда объект, интерфейс или функциональный объект), добавляя ему дополнительное поведение. Можно обернуть целый API интерфейс и даже асинхронную функцию вместе с колбеками (если известен контракт).

```
const add = (a, b) => a + b;

console.log(`Add nums:    ${add(5, 2)}`);
console.log(`Add float:   ${add(5.1, 2.3)}`);
console.log(`Concatenate: ${add('5', '2')}`);
console.log(`Subtraction: ${add(5, -2)}`);
```

2.7. Замыкания, функции обратного вызова, обертки и события

No translation

2.8. Исключения и обработка ошибок

Отладка (Debug) — процесс обнаружения и устранения ошибок в программном обеспечении.

Отладку реализовывают при помощи вывода сообщений или инструментов: отладчика, профилировщика, декомпилятора, систем мониторинга ресурсов и логирования, систем непрерывной интеграции и тестирования.

«Отладка кода вдвое сложнее, чем его написание. Так что если вы пишете код настолько умно, насколько можете, то вы по определению недостаточно сообразительны, чтобы его отлаживать.» // Брайан Керниган

«Если отладка — процесс удаления ошибок, то программирование должно быть процессом их внесения» // Эдсгер Дейкстра

2.9. Мономорфный код в динамических языках

No translation

3. Состояние приложения, структуры данных и коллекции

No translation

3.1. Подходы к работе с состоянием: stateful and stateless

No translation

3.2. Структуры и записи

```
#include <stdio.h>

struct date {
  int day;
  int month;
  int year;
};

struct person {
  char *name;
  char *city;
  struct date born;
};

int main() {
  struct person p1;
  p1.name = "Marcus";
  p1.city = "Roma";
  p1.born.day = 26;
  p1.born.month = 4;
  p1.born.year = 121;

  printf(
    "Name: %s\nCity: %s\nBorn: %d-%d-%d\n",
    p1.name, p1.city,
    p1.born.year, p1.born.month, p1.born.day
```

```
  );

  return 0;
}
```

Pascal

```
program Example;

type TDate = record
  Year: integer;
  Month: 1..12;
  Day: 1..31;
end;

type TPerson = record
  Name: string[10];
  City: string[10];
  Born: TDate;
end;

var
  P1: TPerson;
  FPerson: File of TPerson;

begin
  P1.Name := 'Marcus';
  P1.City := 'Roma';
  P1.Born.Day := 26;
  P1.Born.Month := 4;
  P1.Born.Year := 121;
  WriteLn('Name: ', P1.Name);
  WriteLn('City: ', P1.City);
  WriteLn(
    'Born: ',
    P1.Born.Year, '-',
    P1.Born.Month, '-',
    P1.Born.Day
  );
```

```
  Assign(FPerson, './record.dat');
  Rewrite(FPerson);
  Write(FPerson, P1);
  Close(FPerson);
end.
```

Rust

```
struct Date {
    year: u32,
    month: u32,
    day: u32,
}

struct Person {
    name: String,
    city: String,
    born: Date,
}

fn main() {
    let p1 = Person {
        name: String::from("Marcus"),
        city: String::from("Roma"),
        born: Date {
            day: 26,
            month: 4,
            year: 121,
        },
    };

    println!(
        "Name: {}\nCity: {}\nBorn: {}-{}-{}\n",
        p1.name, p1.city,
        p1.born.year, p1.born.month, p1.born.day
    );
}
```

TypeScript: Interfaces

```
interface IDate {
  day: number;
  month: number;
  year: number;
}

interface IPerson {
  name: string;
  city: string;
  born: IDate;
}
```

```
const personToString = (person: IPerson): string => {
  const { name, city, born } = person;
  const { year, month, day } = born;
  const fields = [
    `Name: ${name}`,
    `City: ${city}`,
    `Born: ${year}-${month}-${day}`,
  ];
  return fields.join('\n');
};
```

```
const person: IPerson = {
  name: 'Marcus',
  city: 'Roma',
  born: {
    day: 26,
    month: 4,
    year: 121,
  },
};

console.log(personToString(person));
```

TypeScript: Classes

```
class DateStruct {
  day: number;
  month: number;
  year: number;
}

class Person {
  name: string;
  city: string;
  born: DateStruct;
}
```

JavaScript: Classes

```
class DateStruct {
  constructor(year, month, day) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}

class Person {
  constructor(name, city, born) {
    this.name = name;
    this.city = city;
    this.born = born;
  }
}
```

```
const personToString = (person) => {
  const { name, city, born } = person;
  const { year, month, day } = born;
  const fields = [
    `Name: ${name}`,
```

```
    `City: ${city}`,
    `Born: ${year}-${month}-${day}`,
  ];
  return fields.join('\n');
};
```

```
const date = new DateStruct(121, 4, 26);
const person = new Person('Marcus', 'Roma', date);
console.log(personToString(person));
```

JavaScrip: Objects

```
const person = {
  name: 'Marcus',
  city: 'Roma',
  born: {
    day: 26,
    month: 4,
    year: 121,
  },
};

console.log(personToString(person));
```

JavaScrit: strucute serialization

```
const v8 = require('v8');
const fs = require('fs');
```

Take from previous example:

- class DateStruct - class Person

```
const date = new DateStruct(121, 4, 26);
const person = new Person('Marcus', 'Roma', date);
```

```
const v8Data = v8.serialize(person);
const v8File = './file.dat';
fs.writeFile(v8File, v8Data, () => {
  console.log('Saved ' + v8File);
});
```

File: file.dat

```
FF 0D 6F 22 04 6E 61 6D 65 22 06 4D 61 72 63 75
73 22 04 63 69 74 79 22 04 52 6F 6D 61 22 04 62
6F 72 6E 6F 22 03 64 61 79 49 34 22 05 6D 6F 6E
74 68 49 08 22 04 79 65 61 72 49 F2 01 7B 03 7B
03
```

Nested structures

```
#include <stdio.h>
#include <map>
#include <string>
#include <vector>

struct Product {
  std::string name;
  int price;
};

void printProduct(Product item) {
  printf("%s: %d\n", item.name.c_str(), item.price);
}

void printProducts(std::vector<Product> items) {
  for (int i = 0; i < items.size(); i++) {
    printProduct(items[i]);
  }
}

int main() {
```

```
  std::map<std::string, std::vector<Product>> purchase {
    { "Electronics", {
      { "Laptop", 1500 },
      { "Keyboard", 100 },
      { "HDMI cable", 10 },
    } },
    { "Textile", {
      { "Bag", 50 },
    } },
  };

  std::vector electronics = purchase["Electronics"];
  printf("Electronics:\n");
  printProducts(electronics);

  std::vector textile = purchase["Textile"];
  printf("\nTextile:\n");
  printProducts(textile);

  Product bag = textile[0];
  printf("\nSingle element:\n");
  printProduct(bag);

  int price = purchase["Electronics"][2].price;
  printf("\nHDMI cable price is %d\n", price);
}
```

Python

```
purchase = {
  'Electronics': [
    { 'name': 'Laptop', 'price': 1500 },
    { 'name': 'Keyboard', 'price': 100 },
    { 'name': 'HDMI cable', 'price': 10 },
  ],
  'Textile': [
    { 'name': 'Bag', 'price': 50 },
  ],
}
```

```
electronics = purchase['Electronics']
print({ 'electronics': electronics })

textile = purchase['Textile']
print({ 'textile': textile })

bag = textile[0]
print({ 'bag': bag })

price = purchase['Electronics'][2]['price']
print({ 'price': price })
```

JavaScript

```
const purchase = {
  Electronics: [
    { name: 'Laptop', price: 1500 },
    { name: 'Keyboard', price: 100 },
    { name: 'HDMI cable', price: 10 },
  ],
  Textile: [{ name: 'Bag', price: 50 }],
};

const electronics = purchase.Electronics;
console.log(electronics);

const textile = purchase['Textile'];
console.log(textile);

const bag = textile[0];
console.log(bag);

const price = purchase['Electronics'][2].price;
console.log(price);

const json = JSON.stringify(purchase);
console.log(json);
const obj = JSON.parse(json);
```

```
console.log(obj);
```

3.3. Массив, список, множество, кортеж

No translation

3.4. Словарь, хэш-таблица и ассоциативный массив

No translation

3.5. Стек, очередь, дэк

No translation

3.6. Деревья и графы

No translation

3.7. Проекции и отображения наборов данных

No translation

3.8. Оценка вычислительной сложности

No translation

4. Расширенные концепции

No translation

4.1. Что такое технологический стек

No translation

4.2. Среда разработки и отладка кода

No translation

4.3. Итерирование: рекурсия, итераторы и генераторы

No translation

4.4. Структура приложения: файлы, модули, компоненты

No translation

4.5. Объект, прототип и класс

No translation

4.6. Частичное применение и каррирование, композиция функций

No translation

4.7. Чейнинг для методов и функций

No translation

4.8. Примеси (mixins)

No translation

4.9. Зависимости и библиотеки

No translation