Вопросы-Обновленные

Вопрос 1

Чистые и грязные функции. Изменяемое состояние. Тотальность функций.
Ответ:
Функцией называется отображение множества входных значений в множество результатов, где выполняется главное условие: одному элементу входного множества соответствует один элемент из выходного множества - чисто определение из математики.

В программировании функции можно разделить на два вида:

- Чистые
- Грязные

Чистая функция - это та функция, у которой отсутствуют side-effect-ы - это действия, которые изменяют то, что определено извне данной функции. Примером типичного side-effect-а может быть изменение глобальной переменной. К преимуществам чистых функций можно отнести:

- Простоту тестирования - side-effect-ы могут вызывать неявные изменения состояния программы, которые тяжело отлаживать
- Безопасность использования - чистые функции можно вызывать бесконечное количество раз, не беспокоясь за то, что эти вызовы повлияют на состояние программы

Тем не менее без побочных эффектов не обойтись - мы вынуждены делать наши функции грязными, потому что программам нужно считывать и записывать данные сторонних источников (например, файлов), выводить данные - это все side-effect-ы. Примером такой функции может послужить функция append из Python, которая принимает массив и значение, а потом кладет это значение в переданный массив.

Одним из ключевых отличий многих функциональных языков программирования от остальных является использование неизменяемых данных. Например, в Haskell даже нет оператора присваивания. Но так ли плохо само по себе изменяемое состояние? Вопрос достаточно сложный.

У изменяемого состояния есть явные недостатки:

- Подавляющее большинство многопоточных программ завязано на изменении разделяемых данных. Для обеспечения целостности этих данных и корректности

работы программы нужно добавлять дополнительный over-head в виде примитивов синхронизации, неблокирующих структур данных и прочее
- Изменяемые данные усложняют корректное кеширование. Например, в интернет-магазине есть какая-то структура данных, связанная с корзиной, и переменная, хранящая суммарную стоимость товаров. Для эффективной работы стоимость и корзина кешируются. Немала вероятность ситуации, в которой корзина поменяет состав, а закешированная цена останется прежней
  Как итог, при изменяемом состоянии есть сложности с обеспечением целостности состояния программы, сохранения выполнения ее инвариантов.

Тем не менее полностью отказаться от модели изменяемого состояния невозможно:

- Многие техники из функциональных языков, не применимы в традиционных языках (Java, C++)
- Для некоторых и структур данных мало известно или вообще не существует аналогов по эффективности без присваивания (например, для хэш-таблицы)
- Многие предметные области по своей сути содержат изменяемые объекты (например, банковские счета)

Тотальность функции - это свойство функции, которое означает, что функция определена для всего множества своих входных значений. Например, если мы реализуем функцию деления, то нужно учесть случай деления на 0, чтобы программа корректно обработала данную ситуацию.

Вопрос 2

Рекурсия. Хвостовая рекурсия. Структурная рекурсия.
Ответ:
Рекурсия - это, как нам известно, вызов функции самой себя. Этот инструмент активно используется в функциональных языках программирования. Например, в функциональных языках нет операторов для работы с циклами (по крайней мере, я не видел), но, тем не менее, циклическая обработка чего-нибудь вполне элегантно реализуется как раз через рекурсию: например, если нужно пройтись по списку, то с каждым вызовом функции мы отцепляем первый элемент и передаем в следующий вызов хвост. По такому принципу устроены основные операции по работе со структурами в функциональных языках:

- filter
- reduce
- map

Так же нам известно, что при работе рекурсии активно задействуется стек, поэтому тут же мы сталкиваемся с потенциальной проблемой: если глубина рекурсии (т.е. количество рекурсивных вызовов) большая, то у нас закончится стек и мы отлетим с NoMemoryError. Но для предотвращения такой проблемы придумали оптимизацию рекурсии - хвостовую рекурсию. Рекурсия будет считаться хвостовой, если рекурсивный вызов в функции является последним действием, т.е. ВАЖНО ПОДМЕТИТЬ требуется только вызов без дополнительных действий из разряда сложения и пр. В таком случае пространство на стеке переиспользуется и таким образом стек не растет.

Структурной, или вполне обоснованной, рекурсия будет считаться в том случае, если с каждым вызовом рекурсия приближается к условию останова. Например, функция проходится по списку и ее терминирующим условием является пустой список: если с каждым вызовом функция отщепляет один элемент от списка и в следующий рекурсивный вызов передает ее хвост, то мы можем быть уверены, что рано или поздно функция придет к условию останова, т.е. рекурсия считается структурной, или вполне обосонованной.

Вопрос 3

Мемоизация.
Ответ:
Зачастую алгоритмам приходится пересчитывать по несколько раз одни и те же данные. В таком случае хорошей оптимизацией будет сохранять результаты этих вычислений в кэше и затем по входным параметрам быстро подставлять ответ, если он уже был посчитан. Главное, надо быть уверенным, что если мы одни и те же значения подадим на вход функции, то результат будет одним и тем же независимо от количества попыток. Популярным примером рекурсивного алгоритма, который ускоряется за счет мемоизации - это подсчет чисел Фибоначчи: мемоизацию можно самому реализовать (например, при помощи словаря) или воспользоваться стандартными средствами языка (например, в Python такое есть).

Вопрос 4

Чисто функциональные структуры данных.
Ответ:
Чисто функциональные структуры данных представляют из себя особый класс структур данных, используемый в функциональных языках программирования. От структур данных императивных языков их отличает:

- Неизменяемость - при изменении структуры данных создается копия данной структуры данных, в которой производятся изменения (старая версия остается нетронутой). Это максимально деревянный подход к реализации иммутабельности

этих структур, но достаточно дорогой по памяти. Для повышения эффективности использования памяти используются специальные алгоритмы, которые не полностью копируют исходную структуру данных. В зависимости от типа структуры данных эти алгоритмы могут быть разными по сложности.
- Персистентность, или версирование - при изменении структуры данных, как мы сказали, создаются копии, т.е. это предоставляет нам возможность спокойно откатываться к предыдущим версиям
- Рекурсия - как мы знаем, функциональные языки подвешены на рекурсии, так что логично, что функциональные структуры данных реализованы при помощи рекурсии, что потом позволяет также рекурсивно обрабатывать данные

Яркими примерами таких структур являются:

- Связный список
- Дерево

Преимущества:

- Неизменяемость
- Версирование
- Безопасность многопоточности
- Упрощение GC
  Недостатки:
- Over-Head по памяти
- Мало источников, как следствие, сложность поддержки и реализации

Вопрос 5

Ссылочная прозрачность. Сопоставление с образцом.
Ответ:
Ссылочная прозрачность - это одно из ключевых концепций/свойств функциональных языков программирования, которое нечасто можно встретить в императивных языках. Ее ключевая идея заключается в том, что если мы видим две конструкции, которые выглядят одинаково, то они и есть одинаковые, а, значит, они взаимозаменяемы без изменения поведения программы. Где же это может пригодиться:

- Оптимизация компилятором:
  - Если у нас есть обычная функция сложения и ее вызов в коде, то мы можем просто заменить вызов сразу результатом. Выглядят одинаково и результат тот же.

    - Зачастую компиляторы выполняют разворачивание циклов: вместо шага +1 мы 10 раз повторим тело цикла и шаг изменим на +10
- Сопоставление с образцом - про это поподробнее

Сопоставление с образцом - это часто используемый инструмент в функциональных языках программирования, благодаря которому можно проводить проверку структур данных и извлекать из них значения, также таким образом реализуется поведение функции под конкретные значения входных параметров. Этот механизм связан с ссылочной прозрачностью, т.к. без нее одинаковые структуры просто бы не сравнивались должным образом.

Вопрос 6

Алгебраический (индуктивный) тип данных. Сравнение с системой типов языка Си. Сопоставление с образцом.
Ответ:
Алгебраический, или индуктивный, тип данных - это составной тип данных, который определяется как сумма других типов. Его ключевой особенностью является безопасность типов - компилятор может проверить все возможные варианты корректных значений, которые применяются к данному типу, на этапе компиляции.

В языке С нет поддержки алгебраических типов данных, вместо этого разработчики используют структуры данных и объединения для создания более сложных и составных данных. НО с большой буквы: при создании таких типов на С создается так называемое декартово произведение типов, в множестве которого есть как корректные комбинации использования, так есть и некорректные (компилятор уже тут не подскажет). Примером такой ситуации можно рассмотреть составной тип данных геометрических фигур, в который входит переменная-enum, указывающая на тип фигуры, и объединение структур, отвечающих за данные, которые каждая фигура может хранить: например, у треугольника это три точки, у круга это радиус с центром и т.д. Неприятная ситуация будет проявляться тогда, когда мы укажем тип Круг, а захотим получить значения радиуса и центра - это пример некорректного использования нашего составного типа. Чтобы таких вещей избежать, нужно делать дополнительный программный over-head, в котором будет учитываться это (например, что-то типа фильтра), но это долго.

Сопоставление с образцом - это один из ключевых инструментов функциональных языков, благодаря которому удобно проверять структуры данных и доставать из них значения. К тому же благодаря сопоставлению с образцом можно таким же образом взаимодействовать с индуктивным типом.

Вопрос 7

Функции высших порядков. Замыкания (Closure). Работа с изменяемым состоянием и областью видимости переменных в функциональных и нефункциональных языках.
Ответ:
Функции высших порядков - это функции, которые могут:

- принимать в качестве аргументов функции
- выдавать в качестве результата функции
  Главное, чтобы хотя бы одно из этого удовлетворялось.
  Примерами использования могут послужить:
- параметризация обобщенных функций (например, сортировка)
- декораторы (модификация стандартного поведения; может использоваться в Python - например, использование логирования)

В настоящем мире чистые функции не так часто встречаются: нередко применяется такая вещь, как замыкание. Замыканием называют функцию, в теле которой имеется ссылка на переменную, объявленную вне тела данной функции где-то в окружающем коде и не являющуюся ее параметром. Несмотря на то, что мы противоречим определению чистой функции, существуют т.н. чистые замыкания, в которых вызов функции не влияет на внешнее состояние программы (например, функция использует enum, который не меняется - дни недели). Можно даже заикнуться, что при помощи замыканий можно делать что-то в роде ООП, т.к. у нас появляется функция, выступающая в роли объекта и ее состояние, в роли которого выступают внешние переменные.

Что касается изменяемого состояния в языках:

- В функциональных языках программирования акцент идет на иммутабельности переменных. Причем, в некоторых языках для обновления значения нужно создавать новую переменную (т.к. не имеют оператора присваивания), а современные языки (например, Elixir) имеют оператор присваивания, но под капотом переменная не обновляется, а она сохраняет новое значение в другом участке памяти
- В нефункциональных языках идея неизменяемого состояния не так популярна по ряду причин, поэтому активно там применяется изменяемое состояние

Что касается области видимости переменных, то в функциональных языках в этом плане строже: используются локальные переменные (глобальные переменные осуждаются, если их можно использовать) и константы.

Вопрос 8

Свёртки. Отображения. Фильтрация. Map-reduce. Бесточечный стиль, комбинаторы.
Ответ:
Свертка, отображение и фильтрация - это базовые операции над структурами данных,

благодаря которым мы можем много чего реализовать. И прежде, чем начать говорить отдельно про каждую из них, стоит обратить внимание, что на этой троице пишется значительная часть функционального кода.

Свертка - это операция по приведению структуры данных к одному объекту. В функцию свертки в качестве аргументов передаются функция, начальное значение объекта и сама структура данных. Также существуют свертки левые и правые, которые определяют порядок прохода по структуре, в языках может отличаться производительность левых и правых сверток - это лишь технические детали.

Отображение - это операция по изменению элементов коллекции, применяя к каждому переданную функцию отображения

Фильтрация - это операция по отбору элементов, удовлетворяющих критерию или критериям. Обычно фильтрация осуществляется в виде функций высших порядков, принимающих в качестве аргументов структуру данных и предикат, по которому производится фильтрация, а в качестве результата выдается отфильтрованная структура данных.

Map-Reduce - модель распределенных вычислений, придуманная компанией Google для обработки больших объемов данных. Идея этой модели достаточно проста: большой объем данных делится на части (например, на столько, сколько доступно устройств), затем из этих частей отбирается нужная информация, которая в итоге сворачивается, в конечном итоге все свернутые части повторно сворачиваются в один объект.
У данной модели есть несколько проблем:

- Случай возникновения ошибки: если у нас произойдет ошибка или откажет узел, то будет проблемно указать оставшимся узлам нашей распределенной сети, что нужно отменить вычисления, также тяжело судить, является ли возникшая ошибка локальной, которая никак не повлияет на остальные узлы, или глобальной, которая бракует работу остальных узлов.
- Консистентность данных: мы работаем с большим датасетом, которым, более вероятно, пользуется немало пользователей - эти пользователи постоянно меняют эти данные, так что достаточно тяжело поддерживать консистентность данных.
- Отказ узлов: в случае отказа одного из узлов следует наиболее эффективным образом перекоммутировать узлы сети, чтобы потери по времени были минимальными.

Бесточечный стиль - это подход, при котором функции определяются без явного указания аргументов. При этом широко используются такие инструменты:

- Функции высшего порядка

- Композиция - функции связываются в цепочку: результаты одной переходят к другой

Комбинатор - это функция, которая принимает другие функции и возвращает новую функцию.

Вопрос 9

Ленивые структуры данных, итераторы, генераторы. Побочные эффекты.
Ответ:
Ленивые вычисления, или ленивые структуры данных - это концепция, при которой вычисления откладываются до тех пор, пока данные не потребуются. Это позволяет создавать бесконечные структуры данных за счет того, что память не расходуется на всю эту структуру. Примером ленивой структуры может послужить ленивый список.

Итераторы и генераторы - это механизмы для работы с последовательностями данных:

- Генераторы - создают последовательности "на лету" с использованием механизма ленивых вычислений (например, оператор yield в Python).
- Итераторы - предоставляют интерфейс для последовательного доступа к структуре данных, не беспокоясь об устройстве

Побочные эффекты, или side-effect-ы, как из привыкли слышать - это изменение состояния программы извне вызванной функции. В функциональных языках дается предпочтение чистым функциям, в которых нет побочных эффектов, потому что это небезопасно (если много раз вызывать функцию, то может что-то сломаться) и дебажить побочные эффекты неприятно и больно.

Вопрос 10

Динамическая верификация. Модульное и интеграционное тестирование. REPL. Doctest.
Ответ:
Верификация - это процесс проверки реализованной системы на соответствие спецификации (т.е. проверка на то, проходит ли система критериям, заявленным в документах). Верификация подразделяется на два вида:

- Статическая
- Динамическая
  Мы поговорим сейчас про динамическую верификацию. Ее основная идея заключается в том, что проверка на соответствие производится через эксперимент.

Преимуществом динамического подхода является его быстрота и простота реализации и в какой-то степени распространенность (по крайней мере, на мой взгляд динамическим тестированием почаще пользуются, поэтому больше по этому поводу информации). Но

этот подход и не лишен недостатка: стоит иметь в виду, что динамическое тестирование доказывает наличие ошибки, а не ее отсутствие (например, мы можем провести один тестовый сценарий, в котором исследуемая ошибка не выскочит, а в другом не затронутом варианте она, оказывается, появляется, а мы об этом не догадываемся), а также наша система должна себя вести более-менее детерминированно, чтобы ее можно было протестировать.

Динамическая верификация включает в себя модульное и интеграционное тестирования.

Модульное, или unit, тестирование - это тестирование, предназначенное на тестирование самостоятельных отдельных компонент. Эти тесты наиболее часто пишут, потому что они проще всего.

Интеграционное тестирование расширяет масштаб предыдущего вида тестирования и затрагивает не один, а сразу несколько модулей. В таком тестировании проверяет корректность взаимодействия компонент между собой. В таких тестах применяется подход мокирования (т.е. создания заглушек), чтобы изолировать взаимодействие модулей и рассмотреть конкретно их.

REPL, или READ-EVAL-PRINT-LOOP, относится к ручному тестированию. Его идея заключается в том, что предоставляется интерактивная среда, которая считывает входные данные/команды пользователя, обрабатывает их и выдает результат, так делается по циклу. Развитыми примерами REPL могут послужить:

- в Erlang с его моделью большого числа взаимодействующих процессов мы можем в виртуальной машине запустить процесс и на лету патчить его
- в Smalltalk при обращении к несуществующему методу система не упадет, а предложит написать реализацию отсутствующего кода.

Doctest - это вид автоматического тестирования, который заключается в том, что сами тесты прописываются в комментариях-документации исходного кода. Такая вещь есть в Python.

Вопрос 11

Динамическая верификация. Golden tests. Fuzzy/Monkey testing. Property-Based Testing.
Ответ:
Верификация - это процесс проверки реализованной системы на соответствие спецификации (т.е. проверка на то, проходит ли система критериям, заявленным в документах). Верификация подразделяется на два вида:

- Статическая

- Динамическая
  Мы поговорим сейчас про динамическую верификацию. Ее основная идея заключается в том, что проверка на соответствие производится через эксперимент.

Преимуществом динамического подхода является его быстрота и простота реализации и в какой-то степени распространенность (по крайней мере, на мой взгляд динамическим тестированием почаще пользуются, поэтому больше по этому поводу информации). Но этот подход и не лишен недостатка: стоит иметь в виду, что динамическое тестирование доказывает наличие ошибки, а не ее отсутствие (например, мы можем провести один тестовый сценарий, в котором исследуемая ошибка не выскочит, а в другом не затронутом варианте она, оказывается, появляется, а мы об этом не догадываемся), а также наша система должна себя вести более-менее детерминированно, чтобы ее можно было протестировать.

Рассмотрим виды автоматического тестирования.

Golden-тесты крайне полезны в тех случаях, когда у нашего приложения сложный вывод данных (например, генерация или редактирование изображения или большого текста). Идея наиболее проста: задаем "золотой стандарт" для вывода нашей программы для определенных данных и затем сравниваем вывод программы с этим стандартом. В случае, если логика программы поменялась, не беда - просто пересоздаем стандарт. Лайфхак: если нет возможности использовать специальную библиотеку для golden-тестов, то можно просто проверять корректность при помощи git: храним файл стандарта -> прогоняем тест, после которого файл перезапишется -> смотрим, изменился ли файл при помощи git status.

Зачастую для сложных систем возникает проблема с подбором тестовых данных: разброс данных на столько широкий, что сложно покрыть (например, тестирование компилятора языка). Fuzzy-тестирование заключается в том, что мы подаем рандомные данные, которые либо чисто рандомные данные, либо данные, направленные вызвать ошибку, и мы в итоге смотрим, устояла ли система (такой подход может применяться при пентестах; например, тестирование распределенной сети).

Monkey-тестирование предназначено больше для распределенных систем, а именно: мы будто запускаем по нашей системе обезьянку, которая последовательно отключает/ломает узлы нашей системы или создает помехи при взаимодействии частей (например, потеря пакетов).

Property-Based тестирование чем-то похоже на fuzzy-тестирование, но отличается целью: она заключается в том, что мы собираемся проверить какое-либо свойство, а не уронить нашу систему. Например, мы хотим в нашей программе доказать, что сложение двух элементов при перестановке элементом результат остается прежним. Мы создаем

генератор, который подает входные данные и мы по этим данным проверяем наше свойство. Хорошая библиотека property-based тестирования при возникновении ошибки на большом наборе данных позволяет сделать shrinking - ужать данные, чтобы лучше обнаружить ошибку. Одной из проблем данного вида тестирования является то, что пройденные тесты не гарантируют нам корректность (например, если мы проверяем симметричность тестирования, где функция суммирования дает 0 - тест пройдет, а смысла нет), т.е. надо быть уверенными, что тестируемая функция работает корректно.

Вопрос 12

Статическая верификация. Виды систем типов. Понятие "объекта первого класса".
Ответ:
Верификация - это процесс проверки реализованной системы на соответствие спецификации (т.е. проверка на то, проходит ли система критериям, заявленным в документах). Верификация подразделяется на два вида:

- Статическая
- Динамическая
  Мы поговорим сейчас про статическую верификацию. Особенность данного подхода заключается в том, что система может даже не существовать, но за счет статического анализа документации можем уже как-то верифицировать нашу будущую систему.

Преимуществом статической верификации считается то, что она покрывает много случаев. Это осуществляется за счет того, что статическая верификация оперирует свойствами, а не конкретными примерами (примерами оперирует динамическая верификация). Тем не менее, статическая верификация порядком сложнее динамической и обладает меньшим числом ресурсов по данной теме в виду меньшей распространенности.

Существует немало инструментов для выполнения статической верификации. Начиная просто от синтаксиса и семантики языка + линтера, заканчивая Model-Driven Engineering, ярким примером которого является технология xtUML: эта технология потрясла разработчиков в свое время (как нейросети нас сегодня) тем, что при помощи UML можно генерировать код программы (за счет самого UML верифицировалось, что предусмотрены все сценарии использования системы).

Система типов - один из распространенных видов статической верификации. По видам системы типов различают на:

- Статические и динамические:
  - Статическая - прописываем типы до компиляции

    - Динамический - тип определяется в рантайме
- Строгие и слабые:
    - Строгая - запрет на совместное использование разных типов и неявные преобразования сведены к минимуму
    - Слабая - можно совместно использовать разные типы и много неявных преобразований

Языки программирования предоставляют нам объекты (грубо говоря, возможности языка), которые по-разному работают и предназначены для различных целей.
Понятие объекта первого класса было введено достаточно давно. Оно характеризуется как базовые возможности языка, которые можно широко использовать.
Об объекте первого класса:

- Объекты первого класса могут быть параметрами функций
- Объекты первого класса могут быть возвращены как результат функций
- Объекты первого класса могут быть использованы в правой части выражения присваивания переменной
- * Объекты первого класса могут быть проверены на эквивалентность (Не всегда можно определить, как сравнивать объекты, например, функции по исходному коду разные, а выдают одинаковые результаты - эквивалентны или нет?)

Примеры таких объектов первого класса:

- значения и ссылки
- указатели на функции

Вопрос 13

Статическая верификация. Механика автоматического вывода типов.
Ответ:
Верификация - это процесс проверки реализованной системы на соответствие спецификации (т.е. проверка на то, проходит ли система критериям, заявленным в документах). Верификация подразделяется на два вида:

- Статическая
- Динамическая
  Мы поговорим сейчас про статическую верификацию. Особенность данного подхода заключается в том, что система может даже не существовать, но за счет статического анализа документации можем уже как-то верифицировать нашу будущую систему.

Преимуществом статической верификации считается то, что она покрывает много случаев. Это осуществляется за счет того, что статическая верификация оперирует свойствами, а не конкретными примерами (примерами оперирует динамическая верификация). Тем не менее, статическая верификация порядком сложнее динамической и обладает меньшим числом ресурсов по данной теме в виду меньшей распространенности.

Механика автоматического вывода типов относится к неявной статической типизации, которая проявляется в том, что мы в исходном коде статического языка не указываем тип переменной, а компилятор сам подбирает тип. В принципе, удобная вещь, только нужно знать, как компилятор выводит эти типы. Активно такой механизм используется в Golang

Вопрос 14

Полиморфизм. Универсальный (параметрический и через наследование). и специальный (ограниченный, через перегрузку, через приведение типов).
Ответ:
Полиморфизм - это концепция современных языков программирования, которую везде объясняют по-разному, но, проще говоря, ее основная идея заключается в том, мы можем обрабатывать разные типы данных с помощью одного интерфейса.

Полиморфизм можно разделить на:

- Универсальный - пишем универсальный код для всех типов
- Специальный - пишем отдельный код под определенный тип

Поговорим про универсальный полиморфизм.

Параметрический полиморфизм называется параметрическим не спроста: мы пишем основной/гибридный код, который в качестве аргумента принимает тип данных (в самом коде этот тип обозначается что-то типа переменной), затем для дальнейшего использования данного кода мы передаем нужный тип данных и на место той переменной ставится тип данных. Такой полиморфизм по-разному может работать ниже уровнем:
- При компиляции создаются физически разный код под нужный тип данных
- Динамически в рантайме

Что касается наследования. Один интерфейс - разный код - ООП-шная история. Если разные классы реализуют единый интерфейс, то мы можем спокойно одинаково с ними взаимодействовать, ведь у нас определен единый интерфейс (что касается того, как этот интерфейс у классов реализован - это уже другая история). Помимо классических языков (типа Python, Java), в функциональных языках существует похожая история: например, к CommonLisp определяется обобщенная функция, от которой создаются функции-

экземпляры. Это относится к универсальному полиморфизму по той причине, что в рантайме производятся сложные операции по вызову конкретной реализации.

Вопрос 15

Полиморфизм. Специальный (ограниченный, через перегрузку, через приведение типов).
Ответ:
Полиморфизм - это концепция современных языков программирования, которую везде объясняют по-разному, но, проще говоря, ее основная идея заключается в том, мы можем обрабатывать разные типы данных с помощью одного интерфейса.

Полиморфизм можно разделить на:

- Универсальный - пишем универсальный код для всех типов
- Специальный - пишем отдельный код под определенный тип

Поговорим про специальный полиморфизм.

Перегрузка имен функций (overloading) - пишем несколько функций с одинаковым именем, но разными типами параметров и возвращаемых значений. Компилятор поймет, какую реализацию вызвать.

Возникают нередко ситуации, когда нам в одном выражении нужно использовать значения/переменные разных типов данных. Зачастую просто замучаешься приводить типы данных, чтобы все это работало. Решением данной проблемы является полиморфизм через приведение типов, более просто - компилятор за нас делает эти приведения.

Ограниченный полиморфизм - непонятно, почему это не относится к подтипу универсального полиморфизма, а конкретно - параметрического. Тем не менее идея крайне похожа на параметрический полиморфизм - так же в качестве переменной передается тип данных, но за одним лишь исключением, что теперь тип данных ограничивается - с этим случаем связано такое понятие, как Type Class. Проще говоря, передаваемый тип должен удовлетворять определенным ограничениям (например, реализовывать определенный интерфейс). Такая вещь встречается в дженериках Java: можем ограничить как сверху, так и снизу интерфейс/класс.

Вопрос 16

Полиморфизм. Множественный полиморфизм.
Ответ:
Полиморфизм - это концепция современных языков программирования, которую везде

объясняют по-разному, но, проще говоря, ее основная идея заключается в том, мы можем обрабатывать разные типы данных с помощью одного интерфейса.

Что касается множественной диспетчеризации (Multiple Dispatch), то нужно для начала ввести для понимания два понятия:

- Абстрактная (или еще называются ее генеральной) функция - это функция, которая принимает аргументы, но не определяет их типов
- Метод (или поведение) - это не тот метод, который встречается в ООП, это конкретная реализация абстрактной функции под конкретные типы данных, т.е. определяется поведение функции
  Соответственно, мы определяем абстрактную функцию и поведения для нее. Теперь при вызове данной функции с конкретными аргументами будет диспетчеризоваться вызов к функции по всем ее аргументам.

На самом деле я не вижу сильной разницы между полиморфизмом с перегрузкой, т.к. идея та же. Единственное, чем аргументируют разницу - это то, что в перегрузке идет диспетчеризация в большей мере относительно объектов и классов, от которых они реализуются, а во множественной диспетчеризации это происходит только относительно аргументов фукнции.

Вопрос 17

Лямбда исчисление. Виды термов. Связанные и свободные переменные. Альфа, бета и эта преобразования. Порядок редукции. Булева алгебра и нумералы Чёрча.
Ответ:
Параллельно с развитием идеи машины Тьюринга, где в явном виде идет разделение между памятью и управлением, с небольшим запозданием началось развитие иного представления вычислительного процесса - лямбда-исчислений, подхода с математическим уклоном с описанием через символьную форму (здесь программа как бы является и данными, и инструкциями; а за состояние "процессора" отвечают последовательности символов, которые в правильном виде изменяются для решения задач). В настоящее время в современных языках программирования нередко можно встретить лямбда-исчисления.

Я лямбда-исчислениях всего 4 вида лямбда-термов:

- Переменные - как и в привычных нам языках.
- Константы - похожи на переменные за тем исключением, что в переменные подставляются значения, а константы принимаются такими, какие они уже есть
- Комбинации (или применение функции, или аппликация) - это когда два терма стоят подряд, где первым термом обозначается функция, а вторым - аргумент, который

подставляется внутрь этой функции
- Абстракции - предназначены для объявления функций, пишется лямбда-оператор со связанной переменной + точка + тело функции. Чтобы передать несколько аргументов, делаются вложенные функции с несколькими лямбда-операторами.

В лямбда-исчислениях переменные бывают связанные и свободные. Связанной переменная будет считаться, если она находится под действием лямбда-оператора (проще говоря, в переменная записана в функции в качестве аргумента). Если переменная не записана после лямбда-оператора, но используется в теле функции, то она является свободной.

Conversions of Lamda-terms

- **Alpha conversion**: $\lambda x.\, s \xrightarrow{\alpha} \lambda y.\, s[y/x]$ provided $y \notin FV(s)$.
  - E.g. $\lambda u.\, u\ v \xrightarrow{\alpha} \lambda w.\, w\ v$.
- **Beta conversion**: $(\lambda x.\, s)\ t \xrightarrow{\beta} s[t/x]$.
- **Eta conversion**: $\lambda x.\, t\ x \xrightarrow{\eta} t$ provided $x \notin FV(t)$.
  - E.g. $\lambda u.\, v\ u \xrightarrow{\eta} v$.

Помимо термов существуют преобразования, которые используются для переписывания лямбда-выражений:

- альфа-преобразование - это когда мы хотим переименовать переменную (например, для разрешения конфликтов по именам; главное, чтобы сама замена не конфликтовала)
- бета-преобразование - применение переменной к абстракции. То есть мы сначала в скобочках определили функцию (написали абстракцию), а затем за скобками написали переменную - эту конструкцию можно заменить на просто тело функции с замененной связанной переменной
- эта-преобразование - вырожденный случай из бета-преобразования, в котором происходит применение аргумента к абстракции с одинаковым именем связанной переменной

Boolean and If Statement

$$true \equiv \lambda x\ y.\ x$$

$$false \equiv \lambda x\ y.\ y$$

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

How it works:

- $if\ true\ then\ E_1\ else\ E_2$
- $= true\ E_1\ E_2$
- $= (\lambda x\ y.\ x)\ E_1\ E_2$
- $= E_1$

And basic logic functions:

$$not\ p \equiv if\ p\ then\ false\ else\ true$$

$$p\ and\ q \equiv if\ p\ then\ q\ else\ false$$

$$p\ or\ q \equiv if\ p\ then\ true\ else\ q$$

Для выполнения бета-преобразования (бета-редукции) существуют два порядка:

- Нормальный - идем от широко к мелкому (т.е. идем извне)
- Аппликативный - идем извне (т.е. сначала применяем внутреннее)

Pairs

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

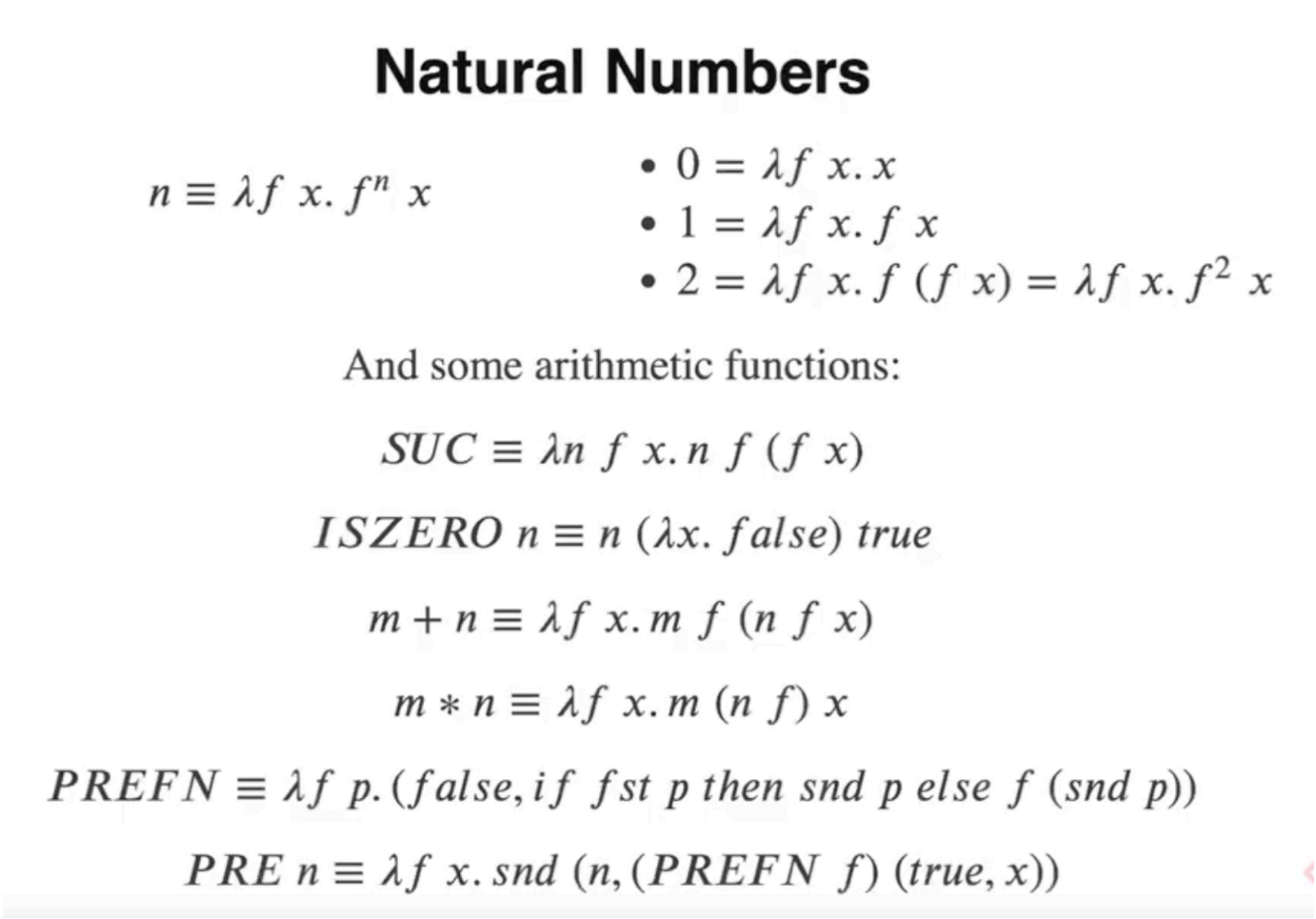
$$fst\ p \equiv p\ true$$

$$snd\ p \equiv p\ false$$

How it works:

- $fst\ (p, q) = (p, q)\ true$
- $= (\lambda f.\ f\ p\ q)\ true$
- $= true\ p\ q$
- $= p$

Для булевой алгебры определяем две функции, принимающие два аргумента и возвращающие один - true - первый, false - второй. Например, в наших примерах эти функции мы применяли для реализации структуры данных pair.



Для представления натуральных чисел используются нумералы Черча.

Вопрос 18

Структурное программирование, преимущества перед языками с Go To. Аргументация Дейкстры против Go To. Варианты практичного использования Go To сегодня. Механизм Defer языка Golang.
Ответ:
Структурное программирование — это методологический подход к написанию программного кода, который основывается на использовании структурированных блоков и избегании безусловных переходов (операторов `Go To`). Этот подход был разработан для улучшения читаемости, надежности и поддерживаемости кода.

Преимущества:

- Читаемость кода: Структурное программирование упрощает понимание кода, так как логически связанные операторы располагаются близко друг к другу, что делает код более последовательным и понятным
- Упрощение отладки: Код, написанный с использованием структурного подхода, легче тестировать и отлаживать благодаря четкой структуре и уменьшению сложности
- Улучшенное обслуживание: Легкость изменения и обновления кода делает его более удобным для поддержки в долгосрочной перспективе

Блок - это последовательность инструкций, у которой есть как точка входа, так и выхода. Данные блоки могут комбинироваться разными путями:

- Последовательно - блоки идут по цепочке друг за другом
- Conditional - с использованием условного оператора, при помощи которого в зависимости от условия мы попадаем в какой-то из нескольких блоков
- Циклический
- Вызовы (например, функция)

Дейстра аргументировал тем, что код с Go To имеет трудности в отладке, так как трудно предсказать последовательность выполнения, поэтому при отладке приходится тащить за собой трассу того, что уже произошло ранее.

Go To может быть полезен в:

- Выходе из вложенных циклов
- Освобождении ресурсов - просто прыгнули на участок кода очистки
- Конечные автоматы

Про механизм defer в Golang - это замечательный инструмент, который наиболее часто используется для освобождения ресурсов. Принцип его работы максимально простой: открыли ресурс, после этого мы в defer пишем код для освобождения этого ресурса. Вызов defer кладется на стек и после выполнения основной функции вызывает этот код.

Вопрос 19

Нарушение структурной организации кода. Исключения. Состояния и перезапуски.
Ответ:
Несмотря на то, что программы стараются писать в чистом структурном стиле, существуют в языках механизмы, которые нарушают эту концепцию для достижения какой-то цели.

Исключения - мы и так знаем, что это средство для работы с неожиданными и нестандартными ситуациями в программе. Нарушение структурной организации заключается в том, что при выбросе исключения оно пробивается вверх через внешние блоки вплоть до того, пока не встретит какой-либо обработчик (на уровне нашей программы либо ОС). Нарушение структурной организации можно аргументировать тем, что это делается для быстрого аварийного завершения программы или возвращения ее в корректное состояние.

Есть другой интересный механизм, который так же нарушает структурную организацию - механизм состояний и перезапусков. Идея его заключается в том, что при выбросе

исключения состояние отлавливается и пробрасывается вплоть до обработчика, в котором можно решить, что с этим делать дальше и затем продолжить с этого же места, в котором остановились. Такой механизм используется в Common Lisp.

Вопрос 20

Нарушение структурной организации кода. Динамическая область видимости переменных.
Ответ:
Несмотря на то, что программы стараются писать в чистом структурном стиле, существуют в языках механизмы, которые нарушают эту концепцию для достижения какой-то цели.

У переменных может быть scope:

- Лексический - визуально можем по коду оценить, где доступна та или иная переменная (это видно по самим структурным блокам)
- Динамический - переменная определяется не лексически в функции, а, грубо говоря, на стеке вызовов. Это бывает полезно и удобно в случаях, когда нам нужно одну и ту же переменную прокидывать на несколько функций вглубь (например, при запросах к серверу, требующему авторизацию, нужно передавать везде токен доступа; также, например, в Closure переменные из динамического scope применяются для конфигурации: например, для переопределения стандартного ввода/вывода)

Вопрос 21

Монады. Связывание операций. Монада Identity, Maybe и State.
Ответ:
В сообществе любителей Haskell прижилась шутка, что каждый Haskell-программист должен в процессе своего обучения написать одно или несколько руководств по монадам. Так что в текущем информационном пространстве много разнообразных определений монад, что, по правде говоря, ломает голову.

Тем не менее, наиболее приземленным понятием монад: отличный инструмент для организации последовательных вычислений, который заключается в том, что мы операции связываем в цепочку (как минимум, в примерах определяется функция bind, предназначенная для этого связывания).

Самым простым видом монады считается Identity, которая является входным типом в мир монад. Ее смысл заключается в самом предназначении монад, а именно в связывании операций. Рассматривали пример того, как можно сложить несколько переменных: определили функцию связывания как применение функции к аргументу и вкладывали

bind в последующие функции, что в итоге развернулось в сумму значений. Возникает соответствующий вопрос "А нафига нам просто вызывать через монаду последовательно функции - не проще ли просто какую-нибудь композицию сделать?". Преимуществом использования даже такой простой монады является контроль над этой последовательностью вычислений: мы можем сложнее определить функцию bind, что может дать нам дополнительный функционал (например, подробный отладочный вывод результатов вычислений или что-то еще сложнее).

В современном программировании распространен тип данных Maybe, он также может называться Optional и т.п. Идея этого типа заключается в том, что он может содержать в себе как Nothing (ничего), так и Just (что-то) - к этому добавляются методы по проверке и извлечению значения. Теперь что касается Maybe монады: представим себе ситуацию, что мы рандомно получаем значение для нескольких переменных, тип которых Maybe, т.е. мы не знаем, в каких лежит значение, а в каких - нет. Нам нужно сложить значения этих переменных: мы сначала должны проверить, что все значения представлены, а потом только сложить - если хоть где-то Nothing - возвращаем Nothing. В чем проблема: нам для реализации такого, какзалось бы, простого алгоритма следует либо проверять все переменные сразу, а потом делать вычисления, либо как-то вложенно, через большое число вложенных if-ов делать проверки - не пойдет. На помощь тут приходит Maybe монада:

- Содержит в себе тип Maybe
- Определяем функцию связывания в цепочку bind, которая принимает значение и следующую функцию: если значение Nothing, то возвращаем Nothing и прерываем цепочку, либо вызываем следующую функцию.

Свойства монад:

- Левая идентичность: `bind (return a) (fun x → f x) ≡ f a`
- Правая идентичность: `bind m (fun x → return x) ≡ m`
- Ассоциативность (после η редукции):
  - `bind (bind m f) g ≡ bind m (fun x → bind (f x) g)`

В дополнение к Maybe монаде есть монада Either/Result: вместо Nothing хранит второе значение своего типа - обычно предназначена для хранения описания ошибки.

Монада State на то и монада состояния, что в ней определяется контекст, с которым идет в работа в цепочке вычислений. Каждый вызов данной монады - функция, которая переводит состояние в выкинутое значение (которое может использоваться) и новое состояние. Как устроена функция связывания:

- Принимаем предыдущий вызов монады и функцию, которая принимает выброшенное значение и выдает новое преобразование
- Вызываем переданную функцию на выброшенное значение и меняем состояние.

Есть новые функции:

- get - получить состояние
- put - обновить состояние

Пример применения - счетчик.