Если вы видите что-то необычное, просто сообщите мне.

Мозги Haskell

Чисто технически, Haskell бросает нам большой вызов. Но часто это больше чем просто грубый технический навык знания какого-то нового языка. В этой части, мы обсудим некоторые причины, почему люди видят Haskell вызовом. Так же направим мысли при изучения языка взглянуть на некоторые техники для ускорения результата.

Часть 1: Ослабим страшный взгляд Haskell

Добро пожаловать в первую часть серии "Мозги Haskell"! Кроме описанных базовых понятий в прошлой серии статей, остается еще много работы, которая требует изучения нового! Эта серия статей затрагивает психологиеческие барьеры людей встречающие изучение нового языка. И дает советы для преодоления проблем.

Первая часть будет иметь совй взгляд на Haskell, в большом сообществе программистов. Мы посмотрим почему Haskell часто воспринимается как пугающий и трудный, и почему это не должно вас пугать.

Если вы бесстрашны и хотите попасть с корабля на бал, можно двигаться сразу ко второй статье. Там мы обсудим текущие процессы изучения языка. если вы хотите перейти сразу к изучению и написанию кода, то пожалуйста.

Академический язык

Люди долго считали Haskell в основном языком исследования. Он построен на лябда вычислениях, возможно наипростейший, чистейший язык программирования. Это дает огромное количество возможностей связывания с отличными идеями в абстрактной математике, в первую очередь для студентов, профессоров и докторов. Эта связь настолько элегантна, что математические идеи могут быть легко представленны в Haskell.

Но эта связь имеет цену доступа. Важные идеи Haskell включают функторы, монады, категории и т.д. Они хороши, но только некоторые без математической степени имеют представление что значат эти понятия. Сравним эти понятия с другими языками: класс, итератор, цикл, шаблон. Эти гораздо понятнее, и языки используют в качестве преимущества.

Отходя от этой терминологии, большой академический интерес это отличная вещь. Однако, на стороне производства, инструментарий не будет достаточный. Просто сложно обслуживать большие Haskell проекты. Как результат, компании не имеют большого интереса использовать его. Это значит, что нет особого влияния на академический баланс языка.

Распространение знания.

Сетевые результаты Haskell академического первенства это перекошенная база знаний. В академии несколько человек проводят много времени на относительно маленьких проблемах. Учитывая другие академические поля, как вирусология. У вас есть некоторые эксперты которые понимаю вирусы на достаточно высоком уровне, и большинство не знают об этом ничего. Нет вирусологов-любителей. К сожалению, этот тип распространения знаний неблагоприятный для обучения новых людей теме.

Естественн, люди должны общаться с теми, кто знает больше их. Но правда в том, что они не хотят чтобы учителя тоже учились. Это сильно помогает в изучении если общаться с человеком который надавно касался этой темы. Скорей всего они помнят подводные камни и разочарования которые они встречали ранее, поэтому они смогут помочь вам избежать этого. Но когда распределение приходит в экстремум, нет среднего класса. Есть несколько человек которые могут обучить слушателей. В добавок не помня старых ошибок, эксперты используют сложную терминологию. Новые люди в теме могут чувствовать пугающее отчаяние.

Перелом в производстве

Недостаток производственной работы, который обсуждался выше существенно способствует этому разрыву. Другие языки типа C++, имеет строгих академические последователей. Но после использования его компаниями в производстве, он не столкнулся с проблемой передачи знаний, которые имеет Haskell. Компании использующие C++ не имеют выбора, кроме как обучать людей языку. Множество этих людей застряли в языке достаточно, чтобы обучить следующее поколение. Это создает более плавную кривую

обучения.

Хорошие же новости для Haskell заключаются в том, что есть множество улучшенных инструментов за несколько последних лет. Это привнесло возрождение в язык. Множество компаний начали использовать его в производстве. Проходят больше встреч, больше людей пишут библиотеки, для большинства критических задач. Если это продолжится, Haskell надеемся достигнет переломного момента где распространение становится уже нормальным.

Ключевая информация

Если вы один из тех кто заинтересован в изучении Haskell, или кто пытался изучить Haskell в прошлом, есть одна ведь которую нужно знать. В то время как абстрактная математика это излишество в повседневной жизни. Десятки языковых расширений должны выглядеть пугающими, но вы можете выбрать по одной.

На встрече Haskell eXchange 2016, Дон Стюарт из Standard Chartered начал разговоро о компаниях которые используют Haskell. Он объяснил, что они не часто используют что-то вне констукруций ванильного Haskell. Они просто им не нужны. Всё что вам нужно, скажем, линзы, вы можете получить без них.

Haskell отличается от большинства языков. Он ограничивает вас по своему. Но эти ограничения совершенно не являются тем на что они похоожи. Вы не можете исопльзовать их для цикла. Используйте рекурсии. Вы не можете изменять переменные. Поэтому создавайте новые используемые в выражениях. Просто каждый раз берете новую.

Куда дальше?

Теперь зная, что Haskell не что-то страшное. Вы должны двигаться дальше. Зная подробнее о процессе изучения и нескиольких хитростях вы можете продолжить изучение дальше.

Часть 2: Обучение обучению

В первой части этой главы, мы проверили пугающий фактор Haskell. Увидели пару причин, почему люди видят Haskell вызывающим, и почему, возможно, они не дложны этого делать. В этой главе, мы затроним несколько тем прошлой статьи. Изучи как изучать Haskell(и другие вещи). Изучим некоторые общие идеи обучения и обсудим как применять их к программированию.

Дальше перейдем к части 3, где вы изучите еще больше специфических техник для обучения. Мы начнем погружаться в применение этим идей к Haskell.

Уорен Баффетт и составной интерес

Уорен Баффетт часто говорит о производительности. Он говорит, что он читатет порядка 500 страниц в день, и это один из ключевых моментов его успеха. Знание, согласно Баффетту, это составной интерес. Чем больше ты получаешь и устанавливаешь связи, тем большее это собирается в единую картину и становится возможным строить на её основе.

В чем заключается апофеоз этой фразы. Я нахожу её правильно звучащей при изучении разных тем. Я увидел, как мои знания стали строится сами по себе. До сих пор неправильное понимание этой фразы ведет людей проводя много времени реализуя этот принцип.

Простой факт, что средний человек, не имеет времени для чтения 500 страниц в день. Первое, если он читает так много, Уорен Баффетт скорей всего опытный быстрочитающий человек, поэтому ему нужно меньше времени. Второе, он гораздо больше контроллирует свое время, в отличии от большинства других людей. В моей работе разработчика ПО, я н

емогу проводить полностью 80% моей работы в чтении и думании. Этим я заставлю свою команду и проект менеджера делать, что-то со мной.

В среднем люди будут видеть этот совет и решат, начать читать тонну литературы вне рабочего времени. И они даже преуспеют в чтении 500 страниц в день ... на пару дней. Ну а потом жизнь вернется в обычное русло. Они не захотят тратить своё время через несколько дней на чтение, и привычка будет отложена.

Лучшее применение

Ну что же как достичь эффект описанный выше? Реальное непонимание, я нашел в следующем. Ключевой момент в подходе это время, но не среднее. Делая маленькие, повторяющиеся вклады, будут иметь большее вознаграждение позже. Конечно, чем больше это вложение тем больше вознаграждение тоже. Но если вложение заставляет нас бросить привычку, то это плохо.

Пользуясь этой идей, мы можем применить её к другим темам, включая Haskell. Мы можем быть настроены посвятить час каждый день для изучения некоторые частичек идей Haskell. Но это часто не возможно. Гораздо проще посвятить 15 минут в день, или даже 10 минут в день. Это будет признаком того, что мы тратим на обучение. В любой день, может быть трудно выделить это время для чего-то. Ваше расписание, не должно позволять длится этому долго. Но вы высегда можете найти 15 минут. Это будет гораздно проще, чем "начать в любой день", и даст больший результат.

Согласно принципу, прогресс основан на времени. Отдавая 15 минут паре различных проектов, я довольно далеко продвинулся. Мне удалось гораздо больше, чем если бы я вытался получить час времени тут и там. Я смог начать писать статьи, так как этому посвятил 20 минут в день. И как только я провел месяц таким образом, я оказался в отличной форме.

Джош Вайцкин и преодоление труностей.

Еще с одной хорошей идеей обучения обучению я столкнулся в "The Art of Learning" Джош Вайцкин. Он одаренный шахматист и международный мастер. Он описал историю, которая была всем слишком знакома, так как в детстве я тоже играл в шахматы. Он видел множество молодых ребят со способностями. Он могли победить всех вокруг в школе и в шахматном кружке. Но они никогда не боролись с сильными игроками. Как результат, они заканчивали выходом из шахмат вовсе. Они столько вкладывали в идею победы в каждой игре, что сильно ущемляло гордость в моменты когда они проигрывали.

Если мы слишком сосредоточимся на нашем эго, мы испугаемся показаться слабыми. Это заставляет нас избегать конфронтации со знаниями, где мы слабы. Это и есть то, что нам нужно усилить. Если мы никогда не обращались в эту часть, мы никогда не улушим её, и не сможем побороть большой вызов.

Побороть HASKELL

Как это влияет на изучение Haskell, или на программирование в общем? В конце концов, программирование не соревновательная игра. И все еще есть способы которые могут повредить нашему мышлению. Наверное, стоит держаться по дальше от этой темы, так как она кажется сложной. Мы сомневаемся, что можем преуспеть в изучении. И переживаем что эта неудача раскроет нам, что мы совершенно не подходим для работы разработчиком на Haskell. Хуже если мы боимся просить других разработчиков о помощи. Что если они посмотрят на нас сверху вниз если у нас не будет хватать знаний?

У меня есть на это три ответа. Первый, я повторюсь заметкой из первой части. Тема кажется бесконечно пугающей когда вы ничего о ней не знаете. Как только вы узнаете базовые вещи, у вас есть понимание того, что вы успускаете из виду. Поймите идею как можете, запишите её простым языком. Вы можете не знать сам объект. Но он не будет для вас чем-

то неизведанным.

Второе, кого волнует, результат приложенны сил к вашему обучению? Попробуйте еще раз! Изучение темы может потребовать несколько подходов, прежде чем вы поймете её. У меня это заняло три попытки прежде чем я понял монады!

Наконец - те самые люди, перед которыми мы боимся признать нашу слабость, это те же люди, которые на самом деле могут помочь нам преодолеть эту самую слабость. Даже больше, они часто рады нам помочь! Это результат нашего первобытного страха показаться неполноценным и быть отвергнутым другими. Это сложно, но ни не возможно.

Заключение

Поэтому помните, главное! Сфокусируйтесь на малом в начале. Не тратте на изучение больше чем 15 минут в день, возьмите проект с явным прогрессом. Сохраняйте импульс продолжая работать каждый день. Не переживайте если идея кажется вам сложной! Вполне нормально, если вам потребуется несколько попытко, чтобы что-то изучить. И самое главное, не бойтесь просить помощи.

Отличный способ сохранить импульс - это прочитать главу 3. Мы углубимся в практики и применим их!

Часть 3: Haskell и взвешенная практика

Вы были в ситуации когда вы пытаетесь изучить что-то в определенное время, и застряли на этом? Шанс, что вы изучаете предмет не лучшим способом. Но как вы можете узнать, что входит в это "хорошее" изучние вашей темы? Оно может разочаровать при попытке найти что-то в интернете. Большинство людей не думают о том способе которым они изучают новое. Они изучают, но они не могут выразить и обучить других людей тому, что они делают, потому что для этого нет инструкции.

Часть 1 этой главы говорит нам о том, почему вы не должны бояться пробовать изучить Haskell. Часть 2 обсуждает некоторые техники на высшем уровне. Эта часть пройдется по паре ключевых идей из "Art of Learning" Джоша Вайтцкина. Мы рассмотрим на то, как избежать проблему застрявания.

Первая идея, о которой мы загооврим это идея взвешенной практики. Цель этой практики прицелиться в конкретную идею и попробовать улушчить определенные навыки до тех пор, пока они будут только в подсознании. Вторая идея - роль ошибок в изчении любого нового навыка. Мы будем использовать это для стимула ведущего дальше, которы предостережот нас от этой же идеи в будущем.

Мы раскроем это в 4 части, в которой разберем применение этой техники в изучении Haskell.

Некоторые техники проще понять, если вы уже имеете какую-то практику в изучении Haskell.

Взвешенная практика

Предположим, на минуточку, вы учите композицию на фортепьяно. Наибольшее искушение здесь это "учите" часть повторяя композицию от начала до конца. Часть у вас получлается,

часть - нет. В итоге, вы изучили большую часть. Это заманчивый способ практки по нескольким причинам:

1. Это "очевидный" выбор.
2. Он позволяет нам проиграть часть, которую мы уже узнали и получить удовольствие от этого.
3. Скорей всего в результате мы выучим композицию.

Однако, это не оптимальный метод со стороны обучения. Если вы хотите улучшить вашу возможность играть вашу композицю от начала до конца, вы должны сфокусироваться на слабых местах. Вам нужно найти определенные пассажи, которые вам даются хуже всего. Как только вы их определите, вы можете разбирать их дальше. Вы можете найти определенную размерность или даже ноты с которыми у вас проблемы. Вы должны практиковать слабые места снова и снова, исправляя одну маленькую вещь за другой. Отсюда, вам нужно пройти весь путь и это заставляет вас ожидать, что вы будете уверенны что вы изучили все свои "слабыу" места.

Фокусируясь на маленьких вещях - главная часть. Вы не можете взять хитрый пассаж о котором ничего не знаете и сыграть его правильно от начала до конца. Вы можете начать с одной части, которая заставляет вас ускорится. Вы можете тренироваться много раз просто фокусируясь на получении последней ноты и двигаться руку дальше. Вас не волнует ничего кроме реппетиции. Как только движение вашей руки переходит в подкорку, вы можете идти дальше к следующей части. Следующий шаг уже нужен, чтобы убедиться что вы достигли первых трех нот.

Это идея взвешенной практики. Мы фокусируемся на одной вещи во времени, и практикуем её неспеша. Бездумная практика, или практика рази практики будет давать вам маленький прогресс. Даже может уменьшить прогресс если мы заимеем плохие привычки. Мы можем применять это к любому навыкы, включая программирование. Мы хотим нахватать маленькие привычки, которые будут постепенно делать нас лучше.

Ошибки

Взвешенная практика это система построения навыков, которую мы хотим. Однако, есть множество привычек, которые мы не хотим. Мы делаем много вещей, ошибки которых мы поймем позже. И наихудшая вещь это когда мы понимаем, что мы повторяем одну и ту же ошибку.

Вайцкин отмети в "Art of Learning": Если студент любого предмета может избежать хотябы повтора одной и той же ошибки дважды, они быстро достигнут высот в изучаемой теме. Мы так же можем сделать это если будем избегать ошибок в целом, но это не возможно. Мы всегда делаем ошибки, пробуя что-то впервые.

Сначала нужно принять, что ошибки будут случаться. Как только мы это сделаелм, у нас будет решение как этого избежать. Невозможно избежать повторения ошибок, но мы можем предпринять шаги, которые сократят их количество. И если мы такое можем сделать, то увидим существенное улучшение. Наше решение будет хранить все ошибки которые мы сделаем. Описывая всё, что происходит, мы резко сократим повторение ошибок.

Практикуем HASKELL

Теперь нам нужно сделать шаг назад в мир написания кода и спросить себя, как мы можем применить эти идеи к Haskell. На каких темах практики написания кода мы можем остановиться?

Вы можете написать приложение, и сфокусироваться на приобретении следующей привычки: прежде чем вы напишете функцию, нужно вынести неопределенности и убедиться, что типы сигнатур комплируются(мы это обсудим дальше в части 4). Не важно если вы всё остальное сделали правильно! Приобретя эту привычку можноприступать к следующему шагу. Вы можете убедиться в том, что вы всегда пишете вызов функции прежде чем вы её реализуете.

Я выбрал эти примеры, потому что есть две вещи которые тормозят нас больше всего при написании функции. Первое - это нехватка ясности, потому что мы не знаем точно как наш код будет использовать функцию. Второе - повторение работы когда мы поняли. что нам нужно переписать функцию. Это случается, когда мы не учитываем дополнительные возможности. Эти привычки, продуманны таким образом, чтобы ваша жизни становилась

легче, когда нужно реализовать их.

Есть еще несколько похожих идей:

1. Прежде чем писать функцию, напишите коммент описывающий эту функцию.
2. Прежде чем использовать выражение из библиотеки, добавьте её в ваш `.cabal` файл. Затем, напишите выражение `import` чтобы убедиться, что вы используете правильную зависимость.

Еще один способ - знать как проверить кусок функциональности прежде чем реализовать её. В идеале, написать юнит тесты для функции. Но если это что-то простое, как получение строки ввода и её обработка каким-то способом, вы можете опустить простые идеи. Вы можете вложить в рабочую программу, для командной строки, два вида входных данных. Если вы знаете свой подход до того, как начнете программировать, это имеет значение. Имеет смысл написать план тестирования в каком-то документе для начала.

В большинстве случаев триггером для приобретения этой привычки это написание новой функции. Триггер это важная часть приобретения новой ошибки. Это действие которое подсказывает вашему мозгу, что вы должны делать что-то не привычное. В этом случае, триггером будет написание `::` для сигнатуры. Каждый раз, выполняя это, вспоминайте о вашей цели.

Есть еще идея с множеством триггеров. Каждый раз вы выбираете структуру для хранения ваших данных(списко, последовательность, набор, карты и т.д.), как минимум три разных типа. Как только выходите за предели базовых вещей, вы найдете уникальную силу в каждой из структур. Будет ползено если вы выпишите условия сделанного выбора. Триггер, в этом случае, будет проявляться каждый раз, когда вы пишете ключевые данные. Для более сложной версии, триггером может быть быть написание левой скобки для начала списка. Каждый раз, делая это, спросите себя: можете ли вы использовать другую структуру?

И последняя возможность. Каждый раз делая синонимы типов, спросите: стоит ли создавать новый тип? Это часто ведет к улучшеню времени компиляции. Вы скорей всего видели сообщения об ошибках и большую часть получите еще при сборке. Триггер тут тоже простой: каждый раз когда пишете ключевое слово типа.

Это важдые вещи. Не пытайтесь выучить сразу все! Вам нужно выбрать одну, практиковать её пока она будет работать подсознательно, и затем двигайтесь к другой. Это самая сложная часть взвешенной практики: управлять своим терпением. Самое большое влечение это двигаться и пробовать новые вещи, прежде чем усвоится привычка. Как только вы переключитесь на другие вещи, вы можете потерять, то над чем работали. Помните, обучение сложный процесс! Вам нужно сделать небольшое исследование которое требует определенное время. Вы не сможете ускорить этот процесс.

Отслеживание ошибок

Давайте представим различные способы, которые помогут нам избежать повторения ошибок в будущем. Опять, эти различные навыки вы приобретаете с помощью взвешенной практики. Они не появляются часто, и вам не нужно их "практиковать". Вам нужно просто помнить, как исправить некоторые ошибки, чтобы в тот момент когда они появятся вы сможете это применить еще раз. Вы можете вести список ужасных ошибок в электронном виде, которые встречаете во время программирования.

1. Как себя повел сборщик?
2. В чем заключалась проблема с кодом?
3. Как это можно исправить?

Для примера, подумайте о времени когда вы были уверенны о том, что код верен. Взгляните на ошибку, затем на ваш код, опять на ошибку. И вы всё еще уверенны, что код правильный. Конечно, компилятор почти всегда прав. Вы хотите это записать, чтобы вас нельзя было обвести вокруг пальца еще раз.

Другой хороший кандидат, это ошибки исполнения где вы не можете никаким образом понять где ошибка возникает в вашем коде. Вы хотите записать этот опыт таким образом, чтобы в следующий раз вы могли быстро решить проблему. Выписывание заставляет вас избегать повторений ошибок.

Есть еще глупые ошибки, которые вы должны записывать потому, что они учат вас быстрее. Например, вначале вы можете использовать `(+)` оператор для складывания двух строк, вместо `(++)` оператора. Выписывание таких ошибок позволит выучить свойство гораздо

быстрее.

Последняя группа вещей, которые нужно выписывать отличноее решение. Не просто исправление багов, но решение ваших главных проблем программирования. Для пример, вы находите вашу программу слишком медленной, но вы использовали наилучшие сткруктуры данных для улучшения. У вас так же имеются записи того, что вы делали. Таким образом, у вас будет возможность применить решение еще и в следующий раз. Эти вещи дают пищу для размышления в собеседованиях. Собеседования часто можно услышать вопросы о проблемах которые вы решали, чтобы понять ваши мотивационные способности.

У меня есть один пример, который показывает хорошее и плохое распознавание ошибок. У меня есть жуткий баг, когда я пытаюсь собрать Haskell проект используя Cabal. Я помню была ошибка компоновщика, которая не указаывала на отдельный файл. Я сделал отличную мысленную заметку, что решение было добавить что-то в файл `.cabal`. Но я не записал полнстью контекст или решение. В будущем, я увидел ошибку компоновщика и зная что нужно что-то сделать в `.cabal` файле, но я не помнил, что именно нужно. Поэтому мне приходилось повторять эту ошкбу пока я не выписал полностью решение вопроса.

Заключение

Это часто повторяемая мантра, что практика делат всё лучше. Но кто улучшает свои навыки, может вам сказать, только хорошие практики улучшают. Плохие или безумные практики будет мешать вам двигаться дальше. Или хуже, будут прививать вам плохие привычки, которые будут требовать дополнительное время для их отмены. Взвешенная практика это процесс укрепления знания с помощью создания привычки. Вы выбираете одну и фокусируетесь на ней, и огнорируете всё остальное. Затем вы её используете пока она не попадет вам на закорку. И только потом вы двигаетесь дальше. Этот подход требует огромного терпения.

Последняя вещь, которую нужно понять о обучении нужно принять возможность ошибки. Как только это будет сделано, мы можем сделать план записи этих ошибок. Тогда мы можем изучить их и не повторять больше. Это резко увеличит скорость разработки.

Часть 4: Обучение управляемое компиляцией

В части 2 и 3, мы обсудили некоторые общие цели идей обучения, и увидели пару их применений к Haskell. Но в какой-то момент этим необходимо воспользоваться. Как мы на самом деле изучаем что-то с нуля?

В этой части я поделюсь своим подходом для решения проблем обучения. Я буду называть это "Обучение управляемое компиляцией".

Дилемма

Представим. Вы сделали отличный прогресс в личном проекте. Вам нужн добавить еще один компонент чтобы всё собрать во едино. Вы используете внешнюю библиотеку в качестве помошника и вы застряли. ВЫ гадаете с чего начать. Вы подглядываете в документацию библиотеки. Но это не помогает.

Так как документация блиблиотек Haskell не всегда хороша, но есть спасительная благодать. Haskell жестко типизирован. В общем, когда он собирается, то это работает как мы ожидаем. На крайняк это более распространнено в Haskell, чем в других языках. Это может быть обоюдоострым мечом, при желании изучить новую библиотеку.

С другой стороны, если вы можете сколотить правильные типы для функций, вы на правильном пути. Однако, если вы не знаете достаточно о типах в библиотеке, то трудно понять откуда начинать. Что делать, если вы не знаете как собирать что-то правильно? Вы можете написать много кода с догадками, но в результате вы получите множество сообщений с ошибками. Так как вы не знакомы с типами, это будет трудно дешифровать.

Чтобы изучить новую библиотеку или систему, вам нужно начать с написания маленького кода, такого, чтобы он мог скомпилироваться. Идея зваимодейстовать с обечением

управляемое компиляцией очень просто для TDD. Для начала давайте взглянем в общем на неё.

TDD

TDD это пример разработки ПО где вы пишете сначала тесты потом сам код. Вы предполагаете эффект, который должен делать код, и какой результат функции должен быть. Затем вы пишете тесты указывая ожидания от функции. Вы пишете только исходный код для свойства как только вы удовлетворенны набором ваших тестов.

Как только вы это выполните, результаты теста ведут разработку. Вам не нужно проводить много времени выясняя, какой кусок кода вы должны реализовать. Вы находите первый падающий тест, исправляете его и повторяете. Задача написать как можно меньше кода для прохождения теста. Очевидно, вы не должны просто хардкодить функцию, для прохождения тестов. Ваш тест должен быть достаточно крепким, насколько это возмоно.

Если вы пытаетесь написать код по возможности проходящий тестирование, вы можете закончить с неорганизованным кодом. Это не есть хорошо. Главная цель в TDD сражаться с циклом Red-Green-Refactor. Сначала вы пишете тесты, которые падают(Red). Делаете так, чтобы все тесты стали(green). Затем реорганизовать ваш код таким образом, чтобы он подчинялся общему стилю который вы используете. После того как это завершено, вы двигаетесь к следующему функциональному коду.

Обучение управляемое компиляцией

TDD это великолепно, но мы не можем его применять к изучению новой библиотеки. Если вы не знаете типы, вы не можете написать хорошие тесты. Поэтому нужно использовать другой процесс. В некотором роде, мы используем систему типов и компилятор в качестве теста понимаем ли мы наш код. Мы можем использовать знания чтобы сделать код, который удовлетворяет двум параметрам:

1. Провести нашу разработку и знать, что именно мы хотим реализовать.
2. Избежать малодушия при виде "горы ошибок".

Подход выглядит сделующим образом:

1. Определим функцию которую реализуем, и затем сделаем заглушку как `undefined`. (Код должен компилироваться)
2. Сделаем небольшие изменения определении функции, так что бы проходила компиляция.
3. Определим следующий кусок кода, для написания, будь-то это `undefined` значение, которое нужно заполнить, или заглушка для конструктора объектов.
4. Повторяем шаги 2-3.

Отметим, что в конце каждого шага этого процесса, вы должны иметь компилируемый код. Здсь значение `undefined` это отличный инструмент. Это значение в Haskell которое может принимать любое значение, таким образом, что вы можете сделать заглушку для любой функции или значения в ней. Ключ в том, чтобы видеть следующий уровень реализации.

CDL на практике

Вот пример, запуска кода через этот процесс от "One Week Apps". Сначала я определяю функцию которую я хочу написать.

```
swiftFileFromView :: OWAAppInfo -> OWAView -> SwiftFile
swiftFileFromView = undefined
```

Эта функция говорит, что мы хотим иметь возможность принимать `App Info` объект нашего Swift приложения, так же как и `View` объект, и создать Swift файл для вида. Теперь мы должны определить следующий шаг. Мы хотим, чтобы наш код собирался всё время пока мы решаем проблему. Тип `SwiftFile` это обертка вокрук списка типа `FileSection`. Поэтому мы можем сделать так:

```
swiftFileFromView :: OWAAppInfo -> OWAView -> SwiftFile
swiftFileFromView _ _ = SwiftFile []
```

И он всё еще компилируется! Предположительно, он совершенно не завершен! Но мы сделали маленький шак в правильном направлении.

Для следующего шага, нам нужно определить какие `FileSection` объекты помещаются в список. В этом случае мы хотим три различных разделов. Первая - у нас есть раздел комментариев в верху. Второе - есть раздел "важное". И есть главный раздел реализации. Мы можем поместить выражение в эти три списка, и затем использовать заглушку ниже:

```
swiftFileFromView :: OWAAppInfo -> OWAView -> SwiftFile
swiftFileFromView _ _ = SwiftFile [commentSection, importsSection, classSection]
  where
    commentSection = undefined
    importsSection = undefined
    classSection = undefined
```

Этот код всё еще компилируется. Теперь мы можем заполнить раздел по очереди, вместо того, чтобы напрягаться написанием кода целиком. Каждый из разделов имеет свою компонентную часть, которую мы разобъем дальше.

Используя наши знания о типе `FileSection`, мы можем использовать конструктор `BlockCommenSection`. Он просто принимает список строк. Так же, мы воспользуемся конструктором `ImportSection` для импорта раздела. Он так же принимает список. Продолжим следующим образом:

```
swiftFileFromView :: OWAAppInfo -> OWAView -> SwiftFile
swiftFileFromView _ _ = SwiftFile [commentSection, importsSection, classSection]
  where
    commentSection = BlockCommentSection []
    importsSection = ImportsSection []
    classSection = undefined
```

И снова наш код компилируется, а мы в свою очередь сделали небольшой прогресс. Теперь определим какая строка нам нужна для раздела комментарив, и добавим её. Теперь можно добавить `Import` объектов для раздела `imports`. Если что-то пойдет не по плану мы увидим только одну ошибку и мы будем знать где она происходит. Это делает процесс разработки гораздо быстрее.

Заключение

Мы поговорили о подходе изучения новых библиотек, но это подходит и к обычной разработке. Избегайте желания уходить с головой и писать сразу сотни строк кода! Вы пожалеете об этом когда увидите кучу сообщений об ошибке! Неспешность и твердость позволит выйграть гонку. Вы выполните гораздо больше если разобъете на маленькие детали, и воспользуетесь компилятором в качестве проверки вашего кода.