Лекция 3: Нетипизированное λ-исчисление

1. Лямбда-исчисление, введение

Лямбда-исчисление — теоретическая основа функционального программирования, поэтому явно заслуживает отдельного обсуждения, тем более что это само по себе математически красивая теория. Обсуждение, правда, будет короче, чем обычно в курсах по ФП, всего одну пару. Так что мы успеем только нетипизированное лямбда-исчисление, тогда как реальные языки программирования базируются на типизированных лямбда-исчислениях, конечно. Но, во-первых, их несколько, во-вторых, они сложнее, и в-третьих, про них есть хорошие книги и хорошие онлайн-курсы (на первой лекции рекомендовались курсы Дениса Николаевича Москвина https://www.lektorium.tv/course/22797, https://stepik.org/course/75 и https://stepik.org/course/693). Тем не менее, общая идея и связь лямбда-исчисления с F# должны быть понятны.

Итак, лямбда-исчисление — это формальная система, основанная на λ-нотации, которая была разработана как ещё один способ формализовать понятие «вычисление» американским логиком Алонзо Чёрчем аж в 1930-х годах (то есть задолго до появления первых ЭВМ). Лямбда-исчисление в каком-то смысле более простая альтернатива машинам Тьюринга и эквивалентно им по вычислительной мощи (то есть всё, что можно посчитать в лямбда-исчислении, можно посчитать на машине Тьюринга и наоборот), известный вам тезис Чёрча[1] формулируется относительно машины Тьюринга, но назван в честь Чёрча (ну, на самом деле его называют тезисом Чёрча-Тьюринга иногда). Кстати, есть ещё нормальные алгорифмы Маркова, которые ещё одна довольно известная альтернатива и машинам Тьюринга и λ-исчислению Чёрча, про них часто забывают, так что напоминаю тут, но речь дальше будет не про них.

Лямбда-исчисление в языках программирования появилось довольно давно, аж язык LISP (а это 1958 год!) черпал вдохновение из лямбда-исчисления (хотя только его относительно новые диалекты можно в полной мере назвать функциональными языками). В современные языки промышленной разработки понятия из лямбда-исчисления начали активно проникать где-то в середине 2000-х, в C# лямбда-функции появились в версии 3 (август 2007), в C++ — в стандарте C++11 (2011 год), в Java — в версии 8 (2014 год). Понятия, типичные для функционального программирования, понятно, существовали в языках и раньше, да и нынешняя поддержка «лямбда-функций» в языках сильно отличается от той теории, про которую речь пойдёт сегодня, но тем не менее. В современном программистском сообществе считается приличным хотя бы немного быть в курсе, что за лямбда-исчисление такое, а знание терминов типа «Y-комбинатор» или «нумералы Чёрча»

[1] «любая функция, которая может быть вычислена, может быть вычислена машиной Тьюринга»

всегда принесёт дополнительные очки при общении с коллегами, даже если все зачем-то программируют на Java.

2. Лямбда-нотация

Лямбда-исчисление имеет в основе лямбда-нотацию, которая была придумана просто для удобного представления функций. В традиционной математике функции обычно именуются как-нибудь вроде f, g и т.д., но даже на лекциях по матанализу не редки были ситуации, когда у лектора заканчивались буквы латинского алфавита. А если у нас всё исчисление построено на функциях, то именовать каждую вообще безнадёжно. Поэтому функция, отображающая x в некое выражение $t[x]$ (то есть $x \to t[x]$ в привычной нотации) записывается в лямбда-нотации как $\lambda x.t[x]$. После λ и до точки идёт список аргументов через пробел (мы скоро увидим, что аргумент нужен только один, несколько аргументов — это синтаксический сахар). После точки идёт выражение, которое может внутри использовать аргументы.

Например, $\lambda x.x$ описывает тождественное отображение, а $\lambda x.x^2$ — функцию, которая свой аргумент возводит в квадрат.

2.1. Применение функции

Применение функции выглядит как «лямбда-выражение пробел аргументы». Это тоже отличается от привычного из матанализа $f(x)$, и на это тоже есть свои причины. Во-первых, в матанализе нередко $f(x)$ — это сама функция, а не значение функции в точке x (то есть матану-то не надо различать формальные и фактические параметры функций, это понятно из контекста). Во-вторых, в матанализе нужны функции от нескольких аргументов, поэтому надо выделять список параметров синтаксически, в лямбда-исчислении не нужны, и скобки были бы неудобны из-за громоздкости.

Итак, в лямбда-исчислении $f(x)$ обозначается как fx — в точности как в F#, да. При этом, правда, возникает неоднозначность, $\lambda x.x+y$ — это $\lambda x.x$, применённая к аргументам $+$ и y, или $\lambda x.(x+y)$. В лямбда-исчислении выбран второй вариант, то есть выражение под лямбдой распространяется вправо настолько, насколько это возможно (то есть до конца строки или закрывающей скобки). Если бы мы хотели записать первый вариант, надо было написать $(\lambda x.x) + y$.

Парочка примеров записи применения функции:

$$(\lambda x.x^2)\ 5 = 25$$

$$(\lambda x.\lambda y.x + y)\ 2\ 5 = 7$$

Обратите внимание, $(\lambda x.\lambda y.x + y)$ — это функция от аргумента x, возвращающая функцию от аргумента y.

2.2. Каррирование

Собственно, соображение, что мы вместо функции от двух аргументов можем иметь функцию от одного аргумента, возвращающую функцию от одного аргумента, и называет-

ся каррированием. Функции от нескольких аргументов просто вводятся как дополнительная удобная нотация для функций, возвращающих функции: $\lambda x.\lambda y.x+y \stackrel{def}{=} \lambda x\ y.x+y$. Ну и понимать это надо как $\lambda x.\lambda y.x+y \equiv \lambda x.(\lambda y.x+y)$, то есть функцию, действующую, например, из множества вещественных чисел в множество функций, действующих из множества вещественных чисел в... ну вы поняли. Формально тип этой функции $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$, что совпадает с обычной нотацией из матанализа и — нотацией для типов функций из F#.

Каррирование делает естественным частичное применение функций. Например, $(\lambda x.\lambda y.x + y)\ 5 \equiv \lambda x.(x + 5)$ — функция, которой «скармливают» один аргумент и получается функция, готовая принять второй.

3. Лямбда-исчисление как формальная система

Пока что мы говорили только о нотации, причём в очень неформальном ключе. Но настоящая мощь лямбда-исчисления в том, что оно представляет собой удобную формальную систему, выводящуюся из набора простых аксиом, как теория множеств. Причём, достаточно мощную формальную систему, чтобы в ней выражались и математические вычисления, и алгоритмы. Поэтому давайте попробуем построить теорию нетипизированного лямбда-исчисления с нуля, и построим в этой теории что-нибудь программистское, например, списки.

Начнём с самого начала, с синтаксиса. Если в теории множеств есть множества и элементы множеств, то в лямбда-исчислении есть понятие «λ-терм». Всё в лямбда-исчислении — это λ-термы. Так что все штуки, которые показывались выше, на самом деле строятся через лямбда-термы, так что функцию вида $(\lambda x.x^2)$ мы сможем построить только ближе к концу пары. Потому что нам потребуется определить что такое степень и даже что такое 2.

Процесс вычисления в лямбда-исчислении тоже строится на λ-термах, как процесс их формального преобразования. Функция — это λ-терм. Данные — это тоже λ-терм, причём никак не отличающийся от функций. Вообще, в нетипизированном исчислении понятие «функция» и «данные» не вводятся и не разделяются никак (опять-таки, как в F#, функции можно передавать как параметры). Причём, в общем-то, λ-термы сделаны так, чтобы быть похожими на функции, так что интуитивно в лямбда-исчислении вообще всё является функцией, даже параметры функций, и никаких «данных» нет вовсе.

Семантика вычислений формально вводится как набор аксиом, говорящих, как преобразовывать термы. Результат вычисления λ-терма — это λ-терм, то есть тот факт, что $(\lambda x.x + 1)3$ равно 4, на самом деле заключается в сведении терма $(\lambda x.x + 1)3$ к терму, обозначающему число 4. Так построенной семантике λ-исчисления не требуется никаких дополнительных формализмов, чтобы определять понятие «вычисление», это называется *операционной семантикой* (или ещё можно сказать, что *аксиоматической семантикой*, раз вводится в виде набора аксиом). В противоположность *денотационной семантике*, которая строится сведением семантики формализма к семантике другого формализма, которая уже известна. Например, семантика вычислений в F# сводится к семантике вычислений .NET-машины, то есть она определена в денотационном стиле. На самом деле, это не как чёрное и белое, семантики могут описываться в более денотационном или боле операционном стиле, бывают всякие экзотические способы определения семантик, но это всё к делу не относится и по идее про это будет много на 4-м курсе.

3.1. Лямбда-термы

Итак, лямбда-терм — это:

- переменная[2]: $v \in V$, где V — некоторое множество, называемое множеством переменных;
- аппликация: если A и B — λ-термы, то $A\ B$ — λ-терм;
- λ-абстракция: если A — λ-терм, а v — переменная, то $\lambda v.A$ — λ-терм.

То есть лямбда-терм, как дерево, вводится рекурсивно, и по сути он и есть дерево. Как записываются лямбда-термы этих трёх видов, мы уже видели, плюс к тому вводятся соглашения об ассоциативности:

- аппликация левоассоциативна: если F, X, Y — λ-термы, то $F\ X\ Y$ читается как $(F\ X)\ Y$;
- λ-абстракция правоассоциативна: если x и y — это переменные, а M — λ-терм, то $\lambda x\ y.M$ читается как $\lambda x.(\lambda y.M)$;
- как мы уже договаривались, λ-абстракция распространяется вправо настолько, насколько возможно: $\lambda x.M\ N = (\lambda x.M\ N)$.

3.2. Свободные и связанные переменные

Положим, $T[x]$ — это лямбда-терм, куда входит переменная x. Тогда говорят, что λ-абстракция $\lambda x.T[x]$ **связывает** переменную x в терме $T[x]$ (то есть x суть параметр «функции»). А вот если значение выражения зависит от значения переменной, то говорят, что переменная **свободно** входит в выражение. Пример, из матанализа:

$$\sum_{m=1}^{n} m = \frac{n(n+1)}{2}$$

Здесь n входит свободно, а m связана. Связанные переменные интересны тем, что их имена можно менять, не меняя при этом интуитивного смысла терма. Однако нельзя менять имя связанной переменной так, чтобы оно начинало совпадать со свободной:

$$\int_0^x 2y + a\ dy = x^2 + ax \longrightarrow \int_0^x 2z + a\ dz = x^2 + ax$$

но

$$\int_0^x 2a + a\ da \neq x^2 + ax$$

Теперь, когда стала понятна интуиция за свободными и связанными переменными (мы её сами строим так, чтобы она была похожа на привычную из матанализа), можно ввести понятие свободной и связанной переменной формально. Как и большинство определений

[2] Иногда в аксиоматике вводят помимо переменных ещё константы, но хорошо и без них

в лямбда-исчислении, свободные переменные определяются рекурсивно по структуре терма. Множество свободных переменных от переменной — это сама переменная, множество свободных переменных от аппликации — объединение множеств свободных переменных кого апплицируют и кто апплицируется (тут может быть небольшой вывих мозга, мы привыкли к функции, которая принимает параметр, тут и то и другое — терм, который может иметь внутреннюю структуру). Множество свободных переменных (FV — Free Variables) от лямбда-абстракции — множество свободных переменных того, что внутри, минус переменная, связываемая лямбдой:

- $FV(x) = x$
- $FV(S\ T) = FV(S) \cup FV(T)$
- $FV(\lambda x.S) = FV(S) \setminus \{x\}$

Со связанными переменными дела обстоят похожим образом:

- $BV(x) = \emptyset$ — переменная сама по себе всегда свободна (BV — Bound Variables);
- $BV(S\ T) = BV(S) \cup BV(T)$ — прямо как для $FV(S\ T)$, рекурсивно по структуре аппликации;
- $BV(\lambda x.S) = BV(S) \cup \{x\}$ — тут противоположно FV, лямбда добавляет переменную в множество связанных.

Например, в терме

$$S = (\lambda x\ y.x)(\lambda x.z\ x)$$

$$FV(S) = \{z\}, BV(S) = \{x, y\}$$

Обратите внимание, что переменная может входить в терм и как связанная, и как свободная, например, в

$$S = (\lambda x\ z.x)(\lambda x.z\ x)$$

переменная z во вторых скобках никем не связана и входит в S свободно, а вот в первых она связана, хоть и не встречается справа от точки (ну а что, у функций бывают неиспользуемые параметры). Прямая аналогия свободных и связанных переменных в F# — let-определения и параметры функции (что на самом деле суть одно и тоже), и переменные, попавшие в замыкание функции (они в функцию входят свободно).

3.3. Подстановка

На самом деле, понятия свободной и связанной переменной нам нужны для того, чтобы формально ввести *подстановку*. Мы хотим в конечном итоге определить что-то, максимально похожее на вычисление функции (ну, весь смысл лямбда-исчисления для нас в том, чтобы с его помощью можно было считать, поэтому мы его проектируем так, чтобы

оно было похоже на вычисление значений выражений). Самый простой способ формализовать передачу параметра в функцию — это что-то вроде макроподстановки, когда внутри функции все имена параметра заменяются на значение, которое вместо этого параметра передали (которое само может быть сложным выражением, но почему нет, в Haskell оно вообще так и работает).

Более формально, подстановка терма S вместо переменной x в терм T — это терм, полученный заменой всех *свободных* вхождений переменной x в терме T на терм S (связанные переменные мы просто пропускаем — это типа сокрытия имён из внешнего контекста в языках программирования). Записывается подстановка как $T[x := S]$. Самый простой пример: $x[x := T] = T$ — если мы вместо переменной x подставим терм T в терм, который как раз просто переменная x, получим терм T (то есть просто заменим одно на другое). Более продвинутый пример:

$$(\lambda y.x + y)[x := \lambda z.z] = \lambda y.(\lambda z.z) + y$$

Однако, есть одна маленькая проблема, при подстановке не должно происходить связывания имён, иначе изменится «смысл» терма (мы ещё не ввели ничего похожего на смысл терма, но интуитивно хотим, чтобы арифметические выражения не портились):

$$(\lambda y.x + y)[x := y] = \lambda y.y + y$$

Эта проблема создаёт некоторую головную боль при попытке формализовать подстановку аксиоматически (мы не можем писать «не должно быть связывания», потому что а что делать, если оно таки произойдёт?). Есть аж несколько известных решений этой проблемы:

- Соглашение Барендрегта: вообще запретить связанным переменным иметь одинаковые имена и называться так же, как свободные (то есть их имена в принципе выбираются из разных множеств). Однако при подстановке может произойти раскопирование терма, что портит инвариант, поэтому надо договориться после подстановки выполнять переименование связанных переменных (мы же можем это делать безболезненно), чтобы восстановить инвариант.

- Индексы де Брауна: вообще не именовать связанные переменные, а нумеровать их числами i, где i означает «переменная, связанная i-й охватывающей лямбдой» (если мы вспомним, что функций с несколькими переменными не бывает и лямбда-абстракция связывает только одну переменную всегда, то станет понятно, что такое представление непротиворечиво).

- Переименовывать связанные переменные «на лету» перед выполнением подстановки. Путь, очень похожий на рефакторинг при выделении функции, и вообще, довольно механистичный, так что мы будем делать именно так.

Итак, формально, подстановка определяется рекурсивно по структуре терма:

- $x[x := T] = T$ — подстановка в переменную просто заменяет переменную на терм;

- $y[x := T] = y$ — подстановка в *не ту* переменную не работает (выражения не меняет);

- $(S_1\ S_2)[x := T] = S_1[x := T]\ S_2[x := T]$ — подстановка в аппликацию просто рекурсивна;
- $(\lambda x.S)[x := z] = \lambda x.S$ — подстановка в связанную переменную терма не меняет (мы говорили, что связанные переменные игнорируются при подстановке);
- $(\lambda y.S)[x := T] = \lambda y.(S[x := T])$, если $y \notin FV(T)$ или $x \notin FV(S)$ — если конфликта имён нет, выполняем подстановку в терм под лямбдой (обратите внимание, если x не входит в S свободно, можно подставлять вместо неё что угодно, всё равно ничего не произойдёт);
- $(\lambda y.S)[x := T] = \lambda z.(S[y := z][x := T])$, иначе ($z$ при этом выбирается так, что $z \notin FV(S) \cup FV(T)$ — если конфликт имён есть, сначала переименовываем связанную переменную так, чтобы конфликта не было, затем выполняем подстановку как в пункте выше.

3.4. Преобразования

Зачем мы вводили подстановку — чтобы определить тот самый «смысл» терма и ввести понятие «посчитать терм». Для этого нам потребуется отношение *равенства* над термами (мы сконструируем это понятие так, чтобы термы, которые «по смыслу» одинаковые, были равны). И отношение *редукции*, которое мы будем конструировать как «термы имеют одинаковое значение», хотя могут сильно структурно отличаться. Причём, пока что в формальной системе мы даже «значение» ещё не определили, так что мы как раз и будем его определять так, чтобы оно было похоже на вычисление функции. Поэтому мы введём понятия *преобразований*, которые являются просто формальными преобразованиями над термами, и будут элементарными шагами нашего «вычисления». Преобразования по смыслу не должны менять значение терма.

α-преобразование : $\lambda x.S \to_\alpha \lambda y.S[x := y]$ при условии, что $y \notin FV(S)$. Даёт возможность переименовывать связанные переменные.

β-преобразование : $(\lambda x.S)T \to_\beta S[x := T]$. Собственно, определяет процесс вычисления. Один шаг вычисления — это когда мы находим аппликацию лямбда-абстракции к чему-то и выполянем подстановку. Ну то есть интуитивно вычисляем значение функции, передавая ей данное значение. Вот зачем нам на самом деле понятие «подстановка».

η-преобразование : $\lambda x.T\ x \to_\eta T$, если $x \notin FV(T)$. Обеспечивает **экстенсиональность** — две функции экстенсионально эквивалентны, если на всех одинаковых входных данных дают одинаковый результат:

$$\forall x\ F\ x = G\ x$$

Наверное, если мы хотим адекватное понятие вычисления, нам не помешает такое свойство. Формально это позволяет «снимать лишнюю лямбду», а в F# это преобразование позволяет использовать переменные функциональных типов (`let f x = printfn "%d" x` ⇔ `let f = printfn "%d"`).

Теперь можно ввести аксиомы равенства лямбда-термов:

$$\frac{S \to_\alpha T \ \text{ или } \ S \to_\beta T \ \text{ или } \ S \to_\eta T}{S = T}$$

— два терма равны, если один получается из другого.

$$\frac{}{T = T}$$

— рефлексивность, терм равен сам себе.

$$\frac{S = T}{T = S}$$

— симметричность, если один терм равен другому, то и другой первому.

$$\frac{S = T \wedge T = U}{S = U}$$

— транзитивность.

$$\frac{S = T}{S\,U = T\,U}$$

— рекурсивное по структуре терма равенство для аппликации номер один — если две функции равны и мы применяем их к одному и тому же аргументу, результаты будут одинаковы.

$$\frac{S = T}{U\,S = U\,T}$$

— рекурсивное по структуре терма равенство для аппликации номер два — если одну и ту же функцию применить к равным аргументам, результаты будут равны

$$\frac{S = T}{\lambda x.S = \lambda x.T}$$

— рекурсивное по структуре терма равенство для лямбда-абстракции — если тела функций равны, то и функции равны.

Получилось отношение эквивалентности, что в принципе отвечает нашим ожиданиям от понятия «равенство». Для определения понятия «вычисление» нам хватит более слабого отношения, которое не обязательно симметрично (то есть более не отношение эквивалентности), это будет отношение *бета-редуцирования*, с вот такими аксиомами (они повторяют аксиомы равенства, но без симметричности):

$$\frac{S \to_\alpha T \ \text{ или } \ S \to_\beta T \ \text{ или } \ S \to_\eta T}{S \to_\beta T}$$

$$\frac{}{T \to_\beta T}$$

$$\frac{S \to_\beta T \wedge T \to_\beta U}{S \to_\beta U}$$

$$\frac{S \to_\beta T}{S\,U \to_\beta T\,U}$$

$$\frac{S \to_\beta T}{U\ S \to_\beta U\ T}$$

$$\frac{S \to_\beta T}{\lambda x.S \to_\beta \lambda x.T}$$

Бета-редукция внезапно хороша тем, что необратима, в отличие от равенства. Она позволяет применять преобразования, не беспокоясь, что существует цепочка преобразований, возвращающая всё как было (что требует симметричность), это позволяет терять информацию о структуре терма и тем самым его упрощать. То есть вычисление — это не последовательность термов, которые равны, это последовательность термов, которые получены с помощью бета-редукции друг из друга. Однако внезапно бета-редукция не всегда терм упрощает. Знаменитый пример:

$$(\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x) \to_\beta$$

$$(\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x) \to_\beta$$

$$(\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x) \to_\beta \ldots$$

Напомним, что подстановка выполняется текстуально, то есть $(\lambda x.x\ x\ x)$, когда подставляется в $(\lambda x.x\ x\ x)$, порождает три копии себя: λx убирается (подстановка убирает лямбду), а вместо *каждого* x появляется $(\lambda x.x\ x\ x)$. Следующая подстановка заменяет первые два терма с лямбдой на три копии следующего за ним (обратите внимание, выбрать для подстановки второй и третий нельзя, в виде дерева этот терм всё-таки представляется как $((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x))\ (\lambda x.x\ x\ x)$).

Обратите внимание, при этом термы не равны с точки зрения равенства, симметричности нет. А вот аксиомы бета-редукции выполняются.

И, кстати, ещё один забавный факт — бета-редукцию можно делать по-разному, заменяя разные аппликации лямбда-абстракций, и получить, вообще говоря, разные результаты:

$$(\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)) \to_\beta$$

$$(\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)) \to_\beta$$

$$(\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)) \to_\beta \ldots$$

— тут мы выбираем для подстановки страшную штуку внутри скобок и, как выше, можем продолжать до бесконечности. И, в отличии от ситуации выше, у нас тут правда есть право выбора, бета-преобразование применимо к двум термам тут и аксиомы не говорят, какой из термов выбрать. Так что если выбрать для подстановки внешний терм (который $\lambda x.y$) и подставить в него то, что справа от него, всё закончится очень быстро и решительно:

$$(\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)) \to_\beta y$$

Ну, терм $((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x))$ подставляется вместо переменной x в терм y, но в терме y нет ни одного вхождения переменной x (потому что это вообще одна переменная), так что большой страшный бесконечно редуцируемый терм тут же сгорает.

3.5. Стратегии редукции

Рассуждения выше приводят нас к двум мыслям:

1. в зависимости от стратегии выбора терма, который мы бета-редуцируем, вычисление может занимать больше или меньше времени, или даже при неудачном выборе терма не закончиться вообще;

2. получилась ерунда какая-то, потому что в зависимости от выбора терма для редукции у нас могут получиться разные ответы — мы хотим формализовать математику, а не «главное не как проголосовали, а как посчитали»[3].

Для того, чтобы с этим разобраться, формализуем понятие «стратегия выбора терма». Для этого введём понятие «редэкс» (Reducible Expression). **Редэксом** называется пара термов, в которой можно выполнить подстановку. То есть выражение вида $(\lambda x.S)T$. Редэкс — это минимальный терм, к которому можно применить бета-редукцию, и получить терм с подставленным значением переменной: $(\lambda x.S)T \to_\beta S[x := T]$. Например,

$$(\lambda f.\lambda x.f\ x\ x)\mathbf{+} \to_\beta \lambda x.\mathbf{+}\ x\ x$$

Терм без редэксов называется термом в **нормальной форме**. Это терм, который нельзя дальше упростить — он и будет результатом «вычисления» в лямбда-исчислении.

Собственно, **стратегия редукции** говорит, какой редэкс в терме заменять на каждом шагу. В принципе, никто не ограничивает вас в выборе, так что стратегий редукции можно придумать миллионы (например, выбирать для замены третий слева редэкс, если сегодня пятница, и второй справа, если нет). Есть две широкоизвестные:

аппликативная стратегия — заменяем самый левый редэкс, не содержащий в себе других редэксов (то есть самое маленькое подвыражение);

нормальная стратегия — заменяем самый левый самый внешний редэкс.

Аппликативная стратегия — это передача параметра по значению в языках программирования, когда мы сначала вычисляем параметр, потом передаём его в функцию. Нормальная стратегия соответствует передаче параметра по имени (или ленивому вычислению), когда мы откладываем вычисление параметра до последнего, в надежде, что он нам не понадобится.

Вспомним про терм

$$(\lambda x.y)\ ((\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x))$$

Аппликативная стратегия требует от нас посчитать сначала параметр, то есть раскрытия редэкса $(\lambda x.x\ x\ x)\ (\lambda x.x\ x\ x)$ внутри скобок. Что, как мы знаем, не приведёт к успеху. Нормальная стратегия говорит сначала вызвать функцию, то есть выполнить подстановку во внешнюю лямбду. Что, как мы знаем, приводит к успеху за один шаг. Итак, кажется, нормальная стратегия лучше. Угадайте, какая стратегия применяется в современных языках программирования чаще.

На самом деле, нормальная стратегия правда лучше:

[3] как говорил Наполеон III

Теорема 1 (Карри о нормализации)**.** *Если у терма есть нормальная форма, то последовательное сокращение самого левого внешнего редекса приводит к ней.*

Если успех возможен, нормальная стратегия редукции всегда к нему приведёт, остальные — как получится. Но что, если нормальная стратегия даёт в ответе один терм, а какая-нибудь другая — другой? На это тоже есть теорема:

Теорема 2 (Чёрча-Россера)**.** *Если терм M β-редукцией редуцируется к термам N и K, то существует терм L такой, что к нему редуцируются и N, и K.*

В общем, нормальная форма не всегда есть (кака мы видели), но если она есть, то её можно получить нормальной стратегией, причём нормальная форма единственная. То есть мы не только развеяли страхи, что у нас вычисление зависит от того, как считали, но ещё и получили «лучший» способ считать. Обратите внимание, тут теоремы приводятся без доказательства, но вообще это формальные теоремы над лямбда-исчислением, для доказательства которых ничего, кроме аксиоматики лямбда-исчисления, не требуется, то есть мы уже получили какие-то строгие знания о языках программирования, только-только начав строить теорию (что Haskell хороший, а все остальные плохие в плане шансов окончить вычисление).

4. Лямбда-исчисление как язык программирования

Мы установили, что с помощью λ-исчисления можно в каком-то смысле считать. Причём, в отличие от матанализа, у нас есть понятие «шаг вычисления», мы можем судить о конечности и бесконечности вычислений, мы можем формально оценивать количество шагов, потребных для вычисления, в зависимости от размера входного терма (напомним, терм — это дерево, так что его размер — это количество узлов в нём, не длина строки, которой он записан). А теперь давайте построим (точнее. начнём строить) из λ-исчисления язык программирования, этакий мини-LISP, на котором можно будет записывать алгоритмы и математически доказывать разные утверждения про них.

4.1. Комбинаторы

Для этого нам потребуется ещё немного теории. Введём понятие «комбинатор». Формально **комбинатор** — это λ-терм без свободных переменных. Неформально, комбинатор, как правило, это функция, которая позволяет комбинировать другие функции, без упоминания данных.

Есть много известных примеров:

$$\mathbf{I} \equiv \lambda x.x$$

— тождественная функция.

$$\omega \equiv \lambda s.s\ s$$

— комбинатор самоприменимости. Он скармливает переданный ему терм сам себе (то есть функцию вызывает от неё же в качестве аргумента).

$$\Omega \equiv \omega\ \omega \equiv (\lambda s.s\ s)(\lambda s.s\ s)$$

— расходящийся комбинатор. Кому-то пришло в голову, что можно применить комбинатор самоприменимости к комбинатору самоприменимости. Получился лямбда-терм, не имеющий нормальной формы (который содержит один редэкс и буквально не меняется при бета-преобразовании, так что считать его можно долго, но толку от этого не будет).

$$K \equiv \lambda x\ y.x$$

— канцеллятор, он «отменяет» свой второй аргумент, оставляя только первый. Нам он понадобится потом, когда будем строить пары (интуитивно, канцеллятор — это функция fst из F#, оставляет от пары только первый элемент). Через канцеллятор же выражаются булевые константы, так что это важная и нужная вещь.

$$K_* \equiv \lambda x\ y.y$$

— второй элемент пары (аналог функции snd из F#).

$$S \equiv \lambda x\ y\ z.x\ z\ (y\ z)$$

— коннектор. z тут играет роль аргумента, который сначала передаётся во вторую функцию, а потом он же и результат второй функции передаётся в первую (что-то вроде fold).

$$B \equiv \lambda f\ g\ x.f\ (g\ x)$$

— композиция (в обычном математическом смысле, композиция двух функций).

Пара примеров. Что будет, если применить тождественное преобразование к... тождественному преобразованию?

$$I\ I \equiv (\lambda x.x)(\lambda x.x) \to_\beta \lambda x.x \equiv I$$

Очевидно, ничего интересного. А что будет, если применить канцеллятор к тождественному преобразованию?

$$K\ I \equiv (\lambda x.\lambda y.x)(\lambda x.x) \to_\beta$$
$$\to_\beta \lambda y.(\lambda x.x) \to_\alpha \lambda x.\lambda y.y \equiv K_*$$

Внезапно, в терминах F#, fst id = snd. Но F# не даст вам применить fst ни к чему, кроме пары. Для этого, собственно, и придуманы типизированные лямбда-исчисления, и когда речь пойдёт про пары, мы ещё раз увидим, что типизация всё сильно упрощает. Но пока типизации у нас нет, можно извращаться как угодно. Причём, большинство этих штук вполне будет работать и в F# — канцеллятор можно объявить как генерик-функцию и прямо проверить *экстенсиональное равенство* левой и правой части:

```
let K x y = x
let K_star x y = y
Check.Quick (fun a1 a2 -> (K id) a1 a2 = K_star a1 a2)
```

4.2. Теорема о неподвижной точке

Можно начинать построение лямбда-исчисления с комбинаторов на самом деле, есть даже «комбинаторная логика», которая строит вычисления из комбинаторов. Но нам комбинаторы нужны для того, чтобы сделать рекурсию. Внезапно это в лямбда-исчислении нетривиальная задача, потому что у функций нет имён. Поэтому они никак не могут на себя ссылаться. Вы можете дать имя терму и потом использовать его по имени, но внутри терма это имя, естественно, недоступно (как в F# без ключевого слова `rec`). Поэтому нам потребуется ещё одна небольшая теорема:

Теорема 3 (О неподвижной точке)**.** *Для любого λ-терма F существует неподвижная точка:*

$$\forall F \ \exists X : F\ X = X$$

Для любого комбинатора, какой бы мы ни взяли, найдётся такой терм, который этот комбинатор не меняет. Неожиданно. Но ещё более неожиданно то, что существует комбинатор, позволяющий эту неподвижную точку найти, просто подставив в него нужный терм. Причём не просто абстрактно существует, а вот он:

Теорема 4 (О комбинаторе неподвижной точки)**.** *Существует комбинатор неподвижной точки*

$$Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

такой, что

$$\forall F \quad F\ (Y\ F) = Y\ F$$

Это и есть известный Y-комбинатор, комбинатор, который, будучи применённым к любому комбинатору (то есть терму без свободных переменных), даёт его неподвижную точку. Причём, эта теорема даже легко доказывается:

Доказательство.

$$Y\ F \equiv (\lambda x.F\ (x\ x))(\lambda x.F\ (x\ x))$$
$$= F\ ((\lambda x.F\ (x\ x))(\lambda x.F\ (x\ x))) = F(Y\ F)$$

□

Тут мы по определению расписываем Y, подставляя туда вместо $f\ F$. Далее выполняем один шаг β-редукции по нормальной стратегии, получается, что самая левая лямбда снимается, F «вылезает на поверхность», а вместо его x-ов подставляются $\lambda x.F\ (x\ x)$. Тут мы замечаем, что то, что написано внутри скобок, это Y от F (в точности то, что написано справа от равенства), заменяем и получаем $F(YF)$, ч. и т.д.

Забавно, но при чём тут рекурсия? А давайте посмотрим на самую простую рекурсивную функцию, что есть на свете, факториал, и запишем её через лямбда-термы:

$$\text{factorial} = \lambda n.\text{if}\ (\text{isZero}\ n)\ 1\ (\text{mult}\ n\ (\text{factorial}\ (\text{pred}\ n)))$$

if, isZero, 1, mult и т.д. — это тоже лямбда-термы, которые мы введём, когда будем делать себе арифметику и булевую логику, пока что не важно, как они внутри устроены. Что важно, это то, что factorial встречается в правой части, где он ещё не определён, так что в древовидном виде такой лямбда-терм непредставим, то есть синтаксически некорректен.

Окей, применим η-преобразование ($\lambda x.Tx = T$) в обратную сторону, вынесем factorial в переменную, свяжем её лямбдой и тут же подставим, чтобы не изменить смысл терма:

$$\text{factorial} = (\lambda f.\lambda n.\text{if } (\text{isZero } n)\ 1\ (\text{mult } n\ (f\ (\text{pred } n)))))\ \text{factorial}$$

Очевидно, что если применить β-редукцию, получится исходный неправильный терм. И наш терм всё ещё неправильный. Но он кое-что напоминает. Давайте переименуем factorial в X, а $(\lambda f.\lambda n.\text{if } (\text{isZero } n)\ 1\ (\text{mult } n\ (f\ (\text{pred } n))))$ обозначим за F. Получим $X = FX$. То есть factorial — это такой терм, который не меняется при применении к нему терма $(\lambda f.\lambda n.\text{if } (\text{isZero } n)\ 1\ (\text{mult } n\ (f\ (\text{pred } n)))))$. А мы как раз научились такие термы искать:

$$\text{factorial} = Y(\lambda f.\lambda n.\text{if } (\text{isZero } n)\ 1\ (\text{mult } n\ (f\ (\text{pred } n)))))$$

Теперь это абсолютно синтаксически корректное выражение (которое не использует ничего необъявленного в правой части), которое описывает лямбда-терм, считающий факториал. То есть рекурсивная функция в лямбда-исчислении — суть неподвижная точка тела функции, где рекурсивный вызов — это ещё одна лямбда-абстракция внутри терма. Если поразмыслить, это даже с точки зрения программирования имеет философский смысл – при рекурсивном вызове вызывается та самая функция, что делает вызов.

Можно даже последовать примеру из курса «Системы типизации лямбда-исчисления» Дениса Москвина на https://www.lektorium.tv (очень рекомендую, можно считать эту лекцию чем-то вроде тизера), откуда позаимствован этот и следующие примеры, и посмотреть, как факториал будет работать. Мы пока не ввели числа, но представим себе, что это лямбда-термы, которые ведут себя как нормальные числа.

Напомним, что

$$F = (\lambda f.\lambda n.\text{if } (\text{isZero } n)\ 1\ (\text{mult } n\ (f\ (\text{pred } n)))))$$

Тогда

$$\begin{aligned}\text{factorial } 3 &= (Y\ F)\ 3\\ &= F\ (Y\ F)\ 3\\ &= \text{if } (\text{isZero } 3)\ 1\ (\text{mult } 3\ ((Y\ F)\ (\text{pred } 3)))\end{aligned}$$

— просто подставим $Y\ F$ в F. Представим себе, что if и isZero работают как обычно, тогда останется только третий параметр if:

$$= \text{mult } 3\ ((Y\ F)\ 2)$$

где мы можем снова раскрыть $Y\ F$ по теореме о неподвижной точке

$$= \text{mult } 3\ (F\ (Y\ F)\ 2)$$

и подставить в тело двойку:

$$= \text{mult } 3\ (\text{mult } 2\ ((Y\ F)\ 1))$$

$$= \text{mult } 3\ (\text{mult } 2\ (\text{mult } 1\ ((Y\ F)\ 0)))$$

— дошли до нуля, if делает своё дело и возвращает 1, образуя базу рекурсии:

$$= \text{mult } 3\ (\text{mult } 2\ (\text{mult } 1\ 1))$$

$$= 6$$

4.3. Булевые выражения

Итак, у нас есть рекурсия, но чтобы заработал хотя бы факториал, нам нужны числа, if и предикаты. Чтобы был if, надо ввести булевые константы. В лямбда-исчислении всё — лямбда-терм, никаких данных нет, поэтому введём true и false так:

$$\text{TRUE} \equiv \lambda x.\lambda y.x$$

$$\text{FALSE} \equiv \lambda x.\lambda y.y$$

И оператор IF:

$$\text{IF} \equiv \lambda b.\lambda t.\lambda f.b\ t\ f$$

То есть IF принимает булевое выражение b, которое вообще может быть произвольным лямбда-термом, но давайте договоримся передавать туда только TRUE или FALSE (у нас нетипизированное исчисление, поэтому формализовать мы это не можем, но если IF будет выдавать странные результаты на странных термах, то и ладно). t и f — это лямбда-термы, к которым должен редуцироваться IF, если передали TRUE и FALSE соответственно. Теперь понятно, почему TRUE и FALSE так объявлены — TRUE, будучи переданным в IF, выкинет второй аргумент и оставит первый, FALSE наоборот, оставит второй. Да, TRUE $= K$ и FALSE $= K_*$. Мы намеренно сконструировали их так, чтобы IF получился таким простым.

А теперь у нас есть IF, так что булевые операторы конструируются легко и приятно:

$$\text{AND} \equiv \lambda a\ b.\text{IF}\ a\ b\ \text{FALSE}$$

$$\text{OR} \equiv \lambda a\ b.\text{IF}\ a\ \text{TRUE}\ b$$

$$\text{NOT} \equiv \lambda b.\text{IF}\ b\ \text{FALSE}\ \text{TRUE}$$

Опять-таки, мы ожидаем, что операторам будут передавать только TRUE или FALSE. Причём, если это так, это сильно упрощает дело. Например, давайте посчитаем нормальную форму NOT, расписав его через лямбда-термы и выполнив бета-редукцию:

$$\begin{aligned}
&\text{NOT} = \lambda b.\text{IF}\ b\ \text{FALSE}\ \text{TRUE} \\
&= \lambda b.((\lambda b'.\lambda t.\lambda f.b'\ t\ f)\ b\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)) \\
&\to_\beta \lambda b.((\lambda t.\lambda f.b\ t\ f)\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)) \\
&\to_\beta \lambda b.((\lambda f.b\ (\lambda x.\lambda y.y)\ f)\ (\lambda x.\lambda y.x)) \\
&\to_\beta \lambda b.(b\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x))
\end{aligned}$$

Что на самом деле равно

$$\to_\beta \lambda b.(b\ \text{FALSE}\ \text{TRUE})$$

Но, если b может быть только TRUE или FALSE, у NOT есть гораздо более простая запись:

$$\text{NOT} = \lambda b\ t\ f.b\ f\ t$$

В равенстве этих термов на значениях TRUE, FALSE можно убедиться, просто подставив их в обе формулы. Для термов в общем виде это не так, но вторая запись естественнее — NOT «переворачивает» аргументы, подающиеся в канцелляторы, так что обращает и значения истинности.

4.4. Нумералы Чёрча

Булевые значения — это уже почти числа, но их всего два. Пришла пора сконструировать настоящие натуральные числа. Сделаем мы это, следуя Чёрчу, так:

$$0 \equiv \lambda s\ z.z$$

$$1 \equiv \lambda s\ z.s\ z$$

$$2 \equiv \lambda s\ z.s\ (s\ z)$$

$$3 \equiv \lambda s\ z.s\ (s\ (s\ z))$$

$$4 \equiv \lambda s\ z.s\ (s\ (s\ (s\ z)))$$

То есть 0 — это снова канцеллятор, K_* (в лямбда-исчислении канцелляторы повсюду). 1 — это когда мы один раз применяем некоторую функцию s к некоторому терму z, 2 — когда два раза применяем некоторую функцию s к некоторому терму z и т.д. Нетрудно догадаться, что Чёрч имел в виду, что s — это инкремент, что-то вроде $f(x) = x + 1$, а z — что-то, что имеет смысл нуля (что ломает мозг, потому что 0 по определению — это что-то, что имеет смысл нуля, к которому 0 раз применяется функция инкремента, но самая жесть впереди).

А ещё число n можно понимать как функцию, которая n раз применяет свой первый аргумент ко второму, то есть если передать числу функцию, оно вернёт функцию, работающую как n повторений переданной.

Давайте на таких числах сделаем сначала комбинатор инкремента, который мог бы взять число и вернуть терм, соответствующий числу, на один большему. Для этого нам надо «внутри терма» вызвать s ещё один раз. Легко:

$$\mathrm{S} \equiv \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$$

то есть

$$\begin{aligned}\mathrm{S}\ n &= (\lambda n\ f\ x.f\ (n\ f\ x))\ n \\ &\to_\beta \lambda f\ x.f\ (n\ f\ x) \\ &= n + 1\end{aligned}$$

За этим стоит на самом деле гениальная и совершенно непонятная интуиция: число есть функция, и число принимает в качестве параметров функции, которые позволяют манипулировать числом. Грубо говоря, число содержит слоты, куда можно передать операции. Например, x в S — это слот под z, терм, который «типа ноль», отправная точка при построении нумерала. f — это слот под s из определения числа, типа операции инкремента. И комбинатор S построен так, что он вызывает «инкремент» от числа, которому скармливают в качестве инкремента этот же «инкремент» и в качестве «нуля» тот «ноль»,

что передали S. Причём, ломает мозг то, что нумерал — это не конструктор числа, а число: неверно, что мы можем получить число, подставив вместо «инкремента» какую-то функцию, а вместо «нуля» ноль — нумерал и есть число.

Теперь, когда у нас есть инкремент, мы можем определить операцию сложения:

$$\text{ADD} \equiv \lambda m\ n.(m\ \text{S})\ n$$

Здесь тоже используется безумная идея, что число имеет слоты для операций. $(m\ S)$ — это мы вместо функции «инкремента» числу передаём комбинатор S, получая функцию, которая m раз применяет S к своему аргументу. А в качестве «нуля» или, тут более правильно, «отправной точки» мы передаём числу другое число — n. То есть к n m раз применяется операция «увеличить на единицу». Разумно.

А можно переписать оператор сложения в виде лямбда-терма:

$$\text{ADD} \equiv \lambda m\ n.\lambda f\ x.(m\ f)\ (n\ f\ x))$$

f и x, как обычно, параметры чисел, инкремент и ноль. Чтобы сложить два числа, надо сделать функцию, m раз применяющую f, затем применить её к числу n (которому переданы его обычные параметры с f в качестве «инкремента»). Обошлись и без комбинатора S.

Теперь понято, как ввести умножение:

$$\text{MUL} \equiv \lambda m\ n.m\ (\text{ADD}\ n)\ 0$$

— строим функцию, которая m раз делает ADD n, и начнём с нуля.

И степень:

$$\text{EXP} \equiv \lambda m\ n.m\ (\text{MUL}\ n)\ 1$$

Ну и ещё проверка на 0:

$$\text{ISZRO} \equiv \lambda n.n\ (\lambda x.\text{FALSE})\ \text{TRUE}$$

— число, которое n раз применяет функцию, возвращающую FALSE к константе TRUE. Если это n happens to be 0, TRUE останется нетронутой, иначе получим FALSE.

4.5. Пары

Сложение и умножение — это просто, а вот вычитание оказывается гораздо менее тривиальным. Мы должны отменить применение «инкремента», что невозможно сделать, просто дописав ещё какой-нибудь терм к числу. Декремент в арифметике нумералов Чёрча — это адская конструкция, которая выполняет инкремент начиная с нуля, до тех пор, пока **следующее** за текущим числом не окажется тем, от которого мы делаем декремент. А чтобы понять, когда нам надо остановиться, мы должны заглянуть в будущее — а точнее, где-то хранить текущий и предыдущий шаг вычисления. Если текущий шаг даёт наше исходное число, предыдущий шаг будет ответом декремента. А для этого нам понадобится первая настоящая структура данных в нашем супер-языке: пары.

Пара вводится очень легко:

$$\text{PAIR} \equiv \lambda x\ y\ f.f\ x\ y$$

Идея построения тут такая же, как у чисел и у булевых констант: давайте сделаем слот для операции и два слота для данных. Если в качестве операции передать канцеллятор K, получим первый слот данных, если K_* — второй. Вообще идея превращения значения в аппликацию некоторой функции, чтобы потом вместо этой функции можно было подставлять действия, используется в лямбда-исчислении повсеместно.

Проекции вводятся тривиально:

$$\text{FST} \equiv \lambda p.p\ K$$

$$\text{SND} \equiv \lambda p.p\ K_*$$

Или, раз у нас есть булевые константы,

$$\text{FST} \equiv \lambda p.p\ \text{TRUE}$$

$$\text{SND} \equiv \lambda p.p\ \text{FALSE}$$

Можно в этом даже убедиться, выполнив β-редукцию:

$$\begin{aligned}\text{FST}\ (\text{PAIR}\ a\ b) &= \text{PAIR}\ a\ b\ \text{TRUE}\\ &\equiv (\lambda x\ y\ f.f\ x\ y)\ a\ b\ \text{TRUE}\\ &= \text{TRUE}\ a\ b\\ &= (\lambda x.\lambda y.x)\ a\ b\\ &= a\end{aligned}$$

И SND:

$$\begin{aligned}\text{SND}\ (\text{PAIR}\ a\ b) &= \text{PAIR}\ a\ b\ \text{FALSE}\\ &\equiv (\lambda x\ y\ f.f\ x\ y)\ a\ b\ \text{FALSE}\\ &= \text{FALSE}\ a\ b\\ &= (\lambda x.\lambda y.y)\ a\ b\\ &= b\end{aligned}$$

4.6. Декремент

И теперь, когда у нас есть пары, наконец, можно ввести декремент. Но для этого нам потребуются две вспомогательные функции:

$$\text{ZP} \equiv \text{PAIR}\ 0\ 0$$

— пара из двух нулей.

$$\text{SP} \equiv \lambda p.\text{PAIR}\ (\text{SND}\ p)\ (\text{SUCC}\ (\text{SND}\ p))$$

— «парный инкремент». Принимает пару, возвращает пару, первый элемент которой равен второму переданной пары, а второй — второму плюс 1 (короче, `let sp p = (snd p, (snd p) + 1)`). Обратите внимание, что SP забывает первый аргумент, но мы-то помним, зачем нам пары — мы хотим определить декремент так, чтобы идти с нуля и остановиться, когда SND станет равным нашему числу. У нас тут пара — это скользящее окно, которое постепенно ползёт к целевому значению.

Ну и сама функция декремента:

$$\text{PRED} \equiv \lambda m.\text{FST}\ (m\ \text{SP}\ \text{ZP})$$

— мы m раз применяем «парный инкремент» к нулевой паре, получаем пару $(m-1, m)$, и берём её первый элемент. Эффективно реализуется на компьютере? Не думаю. Работает? Да. Если вспомнить, что интеграл Римана предлагает складывать бесконечное количество бесконечно малых кусков подграфика, то у нас всё равно лучше.

4.7. Примитивная рекурсия

Что-то кажется, что такой же приём, как мы использовали в PRED, можно использовать и для более интересных вычислений, типа факториала — мы храним текущее значение и счётчик, и идём, пока счётчик не будет равен чему нужно, умножая значение на счётчик. Надо только поменять SP и ZP, потому что иначе мы будем на ноль умножать.

Давайте обобщим ZP до функции, которая будет инициализировать пару первым значением:

$$\text{XZ} \equiv \lambda x.\text{PAIR}\ x\ 0$$

И обобщим SP до функции, которая будет произвольную функцию применять к первому и второму элементу пары, писать то, что получилось, в первый элемент, а второй просто увеличивать на единицу:

$$\text{FS} \equiv \lambda f\ p.\text{PAIR}\ (f\ (\text{FST}\ p)\ (\text{SND}\ p))\ (\text{SUCC}\ (\text{SND}\ p))$$

То есть, второй элемент пары — это счётчик, который будет на каждом шаге применения FS расти на 1, а первый — «полезная нагрузка», которую мы хотим посчитать в зависимости от предыдущего значения и счётчика (что-то типа Seq.unfold).

И тогда можно сделать *комбинатор примитивной рекурсии*, который будет m раз применять генерилку значений (которая передаётся как параметр FS) к начальному состоянию (которое инициализируется XZ):

$$\text{REC} \equiv \lambda m\ f\ x.\text{FST}\ (m\ (\text{FS}\ f)\ (\text{XZ}\ x))$$

То есть, REC работает как-то так:

$$(x, 0) \to$$
$$(f\ x\ 0, 1) \to$$
$$(f\ (f\ x\ 0)1, 2) \to$$
$$(f\ (f\ (f\ x\ 0)\ 1)\ 2, 3) \to ...$$

И тогда PRED можно выразить через REC довольно очевидным образом. XZ передаём начальное состояние, то есть 0, FS передаём функцию, которая по переданному ей состоянию и значению счётчика просто возвращает... значение счётчика, состояние выкидывая:

$$\text{PRED} = \lambda m.\text{REC}\ m\ (\lambda x\ y.y)\ 0$$

Вспомним, что FS определена так, чтобы вторым аргументом был SUCC (SND P), а в f передаётся сам SND P, так что у нас в состоянии вычисления всегда будет $(n, n + 1)$.

Примитивно-рекурсивные функции интересны с точки зрения теории алгоритмов. Они соответствуют структурным программам, которые используют только обычные операторы, if и for с целочисленным счётчиком. Примитивно-рекурсивные функции – «очень хороший» класс функций, они определены на всём множестве входных данных и всегда завершаются за конечное время (что отличает их от *частично-рекурсивных функций*, которые могут не завершаться на некоторых значениях). Про это, может, будут рассказывать на матлогике в конце курса, или в более специализированных элективах по теоретической информатике, но то, что мы за пару с нуля смогли построить теорию, в которой выражается примитивная рекурсия — это само по себе доказывает, что всё просто.

4.8. Списки

Приведём для примера ещё то, как в лямбда-исчислении можно сконструировать списки. Причём, списки в функциональном стиле, как в F#:

```
type 'a list =
    | ([])
    | (::) of 'a * 'a list
```

Идея точно такая же, как была выше — определим термы-«данные» со слотами для операций. Пустой список введём так:

$$\text{NIL} \equiv \lambda c\ n.n$$

Так же, как в нумералах Чёрча, c — это слот под «cons», n — слот под «null». И это снова один из канцелляторов!

Комбинатор CONS более хитро описывается:

$$\text{CONS} \equiv \lambda e\ l\ c\ n.c\ e\ (l\ c\ n)$$

e — это типа данные, которые лежат в ячейке, l — хвост списка, c и n — параметры NIL (cons и null).

Тогда:

$$[] \equiv \text{NIL} \equiv \lambda c\, n.n$$

$$[5;3;2] \equiv \text{CONS } 5\ (\text{CONS } 3\ (\text{CONS } 2\ \text{NIL})) \equiv \lambda c\, n.c\ 5\ (c\ 3\ (c\ 2\ n))$$

Ну и определили мы структуры данных так потому, что так проще реализовать стандартные функции:

$$\text{EMPTY} \equiv \lambda l.l\ (\lambda h\ t.\text{FALSE})\ \text{TRUE}$$

— так же как проверка на 0 в нумералах Чёрча, отдаёт списку константную функцию, возвращающую FALSE, но если список пуст, она ни разу не вызовется и вся эта конструкция вернёт TRUE.

$$\text{HEAD} \equiv \lambda l.l\ (\lambda h\ t.h)\ 0$$

— возвращает 0, если список пуст (в лямбда-исчислении нет исключений), иначе значение в голове списка.

Например,

$$\text{HEAD } (\text{CONS } 2\ \text{NIL}) \equiv \lambda l.l\ (\lambda h\ t.h)\ 0\ (\text{CONS } 2\ \text{NIL}) \to_\beta$$

$$(\text{CONS } 2\ \text{NIL})\ (\lambda h\ t.h)\ 0 \equiv$$

$$((\lambda e\ l\ c\ n.c\ e\ (l\ c\ n))\ 2\ \text{NIL})\ (\lambda h\ t.h)\ 0 \to_\beta$$

$$(\lambda c\ n.c\ 2\ (\text{NIL}\ c\ n))\ (\lambda h\ t.h)\ 0 \to_\beta$$

$$(\lambda c\ n.c\ 2\ n)\ (\lambda h\ t.h)\ 0 \to_\beta$$

$$(\lambda h\ t.h)\ 2\ 0 \to_\beta$$

$$2$$