12+

СЕРГЕЙ КИРНИЦКИЙ

Программирование как мышление

КАК ЯЗЫКИ ПОЛВЕКА ИСКАЛИ ОДНУ ФОРМУ

Сергей Кирницкий

Программирование как мышление. Как языки полвека искали одну форму

https://litres.ru/74257147
ISBN 9785007057622

Аннотация

Программирование — не множество дисциплин, а одна. Полвека десятки инженеров переписывали один язык, убирая ошибки предшественников, — и все пришли к одним ответам. Каждый язык уникален как культурный артефакт, но под этим слоем все они — итерации одного языка. Синтаксис — оболочка; идеи — суть. Эта книга показывает, как из хаоса школ проступила линия, как линия сошлась в точку и как AI обнажил: код был лишь записью мысли.

Содержание

Программирование как мышление
Как языки полвека искали одну форму

Сергей Кирницкий

Дизайнер обложки Created with Grok

ISBN 978-5-0070-5762-2
Создано в интеллектуальной издательской системе Ridero

Часть I. Одно дерево

Одна линия с фигурными скобками

Разработчик, проживший в профессии несколько лет, носит в голове небольшой музей языков. C — суровый и честный. Java — обстоятельная, тяжёлая на подъём. Python — приветливый. Go — нарочито скромный. Rust — требовательный и строгий. Каждый кажется отдельным миром со своими нравами, своей мифологией, своим племенем. И первое, чему учат новичка, — выбрать, к какому племени примкнуть.

Но стоит положить рядом четыре коротких фрагмента, написанных в разные десятилетия на разных языках, как сквозь различие проступает странное единообразие. Те же фигурные скобки очерчивают тело функции. То же слово return *отдаёт результат наружу. То же* if *разводит исполнение на две ветви. Меняются имена, меняется огласовка — грамматика остаётся прежней. Так в речи незнакомца вдруг узнаёшь падежи родного языка, и разноголосица наречий оказывается делом поверхности.*

Узнанное родство — это улика. Улика общего предка, от которого все эти языки унаследовали скелет. И если предок один, то их создатели — не основатели независимых религий, а наследники: каждый получил готовую грамматику и распорядился ею по-своему. За каждым их решением стоит

не техническая деталь, а убеждение — что важнее: свобода или безопасность, контроль или абстракция, мощь или простота. Создатель языка — мыслитель со своей верой и своим слепым пятном, а язык — отпечаток этой веры в синтаксисе.

Перед нами генеалогическое древо. Пока мы стоим под ним и, задрав голову, разглядываем ветви — как они расходятся, спорят друг с другом, тянутся в разные стороны. Снизу кажется, что ветвей много и растут они врозь. Чтобы увидеть ствол, придётся спуститься к корню.

Глава 1. Точка ноль

ALGOL, Ритчи и рождение синтаксиса (1958—1972)

Спросите любого, откуда взялись фигурные скобки, if, for, struct, *— и услышите: «Это синтаксис C». Ответ звучит так естественно, что в нём не замечают арифметической неувязки. C появился в 1972 году. А привычка заключать блок кода в скобки, давать переменной тип, собирать разнородные данные в одну именованную структуру — старше его; она пришла из времени, когда самого C ещё не было.*

Значит, то, что мы зовём «синтаксисом C», C не изобретал. Он это унаследовал, отшлифовал и закрепил на полвека вперёд. Вопрос, стало быть, не в том, кто придумал скобки, а в том, откуда взялась сама идея, что код может иметь форму, независимую от железа, — и почему именно эта форма пережила всё, что было после неё.

За идеей стоит разрыв. С одной стороны — мышление категориями машины: регистр, адрес, прыжок. С другой — мышление категориями задачи: условие, повторение, результат. Между этими двумя берегами и пролёг первый язык, а весь дальнейший путь окажется медленным переходом от первого ко второму. C встанет ровно посередине — достаточно высоко, чтобы думать, достаточно низко, чтобы управлять.

Это корень дерева. С него всё и растёт.

1.1. Мир до языков: код для машины, не для человека

Чтобы оценить, каким переломом станет идея языка, нужно встать в точку, где её ещё не было.

В начале 1950-х программа не описывала задачу — она перечисляла движения машины. Программист, задумавший вычислить факториал, держал в голове не математическое «перемножить числа от единицы до n», а череду операций над железом: поместить значение в регистр, сравнить его с единицей, перейти по такому-то адресу, если результат меньше, умножить содержимое одной ячейки на содержимое другой, сохранить итог туда-то. Код был списком приказов конкретному процессору — на его наречии, в его терминах, под его устройство. Между намерением человека и работой машины не стояло ничего: программист сам спускался на уровень железа и говорил с ним напрямую.

Это был разговор без переводчика. Числовые коды операций, номера регистров, адреса ячеек, условные и безусловные переходы — словарь, в котором попросту не было слова для замысла. У величин не было имён, только адреса; у действий — не названия, а номера. Нельзя было спросить машину, чего ты хочешь добиться; можно было лишь продиктовать, что ей делать по шагам. И диктовать приходилось на

языке этой и только этой машины — на перфокарте, пробитой под её систему команд.

И запись эта была хрупкой до жестокости. Один неверный адрес — и переход уводил исполнение не туда, куда задумано; программа не падала с внятной жалобой, а тихо принималась портить чужую память или считать мусор как ни в чём не бывало. Не было имени, по которому можно окликнуть ошибку, — был лишь номер ячейки, который надо держать в голове самому. Программист вёл сплошную ведомость всего железа разом: где какое число, какой регистр свободен, какой адрес уже занят. Малейший сбой в этой ведомости рушил всё, и отыскать его удавалось, только пройдя весь маршрут машины шаг за шагом, её собственными глазами. Большая часть умственного труда уходила не на задачу, а на то, чтобы не потерять нить в учёте состояний.

Отсюда цена, которую сегодня трудно вообразить. Программа была сращена с процессором, как партитура, записанная не нотами, а перечнем клавиш одного-единственного рояля: третья слева, потом седьмая, потом снова третья. На этом рояле всё звучит. Перенесите запись на другой инструмент — и она обратится в бессмыслицу, потому что описывала не музыку, а механику конкретного корпуса. Перенести программу на другую машину означало не настроить её, а написать заново. Другой набор регистров, другие коды операций, другая карта памяти — и весь труд начинался с чистого листа. Код не принадлежал задаче. Он принадлежал

железу.

Вместе с программой к железу было приковано и умение. Виртуоз одной машины, знавший наизусть её систему команд и повадки её памяти, на другой машине оказывался едва ли не новичком: всё его мастерство держалось на устройстве, которого здесь больше не было. Знание не накапливалось поверх машин — оно испарялось вместе с ними. Каждое новое железо требовало не просто переписать программы, но заново выучить, как вообще на нём думать. Опыт не переносился, потому что был опытом не о задачах, а о конкретном механизме. Один и тот же давно понятый алгоритм человечество выводило заново снова и снова — столько раз, сколько было разных машин.

А раз так, то и думать приходилось категориями железа. Вопрос «как решить задачу» незаметно подменялся вопросом «как заставить эту машину выдать нужный результат». Любую мысль программист переводил на язык регистров и адресов ещё прежде, чем успевал её толком додумать, — и чем сложнее становилась задача, тем больше внимания уходило не на неё, а на бухгалтерию машинных состояний: что сейчас лежит в каком регистре, не затёрта ли нужная ячейка, куда указывает очередной переход. Замысел о задаче тонул в учёте механики. Человек обслуживал машину, а не машина — человека.

Код был прикован к железу короткой цепью. Между мыслью человека и работой процессора не стояло никакого по-

средника — только сама машина, и говорить с ней дозволялось исключительно на её условиях. Не код приближался к замыслу; замысел спускался к коду, разбирался на машинные движения и в них растворялся.

И вот что в этой картине важнее всего: тогда ещё не существовало идеи, что код может быть чем-то иным. Не было представления, что программа способна выражать намерение, а не диктовать движения. Машинный код не казался несовершенным или временным — он был единственным мыслимым способом говорить с вычислителем. Нельзя тосковать по абстракции, о которой не подозреваешь; нельзя счесть запись «слишком низкой», если других этажей ещё никто не придумал. Программирование держалось за то, что под рукой, не оттого, что было примитивным, а оттого, что вертикали попросту не было — подниматься было некуда.

Это и есть нулевая точка. Не первая ступень лестницы, а земля, на которой лестница ещё не построена. Всё, что случится дальше, — полвека языков, поколения создателей, бесконечные споры о компромиссах — будет одним непрерывным движением в одну сторону: прочь от машины, ближе к мысли. Каждый следующий шаг отдалит код от регистров и адресов и придвинет его к тому, что человек на самом деле хочет сказать. Сначала у ячеек появятся имена вместо номеров, и память перестанет быть голой россыпью адресов. Потом вместо череды операций можно будет записать выражение — и машина сама разберётся, в каком порядке его счи-

тать. Потом разрозненные данные соберутся в структуру, у действий появятся названия, у повторения — форма, не зависящая от того, как именно устроен переход внутри процессора. Расстояние между намерением и его записью будет сокращаться десятилетие за десятилетием — и в этом сокращении весь сюжет.

Но чтобы движение началось, кто-то должен был первым высказать ересь: что код описывает не машину, а задачу. Что между человеком и процессором может встать посредник, говорящий на языке алгоритма, а не железа, — и что писать, обращаясь к этому посреднику, окажется не только удобнее, но и правильнее, потому что задача переживёт любую машину, на которой её однажды решали.

1.2. ALGOL — латынь программирования

У ереси появились имя и дата. В 1958 году международная группа математиков и инженеров собралась не затем, чтобы приручить очередную машину, а затем, чтобы записать алгоритм как таковой — безразлично к тому, на каком железе ему потом исполняться. Через два года, в 1960-м, эта работа отлилась в язык, вошедший в историю под именем ALGOL. Его создавала не фирма под свой процессор, а сообщество, и сама эта отвязанность от производителя была заявлением: язык принадлежит задаче, а не машине. Вперт

вые код описывал, что нужно вычислить, а не какими движениями этого добиться на конкретном устройстве.

Что значит «описывать алгоритм»? Это значит, что в тексте программы появляется структура, отсутствовавшая в машинном коде, — структура самой мысли. Действие, которое логически едино, можно очертить как единый блок, отделив его от остального явной границей. Блоки можно вкладывать друг в друга, как вложены друг в друга рассуждения: внутреннее живёт в области видимости внешнего и не протекает наружу. У величин появляются типы — обещание того, какого рода значение здесь хранится, данное заранее, а не выясняемое по факту. А процедура может вызвать саму себя, потому что речь идёт не о прыжке по адресу, а о понятии, определённом через себя. Ничего этого нет в списке приказов процессору. Всё это есть в человеческом рассуждении — и ALGOL впервые перенёс это в код. А ещё — и это, быть может, тише всего прозвучало тогда и громче всего отозвалось потом — такой текст впервые стало возможно прочесть. Программу на ALGOL мог понять человек, в глаза не видевший машины, на которой ей предстояло работать: смысл держался в структуре записи, а не в знании железа. Код перестал быть личной перепиской программиста с процессором и стал текстом, обращённым в том числе к другому человеку.

Была у него и черта, которую легко недооценить. Грамматику ALGOL описали строго и формально — не россыпью примеров, а системой правил, по которым из простых кон-

струкций собираются сложные. Язык, на котором программируют, сам впервые получил язык, на котором его можно точно определить. Это переводило разговор о синтаксисе из области привычки в область математики: отныне можно было доказательно сказать, какая запись правильна, а какая нет, не сверяясь с поведением одной-единственной реализации. Форма кода стала предметом, о котором есть строгое знание.

И вот парадокс, ради которого всё это рассказано. Как рабочий инструмент ALGOL почти не выжил. Он рождался в академической среде, был придирчив к чистоте и равнодушен к прозе производства; в нём долго не находилось общего ответа даже на такой приземлённый вопрос, как ввод и вывод данных, — а без этого язык неудобен там, где надо не рассуждать об алгоритмах, а каждый день что-то считать и печатать. Индустрия повозилась с ним и разошлась по более практичным наречиям. На ALGOL, в строгом смысле, давно никто не пишет. Если мерить языки выживанием их собственных программ, ALGOL — проигравший.

И всё же по другому счёту он победил ещё при жизни. Когда учёному нужно было опубликовать алгоритм — в журнале, в учебнике, в письме коллеге, — он записывал его не на наречии своей машины, которое ничего не сказало бы читателю с другим железом, а на ALGOL. Десятилетиями язык служил способом сообщить алгоритм человеку, а не предписать его машине: на нём печатали, разбирали, обсуждали процедуры, которые в этом самом виде, возможно, никогда

и не запускали. ALGOL стал тем, на чём думали и договаривались, — общей записью, понятной поверх любого железа и любого десятилетия. Уже это делало его не инструментом, а языком в полном смысле слова.

Но есть и третья мера. Латынь тоже мёртвый язык: на ней не торгуются на рынке и не бранятся в очереди, у неё не осталось носителей, для которых она родная. И всё же она не исчезла — она растворилась. Её грамматика держит изнутри половину европейских языков; её падежи, её логика согласования, её корни проступают в итальянском, французском, испанском так, что, выучив один из них, начинаешь угадывать соседний. Латынь не та, на которой говорят, а та, из которой говорящие языки скроены.

ALGOL — латынь программирования ровно в этом смысле. Его собственное тело умерло, но его грамматика оказалась бессмертной. Идея явного блока, идея вложенности и областей видимости, идея типизированных величин, идея языка как формально описанной системы — всё это не похоронили вместе с языком, а унаследовали. Каждый следующий язык, всерьёз претендовавший на промышленную жизнь, строился на этой грамматике, даже когда полемизировал с ней. Программист, никогда не видевший ни строчки на ALGOL, ежедневно говорит на его наследниках и сам того не знает — как человек, не учивший латынь, всё же склоняет слова по её давно растворившимся правилам.

Это и есть корень дерева. Не самая толстая ветвь, не са-

мый заметный ствол — а то, что под землёй, чего не видно сверху и без чего не стоит ничего. ALGOL дал не язык, на котором будут писать, а форму, в которую отольются все языки, на которых будут писать. Он задал грамматику раньше, чем нашёлся словарь; предложил скелет раньше, чем появилось тело.

В этом первый намёк на то, что станет ясно много позже: судьба языка и судьба заложенной в нём идеи — две разные судьбы. Инструмент может устареть, проиграть в удобстве и кануть, не пережив своего десятилетия, — а идея, которую он первым высказал, продолжит жить в чужих телах, набирая силу с каждым новым воплощением. ALGOL умер как программа и остался как грамматика. Это не парадокс и не случайность — это правило, по которому здесь всё устроено.

Грамматике, чтобы жить, нужно тело. Латынь стала бессмертной не сама по себе, а через языки, на которых заговорили живые народы с их базарами, законами и любовными письмами. ALGOL остался бы изящной академической древностью, музейным чертежом грамматики, если бы его форму не подхватил кто-то, кому нужно было не рассуждать об алгоритмах, а строить работающую систему здесь и сейчас, под давлением реальной задачи и реального железа. Такой человек нашёлся — и пришёл он не из комитета, а из машинного зала.

1.3. Ритчи и C — язык, определивший полвека

Он пришёл не из комитета, а из машинного зала. На рубеже 1970-х в Bell Labs небольшая группа инженеров строила операционную систему, которой предстояло называться Unix, и упёрлась в стену, знакомую всякому, кто писал под конкретное железо: система, собранная на ассемблере одной машины, к этой машине и оставалась прикована. Перенести её на другую означало переписать всё заново — ту самую цену система платила в полной мере. Деннису Ритчи нужен был язык, на котором операционную систему можно написать однажды, а запускать на разных машинах. Не изящный, не академический, не образцовый — рабочий.

Грамматику он изобретать не стал. От ALGOL через несколько промежуточных языков тянулась линия, дошедшая до совсем аскетичного B, на котором экспериментировал рядом Кен Томпсон, — и Ритчи взял эту линию и дал ей тело, годное для системной работы. Он добавил то, чего B недоставало для серьёзных программ, прежде всего типы, и подогнал форму под суровую практику: компилятор должен быть простым, код — предсказуемым, язык — близким к железу настолько, чтобы на нём имело смысл писать операционную систему. В 1972 году из этого вышел C.

Решения, которые он принял, не выглядели судьбоносны-

ми. Блок кода Ритчи заключил в фигурные скобки. Условие записал через if, повторение — через for и while, возврат значения — через return. Разнородные данные позволил собрать в одну именованную единицу — struct. Дал прямой доступ к памяти через указатели и неглубокую, аккуратную систему типов. Каждое из этих решений можно было принять иначе; в других языках той поры их и принимали иначе. Но прижились именно эти, ритчевские, — и не на год, а на полвека. Спустя десятилетия мы по-прежнему очерчиваем блок фигурными скобками и выходим из функции словом return, и зовём это «синтаксисом C», хотя добрая половина этих решений старше самого C, а другая половина могла бы быть какой угодно.

В этих решениях сквозит не эстетика, а экономия. Фигурная скобка коротка и не требует от компилятора почти ничего; типы Ритчи держал неглубокими, чтобы язык оставался простым в реализации; форму правил так, чтобы её легко было перенести на новую машину вслед за Unix. Красота здесь ни при чём — есть инженерная бережливость человека, которому важнее, чтобы работало и переносилось, чем чтобы восхищало. Половину «того самого синтаксиса» продиктовала не теория языка, а скупость практика.

Почему прижились именно они? Не потому, что были доказуемо лучшими, — а потому, что пришли не одни. C явился в мир не предложением на бумаге, а с уже написанной на нём главной программой эпохи. Unix переписали на C —

и операционная система впервые перестала быть пленницей одного процессора. Чтобы перенести её на новую машину, теперь переписывали не систему, а компилятор; всё остальное переезжало почти без изменений. Проклятие машинного кода — «другая машина значит весь труд заново» — было снято. А вместе с Unix по университетам и вычислительным центрам расходился язык, на котором та была написана. Кто получал систему — получал и C; кто правил систему — учил C. Язык распространялся не уговорами, а тяжестью того, что на нём уже стояло.

Дальше это лишь нарастало. Операционные системы стали писать на C почти по умолчанию; на нём поднялась та невидимая инфраструктура, на которой потом заработало всё прочее. Языки, пришедшие следом, нередко и сами были сперва написаны на C, обращались к внешнему миру через его соглашения и в конечном счёте компилировались в тот же машинный код тем же путём. C перестал быть просто популярным языком и стал полом, на котором стоят остальные, — настолько привычным, что его уже не замечают, как не замечают фундамент, пока тот держит.

И здесь видно, чем C был по сути. Он встал мостом между двумя берегами, которые до него не сходились. С одной стороны — наследие ALGOL: блоки, функции, типы, именованные структуры, возможность думать категориями задачи. С другой — близость к железу, какой академические языки сторонились: указатель, напрямую адресующий память,

операции, ложащиеся на машинные команды чуть ли не одна к одной, отсутствие тяжёлого посредника между текстом и процессором. На C можно было рассуждать об алгоритме — и одновременно чувствовать металл под пальцами. Достаточно высоко, чтобы мыслить; достаточно низко, чтобы управлять.

Эту двойственность можно было пощупать в каждой строке. Глядя на выражение на C, опытный человек примерно представлял, в какие машинные команды оно обратится: язык почти ничего не прятал, и в нём угадывался процессор. И при этом то же выражение читалось как мысль о задаче, а не как перечень приказов железу. C не уводил от машины так далеко, чтобы потерять её из виду, и не прижимал к ней так близко, чтобы за регистрами пропала задача. Его потому и прозвали портативным ассемблером — полушутя, но метко: вся прямота ассемблера при способности думать поверх отдельной машины.

За этим компромиссом стояло убеждение, тихое и оттого прочное: программисту можно доверять. C не страховал, не запрещал, не отводил руку — он давал полный доступ и отступал в сторону, полагая, что человек, которому вручили власть над памятью, распорядится ею с умом. В этом была честность инструмента, не притворяющегося умнее своего хозяина. В этом же была и приоткрытая дверь, в которую следующие полвека станут входить один за другим те, кто решит, что доверие стоило бы и ограничить. Но это уже их

убеждения, не Ритчи; его собственное было именно таким, и язык получился по его образу.

Так «синтаксис C» сделался тем самым синтаксисом. Не потому, что он совершенен, — он не совершенен, — а потому, что встал ровно посередине: достаточно абстрактным, чтобы на нём думали, достаточно конкретным, чтобы ему доверили машину, и достаточно вовремя, чтобы прийти вместе с системой, которую захотели все. Грамматика ALGOL обрела наконец живое тело, и тело это зажило долго.

Насколько долго — лучше видно не в рассуждении, а глазами. Если решения Ритчи и вправду стали общим фундаментом, их след должен проступать там, где о самом C давно не вспоминают: в языках, родившихся через двадцать, тридцать, сорок лет после него, в чужих культурах и на других материках смысла. Чтобы это увидеть, довольно положить их рядом.

1.4. ДНК, которая живёт 50 лет

Вот одна и та же задача — вычислить факториал — записанная на четырёх языках, между которыми лежит больше сорока лет.

C, 1972:

```
int factorial (int n) {
if (n <= 1) return 1;
return n * factorial (n — 1);
```

```
}
```

Java, 1995:

```
int factorial (int n) {
if (n <= 1) return 1;
return n * factorial (n — 1);
}
```

Go, 2009:

```
func factorial (n int) int {
if n <= 1 {
return 1
}


return n * factorial (n — 1)
}
```

Rust, 2015:

```
fn factorial (n: u64) -> u64 {
if n <= 1 {
return 1;
}
n * factorial (n — 1)
}
```

Четыре языка. Четыре эпохи: начало семидесятых, середина девяностых, конец нулевых, середина десятых. Четыре разные культуры со своими ценностями и своими спорами, рождённые в разных компаниях, разными людьми, ради разных целей. И всё же если приглядеться, видно, что бóльшая часть этого кода — попросту одно и то же.

Тело функции у всех четверых заключено в фигурные скобки. У всех — функция с типизированным параметром на входе и типом результата на выходе. У всех ветвление сделано через if, выход — через return, а само вычисление — умножение на результат рекурсивного вызова — совпадает до знака. Это не похожие решения похожей задачи. Это один и тот же чертёж, перерисованный четыре раза. По прикидке — процентов восемьдесят символов несут одну и ту же мысль одной и той же конструкцией.

И стоит заметить, что именно совпало. Не случайные мелочи — а ровно тот костяк, что собирался на наших глазах: блок в скобках, вложенность, типы пришли от ALGOL; их закрепление и распространение — от C. То, что заложили между 1958 и 1972 годом, и есть та часть, которой четыре языка не коснулись. Совпадает не что попало, а фундамент, — и совпадает он именно потому, что фундамент.

А что же осталось на долю различий? Удивительно мало, и всё оно — на поверхности. C и Java здесь совпадают буква в букву: две строки из разных десятилетий, которые невозможно различить, не зная заранее, где какая. Go перестав-

ляет тип: пишет имя переменной прежде её типа и не ставит скобок вокруг условия. Rust меняет ключевое слово — fn вместо int на месте объявления, — помечает тип результата стрелкой -> и позволяет последнему выражению стать возвращаемым значением без слова return. Вот, в сущности, и весь список. Это различия выговора, а не грамматики: где-то иной порядок слов, где-то другая огласовка того же звука. Меняется акцент. Фраза остаётся той же.

Стоит, правда, оговориться сразу, потому что соблазн велик. Сказать «языки одинаковы» было бы и неправдой, и пошлостью. Различия между ними огромны — но живут они не здесь, не в силуэте функции. u64 в Rust обещает то, чего не обещает int; за скобками Java стоит управляемая память, а за скобками C — ручная память и полное доверие к программисту. Вот эти обещания и есть настоящая разница между языками, и она серьёзна. Но она лежит слоем ниже общего синтаксиса, а не в нём самом. Силуэт у всех один; расходятся они в том, что гарантируют под этим силуэтом.

И тогда становится видно то, чего не было видно, пока языки стояли порознь. Перед нами не четыре языка, делающие одно и то же. Перед нами одна мысль, записанная четыре раза. Замысел «вычислить факториал через самого себя, остановившись на единице» существует поверх всех четырёх записей; каждая из них — лишь его костюм, скроенный по моде своего десятилетия. Сними костюмы — под ними окажется одно тело.

Минуту назад перед нами были четыре разных языка — предмет выбора, повод для спора, четыре племени со своими знамёнами. Стоило поставить их вплотную и спросить об одной задаче — и четыре растворились в одном. Это не вывод, к которому подводят рассуждением; это то, что видно глазами и уже не развидеть.

То, что Ритчи заложил в 1972 году, ведёт себя в точности как наследственный код. Фундамент — функция, тип, блок в скобках, if, return — скопировал себя в Java, оттуда в Go, оттуда в Rust, и при каждом копировании язык-потомок дописывал что-то своё: виртуальную машину, горутины, проверки владения на этапе компиляции. Но несущая последовательность переходила из поколения в поколение нетронутой. Её не пересматривали, потому что незачем; её наследовали, как наследуют форму скелета, — не выбирая и часто не замечая. Это ДНК, которая живёт пятьдесят лет и не собирается умирать.

В живой природе самые древние гены — самые неприкосновенные: эволюция почти не трогает их, потому что они кодируют то, без чего организма попросту нет, и любая их поломка обрывает линию. С этим фундаментом — то же самое. Функция, блок, ветвление, возврат значения пережили полвека не оттого, что иначе никто не пробовал, а оттого, что менять их некуда: убери любое — и перед тобой уже не язык программирования, а нечто иное. Всё, что нарастало сверху, языки переписывали свободно и без сожаления; того, что ле-

жало в основании, не трогал никто. Самое старое оказалось самым живучим — в точности как в биологии.

Не «языки похожи» — похожими бывают и чужие друг другу вещи. А «это один язык», проступающий сквозь четыре имени. Хаос наречий, в котором учили выбирать племя, на просвет оказывается единой грамматикой с разными акцентами. И если это так с факториалом — самой простой из задач, — то стоит спросить, далеко ли расходятся языки там, где задачи серьёзнее.

Мы привыкли называть фигурные скобки, if *и* return *синтаксисом одного языка. Но один и тот же факториал проходит сквозь полвека и четыре чужие друг другу культуры почти неизменным — а значит, это не синтаксис одного языка, а общий фундамент всех. Корень назван. Из земли, которую мы разглядывали, поднимается ствол, и видно, что он один.*

А дальше ствол начнёт ветвиться. И вот загадка, с которой не уйти: если в основании лежит одна и та же грамматика, переходящая из языка в язык без больших потерь, — отчего языков так много и зачем понадобился каждый новый? Что не давало покоя людям, бравшимся переписывать уже работающее? Ответ начинается там, где кто-то впервые посмотрел на чужую работу и подумал: я могу лучше.

Глава 2. Отцы и дети

Шесть поколений: как каждый язык исправлял ошибки предыдущего

У всякого фундамента есть свойство, о котором не думают, пока он держит: на нём можно построить что угодно — и потому он немедленно перестаёт устраивать. Едва общая грамматика была заложена, история языков превратилась в историю недовольства ею. Каждый, кто приходил следом, смотрел на работу предшественника не как ученик на образец, а как мастер на чужую вещь — и видел в ней прежде всего то, что сделал бы иначе.

Это недовольство — не каприз и не тщеславие. За каждым «я могу лучше» стоит вопрос, на который предыдущий язык ответил так, а не иначе, и ответ этот всегда был выбором одного в ущерб другому. Что важнее — мощь или безопасность? Свобода программиста или гарантии компилятора? Возможность сделать всё — или невозможность сделать опасное? На такие вопросы нельзя ответить раз и навсегда. Можно лишь сместить акцент: усилить одно, ослабив другое, — и тем самым открыть новую трещину там, где прежде был монолит. Создатель языка — не изобретатель синтаксиса, а человек с убеждением о том, что для программиста важнее всего. Синтаксис — лишь форма, в которую это убеждение отлито.

Отсюда — закономерность, которую легко принять за насмешку судьбы: всякий, кто исправлял чужую ошибку, совершал собственную. Не по небрежности — по логике выбора. Закрывая одну опасность, он усиливал противоположную сторону, и там, в усилении, рождалась новая слабость, которой предшественник был лишён. Безопасность покупалась многословием. Простота — потерей выразительности. Полнота гарантий — крутизной входа. Каждый ответ был честным ответом своего времени и оставлял после себя вопрос для следующего.

И с каждым поколением ставка росла. Первые поправки были частными — добавить структуру, убрать опасный инструмент. Последние стали тотальными — попыткой одним языком закрыть все ошибки полувека разом. Амбиция не гасла от неудач предшественников, а разгоралась от них: чем больше накопилось нерешённого, тем смелее становился замах. Так получается не прямая линия улучшений, а спираль, набирающая обороты, — каждый виток шире предыдущего, и каждый, отталкиваясь от края прошлого, уходит дальше в ту же сторону.

2.1. C++ (1985) — порядок ценой сложности

Бьёрн Страуструп смотрел на C и видел не язык, а упрямую узость. C прекрасно справлялся с тем, для чего был

рождён, — описывал работу одной машины, близко к железу, без лишних посредников. Но Страуструп работал не с одной машиной и не с одной задачей. Он занимался моделированием больших систем, где программа — это не последовательность инструкций, а множество взаимодействующих сущностей, каждая со своим состоянием и поведением. И на этом масштабе C переставал помогать. Он давал инструменты, чтобы написать функцию, но не давал способа организовать тысячу функций так, чтобы человек удержал их в голове. Диагноз Страуструпа был прост и точен: C не масштабируется.

Это убеждение он принёс не из воздуха. Ещё аспирантом Страуструп работал с Симулой — языком, в котором мир описывался не как поток команд, а как множество объектов, каждый со своим состоянием и кругом доступных действий. Симула давала именно ту организацию мысли, которой не хватало C: программа в ней читалась как модель предметной области, а не как инструкция машине. Но за стройность платили скоростью — Симула была слишком тяжела для задач, где каждая миллисекунда на счету. Страуструп унёс из этого опыта одну неотступную мысль: организацию Симулы нужно совместить со скоростью C, не пожертвовав ни тем, ни другим. Так родилось то, что он сначала назвал «C с классами», — не новый язык, а C, которому добавили способ мыслить вещами. C++ вырос из этой прививки.

За диагнозом стоял вопрос, который C обходил молчани-

ем. Как описать не действие, а вещь — объект, у которого есть свойства и поведение, который знает, что с ним можно делать, а что нельзя? С знал про данные и про функции, но между ними не было связи: структура хранила поля, функции обрабатывали их, и ничто не мешало обработать данные не той функцией. Страуструп захотел связать данные с поведением и оградить внутреннее устройство вещи от внешнего мира. Так в язык вошла структура в большом смысле слова — не struct как набор полей, а способ мыслить программу как сообщество объектов.

Ответ, который он дал, был не точечной правкой, а целым новым слоем мышления, надстроенным над С. Классы связали данные с методами и спрятали детали за интерфейсом. Наследование позволило строить иерархии: общее выносилось вверх, частное уточнялось внизу. Полиморфизм дал возможность работать с разными вещами через одно имя — вызвать метод, не зная заранее, чей именно код выполнится. А шаблоны открыли способ писать алгоритм один раз и применять его к любому типу, не теряя ни скорости, ни проверки на этапе компиляции. Это был колоссальный прирост выразительности. Программа впервые могла говорить на языке предметной области, а не только на языке процессора.

Но Страуструп сделал ещё кое-что, и в этом «кое-что» спрятана вся драма языка. Он не стал ничего отнимать. С++ остался полностью совместим с С: всё, что было опасного

в C, осталось опасным и здесь. Указатели, ручное управление памятью, арифметика адресов, возможность прочитать то, что уже освобождено, или выйти за границу массива и получить не ошибку, а тишину, за которой прячется разрушенная память, — всё это перешло в новый язык нетронутым. К старым опасностям добавились новые, рождённые самой мощью надстройки: множественное наследование, в котором иерархия запутывается сама в себе; перегрузка, в которой один и тот же знак значит в разных местах разное; шаблоны, разворачивающиеся в нечитаемые простыни сообщений об ошибках. Язык умел почти всё — и почти ни от чего не защищал.

У этой незащищённости есть точное имя — undefined behavior. Это область, где язык умывает руки: ты сделал нечто, не разрешённое правилами, и отныне может произойти что угодно — программа упадёт, или выдаст мусор, или будет годами работать правильно, чтобы рухнуть в день, когда это дороже всего. Компилятор не обязан предупреждать; чаще всего он молчит. Ответственность целиком на программисте. Он должен сам, в голове, держать все правила, которые язык не проверяет за него. C++ дал инженеру власть над машиной почти без ограничений — и ровно настолько же оставил его одного перед последствиями.

Была у этого решения и тень иного рода, культурная. Объектная модель C++ оказалась так влиятельна, что на двадцать лет превратилась в догму. Целое поколение училось,

что правильно спроектированная программа — это иерархия классов, что почти всё на свете следует моделировать как объект с наследниками, что мышление об архитектуре — это мышление об иерархиях. Идея, рождённая как инструмент против сложности, сама стала источником сложности: системы обрастали слоями абстракций, заведёнными не задачей, а привычкой видеть мир объектно. Лекарство прижилось так прочно, что начало напоминать болезнь.

За всеми этими решениями — конкретное убеждение, и оно объясняет их разом. Страуструп верил, что программисту нужна мощь и что мощь нельзя обрезать ради удобства слабого. Дать инженеру полный контроль над машиной, ничего не пряча и ничего не запрещая, — а он сам разберётся, как этим не навредить. У хирурга скальпель не снабжён предохранителем, и у мастера-краснодеревщика стамеска не затуплена ради безопасности новичка: профессиональный инструмент опасен ровно настолько, насколько остёр, и тупить его — значит делать бесполезным. Страуструп строил инструмент для профессионала. Это вера в зрелость и компетентность того, кто держит инструмент: настоящему мастеру не нужна защита от собственной руки, ему нужно, чтобы инструмент не мешал. C++ построен на доверии к человеку — и в этом доверии его величие и его рана одновременно. Потому что человек устаёт, ошибается и забывает, а undefined behavior — нет.

Так первый из наследников дал линии структуру, не сняв

ни одной из её прежних опасностей и добавив собственные. Мощь, отданная программисту без страховки, — слишком щедрый и слишком рискованный дар, чтобы остаться последним словом. Следующий, кто посмотрит на эту щедрость, увидит в ней прежде всего опасность — и решится отнять у программиста часть свободы, чтобы вернуть ему безопасность.

2.2. Java (1995) — безопасность ценой свободы

Джеймс Гослинг смотрел на C++ с другой точки, чем Страуструп, — и потому видел в нём другое. Страуструп смотрел глазами мастера, которому нужен острый инструмент. Гослинг смотрел глазами человека, отвечающего за систему, которую пишут не один мастер, а сотни инженеров разной выучки, и в которой ошибка любого из них может обрушить целое. С этой точки главное достоинство C++ — безграничное доверие к программисту — оборачивается главной опасностью. Скальпель без предохранителя хорош в руке хирурга и смертелен в руке уставшего. А в большой системе всегда найдётся уставшая рука. Диагноз Гослинга был не о языке, а о его цене для дела: C++ слишком опасен для бизнеса.

Из этого вырастал вопрос, противоположный вопросу Страуструпа. Не «как дать мастеру полную власть», а «как

устроить язык так, чтобы и самый слабый в команде не смог разрушить всё»? Самые дорогие ошибки в C-линии рождались там, где программист сам распоряжался памятью: обращение к уже освобождённому участку, утечки, выход за границу, висячий указатель, который годами притворяется исправным. Ответ Java был радикален в своей простоте — отнять у программиста память вовсе. Сборка мусора взяла на себя освобождение: ты больше не закрываешь то, что открыл, об этом заботится среда. Указатели уступили место ссылкам, которые нельзя двигать арифметикой. Множественное наследование — клубок, в котором путались иерархии C++, — было просто вырезано. Опасный инструмент убирался не затуплением, а изъятием.

И ещё один ход изменил всё. Программа в Java компилировалась не под конкретный процессор, а под абстрактную машину — виртуальную машину Java. Написанное один раз исполнялось всюду, где была эта машина. Цепь, приковывавшая код к железу с самого начала, была разрублена в новом месте: между программой и процессором встал посредник, говоривший за неё на любом наречии. Это оказалось так удобно, что переживёт сам язык и станет общей идеей, к которой ещё вернутся. Одна и та же собранная программа шла на большой машине в серверной, на рабочем столе, позже — на телефоне, и её не нужно было пересобирать под каждую: посредник переводил её на язык любого железа на месте. Цепь, которую впервые надорвал ещё первый язык об

алгоритме, здесь была перерублена начисто — и место разруба оказалось так удачно, что двое из наследников возьмут эту самую идею готовой.

Из всего этого Java сплела вокруг программиста страховочную сетку. Но в сетке была дыра, и имя ей — null. Убрав опасную память, Java сохранила старую ссылку в никуда — значение, которое означает «здесь ничего нет» и которое система типов не отличала от настоящего объекта. Можно было обратиться к этому ничто как к вещи — вызвать у него метод — и получить не отказ компилятора, а крушение во время работы: NullPointerException, самый знакомый сбой за всю историю языка. Идея null была не нова и не принадлежала Гослингу: человек, введший пустую ссылку в обиход десятилетиями раньше, потом назовёт её своей ошибкой на миллиард долларов. Гослинг унаследовал её, не задумавшись. Сетка ловила всё, кроме одного, — и сквозь это одно проваливались чаще всего.

У null была глубинная природа, которую тогда не разглядели. Всякий язык рано или поздно встаёт перед вопросом: как выразить отсутствие? Как сказать «здесь могло быть значение, но его нет»? Java ответила на этот вопрос самым простым и самым опасным способом: пусть отсутствие выглядит как присутствие, только пустое, — и пусть тип молчит о разнице. Переменная объявлена как объект определённого вида, но в ней может не оказаться ничего, и компилятор об этом не предупредит. Отсутствие притворялось значени-

ем до самого мгновения, когда к нему обращались. Это был неверный ответ на верный вопрос, и вопрос остался стоять, дожидаясь тех, кто ответит иначе.

Безопасность стоила и иначе — словами. Java заставляла программиста объявлять, оборачивать, оговаривать; вокруг содержания нарастал обряд. Чтобы высказать простое намерение, сперва возводили леса: заводили оболочку для метода, дважды называли тип, оборачивали возможную неудачу в предписанную форму. Мысль тонула в собственной раме. Проверяемые исключения требовали для каждой возможной неудачи либо обработать её на месте, либо явно объявить, что она может случиться, — замысел дисциплинирующий, на деле обернувшийся шумом, сквозь который терялся смысл. А сверху язык, созданный, чтобы оградить среднего инженера от ошибки, породил и собственную культуру — культуру архитектуры ради архитектуры, фабрик, производящих фабрики, слоёв, заведённых не задачей, а привычкой к осторожности. Многословие из свойства синтаксиса стало свойством мышления. Безопасность, доведённая до обряда.

За всеми этими решениями — убеждение, и оно зеркально перевёрнуто относительно убеждения Страуструпа. Там, где C++ доверял человеку, Java ему не доверяла. Программисту нельзя доверять память — но можно доверять null. В этой формуле — вся непоследовательность реформы. Недоверие дотянулось до памяти и остановилось перед пустой ссылкой. Почему? Потому что память ощущалась как

власть, а null — как безобидное удобство, «просто отсутствие». Опасность, маскирующаяся под удобство, переживает любую реформу: её не замечают именно потому, что она не выглядит опасной. Гослинг отнял у программиста острые инструменты и оставил ему тупой на вид предмет, оказавшийся самым колющим из всех.

И всё же недоверие Java было честным и оправданным. Чтобы понять этот выбор, стоит выйти за пределы программирования. Частная мастерская и общественное здание подчиняются разной логике. В мастерской мастер волен на что угодно: он один отвечает и за результат, и за себя. Но здание, в которое войдут тысячи, строится по своду правил, и свод этот стесняет каждого строителя, запрещая то, что в частной работе сошло бы с рук, — ради того, чтобы оплошность одного не обрушилась на головы всех. Правила делают здания скучнее и крепче. Java выбрала логику свода: она строила не мастерскую для виртуоза, а каркас для большого, разнородного, сменяющегося коллектива, где цена одной ошибки измеряется не репутацией автора, а упавшей системой. За её недоверием стояла не трусость, а ответственность: язык брал на себя миллиарды строк делового кода, который должен работать годами в руках, которые его не писали. Это не провал воображения, а взрослый выбор — пожертвовать свободой немногих сильных ради того, чтобы не падала система, держащаяся на многих обычных. C++ дал мощь и оставил человека одного с последствиями; Java вернула человеку защи-

ту, отняв часть свободы и заставив платить словами. Каждый ответил на свой вопрос, и каждый ответ открыл новую трещину.

Идея абстрактной машины, на которую можно писать раз и исполнять везде, была слишком хороша, чтобы остаться при одном языке. Кто-то возьмёт её же — и спросит не «как сделать безопасно», а «как сделать удобно», сохранив сетку и срезав обряд.

2.3. C# (2000) — итерация без революции

Андерс Хейлсберг был не теоретиком, а оружейником языков. До C# он сделал Turbo Pascal и Delphi — инструменты, которыми каждый день работали живые программисты, и эта привычка определила его взгляд. Его не занимало, какая идея смелее или чище; его занимало, что именно раздражает человека за клавиатурой и как это раздражение убрать. С такой позиции язык Гослинга выглядел надёжным, но безрадостным: безопасным ценой свободы, защищённым ценой многословия. И вопрос, который поставил Хейлсберг, был скромен до дерзости. Не «какую новую вселенную построить», а «как сделать уже найденное приятным»? Возьмём безопасность Java и её абстрактную машину — и срежем с них обряд. Сделаем Java удобнее.

Ответ складывался не из переворотов, а из множества тщательных сглаживаний. Там, где Java заставляла для каж-

дого поля писать пару обрядовых обёрток для чтения и записи, C# позволил полю самому выступать как свойство, спрятав церемонию внутрь. Это мелочь — но из таких мелочей состоит ежедневный труд. Дальше шли вещи покрупнее. Запросы к данным вошли прямо в язык: коллекцию можно было спрашивать, как спрашивают базу, — данные стали полноправным предметом вопроса, а не грудой, которую перебирают вручную. А затем явился способ писать конкурентный код так, чтобы он читался последовательно, — пара слов, превращавшая запутанную асинхронность в почти линейный текст. Прежде ожидание чего-то медленного — ответа сети, чтения с диска — рассыпало логику на клочки: ты передавал продолжение как отдельный кусок, который выполнится потом, и программа разваливалась на гнёзда обратных вызовов, где порядок чтения уже не совпадал с порядком исполнения. Два слова собрали эти клочки обратно: код снова читался сверху вниз, как рассказ, а машина сама распутывала, что за чем дождаться. C# дал этот способ за годы до того, как он стал всеобщим. Модель, по которой потом будет писать полмира, впервые заработала здесь.

Хейлсберг сделал и то, на что не решилась Java, — вернулся к дыре в сетке. Он научил систему типов отличать ссылку, которая может оказаться пустой, от ссылки, которая пустой быть не может, — и заставил компилятор предупреждать там, где раньше была тишина перед крушением. Вопрос «как выразить отсутствие?», оставленный Java без вер-

ного ответа, получил здесь ответ внятный: пусть тип сам говорит, может ли в нём не оказаться ничего. Ответ пришёл поздно и не сразу обязательным — но он был дан, и дан в той же неревизионной манере: не сломать прошлое, а тихо добавить недостающее.

И всё же каждое из этих улучшений жило за стеной. C# был привязан к одной экосистеме — к Windows, к платформе. NET, к коммерческому миру одной компании. Долгие годы он не работал там, где жил открытый мир, где складывалась культура свободного, кроссплатформенного кода. И потому его находки, нередко первые, доставались огороженному саду, а не общему полю. Способ писать асинхронность по-человечески существовал здесь раньше, чем где бы то ни было, — но профессия в массе своей выучила этот приём позже и у других языков, перенявших его следом. Лучший ответ был дан в комнате, в которую большинство так и не вошло.

За всем этим стояло убеждение, и оно было тише, чем у предшественников, оттого не менее твёрдым. Не нужна революция — нужна хорошая итерация. Хейлсберг не пытался опрокинуть C-линию и переписать парадигму набело; он верил, что движение вперёд — это терпеливое снятие трения, версия за версией, год за годом. Эту же веру он позже подтвердит ещё раз, взявшись за JavaScript: не заменить его новым языком, а тихо надстроить сверху слой типов, сделав привычное безопаснее, не отняв привычного. Один и тот же

инстинкт, приложенный дважды, — это уже не случай, а мировоззрение. И история доказала его правоту: именно так, итерацией, а не переворотом, и движется вся линия. Но у этой правоты оказалась жестокая сноска. Он был прав — а мир, занятый другим, этого не заметил. Тихая правота.

В этом — особая, почти грустная фигура среди создателей. Не пророк, отвергнутый при жизни, и не бунтарь, опрокинувший прежний порядок, а мастер, который раньше многих нашёл верное решение и получил за него меньше всех признания — потому что говорил на наречии, слышном лишь за стеной. У этой судьбы есть точный двойник за пределами программирования. Монах, открывший законы наследования на грядке гороха, опубликовал их в безвестном издании — и они пролежали незамеченными десятилетия, пока другие, заново придя к тому же, не нашли их готовыми. Открытие было верным с первого дня; не хватало лишь того, чтобы оно прозвучало там, где слушают. Так и здесь: правоту Хейлсберга подтвердят не аплодисментами, а тем, что те же приёмы один за другим всплывут в чужих языках как собственные открытия. Идея, рождённая в огороженном саду, прорастёт повсюду — но уже без имени садовника.

Среди шести наследников C# — самый чистый случай движения вперёд без разрушения. Он почти ничего не отнял у Java: взял её устройство как есть и принялся прибавлять — удобство за удобством, версию за версией. Других наследников тянуло то отнять опасное, то снести и построить заново;

C# не сносил и почти не отнимал. Вся его драма уместилась не в том, что он сделал, а в том, где он это сделал: единственным крупным компромиссом стала привязка к домену одной компании, и именно она, а не нехватка идей, держала его в тени. Чистая прибавка плюс цена места рождения — вот вся его арифметика.

C# улучшал прибавлением — со вкусом, с тактом, но всегда добавляя. И каждое поколение до сих пор двигалось так же: структура, безопасность, удобство — куча росла. Настал момент, когда кто-то задал противоположный вопрос. Что, если улучшение — это не добавить правильное, а убрать неправильное? Что, если язык становится лучше не оттого, что в нём прибавилось, а оттого, что из него вычли? Не поправка внутри общего направления, а бунт против самого направления.

2.4. Go (2009) — бунт простоты

У этого языка особое место в семейной хронике, и дело не только в его устройстве. Среди его создателей — Роб Пайк, Роберт Гризмер и Кен Томпсон, а Томпсон был одним из тех, кто стоял у самого корня: он строил Unix и язык, из которого вырос C. Получается почти буквальная сцена: отец, заложивший фундамент, спустя десятилетия возвращается и смотрит на то, во что разрослось его дело. И то, что он увидел вместе с соавторами в стенах Google, было не нехваткой, а

избытком. Огромные сборки, тянущиеся минутами; кодовая база, заросшая слоями абстракций; язык, в котором, чтобы понять чужой файл, нужно сперва распутать иерархию. Диагноз был противоположен всем прежним. Предшественники чинили линию, добавляя ей недостающее; эти трое сказали иное. Всё стало слишком сложным.

И ответ был так же противоположен. Каждое поколение до сих пор двигалось вперёд, прибавляя; Go двинулся вперёд, отнимая. Из языка вырезали наследование — те самые запутанные иерархии, и на их место поставили простое сложение частей: вещь собирается из других вещей, а не выводится из предков. Вырезали исключения — невидимый поток управления, который прыгает туда, где его не ждёшь, — и вернули ошибку на свет, сделав её обычным возвращаемым значением. Убрали тяжёлого посредника между программой и машиной: никакой виртуальной машины, программа собирается в один самодостаточный двоичный файл, который просто запускается где угодно, и собирается за секунды, а не за минуты. Развёртывание, прежде требовавшее установить на машину целую среду нужной версии, свелось к тому, чтобы скопировать туда один файл, — и всё. Программа перестала тащить за собой шлейф зависимостей; она снова стала вещью, а не обрядом установки. А взамен дали лёгкость, которой ни у кого не было, — горутины, способ запускать тысячи параллельных задач, не задумываясь о цене каждой. Прежде поток был тяжёл: их заводили поштучно и

считали, как редкий ресурс. Горутина оказалась так дёшева, что считать перестали вовсе — конкурентность из дефицита, который берегут, превратилась в воздух, которым дышат. И связывались эти задачи не общей памятью, за которую дерутся, а передачей сообщений: не «поделим один кусок и будем сторожить его друг от друга», а «передадим из рук в руки». Язык умещался в голове целиком. Новичок открывал любой файл и понимал его без проводника. Простота была здесь не следствием бедности, а сознательно выбранным решением.

В этом есть жест, знакомый далеко за пределами программирования. Скульптор перед глыбой мрамора не прибавляет — он убирает всё, что не есть фигура, пока она не проступит сама. Совершенство достигается не тем, что уже нечего добавить, а тем, что нечего отнять. Вся прежняя линия работала как живописец, накладывающий слой за слоем; Go впервые взял в руки резец и стал работать как скульптор — улучшая отсечением. Это другой способ мыслить инструмент: не «чего ему не хватает», а «что в нём лишнее».

Но у вычитания есть дно, и Go в него упёрся. Старую дыру он не залатал: nil перешёл в него нетронутым, всё та же пустая ссылка, всё та же тишина перед сбоем. И в этом — тихая улика родства. Язык, отвергнувший наследование, исключения и тяжёлую среду исполнения, язык, объявивший войну всему лишнему, что накопила линия, — этот самый язык сохранил её древнейшую ошибку, ту, что тянется от первой

пустой ссылки. Можно восстать против отцов, отбросить их манеру, их сложность, их догмы — и всё равно нести в себе их старейший просчёт, не замечая его, потому что он давно перестал выглядеть как ошибка. Иное наследство глубже любого бунта. Честность с ошибками он довёл до обряда — if err!= nil, повторяющееся в конце каждого шага, как удары пульса: правдиво, но утомительно, страница за страницей одно и то же. А главное — он больше десяти лет отказывался от обобщённого кода. Нельзя было написать одну функцию для многих типов, не копируя её или не теряя проверку; язык, превыше всего ценивший простоту, заставлял повторяться. Не было и способа сказать «это одна из таких-то форм» — ни ясного выбора по структуре, ни суммы вариантов. Минимализм, бывший достоинством в центре, к краям оборачивался нехваткой.

А потом, в 2022 году, в версии 1.18, Go добавил то самое обобщение, от которого открещивался все эти годы. И само его долгое сопротивление было результатом эксперимента, а не упрямством. Вычитание работает прекрасно — ровно до того дня, когда отнятое начинает стоить дороже сбережённой простоты. Эту границу Go не вывел из теории, а нащупал сам — и, дойдя до неё, сам же признал.

За всем этим — убеждение, перевёрнутое относительно всех предыдущих. Лучше убрать лишнее, чем добавить правильное. Там, где каждый предшественник верил, что прогресс — это верная прибавка, Go поверил, что прогресс —

это верное изъятие. И это был настоящий эксперимент, а не вопрос вкуса: можно ли сделать язык лучше, отнимая? Ответ, который дал Go, точен и обоюдоостр. Да — но лишь до предела. Простота покупает ясность, пока не начинает стоить возможностей; за этой чертой дно отталкивает обратно.

В этом ответе есть та же фигура, что и во всех прежних, только вывернутая наизнанку. Страуструп добавил мощь и получил сложность. Гослинг добавил защиту и получил многословие. Go убрал сложность — и получил нехватку выразительности. Каждый закрывал чужую слабость, обнажая свою; направление менялось, закон оставался. Сильная сторона решения и его слабость растут из одного корня, и нельзя усилить одно, не оголив другое.

Go восстал, вырывая лишнее. Он доказал силу меньшего и нашёл его границу. Следующий шаг будет сделан с противоположным чувством — не бунтом против наследства, а починкой его: взять несовершенную вещь и исправить её, не ломая, сохранив работающее и залатав сломанное.

2.5. Kotlin (2011) и Dart (2011) — два пути реформы

К 2011 году Java была рабочей лошадью уже шестнадцать лет, и её слабости давно перестали быть загадкой. Дыра, через которую проваливался null; многословие, в котором тонула мысль; обряд, наросший вокруг простых действий, —

всё это было не тайной, а хорошо составленной картой ран. И в один и тот же год две команды взялись лечить один и тот же язык. Удивительно не то, что они взялись, а то, что разошлись на первом же шаге — на вопросе, который встаёт перед всяким, кто хочет исправить старое жильё. Чинить дом, в котором живёшь, или построить новый рядом? Диагноз был общий. Стратегии — противоположные.

Kotlin выбрал первый путь — эволюцию изнутри. За ним стояла JetBrains, компания, делавшая инструменты для программистов и потому знавшая ежедневную боль Java не по рассказам, а изнутри, как знает дом жилец, проживший в нём годы. Её ставка была осторожной до дерзости: не покидать виртуальную машину Java, не рвать совместимость — исправлять язык, не выходя из него. Kotlin работал на той же машине, вызывал Java и вызывался из неё; проект можно было переводить файл за файлом, живя сразу в двух языках, не выселяясь ни на день. Это и есть починка дома, в котором живёшь: один файл ещё на старом наречии, соседний уже на новом, и компилятор не видит между ними границы — стены подновляют по комнате, не выгоняя жильцов на улицу. И внутри этого тесного условия он залечил знаменитые раны. null стал частью системы типов: тип, в котором может не оказаться значения, теперь пишется иначе, чем тип, в котором значение есть всегда, и компилятор не позволит их перепутать. Вопрос «как выразить отсутствие?», оставленный Java без верного ответа, получил ответ внятный и обязательный.

Обряд исчез: то, что прежде требовало десятков строк обёрток — описать данные, а потом вручную научить их сравниваться, печататься, отдавать и принимать поля, — свелось к одной строке; данные снова стали просто данными, без церемонии вокруг них. Корутины дали дешёвую конкурентность. «Java, каким он должен был быть» — не новый мир, а тот же самый мир, починенный. И мир согласился: спустя несколько лет крупнейшая платформа, на которой жила Java, объявила Kotlin предпочтительным языком — редкий случай, когда «каким должен был быть» признаётся официально, поверх оригинала.

Dart выбрал второй путь — перестройку с нуля. Ставка Google была обратной: не латать, а строить заново. Dart был новым языком с собственным окружением, не привязанным к платформе Java, и с самого начала нацеленным на определённую область — на построение интерфейсов. Он тоже залатал дыру null — гарантированную сквозную защиту от пустой ссылки он довёл до конца в 2021 году. Он тоже срезал многословие, дал асинхронность по-человечески, дал способ подмешивать поведение в разные сущности, минуя жёсткие иерархии. Но он требовал переезда: новый дом, новый фундамент, а старый ты оставляешь позади. И у этого дома была трудная судьба. Поначалу Dart метил совсем в другую цель — стать заменой языку браузера, встать на место JavaScript прямо внутри Chrome; этот замысел не сбылся, встроенной машины в браузере так и не появилось, и язык остался стоять

с фундаментом, но без жильцов. Долгие годы Dart дрейфовал, язык в поисках причины существовать, — пока Flutter не дал ему эту причину, способ собирать интерфейсы сразу для многих платформ; и тогда новый дом наконец обрёл жильцов. Язык, которому дважды искали назначение, нашёл его не в том, для чего строился, — обычная участь того, кто заложил дом прежде, чем понял, кто в нём будет жить. Компромисс домена — интерфейсы — вписан в самую его суть.

Сними подробности — и под ними окажется одна развилка. Оба языка — дети Java, оба чинят одни и те же две раны, null и многословие, и оба исповедуют одно и то же. Ни один не восстал, как Go, вырывая лишнее; ни один не заперся, как C#, за стеной экосистемы. Они расходятся только в стратегии. Чинить старый дом, оставаясь внутри, пока подновляешь стены, — или построить новый рядом и позвать всех переехать? Правильного ответа здесь нет — есть темперамент. Kotlin доверяет преемственности: лучшее в старом доме стоит сохранить, переезд дорог и рискован. Dart доверяет чистому началу: на старом фундаменте всех ошибок не исправишь, иногда честнее заложить новый. Две дороги от одной двери, и обе ведут прочь от Java — но в разные стороны.

И у каждой дороги своя цена — тот же закон, что правил всеми предыдущими. Кто чинит старый дом, наследует не только его стены, но и его фундамент: Kotlin остался на машине Java и вместе с её охватом принял её границы —

он не может уйти дальше, чем позволяет почва, на которой стоит. Кто строит новый, остаётся на пустом участке один: Dart свободен от старого фундамента, но и отрезан от всего, что на нём выросло, — от библиотек, привычек, накопленного мира Java, ради которого многие и не трогаются с места. Преемственность платит несвободой. Чистое начало платит одиночеством. Усилить одно, не оголив другого, не вышло и здесь.

Примечательно, к чему оба тянутся прежде всего. Не к новым возможностям, не к собственным изобретениям — к старой дыре. Флаг, который каждый поднимает первым, — безопасность от null. Рана, которую Java оставила, вопрос, на который она не сумела ответить, становится сердцем обеих реформ. Один и тот же изъян независимо тянет к себе одну и ту же починку, словно к больному месту само собой возвращается лекарство. За этим стоит общее убеждение, тихое и твёрдое: можно исправить прошлое, не разрушая его. Там, где Go сказал «вырвать», Kotlin и Dart сказали «залечить». Не революция, а реформа; не разрыв, а продолжение. Уважение к работающему наследству, которое не сжигают за то, что в нём есть трещины, а чинят, потому что в нём слишком много живого.

Но обе реформы лечили по частям. Каждая починка была местной: null здесь, обряд там, конкурентность в углу. Прицельно, терпеливо, поодиночке — одна рана за раз, оставляя остальное на потом или на чужие руки. И ни один из них

не задал вопроса, который висел над всей линией с самого начала. Что, если не исправлять изъяны по очереди, а попытаться закрыть все ошибки разом — в одном языке, с самого основания, не уступив ни в одном компромиссе?

2.6. Rust (2015) — попытка закрыть всё разом

У всей линии до сих пор была одна молчаливая аксиома, которую никто не оспаривал, потому что она казалась законом природы: за всё надо платить. C++ взял мощь и заплатил опасностью. Java взяла безопасность и заплатила свободой. Go взял простоту и заплатил выразительностью. Каждый соглашался на сделку, спорил лишь о её условиях — что отдать, что получить, — но самого торга не отменял никто. Нельзя иметь всё сразу; можно только выбрать, чем пожертвовать.

Грэйдон Хоар, начавший Rust и нашедший ему дом в Mozilla, поставил под сомнение саму аксиому. Его замысел был ересью против закона торга: производительность C++ и безопасность — вместе, без компромиссов. Не выбрать между скоростью и защитой, как выбирали все, а отказаться выбирать. Взять то, за что C++ платил undefined behavior, и то, за что Java платила свободой, — и получить оба, не заплатив ни тем, ни другим. Если каждый предшественник закрывал одну рану, обнажая другую, то Rust замахнулся закрыть их

все сразу, не открыв ни одной новой. Это была вершина амбиции, к которой спираль шла полвека.

И замахнуться так он мог лишь потому, что пришёл последним. Он стоял на всех прежних попытках — и, что важнее, на всех задокументированных неудачах. Ему не нужно было гадать, какие раны линии настоящие: полвека уже составили их список. Где Страуструп ещё нащупывал, что вообще болит, Rust получил готовую карту всех болезней разом — и потому смог лечить не наугад, а по полному перечню. Привилегия опоздавшего: вся история ошибок лежала перед ним открытой.

Он пошёл по списку ран, накопленных линией, закрывая каждую как ответ на давний вопрос. Старейшая рана — ручная память и undefined behavior: как дать программисту полный контроль над памятью и при этом не дать ему разрушить её? C++ доверил человеку и проиграл; Java отняла память и заплатила посредником-сборщиком. Rust нашёл третий путь — ownership. Компилятор сам отслеживает, кто владеет каждым значением и когда оно освобождается, и доказывает ещё до запуска, что к освобождённому уже не обратятся. Правило почти аскетично в своей простоте: у каждого значения ровно один владелец; кончился владелец — освободилось значение; одолжить его можно, но компилятор считает все одолжения и не даёт двум рукам писать в него разом. Из трёх несложных правил безопасность памяти выпадает как теорема, а не как надежда. Безопасность памяти без сборщика,

без среды-надзирателя — ровно то, что C++ и Java решили каждый лишь наполовину.

Дальше — дыра на миллиард долларов. Как выразить отсутствие так, чтобы его нельзя было проигнорировать? Option <T> сделал отсутствие отдельным типом: пустоту нельзя принять за значение, компилятор заставит открыть коробку и обработать случай, когда она пуста. Вопрос, оставленный Java без верного ответа, закрыт окончательно и неотвратимо. Как сообщить об ошибке, не пряча её в невидимый поток управления и не превращая в утомительный обряд? Result <T,E> сделал ошибку обычным значением, которое нельзя не заметить, — честным, как в Go, но не изматывающим. Как ветвиться по форме данных, ничего не упустив? match с проверкой полноты: компилятор откажется собирать программу, пока не разобран каждый случай. Как делить поведение между сущностями, не сплетая их в жёсткие иерархии? Traits — общее поведение без родословной C++. И самый старый ужас — гонки данных в конкурентном коде — Rust сделал невозможными на уровне компилятора, распространив ту же идею владения на потоки: к одному куску памяти не дотянутся две руки разом, потому что компилятор этого не позволит. А ведь это был худший класс ошибок из всех: они проявлялись редко и неповторимо, всплывали под нагрузкой в работающей системе и исчезали, стоило начать их искать, — их нельзя было поймать, потому что нельзя было воспроизвести. Rust перенёс весь этот класс страха с ноч-

ных дежурств на этап сборки: то, что прежде ловили месяцами в бою, теперь отвергалось до запуска, одной проверкой.

Стоит сказать честно: почти ничего из этого Rust не изобрёл. Отдельный тип для отсутствия, ветвление по структуре, общее поведение без иерархий — всё это десятилетиями жило в других языках, в стороне от большой дороги, верное, но бездомное. Заслуга Rust была не в изобретении ответов, а в том, что он собрал их в одно тело, которое индустрия наконец смогла принять. Он не открыл — он составил. Правильные идеи, ждавшие языка, способного их вынести, дождались его.

Но закон торга не отменить — можно лишь перенести цену в другое место, и Rust перенёс её на программиста. Компилятор, гарантирующий так много, требует доказать, что код достоин гарантии, — а доказательство даётся тяжело. Borrow checker, проверяющий владение, стал знаменитым противником: он отвергает код, который на вид безупречен, и заставляет перестраивать само мышление о том, кто чем владеет и как долго. Новичок пишет то, что собралось бы в любом другом языке, и смотрит, как это раз за разом отклоняют по причинам, поначалу неотличимым от придирок, — пока однажды модель не щёлкнет, и не окажется, что все эти отказы исподволь учили способу думать, который теперь уже не разучиться. Ownership — это реальный барьер, а не пустяк, о который спотыкаются новички; о него спотыкаются и опытные, и многие отступают. За «без компромиссов»

заплачено не потерей возможностей, а крутизной подъёма. Сделка не исчезла — она переехала из программы в голову того, кто её пишет.

За всем этим — убеждение, и оно зеркально перевёрнуто относительно самого первого. Страуструп доверил человеку всё. Java не доверила ему память, но доверила null. Rust не доверяет человеку вовсе — и выносит знание из ненадёжной головы в неусыпный компилятор. Гарантии больше не держатся на дисциплине; они держатся на доказательстве. Компилятор должен знать больше программиста — это и есть символ веры Rust. Мастер, который не даст тебе порезаться, потому что проверяет каждое движение лезвия прежде, чем оно тронется.

И здесь спираль доходит до вершины. Страуструп исправил одно. Каждый следующий исправлял одно или два. Rust попытался исправить всё разом — самый амбициозный виток за полвека, язык, построенный, чтобы закрыть все раны линии в одном теле. Шире, чем сюда, амбиция уже не дотянется. Спираль раскрутилась до предела. Шесть поколений, каждое поправляет предыдущее, каждая поправка укладывается в следующую — и паттерн проступил так отчётливо, что его пора назвать. А названный, он сам задаёт вопрос. Если все исправляют одно и то же и тянутся к одним и тем же ответам, то спираль не просто раскручивается всё шире. Она куда-то идёт.

2.7. Формула: паттерн, который повторяется неизбежно

Шесть разборов, шесть поколений, шесть убеждений — и за всеми ними один и тот же ход, повторённый с такой настойчивостью, что его пора назвать вслух. Каждый язык брал предыдущий, убирал то, что в нём болело, добавлял то, чего не хватало, и уступал кое-что своей области применения. Из этого складывается формула, по которой устроена вся линия:

Новый = Предыдущий — Ошибки + Решения ± Компромиссы домена.

Проверим её на пройденном. C++ взял C, почти ничего не убрал, добавил структуру — и заплатил сложностью. Java взяла C++, убрала ручную память и указатели, добавила безопасность и абстрактную машину, уступила деловой среде многословием. C# взял Java, не стал почти ничего убирать, добавил удобства — и заплатил привязкой к домену одной компании. Go взял всю отяжелевшую линию, убрал из неё наследование и исключения, добавил простоту и горутины, уступил масштабу. Kotlin и Dart взяли Java, убрали null и обряд, добавили безопасность и краткость, уступили — один платформе, другой области интерфейсов. Rust взял весь список ран сразу, убрал их все, добавил гарантии и заплатил крутизной подъёма. Формула выдержала все шесть подста-

новок без исключения.

У формулы есть и то, чего три её знака не показывают: что каждый язык оставил нетронутым и во что при этом верил. C ++ сохранил указатели и undefined behavior, потому что доверял программисту мощь. Java оставила null и многословие, рассудив, что памяти доверять нельзя, а null — можно. C# не тронул стену чужой экосистемы, считая себя не революцией, а лишь следующей итерацией. Go удержал nil и обряд проверки ошибок, исходя из веры, что лучше убрать, чем добавить. Kotlin и Dart примирились с границами выбранной стратегии, убеждённые, что прошлое можно чинить. Rust сохранил одну лишь крутизну подъёма — цену за символ веры, что компилятор знает больше человека. Разные слова, разные годы, разные компании — а ход один.

И если смотреть не на отдельный язык, а на всю линию, проступает одно непрерывное движение: от C к Rust тянется не россыпь несвязанных языков, а единый след, где каждый шаг отталкивается от предыдущего и уходит в ту же сторону. Это не шесть историй. Это одна история, рассказанная шесть раз.

Стоит вглядеться в знаки формулы. Минус и плюс — общий хребет: все убирают ошибки, все добавляют решения, и в этом они неотличимы. А вот последний член, ± компромиссы домена, — место, где живёт лицо каждого. Именно он объясняет, почему C# пахнет Microsoft, Dart — интерфейсами, Go — масштабом Google, почему у каждого языка

свои идиомы, своя культура, своё сообщество, свой характер. Хребет один, но угол наклона у каждого свой, и из этого угла вырастает всё, что мы принимаем за непохожесть. Здесь важно не оступиться в плоское «языки одинаковые» — они не одинаковы. Различие реально, но оно живёт в последнем члене, а не в первых двух. Общее — движение; уникальное — компромисс, которым каждый платит своей задаче. Под культурным слоем — один ход; в культурном слое — шесть разных лиц.

И раз есть формула, по ней можно считать вперёд. Возьмём нынешний край — Rust, закрывший всё ценой брутального подъёма, — и подставим в ту же формулу. Что уберёт следующий? Очевиднее всего — ту самую крутизну, борьбу с компилятором, цену входа. Что добавит? Те же гарантии, но достижимые без многолетней ломки мышления. Чем уступит? Своей области — как уступали все. Имени следующего языка мы не знаем, но его форму уже можем набросать: безопасность Rust без боли Rust, плюс компромисс под свою задачу. Паттерн стал не наблюдением, а законом, с которым можно работать как с расчётом, — и сама эта предсказуемость должна насторожить. Если будущее линии вычисляется по одной строке, значит, у линии есть не только прошлое, но и направление.

У этой формулы есть и другое свойство, от которого становится не по себе: в ней нет конечного звена. Всякий «Новый», едва родившись, мгновенно становится «Преды-

дущим» для следующего уравнения. Rust — не конец линии, а её сегодняшний вход. А значит, ни один язык не последний: каждый — лишь лучший на сегодня ответ, несущий в себе ещё не найденную собственную ошибку и ждущий того, кто посмотрит на него и скажет знакомое «я могу лучше». Спираль не останавливалась полвека именно потому, что у формулы нет неподвижной точки. Или всё же есть — и мы её просто пока не назвали?

А направление — это уже не про отдельные языки. Создатели не сговаривались. Они работали в разные десятилетия, в разных странах, для разных задач, и каждый считал, что строит своё. Но все убирали одни и те же ошибки и тянулись к одним и тем же решениям — к безопасности от пустоты, к ветвлению по форме, к гарантиям, вынесенным в компилятор. Идущие порознь, они пришли в одну сторону. Значит, они не разбредались — они сходились. И тогда спираль, которая виток за витком казалась раскручивающейся всё шире, оборачивается чем-то иным: не расширением без конца, а движением к точке.

Виток за витком спираль раскручивалась — каждый создатель смелее предыдущего, каждая амбиция шире, каждое «я могу лучше» громче прошлого, — и казалось, ей нет ни конца, ни центра, только вечное расширение наружу. Но когда последний виток дошёл до предела и шесть путей легли рядом, стало видно то, чего не видно было изнутри движения: у спирали есть центр. Все эти годы она раскручивалась

не наружу, а внутрь — к одной точке, которую никто из шестерых не назвал, потому что каждый видел лишь свой виток.

Что это за точка — и сходится ли спираль к ней по-настоящему или это обман перспективы? Виток назвать легко. Назвать центр — труднее.

Глава 3. Другие ветви

Языки, которые не победили, — но чьи идеи победили за них

Тезис о едином дереве удобно проверять на C, Java, Go: они похожи почти на глаз, и родство тут не нужно доказывать — оно бросается в глаза. Но честная мысль проверяется не там, где она очевидна, а там, где она трещит. А трещит она на языках, где мышление устроено иначе от самого основания. Где переменная не меняется. Где цикл — почти признание в слабости. Где программа не отдаёт машине команды, а описывает истину и ждёт, пока машина сама догадается, как её добыть. Если все языки — ветви одного дерева, то именно здесь, на этих ветвях, тезис должен либо подтвердиться, либо рассыпаться.

Разгадка в том, что судьба языка и судьба его идеи — две разные судьбы. Язык — это организм: ему нужна экосистема, библиотеки, инструменты, нанимаемые люди, чья-то корпорация за спиной. Без этого он не проходит индустриальный порог, как бы хороши ни были его мысли. Но идея порога не знает. Идея не нуждается в библиотеках. Она умеет покинуть умирающее тело и перебраться в чужое, более крепкое, — и продолжить жить там под другим именем, в другой оболочке, нередко даже не упоминая, откуда пришла. Язык может проиграть начисто и при этом победить всем,

что он думал.

И тогда дерево перестаёт быть одной линией с боковыми отростками. Его корни уходят в почву глубже, чем видно с поверхности, — туда, где растут совсем не С-подобные языки, и питают ствол идеями, найденными вдали от него. Чтобы это увидеть, придётся спуститься к корням.

Конец ознакомительного фрагмента.