С. Бакина, Е. Зуев
Университет Иннополис

Как создавать языки программирования: идеальная модель

В статье рассматривается феномен языков программирования, которые в настоящее время служат важнейшими инструментами разработки компьютерных программ. Вводится и обосновывается трехуровневая модель языков программирования, состоящая из научного, технологического и социального измерений. Подробно рассматриваются основные этапы создания и существования языка в аспекте указанных трех измерений. Делается вывод о важности учета всех этих аспектов в процессе проектирования новых языков.

Предлагаемая статья задумана как первая в серии публикаций, рассматривающих феномен языка программирования с различных аспектов. В последующих статьях предполагается обсудить базовые требования к «идеальному» языку, принципиальные технологические аспекты его реализации, направления его социального бытования, а также влияние новых технологий, прежде всего ИИ, на различные аспекты проектирования и использования языков.

Введение

Создание языка программирования обычно описывают как последовательность технических действий. Осознается и формулируется идея, проектируются синтаксис и семантика, разрабатывается компилятор или интерпретатор [8], создаются библиотеки, подготавливается документация, публикуется первая версия — и язык считается созданным. Такое представление удобно, потому что оно позволяет увидеть в языке инженерный артефакт, который можно спроектировать, реализовать и запустить. Однако именно эта ясность и оказывается обманчивой.

Язык программирования не возникает только в синтаксисе и не исчерпывается своей первой реализацией. Он начинается раньше — в момент, когда существующие способы организации вычисления начинают восприниматься как недостаточные, — и продолжается дольше, чем живет его первая версия. Язык не просто описывает вычисление. Он задает, что именно считается вычислением, какие преобразования допустимы, где локализуется корректность, кто несет ответственность за безопасность программы и каким образом все это становится воспроизводимой коллективной практикой. Поэтому анализ языкового проектирования требует более широкой рамки, чем чисто инженерное описание.

В этой статье мы исходим из того, что язык программирования необходимо рассматривать *в трех измерениях*.

Во-первых, язык – это ***научный* объект**: язык определяет модель вычисления, фиксирует допустимые преобразования и формализует инварианты корректности.

Во-вторых, язык представляет собой ***технологический*** **объект**: язык реализуется в компиляторах, средах поддержки выполнения, системах типов, библиотеках, пакетных менеджерах и инструментах разработки.

В-третьих, это ***социальный*** **объект**: язык используется и поддерживается сообществами, закрепляется в образовательных программах, входит в рынок труда, становится частью профессиональной идентичности и коллективной культуры программирования [14].

Такая трехуровневая модель полезна не только для анализа уже сложившихся языков. Её главная ценность проявляется в том, что она позволяет по-новому понять сам процесс возникновения и бытования языка и тем самым предложить достаточно полный алгоритм создания и успешного жизненного цикла языков программирования.

Если рассматривать язык одновременно как научный, технологический и социальный феномен, становится ясно, что каждый этап языкового проектирования должен включать не один, а сразу три типа решений. На каждом шаге необходимо ответить не только на вопрос, как язык должен выглядеть, но и на вопросы, какую модель вычисления он утверждает, какой инженерной ценой она будет реализована и для какой модели коллективной практики она имеет смысл.

Таким образом, основной смысл настоящей статьи состоит в следующем утверждении: успешный язык программирования создается не тогда, когда удачно придуманы его синтаксис и реализация, а когда научное, технологическое и социальное измерения оказываются согласованными между собой. Язык оказывается сильным не потому, что он «лучший вообще», а потому, что его формальная модель, инженерная реализация и социальный режим воспроизводства поддерживают друг друга [12]. Именно поэтому предлагаемую трехуровневую модель следует понимать не как внешний комментарий к языку, а как внутреннюю логику самого языкового проекта.

Создание языка начинается не с синтаксиса, а с дефицита

Ни один сколько-нибудь значимый язык программирования не начинается с вопроса о том, какими будут его скобки, служебные слова или форма объявления функций. Он начинается с более глубокого вопроса: чего именно не хватает в уже существующих языках, какие проблемы существующие языки не решают или решают недостаточно полно. До тех пор, пока ответы на эти вопросы не сформулированы, новый язык остается спонтанным проектом, а не необходимой реакцией на реальные проблемы.

Этот исходный дефицит может выражаться по-разному. Он может иметь научную природу: существующие языки не позволяют выразить важные вычислительные инварианты, слабо поддерживают формальные гарантии или навязывают такую модель вычисления, которая мешает строгому рассуждению о корректности.

В других случаях дефицит имеет технологический характер: существующие языки оказываются неудобны не потому, что они формально невыразительны, а потому, что цена их использования слишком высока: трудно безопасно работать с памятью [1, 2], велик объем внешней дисциплины [3, 4], тяжело масштабируется код, слишком дорого обходятся ошибки.

Наконец, дефицит может быть социальным. Существующие языки могут быть трудны для обучения новичков и коммуникации между разработчиками, программы – быть трудно переносимыми между прикладными областями, структура и особенности языка могут быть не соответствующими складывающейся профессиональной культуре или слишком дорогими с точки зрения входа в практику [14].

Именно на этом этапе трехуровневая модель перестает быть фоном и становится рабочим инструментом. Если проектировщик языка формулирует проблему только на одном уровне, он почти неизбежно получает неполный ответ. Язык, задуманный только как более элегантная формальная система, рискует оказаться технологически тяжеловесным и социально нишевым. Язык, задуманный только как инженерное улучшение, может не иметь собственной концептуальной целостности. Язык, ориентированный только на удобство и доступность, может оказаться слишком слабым в выражении критически важных инвариантов.

Поэтому первый шаг создания языка можно описать как *тройную диагностику дефицита*. Научное измерение отвечает на вопрос, какие аспекты вычисления остаются плохо формализованными или неудачно выраженными. Технологическое измерение показывает, где именно текущие инженерные издержки становятся чрезмерными. Социальное измерение уточняет, кто именно испытывает этот дефицит и в какой коллективной практике новый язык вообще должен стать значимым. Иначе говоря, язык рождается не из желания “создать еще один язык”, а из пересечения трех напряжений: формального, инженерного и социального.

Отсюда следует практический вывод: создание языка следует начинать не с проектирования формы, а с карты дефицитов. Если неясно, какие именно ограничения существующих средств проект пытается снять, то дальнейшие решения о модели вычисления, типовой дисциплине и реализации будут случайными.

Следующий шаг: что язык считает вычислением?

После того как сформулирован исходный дефицит, возникает более фундаментальный вопрос: что именно новый язык будет считать вычислением. На первый взгляд этот вопрос кажется излишне абстрактным, но именно здесь задается базовая философия языка. Будет ли вычисление пониматься как изменение состояния памяти, как взаимодействие объектов с состоянием и поведением, как редукция выражений, как вывод из системы ограничений, как поток реакций на события, как композиция эффектов? От ответа на этот вопрос

зависит не только стиль программирования, но и сам способ мыслить о программе.

Научное значение этого шага состоит в том, что здесь новый язык должен определить свою модель вычисления. Императивная традиция [1] делает центральным изменение состояния; функциональная [10] — преобразование выражений; логическая — вывод и удовлетворение ограничений. Это не просто разные способы записи одной и той же идеи. Разные вычислительные модели задают разные базовые интуиции о том, что является естественной единицей программы, как формулируется корректность, где локализуется состояние и какие инварианты становятся основными.

Но это решение одновременно является и технологическим. Выбранная модель вычисления неизбежно определяет и ограничивает архитектуру реализации. Язык, который строится вокруг прямой мутации состояния и близости к аппаратной архитектуре, естественным образом тяготеет к одним формам компиляции, представления памяти и управления ресурсами. Язык, который делает центральными чистоту, композиционность и выразительную систему типов, требует другой внутренней инфраструктуры: иных промежуточных представлений, иной роли runtime support, иной стоимости абстракций. Иными словами, уже на уровне выбора вычислительной модели создатель языка частично выбирает и его будущую инженерную судьбу.

Социальное значение этого шага не менее существенно. Выбрать, что язык считает вычислением – значит одновременно выбрать, какому стилю профессионального мышления он будет благоприятствовать. Одни сообщества воспринимают как естественное явное управление памятью и ресурсами, другие — декларативное описание преобразований, третьи — быстрый переход от идеи к работающему результату даже ценой более слабого статического контроля. Поэтому выбор вычислительной модели — это всегда не только теоретическое решение, но и решение о том, кого язык будет когнитивно поддерживать.

Именно здесь трехуровневая модель помогает избежать одной из типичных ошибок языкового проектирования. Нельзя просто взять привлекательную вычислительную идею и предположить, что все остальное “достроится” позже. Если новая модель вычисления плохо укладывается в ожидаемую инженерную среду или оказывается слишком чуждой практикам предполагаемого сообщества, язык получает внутренний разрыв уже в самом основании. Поэтому второй шаг создания нового языка — это не просто выбор парадигмы, а согласование формальной модели вычисления, ее реализуемости и ее когнитивной приемлемости.

Практический вывод отсюда можно сформулировать так: хороший языковой проект должен определять не только «что нового можно будет выразить» с его помощью, но и «какой образ вычисления станет в нем естественным» и «для кого эта естественность действительно будет таковой».

Язык задает не только выразительность, но и границы допустимого

После определения модели вычисления, начинается этап, который чаще всего и ассоциируется с собственно проектированием языка. Здесь возникает синтаксис, семантические инварианты и правила, включая архитектуру типов, формы абстракции, механизмы композиции, правила работы с ресурсами.

Именно на этом этапе важно отказаться от упрощенного представления, будто язык проектируется прежде всего как пространство возможностей. Не менее важно то, что язык одновременно задает *пространство запретов*.

Любой язык определяет не только то, что можно выразить, но и то, какие состояния считаются нормальными, а какие исключаются, затрудняются или выносятся за пределы «естественного» программирования. Научное измерение этого этапа связано с тем, какие свойства вообще признаются достойными формализации. Система типов в этом смысле не является только средством классификации значений. Она задает, какие инварианты разработчики языка считают настолько существенными, что готовы встроить их в саму структуру допустимых программ.

Именно поэтому различия, например, между Rust [6, 7] и Python [3–5] следует понимать глубже, нежели просто как различие между статической и динамической типизацией. В Rust типовая система выступает носителем не только логической, но и ресурсной дисциплины. Владение, заимствование и ограничения на изменяемый доступ означают, что некоторые потенциально опасные состояния исключаются заранее и не признаются нормальной частью мира программы. В Python, напротив, в течение долгого времени язык строил свою силу на гораздо более широком пространстве программ, допустимых к запуску. Это снижало входной барьер, способствовало его популярности, ускоряло прототипирование и делало язык исключительно гибким, но одновременно сдвигало часть ответственности на runtime, тестирование, внешние анализаторы и командные соглашения.

Технологическое измерение этого этапа проявляется в том, какие проверки будут встроены в компилятор [8], какие останутся делом интерпретатора или runtime, а какие будут переложены на внешние инструменты. Проектирование языка здесь превращается в проектирование режимов обнаружения ошибок, распределения ответственности и стоимости некорректных состояний. Формальный запрет — это всегда еще и инженерное решение: он изменяет не только теорию языка, но и повседневный опыт программирования в нем.

Социальное измерение на этом шаге особенно легко недооценить. Исключительно важно, что язык проектируется не только для машины, но и для разработчиков программ. Читаемость, лаконичность, ясность сообщений об ошибках, степень «психологической переносимости» правил языка, эстетика кода [4] — все это влияет на то, как язык будет осваиваться и восприниматься. Язык с чрезвычайно сильными встроенными ограничениями может оказаться трудным для массового обучения; язык с очень слабой дисциплиной может быть удобным на входе, но дорого стоить командам на длинной дистанции жизненного цикла.

Поэтому третий шаг создания языка разумно понимать как проектирование границ: что язык считает корректным по умолчанию, какие опасности он стремится исключить заранее, а какие допускает ради гибкости, скорости входа или выразительности. Практический вывод здесь особенно важен: проектировать язык — значит проектировать не только его возможности, но и режим распределения риска.

Язык успешен, когда его идеи платят инженерную цену

До тех пор, пока язык существует как спецификация, набор принципов или интеллектуально убедительная теория, он остается обещанием. Он может быть концептуально сильным и даже формально элегантным, но еще не быть реальной вычислительной средой. Решающий перелом происходит, когда язык начинает существовать технологически: в компиляторе и его внутренних структурах, поддержке времени выполнения, механизмах и возможностях оптимизации, модульной архитектуре и инструментах диагностики.

Именно здесь трехуровневая модель важна как метод оценки, а не только описания. Научная сила языка еще не гарантирует его технологической жизнеспособности. Формально привлекательные решения могут требовать слишком дорогой реализации, сложной инфраструктуры, тяжелого runtime или высокой стоимости поддержки инструментов. История языков многократно показывает, что две одинаково красивые на бумаге языковые идеи могут иметь совершенно разную инженерную судьбу. Характерен, например, контраст Smalltalk и Java. Smalltalk предложил чрезвычайно цельную и концептуально элегантную объектную модель, в которой практически все мыслилось как объект и сообщение [15], однако эта чистота предполагала специфическую среду исполнения и особую инструментальную инфраструктуру [15]. Java, сохранив объектную ориентацию как центральный принцип, отказалась от такой радикальной цельности, но выиграла в стандартизации, переносимости, предсказуемости реализации [16], а так же институциональной поддержке [13], что обеспечило ей гораздо более широкую инженерную судьбу.

Научное измерение на этом шаге проявляется в том, как именно формальные идеи материализуются. Если язык утверждает определенную модель вычисления, эта модель должна получить операционную или компиляционную поддержку. Семантика должна стать траекторией перевода программы [8] в исполняемый объект. Типовая дисциплина должна стать механизмом реальной локализации ошибок. Если язык фиксирует инварианты, он должен уметь превратить их из теоретических утверждений в практические ограничения поведения программы.

Технологическое измерение здесь, разумеется, центрально. Как будет устроена цепочка трансляции? Будет ли язык компилироваться в нативный код или в байткод [8] виртуальной машины? Будут ли (должны ли) использоваться продвинутые технологии вроде JIT [9]? Насколько тяжела технологическая цена абстракций? Где будет лежать основная стоимость — в этапе компиляции, в исполнении, в управлении памятью, в монополизации инстанцирования типов, в работе runtime? Именно на этом уровне становится

видно, что язык — это не просто набор конструкций, а особый режим существования программы.

Социальное измерение тоже включено в этот этап гораздо сильнее, чем может показаться. Разработчик инструментария взаимодействует не с «чистой семантикой», а с конкретным опытом использования языка: скоростью сборки, качеством сообщений компилятора, предсказуемостью runtime, поведением пакетного менеджера, устойчивостью стандартной библиотеки, удобством диагностики. То, что было научным и технологическим решением, здесь становится переживаемой практикой. А переживаемая практика — это уже социальный факт, потому что именно через нее формируются доверие к языку, репутация его надежности и готовность сообществ инвестировать [13] в его освоение.

Следовательно, четвертый шаг создания языка — это не просто «написать компилятор». Это этап, на котором формальная модель впервые проходит проверку на инженерную выполнимость и социальную переносимость. Практический вывод прост: языковой проект нельзя оценивать только по элегантности его идей; его нужно оценивать и по тому, как эти идеи переживаются в реальном цикле разработки.

Язык становится успешным только как экосистема

Даже после появления компилятора или интерпретатора язык еще не завершен. Им можно воспользоваться, но в нем еще трудно жить. Настоящая жизнеспособность начинается не в момент публикации первой версии, а в момент появления устойчивой среды обитания: пакетного менеджера, документации, линтеров, форматтеров, тестовых инструментов, интеграции в IDE, библиотек, стандартных идиом и привычных способов решения типовых задач.

Научное измерение этого этапа часто не замечают, хотя оно присутствует и здесь. Экосистема не является лишь внешним слоем вокруг уже готового языка. Она продолжает интерпретировать и конкретизировать то, что язык считает естественным программированием. Какие абстракции получают стандартную библиотеку, а какие остаются частью ядра языка? Какие паттерны становятся каноническими? Какие решения считаются “идиоматичными”? Иными словами, экосистема закрепляет практическую интерпретацию формальной модели языка.

Технологическое измерение на этом шаге очевидно. Без зрелых библиотек и стабильной инструментальной среды язык остается экспериментом, а не платформой для работы. Даже очень сильный язык проигрывает, если переход от идеи к результату в нем слишком дорог. Именно поэтому Python приобрел исключительную практическую плотность: его выбирают не по причине «простого» синтаксиса и не только из-за сравнительно высокой скорости написания кода, а потому, что экосистема резко снижает стоимость входа в практику и ускоряет получение результата.

Социальное измерение здесь становится центральным. Экосистема — это институционализированная форма коллективной поддержки языка. Она создается не одной командой, а распределенной сетью участников: авторами библиотек, преподавателями, корпорациями, командами сопровождения программных систем, авторами документации и пользователями, которые воспроизводят язык в реальных задачах. В этом смысле экосистема — не приложение к языку, а один из главных механизмов его исторической устойчивости.

Практический вывод отсюда важен для самого замысла языкового проектирования. Создавать язык — значит создавать не только синтаксис и реализацию, но и обитаемую среду. Если на этапе проектирования не думать о том, какие типовые задачи в нем будут решаться, как в нем будет происходить вход новичка, как будет выглядеть пакетная инфраструктура и как язык станет частью повседневной инженерной жизни, то даже удачная формальная конструкция рискует остаться локальным экспериментом.

Языковой проект становится завершенным только через социальное закрепление

Наиболее важная, но часто недооцениваемая часть истории языка начинается тогда, когда он выходит за пределы своей исходной команды разработчиков. В этот момент язык перестает быть только проектом и начинает становиться институтом. У него появляется сообщество, складываются представления о “хорошем” и “плохом” стиле, возникают общественные и корпоративные нормы code review, style guides и стандарты, начинается борьба за обратную совместимость, появляются учебные курсы, вакансии и профессиональные траектории.

Этот этап нельзя считать чем-то внешним по отношению к языку. Социальное закрепление — это не постфактум добавленное окружение, а продолжение самого языкового проектирования. Научные и технологические свойства языка только тогда становятся исторически значимыми, когда они получают форму коллективно разделяемых ожиданий. Иначе говоря, язык начинает жить по-настоящему не тогда, когда доказана сила его модели, а когда вокруг этой силы возникает воспроизводимая практика доверия.

Научное измерение здесь сохраняется в виде репутации языка как носителя определенного способа мыслить о вычислении. Одни языки начинают ассоциироваться с формальной строгостью, другие — с инженерной надежностью, третьи — с гибкостью и скоростью входа. Эти репутации не сводятся к реальным свойствам языка, но и не отрываются от них: они являются их социальной переработкой.

Технологическое измерение закрепляется в форме доверия к конкретным инженерным обещаниям языка. Так, рост Rust объясняется не только тем, что он предлагает формальную модель безопасности памяти [7], но и тем, что эта модель была технологически реализована [6] и затем социально понята как обещание надежной современной инженерии. Напротив, история Haskell [10]

показывает, что глубокое теоретическое влияние и даже технологическая оригинальность еще не гарантируют массового институционального закрепления. Между формальной силой языка и его широким распространением всегда лежит труд обучения, адаптации, экосистемного роста и культурному признанию.

Социальное измерение на этом этапе становится решающим. Язык считается по-настоящему созданным не тогда, когда опубликована его спецификация, а тогда, когда им можно учить, на нем можно нанимать, вокруг него можно строить карьеру, с ним можно входить в профессию и передавать его между поколениями разработчиков.

В этот момент язык перестает быть только техническим средством и становится устойчивой формой коллективной координации.

Практический вывод отсюда состоит в следующем: проектирование языка должно включать проектирование его институциональной траектории. Если язык невозможно преподавать, вокруг него трудно формировать сообщество, у него нет понятного пути в профессию и нет культурно различимой идентичности, то его создание остается незавершенным.

Трехуровневая модель как метод языкового проектирования

До сих пор трехуровневая модель рассматривалась как средство объяснить, что на самом деле происходит на разных этапах создания языка. Однако ее значение этим не исчерпывается. Она полезна не только как аналитическая рамка, но и как практический метод улучшения самого процесса языкового проектирования.

Если применять эту модель последовательно, каждый этап создания языка начинает выглядеть иначе. На этапе постановки проблемы она заставляет различать, является ли дефицит формальным, инженерным или социальным, и не подменять один другим. На этапе выбора вычислительной модели она требует согласовывать теоретическую новизну с реализуемостью и когнитивной приемлемостью. На этапе проектирования ограничений она помогает увидеть, что речь идет не только о выразительности, но и о распределении риска и ответственности. На этапе реализации она не позволяет отделить элегантность идеи от цены ее компиляционной и runtime-материализации. На этапе построения экосистемы она заставляет рассматривать инструменты и библиотеки как часть самого языка. Наконец, на этапе социального закрепления она показывает, что язык должен быть не только сильным, но и передаваемым, обучаемым и институционально устойчивым.

Именно здесь становится особенно заметной практическая полезность проведенного рассмотрения. Трехуровневая модель позволяет не просто ретроспективно описывать историю удачных и неудачных языков, а выявлять точки разрыва в языковых проектах еще до того, как эти проекты столкнутся с реальностью. Если в языке сильна научная составляющая, но слабо продумана экосистема, можно заранее ожидать риск нишевости. Если язык

технологически убедителен, но не имеет отчетливой концептуальной позиции, можно ожидать распада проекта на набор ситуативных решений. Если язык социально доступен, но слишком многое переносит во внешнюю дисциплину, можно заранее увидеть будущую цену этой гибкости на уровне качества и поддержки.

Из этого следует один из главных выводов статьи: создать «лучший» язык программирования можно не за счет “еще более сильной” работы только на одном из уровней, а за счет согласования решений между всеми тремя. Язык становится лучше не когда он просто строже, быстрее или популярнее, а тогда, когда его формальная модель, технологическая реализация и социальная траектория взаимно усиливают друг друга.

Заключение

Создание языка программирования обычно представляют как инженерное конструирование: проектирование синтаксиса, реализация компилятора, выпуск первой версии. Однако более внимательный анализ показывает, что язык возникает в гораздо более сложной динамике. Сначала он появляется как ответ на тройной дефицит — формальный, технологический и социальный. Затем он оформляется как определенная модель вычисления. После этого он задает границы допустимого, распределяет корректность и ответственность, материализуется в компиляторах, runtime-средах и инструментах, обрастает экосистемой и либо закрепляется как коллективный институт, либо остается локальным экспериментом.

Иными словами, язык создается не в одной точке и не одним актом. Он создается как минимум дважды: сначала как формально-технологическая конструкция, затем как социально воспроизводимая практика. Можно сказать еще сильнее: язык создается непрерывно, потому что социальное закрепление возвращается к его научным и технологическим основаниям и начинает заново перестраивать их через требования эргономики, стабильности, обучаемости, обратной совместимости и коллективного доверия.

Предложенная трехуровневая модель позволяет увидеть в этом процессе не просто три стороны одного объекта, а внутреннюю логику самого языкового проекта. Научное измерение дает языку концептуальный скелет. Технологическое — делает этот скелет исполнимым телом. Социальное — придает ему историческую продолжительность и превращает его в воспроизводимую форму коллективной практики. Только в их согласовании язык действительно становится языком программирования, а не просто красивой идеей, удачной инженерной системой или случайно популярным инструментом.

Поэтому, если мы хотим понимать, как создаются языки программирования и как делать их лучше, недостаточно спрашивать, как придумать синтаксис или как написать компилятор. Нужно ответить, какую модель вычисления язык утверждает, какой инженерной ценой эта модель реализуется и каким образом коллективная практика делает ее устойчивой, передаваемой и значимой.

Именно в этом смысле язык программирования следует понимать не как нейтральный контейнер для алгоритмов, а как *культурный феномен, возникающий на пересечении науки, технологий и социума*.

В заключение отметим, что данная статья задумана как первая в серии публикаций, рассматривающих феномен языка программирования с различных сторон. В последующих статьях предполагается обсудить базовые требования к «идеальному» (в смысле рассмотренных в данной статье аспектов) языку, а также принципиальные технологические аспекты его реализации и направления его социализации.

Список источников

1. Kernighan B. W., Ritchie D. M. The C Programming Language. 2nd ed. Englewood Cliffs: Prentice Hall, 1988. 274 p.
2. Stroustrup B. The C++ Programming Language. 4th ed. Boston: Addison-Wesley, 2013. 1366 p.
3. Van Rossum G., Drake F. L. The Python Language Reference [Электронный ресурс]. Python Software Foundation. URL: https://docs.python.org/3/reference/ .
4. Peters T. PEP 20 – The Zen of Python [Электронный ресурс]. Python Enhancement Proposals, 2004. URL: https://peps.python.org/pep-0020/ .
5. Van Rossum G., Lehtosalo J., Langa Ł. PEP 484 – Type Hints [Электронный ресурс]. Python Enhancement Proposals, 2014–2015. URL: https://peps.python.org/pep-0484/ .
6. Klabnik S., Nichols C. The Rust Programming Language. San Francisco: No Starch Press, 2018. URL: https://doc.rust-lang.org/book/ .
7. Jung R., Jourdan J.-H., Krebbers R., Dreyer D. RustBelt: Securing the Foundations of the Rust Programming Language // Proc. ACM Program. Lang. 2018. Vol. 2, POPL. Art. 66. DOI: https://doi.org/10.1145/3158154
8. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: Principles, Techniques, and Tools. 2nd ed. Boston: Pearson, 2006. 1040 p.
9. Würthinger T., Wimmer C., Wöß A. [и др.] One VM to Rule Them All // Proc. ACM Int. Symp. New Ideas, New Paradigms, Reflections Program. Softw. (Onward! 2013). 2013. P. 187–204. DOI: https://doi.org/10.1145/2509578.2509581
10. Peyton Jones S., ed. Haskell 98 Language and Libraries: The Revised Report. Cambridge: Cambridge Univ. Press, 2003. 255 p. URL: https://www.haskell.org/onlinereport/
11. *(опционально)* Wadler P. Monads for Functional Programming // Advanced Functional Programming / ed. J. Jeuring, E. Meijer. Berlin; Heidelberg: Springer, 1995. (LNCS; vol. 925). P. 24–52. DOI: https://doi.org/10.1007/3-540-59451-5_2
12. Petricek T. Cultures of Programming: Understanding the History of Programming Through Controversies and Technical Artifacts // Proc. ACM Program. Lang. 2019. Vol. 3. DOI: https://doi.org/10.1145/3360318
13. Meyerovich L. A., Rabkin A. S. Empirical Analysis of Programming Language Adoption // Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA). 2013. P. 1–18. DOI: https://doi.org/10.1145/2541928.2541929
14. Wenger E. Communities of Practice: Learning, Meaning, and Identity. Cambridge: Cambridge Univ. Press, 1998. 318 p.

15. 15. Kay A. C. The Early History of Smalltalk // Proc. 2nd ACM SIGPLAN Conf. History Program. Lang. (HOPL-II). 1993. P. 69–95.
DOI: https://doi.org/10.1145/154766.155364
16. 16. Gosling J. The Java Language Environment: A White Paper [Электронный ресурс]. Sun Microsystems, 1995.
URL: https://www.stroustrup.com/1995_Java_whitepaper.pdf.