

# Язык программирования

Материал из Википедии — свободной энциклопедии

**Язы́к программи́рования** — формальный язык, предназначенный для записи компьютерных программ<sup>[1][2]</sup>. Язык программирования определяет набор лексических, синтаксических и семантических правил, определяющих внешний вид программы и действия, которые выполнит исполнитель (обычно — ЭВМ) под её управлением.

Со времени создания первых программируемых машин человечество придумало более восьми тысяч языков программирования (включая эзотерические, визуальные и игрушечные) <sup>[3]</sup>. Каждый год их число увеличивается. Некоторыми языками умеет пользоваться только небольшое число их собственных разработчиков, другие становятся известны миллионам людей. Профессиональные программисты могут владеть несколькими языками программирования.

Язык программирования предназначен для написания компьютерных программ, которые представляют собой набор правил, позволяющих компьютеру выполнить тот или иной вычислительный процесс, организовать управление различными объектами, и т. п. Язык программирования отличается от естественных языков тем, что предназначен для управления ЭВМ, в то время как естественные языки используются, прежде всего, для общения людей между собой. Большинство языков программирования использует специальные конструкции для определения и манипулирования структурами данных и управления процессом вычислений.

Как правило, язык программирования определяется не только через спецификации *стандарта языка*, формально определяющие его синтаксис и семантику , но и через *воплощения (реализации) стандарта* — программные средства, обеспечивающих трансляцию или интерпретацию программ на этом языке ; такие программные средства различаются по производителю, марке и варианту (версии), времени выпуска, полноте воплощения стандарта, дополнительным возможностям; могут иметь определённые ошибки или особенности воплощения, влияющие на практику использования языка или даже на его стандарт.

<b>Содержание</b>
<b>История</b>
<u>Ранние этапы развития</u>
<u>Совершенствование</u>
<u>Объединение и развитие</u>
<b>Спецификация языков</b>
<u>Стандартизация</u>
<u>Алфавит</u>
<u>Грамматика</u>
<u>Семантика</u>
<b>Классификация</b>
<u>Языки низкого и высокого уровня</u>
<u>Безопасные и небезопасные языки</u>

[Компилируемые, интерпретируемые и встраиваемые языки](#)

[Языки первого и высшего порядка](#)

[Начальные сведения](#)

[Выразительность](#)

[Изучение](#)

[Парадигма программирования](#)

[Языки для программирования в мелком и крупном масштабе](#)

[Концептуальная целостность языков](#)

[Особые категории языков](#)

**[Формальные преобразования и оптимизация](#)**

**[Популярность языков](#)**

**[См. также](#)**

**[Примечания](#)**

**[Литература](#)**

**[Ссылки](#)**

## История

---

### Ранние этапы развития

Можно сказать, что первые языки программирования возникали ещё до появления современных электронных вычислительных машин: уже в XIX веке были изобретены устройства, которые можно с долей условности назвать программируемыми — к примеру, музыкальная шкатулка (и позднее механическое пианино) посредством металлического цилиндра и Жаккардовый ткацкий станок (1804) посредством картонных карт. Для управления ими использовались наборы инструкций, которые в рамках современной классификации можно считать прототипами предметно-ориентированных языков программирования. Значимым можно считать «язык», на котором леди Ада Августа (графиня Лавлейс) в 1842 году написала программу для вычисления чисел Бернулли для аналитической машины Чарльза Бэббиджа, ставшей бы, в случае реализации, первым компьютером в мире, хотя и механическим — с паровым двигателем.

В 1930—1940 годах А. Чёрч, А. Тьюринг, А. Марков разработали математические абстракции (лямбда-исчисление, машину Тьюринга, нормальные алгоритмы соответственно) — для формализации алгоритмов.

В это же время, в 1940-е годы, появились электрические цифровые компьютеры и был разработан язык, который можно считать первым высокоуровневым языком программирования для ЭВМ — «Plankalkül», созданный немецким инженером К. Цузе в период с 1943 по 1945 годы<sup>[4]</sup>.

Программисты ЭВМ начала 1950-х годов, в особенности таких, как UNIVAC и IBM 701, при создании программ пользовались непосредственно машинным кодом, запись программы на котором состояла из единиц и нулей и который принято считать языком программирования первого поколения (при этом разные машины разных производителей использовали различные коды, что требовало переписывать программу при переходе на другую ЭВМ).

Первым практически реализованным языком стал в 1949 году так называемый «Краткий код», в котором операции и переменные кодировались двухсимвольными сочетаниями. Он был разработан в компании Eckert–Mauchly Computer Corporation, выпускавшей UNIVAC-и, созданной одним из сотрудников Тьюринга, Джоном Мокли. Мокли поручил своим сотрудникам разработать транслятор математических формул, однако для 1940-х годов эта цель была слишком амбициозна. Краткий код был реализован с помощью интерпретатора<sup>[5]</sup>.

Вскоре на смену такому методу программирования пришло применение языков второго поколения, также ограниченных спецификациями конкретных машин, но более простых для использования человеком за счёт использования мнемоник (символьных обозначений машинных команд) и возможности сопоставления имён адресам в машинной памяти. Они традиционно известны под наименованием языков ассемблера и автокодов. Однако при использовании ассемблера становился необходимым процесс перевода программы на язык машинных кодов перед её выполнением, для чего были разработаны специальные программы, также получившие название ассемблеров. Сохранились и проблемы с переносимостью программы с ЭВМ одной архитектуры на другую, и необходимость для программиста при решении задачи мыслить терминами «низкого уровня» — ячейка, адрес, команда. Позднее языки второго поколения были усовершенствованы: в них появилась поддержка макрокоманд.

С середины 1950-х начали появляться языки третьего поколения, такие как Фортран, Лисп и Кобол<sup>[6]</sup>. Языки программирования этого типа более абстрактны (их ещё называют «языками высокого уровня») и универсальны, не имеют жёсткой зависимости от конкретной аппаратной платформы и используемых на ней машинных команд. Программа на языке высокого уровня может исполняться (по крайней мере, в теории, на практике обычно имеется ряд специфических версий или диалектов реализации языка) на любой ЭВМ, на которой для этого языка имеется транслятор (инструмент, переводящий программу на язык машины, после чего она может быть выполнена процессором).

Обновлённые версии перечисленных языков до сих пор имеют хождение в разработке программного обеспечения, и каждый из них оказал определённое влияние на последующее развитие языков программирования<sup>[7]</sup>. Тогда же, в конце 1950-х годов, появился Алгол, также послуживший основой для ряда дальнейших разработок в этой сфере. Необходимо заметить, что на формат и применение ранних языков программирования в значительной степени влияли интерфейсные ограничения<sup>[8]</sup>.

## Совершенствование

В период 1960-х — 1970-х годов были разработаны основные парадигмы языков программирования, используемые в настоящее время, хотя во многих аспектах этот процесс представлял собой лишь улучшение идей и концепций, заложенных ещё в первых языках третьего поколения.

- Язык APL оказал влияние на функциональное программирование и стал первым языком, поддерживавшим обработку массивов<sup>[9]</sup>.
- Язык ПЛ/1 (NPL) был разработан в 1960-х годах как объединение лучших черт Фортрана и Кобола.
- Язык Snobol, разработанный и совершенствуемый в течение 1960-х годов, ориентированный на обработку текстов, ввёл в число базовых операций языков программирования сопоставление с образцом<sup>[10][11][12]</sup>.
- Язык Симула, появившийся примерно в это же время, впервые включал поддержку объектно-ориентированного программирования. В середине 1970-х группа специалистов представила язык Smalltalk, который был уже всецело объектно-ориентированным.

- В период с 1969 по 1973 годы велась разработка языка Си, популярного и по сей день<sup>[13]</sup> и ставшего основой для множества последующих языков, например, столь популярных, как C++ и Java.
- В 1972 году был создан Пролог — наиболее известный (хотя и не первый, и далеко не единственный) язык логического программирования.
- В 1973 году в языке ML была реализована расширенная система полиморфной типизации, положившая начало типизированным языкам функционального программирования.

Каждый из этих языков породил по семейству потомков, и большинство современных языков программирования в конечном счёте основано на одном из них.

Кроме того, в 1960—1970-х годах активно велись споры о необходимости поддержки структурного программирования в тех или иных языках<sup>[14]</sup>. В частности, голландский специалист Э. Дейкстра выступал в печати с предложениями о полном отказе от использования инструкций GOTO во всех высокоуровневых языках. Развивались также приёмы, направленные на сокращение объёма программ и повышение продуктивности работы программиста и пользователя.

## Объединение и развитие

В 1980-е годы наступил период, который можно условно назвать временем консолидации. Язык C++ объединил в себе черты объектно-ориентированного и системного программирования, правительство США стандартизировало язык Ада, производный от Паскаля и предназначенный для использования в бортовых системах управления военными объектами, в Японии и других странах мира осуществлялись значительные инвестиции в изучение перспектив так называемых языков пятого поколения, которые включали бы в себя конструкции логического программирования<sup>[15]</sup>. Сообщество функциональных языков приняло в качестве стандарта ML и Лисп. В целом этот период характеризовался скорее опорой на заложенный в предыдущем десятилетии фундамент, нежели разработкой новых парадигм.

Важной тенденцией, которая наблюдалась в разработке языков программирования для крупномасштабных систем, было сосредоточение на применении модулей — объёмных единиц организации кода. Хотя некоторые языки, такие, как ПЛ/1, уже поддерживали соответствующую функциональность, модульная система нашла своё отражение и применение также и в языках Модула-2, Оберон, Ада и ML. Часто модульные системы объединялись с конструкциями обобщённого программирования<sup>[16]</sup>.

Важным направлением работ становятся визуальные (графические) языки программирования, в которых процесс «написания» программы как текста заменяется на процесс «рисования» (конструирования программы в виде диаграммы) на экране ЭВМ. Визуальные языки обеспечивают наглядность и лучшее восприятие логики программы человеком.

В 1990-х годах в связи с активным развитием Интернета распространение получили языки, позволяющие создавать сценарии для веб-страниц — главным образом Perl, развившийся из скриптового инструмента для Unix-систем, и Java. Возрастала также и популярность технологий виртуализации. Эти изменения, однако, также не представляли собой фундаментальных новаций, являясь скорее совершенствованием уже существовавших парадигм и языков (в последнем случае — главным образом семейства Си).

В настоящее время развитие языков программирования идёт в направлении повышения безопасности и надёжности, создания новых форм модульной организации кода и интеграции с базами данных.

# Спецификация языков

---

## Стандартизация

Для многих широко распространённых языков программирования созданы международные стандарты. Специальные организации проводят регулярное обновление и публикацию спецификаций и формальных определений соответствующего языка. В рамках таких комитетов продолжается разработка и модернизация языков программирования и решаются вопросы о расширении или поддержке уже существующих и новых языковых конструкций.

## Алфавит

Современные языки программирования рассчитаны на использование ASCII, то есть доступность всех *графических* символов ASCII является необходимым и достаточным условием для записи любых конструкций языка. *Управляющие* символы ASCII используются ограниченно: допускаются только возврат каретки CR, перевод строки LF и горизонтальная табуляция HT (иногда также вертикальная табуляция VT и переход к следующей странице FF).

Ранние языки, возникшие в эпоху 6-битных символов, использовали более ограниченный набор. Например, алфавит Фортрана включает 49 символов (включая пробел): A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 = + - \* / ( ) . , \$ ' :

Заметным исключением является язык APL, в котором используется очень много специальных символов.

Использование символов за пределами ASCII (например, символов KOI8-R или символов Юникода) зависит от реализации: иногда они разрешаются только в комментариях и символьных/строковых константах, а иногда и в идентификаторах. В СССР существовали языки, где все ключевые слова писались русскими буквами, но большой популярности подобные языки не завоевали (исключение составляет Встроенный язык программирования 1С:Предприятие).

Расширение набора используемых символов сдерживается тем, что многие проекты по разработке программного обеспечения являются международными. Очень сложно было бы работать с кодом, где имена одних переменных записаны русскими буквами, других — арабскими, а третьих — китайскими иероглифами. Вместе с тем, для работы с текстовыми данными языки программирования нового поколения (Delphi 2006, C#, Java) поддерживают Unicode.

## Грамматика

- Контекстно-свободная грамматика
- Контекстно-зависимая грамматика
- регулярный язык
  - регулярные выражения
- Грамматика с фразовой структурой
- LL(n)
- LALR(1)
- Yacc
- ANTLR

- [Parsec](http://www.haskell.org/haskellwiki/Parsec) (<http://www.haskell.org/haskellwiki/Parsec>)
- [AST](#)
- [Дерево разбора](#)
- [Абстрактный синтаксис первого порядка](#)
- [Абстрактный синтаксис высшего порядка](#)

## Семантика

Существует несколько подходов к определению семантики языков программирования. Основных три: [операционная](#), [аксиоматическая](#) и [денотационная](#).

- При описании семантики в рамках *операционного* подхода обычно исполнение конструкций языка программирования интерпретируется с помощью некоторой воображаемой (абстрактной) ЭВМ.
- *Аксиоматическая* семантика описывает последствия выполнения конструкций языка с помощью языка логики и задания пред- и постусловий.
- *Денотационная* семантика оперирует понятиями, типичными для математики — множества, соответствия, а также суждения, утверждения и др.

## Классификация

---

Не существует общепринятой систематичной таксономии языков программирования. Есть множество черт, согласно которым можно производить классификацию языков, причём одни из них однозначно проводят разделы между языками на основе технических свойств, другие основываются на доминирующих признаках, имеют исключения и более условны, а третьи полностью субъективны и нередко сопровождаются заблуждениями, но на практике весьма распространены.

Конкретный язык программирования в подавляющем большинстве случаев имеет более одного языка-предка. Многие языки создаются как сочетание элементов различных языков. В одних случаях такое сочетание проходит математический анализ на предмет непротиворечивости (см., например, [Определение Standard ML](#)), в других — язык формируется исходя из практических потребностей, для решения актуальных проблем с целью получения коммерческого успеха, но при этом без соблюдения математической строгости и с включением в язык взаимоисключающих идей (как в случае [C++](#)<sup>[17][18][19][20][21]</sup>).

## Языки низкого и высокого уровня

Обычно под «уровнем языка» понимается:

- степень отличия семантики языка от [машинного кода](#) целевой [архитектуры процессора](#) — другими словами, наименьший масштаб преобразований, которые должен претерпеть код программы перед тем, как он сможет исполняться (зачастую с существенной потерей эффективности)
- степень, в которой семантика языка учитывает особенности мышления человека, нежели машины — то есть уровень языка тем «ниже», чем он «ближе к машине», и тем «выше», чем он «ближе к человеку».

Эта двойственность появилась в 1950-е годы, при создании языков Планкалкюль и Фортран. При их разработке ставились прямые намерения обеспечить более краткую запись часто встречающихся конструкций (например, арифметических выражений), чем требовали процессоры того времени. В этих языках вводился новый слой абстракции и предполагались преобразования программ в машинный язык, поэтому их называли языками «высокого уровня», то есть надстройкой, надслоением над языком машины. Однако вскоре стало ясно, что эти определения вовсе не обязательно идут бок о бок. Так, история знает случаи, когда язык, традиционно считающийся «высокоуровневым», реализовывался аппаратно (см. Лисп-машина, Java Optimized Processor), или когда язык, являющийся «низкоуровневым» на одной платформе, компилировался как «высокоуровневый» на другой (таким образом программы на CISC-ассемблере VAX использовались на RISC-машинах DEC Alpha — см. VAX Macro). Таким образом, понятие уровня языка является не строго формальным, а скорее условным.

К языкам низкого уровня относят, в первую очередь, машинные языки (или, на общеупотребимом жаргоне — машинные коды), то есть языки, реализованные непосредственно на аппаратном уровне. Их относят к первому поколению языков программирования. Вскоре после них появились языки второго поколения — так называемые «языки ассемблера». В простейшем случае они реализуют мнемонику над машинным языком для записи команд и их параметров (в частности, адресов в памяти). Кроме того, многие языки ассемблера включают и весьма развитый макроязык. Языки *первого* и *второго* поколения позволяют точно контролировать, как требуемая функциональность будет исполняться на данном процессоре с учётом особенностей его архитектуры. С одной стороны, это обеспечивает высокое быстродействие и компактность программ, но с другой, для переноса программы на другую аппаратную платформу её нужно перекодировать (а часто из-за различий архитектуры процессоров — и перепроектировать) с нуля. Большинство языков ассемблера являются бестиповыми, но существуют и типизированные языки ассемблера, нацеленные на обеспечение минимальной безопасности низкоуровневых программ.

К 1970-м годам сложность программ выросла настолько, что превысила способность программистов управляться с ними, и это привело к огромным убыткам и застою в развитии информационных технологий<sup>[22]</sup>. Ответом на эту проблему стало появление массы языков высокого уровня, предлагающих самые разные способы управления сложностью (подробнее см. парадигма программирования и языки для программирования в мелком и крупном масштабе). Программы на языках «высокого уровня» гораздо легче модифицируются и совсем легко переносятся с компьютера на компьютер. На практике, наибольшее распространение получили языки *третьего* поколения, которые лишь *pretendуют* на звание «высокоуровневых», но реально предоставляют лишь те «высокоуровневые» конструкции, что находят однозначное соответствие инструкциям в машине фон Неймана<sup>[23]</sup>.

К языкам *четвёртого* поколения относят языки высшего порядка. Иногда выделяется категория языков *пятого* поколения, но она не является общепринятой — чаще используется термин «язык сверхвысокого уровня» (англ. *very high level language*). Это языки, реализация которых включает существенную алгоритмическую составляющую (то есть когда интерпретация небольшого исходного кода требует весьма сложных вычислений). Чаще всего так называют логические языки, про которые также говорят, что это просто языки четвёртого поколения, дополненные базой знаний<sup>[24]</sup>. Кроме того, к «языкам сверхвысокого уровня» относят визуальные языки и языки, основанные на подмножестве естественного языка (например, так называемой «деловой прозы»).

Важной категорией являются предметно-ориентированные языки (англ. *DSL — Domain Specific Language*). Отнесение языка к этой категории является весьма условным и зачастую спорным; на практике этот термин могут применять к представителям и третьего, и четвёртого, и пятого поколений языков. Порой так даже классифицируют язык Си, который можно отнести к поколению «2,5». Он изначально позиционировался как «высокоуровневый ассемблер»; его также часто называют «языком среднего уровня». Он позволяет в значительной степени контролировать *способ*

реализации алгоритма с учётом свойств, типичных для весьма большого числа аппаратных архитектур. Однако есть платформы, под которые реализации Си (даже в нестандартном виде) отсутствуют по причине принципиальной невозможности или нецелесообразности их создания. Со временем появились и другие языки среднего уровня, например, LLVM, C++.

Первые три поколения языков формируют императивную парадигму программирования, а последующие — декларативную<sup>[24]</sup>. Термин «императив» означает «приказной порядок», то есть программирование посредством *пошагового инструктирования* машины, или детального указания уже придуманного программистом *способа реализации* технического задания. Термин «декларатив» означает «описание», то есть программирование посредством предоставления *формализации* технического задания в виде, пригодном для автоматических преобразований, с предоставлением свободы выбора транслятору языка. Императивные языки нацелены на описание того, *как* получить результат, тогда как языки более высокого уровня нацелены на описание того, *что* требуется в результате. Поэтому первые называют *как-языками* (или языками, ориентированными на машину), а вторые — *что-языками* (или языками, ориентированными на человека). Для множества задач полностью автоматическое порождение по-настоящему эффективной реализации алгоритмически неразрешимо, так что на практике даже на *что-языках* нередко используются определённые алгоритмические ухищрения. Однако существуют методы получения эффективных реализаций из основанных на определении (реализаций «в лоб») — такие как изобретённая в СССР суперкомпиляция.

В большинстве случаев языки высокого уровня порождают машинный код большего размера и исполняются медленнее. Однако некоторые языки высокого уровня для алгоритмически и структурно сложных программ могут давать заметное преимущество в эффективности, уступая низкоуровневым лишь на небольших и простых программах (подробнее см. эффективность языков). Иначе говоря, потенциальная эффективность языка меняется с повышением его «уровня» нелинейно и вообще неоднозначно. Однако скорость разработки и трудоёмкость модификации, устойчивость и другие показатели качества в сложных системах оказываются гораздо важнее предельно возможной скорости исполнения — они обеспечивают различие между программой, что работает, и той, что нет<sup>[25]</sup> — так что экономически более целесообразна эволюция аппаратного обеспечения (исполнение большего числа инструкций в единицу времени) и методов оптимизирующей компиляции (более того, последние десятилетия эволюция аппаратного обеспечения движется в направлении поддержки методов оптимизирующей компиляции для языков высокого уровня). К примеру, автоматическая сборка мусора, присутствующая в большинстве высокоуровневых языков программирования, считается одним из важнейших улучшений, благотворно повлиявших на скорость разработки<sup>[26]</sup>.

Поэтому в наши дни языки низкого уровня используются только в задачах системного программирования. Распространено мнение, что в задачах, где необходим точный контроль за ресурсами, язык сам должен требовать как можно меньше преобразований, иначе все усилия программиста окажутся напрасными. В действительности есть примеры, опровергающие это. Так, язык BitC является представителем четвёртого поколения (функциональной парадигмы программирования), но целиком и полностью ориентирован именно на системное программирование и уверенно конкурирует по скорости с Си. То есть, это «высокоуровневый язык», предназначенный для «низкоуровневого программирования». Языки третьего поколения C# и Limbo разрабатывались для использования одновременно как в системном программировании (с целью повышения отказоустойчивости операционной системы), так и в прикладном — это обеспечивает единство платформы, что сокращает потери при трансляции.

## Безопасные и небезопасные языки



Современные компьютеры представляют сложные данные реального мира в виде чисел в памяти компьютера. Это вводит в дисциплину программирования риск человеческого фактора, в том числе вероятность ошибок доступа к памяти. Поэтому многие языки программирования сопровождаются средством контроля смысла операций над двоичными данными на основе сопровождающей их логической информации — системой типов. Однако существуют и бестиповые языки, например, Forth.

Системы типов языков делятся на динамические (потомки Lisp, Smalltalk, APL) и статические, а последние, в свою очередь, делятся на неполиморфные (потомки Алгола и BCPL) и полиморфные (потомки ML)<sup>[27]</sup>. Кроме того, они делятся на явные (англ. *explicit*) и неявные (англ. *implicit*) — другими словами, требующие манифестной декларации типов для объектов в программе или статически выводящие их самостоятельно.

Системы типов бывают сильные и слабые. Сильная система типов назначает тип для всякого выражения раз и навсегда (когда бы конкретно это ни происходило — в динамике или в статике), а слабая позволяет впоследствии переназначать типы. Сильная типизация порой ошибочно отождествляется со статической.

В общем и целом, язык называется безопасным, если программы на нём, которые могут быть приняты компилятором как правильно построенные, в динамике никогда не выйдут за рамки допустимого поведения<sup>[28]</sup>. Это не значит, что такие программы не содержат ошибок вообще. Термин «хорошее поведение программы» (англ. *good behavior*) означает, что даже если программа содержит некий баг (в частности, логическую ошибку), то она тем не менее *не способна* нарушить целостность данных и обрушиться (англ. *crash*). Хотя термины неформальны, безопасность некоторых языков (например, Standard ML) математически доказуема<sup>[27]</sup>. Безопасность других (например, Ada) была обеспечена ad hoc-образом, без обеспечения концептуальной целостности, что может обернуться катастрофами, если положиться на них в ответственных задачах (см. концептуальная целостность языков). Неформальная терминология была популяризована Робином Милнером, одним из авторов теории формальной верификации и собственно языка Standard ML.

Степень контроля ошибок и реакция языка на них могут различаться. Простейшие системы типов запрещают, к примеру, вычитать строку из целого числа. Однако целыми числами могут представляться и миллиметры, и дюймы, но было бы логической ошибкой вычитать дюймы из миллиметров. Развитые системы типов позволяют (а наиболее развитые — принуждают) внедрять в программу такую логическую информацию. Для ЭВМ она является избыточной и полностью удаляется при порождении машинного кода тем или иным образом. В частности, Standard ML не допускает над данными никаких операций, кроме тех, что разрешены явно и формализованы; однако программы на нём всё же могут завершаться порождением необработанного исключения (например, при попытке деления на ноль). Его потомок, MLPolyR гарантирует также и отсутствие необработанных исключений. Такие языки называются «типобезопасными». Java и C# менее строги и контролируют лишь утечки памяти, поэтому в их контексте чаще используют более узкий термин «безопасность типов в отношении доступа к памяти» (англ. *memory type safety*) или (чаще) просто «безопасность доступа к памяти». Сильно динамически типизируемые языки отслеживают поведение программ в динамике (что влечёт снижение быстродействия) и реагируют на ошибки порождением исключения. Все эти языки ориентированы на практичность, предоставляя оптимальный компромисс между пресечением серьёзных сбоев и высокой скоростью разработки программ. Но существуют и языки, предназначенные для написания программ, которые *верны по построению*, то есть обеспечивают гарантию того, что исполнимая программа по структуре и поведению будет *тождественна* её спецификации (см. параметричность, зависимый тип). Как следствие, программы на таких языках часто называют «исполнимыми спецификациями» (см. Соответствие Карри — Говарда). Трудоёмкость разработки на таких языках возрастает на порядки, кроме того, они требуют очень высокой квалификации разработчика, поэтому они используются только в формальной верификации. Примерами таких языков служат Agda, Coq.

Языки Си и его потомок С++ являются небезопасными<sup>[29]</sup>. В программах на них обширно встречаются ситуации ослабления типизации (приведение типов) и прямого её нарушения (каламбур типизации), так что ошибки доступа к памяти являются в них статистической нормой (но крах программы наступает далеко не сразу, что затрудняет поиск места ошибки в коде). Самые мощные системы статического анализа для них (такие, как PVS-Studio<sup>[30][31]</sup>) способны обнаруживать не более 70 — 80 % ошибок, но их использование обходится очень дорого в денежном смысле. Достоверно же гарантировать безотказность программ на этих языках невозможно, не прибегая к формальной верификации, что не только ещё дороже, но и требует специальных знаний. У Си есть и безопасные потомки, такие как Cyclone.

Язык Forth не претендует на звание «безопасного», но тем не менее на практике существование программ, способных повредить данные, почти исключено, так как содержащая потенциально опасную ошибку программа аварийно завершается на первом же тестовом запуске, принуждая к коррекции исходного кода. В сообществе Erlang принят подход «let it crash» (с англ. — «дай ей обрुшиться»), также нацеленный на раннее выявление ошибок.

## Компилируемые, интерпретируемые и встраиваемые языки

Можно выделить три принципиально разных способа реализации языков программирования: компиляция, интерпретация и встраивание. Распространено заблуждение, согласно которому способ реализации является *присущим* конкретному языку свойством. В действительности, это деление до определённой степени условно. В ряде случаев язык имеет *формальную семантику, ориентированную на интерпретацию*, но все или почти все его действительные реализации являются компиляторами, порой весьма эффективно оптимизирующими (примерами могут служить языки семейства ML, такие как Standard ML, Haskell). Есть языки, размывающие границы между интерпретацией и компиляцией — например, Forth.

Компиляция означает, что исходный код программы сперва преобразуется в целевой (машинный) код специальной программой, называемой компилятором — в результате получается исполнимый модуль, который уже может быть запущен на исполнение как отдельная программа. Интерпретация же означает, что исходный код выполняется непосредственно, команда за командой (иногда — с минимальной подготовкой, буквально после разбора исходного кода в AST), — так что программа просто не может быть запущена без наличия интерпретатора. Встраивание языка можно философски рассматривать как «реализацию без трансляции» — в том смысле, что такой язык является синтаксическим и семантическим подмножеством некоего другого языка, без которого он не существует. Говоря же более точно, встраиваемые языки добавляют к сказанному ещё четыре способа реализации.

Естественный для языка способ реализации определяется *временем связывания* программных элементов с их характеристиками. В частности, в языках со статической типизацией переменные и другие объекты программы связываются с типом данных на этапе компиляции, а в случае типизации динамической — на этапе выполнения, как правило — в произвольной точке программы. Некоторые свойства элементов языка, такие как значение арифметических операторов или управляющих ключевых слов, могут быть связаны уже на этапе определения языка. В других языках возможно их переназначение (см. связывание имён). Раннее связывание обычно означает большую эффективность программы, в то время как позднее — большую гибкость, ценой которого является меньшая скорость и/или усложнение соответствующего этапа<sup>[32]</sup>. Однако, даже из, казалось бы, очевидных случаев есть исключения — например, интенциональный полиморфизм откладывает обработку статической типизации до этапа выполнения, но не замедляя, а повышая общее быстродействие (по крайней мере, в теории).

Для любого традиционно компилируемого языка (такого как Паскаль) можно написать интерпретатор. Но многие *интерпретируемые языки* предоставляют некоторые дополнительные возможности, такие как динамическая генерация кода (см. eval), так что их компиляция должна быть динамической (см. динамическая компиляция). Таким образом, составной термин «язык + способ его реализации» в ряде случаев оказывается уместен. Кроме того, большинство современных «чистых» интерпретаторов не исполняют конструкции языка непосредственно, а компилируют их в некоторое высокоуровневое промежуточное представление (например, с разыменованием переменных и раскрытием макрокоманд). Большинство традиционно интерпретируемых или компилируемых языков могут реализовываться как встраиваемые, хотя метаязыков, которые были бы способны охватить другие языки как своё подмножество, не так много (наиболее ярким представителем является Lisp).

Как правило, скомпилированные программы выполняются быстрее и не требуют для выполнения дополнительных программ, так как уже переведены на машинный язык. Вместе с тем, при каждом изменении текста программы требуется её перекомпиляция, что замедляет процесс разработки. Кроме того, скомпилированная программа может выполняться только на том же типе компьютеров и, как правило, под той же операционной системой, на которую был рассчитан компилятор. Чтобы создать исполняемый файл для машины другого типа, требуется новая компиляция. Интерпретируемые языки позволяют запускать программы сразу же после изменения, причём на разных типах машин и операционных систем без дополнительных усилий, а гомоиконичные — и вовсе динамически перемещать программу между разными машинами без прерывания её работы (наиболее общий случай сериализации), позволяя разрабатывать системы непрерывной доступности (см. *т.ж.* системы высокой доступности). Портируемость интерпретируемой программы определяется только наличием реализаций интерпретаторов под те или иные аппаратные платформы. Ценой всего этого становятся заметные потери быстродействия; кроме того, если программа содержит фатальную ошибку, то об этом не будет известно, пока интерпретатор не дойдёт до её места в коде (в отличие от статически типобезопасных языков).

Некоторые языки, например, Java и C#, находятся между компилируемыми и интерпретируемыми. А именно, программа компилируется не в машинный язык, а в машинно-независимый код низкого уровня, байт-код. Далее байт-код выполняется виртуальной машиной. Для выполнения байт-кода обычно используется интерпретация, хотя отдельные его части для ускорения работы программы могут быть транслированы в машинный код непосредственно во время выполнения программы по технологии компиляции «на лету» (Just-in-time compilation, JIT). Для Java байт-код выполняется виртуальной машиной Java (Java Virtual Machine, JVM), для C# — Common Language Runtime. Подобный подход в некотором смысле позволяет использовать плюсы как интерпретаторов, так и компиляторов.

## Языки первого и высшего порядка

### Начальные сведения

Математическая логика классифицируется по порядку — см. логика первого порядка и логика высшего порядка. Эта терминология естественным образом наследуется информатикой, образуя семантики, соответственно, первого и высшего порядка<sup>[33]</sup>. Языки первого порядка (например, потомки Алгола, такие как Basic или классический Pascal Вирта) позволяют определять только зависимости первого порядка между величинами. Например, значение square  $x$  зависит от значения  $x$ . Такие зависимости называются функциями. Языки высшего порядка позволяют определять зависимости между зависимостями. Например, значение map  $f$   $x$  зависит от значений  $f$  и  $x$ , где значение  $f$  само выражает абстрактную зависимость (другими словами, параметр  $f$  варьируется над множеством функций определённой сигнатуры). Такие зависимости называются

функциями высшего порядка. При этом в большинстве случаев говорят, что такой язык рассматривает зависимости (функции) как объекты первого класса, иначе говоря, допускает функции первого класса<sup>[34]</sup> (некоторые языки, например Си, не поддерживают первоклассные функции, но предоставляют ограниченные возможности строить функции высшего порядка). Эти термины ввёл Кристофер Стрэчи. К языкам высшего порядка относятся почти все функциональные языки (исключения очень редки; примером функционального языка первого порядка долгое время являлся SISAL, но в 2018 году в него была добавлена поддержка первоклассных функций). С развитием систем типов различие порядков распространилось и на типы (см. конструктор типов).

## Выразительность

Языки первого порядка позволяют воплощать в виде кода алгоритмы, но не архитектуру программ. По мнению Стрэчи, это ограничение унаследовано языком Алгол (а от него другими языками) из классической математики, где используются только константные операции и функции, однозначно распознаваемые вне контекста, и отсутствует систематичная нотация для произвольной работы с функциями (в качестве такой нотации в 1930-х годах было построено лямбда-исчисление, которое позже легло в основу языков высшего порядка)<sup>[35]</sup>. Схемы взаимодействия компонентов (процедур, функций, объектов, процессов и др.) для программ на языках первого порядка могут существовать лишь на условном уровне, вне самих программ. Со временем были обнаружены многократно повторяющиеся однотипные схемы такого рода, в результате чего вокруг них выстроилась самостоятельная методология — шаблоны проектирования. Языки высшего порядка позволяют воплощать такие схемы в виде исполнимого кода, пригодного для многократного использования (функций, предназначенных для преобразования и композиции других функций — см., например, конверторы и сканеры в SML)<sup>[36][37]</sup>. В результате, решения, которые на языках первого порядка могут быть представлены фрагментами программ (порой довольно сложными и громоздкими), на языках высшего порядка могут сокращаться до одной команды или вообще использования элемента семантики самого языка, не имеющего синтаксического выражения. Например, шаблон «Команда», часто применяемый в языках первого порядка, эквивалентен непосредственно самому понятию функции первого класса. То же распространяется и на более высокие слои языков — типизацию (см. полиморфизм в высших родáх) и типизацию типизации (см. полиморфизм родóв).

Сказанное преимущественно относится к языкам, семантика которых основана на лямбда-исчислении (потомки Lisp, ML). Однако некоторые языки иной природы также предоставляют возможность программирования высшего порядка. Примерами служат стековые языки (Forth) и определённая разновидность объектно-ориентированных языков (Smalltalk, CLOS, см. сообщение высшего порядка).

## Изучение

Введя терминологию «сущностей первого и второго класса», Стрэчи тут же акцентировал внимание на том, что из личного опыта и обсуждений со множеством людей он убедился, что невероятно тяжело перестать думать о функциях как об объектах второго класса<sup>[35]</sup>. То есть порядок языка имеет ярко выраженное психологическое влияние (см. гипотеза Сепира — Уорфа). Владение языками более высокого уровня поможет программисту думать в терминах более высокоуровневых абстракций<sup>[38]</sup>.

Низкоуровневые же языки могут навязывать обратное, в связи с чем широко известно следующее высказывание:

\_\_\_\_\_

Практически невозможно обучить хорошему программированию студентов, имевших опыт работы с Бейсиком: как потенциальные программисты они ментально исковерканы без надежды на восстановление.

#### Оригинальный текст (англ.)

*It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration*

— Эдсгер Дейкстра

Это значит, что само по себе использование языка высшего порядка не означает автоматически изменение архитектуры и повышение коэффициента повторного использования (см. серебряной пули нет) — определяющим фактором является умение конкретного разработчика применять соответствующие идиомы<sup>[39]</sup>.

Понимание возможностей и ограничений высокоуровневых конструкций, базовых принципов их реализации не только дают программисту возможность наиболее эффективно использовать изученный им язык, но и позволят создавать и использовать аналогичные механизмы в случае разработки на языке, где они не реализованы<sup>[38]</sup>.

Разработчику, владеющему бóльшим спектром языков программирования, будет проще выбрать среди них инструмент, наиболее подходящий для решения стоящей перед ним задачи, изучить, в случае необходимости, новый язык или реализовать предметно-ориентированный язык, к которому, к примеру, можно отнести интерфейс командной строки достаточно сложной программы<sup>[40]</sup>.

## Парадигма программирования

Отнесение языков к парадигмам может производиться по нескольким признакам, из которых одни соответствуют конкретным техническим характеристикам языков, а другие весьма условны.

Технически языки делятся, например, на допускающие побочные эффекты и ссылочно-прозрачные. Во втором случае говорят, что язык принадлежит к «чисто функциональной парадигме». В качестве парадигмы также иногда рассматриваются определённые свойства системы типов и стратегии вычисления языка, например, для параметрически полиморфных систем типов нередко говорят о реализации парадигмы обобщённого программирования. Другим примером может служить свойство гомоиконичности, открывающее целый спектр разновидностей метапрограммирования. Существует масса «языков, наследованных от математики», многие из которых формируют уникальные парадигмы. Яркими представителями являются Lisp, впервые воплотивший лямбда-исчисление и положивший таким образом начало функциональной парадигме, Smalltalk, впервые воплотивший объектно-ориентированную парадигму (появившаяся за много лет до него Симула поддерживала понятие класса, но воплощала структурную парадигму), и стековый язык Forth, воплощающий конкатенативную парадигму.

Более условно языки делятся на поколения. Первые два поколения являются низкоуровневыми, то есть ориентированными на специфику конкретного аппаратного обеспечения, и в принципе не соотносятся с какой-либо парадигмой (хотя конкретный разработчик на них, разумеется, может идеологически следовать определённым тенденциям). Вместе с третьим поколением они формируют императивную парадигму программирования, а последующие поколения — декларативную (более подробно см. раздел Языки низкого и высокого уровня). Многие декларативные языки включают в себя определённые императивные возможности, иногда — наоборот.

С ростом размера и сложности программ уже при использовании языков второго поколения начала формироваться парадигма процедурного программирования, требующая производить декомпозицию крупных процедур в цепочку иерархически связанных более мелких. Примерно в то же время появились первые языки третьего поколения и сформировалось сперва структурное программирование как прямое развитие процедурного, а затем и модульное. Со временем появилось огромное количество разнообразных путей решения проблемы комплексирования растущих программных систем с сохранением исходно императивного подхода в основе. В некоторых случаях достигнуто существенное влияние на показатели скорости разработки и качества, но в общем и целом, как выше отмечено, языки третьего поколения абстрагируются от машинной логики лишь до определённого уровня и незначительно подвержены эквивалентным преобразованиям. К настоящему времени третье поколение языков представлено наиболее обширным спектром разнообразных парадигм.

К четвёртому поколению относят функциональные языки, из которых выделяются «чисто функциональные» (англ. *purely functional*, соответствующие выше упомянутой технической категории ссылочно-прозрачных), а остальные называются «не чисто функциональными» (англ. *impurely functional*).

К пятому поколению относят языки логического программирования, в котором, помимо традиционного, выделяется несколько особых форм, например, программирование ограничениями. Фактически, языки пятого поколения — это языки четвёртого поколения, дополненные базой знаний<sup>[24]</sup> — поэтому эта категория, как уже выше отмечено, не является общепринятой.

Многие парадигмы являются условно провозглашёнными методиками организации структуры программы и применимы к большому множеству языков. Наиболее широкий охват имеют структурная и модульная — они применяются и в императивных, и в декларативных языках. Другие парадигмы тесно связаны с техническими свойствами. Например, подмножество языка C++ — шаблоны — формально может рассматриваться как полный по Тьюрингу чисто функциональный язык, но C++ не обладает присущими функциональным языкам свойствами (ссылочная прозрачность, типобезопасность, гарантия оптимизации хвостовых вызовов и др.). Как следствие, применяемые в компиляции функциональных языков алгоритмы не могут быть применены к C++, и потому ведущие исследователи функциональной парадигмы отзываются о C++ весьма скептически (подробнее см. критика шаблонов C++).

## Языки для программирования в мелком и крупном масштабе

Программы могут решать задачи различного масштаба: одна программа строит график для заданной функции, а другая управляет документооборотом крупного предприятия. Различные языки программирования рассчитаны на разный исходный масштаб задачи и, что ещё более важно, по-разному справляются с ростом сложности программных систем. Ключевым качеством языка, от которого зависит, как меняется трудоёмкость разработки по мере наращивания системы, является абстракция, то есть возможность *отделять* смысл (поведение) компонента системы от *способа его реализации*<sup>[41][42]</sup>.

Рост сложности любой программной системы принципиально ограничен тем пределом, до которого ещё можно сохранять контроль над ней: если объём информации, требуемый для осмысления компонента этой системы, превышает «вместимость» мозга одного человека, то этот компонент не будет до конца понят. Станет чрезвычайно тяжело дорабатывать его или исправлять ошибки, и от каждой корректировки можно ждать введения новых ошибок из-за этого неполного знания.

*There is a fundamental limit to complexity of any software system for it to be still manageable: if it requires more than «one brainfull» of information to understand a component of the system, then that component will not be understood fully. It will be extremely difficult to make enhancements or fix bugs, and each fix is likely to introduce further errors due to this incomplete knowledge.*

— Martin Ward, «Language Oriented Programming»<sup>[43]</sup>

Такие показатели качества исходного кода, как тестируемость и модифицируемость, очевидным образом определяются коэффициентом повторного использования. Это может означать как применение разных функций к одному и тому же компоненту, так и возможность применять одну и ту же функцию к разным компонентам. Параметрически полиморфные (особенно выводящие) и динамические системы типов существенно повышают коэффициент повторного использования: например, функция, вычисляющая длину массива, будет применима к бесконечному множеству типов массивов<sup>[27][44]</sup>. Если же язык требует в сигнатуре функции указывать конкретный способ реализации входных данных, то этот коэффициент резко страдает. Например, Pascal критиковался за необходимость всегда указывать конкретный размер массива<sup>[45]</sup>, а C++ — за необходимость различать `.` и `->` при обращении к компонентам составных данных<sup>[46]</sup>. Языки высшего порядка позволяют выделять схемы взаимодействия функций в многократно вызываемый блок кода (функцию высшего порядка)<sup>[36][47]</sup>, а наибольших значений повторное использование достигает при переходе к языку более высокого уровня — при необходимости специально разрабатываемого для данной задачи — в этом случае повторно используется язык, а не одна функция<sup>[43]</sup>, а сама разработка языка может вестись с интенсивным повторным использованием компонентов компилятора<sup>[48]</sup>.

С развитием языков появились особые (присущие исключительно программированию, не требовавшиеся ранее в математике) категории компонентов и зависимостей: монады, классы типов, полиморфные ветвления, аспекты и др. Их использование позволяет выражать бóльшую функциональность в том же объёме кода, тем самым переводя программирование-по-крупному в более мелкий масштаб.

Другие фундаментальные проблемы, связанные со сложностью крупных систем, лежат вне самих программ: это взаимодействие разрабатывающих её программистов между собой, документирование и т. д. Помимо обеспечения абстракции, не последнюю роль в этом играет концептуальная целостность выбранного языка программирования<sup>[49][43]</sup>.

Кроме свойств семантики языка, повторное использование может обеспечиваться посредством модульной структуры программной системы или комплекса. Более того, сколь бы гибким ни был язык, работа с огромными объёмами кодов, особенно множеством людей, требует их декомпозиции на модули тем или иным образом. Модульная структура подразумевает не просто разбиение монолитного исходного кода программы на множество текстовых файлов, а обеспечение абстракции в более крупном масштабе, то есть определение интерфейса для всякого логически завершённого фрагмента и сокрытие деталей его реализации. В зависимости от применённых в языке правил определения области видимости язык может допускать или не допускать автоматическое определение зависимостей. Если согласно правилам возможен конфликт имён, то автоопределение зависимостей невозможно, и тогда в заголовке модуля требуется явно перечислять имена модулей, компоненты которых в нём используются.

Некоторые языки (например, Basic или классический Pascal Вирта) ориентированы исключительно на разработку мелких, структурно простых программ. Они не обеспечивают ни развитой системы модулей, ни гибкости конкретных фрагментов. Язык Си создавался как «высокоуровневый ассемблер», что само по себе не предполагает разработку систем выше некоторого порога сложности, поэтому поддержка крупномасштабного программирования в него заложена также не



была. Некоторые языки высокого и сверхвысокого уровня (Erlang, Smalltalk, Prolog) предоставляют в качестве базовых примитивных элементов концепции, которые в других языках представляются конструктивно и алгоритмически сложными (процессы, классы, базы знаний) — аналогично разнообразным математическим исчислениям (см. также концептуальная целостность языков). Поэтому такие языки нередко рассматриваются в роли предметно-специфичных — на них выглядят простыми некоторые (но далеко не все) задачи, которые на других языках выглядят сложными. Однако расширение функциональности в других аспектах на этих языках может оборачиваться затруднениями. Standard ML и его родственники расслаиваются на два языка, из которых один — «язык-ядро» (англ. *core language*) — ориентирован на разработку простых программ, а другой — «язык модулей» (англ. *module language*), — соответственно, на нелинейную компоновку их в сложные программные системы. Со временем были построены варианты слияния их воедино (1ML). Многие другие языки также включают системы модулей, но большинство из них являются *языками модулей первого порядка*. Язык модулей ML является единственным в своём роде *языком модулей высшего порядка*. Языки Lisp и Forth позволяют наращивать системы произвольно и безгранично, в том числе позволяя создавать встраиваемые предметно-специфичные языки внутри себя (как своё синтаксическое и семантическое подмножество) — поэтому их нередко называют метаязыками.

Наиболее популярным на сегодняшний день подходом к решению проблемы комплексирования является объектно-ориентированное программирование, хотя успешность его применения на протяжении десятилетий существования неоднократно подвергалась скепсису, и до сих пор отсутствуют достоверные данные о том, что он приносит выигрыш по сравнению с другими подходами по тем или иным показателям качества. Ему сопутствуют (а порой конкурируют) различные технологии регламентирования зависимостей между компонентами: метаклассы, контракты, прототипы, примеси, типажи и др.

Более мощным подходом исторически считалось использование различных форм метапрограммирования, то есть автоматизации самого процесса разработки на различных уровнях. Принципиально различается метапрограммирование *внешнее* по отношению к языку и доступное в *самом* языке. При использовании языков первого порядка сложность растущих программных систем быстро переходит порог способностей человека по восприятию и переработке информации, поэтому применяются *внешние* средства предварительного визуального проектирования, позволяющие обозревать сложные схемы в упрощённом виде и в уменьшенном масштабе и затем автоматически порождать каркас кода — см. CASE. В сообществах разработчиков, использующих языки высшего порядка, доминирует прямо противоположный подход — пресекать саму возможность выхода сложности из-под контроля за счёт разделения информационных моделей на независимые составляющие и разработки средств автоматического преобразования одних моделей в другие — см. языково-ориентированное программирование.

## Концептуальная целостность языков

Фредерик Брукс<sup>[50]</sup> и Ч. Э. Р. Хоар<sup>[51]</sup> делают акцент на необходимости обеспечения *концептуальной целостности* информационных систем вообще и языков программирования в частности, чтобы в каждой части системы использовались сходные синтаксические и семантические формы и не требовалось осваивать помимо собственно состава системы также и правила её *идиоматического* использования. Хоар предсказывал, что сложность Ады станет причиной катастроф. Алан Кэй отделяет языки, являющиеся «стилем во плоти» (англ. *crystalization of style*) от прочих языков, являющихся «склеиванием возможностей» (англ. *agglutination of features*)<sup>[52]</sup>. Грег Нельсон<sup>[53]</sup> и Эндрю Аппель<sup>[27]</sup> выделяют в особую категорию «языки, наследованные от математики» (англ. *mathematically-derived languages*).



Эти акценты призывают к использованию языков, воплощающих некое математическое исчисление, аккуратно адаптированное для того, чтобы быть более практичным языком для разработки реальных программ. Такие языки отличаются ортогональностью, и хотя это означает необходимость вручную реализовывать многие распространённые идиомы, доступные в более популярных языках в качестве примитивов языка, выразительность таких языков в целом может быть существенно выше.

Лишь некоторые языки попадают под эту категорию; большинство же языков проектируются приоритетно исходя из возможности эффективной трансляции в машину Тьюринга. Многие языки опираются на общие теории, но при разработке они почти никогда не проверяются на безопасность совместного использования конкретных языковых элементов, являющихся частными приложениями этих теорий, что неизбежно приводит к несовместимости между реализациями языка. Эти проблемы либо игнорируются, либо начинают преподноситься как естественное явление (англ. «*not a bug, but a feature*»), но в действительности их причиной является то, что язык не был подвергнут математическому анализу<sup>[54]</sup>.

Примеры математически обоснованных языков и воплощаемых ими математических моделей:

- Agda, Epigram, Idris — интуиционистская теория типов Мартин-Лёфа.
- APL и его потомки (J, K) — оригинальная семантика, не имеющая названия, воплощающая нотацию Айверсона для исчисления массивов (часто встречается термин «*array languages*»).
- Coq — исчисление индуктивных конструкций.
- Erlang — исчисление процессов (первоначально в форме модели акторов, позже также построено обоснование на  $\pi$ -исчислении<sup>[55]</sup>).
- Forth — стековая машина и конкатенативный язык программирования.
- Haskell — теория категорий (включая «декартово замкнутую категорию», воплощающую лямбда-исчисление; категорию монад для моделирования побочных эффектов; расширение системы типов Хиндли — Милнера; систему родóв; и др.).
- Joy — композиция функций и гомоморфизм (иначе говоря, чистый конкатенативный язык программирования и, как следствие, чистый функциональный).
- Lisp — лямбда-исчисление Чёрча (в том числе язык S-выражений, воплощающий нотацию пар Чёрча).
  - Scheme — «облагороженный» диалект Лиспа (сильнее типизированный, в большей степени гомоиконичный, ограничивающийся гигиеническими макроопределениями и соблюдающий числовую башню), дополненный нотацией продолжений.
- ML — типизированное лямбда-исчисление, то есть лямбда-исчисление, дополненное системой типов Хиндли — Милнера.
- Prolog — исчисление предикатов.
  - Mercury — исчисление предикатов, дополненное системой типов Хиндли — Милнера.
- Smalltalk — теория множеств<sup>[56]</sup> (с соблюдением числовой башни).
- SQL — исчисление кортежей (вариант реляционного исчисления, в свою очередь основанного на исчислении предикатов первого порядка).
- SGML и его потомки (HTML, XML) — нотация деревьев (важный случай графов).
- Unlambda — комбинаторная логика.
- Регулярные выражения.
- Рефал — оригинальная семантика Турчина, носящая название «*Рефал-машины*» или «*Рефал-автомата*», созданная на основе нормального алгоритма Маркова, воплощающая композицию теории автоматов, сопоставления с образцом и переписывания термов.

Наличие математического обоснования для языка может гарантировать (или, как минимум, обещать с очень высокой вероятностью) некоторые или все из следующих положительных свойств:

- Существенное повышение стабильности программ. В одних случаях — за счёт построения доказательства надёжности для самого языка (см. типобезопасность), существенного упрощения формальной верификации программ и даже получения языка, который сам является системой автоматического доказательства (Coq, Agda). В других случаях — за счёт раннего обнаружения ошибок на первых же пробных запусках программ (Forth и регулярные выражения).
- Обеспечение *потенциально* более высокой эффективности программ. Даже если семантика языка далека от архитектуры целевой платформы компиляции, к нему могут быть применимы формальные методики глобального анализа программ (хотя трудоёмкость написания даже тривиального транслятора может оказаться выше). Например, для языков Scheme и Standard ML существуют развитые полнопрограммно-оптимизирующие компиляторы и суперкомпиляторы, результат работы которых может уверенно конкурировать по скорости с языком низкого уровня Си и даже опережать последний (хотя ресурсоёмкость работы самих компиляторов оказывается значительно выше). Одна из самых быстрых СУБД — KDB<sup>[57]</sup> — написана на языке К. Язык Scala (унаследовавший математику от ML) обеспечивает на платформе JVM более высокую скорость, чем «родной» для неё язык Java. С другой стороны, Forth имеет репутацию одного из самых нетребовательных к ресурсам языков (менее требователен, чем Си) и используется для разработки приложений реального времени под самые маломощные ЭВМ; кроме того, транслятор Форта является одним из наименее трудоёмких в реализации на ассемблере.
- Заранее известный (неограниченный или, наоборот, чётко очерченный) предел роста сложности программных компонентов, систем и комплексов, которые можно выразить средствами этого языка с сохранением показателей качества<sup>[27][58]</sup>. Языки, не имеющие математического обоснования (а именно такие наиболее часто применяются в мейнстриме: C++, Java, C#, Delphi и др.), на практике ограничивают реализуемую функциональность и/или снижают качество по мере усложнения системы<sup>[59]</sup>, так как им присущи экспоненциальные кривые роста сложности как относительно работы одного отдельно взятого человека, так и относительно сложности управления проектом в целом<sup>[49][60]</sup>. *Прогнозируемая сложность системы* приводит либо к поэтапной декомпозиции проекта на множество более мелких задач, каждая из которых решается соответствующим языком, либо к языково-ориентированному программированию для случая, когда адресуемой языком задачей является как раз описание семантик и/или символьные вычисления (Lisp, ML, Haskell, Рефал, Регулярные выражения). Языки с неограниченным пределом роста сложности программ нередко относят к метаязыкам (что в непосредственном толковании термина не верно, но практике сводимо, так как всякий мини-язык, выбранный для решения некоторой подзадачи в составе общей задачи, может быть представлен в виде синтаксического и семантического подмножества данного языка, не требуя трансляции<sup>[61]</sup>).
- Удобство для человека при решении задач, на которые этот язык ориентирован по своей природе (см. проблемно-ориентированный язык), что в некоторой степени также способно (косвенно) повлиять на повышение стабильности результирующих программ за счёт повышения вероятности обнаружения ошибок в исходном коде и снижения дублирования кода.

## Особые категории языков

- Учебные
- Предметно-специфичные

- Эзотерические
- Визуальные

## Формальные преобразования и оптимизация

---

В. Ф. Турчин отмечает<sup>[62]</sup>, что достоинства всякого формализованного языка определяются не только тем, сколь он удобен для непосредственного использования человеком, но и тем, в какой степени тексты на этом языке поддаются формальным преобразованиям.

Например, ссылочная прозрачность означает, что параметры функций не обязаны вычисляться перед вызовом — вместо этого фактически переданное выражение может быть целиком подставлено на место переменной в функции, и поведение функции от этого не изменится. Это открывает возможности почти произвольных автоматических преобразований программ: могут устраняться ненужные промежуточные представления данных, редуцироваться сложные цепочки вычислений, подбираться оптимальное количество параллельных процессов, вводиться  мемоизация, и пр. С другой стороны, это означает полное отсутствие побочных эффектов, а это делает реализацию некоторых алгоритмов заведомо менее эффективной, чем при использовании изменяемого состояния.

Для небольших и простых программ языки высокого уровня порождают машинный код большего размера и исполняются медленнее. Однако для алгоритмически и структурно сложных программ преимущество может быть на стороне некоторых языков высокого уровня, так как человек физически не способен выражать сложные концепции с учётом их эффективного исполнения на языке машины. К примеру, существует бенчмарк, на котором MLton и Stalin Scheme уверенно опережают GCC. Есть масса частных причин, по которым автоматическая оптимизация в ходе трансляции языков высокого уровня даёт *в принципе* более высокую скорость исполнения, чем сознательный контроль способа реализации на языках низкого уровня. Например, имеются достоверные данные о том, что автоматическое управление памятью более эффективно, чем ручное, уже только при использовании динамического метода (см. сборка мусора)<sup>[63]</sup>, а существует и потенциально более эффективный статический метод (см. управление памятью на основе регионов). Далее, для каждого микроконтекста необходимо распределить регистры с учётом минимизации обращения к памяти, а это требует решения задачи раскраски графа. Такого рода особенностей машинной логики очень много, так что общая информационная сложность возрастает экспоненциально при каждом «шаге на уровень вниз», а компиляция языка высокого уровня может включать *десятки* таких шагов.

Существует множество стратегий автоматической оптимизации. Некоторые универсальны, другие могут быть применимы лишь к языкам определённой природы, а некоторые зависят от *способа использования* языка. Примером может служить оптимизация хвостовых вызовов и её частный случай — оптимизация хвостовой рекурсии. Хотя компиляторы многих языков осуществляют оптимизацию хвостовой *рекурсии* при определённых условиях, лишь некоторые языки способны семантически гарантировать оптимизацию хвостовых вызовов в общем случае. Стандарт языка Scheme требует, чтобы всякая реализация гарантировала её. Для многих функциональных языков она в принципе применима, но лишь оптимизирующие компиляторы её выполняют. В языках вроде Си или С++ она может производиться лишь в определённых случаях и лишь при использовании глобального анализа потока управления<sup>[64]</sup>.

Языки высшего порядка в большинстве случаев вынуждены исполняться медленнее, чем языки первого порядка. Причины лежат как в самой декомпозиции линейного кода на цепочку вложенных вызовов, так и в вытекающих особенностях низкоуровневого представления функций (см.

замыкание) и данных (обёрнутое (англ. *boxed*), теговое). Однако существуют техники агрессивной оптимизации программ, позволяющие редуцировать языки высшего порядка до языков первого порядка (см. дефункционализация, MLton, Stalin Scheme).

## Популярность языков

---

Трудно определить, какой язык программирования наиболее популярен, так как значение слова «популярность» зависит от контекста (в английском языке используется термин «usage», имеющий ещё более размытое значение). Один язык может отнимать наибольшее количество человеко-часов, на другом написано наибольшее число строк кода, третий занимает наибольшее процессорное время, а четвёртый наиболее часто служит исследовательской базой в академических кругах. Некоторые языки очень популярны для конкретных задач. Например, Кобол до сих пор доминирует в корпоративных дата-центрах, Фортран — в научных и инженерных приложениях, вариации языка Си — в системном программировании, а различные потомки ML — в формальной верификации. Другие языки регулярно используются для создания самых разнообразных приложений.

Существуют различные метрики для измерения популярности языков, каждая из которых разработана с пристрастием к определённому смыслу понятия популярности:

- подсчёт числа вакансий, упоминающих язык;
- количество проданных книг (учебников или справочников);
- оценка количества строк кода, написанных на языке (что не принимает в расчёт редко публикуемые случаи использования языков);
- подсчёт упоминаний языка в запросах поисковиков.

Следует заметить, что высокие оценки по этим показателям не только никак не свидетельствуют о высоком техническом уровне языка и/или оптимизации расходов при его использовании, но и, напротив, порой могут говорить об обратном. Например, язык Кобол входит в число лидеров по количеству написанных на нём строк кода, но причиной этому является крайне низкий показатель модифицируемости кода, что делает этот код не повторно используемым, а legacy-кодом. Как следствие, поддержка программ на Коболе в кратковременной перспективе обходится значительно дороже, чем программ на большинстве современных языков, но переписывание их с нуля потребовало бы значительных единовременных вложений и может сравниться только с долговременными расходами. Техническое несовершенство Кобола обусловлено тем, что его разрабатывали без привлечения экспертов в области информатики<sup>[65][66]</sup>.

## См. также




---



- Компьютерный язык
- Программирование
- Hello, world!
- Стандарт оформления кода
- Теория языка программирования


## Примечания

---

1. ISO/IEC/IEEE 24765:2010 Systems and software engineering — Vocabulary
2. ISO/IEC 2382-1:1993, Information technology — Vocabulary — Part 1: Fundamental terms



3. Список языков программирования (<http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>) (англ.) (недоступная ссылка). Дата обращения: 3 мая 2004. Архивировано (<https://web.archive.org/web/20040612042127/http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>) 12 июня 2004 года.
4. Rojas, Raúl, et al. (2000). «Plankalkül: The First High-Level Programming Language and its Implementation». Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000. (full text) (<http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm>) Архивная копия (<http://web.archive.org/web/20141018204625/http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm>) от 18 октября 2014 на Wayback Machine
5. Computer Languages, 1989, 1. Невидимый конструктор § Создание кодов, понятных человеку, с. 16.
6. Linda Null, Julia Lobur, *The essentials of computer organization and architecture*, Edition 2, Jones & Bartlett Publishers, 2006, ISBN 0-7637-3769-0, p. 435
7. O'Reilly Media. History of programming languages ([http://www.oreilly.com/news/graphics/prog\\_lang\\_poster.pdf](http://www.oreilly.com/news/graphics/prog_lang_poster.pdf))  (PDF) (недоступная ссылка). Дата обращения: 5 октября 2006. Архивировано ([https://web.archive.org/web/20080228004804/http://www.oreilly.com/news/graphics/prog\\_lang\\_poster.pdf](https://web.archive.org/web/20080228004804/http://www.oreilly.com/news/graphics/prog_lang_poster.pdf))  28 февраля 2008 года.
8. Frank da Cruz. IBM Punch Cards (<http://www.columbia.edu/acis/history/cards.html>) Columbia University Computing History (<http://www.columbia.edu/acis/history/index.html>).
9. Richard L. Wexelblat: *History of Programming Languages*, Academic Press, 1981, chapter XIV.
10. Пратт, 1979, 4.6. Сопоставление с образцом, с. 130—132.
11. Пратт, 1979, 15. Снобол 4, с. 483—516.
12. Пратт, Зелковиц, 2002, 8.4.2. Сопоставление с образцом, с. 369—372.
13. François Labelle. Programming Language Usage Graph (<http://www.cs.berkeley.edu/~flab/languages.html>) (недоступная ссылка). SourceForge. Дата обращения: 21 июня 2006. Архивировано (<https://web.archive.org/web/20060617055109/http://www.cs.berkeley.edu/%7Eflab/languages.html>) 17 июня 2006 года.
14. Hayes, Brian. The Semicolon Wars (англ.) // *American Scientist* : magazine. — 2006. — Vol. 94, no. 4. — P. 299—303.
15. Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, Akihiko Nakase (December 1994). «KLIC: A Portable Implementation of KL1» *Proc. of FGCS '94, ICOT Tokyo*, December 1994. <http://www.icot.or.jp/ARCHIVE/HomePage-E.html> Архивная копия (<http://web.archive.org/web/20060925132105/http://www.icot.or.jp/ARCHIVE/HomePage-E.html>) от 25 сентября 2006 на Wayback Machine KLIC is a portable implementation of a concurrent logic programming language KL1.
16. Jim Bender. Mini-Bibliography on Modules for Functional Programming Languages (<http://readscheme.org/modules/>) (недоступная ссылка). ReadScheme.org (15 марта 2004). Дата обращения: 27 сентября 2006. Архивировано (<https://web.archive.org/web/20060924085057/http://readscheme.org/modules/>) 24 сентября 2006 года.
17. Stroustrup, Bjarne *Evolving a language in and for the real world: C++ 1991-2006* (<http://stroustrup.com/hopl-almost-final.pdf>) .
18. Т. Пратт, М. Зелковиц. Языки программирования. Разработка и реализация. — 4. — Санкт-Петербург : Питер, 2002. — С. 203. — 688 с. — 4000 экз. — ISBN 5-318-00189-0.
19. Страуструп Б. Дизайн и эволюция C++. — Санкт-Петербург : Питер, 2006. — С. 74—76. — 448 с. — 2000 экз. — ISBN 5-469-01217-4.
20. Сейбел - Кодеры за работой, 2011, Глава 12. Кен Томпсон, с. 414.
21. Зуев Е.А., Кротов А.Н., Сухомлин В.А. Язык программирования Си++: этапы эволюции и современное состояние (<http://citforum.ru/programming/prg96/76.shtml>) (4 октября 1996). Дата обращения: 16 января 2017.
22. Paulson, «ML for the Working Programmer», 1996, с. 213.
23. Paulson, «ML for the Working Programmer», 1996, с. 1.

24. Mernik, 2012, с. 2—12.
25. Paulson, «ML for the Working Programmer», 1996, с. 9.
26. Rick Byers. Garbage Collection Algorithms (<https://courses.cs.washington.edu/courses/csep521/07wi/prj/rick.pdf>)  .courses.cs.washington.edu. — Project for CSEP 521, Winter 2007. Дата обращения: 28 декабря 2016.
27. Appel - A Critique of Standard ML, 1992.
28. Harper — Practical Foundations for Programming Languages, 2012, Chapter 4. Statics, с. 35.
29. Mitchel, 2004, 6.2.1 Type Safety, с. 132—133.
30. Comparison of static code analyzers: CppCat, Cppcheck, PVS-Studio and Visual Studio (<http://www.viva64.com/en/b/0241/>)
31. Comparing PVS-Studio with other code analyzers (<http://www.viva64.com/en/a/0052/#ID0EBKAE>)
32. Пратт, 1979, 2.7. Связывание и время связывания, с. 46—51.
33. Reynolds, «Theories of programming languages», 1998, 12.4 Deriving a First-Order Semantics.
34. Strachey — Fundamental Concepts, 1967, 3.5.1. First and second class objects., с. 32—34.
35. Strachey — Fundamental Concepts, 1967, 3.5.1. First and second class objects, с. 32—34.
36. SICP.
37. Harper — Practical Foundations for Programming Languages, 2012, 8.2 Higher-Order Functions, с. 67.
38. Пратт, Зелковиц, 2002, 1.1 Зачем изучать языки программирования, с. 17—18.
39. Bruce A. Tate. Foreword // Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages (<https://archive.org/details/sevenlanguagesin00tate>). — Pragmatic Bookshelf, 2010. — С. 14 (<https://archive.org/details/sevenlanguagesin00tate/page/14>)—16. — ISBN 978-1934356593.
40. Пратт, Зелковиц, 2002, 1.1 Зачем изучать языки программирования, с. 18.
41. Ахо, Ульман, 1992.
42. Joyner, 1996, 2.2 Communication, abstraction and precision, с. 4.
43. Ward, 1994.
44. Paulson, "ML for the Working Programmer", 1996, с. 63—64.
45. Kernigan about Pascal, 1981.
46. Joyner, 1996, 3.17 '.' and '->', с. 26.
47. Paulson, «ML for the Working Programmer», 1996, с. 177—178.
48. Hudak, 1998.
49. Брукс, 1975, 1995.
50. Брукс, 1975, 1995, Достижение концептуальной целостности, с. 30.
51. C.A.R. Hoare — The Emperor's Old Clothes, Communications of the ACM, 1981
52. Алан Кэй. The Early History of Smalltalk (<http://stephane.ducasse.free.fr/FreeBooks/SmalltalkHistoryHOPL.pdf>)  . — Apple Computer, ACM SIGPLAN Notices, vol.28, №3, March 1993.
53. Greg Nelson. Systems Programming with Modula-3. — NJ: Prentice Hall, Englewood Cliffs, 1991. — 288 с. — ISBN 978-0135904640.
54. Commentary on SML, 1991, Aims of the Commentary, с. vii.
55. Thomas Noll, Chanchal Kumar Roy. Modeling Erlang in the Pi-Calculus (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.3913&rep=rep1&type=pdf>). — ACM 1-59593-066-3/05/0009, 2005.
56. Design Principles Behind Smalltalk (<http://c2.com/cgi/wiki?DesignPrinciplesBehindSmalltalk>)
57. kx: Calibrated performance (<http://kx.com/benchmarks.php>)






58. *Luca Cardelli*. *Typeful programming* (<http://www.lucacardelli.name/Papers/TypefulProg.pdf>) . — IFIP State-of-the-Art Reports, Springer-Verlag, 1991.
59. Ward, 1994: «There is a fundamental limit to complexity of any software system for it to be still manageable: if it requires more than «one brainfull» of information to understand a component of the system, then that component will *not* be understood fully. It will be extremely difficult to make enhancements or fix bugs, and each fix is likely to introduce further errors due to this incomplete knowledge.».
50. Гласс, 2004.
51. Czarnecki et al, 2004.
52. Турчин В. Ф. Эквивалентные преобразования программ на РЕФАЛе: Труды ЦНИПИАСС 6: ЦНИПИАСС, 1974.
53. *B. Zorn*. The Measured Cost of Conservative Garbage Collection. Technical Report CU-CS-573-92. // University of Colorado at Boulder. — 1993. — doi:10.1.1.14.1816 (<https://dx.doi.org/10.1.1.14.1816>).
54. Ehud Lamm.
55. *Richard L. Conner*. *Cobol, your age is showing* (<https://books.google.com/books?id=BrEo9KtAQH4C&pg=RA1-PA61>) (англ.) // *Computerworld : magazine*. — International Data Group, 1984. — 14 May (vol. 18, no. 20). — P. ID/7—ID/18. — ISSN 0010-4841 (<https://www.worldcat.org/search?fq=x0:jrnl&q=n2:0010-4841>).
56. *Robert L. Mitchell*. *Cobol: Not Dead Yet* ([http://www.computerworld.com/s/article/266156/Cobol\\_Not\\_Dead\\_Yet](http://www.computerworld.com/s/article/266156/Cobol_Not_Dead_Yet)). *Computerworld* (4 октября 2006). Дата обращения: 27 апреля 2014.

## Литература



---

- *Гавриков М. М., Иванченко А. Н., Гринченков Д. В.* Теоретические основы разработки и реализации языков программирования. — КноРус, 2013. — 178 с. — ISBN 978-5-406-02430-0.
- *Криницкий Н. А., Миронов Г. А., Фролов Г. Д.* Программирование. — ГИФМЛ, 1963. — 384 с.
- *Братчиков И. Л.* Синтаксис языков программирования. — Наука, 1975. — 230 с.
- *Лавров С. С.* Основные понятия и конструкции языков программирования. — Финансы и статистика, 1982. — 80 с.
- *Christopher Strachey*. *Fundamental Concepts in Programming Languages* (<http://www.itu.dk/courses/BPRD/E2009/fundamental-1967.pdf>)  (англ.). — 1967. Архивировано (<https://web.archive.org/web/20170812012310/http://www.itu.dk/courses/BPRD/E2009/fundamental-1967.pdf>)  12 августа 2017 года.
  - Повторно опубликовано: *Christopher Strachey*. *Fundamental Concepts in Programming Languages* (англ.) // *Higher-Order and Symbolic Computation*. — 2000. — Т. 13. — С. 11—49. — doi:10.1023/A:1010000313106 (<https://dx.doi.org/10.1023%2FA%3A101000313106>).
- *Харольд Абельсон, Джеральд Джей Сассман, Джули Сассман*. *Структура и интерпретация компьютерных программ (SICP)*.
- *Фредерик Брукс*. *Мифический человеко-месяц или Как создаются программные системы*. — Addison-Wesley, 1975, 1995. — ISBN ISBN 5-93286-005-7 (1-е изд.).



- Теренс Пратт. Языки программирования: разработка и реализация = Programming Language Design and Implementation (PLDI). — 1-е издание. — Мир, 1979.
- Альфред Ахо, Рави Сети, Джеффри Ульман. Компиляторы: принципы, технологии и инструменты. — Addison-Wesley Publishing Company, Издательский дом «Вильямс», 1985, 2001, 2003. — 768 с. — ISBN 5-8459-0189-8 (рус.), 0-201-10088-6 (ориг.).
- Time-Life Books. Язык компьютера = Computer Languages. — М.: Мир, 1989. — Т. 2. — 240 с. — (Understanding Computers). — 100 000 экз. — ISBN 5-03-001148-X.
- Лука Карделли. Typeful programming (<http://www.lucacardelli.name/Papers/TypefulProg.pdf>)  (англ.) // IFIP State-of-the-Art Reports. — Springer-Verlag, 1991. — Вып. Formal Description of Programming Concepts. — С. 431—507.
- Robin Milner, Mads Tofte. Commentary on Standard ML (<https://www.itu.dk/people/tofte/publ/1990sml/1991commentaryBody.pdf>) . — MIT Press, 1991. — ISBN 0-262-63132-7. Архивировано (<https://web.archive.org/web/20141201213417/https://www.itu.dk/people/tofte/publ/1990sml/1991commentaryBody.pdf>)  1 декабря 2014 года.
- Альфред Ахо, Джеффри Ульман. Foundations of Computer Science. — Computer Science Press, 1992.
- Andrew W. Appel. A Critique of Standard ML (<http://www.cs.princeton.edu/research/techreps/TR-364-92>). — Princeton University, revised version of CS-TR-364-92, 1992.
- Martin Ward (<http://www.cse.dmu.ac.uk/~mward/>). Language Oriented Programming (<http://www.cse.dmu.ac.uk/~mward/martin/papers/middle-out-t.pdf>) . — Computer Science Department, Science Labs, 1994.
- Ian Joyner. A Critique of C++ and Programming and Language Trends of the 1990s - 3rd Edition. (<http://www.quinn.echidna.id.au/quinn/C++-Critique-3ed.pdf>)  // копирайт и список изданий (<http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/cppcrit/index008.htm>). — 1996.
- Lawrence C. Paulson. ML for the Working Programmer. — 2nd. — Cambridge, Great Britain: Cambridge University Press, 1996. — 492 с. — ISBN 0-521-57050-6 (твёрдый переплёт), 0-521-56543-X (мягкий переплёт).
- John C. Reynolds. Theories of programming languages (<http://booksee.org/book/696403>). — Cambridge University Press, 1998. — ISBN 978-0-521-59414-1 (hardback), 978-0-521-10697-9 (paperback).
- Andrew W. Appel. Modern compiler implementation in ML (in C, in Java) (неопр.). — Cambridge, Great Britain: Cambridge University Press, 1998. — 538 с. — ISBN (ML) 0-521-58274-1 (hardback), 0-521-60764-7 (paperback).
- Paul Hudak. Modular Domain Specific Languages and Tools ([http://haskell.cs.yale.edu/?post\\_type=publication&p=125](http://haskell.cs.yale.edu/?post_type=publication&p=125)). — IEEE Computer Society Press, Department of Computer Science, Yale University, 1998. Архивировано ([https://web.archive.org/web/20131017021928/http://haskell.cs.yale.edu/?post\\_type=publication&p=125](https://web.archive.org/web/20131017021928/http://haskell.cs.yale.edu/?post_type=publication&p=125)) 17 октября 2013 года.
- Роберт У. Себеста. Основные концепции языков программирования = Concepts of Programming Languages / Пер. с англ. — 5-е изд. — М.: Вильямс, 2001. — 672 с. — 5000 экз. — ISBN 5-8459-0192-8 (рус.), ISBN 0-201-75295-6 (англ.).
- Вольфенгаген В. Э. Конструкции языков программирования. Приёмы описания. — М.: Центр ЮрИнфоР, 2001. — 276 с. — ISBN 5-89158-079-9.



- *Паронджанов В. Д.* Как улучшить работу ума. Алгоритмы без программистов — это очень просто! — М.: Дело, 2001. — 360 с. — ISBN 5-7749-0211-0.
- *Pierce, Benjamin C.* *Types and Programming Languages* (<http://www.cis.upenn.edu/~bcpierce/tapl/>). — MIT Press, 2002. — ISBN 0-262-16209-1.
  - Перевод на русский язык: *Пирс Б.* Типы в языках программирования. — Добросвет, 2012. — 680 с. — ISBN 978-5-7913-0082-9.
- *Теренс Пратт, Марвин Зелковиц.* Языки программирования: разработка и реализация. — 4-е издание. — Питер, 2002. — (Классика Computer Science). — ISBN 978-5-318-00189-5.
- *Martin Campbell-Kelly.* *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry* ([https://archive.org/details/fromairlinereser00mart\\_0](https://archive.org/details/fromairlinereser00mart_0)). — MIT Press, 2003. — 372 с. — (History of Computing). — ISBN 978-1422391761.
- *Роберт Гласс.* Факты и заблуждения профессионального программирования ([http://www.rsdn.ru/res/book/prog/Facts\\_and\\_Fallacies.xml](http://www.rsdn.ru/res/book/prog/Facts_and_Fallacies.xml)). — «Символ-Плюс», 2004. — 240 с. — ISBN 5-93286-092-8, 978-5-93286-092-2.
- *John C. Mitchell.* *Concepts in Programming Languages*. — Cambridge University Press, 2004. — ISBN 0-511-04091-1 (eBook in netLibrary); 0-521-78098-5 (hardback).
- *K. Czarnecki, J. O'Donnell, J. Striegnitz, W. Taha.* DSL implementation in metaocaml, template haskell, and C++ (<http://camlunity.ru/swap/Library/Computer%20Science/Metaprogramming/Domain-Specific%20Languages/DSL%20Implementation%20in%20MetaOCaml,%20Template%20Haskell%20and%20C++.pdf>) . — University of Waterloo, University of Glasgow, Research Centre Julich, Rice University, 2004. Архивировано (<https://web.archive.org/web/20160305042746/http://camlunity.ru/swap/Library/Computer%20Science/Metaprogramming/Domain-Specific%20Languages/DSL%20Implementation%20in%20MetaOCaml,%20Template%20Haskell%20and%20C++.pdf>)  5 марта 2016 года.
- *Ф. Бьянкуцци, Ш. Уорден.* Пioneры программирования. Диалоги с создателями наиболее популярных языков программирования (<http://www.symbol.ru/alphabet/792588.html>). — СПб.: Символ-Плюс, 2010. — 608 с. — ISBN 978-5-93286-170-7.
- *Питер Сейбел* (<http://c2.com/cgi/wiki?PeterSeibel>). Кодеры за работой. Размышления о ремесле программиста. — Символ-Плюс, СПб. — 2011. — ISBN 978-5-93286-188-2, 978-1-4302-1948-4 (англ.).
- *Robert Harper.* *Practical Foundations for Programming Languages* (<http://bookfi.org/book/1076504>). — version 1.37 (revised 01.11.2014). — licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License., 2012. — 544 с. Архивная копия (<http://web.archive.org/web/20151024155104/http://bookfi.org/book/1076504>) от 24 октября 2015 на Wayback Machine
- *Marjan Mernik.* *Formal and Practical Aspects of Domain-Specific Languages* (<http://sharebookfree.com/formal-and-practical-aspects-of-domain-specific-languages-recent-developments/>). — IGI Global, 2012. — ISBN 978-1-4666-2092-6.

## Ссылки

- *The Language List* (<https://web.archive.org/web/20040612042127/http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>) (недоступная ссылка — *история* ([https://web.archive.org/web/\\*/http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm](https://web.archive.org/web/*/http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm))) (англ.) — более 2500 языков с кратким описанием

- [Computer Languages History \(http://www.levenez.com/lang/\)](http://www.levenez.com/lang/) (англ.) — история языков программирования (с 1954 по май 2004) (содержит регулярно обновляемую диаграмму)
- [Examples \(https://web.archive.org/web/20040404130307/http://www.ntecs.de/old-hp/uu9r/lang/html/lang.en.html\)](https://web.archive.org/web/20040404130307/http://www.ntecs.de/old-hp/uu9r/lang/html/lang.en.html) (недоступная ссылка — *история* ([https://web.archive.org/web/\\*/http://www.ntecs.de/old-hp/uu9r/lang/html/lang.en.html](https://web.archive.org/web/*/http://www.ntecs.de/old-hp/uu9r/lang/html/lang.en.html))) (англ.) — примеры программирования на 162 языках
- [Programming Language Popularity \(https://web.archive.org/web/20140226220709/http://langpop.com/\)](https://web.archive.org/web/20140226220709/http://langpop.com/) (англ.) — регулярно обновляемое исследование популярности языков программирования
- [10 языков программирования, которые стоит изучать \(2006 г.\) \(http://www.realcoding.net/articles/10-yazykov-programmirovanie-kotorye-stoit-izuchat.html\)](http://www.realcoding.net/articles/10-yazykov-programmirovanie-kotorye-stoit-izuchat.html)
- [Programming Community Index \(http://www.tiobe.com/tpci.htm\)](http://www.tiobe.com/tpci.htm) (англ.) — регулярно обновляемый рейтинг популярности языков программирования
- [Computer Language Shootout Benchmarks \(https://web.archive.org/web/20090627025632/http://shootout.alioth.debian.org/\)](https://web.archive.org/web/20090627025632/http://shootout.alioth.debian.org/) (недоступная ссылка — *история* ([https://web.archive.org/web/\\*/http://shootout.alioth.debian.org/](https://web.archive.org/web/*/http://shootout.alioth.debian.org/))) (англ.) — сравнение языков программирования по эффективности
- [Programming Languages that are Loved \(http://bluebones.net/2004/04/programming-languages-that-are-loved/\)](http://bluebones.net/2004/04/programming-languages-that-are-loved/) (англ.) — сравнение языков программирования по «любви» и «ненависти» к ним
- *Брайан Керниган*. Why Pascal is Not My Favorite Programming Language. — 1981.
- *Ehud Lamm*. Hidden complexities of tail-call/tail-recursion optimization (<http://lambda-the-ultimate.org/classic/message1532.html>). LtU Classic Archives (англ.). Lambda the Ultimate (7 December 2003). Дата обращения: 30 ноября 2016.

---

Источник — [https://ru.wikipedia.org/w/index.php?title=Язык\\_программирования&oldid=113833716](https://ru.wikipedia.org/w/index.php?title=Язык_программирования&oldid=113833716)

---

Эта страница в последний раз была отредактирована 25 апреля 2021 в 23:54.

Текст доступен по лицензии Creative Commons Attribution-ShareAlike; в отдельных случаях могут действовать дополнительные условия.

Wikipedia® — зарегистрированный товарный знак некоммерческой организации Wikimedia Foundation, Inc.