Языки программирования. Краткий обзор

Язы́к программи́рования — формальная знаковая система, предназначенная для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и семантических правил, определяющих внешний вид программы и действия, которые выполнит исполнитель (обычно — ЭВМ) под её управлением.

История развития или как мы до этого дожили

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, механические пианино и ткацкие станки. 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 написала программу для вычисления чисел Бернулли для Аналитической машины Чарльза Бэббиджа, ставшей бы, в случае реализации, первым компьютером — хотя и механическим, с паровым двигателем — в мире.

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

Первые [программы](https://ru.wikipedia.org/wiki/Компьютерная_программа) заключались в установке ключевых переключателей на передней панели вычислительного устройства. Очевидно, таким способом можно было составить только небольшие программы. Одну из первых попыток создать полноценный язык программирования предпринял немецкий учёный [Конрад Цузе](https://ru.wikipedia.org/wiki/Цузе,_Конрад), который в период с 1943 по 1945 год разработал язык [**Plankalkül**](https://ru.wikipedia.org/wiki/Plankalkül)**(Планкалкюль)**. Это был очень перспективный язык, фактически являвшийся языком высокого уровня, однако из-за [военных действий](https://ru.wikipedia.org/wiki/Вторая_мировая_война) он не получил практической реализации, а его описание было опубликовано только в 1972 году.

**Машинный язык**

Неизвестно, насколько бы ускорилось развитие программирования, если бы наработки Цузе стали доступны другим учёным в конце 40-х годов, но на практике с развитием компьютерной техники сначала получил распространение машинный язык. С его помощью программист мог задавать команды, оперируя с ячейками памяти, полностью используя возможности машины. Суть этого языка — набор кодов, обязательно понятных процессору, к кому обращаются. Части («слова») этого языка называются инструкциями, каждая из которых представляет собой одно элементарное действие для центрального процессора, как, например, считывание информации из ячейки памяти. Лишь при понимании устройства компьютерного оборудования и знания этих целочисленных кодов можно было непосредственно управлять процессором. Тогда ещё компьютеры были простыми вычислительными машинами, применяемыми для различных математических расчётов. Но они развивались, а использование большинства компьютеров на уровне машинного языка затруднительно, особенно сложным было чтение и модификация подобных программ, что усугублялось использованием абсолютной адресации памяти. Поэтому со временем от использования машинных кодов пришлось отказаться.

Например, для организации чтения блока данных с гибкого диска программист может использовать 16 различных команд, каждая из которых требует 13 параметров, таких как номер блока на диске, номер сектора на дорожке и т.п. Когда выполнение операции с диском завершается, контроллер возвращает 23 значения, отражающие наличие и типы ошибок, которые необходимо анализировать. Уже одно обращение к процессору громоздко, а анализ ошибок и вовсе представляется невообразимым, особенно, если не именно с этим процессором приходиться работать. Таким образом, набор команд машинного языка сильно зависит от типа процессора.

**Язык ассемблера**

На протяжении 1950-х годов запросы на разработку программного обеспечения возросли и программы стали очень большими. Приходилось писать очень много кода, хотя обеспечение и было весьма простым: по тем временам дизайн рабочего стола был проще нынешнего, программы работали с элементарными вещами, а компьютер только ещё начинал победно шествовать. Однако программы запутывались всё больше, их структура усложнилась, потому что всё время развивалась компьютерная техника. Тогда стали пользоваться специальными программами-сборщиками программ из маленьких кусочков кодов — ассемблерами. Начался новый этап развития.

Теперь, когда была нужна эффективная программа, вместо машинных языков использовались близкие к ним машинно-ориентированные языки ассемблера. К таковым относились, например, Autocode, с 1954-го г. — IPL (предшественник языка LISP) и, с 1955-го г. — FLOW-MATIC (предшественник языка COBOL). Теперь люди стали использовать мнемонические команды взамен машинных команд.

Но даже работа с ассемблером достаточно сложна и требует специальной подготовки. Например, для процессора Zilog Z80 машинная команда 00000101 предписывает процессору уменьшить на единицу свой регистр B. На языке ассемблера это же будет записано как DEC B.

**Языки высокого уровня**

Следующий шаг был сделан в 1954 году, когда была начата разработка языка высокого уровня — Фортран (англ. FORTRAN — FORmula TRANslator), компилятор для которого впервые появился в апреле 1957 года. К разработке такого языка подтолкнули новые возможности внедрённого в 1954 году компьютера IBM 704, в котором на аппаратном уровне были реализованы индексная адресация и операции с плавающей точкой. Вслед за ним появились и некоторые другие языки, например: LISP, ALGOL 58, FACT (ещё один предшественник языка COBOL). Языки высокого уровня имитируют естественные языки, используя некоторые слова разговорного языка и общепринятые математические символы. Эти языки более удобны для человека, с помощью них можно писать программы до нескольких тысяч строк длиной. Условными словами можно было, как привычно человеку, гораздо более просто выразить сложную программную операцию из битов. Однако ранние варианты Фортрана значительно уступают поздним концепциям и языкам, использовался он для создания относительно простых по современным меркам программ.

Во второй половине 50-х интернациональная команда разработчиков попыталась создать универсальный язык программирования. В результате появился ALGOL 58 (англ. ALGOrithmic Language), по многим параметрам являвшийся наследником Фортрана. В него были добавлены новые концепции и обобщения, формализована концепция типов данных, разрешено использование идентификаторов любой длины, когда в Фортране было ограничение в 6 символов. Этот вариант языка был скорее черновым, поэтому в январе 1960 года в Париже состоялось второе собрание комитета по его разработке, где было решено внести значительные изменения. Новый вариант получил название ALGOL 60, основными новшествами в нём были: концепция блочной структуры, возможность создания рекурсивных процедур, автоматические массивы. Несмотря на свои многочисленные достоинства, ALGOL так и не получил большого распространения, в первую очередь из-за сложности в его реализации и отсутствии поддержки от корпорации IBM.

В дальнейшем появились COBOL (1959), Паскаль (1970), Си (1972).

**Появление структурного программирования**

К концу 1960-х годов в связи с ростом сложности программ и дальнейшим развитием программных средств возникла необходимость увеличить производительность труда программистов, что привело к разработке структурного программирования. Основоположником данной методологии считается Эдсгер Дейкстра, который в 1968 году опубликовал своё знаменитое письмо «Оператор Goto считается вредным», а также описал основные принципы структурного программирования. Впервые подобные сомнения высказал Хайнц Земанек (Heinz Zemanek) на совещании по языку [Алгол](https://ru.wikipedia.org/wiki/Алгол) в начале 1959 года в Копенгагене. С развитием структурного программирования следующим достижением были процедуры и функции. То есть, если есть задача, которая выполняется несколько раз, то её можно объявить как функцию или как процедуру и в выполнении программы просто вызывать её. Общий код программы в данном случае становится меньше. Это способствовало созданию модульных программ.

Код с goto трудно форматировать, так как он может нарушать иерархичность выполнения (парадигму [структурного программирования](https://ru.wikipedia.org/wiki/Структурное_программирование)) и потому отступы, призванные отображать структуру программы, не всегда могут быть выставлены правильно. Goto также мешает [оптимизации компиляторами](https://ru.wikipedia.org/wiki/Оптимизация_компилятора) управляющих структур.

Следующим достижением было объединение разнородных данных, которые используются в программе в связке, в структуры.

Структуры — это составные типы данных, построенные с использованием других типов данных. Например, структура времени разбивается на: часы, минуты, секунды. В свою очередь и часы, и минуты, и секунды описаны при помощи более простых и элементарных типов данных. И в место работы с отдельными переменными, в которых легко запутаться, можно перейти к структуре «время», включающее в себя уже часы, минуты и секунды, и работать с ней, как с единым типом одного формата.

Структурное программирование предполагает точно обозначенные управляющие структуры, программные блоки, отсутствие инструкций безусловного перехода (GOTO), автономные подпрограммы, поддержка рекурсии и локальных переменных. Суть такого подхода заключается в возможности разбиения программы на составляющие элементы с увеличением читабельности программного кода.

Также создавались функциональные (аппликативные) языки (Пример: Lisp — англ. LISt Processing, 1958) и логические языки (пример: Prolog — англ. PROgramming in LOGic, 1972).

Хотя внедрение структурного программирование дало положительный результат, даже оно оказывалось несостоятельным тогда, когда программа достигала определённой длины. Для того чтобы написать более сложную и длинную программу, нужен был новый подход к программированию.

**ООП**

При использовании структур данных в программе вырабатываются и соответствующие им функции для работы с ними. это привело к мысли их объединить и использовать совместно, так появились классы.

Класс — это структура данных, содержащая в себе не только переменные, но и функции, которые работают с этими переменными.

Коротко, это достижение в области программирования было очень велико. Теперь программирование можно было разбить на классы и тестировать не всю программу, состоящую из 10 000 строк кода, а разбить программу на 100 классов, и тестировать каждый класс. Это существенно облегчило написание программного продукта.

В итоге в конце 1970-х и начале 1980-х были разработаны принципы объектно-ориентированного программирования. ООП сочетает лучшие принципы структурного программирования с новыми концепциями инкапсуляции, полиморфизма подтипов и наследования.

Первым объектно-ориентированным языком программирования является Симула-67, в котором впервые появились классы. Концепции ООП получили дальнейшее развитие в языке Smalltalk, в котором также были заложены основы систем с оконным управлением. Более поздними примерами объектно-ориентированных языков являются Object Pascal, C++, Java, C# и др.

ООП позволяет оптимально организовывать программы, разбивая проблему на составные части, и работая с каждой по отдельности. Программа на объектно-ориентированном языке, решая некоторую задачу, по сути, описывает часть мира, относящуюся к этой задаче.

Стандартизация языков программирования

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.

**Типы данных**

Современные цифровые компьютеры являются двоичными и данные хранят в двоичном (бинарном) коде (хотя возможны реализации и в других системах счисления). Эти данные как правило отражают информацию из реального мира (имена, банковские счета, измерения и др.), представляющую высокоуровневые концепции.

Особая система, по которой данные организуются в программе, — это система типов языка программирования; разработка и изучение систем типов известна под названием теория типов. Языки можно поделить на имеющие *статическую типизацию* и *динамическую типизацию*, а также *бестиповые языки* (например, Forth).

Статически типизированные языки могут быть в дальнейшем подразделены на языки с *обязательной декларацией*, где каждая переменная и объявление функции имеет обязательное объявление типа, и *языки с выводимыми типами*.

**Структуры данных**

Системы типов в языках высокого уровня позволяют определять сложные, составные типы, так называемые структуры данных. Как правило, структурные типы данных образуются как декартово произведение базовых (атомарных) типов и ранее определённых составных типов.

Основные структуры данных (списки, очереди, хеш-таблицы, двоичные деревья и пары) часто представлены особыми синтаксическими конструкциями в языках высокого уровня. Такие данные структурируются автоматически.

**Семантика языков программирования**

Существует несколько подходов к определению семантики языков программирования.

Наиболее широко распространены разновидности следующих трёх: операционного, деривационного (*аксиоматического*) и денотационного (*математического*).

При описании семантики в рамках операционного подхода обычно исполнение конструкций языка программирования интерпретируется с помощью некоторой воображаемой (абстрактной) ЭВМ.

Аксиоматическая (Деривационная) 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.

**Парадигма программирования**

Язык программирования строится в соответствии с той или иной *базовой моделью вычислений и парадигмой программирования*.

Несмотря на то, что большинство языков ориентировано на *императивную модель вычислений*, задаваемую фон-неймановской архитектурой ЭВМ, существуют и другие подходы. Можно упомянуть языки со *стековой вычислительной моделью* (Форт, Factor, PostScript и др.), а также функциональное (Лисп, Haskell, ML, F#, РЕФАЛ, основанный на модели вычислений, введённой советским математиком А. А. Марковым-младшим и др.) и логическое программирование (Пролог).

В настоящее время также активно развиваются декларативные и визуальные языки программирования, а также методы и средства разработки проблемно-специфичных языков.

**Способы реализации языков**

Языки программирования могут быть реализованы как *компилируемые,* *интерпретируемые и встраиваемые.*

Программа на к*омпилируемом* языке при помощи **компилятора** (особой программы) преобразуется (компилируется) в машинный код (набор инструкций) для данного типа процессора и далее собирается в исполнимый модуль, который может быть запущен на исполнение как отдельная программа. Другими словами, компилятор переводит исходный текст программы с языка программирования высокого уровня в двоичные коды инструкций процессора.

Если программа написана на интерпретируемом языке, то интерпретатор непосредственно выполняет (интерпретирует) исходный текст без предварительного перевода. При этом программа остаётся на исходном языке и не может быть запущена без интерпретатора. Процессор компьютера, в этой связи, можно назвать интерпретатором для машинного кода.

Разделение на компилируемые и интерпретируемые языки является условным. Так, для любого традиционно компилируемого языка, как, например, Паскаль, можно написать интерпретатор. 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 макросов).

Для любого интерпретируемого языка можно создать компилятор — например, язык Лисп, изначально интерпретируемый, может компилироваться без каких бы то ни было ограничений. Создаваемый во время исполнения программы код может так же динамически компилироваться во время исполнения.

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 обладают некоторыми специфическими дополнительными возможностями (см. выше), кроме того, программы на них можно запускать сразу же после изменения, что облегчает разработку. Программа на интерпретируемом языке может быть зачастую запущена на разных типах машин и операционных систем без дополнительных усилий.

Однако интерпретируемые программы выполняются заметно медленнее, чем компилируемые, кроме того, они не могут выполняться без программы-интерпретатора.

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 исполняется 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. Следует упомянуть, что есть языки, имеющие и интерпретатор, и компилятор (Форт).

**Языки программирования низкого уровня**

Первые компьютеры приходилось программировать двоичными машинными кодами. Однако программировать таким образом — довольно трудоемкая и тяжелая задача. Для упрощения этой задачи начали появляться языки программирования низкого уровня, которые позволяли задавать машинные команды в понятном для человека виде. Для преобразования их в двоичный код были созданы специальные программы — *трансляторы.*

**Трансляторы** делятся на:

*компиляторы* — превращают текст программы в машинный код, который можно сохранить и после этого использовать уже без компилятора (примером является исполняемые файлы с расширением \*.exe).

*интерпретаторы* — превращают часть программы в машинный код, выполняют его и после этого переходят к следующей части. При этом каждый раз при выполнении программы используется интерпретатор.

Примером языка низкого уровня является ассемблер. Языки низкого уровня ориентированы на конкретный тип процессора и учитывают его особенности, поэтому для переноса программы на ассемблере на другую аппаратную платформу её нужно почти полностью переписать. Определенные различия есть и в синтаксисе программ под разные компиляторы. Правда, центральные процессоры для компьютеров фирм AMD и Intel практически совместимы и отличаются лишь некоторыми специфическими командами. А вот специализированные процессоры для других устройств, например, видеокарт и телефонов содержат существенные различия.

Языки низкого уровня, как правило, используют для написания небольших системных программ, драйверов устройств, модулей стыков с нестандартным оборудованием, программирование специализированных микропроцессоров, когда важнейшими требованиями являются компактность, быстродействие и возможность прямого доступа к аппаратным ресурсам.

Ассемблер — язык низкого уровня, широко применяется до сих пор.

Основным недостатком является меньшая производительность по сравнению с компилируемыми языками, преобразуемыми в [машинный код](https://ru.wikipedia.org/wiki/Машинный_код). Трансляция в [байт-код](https://ru.wikipedia.org/wiki/Байт-код) и [IT-компиляция](https://ru.wikipedia.org/wiki/JIT-компиляция) не решают этой проблемы полностью. Дополнительный слой [интерпретатора](https://ru.wikipedia.org/wiki/Интерпретатор) или [виртуальной машины](https://ru.wikipedia.org/wiki/Виртуальная_машина) замедляет выполнение программы и может требовать больше ресурсов.

Из-за отсутствия компиляции всего проекта большинство ошибок можно поймать только во время выполнения. В результате в среднем интерпретируемый код следует тестировать тщательнее компилируемого, строже придерживаться соглашений по оформлению программ и использовать дополнительные анализаторы качества кода. Последний недостаток выражен несильно, так как при серьезной разработке на компилируемых языках также необходимо применение этих средств.

**Языки программирования высокого уровня**

Особенности конкретных компьютерных архитектур в них не учитываются, поэтому созданные приложения легко переносятся с компьютера на компьютер. В большинстве случаев достаточно просто перекомпилировать программу под определенную компьютерную архитектурную и операционную систему. Разрабатывать программы на таких языках значительно проще и ошибок допускается меньше. Значительно сокращается время разработки программы, что особенно важно при работе над большими программными проектами.

Сейчас в среде разработчиков считается, что языки программирования, которые имеют прямой доступ к памяти и регистров или имеют ассемблерные вставки, нужно считать языками программирования с низким уровнем абстракции. Поэтому большинство языков, считавшихся языками высокого уровня до 2000 года сейчас уже таковыми не считаются.

Адресный язык программирования

Фортран

Pascal

Pascal ABC

Python

Java

C

Basic

C++

Objective-C

Smalltalk

C#

Delphi

Недостатком некоторых языков высокого уровня является большой размер программ в сравнении с программами на языках низкого уровня. С другой стороны, для алгоритмически и структурно сложных программ при использовании суперкомпиляции преимущество может быть на стороне языков высокого уровня. Сам текст программ на языке высокого уровня меньше, однако, если взять в байтах, то код, изначально написанный на ассемблере, будет более компактным. Поэтому в основном языки высокого уровня используются для разработки программного обеспечения компьютеров и устройств, которые имеют большой объём памяти. А разные подвиды ассемблера применяются для программирования других устройств, где критичным является размер программы.

Статистика

1. Java

Java является одним из самых популярных языков для бэкэнд-разработки современных корпоративных веб-приложений. С Java и основанными на нём фреймворками разработчики могут создавать масштабируемые веб-приложения для широкого круга пользователей. Java — также основной язык, используемый для разработки родных Android-приложений для смартфонов и планшетов.

2. JavaScript

Каждый современный сайт использует JavaScript. Это ключевой язык для создания интерактивности сайта или построения пользовательских интерфейсов с одним из десятка популярных JavaScript-фреймворков.

3. C#

C # является основным языком для разработки на платформах и сервисах Microsoft. Будь то разработка современных веб-приложений с использованием Azure и .NET, приложений для «девайсов» Windows или мощных «настольных» приложений для бизнеса, C# — самый быстрый способ использовать всё, что может предложить Microsoft. Кроме того, это и один из основных языков движка для разработки игр Unity.

4. PHP

Пишите веб-приложение для работы с данными? Язык PHP наряду с базами данных (например, MySQL) является важным инструментом для создания современных веб-приложений. На PHP разработано большинство сайтов, ориентированных на большой объём данных. Это также основополагающая технология мощных систем управления контентом, как WordPress.

5. С++

Если для максимальной отдачи мощности процессора вам необходимо подключиться непосредственно к железу, поможет язык C++. Это идеальный выбор для разработки мощного «настольного» программного обеспечения, игр с функцией аппаратного ускорения, а также приложений для ПК, консолей и мобильных устройств, требующих большого объёма памяти для работы.

6. Python

Python может сделать почти всё вышеперечисленное. Веб-приложения, пользовательские интерфейсы, анализ данных, статистика — для какой бы задачи вам не предстояло найти решение, в Python, скорее всего, найдётся подходящий фреймфорк. Совсем недавно учёные пришли к выводу, что Python модно использовать в качестве основного инструмента для обработки гигантских объёмов данных практически в любой отрасли.

7. C

Почему язык C по-прежнему популярен? Из-за размера: маленький, быстрый и мощный. Если вы разрабатываете программное обеспечение для встраиваемых систем, работаете с системными ядрами или просто хотите выжать из имеющихся по рукой ресурсов всё до последней капли, C — то, что нужно.

8. SQL

Данные — всеобъемлющие и всепроникающие. SQL даёт возможность найти необходимую информацию быстрым и надёжным способом. Используя SQL, вы можете легко запрашивать и извлекать значительные объёмы данных из больших и сложных баз данных.

9. Ruby

Хотите запустить проект в рекордно короткие сроки или создать прототип новой идеи для крутого веб-приложения? С помощью Ruby (и Ruby on Rails) это возможно довольно быстро. Обладая невероятной мощностью, язык прост в освоении. Плюс на нём написаны тонны популярных веб-приложений по всему миру.

10. Objective-C

Собираетесь написать приложение для iOS? Тогда вы просто обязаны знать Objective-C. Несмотря на прошлогоднюю шумиху вокруг нового языка Apple Swift, Objective-C по-прежнему остаётся основополагающим языком приложений для экосистемы Apple. С Objective-C и официальным инструментом разработки ПО от Apple XCode до App Store — рукой подать.

11. Perl

Можно ли назвать Perl эзотерическим языком? Да. Сбивает ли он с толку? Да. Является ли он супермощным языком и ключевым компонентом в арсенале кибербезопасности? Снова да. Разработчики используют Perl с самых истоков интернета, и он до сих пор считается ключевым инструментом для любого ИТ-специалиста.

12. .NET

Хотя и не язык сам по себе, .NET является ключевой платформой Microsoft для разработки облачных и не очень сервисов и приложений. Становится более продвинутым и ценным с каждым новым релизом. Благодаря последним усилиям Microsoft в области разработки с открытым исходным кодом, .NET теперь приходит на платформы Google и Apple. Как результат, вы можете использовать .NET с различными языками программирования для создания мультиплатформенных приложений.

13. Visual Basic

Ключевой язык .NET-платформы, Visual Basic позволяет создавать приложения для поддержки бизнеса, а также автоматизировать мощные приложения MSOffice.

14. R

R двигает революцию больших данных. В 2015 году это обязательный язык для всех, кто нуждается в серьёзном анализе данных: от сфер науки и бизнеса до развлечений и социальных медиа.

15. Swift

За менее чем год существования язык программирования Swift привлёк внимание разработчиков во всём мире как новый, простой и быстрый способ разработки для операционных систем OS X и iOS. Широкие полномочия и дружественный синтаксис Swift позволяют написать очередное убойное приложение для пользователей Apple.

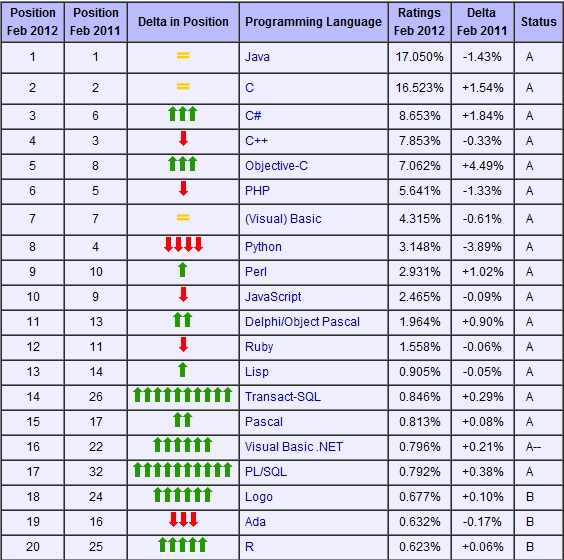
Опираясь на статистику и индекс популярности ЯП, можно однозначно говорить о том, что будущее за динамическими языками программирования(с динамической типизацией). Динамический язык - это такой язык программирования и такой транслятор, которые позволяют определять типы данных и осуществлять синтаксический анализ и трансляцию "на лету", непосредственно на этапе выполнения.

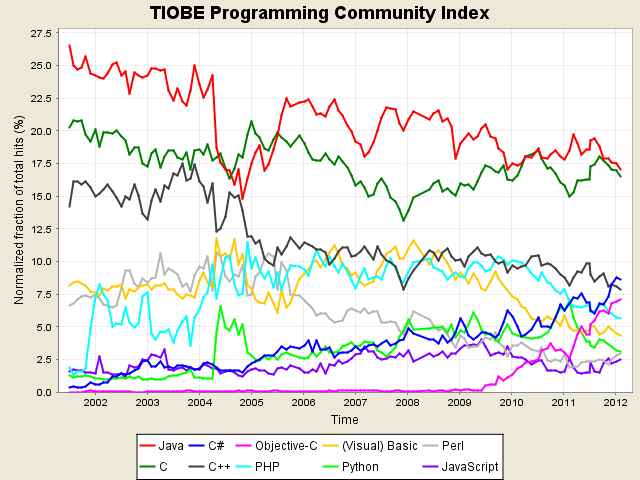
(JavaScript, Python, Ruby, Basic, Tcl, Smaltalk, PHP) Forth - безтиповый язык.

В этом заключается главное преимущество динамических ЯП.

Вместе с языками активно набирающих всеобщую популярность, существуют и вымирающие или мертвые языки. К таким можно отнести все семейство COBOL, ada, Haskel, assembler, Basic.

Отдельным особняком стоят визуальные языки программирования.



****