Классификация языков программирования

Существует множество критериев, по которым можно классифицировать языки программирования. Частые варианты классификации включают:

- По парадигме (декларативные, императивные, структурированные и т.п.)
- По системе типов (динамические, статические, сильно- и слаботипизированные, нетипизированные и т.п.)
- По степени зависимости от аппаратных средств (высокого, низкого уровня)
- По модели исполнения (компилируемые, интерпретируемые)
- По «поколению»

Чёткой классификации не существует, по той простой причине, что существуют буквально тысячи ЯП, и в любой категории классификации обнаруживается практически непрерывный спектр.

По системе типов

Нетипизированные языки

позволяют производить любую возможную операцию над любыми данными. Это обычно какие-либо языки ассемблера, которые работают непосредственно с двоичным представлением данных в памяти.

С точки зрения теории типов очень немногие из современных языков являются типизированными в полном смысле этого слова. Большинство являются типизированными *в некоторой мере*. Так, многие языки позволяют выходить за пределы системы типов, принося типобезопасность в жертву более точному управлению исполнением программы.

Типизированные языки

определяют типы данных, с которыми работает любая операция. Например, операция деления работает над числами – для строк эта операция не определена.

Типизированные языки, в свою очередь, могут классифицироваться по моменту проверки типов и по строгости этой проверки.

По моменту проверки типов ЯП делятся на статически и динамически типизированные (или просто, статические и динамические).

Статически типизированные языки. При статической типизации, типы всех выражений точно определены до выполнения программы, и обычно проверяются при компиляции.

Языки со статической типизацией, в свою очередь могут быть

- *явно типизированными* (manifestly typed), они требуют явного указания типов. К ним относятся, например, C, C++, C#, Java
- *типовыводящими* (type-inferred), они определяют (выводят) типы большинства выражений автоматически, и требуют явного аннотирования только в сложных и неоднозначных случаях, к ним относятся, например, Haskell и OCaml.

Надо заметить, что многие явно типизированные языки умеют выводить типы в *некоторых* случаях (например, auto в C++), поэтому чёткую грань здесь провести можно не всегда.

Статическая типизация:

- ✓ требует больше усилий при разработке программы, необходимо явно задавать типы объектов
- ✓ программа (существенно) надежнее: много ошибок выявляются до выполнения программы (в процессе компиляции)
- ✓ программы читабельнее; их легче читать, понимать и сопровождать

Динамически типизированные языки производят проверку типов на этапе выполнения. Иначе говоря, типы связаны *со значением при выполнении*, а не с *текстовым выражением*. Как и типовыводящие языки, динамически типизированные не требуют указания типов выражений. Помимо прочего, это позволяет одной переменной иметь значения разных типов в разные моменты исполнения программы. Однако ошибки типов не могут быть автоматически обнаружены, пока фрагмент кода не будет выполнен. Это усложняет отладку и несколько подрывает идею типобезопасности в целом. Примерами динамически типизированных языков являются Lisp, Perl, Python, JavaScript, Ruby.

Динамическая типизация:

- ✓ программу писать легче: нет нужды заботиться о типах объектов
- ✓ программа более гибкая: не нужно вводить новые объекты для различных целей
- ✓ программа нечитабельная; для ее понимания и сопровождения требуется больше усилий
- ✓ программы ненадежны и неэффективны

По строгости типизации языки делятся на

- слабо типизированные – неявно конвертируют один тип в другой, скажем, строки в числа и наоборот. Это может быть удобно в некоторых случаях, однако многие программные ошибки могут быть пропущены. Усложняется отладка.
- сильно типизированные – не позволяют неявную конверсию, и требуют явной. Дают сильные гарантии типобезопасности, но код может становиться крайне многословным.

В целом, чёткую грань провести оказывается опять-таки достаточно сложно, поскольку неявное преобразование типов в той или иной мере производится в большинстве языков. Однозначно к слабо типизированным относят Perl, JavaScript и C (в силу свободной конверсии void*). К сильно типизированным относят C++, Java, Haskell, Ada и другие.

По степени зависимости от аппаратных средств

Низкоуровневые языки программирования ориентируются на конкретный тип процессора, с учетом архитектуры и технических особенностей компьютера. Их еще называют *машинно-ориентированными*. Раньше почти для каждого типа процессора существовал свой низкоуровневый язык программирования и программу, написанную для одного типа, нельзя было использовать для другого. Языки низкого уровня – это машинный код и языки ассемблера.

Языки программирования высокого уровня (*машинно-независимые*) – языки, на которых программы могут использоваться на компьютерах различных типов и которые более доступны человеку, чем языки низкого уровня.

Языки высокого уровня могут значительно упрощать реализацию сложных алгоритмов, однако, написанные на них программы потенциально менее производительны.

По модели исполнения

ЯП может быть компилируемым, транс-компилируемым или интерпретируемым.

Интерпретируемые языки исполняются непосредственно, без этапа компиляции. Программа, называемая *интерпретатором*, читает каждое выражение, определяет сообразное действие, и совершает его. Гибридный вариант может генерировать машинный код «на лету» и исполнять его. Интерпретируемые языки: PHP, Perl, Bash, Python, JavaScript.

Компилируемый язык компилируется, т.е. переводится в исполнимую форму до выполнения. Компиляция может производиться непосредственно в машинный код, или в какое-либо промежуточное представление (байт-код), которое потом интерпретируется виртуальной машиной. Компилируемые языки (машинный код): Ada, C, C++, Algol, Go, Delphi, Fortran, Pascal и др. Компилируемые языки (байт-код): Java, Kotlin, Scala, C#, Erlang и др.

Транс-компилируемые языки – это языки, которые для компиляции или выполнения транслируются в другой язык. Частой целью для транс-компилируемых языков является C. Также последнее время популярной целью является JavaScript (как единственный язык, исполняемый в браузере).
Транс-компилируемые языки: C++ (исторически, в C), Haskell (исторически, в C), Fortran (иногда, в C), Dart, TypeScript, Elm (в JavaScript)

Линии сильно размыты, поскольку существуют компиляторы для традиционно интерпретируемых языков, и, напротив, интерпретаторы для традиционно компилируемых.

По «поколению»

Поколение – несколько условная характеристика, которая в значительной мере связана с историей появления современных языков программирования.

Языки первого поколения (1GL)

(1GL – *first-generation programming language*)

Это машинные языки. Исторически, программы на этих языках вводились при помощи переключателей на передней панели ЭВМ, либо «писались» на перфокартах и позже перфолентах. Программа на 1GL состоит из 0 и 1 и сильно привязана к конкретному железу, на котором она должна исполняться.

Языки второго поколения (2GL)

Это общая категория для различных языков ассемблера. С одной стороны, код языков 2GL может читать человек, и он должен быть конвертирован в машиночитаемую форму (этот процесс называется ассемблированием, или сборкой). С другой стороны, этот язык специфичен к процессору и прочему аппаратному окружению.

Языки третьего поколения (3GL)

1950-е годы. Более абстрактные, чем 2GL, это языки, которые перекладывают заботу о непринципиальных деталях с плеч программиста на плечи компьютера. Fortran, Algol и Cobol являются первыми 3GL. C, C++, Java, BASIC и Pascal так же могут быть отнесены к 3GL, хотя в общем 3GL подразумевает только структурную парадигму (в то время как C++, Java работают в том числе в ООП).

Языки четвертого поколения (4GL)

С 1970-х по начало 1990-х годов. Определение несколько расплывчато, однако в целом сводится к еще более высокому уровню абстракции, чем 3GL. Однако, подобный уровень абстракции часто требует сужения области применения. Так, например, FoxPro, LabView G, SQL, Simulink являются 4GL, однако находят применение в узкой специфической области. Некоторые исследователи считают, что 4GL являются подмножеством DSL (domain specific language, язык, специфичный к области, или проблемно-ориентированный).

Языки пятого поколения (5GL)

В конце 80-х – начале 90-х была попытка разработать класс языков, которые «пишут программы сами». По идее, программист должен был описывать как программа должна себя вести, а остальное должен был делать компьютер. К примерам можно отнести Prolog, OPS5, Mercury. Хотя эти языки остаются интересными с теоретической точки зрения, широкого практического применения они пока не нашли.

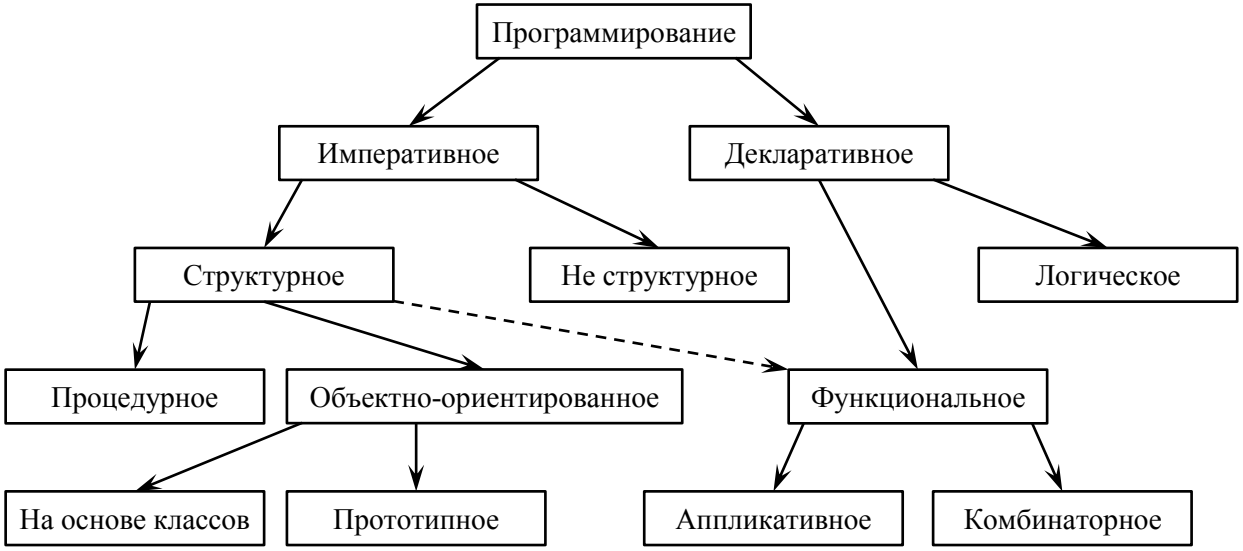
Инструкции вводятся в компьютер в максимально наглядном виде (часто с помощью визуальных средств разработки) с помощью методов, наиболее удобных для человека, не знакомого с программированием. На данный момент стоит отметить также такие понятия, как *lowcode* и даже *zerocode* программирование, а также генерацию программ с помощью искусственного интеллекта, например, ChatGPT.

По парадигме программирования

Парадигма программирования – это набор концепций, правил и абстракций, определяющих стиль программирования. В соответствии с ними в каждой парадигме заложен подход к использованию ключевых конструкций.

Парадигма программирования не определяется однозначно языком программирования; практически все современные языки программирования в той или иной мере допускают использование различных парадигм.

Так на языке C, который не является объектно-ориентированным, можно работать в соответствии с принципами объектно-ориентированного программирования, хотя это и сопряжено с определёнными сложностями; функциональное программирование можно применять при работе на любом императивном языке, в котором имеются функции (для этого достаточно не применять присваивание), и т.д.


Программирование
Императивное
Декларативное
Структурное
Не структурное
Логическое
Процедурное
Объектно-ориентированное
Функциональное
На основе классов
Прототипное
Аппликативное
Комбинаторное

Императивное программирование

Императивное программирование характеризуется в основном:

- в исходном коде программы записываются инструкции (команды);
- инструкции должны выполняться последовательно;
- каждая инструкция может изменять некое глобальное “состояние” программы

При императивном подходе к составлению кода (в отличие от функционального подхода, относящегося к декларативной парадигме) широко используется присваивание. Наличие операторов присваивания увеличивает сложность модели вычислений и делает императивные программы подверженными специфическим ошибкам, не встречающимся при функциональном подходе.

Практически всё аппаратное обеспечение в основе своей императивное.

Неструктурное программирование

Характерно для наиболее ранних языков программирования.

В основном характеризуется:

- строки как правило нумеруются
- из любого места программы возможен переход к любой строке

Характерной особенностью неструктурного программирования является сложность реализации рекурсии.

Структурное программирование

В отличие от неструктурного программирования, характеризуется:

- ограниченным использованием условных и безусловных переходов
- широким использованием подпрограмм и прочих управляющих структур (циклов, ветвлений, и т.п.)
- блочной структурой

Концепция структурного программирования основана на теореме Бёма-Якопини:

Любая вычислимая функция может быть представлена комбинацией трёх управляющих структур:

- Последовательности
- Ветвления
- Итерации

Последовательность – это выполнение сначала одной подпрограммы, затем другой.

Ветвление – это выполнение либо одной, либо другой подпрограммы в зависимости от значения некого булева (логического) выражения.

Итерация – это многократное выполнение подпрограммы пока некое булево выражение истинно.

Процедурное программирование

Процедурное программирование можно рассматривать как небольшую вариацию на тему структурного программирования, основанную на концепции вызова процедуры.

Основная идея заключается в том, чтобы сделать подпрограммы более модульными за счёт:

- локальных переменных
- относительно простой рекурсии

Оба этих пункта реализуются за счёт использования стека вызовов.

Объектно-ориентированное программирование

Объектно-ориентированное программирование основано на концепции «объекта».

Объекты могут содержать данные (поля, свойства, атрибуты) и поведение (код, процедуры, методы). Наиболее популярной формой ООП является ООП на основе классов. В данном подходе, все объекты являются экземплярами классов, и классы определяют так же тип объектов.

Прототипное программирование – стиль объектно-ориентированного программирования, при котором отсутствует понятие класса, а наследование производится путём клонирования существующего экземпляра объекта — прототипа. Каноническим примером прототипно-ориентированного языка является язык Self. В дальнейшем этот стиль программирования начал обретать популярность и был положен в основу таких языков программирования, как JavaScript, Lua, Io, REBOL и др.

Парадигмы ООП, не показанные на рисунке

Компонентно-ориентированное программирование (англ. component-oriented programming, COP) – парадигма программирования, существенным образом опирающаяся на понятие компонента – независимого модуля исходного кода программы, предназначенного для повторного использования и развёртывания и реализующегося в виде множества языковых конструкций (например, «классов» в объектно-ориентированных языках программирования), объединённых по общему признаку и организованных в соответствии с определёнными правилами и ограничениями.

Аспектно-ориентированное программирование (АОП) – парадигма программирования, основанная на идее разделения функциональности для улучшения разбиения программы на модули. Методология АОП была предложена группой инженеров исследовательского центра Xerox PARC под руководством Грегора Кичалеса (Gregor Kiczales). Ими же было разработано аспектно-ориентированное расширение для языка Java, получившее название AspectJ — (2001 год).

Обобщённое программирование (англ. generic programming) – парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание. В том или ином виде поддерживается разными языками программирования. Возможности обобщённого программирования впервые появились в виде дженериков (обобщённых функций) в 1970-х годах в языках Клу и Ада, затем – в виде параметрического полиморфизма в ML и его потомках, а затем – во многих объектно-ориентированных языках, таких как C++, Python, Java, Object Pascal, D, Eiffel, языках для платформы .NET и других.

Декларативное программирование

Декларативное программирование — это парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат. Противоположностью декларативного является императивное программирование, описывающее на том или ином уровне детализации, как решить задачу и представить результат.

Как следствие, декларативные программы не используют понятия состояния, то есть не содержат переменных и операторов присваивания.

К подвидам декларативного программирования также зачастую относят функциональное и логическое программирование — несмотря на то, что программы на таких языках нередко содержат алгоритмические составляющие.

«Чисто декларативные» компьютерные языки зачастую не полны по Тьюрингу – примерами служат SQL и HTML — так как теоретически не всегда возможно порождение исполняемого кода по декларативному описанию. Это иногда приводит к спорам о корректности термина «декларативное программирование».

Функциональное программирование

Основные концепции:

- отсутствие неявных побочных эффектов
- ссылочная прозрачность
- отсутствие неявного состояния
- данные и функции – это концептуально одно и то же

Основано на лямбда-исчислении

Аппликативное

Аппликативное программирование – один из видов декларативного программирования, в котором написание программы состоит в систематическом осуществлении применения одного объекта к другому. Результатом такого применения вновь является объект, который может участвовать в применениях как в роли функции, так и в роли аргумента и так далее. Это делает запись программы математически ясной. Тот факт, что функция обозначается выражением, свидетельствует о возможности использования значений-функций – функциональных объектов – на равных правах с прочими объектами, которые можно передавать как аргументы, либо возвращать как результат вычисления других функций.

Примерами аппликативных языков программирования служат функциональные языки Лисп и ML. В языке Haskell эта парадигма программирования реализована в виде аппликативного функтора, расширяющего возможности механизма функциональной абстракции высших порядков до многоместной.

Комбинаторное

Комбинаторное программирование (англ. function-level programming) – парадигма программирования, использующая принципы комбинаторной логики.

Является особой разновидностью функционального программирования, но, в отличие от основного его направления, комбинаторное программирование не использует λ-абстракцию.

На практике это выливается в отсутствие «переменных», содержащих данные.

Концептуализировал и популяризовал парадигму Джон Бэкус в тьюринговской лекции 1977 года «Можно ли освободить программирование от стиля фон Неймана», в которой представил язык FP[en]. В конце 1980-х Бэкус с коллегами из Алмаденского исследовательского центра IBM в развитие идей FP и конкатенативной парадигмы разработали язык FL[en]. При этом элементы конткатенативного программирования проявляются уже в APL, а в более поздних его разновидностях – языках J и K – заимствованы многие идеи FP, и оформлены в концепцию бесточечного стиля, которая применима не только для функционального программирования в строгом смысле (в частности, элементы такого стиля имеют место в оболочках UNIX при применении конвейеров для перенаправления ввода-вывода).

Логическое программирование

Логическое программирование – парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздела дискретной математики, изучающего принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций.

Самым известным языком логического программирования является Prolog.