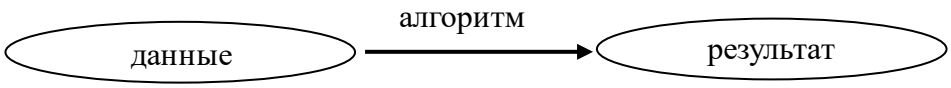
1. Введение

1.1 Алгоритм и программа

Программирование - процесс описания последовательности действий решения задачи средствами конкретного языка программирования и оформление результатов описания в виде программы. Эта работа требует точности, аккуратности и терпения. Команды машине должны формулироваться абсолютно четко и полно, не должны содержать никакой двусмысленности.

Алгоритм – точное предписание, определяющий вычислительный процесс, идущий от изменяемых начальных данных к конечному результату, т. е. это рецепт достижения какой-либо цели.



Совокупность средств и правил для представления алгоритма в виде пригодном для выполнения вычислительной машиной называется **языком программирования**, алгоритм, записанный на этом языке, называется **программой**.

Сначала всегда разрабатывается алгоритм действий, а потом он записывается на одном из языков программирования. Текст программы обрабатывается специальными служебными программами – **трансляторами**. Языки программирования – это искусственные языки. От естественных языков они отличаются ограниченным числом «слов» и очень строгими правилами записи команд (операторов). Совокупность этих требований образует синтаксис языка программирования, а смысл каждой конструкции – его семантику.

1.2 Свойства алгоритма

1. ***Массовость:*** алгоритм должен применяться не к одной задаче, а к целому классу подобных задач (алгоритм для решения квадратного уравнения должен решать не одно уравнение, а все квадратные уравнения).

2. ***Результативность:*** алгоритм должен приводить к получению результата за конкретное число шагов (при делении 1 на 3 получается периодическая дробь 0,3333(3), для достижения конечного результата надо оговорить точность получения этой дроби, например, до 4 знака после запятой).

3. ***Определенность (детерминированность)***: каждое действие алгоритма должно быть понятно его исполнителю (инструкция к бытовому прибору на японском языке для человека не владеющего японским языком не является алгоритмом, т .к не обладает свойством детерминированности).

4. ***Дискретность:*** процесс должен быть описан с помощью неделимых операций, выполняемых на каждом шаге (т. е. шаги нельзя разделить на более мелкие шаги).

Алгоритмы можно представить в следующих формах:

1) словесное описание алгоритма.
2) графическое описание алгоритма.
3) с помощью алгоритмического языка программирования

1.3 Компиляторы и интерпретаторы

С помощью языка программирования создается текст, описывающий ранее составленный алгоритм. Чтобы получить работающую программу, надо этот текст перевести в последовательность команд процессора, что выполняется при помощи специальных программ, которые называются **трансляторами**.

Трансляторы бывают двух видов: компиляторы и интерпретаторы.

Компилятор транслирует текст исходного модуля в машинный код, который называется объектным модулем за один непрерывный процесс. При этом сначала он просматривает исходный текст программы в поисках синтаксических ошибок.

Интерпретатор выполняет исходный модуль программы в режиме оператор за оператором, по ходу работы, переводя каждый оператор на машинный язык.

2. Языки программирования

Разные типы процессоров имеют разный набор команд. Если язык программирования ориентирован на конкретный тип процессора и учитывает его особенности, то он называется языком программирования низкого уровня. Языком самого низкого уровня является язык ассемблера, который просто представляет каждую команду машинного кода в виде специальных символьных обозначений, которые называются мнемониками. С помощью языков низкого уровня создаются очень эффективные и компактные программы, т .к разработчик получает доступ ко всем возможностям процессора. Т .к. наборы инструкций для разных моделей процессоров тоже

разные, то каждой модели процессора соответствует свой язык ассемблера, и написанная на нем программа может быть использована только в этой среде. Подобные языки применяют для написания небольших системных приложений, драйверов устройств и т. п..

Языки программирования высокого уровня не учитывают особенности конкретных компьютерных архитектур, поэтому создаваемые программы на уровне исходных текстов легко переносятся на другие платформы, если для них созданы соответствующие трансляторы. Разработка программ на языках высокого уровня гораздо проще, чем на машинных языках.

Языками высокого уровня являются:

1. ***Фортран*** – первый компилируемый язык, созданный в 50-е годы 20 века. В нем были реализован ряд важнейших понятий программирования. Для этого языка было создано огромное количество библиотек, начиная от статистических комплексов и заканчивая управлением спутниками, поэтому он продолжает использоваться во многих организациях.
2. ***Кобол*** – компилируемый язык для экономических расчетов и решения бизнес-задач, разработанный в начале 60-х годов. В Коболе были реализованы очень мощные средства работы с большими объемами данных, хранящихся на внешних носителях.
3. ***Паскаль*** – создан в конце 70-х годов швейцарским математиком Никлаусом Виртом специально для обучению программированию. Он позволяет выработать алгоритмическое мышление, строить короткую, хорошо читаемую программу, демонстрировать основные приемы алгоритмизации, он также хорошо подходит для реализации крупных проектов.
4. ***Бейсик*** – создавался в 60-х годах также для обучения программированию. Для него имеются и компиляторы и интерпретаторы, является одним из самых популярных языков программирования.
5. ***Си*** – был создан в 70- е годы первоначально не рассматривался как массовый язык программирования. Он планировался для замены ассемблера, чтобы иметь возможность создавать такие же эффективные и короткие программы, но не зависеть от конкретного процессора. Он во многом похож на Паскаль и имеет дополнительные возможности для работы с памятью. На нем написано много прикладных и системных программ, а также операционная система Unix.
6. ***Си++*** - объектно-ориентированное расширение языка Си, созданное Бьярном Страуструпом в 1980г.
7. ***Java*** – язык, который был создан компанией Sun в начале 90-х годов на основе Си++. Он призван упростить разработку приложений на СИ++ путем исключения из него низкоуровневых возможностей. Главная особенность языка – это то, что он компилируется не в машинный код, а в платформно-независимый байт-код (каждая команда занимает один байт).

Этот код может выполняться с помощью интерпретатора – виртуальной Java-машины (JVM).

8. ***Python*** — высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Синтаксис ядра Python минималистичен. В то же время стандартная библиотека включает большой объём полезных функций. Поддерживает структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное программирование. Основные архитектурные черты — динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений, высокоуровневые структуры данных. Поддерживается разбиение программ на модули, которые, в свою очередь, могут объединяться в пакеты.

3. Парадигмы программирования

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Важно отметить, что парадигма программирования не определяется однозначно языком программирования; практически все современные языки программирования в той или иной мере допускают использование различных парадигм (мультипарадигмальное программирование). Так, на языке Си, 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, и т. д.

3.1 Обзор парадигм

Существует три основных парадигмы:

- структурное
- объектно-ориентированное
- функциональное

Интересно, что сначала было открыто функциональное, потом объектно-ориентированное, и только потом структурное программирование, но применяться повсеместно на практике они стали в обратном порядке.

Структурное программирование было открыто Дейкстрой в 1968 году. Он понял, что goto – это зло, и программы должны строиться из трёх базовых структур: последовательности, ветвления и цикла.

Объектно-ориентированное программирование было открыто в 1966 году.

Функциональное программирование открыто в 1936 году, когда Чёрч придумал лямбда-исчисление. Первый функциональный язык LISP был создан в 1958 году Джоном МакКарти.

Каждая из этих парадигм убирает возможности у программиста, а не добавляет. Они говорят нам скорее, что нам не нужно делать, чем то, что нам нужно делать.

Все эти парадигмы очень связаны с архитектурой. Полиморфизм в ООП нужен, чтобы наладить связь через границы модулей. Функциональное программирование диктует нам, где хранить данные и как к ним доступаться. Структурное программирование помогает в реализации алгоритмов внутри модулей.

3.1.1 Структурное программирование

Дейкстра понял, что программирование – это сложно. Большие программы имеют слишком большую сложность, которую человеческий мозг не способен контролировать.

Чтобы решить эту проблему, Дейкстра решил сделать написание программ подобно математическим доказательствам, которые также организованы в иерархии. Он понял, что если в программах использовать только if, do, while, то тогда такие программы можно легко рекурсивно разделять на более мелкие единицы, которые в свою очередь уже легко доказуемы.

С тех пор оператора goto не стало практически ни в одном языке программирования.

Таким образом, структурное программирование позволяет делать функциональную декомпозицию.

Однако на практике мало кто реально применял аналогию с теоремами для доказательства корректности программ, потому что это слишком накладно. В реальном программировании стал популярным более «лёгкий» вариант: тесты. Тесты не могут доказать корректности программ, но могут доказать их некорректность. Однако на практике, если использовать достаточно большое количество тестов, этого может быть вполне достаточно.

3.1.2 Объектно-ориентированное программирование

ООП – это парадигма, которая характеризуется наличием инкапсуляции, наследования и полиморфизма.

Инкапсуляция позволяет открыть только ту часть функций и данных, которая нужна для внешних пользователей, а остальное спрятать внутри класса.

Однако в современных языках инкапсуляция наоборот слабее, чем была даже в C. В Java, например, вообще нельзя разделить объявление класса и его определение. Поэтому сказать, что современные объектно-ориентированные языки предоставляют инкапсуляцию можно с очень большой натяжкой.

Наследование позволяет делать производные структуры на основе базовых, тем самым давая возможность осуществлять повторное использование этих структур. Наследование было реально сделать в языках до ООП, но в объектно-ориентированных языках оно стало значительно удобнее.

Наконец, *полиморфизм* позволяет программировать на основе интерфейсов, у которых могут быть множество реализаций. Полиморфизм осуществляется в ОО-языках путём использования виртуальных методов, что является очень удобным и безопасным.

Полиморфизм – это ключевое свойство ООП для построения грамотной архитектуры. Он позволяет сделать модуль независимым от конкретной реализации (реализаций) интерфейса. Этот принцип называется инверсией зависимостей, на котором основаны все плагинные системы.

Инверсия зависимостей так называется, что она позволяет изменить направление зависимостей. Сначала мы начинаем писать в простом стиле, когда высокоуровневые функции зависят от низкоуровневых. Однако, когда программа начинает становиться слишком сложной, мы инвертируем эти зависимости в противоположную сторону: высокоуровневые функции теперь зависят не от конкретных реализаций, а от интерфейсов, а реализации теперь лежат в своих модулях.

Любая зависимость всегда может быть инвертирована. В этом и есть мощь ООП.

Таким образом, между различными компонентами становится меньше точек соприкосновения, и их легче разрабатывать. Мы даже можем не перекомпилировать базовые модули, потому что мы меняем только свой компонент.

3.1.3 Функциональное программирование

В основе функционального программирования лежит запрет на изменение переменных. Если переменная однажды проинициализирована, её значение так и остаётся неизменным.

Какой профит это имеет для архитектуры? Неизменяемые данные исключают гонки, дедлоки и прочие проблемы конкурентных программ. Однако это может потребовать больших ресурсов процессора и памяти.

Применяя функциональный подход, мы разделяем компоненты на изменяемые и неизменяемые. Причём как можно больше функциональности нужно положить именно в неизменяемые компоненты и как можно меньше в изменяемые. В изменяемых же компонентах приходится работать с изменяемыми данными, которые можно защитить с помощью транзакционной памяти.

Интересным подходом для уменьшения изменяемых данных является Event Sourcing. В нём мы храним не сами данные, а историю событий, которые привели к изменениям этих данных. Так как в лог событий можно только дописывать, это означает, что все старые события уже нельзя изменить. Чтобы получить текущее состояние данных, нужно просто воспроизвести весь лог. Для оптимизации можно использовать снапшоты, которые делаются, допустим, раз в день.

3.1.4 Дополнительные парадигмы

Также существуют

- Декларативное
- Императивное
- Логическое
- Процедурное

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. В общем и целом, декларативное программирование идёт от человека к машине, тогда как императивное — от машины к человеку. Как следствие, декларативные программы не используют понятия состояния, то есть не содержат переменных и операторов присваивания (см. также ссылочная прозрачность).

Императивное программирование — это парадигма программирования (стиль написания исходного кода компьютерной программы), для которой характерно следующее:

- в исходном коде программы записываются инструкции (команды);
- инструкции должны выполняться последовательно;
- данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями;

- данные, полученные при выполнении инструкции, могут записываться в память.

Логическое программирование — парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий 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.

Процедурное программирование — программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка[1]. Процедурное программирование является отражением архитектуры традиционных ЭВМ, которая была предложена Фон Нейманом в 1940-х годах. Теоретической моделью процедурного программирования служит абстрактная вычислительная система под названием машина Тьюринга.