МТИ

МОСКОВСКИЙ ТЕХНОЛОГИЧЕСКИЙ ИНСТИТУТ

Программирование на языке высокого уровня.

Бужинский В.А. ктн доцент
bva2516@mail.ru

Москва 2016

Вебинар № 4.

Функциональное программирование

Основная литература

Ашарина И.В. Объектно-ориентированное программирование в C++ [Электронный ресурс]: учебное пособие/ Ашарина И.В.— Электрон. текстовые данные.— М.: Горячая линия - Телеком, 2012.— 320 с.— Режим доступа: http://www.iprbookshop.ru/12008.

Казанский А.А. Объектно-ориентированное программирование на языке Microsoft Visual C# в среде разработки Microsoft Visual Studio 2008 и .NET Framework. 4.3 [Электронный ресурс]: учебное пособие и практикум/ Казанский А.А.— Электрон. текстовые данные.— М.: Московский государственный строительный университет, ЭБС АСВ, 2011.— 180 с.— Режим доступа: http://www.iprbookshop.ru/19258.

Агапов В.П. Основы программирования на языке C# [Электронный ресурс]: учебное пособие/ Агапов В.П.— Электрон. текстовые данные.— М.: Московский государственный строительный университет, ЭБС АСВ, 2012.— 128 с.— Режим доступа: http://www.iprbookshop.ru/16366.

Высокоуровневый язык программирования — язык программирования, разработанный для быстроты и удобства использования программистом. Основная черта высокоуровневых языков — это абстракция, то есть введение смысловых конструкций, кратко описывающих такие структуры данных и операции над ними, описания которых на машинном коде (или другом низкоуровневом языке программирования) очень длинны и сложны для понимания.

Высокоуровневые языки программирования были разработаны для платформенной независимости сути алгоритмов. Зависимость от платформы перекладывается на инструментальные программы — трансляторы, компилирующие текст, написанный на языке высокого уровня, в элементарные машинные команды (инструкции). Поэтому, для каждой платформы разрабатывается платформенно-уникальный транслятор для каждого высокоуровневого языка, например, переводящий текст, написанный на Delphi в элементарные команды микропроцессоров семейства x86.

Так, высокоуровневые языки стремятся не только облегчить решение сложных программных задач, но и упростить портирование программного обеспечения. Использование разнообразных трансляторов и интерпретаторов обеспечивает связь программ, написанных при помощи языков высокого уровня, с различными операционными системами программируемыми устройствами и оборудованием, и, в идеале, не требует модификации исходного кода (текста, написанного на высокоуровневом языке) для любой платформы.

Программы, написанные на языках высокого уровня, проще для понимания программистом, но менее эффективны, чем их аналоги, создаваемые при помощи низкоуровневых языков. Одним из следствий этого стало добавление поддержки того или иного языка низкого уровня (язык ассемблера) в ряд современных профессиональных высокоуровневых языков программирования.

Примеры: C++, C#, Java, JavaScript, Python, PHP, Ruby, Perl, Паскаль, Delphi, Лисп. Языкам высокого уровня свойственно умение работать с комплексными структурами данных. В большинстве из них интегрирована поддержка строковых типов, объектов, операций файлового ввода-вывода и т. п.

Первым языком программирования высокого уровня считается компьютерный язык Plankalkül, разработанный немецким инженером Конрадом Цузе ещё в период 1942—1946 годах. Однако транслятора для него не существовало до 2000 года. Первым в мире транслятором языка высокого уровня является ПП (Программирующая Программа), он же ПП-1, успешно испытанный в 1954 году. Транслятор ПП-2 (1955 год, 4-й в мире транслятор) уже был оптимизирующим и содержал собственный загрузчик и отладчик, библиотеку стандартных процедур, а транслятор ПП для ЭВМ Стрела-4 уже содержал и компоновщик (linker) из модулей. Однако, широкое применение высокоуровневых языков началось с возникновением Фортрана и созданием компилятора для этого языка (1957).

В 1940-х годах появились первые цифровые компьютеры, которые программировались переключением различного рода тумблеров, проводков и кнопок. Число таких переключений достигало порядка нескольких сотен и росло с усложнением программ. Потому следующим шагом развития программирования стало создание всевозможных ассемблерных языков с простой мнемоникой.

Но даже ассемблеры не могли стать тем инструментом, которым смогли бы пользоваться многие люди, поскольку мнемокоды всё ещё оставались слишком сложными, а всякий ассемблер был жёстко связан с архитектурой, на которой исполнялся. Шагом после ассемблера стали так называемые императивные языки высокого уровня: Бейсик, Паскаль, Си, Ада и прочие, включая объектно-ориентированные. Императивными («предписывающими») такие языки названы потому, что ориентированы на последовательное исполнение инструкций, работающих с памятью (т. е. присваиваний), и итеративные циклы. Вызовы функций и процедур, даже рекурсивные, не избавляли такие языки от явной императивности.

В парадигме функционального программирования краеугольный камень, — это функция. Вспомнив историю математики, можно оценить возраст понятия «функция»: ему уже́ около четырёхсот лет, и математики придумали очень много теоретических и практических аппаратов для оперирования функциями, начиная от обыкновенных операций дифференцирования и интегрирования, заканчивая заумными функциональными анализами, теориями нечётких множеств и функций комплексных переменных.

Математические функции выражают связь между исходными данными и итоговым продуктом некоторого процесса. Процесс вычисления также имеет вход и выход, поэтому функция — вполне подходящее и адекватное средство описания вычислений. Именно этот простой принцип положен в основу функциональной парадигмы и функционального стиля программирования. Функциональная программа представляет собой набор определений функций. Функции определяются через другие функции или рекурсивно через самих себя. При выполнении программы функции получают параметры, вычисляют и возвращают результат, при необходимости вычисляя значения других функций. На функциональном языке программист не должен описывать порядок вычислений. Нужно просто описать желаемый результат как систему функций.

Функциональное программирование, как и логическое программирование, нашло большое применение в теории искуственного интеллекта и её приложениях.

Большинство функциональных языков программирования реализуются как интерпретируемые, следуя традициям Лиспа (примечание: большая часть современных реализаций Лиспа содержат компиляторы в машинный код). Таковые удобны для быстрой отладки программ, исключая длительную фазу компиляции, укорачивая обычный цикл разработки. С другой стороны, интерпретаторы в сравнении с компиляторами обычно проигрывают по скорости выполнения. Поэтому помимо интерпретаторов существуют и компиляторы, генерирующие неплохой машинный код (например, Objective Caml) или код на Си/Си++ (например, Glasgow Haskell Compiler). Что показательно, практически каждый компилятор с функционального языка реализован на этом же са́мом языке. Это же характерно и для современных реализаций Лиспа, кроме того среда разработки Лиспа позволяет выполнять компиляцию отдельных частей программы без остановки программы (вплоть до добавления методов и изменения определений классов).

Как основные свойства функциональных языков кратко рассмотрим следующие:

краткость и простота;
строгая типизация;
модульность;
функции — это значения;
чистота (отсутствие побочных эффектов);
отложенные (ленивые) вычисления.

Краткость и простота

Программы на функциональных языках обычно короче и проще, чем те же самые программы на императивных языках. Сравним программы на Си и на абстрактном функциональном языке на примере сортировки списка быстрым методом Хоара (пример, уже ста́вший классическим при описании преимуществ функциональных языков).

Пример 1. Быстрая сортировка Хоара на Си

```
void qsort(int *ds, int *de, int *ss){
    int vl = *ds, *now = ds, *inl = ss, *ing = ss + (de - ds);
    if ( de <= ds + 1 ) return;
    for( ; now != de; ++now ){
        if ( *now <= vl ) *inl++ = *now;
        else *ing-- = *now;
    }
    *++inl = vl;
    qsort(ds, ds + (inl - ss), ss);
    qsort(ds + (inl - ss), de, inl + 1);
}
```

Пример 2. Быстрая сортировка Хоара на абстрактном функциональном языке

$$\text{quickSort}([]) = []$$

$$\text{quickSort}([x : xs]) = \text{quickSort}([y \mid y \in xs, y < x]) + [x] + \text{quickSort}([y \mid y \in xs, y \geq x])$$

Как видно на этом примере, функциональная формулировка алгоритма короче и яснее чем императивная, хоть последняя в данном случае и выигрывает в эффективности исполнения, сортируя массив "на месте", внутри себя, в то время как версия на Хаскеле создает новые списки в ходе работы.

Кроме того, все операции с памятью выполняются автоматически. При создании какого-либо объекта под него автоматически выделяется память. Когда объект выполнит своё предназначение, он вскоре будет также автоматически уничтожен сборщиком мусора, который имеется в любом функциональном языке.

Ещё одно средство, позволяющее сократить программу, — встроенный механизм сопоставления с образцом. С ним можно описывать функции как индуктивные определения.

Пример 3. Быстрая сортировка Хоара на языке Haskell

```
quickSort []     = []
quickSort (x:xs) = quickSort [y | y <- xs, y < x] ++ [x] ++
                   quickSort [y | y <- xs, y >= x]
```

Строгая типизация

Из современных языков программирования многие суть строго типизированные. Строгая типизация позволяет компилятору оптимизировать программы, использовать конкретные типы и контейнеры конкретных типов вместо шаблонных, вариантных типов, более громоздких в реализации. Кроме того, строгая типизация позволяет оградиться от части ошибок, связанных с неожидаемым «видом» входных (и выходных) данных, причем это происходит на стадии компиляции, не отнимая на такие проверки время при работе программы. Система типов также способствует «документированию» программы: любая подпрограмма является функцией в математическом смысле слова, отображая одно множество (входное) на другое (выходное), и типы определяют эти множества. Читабельность программ повышается, если используются псевдонимы типов или сложные типы, собранные на основе простых, вместо базовых элементарных *целых*, *строк* и т. п.

В примере с быстрой сортировкой Хоара видно, что есть ещё одно важное отличие между вариантом на Си и вариантом на Хаскеле: функция на Си сортирует список значений типа int (целых чисел), а функция на абстрактном функциональном языке — список значений любого типа, принадлежащего к классу упорядоченных величин. Последняя функция может сортировать и список целых чисел, и список чисел с плавающей точкой, и список строк. Можно описать какой-нибудь новый тип. Определив для этого типа операции сравнения, возможно без перекомпиляции использовать функцию quickSort и со списками значений этого нового типа. Это полезное свойство системы типов называется параметрическим или истинным полиморфизмом, и поддерживается большинством функциональных языков.

Ещё одно проявление полиморфизма — перегрузка функций, позволяющая давать разным, но подобным функциям одинаковые имена. Типичный пример перегруженной операции — обычная операция сложения. Функции сложения для целых чисел и чисел с плавающей точкой различны, но для удобства они носят одно имя. Некоторые функциональные языки помимо параметрического полиморфизма поддерживают и перегрузку операций.

В языке Си++ имеется такое понятие, как шаблон, которое позволяет программисту определять полиморфные функции, подобные quickSort. В стандартную библиотеку Си++ — STL — входит такая функция и множество других полиморфных функций. Но шаблоны Си++, как и родовые функции Ады, на самом деле порождают множество перегруженных функций, которые, кстати, нужно каждый раз компилировать, что неблагоприятно сказывается на времени компиляции и размере кода. А в функциональных языках полиморфная функция quickSort — это одна единственная функция. С другой стороны, функциональные языки в этом плане проигрывают по скорости выполнения.

Модульность

Механизм модульности позволяет разделять программы на несколько сравнительно независимых частей (модулей) с чётко определёнными связями между ними. Так облегчается процесс проектирования и последующей поддержки больши́х программных систем. Поддержка модульности не есть свойство именно функциональных языков программирования, но поддерживается большинством таких языков. Существуют очень развитые модульные императивные языки. Примеры: Modula-2 и Ada-95.

Функции суть значения

В функциональных языках, равно как и вообще в языках программирования и математике, функции могут быть переданы другим функциям в качестве аргумента или возвращены в качестве результата. Функции, принимающие функциональные аргументы, называются функциями высших порядков или функционалами. Самый, пожалуй, известный функционал — функция map. Она применяет некоторую функцию ко всем элементам списка, формируя из результатов заданной функции другой список. Например, определив функцию возведения целого числа в квадрат как:

```
square n = n * n
```

Можно воспользоваться функцией `map` для возведения в квадрат всех элементов некоторого списка:

```
squareList = map square [1, 2, 3, 4]
```

Результатом будет список `[1, 4, 9, 16]`.

Чистота
(отсутствие побочных эффектов)

В императивных языках функция в процессе своего выполнения может читать и изменять значения глобальных переменных и осуществлять операции ввода-вывода. Поэтому, если вызвать одну и ту же функцию дважды с одним и тем же аргументом, может случиться так, что в качестве результата вычислятся два различных значения. Изменение функцией состояния программы иначе, чем через возвращение значения, называется побочным эффектом.
Описывать функции без побочных эффектов позволяет практически любой язык. Однако некоторые языки поощряют или даже требуют от функции побочных эффектов. Например, во многих объектно-ориентированных языках в функцию-член класса передаётся скрытый параметр (чаще он называется this или self), который эта функция неявно изменяет.

Отложенные вычисления

В традиционных языках программирования (например, Си++) вызов функции приводит к вычислению всех аргументов. Этот метод вызова функции называется **вызов по значению**. Если какой-либо аргумент не использовался в функции, то результат вычислений пропадает, следовательно, вычисления были произведены впустую. В каком-то смысле противоположностью вызова по значению является **вызов по необходимости**. В этом случае аргумент вычисляется, только если он нужен для вычисления результата. Примером такого поведения можно взять оператор конъюнкции всё из того же Си++ (&&), который не вычисляет значение второго аргумента, если первый аргумент имеет ложное значение.

Если функциональный язык не поддерживает отложенные вычисления, то он называется строгим. На самом деле, в таких языках порядок вычисления строго определён. В качестве примера строгих языков можно привести Scheme, Standard ML и Caml. Языки, использующие отложенные вычисления, называются нестрогими. Haskell — нестрогий язык, так же как, например, Gofer и Miranda. Нестрогие языки зачастую являются чистыми.

http://www.haskell.org/ - сайт, посвящённый функциональному программированию в общем и языку Haskell в частности. Содержит различные справочные материалы, список интерпретаторов и компиляторов Haskell'а (в настоящий момент все интерпретаторы и компиляторы бесплатны). Кроме того, имеется обширный список интересных ссылок на ресурсы по теории функционального программирования и другим языкам (Standard ML, Clean).