Модуль 4. Алгоритмы и структуры данных

Тема 4.17*. Конечные автоматы

Оглавление

4.17. Конечные автоматы

4.17.1 Понятие конечного автомата

Конечный автомат - это математическая абстракция устройства, которое имеет один вход, один выход и в каждый момент времени находится в одном из конечного количества состояний. На вход этому автомату подается последовательность из символов некоторого входного алфавита (например, последовательность чисел, символов Unicode, или какого-то подмножества - только английских букв), а на выходе он генерирует новую последовательность из символов, в общем случае, другого, выходного алфавита (так, по последовательности Unicode символов конечный автомат может генерировать последовательность чисел, например, каждое число которой соответствует количеству найденных подряд гласных букв).

Конечные автоматы имеют большое практическое значение. В частности, они широко используются при разработке синтаксических и лексических анализаторов. В процессе компиляции программы на любом языке программирования вначале, как правило, работает лексический анализатор, задача которого - разбить написанный программистом текст на набор лексем (где лексема - это ключевое слово, имя идентификатора или переменной, встречающаяся в коде числовая или строковая константа и так далее). Именно с этой последовательностью лексем далее работает синтаксический анализатор, который по последовательности лексем должен сгенерировать байт-код или машинный код (вначале, как правило, он строит абстрактное синтаксическое дерево, и только потом переходит к генерации кода; заинтересованного читателя можем отослать к книгам [1] и [2], в которых дается более подробная информация о работе компиляторов).

Кроме анализа сложных структурированных текстов, таких как исходные коды программ, конечные автоматы удобно применять и в решении более простых практических задач. Например, проверка того, что введенная пользователем строка является допустимым email адресом можно реализовать как простой конечный автомат (стоит отметить, что на практике, для этой задачи часто используют регулярные выражения; само вычисление же регулярного выражения, зачастую, реализовано конечным автоматом).

Определившись с практическим применением конечных автоматов, рассмотрим более подробно, что собой представляет такой автомат. Конечный автомат можно описать как совокупность следующих частей:

- входной алфавит (конечное множество входных символов), из которого формируется входная цепочка символов, подаваемая на вход конечному автомату;
- выходной алфавит (конечное множество выходных символов), из которого автомат формирует результирующую цепочку символов; в общем случае, может отличаться от входного алфавита;
- множество состояний конечного автомата, включающее некоторое начальное состояние и подмножество заключительных состояний;
- функция переходов, определяющая по текущему состоянию и текущему символу из входной цепочки в какое состояние автомат должен перейти.
- функция выходов, определяющая по текущему состоянию и текущему символу из входной цепочки, что отправить на выход.

Конечный автомат всегда находится в каком-то состоянии и обрабатыват входящую цепочку символов в цикле, пока не перейдет в некоторое конечное состояние (или пока цепочка символов не закончится). При обработке очередного символа, конечный автомат принимает решение на основе этого текущего символа и текущего состояния автомата. В результате обработки очередного символа, автомат может перейти в другое состояние и отправить что-нибудь на выход.

Основное свойство конечного автомата состоит в том, что для ответа на вопрос “как обработать очередной символ входной цепочки” он использует исключительно знание о состоянии, в котором он находится. Логика работы конечного автомата не может и не должна зависеть от того, как он попал в текущее состояние. Это ограничение упрощает проектирование и анализ конечного автомата, так как отдельные состояния конечного автомата не зависят от истории того, как автомат в это состояние пришел.

4.17.2 Простой пример строки регулярной структуры

Предположим, что вам необходимо разработать приложение для супермаркета, которое поможет владельцу вести учет проданных товаров. Касса супермаркета (программное обеспечение которой разрабатывалось 30 лет назад, поэтому, она не поддерживает экспорт в xml, json или другой распространенный формат, к корому уже есть готовый парсер) предоставляет информацию о каждой покупке в виде следующей строки:

```
Ivan Ivanov | 359 "apples", 90 "coffee", 30 "legumes (peas, beans,
peanuts)".
```

Первым элементом является полное имя покупателя, далее идет символ-разделитель в виде вертикальной черты, после которого идет список произвольной длины, разделенный запятыми; оканчивается список точкой. Каждый элемент списка состоит из пары значений, где первое значение - сумма денег, которые были потрачены, а второе значение - заключенное в кавычки название товара, на который были потрачены деньги. В данном примере, Иван Иванов потратил 359 рублей на яблоки, 90 рублей на кофе и 30 рублей на различные бобовые.

Стоит обратить внимание, что внутри наименования товара, в общем случае, могут встречаться любые символы, кроме двойных кавычек. В том числе, допустимым названием товара является "legumes (peas, beans, peanuts)", которое содержит запятые, которые являются разделителем элементов списка. Это не позволяет обойтись простым использованием функций стандартной библиотеки Java (например, методом String#split).

Также, для полноты постановки задачи, уточним, что каждый покупаемый товар встречается в одной строке не более одного раза, список товаров содержит как минимум одну пару “стоимость - имя товара”, все числа - целые, неотрицательные, не содержат ведущих нулей и не превышают 2^{31}-1, а общая длина строки не превышает 65536 символов. Строка может содержать пробелы, которые должны быть проигнорированы. Дополнительные пробелы могут содержаться в начале и в конце строки, между любыми элементами и разделителями, но не внутри элементов. Дополнительные пробелы не могут присутствовать внутри названий товара (если только пробел не является частью названия), внутри стоимости и внутри имени и фамилии (имя и фамилия всегда отделены друг от друга ровно одним пробелом).

Задача создаваемой программы - разбить одну получаемую от кассы строку на элементы и вывести их каждый в своей строке следующим образом:

```
Ivan Ivanov
359
apples
90
coffee
30
legumes (peas, beans, peanuts)
```

Умея разбивать строку на компоненты и имея в распоряжении множество строк, соответствующее всем покупкам в магазине за неделю, можно посчитать, на какую сумму всего было продано кофе, или сколько всего денег потратил на яблоки Иван Иванов (если он приходил в магазин несколько раз и ему соответствует несколько строк). Мы оставим практические аспекты данной задачи вне рамок рассмотрения, а сосредоточимся исключительно на разборе строки указанного формата.

4.17.3 Проектирование конечного автомата

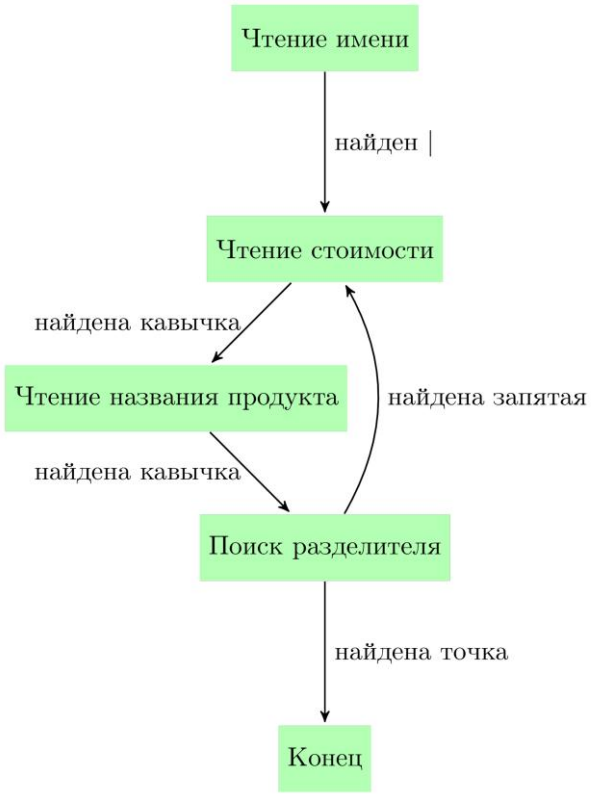
Попробуем решить сформулированную задачу с помощью конечного автомата. Этот автомат будет поглощать строку посимвольно. Какие состояния имеет смысл в нем предусмотреть?

Изначально, автомат должен находиться в состоянии чтения имени покупателя. В этом режиме, он будет выводить информацию об имени покупателя на первой строке. Автомат должен перейти из этого состояния к чтению списка товаров, как только он встретит символ "|".

Чтение списка товаров удобно представить как несколько состояний. Первое из них - это состояние чтения суммы денег. В этом состоянии автомат читает число, а как только он находит символ кавычки - это сигнал для перехода в состояние чтения названия продукта.

В состоянии чтения названия продукта, автомат читает название товара и продолжает это до тех пор, пока не встретит символ кавычки, означающий конец названия. После этого, автомат должен найти либо запятую, что означает, что имеется следующий элемент списка, либо точку, что означает конец списка. Назовем это состояние поиска состоянием поиска разделителя. Из него автомат перейдет либо снова в состояние чтения суммы денег, если он нашел запятую, либо в состояние окончания, если он нашел точку.

Граф состояний (обозначены прямоугольниками) и переходов (обозначены стрелками) создаваемого автомата будет выглядеть следующим образом:



Логика переходов в этом графе напрямую соответствует формату разбираемой строки. При чтении имени, признаком конца имени является вертикальная черта, и именно она является условием перехода автомата в следующее состояние. При чтении стоимости, нахождение кавычки является признаком того, что стоимость закончилась и дальше идет название товара, чему соответствует переход автомата из состояния “чтение стоимости” в состояние “чтение названия продукта”, и так далее.

4.17.4 Реализация конечного автомата на языке Java

Чтобы создать конечный автомат для решения указанной задачи на языке Java, необходимо, во-первых, хранить текущее состояние автомата, а во-вторых, каким-то образом задать функцию выходов, которая по текущему состоянию и входному символу сгенерирует какой-то выходной результат, и функцию переходов, которая по текущему состоянию и входному символу определит, в какое состояние должен перейти конечный автомат.

Для хранения текущего состояния конечного автомата в некоторых языках программирования могут использоваться целочисленные переменные. Например, мы могли бы пронумеровать все состояния числами и договориться, что если в переменной типа int хранится число 1, значит текущее состояние - чтение имени пользователя, а если число 2, значит текущее состояние - чтение стоимости, и так далее. Чтобы не оперировать “магическими числами” в коде мы могли бы создать именованные константы, например:

```
public static final int READING_NAME_STATE = 1;
public static final int READING_COST_STATE = 2;
```

```
public static final int READING_PRODUCT_NAME_STATE = 3;
// и так далее
```

В некоторых языках программирования это может быть практически единственным способом для хранения состояния. На самом деле, в языке Java до версии 5 (инженерный номер версии 1.5), вышедшей в 2004 году, сложно было придумать что-то лучшее для того, чтобы хранить состояния конечного автомата.

Тем не менее, можно заметить, что у такого способа представления состояния автомата есть определенные недостатки:

- не используются преимущества проверки типов; так как состояния хранятся как целые числа и над ними определены все те же действия, что и над целыми числами; например, можно случайно сложить два состояния, и хотя, с точки зрения конечного автомата это бессмысленное действие, компилятор не предупредит программиста об ошибке; можно вместо состояния случайно передать другое целое значение, которое не соответствует ни одному из состояний конечного автомата, и эта ошибка тоже не может быть определена на стадии компиляции, что плохо: ошибки на стадии компиляции предпочтительнее, чем на стадии выполнения программы; тогда как ошибки во время компиляции нельзя проигнорировать и программисту придется исправить программу, ошибка во время выполнения может не проявить себя с теми входными данными, с которыми программист проверял программу, и программа так и останется ошибочной;
- проверка уникальности лежит на программисте и не проверяется компилятором; если вы разрабатываете сложный конечный автомат с множеством состояний, программисту нужно следить, чтобы два разных состояния не получили одинаковую числовую константу; если они получат, компилятор не сможет предупредить программиста об ошибке;
- если в целях отладки программист решит вывести текущее состояние конечного автомата, или просмотреть его при пошаговой отладке, то оно будет представлять собой просто неинформативное число; чтобы получить строковый эквивалент - название соответствующего состояния, программисту придется или вручную искать среди констант, или писать и поддерживать в актуальном состоянии дополнительный метод, который по числовой константе вернет ее строковое описание.

К счастью, начиная с 5 версии в языке Java есть концепция перечислений (enums), которая не имеет перечисленных недостатков. Таким образом, для хранения состояния конечного автомата мы будем использовать перечисление (enum). В нашем случае, enum может выглядеть следующим образом:

```
enum State {READING_NAME, READING_COST, READING_PRODUCT, READING_DELIMITER,
FINISHED}
```

Что касается функции выходов и функции переходов. На практике, часто их удобно объединить в один метод, так как оба решения “в какое состояние перейти” и “что выдать на выход”, как правило, анализируют одни и те же данные. Например, когда мы находимся в состоянии “чтение имени”, любой прочитанный символ, если только это не "|", мы должны вывести на экран, так как это часть имени; аналогично, пока текущий символ не "|", мы должны оставаться в том же самом состоянии чтения имени, в противном случае - перейти к состоянию чтения стоимости.

Производить анализ текущего символа на предмет равенства "|" в двух разных методах, во-первых, неоправданно усложнит код и приведет к дублированию кода, во-вторых, повторная проверка не выгодна с точки зрения производительности.

Таким образом, в данной задаче стоит объединить генерацию выхода и возможный переход в другое состояние в рамках одного метода. Такой метод мог бы состоять из одного большого оператора switch, который, в зависимости от текущего состояния принимал бы решение о том, что делать с текущим символом, что подавать на выход и в какое состояние переходить. Например, этот метод мог бы выглядеть примерно так:

```
/**
* Возвращает новое состояние конечного автомата на основании текущего
* состояния currentState, текущего входного символа currentChar
* и пишет необходимые выходные данные в output.
*/
public State process(State currentState, char currentChar, StringBuilder output) {
   switch (currentState) {
       case READING_NAME:
           if (currentChar == '|') {
               output.append("\n"); // мы закончили с именем - осуществим перевод
на новую строку
               return READING_COST;
           }
           output.append(currentChar);
           return READING_NAME;

       case READING_COST:
           // реализация остальных состояний опущена для краткости ...
           // ...
   }
}
```

У такой реализации метод выхода и перехода есть незначительные недостатки. Например, представим себе ситуацию, что программист забыл написать один case и, соответственно, реализовать одну из веток для одного из состояний конечного автомата. Компилятор не имеет возможности предупредить программиста об ошибке на стадии компиляции, так как не знает, что в switch каждому состоянию должна соответствовать ровно одна ветка. Данную ошибку может быть довольно сложно обнаружить, особенно, в большом конечном автомате с множеством состояний. При добавлении и удалении новых состояний программист должен также не забывать добавлять и удалять ветку в этом большом операторе switch.

Для того, чтобы преодолеть эти недостатки, можно воспользоваться тем, что, во-первых, перечисления в Java являются полноценными объектами и могут иметь методы, а, во-вторых, механизмом перегрузки (override) методов в Java. В этом случае, нам не будет необходимости писать глобальный switch - этот switch будет сам происходить внутри виртуальной машины в момент вызова метода, когда JVM будет решать, какую из реализаций метода необходимо вызвать. При этом, если мы объявим метод абстрактным, программист, по сути, будет обязан для каждого состояния реализовать данный метод. Если он забудет это сделать, компилятор напомнит ему об этом ошибкой компиляции, и программисту не придется долго сидеть в отладке в поисках проблемы.

Давайте посмотрим, как может выглядеть такая реализация:

```
/**
 * Состояние конечного автомата
 */
public enum State {
  READING_NAME {
    @Override
    public State process(char currentChar, StringBuilder output) {
      if (currentChar == '|') {
        output.append("\n"); // мы закончили с именем - осуществим перевод на новую строку
        return READING_COST;
      }
      output.append(currentChar);
      return READING_NAME;
    }
  },
  READING_COST {
    @Override
    public State process(char currentChar, StringBuilder output) {
      // реализация остальных состояний и методов опущена для краткости
    }
  };

  /**
   * Возвращает новое состояние конечного автомата на основании
   * текущего входного символа currentChar и пишет необходимые
   * выходные данные в output.
   */
  public abstract State process(char currentChar, StringBuilder output);
}
```

Мы объявили абстрактный метод process внутри State. В методе больше нет параметра currentState, так как у этого метода есть несколько реализаций для каждого из состояний. Теперь, каждый State должен иметь реализацию данного метода и компилятор не даст программисту забыть ее написать и избавит его от возможных проблем с оператором switch.

Осталось написать реализацию метода выходов и переходов для всех остальных состояний и написать главный цикл, который будет переводить конечный автомат из состояния в состояние пока у нас не закончится входная строка или пока тот не придет в конечное состояние. Ниже вы можете увидеть главный цикл внутри метода main:

```
    public static void main(String[] args) {
        // тестовая входная строка
        String input = "Ivan Ivanov | 359 \"apples\", 90 \"coffee\", 30 \"legumes
(peas, beans, peanuts)\".";

        // сюда мы будем собирать результат
        StringBuilder output = new StringBuilder();

        // текущее состояние конечного автомата: изначально это чтение имени
        State currentState = State.READING_NAME;
        for (char c: input.toCharArray()) { // проходим по всем символам строки
            // вычисляем новое состояние конечного автомата на основании текущего
```

```
состояния; передаем в метод process
            // текущий обрабатываемый символ c и output, куда будет добавляется
результат
            currentState = currentState.process(c, output);
        }

        // после окончания цикла мы ожидаем, что конечный автомат будет в состоянии
FINISHED
        if (currentState != State.FINISHED)
            throw new IllegalStateException("По окончании парсинга строки мы не
оказались в финальном состоянии. Строка не соответствует формату.");

        // выведем результат
        System.out.print(output);
    }
```

Полный текст приложения с реализованным конечным автоматом имеется в следующем упражнении.

Упражнение 4.17.1

Ознакомьтесь с исходным кодом приложения, демонстрирующего конечный автомат для разбора строк регулярного формата. Решения, которые принимались в процессе реализации этого конечного автомата и причины этих решений обсуждались в предыдущем разделе (4.17.4).

Дадим еще несколько комментариев по поводу реализации данного автомата и попробуем выявить его недостатки. Во-первых, главный цикл идет до тех пор, пока не закончится строка, а не пока конечнй автомат достигнет финального состояния. Можно было бы добавить внутрь цикла дополнительное условие и прерывать цикл при наступлении конечного состояния, однако, в этом нет явной необходимости. Ожидая, пока кончится строка, мы также проверяем ее соответствие формату - после заключительной точки должны идти исключительно пробелы.

Во-вторых, при запуске данная программа выводит дополнительные пробелы перед некоторыми сущностями. Вывод программы в текущем виде - следующий:

```
Ivan Ivanov
 359
apples
 90
coffee
 30
legumes (peas, beans, peanuts)
```

Пробелы перед числами были в исходной строке и так и были выведены на экран. Чтобы избавиться от этих пробелов можно создать дополнительные состояние “пропуск пробелов перед суммой” и переходить в это состояние вместо перехода в состояние “чтение стоимости”. В рамках этого промежуточного состояния все пробелы должны пропускаться, а при наличии непробельного символа, состояние должно меняться на “чтение стоимости”.

В-третьих, стоит отметить, что данная программа содержит всю обрабатываемую строку и весь результат работы конечного автомата в памяти. Тогда как для решения текущей задачи это не является проблемой, есть ситуации, когда такая трата ресурсов недопустима. Конечный автомат не требует от нас иметь в памяти всю входную строку, так как он потребляет ее посимвольно. Если бы входная последовательность состояла из десятка гигабайт текста, хранящихся на внешнем носителе, то загрузить их одновременно в оперативную память зачастую было бы просто невозможно, однако, мы могли бы читать их с диска последовательно, не имея сразу всей строки в памяти одновременно. Аналогично, если результат работы рассматриваемого конечного автомата - последовательность символов, то нет необходимости хранить ее всю в памяти в объекте класса StringBuilder, можно было бы выводить ее в консоль (или куда-то еще) посимвольно.

В-четвертых, данная реализация вызывает метод process для каждого символа исходной строки. Хотя вызов метода занимает сравнительно малое, но ненулевое время, в нашей программе множества вызовов можно было бы избежать. Метод process довольно часто возвращает то же самое состояние конечного автомата, которое было до вызова. Например, когда происходит чтение имя покупателя и найдена очередная буква, автомат остается в состоянии “читаем имя покупателя”. Если бы метод process получал не одну букву в качестве параметра, а имел возможность прочесть больше символов, можно было бы прямо внутри метода process иметь внутренний цикл, который продолжал бы читать буквы из входного потока до тех пор, пока не возникнет необходимость изменить состояние конечного автомата. Для этого можно было бы принимать в качестве параметра метода process объект класса Reader и использовать его метод read, который возвращает очередной прочитанный символ. При этом, достигется бóльшаю универсальность создаваемого конечного автомата, так как, если нужно обработать строку (в виде переменной типа String), то ему можно передать в качестве фактического параметра объект класса StringReader, а если бы потом возникла необходимость разбирать строку из файла, можно было бы воспользоваться InputStreamReader’ом в комбинации с FilteInputStream’ом, причем, не пришлось бы вносить изменения в код самого конечного автомата. Для единообразия, в качестве объекта, куда пишется результат, можно было бы использовать Writer, чтобы также легко получать результат в виде строки, или сразу записывать его в файл или куда-то еще.

Дополнительно, имея возможность внутри обработки состояния читать несколько символов, можно избежать создания дополнительных состояний для пропуска начальных пробелов, что обсуждалось выше. Мы можем пропускать пробельные символы внутри метода process, начиная непосредственную обработку как только мы встретили непробельный символ.

В-пятых, вместо использования константы в качестве входной строки, можно читать строку из стандартного потока ввода или, в случае андроид приложения, из поля ввода в нашем приложении.

В-шестых, имело бы смысл создать класс, содержащий в себе все данные о совершенной покупке и в качестве выхода конечного автомата использовать экземпляр такого класса. Это позволило бы нам в дальнейшем производить анализ множества объектов такого класса.

Наконец, данная реализаци конечного автомата не проверяет, что там, где в строке должно быть число, соответствующее стоимости, действительно содержится число. Автомат просто добавляет символы в результат без какой-либо проверки.

Реализацию конечного автомата с учетом всех предложенных улучшений и оптимизаций можно найти в Упражнении 4.17.2.

Упражнение 4.17.2

Ознакомьтесь с исходным кодом упражнения. Этот вариант реализации конечного автомата несколько сложнее, чем предыдущий, поэтому, давайте разберем его более подробно.

Был добавлен класс ShopPurchase, который служит для хранения полученной из строки информации о покупке, произведенной в магазине. По сути, задача нового конечного автомата - это преобразовать подаваемую ему на вход строку в объект такого класса. ShopPurchase достаточно простой класс, он состоит из двух полей: String buyerName для хранения имени покупателя и SortedMap<String, Integer> products для хранения списка товаров.

Отменим некоторые особенности реализации этого класса. Он неизменяемый (immutable), что означает, что после того, как создан объект этого класса, нет возможности его изменить. В этом плане, ShopPurchase во многом похож на стандартный класс String - его тоже нельзя изменить после создания, можно только создать новый String с измененными данными.

Неизменяемость (immutability, немутабельность) класса часто упрощает как написание кода, так и его понимание в последующем. Дополнительными бонусами от того, что данный класс неизменяемый, являются:

- *возможность безопасно использовать его в качестве ключей для различных коллекций (например, для HashMap), для чего нужно перекрыть методы equals и hashCode; если бы класс был изменяемым, могла возникнуть ситуация, когда объект данного класса был добавлен в качестве ключа в коллекцию, а потом изменен, что не поддерживается стандартными коллекциями;*
- *возможность без опасений передавать его любым методам (в том числе, написанным другими программистами); так как экземпляр данного класса в принципе нельзя изменить после создания, можно не опасаться, что сторонний метод каким-то образом изменит внутреннее состояние переданного ему объекта; нет нужды создавать копий объекта для передачи в сторонний метод "на всякий случай";*
- *возможность не синхронизировать обращение к объекту данного класса из разных потоков выполнения; так как данный объект нельзя изменять, он всегда находится в консистентном состоянии и нет смысла синхронизировать доступ к нему из разных потоков; иначе говоря, он полностью потокобезопасен (thread safe).*

Тогда как эти соображения не вполне актуальны для рассматриваемой демонстрационной программы, которая только выводит строковое представление объекта класса ShopPurchase на экран, если расчитывать на дальнейшее развитие проекта, решение сделать этот класс неизменяемым выглядит уместным, поэтому, он реализован именно в таком виде.

Другими особенностями этого класса является использование TreeMap для хранения списка товаров. Каждому названию товара в этом ассоциативном массиве сопоставлено целое число - количество денег, потраченное на приобретение данного товара. TreeMap отличается тем, что он реализует интерфейс SortedMap и является упорядоченным ассоциативным массивом. Он может возвращать хранящиеся в нем элементы в порядке возврастания (или убывания) ключей. Так как ключом является название товара, мы получаем возможность выводить товары по алфавиту, а не в том порядке, в котором они были указаны в изначальной строке. Если бы для нас был важен изначальный порядок, вместо упорядоченного отображения мы могли быть использовать список (List), где каждый элемент списка - это пара значений, состоящая из стоимости и имени товара.

Также, мы используем метод Collections.unmodifiableSortedMap, чтобы получить немодифицируемый вариант упорядоченного ассоциативного массива. Это позволяет нам

гарантировать, что тот, кто вызовет метод getProducts и получит ссылку на нашу коллекцию не сможет что-нибудь в нее добавить или удалить, гарантируя неизменяемость нашей коллекции и, соответственно, нашего объекта. Вызов метода put или других методов, изменяющих коллекцию приведет к исключению.

Кроме того, в классе ShopPurchase перекрыт метод toString, что позволяет удобным образом выводить его экземпляры на экран в требуемом виде.

Продолжим исследование новой версии конечного автомата. Конечному автомату необходимо иметь некоторый “выходной” объект, в который он будет отдавать прочитанные им данные. Мы не можем использовать для этого ShopPurchase, так как конечный автомат вначале читает имя объекта, а потом читает список покупок по одной, и у нас нет единого места, где бы мы имели все данные, нужные для создания ShopPurchase сразу. Для того, чтобы решить данную проблему, используется класс ShopPurchaseBuilder. Он хранит текущие извлеченные данные и имеет методы для их добавления. Код данного класса приведен ниже:

```
public class ShopPurchaseBuilder {
    private String buyerName;
    private Map<String, Integer> products = new HashMap<>();
    private int lastCost;

    /**
     * Позволяет установить имя покупателя
     *
     * @param name имя покупателя
     */
    public void setBuyerName(String name) {
        buyerName = name;
    }

    /**
     * Позволяет начать добавление товара, указав его стоимость. После этого метода
     * должен быть вызван метод {@link #finishAddingProduct(String)}, который
укажет название
     * добавляемого товара и завершит добавление.
     *
     * @param cost стоимость добавляемого товара
     */
    public void startAddingProduct(int cost) {
        lastCost = cost;
    }

    /**
     * Позволяет закончить добавление товара. Этот метод должен быть вызван после
метода
     * {@link #startAddingProduct(int)}, который бы указал стоимость добавляемого
товара.
     *
     * @param name название добавляемого товара
     */
    public void finishAddingProduct(String name) {
        products.put(name, lastCost);
    }

    /**
     * Позволяет получить {@link ShopPurchase}
     * из данного "строителя"
```

```
     *
     * @return сформированная покупка
     */
    public ShopPurchase toShopPurchase() {
        return new ShopPurchase(buyerName, products);
    }
}
```

Он содержит три основных метода. setBuyerName позволяет установить имя покупателя, после того, как конечный автомат его прочел. Вместо него можно было бы реализовать метод addBuyerNameCharacter(char c), который добавлял бы одну букву к имени покупателя, но так как чтение имени покупателя происходит в одном и том же состоянии конечного автомата, в этом нет необходимости.

Метод startAddingProduct позволяет запомнить прочитанную конечным автоматом цену товара. На этом этапе, мы еще не может добавить товар в ассоциативный массив, так как нам нужно дождаться, пока конечный автомат прочтет название товара. Как только он это сделает, он должен вызвать метод finishAddingProduct, передав туда прочитанное название. Это позволит добавить всю необходимую информацию об очередной прочитанной покупке в ассоциативный массив.

Кроме основных трех методов, которые использует, собственно, конечный автомат, этот класс содержит метод toShopPurchase, который позволяет сконструировать ShopPurchase на основе данных, которые конечный автомат передал ShopPurchaseBuilder'у.

Осталось разобраться непосредственно с состояниями конечного автомата. Они, как и в предыдущем упражнении, представлены в виде enum'а, однако метод process был изменен. Теперь, вместо одиночного символа, он принимает объект класса Reader, что позволяет ему читать произвольное количество символов из потока и не возвращать управление в главный цикл, пока не настанет необходимость смены состояния. Вместо StringBuilder'а для аккумулирования результата он получает объект класса ShopPurchaseBuilder и использует его методы для сохранения извлеченной информации о покупке.

Рассмотрим более подробно одну из реализаций метода process (они достаточно похожи у разных состояний):

```
READING_NAME {
    @Override
    public State process(Reader input, ShopPurchaseBuilder output) throws
IOException {
        // пропустим возможные стартовые пробелы
        int currentChar = input.read();
        while (currentChar == ' ')
            currentChar = input.read();

        // здесь мы будем собирать имя покупателя
        StringBuilder name = new StringBuilder();
        while (true) {
            switch (currentChar) {
                case '|':   // мы нашли разделитель - вертикальную черту
                    output.setBuyerName(name.toString().trim()); // мы закончили
собирать имя покупателя
                    return READING_COST;
                case -1:   // у нас закончилась строка
                    throw new IllegalStateException("Ожидал разделитель '|', но
```

```
нашел конец строки. Строка неправильного формата.");
                default: // для всех остальных символов - добавляем их в имя
                    name.append((char) currentChar);
            }
            currentChar = input.read(); // читаем следующий символ
        }
    }
}
```

Во-первых, мы теперь имеем возможность пропустить стартовые пробелы (первый цикл while), так как мы не должны сразу возвращать управление главному циклу в методе main. Метод Reader#read возвращает int, потому что он может вернуть специальное значение -1, которое означает, что символьный поток закончился и у нас больше нет символов для чтения.

Далее, у нас имеется цикл while, где мы используем switch, для того, чтобы в зависимости от значения текущего прочитанного символа произвести соответствующие ему действия. Этот while не возвращает управление в основной цикл в методе main до тех пор, пока не возникнет необходимость перейти в следующее состояние. Стоит отметить, что хотя этот цикл выглядит бесконечным (while true), на самом деле он завершится одним из двух событий - либо произойдет возврат из метода оператором return (если найден разделитель и нужно перейти в новое состояние), либо, когда прочитаны все символы из входной последовательности, Reader вернет значение -1 и метод выбросит соответствующее исключение, так как строка не соответствовала ожидаемому конечным автоматом формату. Отдельные символы имени пользователя аккумулируются в локальной переменной name и, когда настает момент сменить состояние, полученное имя пользователя передается в ShopPurchaseBuilder, который сохранит его и после завершения всей обработки на его основе создаст ShopPurchase.

Реализация метода process у других состояний по структуре похожа на реализацию в состоянии чтения имени, поэтому, мы предоставим читателю самому разобраться в их реализации.

Отметим также изменения в методе main. Теперь используется BufferedReader и его метод readLine чтобы прочитать первую строку из стандартного потока ввода (stdin), вместо того, чтобы запускать конечный автомат на константной строке. После запуска приложения нужно ввести строку подходящего формата и она будет разобрана. Для того, чтоб передать строку в виде Reader'а конечному автомату, создается StringReader.

Главный цикл теперь идет до тех пор, пока не наступит конечное состояние автомата. Мы могли бы использовать PushbackReader, чтобы проверять, не закончился ли поток, читая из него один символ, и возвращая его назад методом unread, но это усложнило бы нашу программу не принеся особой выгоды. После окончания цикла, мы получаем ShopPurchase из ShopPurchaseBuilder'а и выводим ее на экран. Естественно, вместо вывода на экран, мы могли бы обработать другие строки и использовать полученные объекты ShopPurchase для того, чтобы произвести какой-то анализ произошедших в магазине покупок, например, вычислить общую сумму заработанных магазином денег или среднее количество денег, которое покупатель тратит на кофе, но это выходит за рамки текущего упражнения.

4.17.5 Итоги

Конечные автоматы являются удобным средством для обработки последовательностей, как символьных строк, так и числовых рядов, если в них есть определенная регулярная структура.

Если в стандартной библиотеке языка Java и в широкораспространенных библиотеках общего назначения (таких, как StringUtils из ApacheCommonsLang) не нашлось нужного метода для обработки строк, вполне возможно, что имеет смысл разработать конечный автомат для решения этой задачи.

Конечный автомат на каждом шаге совершает действия в зависимости только от своего состояния и текущего символа, поданного на вход. Соответственно, конечному автомату не требуется иметь в памяти всю обрабатываемую последовательность, достаточно иметь текущий символ. Действия конечного автомата не зависят от предыстории - того, как автомат попал в это состояние. На каждом шаге автомат может сгенерировать какие-то данные (функция выходов) и перейти в какое-то новое состояние (функция переходов).

При реализации конечного автомата на языке Java, состояния удобно представлять с помощью типа enum. Поведение, уникальное для каждого состояния (например, определяющее в какое новое состояние перейти и при каких условиях), удобно представлять в виде абстрактного метода в enum'е и конкретных его реализаций для каждого из состояний. Для чтения символьных последовательностей конечным автоматом часто удобно использовать конкретных наследников абстрактного класса Reader из стандартной библиотеки.

Задание 4.17.1

Напишите программу, которая получает на вход строку с набором отдельных колонок и выводит значение каждой из колонок на экран в виде отдельной строки. Приведем пример входной строки, а потом дадим более точное определение входного формата.

Строка, подаваемая на вход программе	Ожидаемый вывод программы
Ivan Ivanov;56;smart\;tall;c:\\photos\\ivan.png	Ivan Ivanov 56 smart;tall c:\photos\ivan.png

Значения отдельных колонок во входной строки разделяются символом точка с запятой (;). Если в значении колонки должна была содержаться точка с запятой, она заменяется на управляющую последовательность в виде пары символов обратный слеш и точка с запятой (\;). Если в значении колонки должен содержаться обратный слеш, он удваивается. После одиночного обратного слеша может идти только точка с запятой или второй обратный слеш, в противном случае, исходная строка имеет неверный формат.

Ваша программа должна вывести значения всех колонок из переданной ей строки, соответствующей описанному формату, на экран в столбик, так, как показано в примере.

Задание 4.17.2

Напишите программу, которая разбирает строку из файла формата csv (более подробно об этом текстовом формате можно прочитать, например, в википедии) и выводит значение каждого из столбцов на экран в виде отдельной строки. Обработку строк формата csv удобно реализовать с помощью конечного автомата. Ниже дадим более точное определение формата входной строки, а вначале пример:

Строка, подаваемая на вход программе	Ожидаемый вывод программы
"a;b";Chevy;"Venture ""Extended Edition""";"";490	a;b Chevy Venture "Extended Edition" 490

Значения отдельных колонок в строке формата csv разделяются разделительным символом - точкой с запятой (;). Значения, содержащие зарезервированные символы (двойная кавычка, точка с запятой) обрамляются двойными кавычками ("); если в значении встречаются кавычки — они представляются в строке в виде двух кавычек подряд.

Ваша программа должна вывести значения всех колонок из переданной ей строки, соответствующей описанному формату, на экран в столбик, так, как показано в примере.

Задание 4.17.3

Разработайте программу, которая на вход получает исходный текст на языке Java, а на выходе возвращает тот же самый текст, из которого удалены все комментарии. Проверьте вашу программу на исходном коде этой программы (не забудьте оставить в нём достаточное количество комменатриев нескольких видов). Обработку структурированного текста - исходного кода на языке Java - удобно реализовать с помощью конечного автомата.

Не забудьте учесть следующие особенности синтаксиса языка Java:

- язык Java поддерживает два вида комментариев, однострочный, начинающийся с пары символов // и идущий до конца строки, и многострочный, начинающийся с /* и заканчивающийся */;
- в языке Java не поддерживаются вложенные комментарии; последовательность символов /* не начинает многострочный комментарий, если находится внутри однострочного комментария, и наоборот;
- в языке Java есть символьные константы, заключенные в двойные кавычки; они могут содержать последовательности символов // или /*, при этом, такие символы не начинают комментария;
- внутри строковых констант может встречаться символ \, который экранирует следующий за ним символ; в частности, так "\"" выглядит строковая константа, содержащая один символ кавычки;
- экранированная кавычка может встречаться и в символьной константе в одинарных кавычках, например, так: '\"' можно задать символ двойной кавычки в исходном тексте программы на языке Java;
- многострочный комментарий также заменяет пробельный символ; при удалении его необходимо заменить на хотя бы один пробел, чтобы не нарушить текст программы; например, следующий участок кода является допустимым объявлением класса без полей и методов на языке Java; если удалить в нем комментарий и не вставить пробельный символ, программа перестанет компилироваться:

```
class/* the comment */MyClass { }
```

Дополнительная литература

[1] Джек Креншоу “Давайте создадим компилятор!”, 1988-1995. Нетехническое введение в создание компиляторов на языка Turbo Pascal.

[2] Seth D. Bergmann, “Compiler Design: Theory, Tools, and Examples”, 2010. Описание принципов создания компиляторов. Первое издание использует в примерах язык Pascal, второе - C++.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Козловскому Павлу Александровичу.