Лабораторная работа №4
Исследование способов реализации алгоритмов сопоставления с образцом в языке Scala

1. Цель работы

Исследовать особенности реализации алгоритмов сопоставления с образцом в языке Scala. Получить навыки использования case-классов и класса Option в функциональном программировании.

2. Основные положения

2.1. Match

В языке Scala имеется аналог оператора switch из C-подобных языков, но он использует сопоставление с образцом и, благодаря этому, обладает более широкими возможностями.

Для начала рассмотрим применение match в стиле C-подобных языков:

```
val ch: Char =  // ...
var sign: Int = // ...

ch match {
  case ‘+’ => sign = 1
  case ‘-‘ => sign = -1
  case _   => sign = 0
}
```

Вместо default используется образец «_», с которым сопоставляется всё что угодно. Если его не включить в match, то, при отсутствии должного образца, будет возбуждено исключение MatchError.

Сопоставление с образцами идёт сверху вниз (как и в switch), однако в match отсутствует оператор break.

Так же, как и if, match это не оператор, а выражение, следовательно, он имеет своё значение, которое можно присваивать. Рассмотрим концептуально правильное написание предыдущего примера:

```
val ch: Char =  // ...

val sign = ch match {
  case ‘+’ => 1
  case ‘-‘ => -1
  case _   => 0
}
```

Здесь выражение match возвращает одно из значений, в зависимости от значения ch, и происходит присваивание возвращенного значение в sign.

2.2. Ограничители

К образцам можно добавлять ограничители – логические условия, при которых происходит сопоставление. Допустим, в предыдущем примере нам, помимо символов «+» и «-» необходимо еще распознавать все цифровые знаки. В C-подобном switch нам бы пришлось перечислять все возможные случаи (case ‘1’: case ‘2’: case ‘3’ и т.д.). В Scala же

можно решить эту задачу более элегантным способом – добавив ограничитель на общий образец:

```
val ch: Char =  // ...

val value = ch match {
  case ‘+’ => 1
  case ‘-‘ => -1
  case _ if Character.isDigit(ch) => Character.digit(ch, 10)
  case _   => 0
}
```

Обратите внимание на отсутствие скобок вокруг условия ограничителя (в отличии от обычного if). Третий образец стоит интерпретировать следующим образом: «**ch это всё что угодно** *(за это отвечает символ _)* **но только если ch является цифрой** *(за это отвечает условие Character.isDigit(ch))*».

2.3. Переменные в образцах

При сопоставлении с образцом можно использовать переменные, например:

```
val somePair = (42, “I am a String”)

somePair match {
  case (i, s) =>
    println(“I found this:”)
    println(i)
    println(“and this:”)
    println(s)
}
```

Пример выше показывает:

1) можно использовать сопоставление с образцом для работы со значениями в кортежах;

2) при сопоставлении можно указать несколько требуемых действий.

Если нам нужно только первое значение из пары, то на место второго значения можно поставить символ _.

```
somePair match {
  case (i, _) => println(“First element of pair is: ” + i)
}
```

В данном случае ко второму значению пары обратиться никак нельзя.

2.4. Сопоставление с типом

Предыдущий пример разбора пары значений будет сопоставляться с парой любых значений. Допустим, надо сделать так, чтобы он сопоставлялся только с парой типа (Int, String). Для этого можно указать тип в образце для сопоставления:

```
val somePair = (42, “I am a String”)

somePair match {
  case (i: Int, s: String) => {
    println(“I found this:”)
    println(i)
```

```
    println(“and this:”)
    println(s)
  }
}
```

Теперь наш пример будет сопоставляться только с парами типа (Int, String), а на все остальные возбуждать исключение MatchError (т.к. не указан общий образец _).

Рассмотрим более практичный пример указания типа в сопоставлении с образцом. Допустим, есть следующая структура классов:

```
abstract class Parent

class FirstChild extends Parent {
  def methodOfFirstChild() = println("I'm The First Child!")
}

class SecondChild extends Parent {
  def methodOfSecondChild() = println("I'm The Second Child!")
}
```

От класса Parent наследуются два класса FirstChild и SecondChild, каждый из которых определяет свой собственный метод.

Предположим, что у нас список типа List[Parent] и необходимо обработать каждый элемент этого списка (в нашем случае вызвать метод класса), при этом необходимо знать с каким именно классом мы работаем (FirstChild или SecondChild). В Java пришлось бы делать проверки isInstance и приведение типов или оборачивать необходимое действие в интерфейс, общий для обоих классов. В Scala можно «извлечь» тип при сопоставлении с образцом.

```
val collection: List[Parent] = List(
  new SecondChild(),
  new SecondChild(),
  new FirstChild()
)

for (obj <- collection) {
      obj match {
        case o: FirstChild  => o.methodOfFirstChild()
        case o: SecondChild => o.methodOfSecondChild()
      }
    }
```

Сопоставление со списками, массивами

Ниже представлен пример сопоставления списка с различными образцами:

```
val lst: List[Int] = ...

lst match {
  case Nil          => println(“пустой список”)
  case head :: Nil  =>
    println(“Список содержит один элемент: ” + head)
  case head :: tail =>
    println(“Список содержит голову ” + head)
    println(“и хвост ” + tail)
```

```
  case head :: second :: Nil =>
    println("Список содержит ВСЕГО 2 элемента")
  case head :: second :: rest =>
    println("Список содержит КАК МИНИМУМ 2 элемента")
}
```

В данном примере Nil – явный конец списка, а tail – имя значения, в котором содержится хвост списка (который может быть пустой, а может быть и не пустой). Наиболее явно это различие демонстрируют последние два образца и соответствующие им сообщения в println.

Аналогичным образом можно проводить сопоставление с элементами массива:

```
val arr = Array(4, 5, 7, 8)

arr match {
  case Array(x, y, _*) => println(x); println(y)
}
```

Значение _* означает «сколько угодно (в том числе и 0) элементов в конце» и может применяться только в конце образца. Образец Array(x, _*, z) вызовет ошибку. К сожалению, невозможно с помощью образца получить «хвост» массива, как это было со списками.

Также стоит отметить, что данный список не совпадёт с образцом Array(x, y, z), т.к. образец подразумевает, что массив имеет только 3 элемента.

2.5. Объявление значений через сопоставление с образцом

Механизм сопоставления с образцом можно использовать вне match для раскрытия значений.

```
// данная строка создаст две раздельные переменные d: Int и s: String
// и присвоит им соответствующие значения
val (d, s) = (42, “String”)


// извлекает из массива первые два значения и присваивает их переменным
val Array(d, s) = Array(42, 13)

val Array(d, s) = Array(42, 13, 14)      // ошибка
val Array(d, s, _*) = Array(42, 13, 14)  // ОК
```

2.6. Case-классы

Case-классы это обычные классы, которые обладают рядом особенностей:

1) При объявлении case-класса автоматически создается объект-компаньон с методом apply, который позволяет создавать экземпляры объекта без помощи new;

2) case-класс и его значения можно использовать в сопоставлении с образцом;

3) Генерируются методы toString, hashCode, equals и copy, если они не были заданы явно.

Приведём сравнение методов toString обычного класса и case-класса:

```
class First(d: Int)
case class Second(d: Int)

// ...
```

```
val v1 = new First(1)
val v2 = Second(2)

println(v1.toString)  // выводит main.scala.First@7907ec20
println(v2.toString)  // выводит Second(2)
```

Сгенерированные методы equals и hashCode применяются для сравнения двух экземпляров case-класса по значениям их аргументов (а не по значению ссылки, как это обычно происходит).

Таким образом, применение case-классов упрощает жизнь.

Рассмотрим пример, демонстрирующий применение case-классов и сопоставления с образцом на примере структуры бинарного дерева:

```
abstract class Tree
case class Leaf(data: Int) extends Tree // лист дерева
case class Node(data: Int, left: Tree, right: Tree) extends Tree // узел дерева

val tree = Node(
  5,
  Node(
    3,
    Leaf(2),
    Leaf(4)
  ),
  Node(
    7,
    Leaf(6),
    Leaf(10)
  )
)

def printTree(tree: Tree, lvl: Int): Unit = {
  tree match {
    case Node(d, l, r) =>
      println("-" * lvl + d)
      printTree(l, lvl+1)
      printTree(r, lvl+1)
    case Leaf(d) => println("-" * lvl + d)
  }
}

printTree(tree, 0)
```

Небольшие пояснения к этому фрагменту кода:

```
println("-" * lvl + d)
```

В Scala можно умножать строку s на целочисленное число n. В результате получается повторение исходной строки s n раз.

Приведённый код выводит в консоль следующее:

```
5
-3
--2
--4
-7
```

--6
--10

2.7. Option

В Scala есть класс Option, который является обёрткой для любых других значений/классов. Его задача – представлять значения, которые могут быть, а могут и не быть. Класс Option имеет два дочерних класса Some и None. Some содержит внутри себя искомое значение, а None представляет несуществующее значение (почти как null).

Например, чтобы получить значение по ключу ассоциативного массива, можно использовать метод get, который возвращает Option:

```
val dictionary: Map[Char, String] = Map(
  'a' -> "alphabet",
  'b' -> "behemoth",
  's' -> "Scala"
)

dictionary.get(‘a’)  // вернёт Some(“alphabet”)
dictionary.get(‘g’)  // вернёт None
```

Также Option можно использовать в сопоставлении с образцом:

```
dictionary.get(‘s’) match {
  case Some(str) => println(str)
  case None => println(“ничего не найдено”)
}
```

3. Задание на лабораторную работу

Все задания необходимо выполнять с помощью сопоставления с образцом (match) и не использовать условные выражения (if).

3.1. Написать функцию типа (List[(Int, Int)]) => List[Option[Double]], которая принимает на вход список из пар двух целочисленных значений и возвращает список результатов деления первого числа пары на второе в виде Option (Some[Double], если второй элемент пары не равен нулю, или None, если второй элемент пары равен нулю).

3.2. Написать функцию типа (List[Option[Double]]) => List[String], которая принимает на вход список Option’ов типа Double (результаты работы функции из п.1) и преобразует его в список строк по следующему правилу: значения Some преобразуются в строку «Результат деления = __число_из_Some__», а значения None преобразуются в строку «Деление на ноль невозможно».

3.3. Реализовать задание из п.1, но вместо пар использовать case-класс.

3.4. На основе case-классов реализовать структуру дерева вычислений. Написать функцию, которая это дерево преобразует в строку обратной польской нотации.