Гуляев Г.М.
Современные технологии программирования
Лекция 4. Функциональное программирование

Центр компетенции СПО Алтайского края

Императивное(процедурное) программирование

- **Обычная программа, как правило, состоит из этапов:**
  1. **инициализация начального состояния переменных**
  2. **изменение состояния переменных в процессе выполнения**
  3. **ветвление в зависимости от состояния переменных**
  4. **при завершении - конечное состояние**
- **То есть программа постоянно занимается изменением состояния переменных. Кроме этого, она может содержать побочные эффекты: чтение/запись в файл, вывод/ввод с консоли, рисование на экране и т.п.**
- **Такая программа является последовательным набором инструкций по изменению состояния, и поэтому этот стиль программирования называется *императивным* или *процедурным*.**
- **Функциональное программирование - нечто другое. Функциональная программа - это просто выражение, а выполнение программы - процесс его вычисления.**

Функциональное программирование (ФП)

- **Функциональное программирование - это стиль программирования без опоры на состояние переменных и без побочных эффектов**
- **Пример операций с побочными эффектами:**
  1. **изменение значения переменной**
  2. **модификация содержания структуры данных**
  3. **изменение поля у объекта**
  4. **генерация исключения или останов из-за ошибки**
  5. **вывод на консоль или ввод данных с консоли**
  6. **чтение или запись в файл**
  7. **рисование на экране**
- **Какие программы сможем мы создать, используя ФП? Ответ: любые, функциональный стиль определяет как писать, а не что писать.**

Функциональное программирование (ФП)

- С точки зрения ФП программа состоит из математических функций Y = f(X) : X - входные данные, Y - результат
- Все функции программы не зависят друг от друга - их можно выполнять параллельно
- Любую программу на любом языке можно пытаться написать в функциональном стиле, но:
  1. язык может не поддерживать такой стиль (C++,Java, ...)
  2. задача проще решается другим стилем программирования
- Поэтому существуют языки программирования, которые поддерживают функциональный стиль (Lisp, APL, Erlang, ML, Haskell, Scala, ...).
- Чистых функциональных языков совсем немного (из перечисленных Haskell) - другие же могут поддерживать несколько стилей, в том числе и императивный.

Ссылочная прозрачность и чистые функции

- В ФП используются так называемые чистые (pure) функции (функции без побочных эффектов)
- Чистые функции обладают свойством ссылочной прозрачности (referentially transparent)

```
val x = "Привет, мир!"
val r1 = x.reverse // !рим ,тевирП
val r2 = x.reverse // !рим ,тевирП
val r1 = "Привет, мир!".reverse // !рим ,тевирП
val r2 = "Привет, мир!".reverse // !рим ,тевирП
```

- Пример не чистой функции:

```
import scala.collection.mutable.StringBuilder
val x = new StringBuilder("Привет")
//val y = x.append(", мир!"); val r1 = y.toString; val r2 = y.toString
val r1 = x.append(", мир!").toString // Привет, мир!
val r2 = x.append(", мир!").toString // Привет, мир!, мир!
```

ФП стимулирует правильное проектирование

- **Задача: online-игра — вывести победителя**

```
case class Player(name: String, score: Int)
def printWinner(p: Player) = println("Победил "+p.name+"!")
def winner(p1: Player, p2: Player) =
        if (p1.score > p2.score) printWinner(p1) else printWinner(p2)
val p1 = Player("Иван", 85)
val p2 = Player("Петр", 93)
winner(p1,p2) // Победил Петр!
```

- **То же самое, но winner — чистая функция:**

```
case class Player(name: String, score: Int)
def printWinner(p: Player) = println("Победил "+p.name+"!")
def winner(p1: Player, p2: Player) = if (p1.score > p2.score) p1 else p2
val p1 = Player("Иван", 85)
val p2 = Player("Петр", 93)
printWinner(winner(p1,p2)) // Победил Петр!
```

ФП стимулирует правильное проектирование

- Последнее решение отлично масштабируется:

```
case class Player(name: String, score: Int)
def printWinner(p: Player) = println("Победил "+p.name+"!")
def winner(p1: Player, p2: Player) = if (p1.score > p2.score) p1 else p2
val players = List(Player("Иван", 85),
                   Player("Петр", 93),
                   Player("Николай", 34))
val  p = players.reduceLeft(winner) // Player(Петр,93)
printWinner(p) // Победил Петр!
```

- Здесь winner чистая функция и ее можно использовать в комбинации с другими (**reduceLeft**)
- В цепочке функций функции с побочными эффектами могут быть только на входе или на выходе:

```
printWinner(players.reduceLeft(winner))
```

Преимущества и недостатки ФП

- **Преимущества:**
  1. программа в ФП = математический объект (легче доказывать)
  2. вычисления функций не зависит от порядка (распараллеливание)
  3. ФП стимулирует правильное проектирование
  4. повышение надежности кода (неизменяемые данные)
  5. проще организовать тестирование (независимо каждую функцию)
  6. лучшая оптимизация при компиляции (независимость функций)
  7. как правило, в несколько раз меньше кода, чем в ООП
- **Недостатки:**
  1. трудно вычислить возможное использование ресурсов
  2. трудно организовать операции ввода/вывода
  3. непредсказуемое потребление памяти (хороший сборщик мусора)
  4. независимость порождает проблемы синхронизации
- **Лучший вариант на практике - сочетание в программе императивного и функционального стилей с тем чтобы использовать преимущество каждого из них.**

Лямбда-исчисление

- **Парадокс Рассела:** $R = \{x: x \notin x\}$ **- множество всех множеств, которые не содержат сами себя.**

$$R \in R \Leftrightarrow R \notin R \text{ (противоречие)}$$

- **Желая разрешить парадокс Рассела, математик Алонзо Чёрч придумал специальную нотацию (λ-нотацию) для записи функций:** $y = f(x) \Longleftrightarrow (\lambda x.f)$

$(\lambda x.x) \Longleftrightarrow y = x$
$(\lambda x.x+1) \Longleftrightarrow y = x+1$
$(\lambda x.x+1)\ 2 \Longleftrightarrow y = 2+1=3$
$(\lambda x\ y.x+y) \Longleftrightarrow z = x+y$
$(\lambda x\ y.x+y)\ 1 \Longleftrightarrow z = 1+y$
$(\lambda x\ y.x+y)\ 1\ 2 = (\lambda y.1+y)\ 2 = 1+2$
$(\lambda x\ y.x+y) = (\lambda x.(\lambda y.x+y))$ // каррирование

- **С парадоксом Рассела ничего не вышло, но λ-нотация была формализована в λ-исчисление, которое позднее и явилось идейным базисом для ФП.**

Анонимные функции

- **Главная идея - описать функцию, без ее именования (анонимные и callback - функции)**
- **В некоторых ЯП для определения анонимной функции так и используется слово lambda:**

```
lambda x, y: x+y  // язык Python,  f(x,y) = x+y
(lambda (x y) (+ x y))  // язык Sheme (клон Lisp),  f(x,y) = x+y
```

- **В Scala же анонимная функция определяется как соответствие: (x, y) => x+y, например (x: Double, y: Double) => x+y**
- **Передача функции как параметра:**

```
Array(2, 3, 7).map(x => x*x)  // Array(4, 9, 49)
val  sqr = (x: Int) => x*x  // sqr(x) = x * x
Array(2, 3, 7).map(sqr)  // Array(4, 9, 49)
Array(3.14, 1.42, 2.0).map((x: Double) => 3 * x)  // Array(9.42, 4.26, 6.0)
Array(3.14, 1.42, 2.0) map {(x: Double) => 3 * x} // Array(9.42, 4.26, 6.0)
```

- **Функции с параметром функции:**

```
def f(y : Double) = (x : Double) => y * x  // f - умножение на y
val f5 = f(5) // f5 - умножение на 5
f5(20)  // 100.0
```

Задание функций

- **Шаблоны при задании функций:**

```
import scala.math._
def vq(f: (Double) => Double) = f(0.25)  // vq - значение f от четверти
vq(ceil)  // Double = 1.0
vq(sqrt)  // Double = 0.5
vq(x => 3*x)  // Double = 0.75
vq(3 * _)  // Double = 0.75
val f = 3 * _  // ошибка определения типа
val f = 3 * (_: Double)  // нет ошибок
val f: (Double) => Double = 3 * _  // нет ошибок
val f =  (x: Double) => 3 * x  // правильный вариант
```

- **Еще примеры:**

```
(1 to 9).map("*" * _).foreach(println _)
(1 to 9).filter(_ % 2 == 0)
(1 to 9).reduceLeft(_ * _) // 9! = 362880
(1 to 9).reduceLeft((x, y) => x*y) // 9! = 362880 (правильный вариант)
```

- **При задании функций, по возможности, следует избегать использования шаблонов, например, вместо `(1 to 9).map(0.1 * _)` следует писать `(1 to 9).map(x => 0.1 * x)`**

Функции высших порядков

- **Функции, которые принимают другие функции в качестве аргументов или возвращают функции, называются функциями высших порядков.**
- **Примеры функций - аргументов:**

```
val a = Array(-3, 15, 1, -5, 2, 10, 0)
a.map(x => x + 5) // Array(2, 20, 6, 0, 7, 15, 5)
a.count(x => x < 0) // 2 - число элементов
a.filter(x => x < 0) // Array(-3, -5)
a.sortWith((x,y) => x < y) // Array(-5, -3, 0, 1, 2, 10, 15)
a.reduceLeft((x,y) => x+y) // 20 - сумма элементов
```

- **Другие примеры:**

```
val cube = (x: Double) => x*x*x // возведение в куб
val root = (x: Double) => pow(x, 1d/3)  // извлечение куб. корня
def comp[A,B,C](f: B => C, g: A => B) = (x: A) => f(g(x)) // композиция
val fun = List(sin _, cos _, cube)  // sin, cos, cube
val inv = List(asin _, acos _, root) // arcsin, arccos, root
val z = (fun zip inv) map (t => comp(t._1,t._2))  // композиция в списке
z map (f => println(f(0.5))) // 0.5 0.4999... 0.5000...
```

Каррирование

- **Под каррированием понимают получение из функции n переменных функции m<n переменных за счет фиксации(задания) остальных n-m переменных:**

```
def f(x: Int, y: Int) = x * y // функция 2-х переменных
f(6,7)  // 42
def f(x: Int) = (y: Int) => x * y // функция возвращает функцию
f(6)(7)  // 42
def f(x: Int)(y: Int) = x * y // еще вариант определения
f(6)(7)  // 42
def f6 = f(6) _  // фиксация x=6
f6 (7)  //  42
def f10 = f(10) _  // фиксация x=10
f10 (7)  //  70
```

- **Еще примеры:**

```
def cat(s1: String)(s2: String) = s1 + s2 // конкатенация
cat("123")("456") // 123456
def cat(s1: String) = (s2: String) => s1 + s2 //  функция от s2
cat("123")("456") // 123456
```

Каррирование

- **Фиксация аргументов - функций:**

```
def fs(f: Int =>Int, s: List[Int]) = s map f // f - функция, s - список
def f2(x: Int) = x * 2  // удвоение
def fx(x: Int) = x * x // возведение в квадрат
def fsf2 = fs(f2, _: List[Int])  // фиксация функции f2
def fsfx = fs(fx, _: List[Int]) // фиксация функции fx

fsf2(List(0,1,2,3))  // List(0, 2, 4, 6)
fsf2(List(2,4,6,8))  // List(4, 8, 12, 16)
fsfx(List(0,1,2,3))  // List(0, 1, 4, 9)
fsfx(List(2,4,6,8))  // List(4, 16, 36, 64)
```

- **Scala дает свободу для реализации своих идей:**

```
val √ = scala.math.sqrt _  // меняем название функции
√(2)   // 1.4142135623730951
case class p(x:Int,y:Int) {def +(v: p) = v(x+v.x,y+v.y)}
p(2,3) + p(4,5)   // p(6,8) - сложение точек (векторов)
implicit def toPoint(x: (Int,Int)) = p(x._1,x._2)   //
(2,3) + (4,5)   // p(6,8)
```

Замыкания

- **Замыканием называют функцию, использующую переменные из области видимости, где она была создана:**

```
var k = 3
val f = (i: Int) => i * k
List(1, 2, 3, 4, 5) map f // List(3, 6, 9, 12, 15)
k = 5
List(1, 2, 3, 4, 5) map f // List(5, 10, 15, 20, 25)

val f = (l: List[Int]) => l.filter(x => x < l(0)) // f - выбрать все < первого
f(List(5, 1 ,7 ,4, 9, 11, 3)) // List(1, 4, 3)

val f = (l: List[Int]) => { // иной вариант той же функции
  val first = l(0)
  val isBelow = (y: Int) => y < first
  l.filter(isBelow)
}

f(List(5, 1 ,7 ,4, 9, 11, 3)) // List(1, 4, 3)
```

Алгоритм Евклида для НОД(gcd)

- **Эффективный алгоритм для (a,b): $a,b \in \mathbb{Z}$, $b \neq 0$**

$$a = bq_0 + r_0$$
$$b = r_0q_1 + r_1$$
$$r_0 = r_1q_2 + r_2$$
$$\dots$$
$$r_{n-2} = r_{n-1}q_n + r_n$$
$$r_{n-1} = r_nq_{n+1}$$

$$(a, b) = (b, r_0) = (r_0, r_1) = \dots = (r_{n-1}, r_n) = r_n$$

- **Пример:**

$$612 = 342 \cdot 1 + 270$$
$$342 = 270 \cdot 1 + 72$$
$$270 = 72 \cdot 3 + 54$$
$$72 = 54 \cdot 1 + 18$$
$$54 = 18 \cdot 3$$

$$(612, 342) = 18$$

Рекурсия

- **Рекурсия играет важную роль в ФП, заменяя собой стандартные циклы:**

```
def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b) // НОД(a,b)
gcd(14, 21) // 7
def fact(n: Int): Int = if (n == 0) 1 else n * fact(n - 1) // n! - факториал
fact(5) // 120
```

- **Недостаток рекурсии - возможное переполнение стека. Так называемая «хвостовая рекурсия» может не использовать стек (gcd - хвостовая рекурсия).**
- **Задача.** Даны две функции:

```
def succ(n: Int) = n + 1 // следующее число
def pred(n: Int) = n - 1 // предшествующее число
```

Написать функцию сложения целых положительных чисел, используя только эти две функции.

- **Источник:** http://blog.tmorris.net/posts/scala-exercises-for-beginners/
- **Решение такого рода задач может способствовать развитию способности использовать ФП и рекурсию.**

Рекурсия

- **Решение задачи:**

```
def add(x: Int, y: Int): Int = {
  if (y>0) add(succ(x),pred(y))
  else if (y<0) add(pred(x),succ(y))
  else x
}

add(10, 15) // 25
add(10, -15) // -5
```

- **Какая рекурсия была использована в функции add? Хвостовая или нет?**
- **Еще один пример неявного преобразования:**

```
class Fact (n: Int) {
  def fact(n: Int): BigInt = if (n == 0) 1 else fact(n-1) * n
  def ! = fact(n)
}
implicit def toFact(n: Int) = new Fact(n)
42! // 1405006117752879898543142606244511569936384000000000
```

Задания для самостоятельной работы

- Напишите функцию **pairs(f: (Int) => Int, low: Int, high: Int)**, возвращающую коллекцию пар. Например, **pairs(x => x*x, -5,5)** должно вернуть коллекцию пар: **(-5,25), (-4,16), (-3,9), … , (5,25)**.
- Как получить наибольший элемент массива при помощи **reduceLeft**?
- Реализуйте функцию вычисления факториала без цикла и рекурсии при помощи **to** и **reduceLeft**.
- Упростите предыдущую функцию, используя **foldLeft** (изучить работу функции по документации scaladoc).
- Напишите функцию **max(f: (Int) => Int, seq: Seq[Int])**, возвращающую наибольшее значение функции **f** на последовательности **seq**. Например, вызов **max(x => 10*x-x*x, 1 to 10)** должен вернуть 25.
- Напишите функцию f(n) = сумме всех четных членов ряда Фибоначчи, не превышающих n. Ряд Фибоначчи: **1,1,2,3,5,8,13,21,34,55,89,...** (каждый член, начиная с третьего равен сумме двух предшествующих).
- Число 145 интересно тем, что оно равно сумме факториалов своих цифр: **145 = 1! + 4! + 5!**

Вычислить сумму всех натуральных чисел, обладающих подобным свойством.

Спасибо за внимание !

www.altailand.ru