Wstęp do programowania

Wykład 14. Programowanie funkcyjne w języku Scala

Plan wykładu

1. Scala

- instalacja,

- pierwszy program,

2. Programowanie funcyjne.

- definicje funkcji

- funkcje złożone

- konstrukcja prostych algorytmów z wykorzystaniem funkcji

3. Programowanie obiektowe.

Scala: instalacja

sudo apt-get install scala

Pierwszy program:

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, world!")
  }
}
```

Kompilacja:

```
scalac HelloWorld.scala
```

Uruchomienie:

```
scala HelloWorld
```

Scala

Scala, podobnie jak Python oferuje także możliwość używania konsoli (polecenie **`scala`**)

Scala jest blisko związana z Javą – programy kompilują się do bytecody Javy. Do programowania w języku Scala można wykorzystywać te same środowiska programistyczne co w przypadku Javy (Eclipse, IntelliJ, ...).

Istnieje wiele materiałów w internecie do nauki języka Scala. Jednym z godnych wyróżnienia jest cykl wykładów przygotowanych przez Martina Odersky'ego – twórcy języka Scala, dostępny (za darmo) na platformie Cursera.

Programowanie funkcyjne

Standardowy paradygmat programowania (tzw. programowanie imperatywne) polega na wydawaniu poleceń, które operują na określonych zmiennych, odpowiednio je przekształcając.

```
int sum1(int a, int b){
    int sum = 0
    for(int i=a; a<=b; i++)
        sum += i;
    return sum;
}
```

W programowaniu funkcyjnym wartości zmiennych nie zmieniają się w trakcie działania programu, a samo programowanie polega na konstrukcji funkcji, które zwracają odpowiedni wynik

Programowanie funkcyjne

Przykład w języku Scala:

```
def sum1(a:Int, b:Int):Int =
  if (a>b)
    0
  else
    a+sum1(a+1, b)
```

W powyższym przykładzie została zdefiniowana funkcja sum, która jako argumenty przyjmuje dwie wartości typu Int i zwraca wynik typu Int. Programowanie funkcyjne powszechnie korzysta z rekursji (wywołania rekurencyjne).

Użycie tej funkcji:

```
sum1(1, 10)
```

Optymalizacja rekursji

Wywołania rekurencyjne są znacząco obciążają zasoby komputera. Dlatego, jeśli korzystamy z rekursji warto definiować funkcje tak, aby zwracały obe bezpośrednio wartość funkcji wywołanej rekurencyjne. Wtedy komputer do działania wywoływanej funkcji może skorzystać z zasobów funkcji wywołującej

```
def sum1(a:Int, b:Int):Int = {
  def sumHelper(sum:Int, a:Int, b:Int):Int =
    if (a>b) sum
    else sumHelper(a+sum, a+1, b)
  sumHelper(0, a, b)
}
```

Tutaj skorzystaliśmy z funkcji pomocniczej **`sumHelper`** zdefiniowanej w obrębie funkcji **`sum`**.

Funkcje złożone

```
def sum1(a:Int, b:Int):Int =
  if (a>b) 0 else a+sum1(a+1, b)

def sum2(a:Int, b:Int):Int =
  if (a>b) 0 else a*a+sum2(a+1, b)

def sum3(a:Int, b:Int):Int =
  if (a>b) 0 else a*a*a+sum3(a+1, b)
```

Czy można zdefiniować jedną uniwersalną funkcę fla tych operacji, tak aby nie „powtarzać” kodu?

```
def sum(f:Int=>Int, a:Int, b:Int):Int =
  if (a>b) 0 else f(a)+sum(a+1, b)
```

Funkcje złożone

```
def sum(f:Int=>Int, a:Int, b:Int):Int =
  if (a>b) 0 else f(a)+sum(a+1, b)


def square(x:Int):Int = x*x
```

wywołanie

```
sum(square, 1, 10)
```

ewentualnie

```
sum(x=>x*x, 1, 10)
```

w ostatnim przykładzie skorzystano z tzw anonimowej funkcji. Możemy napisać

```
def sum1(a:Int, b:Int) = sum(x=>x, 1, 10)
def sum2(a:Int, b:Int) = sum(x=>x*x, 1, 10)
def sum3(a:Int, b:Int) = sum(x=>x*x*x, 1, 10)
```

Czy można to jeszcze uprościć?

Funkcje złożone

Definiujemy funkcję sum, która jako argument przyjmuje funkcję a jako wynik zwraca inną funkcję:

```
def sum(f: Int => Int): (Int,Int) => Int = {
  def sumF(a:Int, b:Int):Int =
    if (a>b) 0
    else f(a) + sumF(a+1,b)
  sumF
}
```

Teraz możemy napisać

```
def sum1 = sum(x=>x)
def sum2 = sum(x=>x*x)
def sum3 = sum(x=>x*x*x)
```

Lub np.

```
def sum2 = sum(square)
```

Funkcje złożone

Przykłady użycia tak zdefiniowanych funkcji:

```
sum2(1,10) + sum1(2,5)
```

albo:

```
sum(square)(1,10)
```

Co jest równoważne:

```
(sum(square))(1,10)
```

Aby ułatwoć operwoanie na takich funkcjach złożonych jak **sum**, wprowadzono specjalną semantykę, która jest równoważna z poprzednią definicją **sum**.

```
def sum(f:Int=>Int)(a:Int, b:Int):Int =
  if(a>b) 0 else f(a) + sum(f)(a+1, b)
```

Funkcje złożone

Przykłady:

```
def product(f:Int=>Int)(a:Int, b:Int):Int =
  if(a>b) 1 else f(a)*product(f)(a+1, b)
```

albo:

```
def factorial(n:Int) = product(x=>x)(1, n)
```

Jak napisać „uniwersalną” funkcję dla **sum** i **product**?

```
def mapReduce(map:Int=>Int, reduce:(Int,Int)=>Int, start:Int)
                                                  (a:Int, b:Int):Int = {
  if (a>b) start
  else reduce(map(a), mapReduce(map, reduce, start)(a+1, b))
}
```

Wtedy:

```
def sum(f:Int=>Int)(a:Int, b:Int) = mapReduce(f, (x,y)=>x+y, 0)(a, b)
def product(f:Int=>Int)(a:Int, b:Int) = mapReduce(f,(x,y)=>x*y, 1)(a, b)
```

Funkcje złożone

```
def findZero(f:Double=>Double)(a:Double, b:Double):Double ={
  val precision = 0.0000001
  def findZeroHelper(f:Double=>Double, a:Double, fa:Double,
                                          b:Double, fb:Double):Double = {
    val mid = (a+b)/2.0
    if ((b-a)<precision)
      mid
    else{
      val fmid = f(mid)
      if (fmid*fa<0)
        findZeroHelper(f, a, fa, mid, fmid)
      else
        findZeroHelper(f, mid, fmid, b, fb)
    }
  }
  findZeroHelper(f, a, f(a), b, f(b))
}
def sqrt(a:Double) = findZero(x=>x*x-a)(0, a)
sqrt(2)
```

Obiekty

```
class Rational(x:Int, y:Int){
  def numerator = x
  def denominator = y

  override def toString = this.numerator + "/" + this.denominator


  def add(r: Rational) :Rational =
    new Rational(this.numerator*r.denominator +
                 r.numerator*this.denominator,
                                        this.denominator*r.denominator)
  def neg:Rational = new Rational(-this.numerator, this.denominator)
  def sub(r:Rational) = this.add(r.neg)
}

var r1 = new Rational(1,2)
var r2 = new Rational(2,3)
var r3 = new Rational(3,4)

r1.add(r2).add(r3).add(r3).sub(r2)
```

Obiekty

W przypadku metod jednoargumentowych, można je wywoływać bez nawiasów:

```
r1.add(r2).add(r3).add(r3).sub(r2)
```

Jest równoważne z

```
r1 add r2 add r3 add r3 sub r2
```

Dodatkowo metody mogą się nazywać +, -, itd.

```
...
  def +(r: Rational) :Rational =
    new Rational(this.numerator*r.denominator +
                 r.numerator*this.denominator,
                                 this.denominator*r.denominator)
  def -(r:Rational) = this + r.neg
...
```

I możemy zapisać.

```
r1 + r2 + r3 + r3 - r2
```

Obiekty

Operatory bezargumentowe (np. **-r3**) można definiować następująco:

```
...
  def unary_- :Rational = new Rational(-this.numerator,this.denominator)
  def -(r:Rational) = this - r
...
```

Oprzy okazji można znormalizować ułamek – najlepiej zrobić to w konstruktorze:

```
class Rational(x:Int, y:Int){
  require(y!=0)
  private def gcd(a:Int, b:Int) :Int = if(b==0) a else gcd(b, a%b)
  val g = gcd(x, y)
  def numerator = x / g
  def denominator = y / g
...
```

Dlaczego programowanie funkcyjne?

Zaletą programów funcyjnych jest możliwość niezależnego obliczania funkcji dla różnych argumentów. Ponieważ funkcje nie zawierają żadnych zmiennych więc ich obliczenia można wykonywać współbieżnie. W ostatnich latach w procesorach nie rosło znacząco taktowanie (prędkości pojedynczego rdzenia), natomiast wzrost wydajności był osiągany poprzez zwiększanie liczby rdzeni. Wykorzystanie takiej mocy obliczeniowej wymaga tworzenia programów współbieżnych. Jeszcze wyraźniej widać to na przykładzie procesorów graficznych, gdzie obecnie mamy kilka tysięcy, działających równocześnie rdzeni. Programowanie funkcyjne bardzo dobrze potrafi to wykorzystać.

Podsumowanie

Wydajność: C/C++, Java/Scala, Python

Uniwersalność: C/C++, Java, Scala/Python

Przenośność kodu: Python/Java/Scala, C/C++

Wygoda programowania: Python, Scala/Java, C/C++

Uwaga:

powyższa klasyfikacja ma jedynie charakter orientacyjny. W konkretnych zastosowaniach kolejność wymienionych języków programowania w ramach podanych kategorii może być zupełnie inna.

Programowanie funkcyjne

Dziękuję za uwagę