Princípios de Programação

Exercícios

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Engenharia Informática

2018/2019

XII Expressões lambda em Java

1. Determine um tipo para cada uma das seguintes expressões:

   (a) `(int a, int b)-> a * a + b * b`

   (b) `() -> 42`

   (c) `() -> { return 3.14; }`

   (d) `s -> s.length()`

   (e) `(double x, double y)-> Math.pow(x, y)`

   (f) `(x, y)-> Math.pow(x, y)`

   (g) `Math::pow`

2. Escreva expressões lambda para as seguintes funções:

   (a) A função quadrado de um número em vírgula flutuante.

   (b) A função que recebe dois números e devolve o maior dos dois.

   (c) A função que recebe três números e devolve o maior dos três. Utilize a função do exercício anterior. Sugestão: defina uma `@FunctionalInterface`.

   (d) A função que recebe um inteiro x e devolve uma função. A função resultado deve receber um outro inteiro y e devolver o produto de x por y.

   (e) A função `condicional` que recebe um inteiro x, um predicado p e uma função f. A função `condicional` devolve o resultado da chamada $f(x)$ se $p(x)$ for verdade, caso contrário devolve x.

3. Uma operação comum em programação funcional é a concatenação de duas listas, que em Java se pode implementar da seguinte forma:

```
<T> List<T> concatenate(List<T> l1, List<T> l2) {
  List<T> result = new ArrayList<>();
  result.addAll(l1);
  result.addAll(l2);
  return result;
}
```

   (a) Utilizando a função `concatenate`, crie uma função lambda que, dada uma lista de inteiros, retorne a lista que resulta de colocar os números 1, 2 e 3 na cauda da lista de entrada.

   (b) Utilizando a função `concatenate`, crie uma função lambda que, dada uma lista de inteiros, retorne a lista que resulta de colocar os números 1, 2 e 3 na frente dos da lista de entrada.

4. Considere a função Haskell `map :: (a -> b)-> [a] -> [b]` que transforma os elementos de uma lista de acordo com uma dada função.

   (a) Utilizando *streams*, escreva um método com a seguinte assinatura:

```
<A, B> List<B> listMap (Function<A, B> f, List<A>
    list)
```

   (b) Fazendo uso da função acima escreva uma expressão Java que converta uma lista de strings `["1","2","3","4"]` numa lista de inteiros `[1,2,3,4]`.

5. Escreva uma expressão Java que ordene uma lista de strings pelo seu comprimento, usando:

   (a) uma expressão lambda;

   (b) uma referência para um método.

6. Considere um método `processarElementos` que processe todos os elementos de um objecto do tipo `Iterable<T>`. O método deverá receber, para além do objecto iterável, um predicado, uma função unária (com um parâmetro) e um consumidor (`Consumer<U>`). Os elementos são primeiro filtrados pelo predicado, e aos que passarem o filtro é aplicada a função. O resultado é passado ao consumidor que consome o valor, e produz nada.

   (a) Implemente o método `processarElementos`.

   (b) Recorrendo ao método `processarElementos`, escreva uma expressão que, dada uma lista de strings, imprima em letras maiúsculas todas as strings com mais de 3 letras.

7. Utilize *streams* para calcular os seguintes valores:
   (a) A soma de todos os inteiros entre 0 e 1000 (exclusivé) que são múltiplos de 3, mas não de 5.
   (b) Todos os valores, ordenados e sem repetições, resultantes de elevar ao quadrado todos os números entre -10 e 4 (exclusivé) que sejam múltiplos de 3.
   (c) A aproximação do integral de $f(x) = x^2$ entre 0 e 1 usando 1000 trapézios. Utilize o método `IntStream.range(inicioInclusive, fimExclusive)`.
   (d) A mesma aproximação, mas agora usando `DoubleStream.iterate(semente, funcao)` e `DoubleStream.limit(n)`.

8. Considere a função Haskell `zipWith :: (a -> b -> c)-> [a] -> [b] -> [c]` que recebe uma função e duas listas, e retorna uma lista cujos elementos resultam de aplicar a função aos elementos na mesma posição nas listas de entrada.
   (a) Utilizando *streams*, escreva um método com a seguinte assinatura:

   ```
   <A,B,C> Stream<C> zipWith (BiFunction<A,B,C> f,
       Stream<A> as, Stream<B> bs)
   ```

   (b) Utilizando a função `zipWith`, escreva uma expressão Java que calcule o produto escalar de dois vetores representados como *streams*.
   (c) Utilizando o método criado na alínea 8a, escreva um método `zip` que tenha o comportamento da função Haskell com o mesmo nome. Este método deverá ter a assinatura

   ```
   <A,B> Stream<Pair<A,B>> zip (Stream<A> as, Stream
       <B> bs)
   ```

   e utilizar a seguinte classe:

   ```
   final class Pair<T1,T2> {
     final T1 a; final T2 b;
     Pair(T1 a, T2 b) {
       this.a = a; this.b = b;
     }
   }
   ```

9. Em Haskell, o conceito de *currying* permite criar novas funções por aplicação parcial dos argumentos de outra função. Infelizmente, esta funcionalidade não é suportada no Java 8. Escreva um método `curry` que, dada uma função do tipo `BiFunction<T1,T2,R>`, retorne a função *curried* equivalente.

Vasco T. Vasconcelos, João Pedro Neto, Alcides Fonseca
13 de novembro de 2018