INFO0054 Question Example

Maxime Deravet

January 2023

1 What is the difference between a pure function and an impure function?

A pure function is a function that has no sides effect (like modifying a variable, printing in the terminal, drawing something on screens) It will always return the same output with the same input. Impure is a function that has side-effects, as described above, or has different outputs.

2 What is currying and how does it works in Scala ?

Currying is a technique where a function that takes multiple arguments is transformed into a series of functions that each take a single argument. In Scala, a curried function is a function that returns another function as its result, until all arguments have been passed and the final result is returned. It allows for a more expressive and readable code, and simplifies passing multiple arguments to a function.

Example:

```
def multiply(x: Int)(y: Int) = x * y
val result = multiply(2)(3)

\\Currying
val timesTwo = multiply(2) _
val result = timesTwo(3)
```

3 What's the importance of tail recursion in functional programming?

Tail recursion is an optimization technique where the last operation in a function before it returns, is a recursive call. The significance of tail recursion is that it allows the compiler or interpreter to optimize the recursive call so that it uses constant space, rather than consuming stack space with each recursive call. This can greatly improve performance and prevent stack overflow errors in large inputs. It's particularly useful in cases where the function calls itself multiple times and the call stack could grow very large.

4 What is a tail-recursive function, and how can you optimize a recursive function for tail-call elimination in Scala?

A tail-recursive function is a function where the last operation before returning is a recursive call. This allows the compiler or interpreter to optimize the function by eliminating the call stack for each recursive call, improving performance and preventing stack overflow errors.

In Scala, it is up to the developer to tell if a function is tail recursive. He can do it with the annotation @annotation.tailrec

Example:

```
@annotation.tailrec
def iter(n: Int, acc: Int): Int =
    if(n == 0) acc
    else iter(n - 1, n * acc)
```

5 What are higher-order functions, and how do you define and use a higher-order function in Scala?

A higher-order function is a function that takes one or more functions as arguments, and/or returns a function as its result. Higher-order functions

allow for more flexible and powerful code by allowing functions to be passed as arguments and returned as values, just like any other type.

In Scala, a higher-order function can be defined using function types (Function1, Function2, etc) as parameter or return types. Here's an example of a higher-order function that takes a function as an argument and applies it to a given value:

```
def applyFunction(f: Int => Int, x: Int) = f(x)
val square = (x: Int) => x * x
val result = applyFunction(square, 2)
```

In this example, the applyFunction is a higher-order function that takes a function of type Int => Int and an Int as arguments. The square variable is assigned a function that takes an Int as an argument and returns its square. The result variable is assigned the result of applying the square function to the value 2.

Higher-order functions can also return functions as their result. Here's an example of a higher-order function that returns a function that adds a given value to its input:

```
def adder(x: Int) = (y: Int) => x + y
val addTwo = adder(2)
val result = addTwo(3)
```

In this example, the adder function is a higher-order function that takes an Int as an argument and returns a function that takes an Int as an argument and returns the sum of the input and the passed Int. The addTwo variable is assigned the result of applying adder to the value 2. The result variable is assigned the result of applying the returned function to the value 3.

Higher-order functions are a powerful feature in functional programming that allow for more flexible and expressive code, by allowing functions to be passed as arguments and returned as values, just like any other type.

6 How do you define and use a (foldRight | foldLeft | fold) for (Lists | Binary Trees | ...) function in Scala?

7 How do you define and use a foldLeft function in Scala?

foldLeft is a higher-order function that is commonly used to reduce a collection of elements to a single value by repeatedly applying a binary operation (a function that takes two arguments) to an accumulator value and each element in the collection.

In Scala, the foldLeft method is available on the Traversable trait, which is inherited by many collection types, such as List, Vector, and Array. Here's an example of using the foldLeft method to sum the elements of a list:

```
val numbers = List(1, 2, 3, 4, 5)
val sum = numbers.foldLeft(0)(_ + _)
```

8 What is referential transparency, and what is its importance in functional programming?

Referential transparency is a property of a function in functional programming that states that for a given input, the function will always produce the same output and have no side effects. In other words, a function is referentially transparent if you can replace a function call with its result without changing the behavior of the program.

Referential transparency is important in functional programming because it makes it easier to reason about the behavior of a program. When a function is referentially transparent, you can understand its behavior by simply looking at its inputs and outputs, without having to consider any hidden state or side effects. This makes it easier to test, debug, and reason about the program, as well as to compose and reuse functions.

For example, the following function square is referentially transparent because it always returns the square of its input and it doesn't have any side effects: `def square(x: Int): Int = x * x`

9 Explain the concept of ADTs and how they are used in functional programming. How do ADTs relate to pattern matching and recursive data structures in Scala?

ADTs (Algebraic Data Types) are a way to define new types using a combination of existing types, such as enumerations, products, and sums. ADTs are used to model complex data structures in a way that is easy to reason about, and they provide a way to define data in terms of its structure, rather than its implementation.

An example of an ADT in Scala is the List data type, which is defined as a recursive data structure that can be either an empty list (Nil) or a non-empty list (Cons) with a head and a tail.

```
enum List[A]:
        case Nil
        case Cons(head: A, tail: List[A])
```

ADTs can be used in functional programming in various ways, one of the most common uses is pattern matching. Pattern matching is a way to decompose a data structure and extract its values based on its shape. In Scala, pattern matching is done using the match expression, which allows you to match on the shape of an ADT and extract its values.

```
 def sum(list: List[Int]): Int = list match {
  case Nil => 0
  case Cons(x, xs) => x + sum(xs)
}
```

In this example, the sum function uses pattern matching to decompose the list ADT and extract its values. The first case matches the Nil constructor and returns 0, the second case matches the Cons constructor, extracts the head x and the tail xs, and returns the sum of the head and the sum of the tail.

It's important to note that ADTs are closely related to pattern matching and recursive data structures, ADTs are a way to define new types and pattern matching is a way to decompose these types. Recursive data structures are used to model complex data, and ADTs provide a way to define these structures and pattern matching provides a way to work with these struc-

tures. Together, ADTs, pattern matching, and recursive data structures provide a powerful toolset for modeling and working with complex data in functional programming.

10 What are the differences between map, flatMap, and foldMap? For each function, give the signature (you do not need to provide the function's body), explain their behavior, and in what type of objects one would typically find these functions.

11 What is a LazyList (Flux) and when would you use LazyLists over Lists in Scala? Illustrate your answer with an example.

A LazyList, also known as a "stream" in functional programming, is a data structure that is similar to a List but with the added property of being lazy. This means that the elements in a LazyList are only evaluated when they are needed, rather than being fully evaluated when the LazyList is created. This can be useful in situations where a data set is very large or infinite, as it allows for the efficient handling of large data sets without requiring them to be fully loaded into memory.

An example of when you might use a LazyList over a List in Scala is when working with large data sets that need to be processed. For example, if you have a large CSV file that you need to process, you could use a LazyList to read and process the file one line at a time, rather than loading the entire file into memory at once.

It's important to note that LazyList does not store the whole data in memory, so it may not be suitable for certain use cases where you need to access any element randomly or need to iterate multiple times.

12 How can we create an infinite stream with a LazyList, and why is this not possible with regular Lists?

In Scala, you can create an infinite stream using a LazyList data structure, which is a lazy variant of a List. A LazyList is similar to a List, but its elements are evaluated only when they are needed.

Here is an example of creating an infinite stream of integers using a LazyList:

```
val infiniteNumbers: LazyList[Int] = LazyList.from(0)(_ + 1)
```

In this example, LazyList.from(0)(_ + 1) creates a LazyList that starts from 0 and generates the next element by applying the function _ + 1 to the previous one. The result is an infinite stream of integers that starts from 0 and goes on indefinitely.

Creating an infinite stream with a LazyList is not possible with regular Lists because the elements of a List are evaluated eagerly, which means that all the elements of the list are computed and stored in memory at the time the list is created. So if we try to create an infinite list, it will cause an infinite loop and a stack overflow error.

On the other hand, a LazyList only evaluates the elements that are needed and doesn't store them in memory until they are needed, this allows it to represent an infinite data structure.

It's important to note that using an infinite stream should be done with caution, as it can cause memory leaks and performance issues if not used properly.

13 What is a monoid, and how is it used in functional programming? Explain the role of monoids in the representation of algebraic structures and the definition of operations over those structures.

A monoid is an algebraic structure that consists of a set of values and a binary operation that satisfies the properties of associativity, an identity element,

and closure.

- An associative binary operation: The operation must be associative, meaning that the order in which it is applied to the values does not matter. For example, (a * b) * c = a * (b * c)
- An identity element: There must be an element in the set that acts as an identity for the operation. For example, the number 0 is an identity element for addition, and 1 is an identity element for multiplication.
- Closure: The operation must be closed over the set, meaning that it must return a value that is also in the set.

It is used in functional programming as a way to combine values and operations in a predictable and composable way. It can be thought as a way to define an operation and its neutral element for a given type.

14 What is a fold (or foldable data structure), and how is it used in functional programming?

A fold, is a higher-order function that is used to reduce a data structure to a single value by repeatedly applying a binary operation to an accumulator value and each element of the data structure. In functional programming, a fold is often used to reduce a collection of values to a single value, such as summing the elements of a list or concatenating the elements of a string.

A foldable data structure is a data structure that can be folded. A foldable data structure is one that can be reduced to a single value using a fold function. Common examples of foldable data structures include lists, trees, and streams.

In functional programming, the fold method is available on many data structures, such as List, Option and Either in scala, which allow to reduce the data structure to a single value by applying a binary operation to an accumulator value and each element of the data structure. The fold method takes two parameters: an initial accumulator value and a binary operation to apply to the accumulator and each element of the data structure.

For example, the following code shows how to use the fold method to sum the elements of a List:

```
val numbers = List(1, 2, 3, 4, 5)
val sum = numbers.fold(0)(_ + _)
```

In this example, the fold method is called on the numbers list with an initial accumulator value of 0. The binary operation is provided as an anonymous function that takes two Int arguments (x,y) and returns the sum of these two values x+y. The _ is a placeholder for the arguments, it's a shorthand notation.

Folding is a powerful and versatile technique that is used in many functional programming libraries and frameworks to simplify working with data structures.

It's important to note that folding is not only available for lists, but also for other data structures like trees, where it can be used to traverse the tree and apply a function to each node.

15 What is a functor, and how is it used in functional programming? Explain the role of functors in the representation of contexts (or containers such as the ADTs we have seen in class) and the application of functions over those contexts (or containers).

a functor is a type of data structure that can be "mapped over." This means that a specific function can be applied to each element within the functor, with the functor itself remaining unchanged. A common example of a functor is a List, which is a data structure that can contain multiple elements of the same type. A specific function can be applied to each element within the list, and the result is a new list with the modified elements.

Functors are used to represent contexts in which a specific operation is to be performed. For example, when working with a list of integers, the context is that the operation is to be performed on each element within the list. By using a functor, the operation can be abstracted away from the specific data type, and can be reused with other types of data structures.

In terms of the application of functions over contexts, a functor allows for the separation of the context in which an operation is performed and

the operation itself. This makes it possible to reuse the same operation in different contexts without having to rewrite the operation for each context.

16 What is a monad, and how is it used in functional programming? Explain the role of monads in the representation of contexts (or containers such as the ADTs we have seen in class) and the application of functions over those contexts (or containers).

A monad is a design pattern that allows for the manipulation of data within a certain context, in a composable and predictable way. Monads are used to structure computations that have side effects, such as input/output, exceptions, state changes, and non-determinism.

A monad is a higher-order type that implements two operations: bind (also known as flatMap or »=) and return (also known as pure or unit). bind is used to chain computations together, by taking a monadic value and a function that maps from the plain value to another monadic value, and producing a new monadic value. return is used to lift a plain value into the monadic context.

A common example of a monad is the Maybe monad, w

ich is used to represent computations that may or may not return a value. The Maybe monad allows you to chain computations together while handling the possibility of a null value at any step. This can be used to avoid null pointer exceptions, and make the code more robust.

Another example of a monad is the IO monad, which is used to represent computations that interact with the outside world, such as reading from a file or printing to the console. The IO monad allows you to chain these computations together in a predictable and composable way, while keeping the side effects separate from the pure code.

17 What is lazy evaluation, and how does one use it in Scala? Why is lazy evaluation important in functional programming? Give an example to illustrate the importance.

Lazy evaluation is a technique in which the evaluation of an expression is delayed until its value is actually needed. This means that if an expression is not used in the program, it will not be evaluated. Lazy evaluation is an important concept because it allows for the manipulation of infinite data structures, and improves performance by avoiding unnecessary computation.

In Scala, you can use the lazy keyword to create a lazy value. A lazy value is evaluated the first time it is accessed, and its value is cached for subsequent accesses. Here is an example of creating a lazy value:

```
lazy val expensiveComputation: Int = {
  println("Evaluating expensive computation...")
  // Some expensive computation that takes a long time
  2 + 2
}
```

In this example, the expensive computation is wrapped in the lazy keyword, which means that it will not be evaluated until the expensiveComputation value is accessed.

One of the most important use cases of lazy evaluation is the ability to work with infinite data structures. For example, imagine a program that generates a sequence of random numbers, if the numbers were generated eagerly, the program would run indefinitely. But with lazy evaluation, the program only generates the numbers that are needed, which allows the program to terminate.

18 What is a type parameter, why are these useful, and how is it used in Scala? Explain the role of type parameters in the definition of functor and monads and the use of type bounds to constrain type parameters.

A type parameter is a placeholder for a specific type that is used in the definition of a generic class, trait, or function. This makes the code reusable and flexible. In Scala, type parameters are defined with angle brackets $< >$, for example: List[T]. The role of type parameters in the definition of functors and monads is to allow the code to be written in a way that is independent of the specific types being used. Type bounds are used to constrain type parameters, for example, the T <: Number in the class definition class MyNumberClass[T <: Number], this means that the type parameter T must be a subtype of the Number class.

19 How do you define and use a higher-kinded type in Scala?

A higher-kinded type (HKT) is a type that takes another type as a parameter. In Scala, higher-kinded types are represented by type constructors, which are types that take other types as parameters.

20 Describe and exemplify the (monoid | foldable | functor | monad) laws.

21 Is « example » a (monoid | foldable | functor | monad)? If yes, define the object in Scala. If not, give a justification or a counter-example.