

Examen 3 - CI3641

Joao Pinto 17-10490

1. Pregunta 1

1.1. Parte a

Para esta pregunta se selecciono **Swift**. **Swift** es un lenguaje compilado de proposito general y multi-paradigma desarrollado por Apple. Realizo su debut en 2014 como el sucesor de Objective-C para el desarrollo en plataforma de Apple. Objective-C no habia sido modificado de forma importante desde 1980, Swift fue la respuesta de Apple hacia la modernizacion de su stack de desarrollo.

Uno de los aspectos fundamentales de swift es su inter-operatividad con objective-C, de hecho, swift se ejecuta utilizando el Objective-C runtime library, lo cual permite la ejecucion de swift, Objective-C, C++ y C en un mismo programa. La filosofia de swift como lenguaje es simplificar la forma en la que escribe codigo, proveer de mecanismo de seguridad para reducir posibles bugs y ser altamente expresivo. En la documentacion no oficial de swift que ha sido desarrollada por la comunidad open source, se describe a swift como un lenguaje amistoso para recién llegados, con calidad industrial, que es tan disfrutable como los scripting lenguajes mas populares. La comunidad open source le ha agrado bastante cariño al lenguaje. Actualmente el uso de swift no esta limitado a desarrollo iOS en el stack de Apple, tambien existen frameworks especializados en distintos campos, Uno de los mejores ejemplos es Vapor, un popular framework de desarrollo de Backend Web que se trabaja en swift.

1.1.1. Parte a.i

1.1.1.1. Creacion de objetos

La sintaxis para definir clases en swift es muy parecida a la que encontramos en la mayoria de los lenguajes orientados a objetos:

```

1 // class seguido del nombre de la clase
2 class VideoMode {
3     // Lista de miembros de la clase
4     var interlaced = false
5     var frameRate = 0.0
6     var name: String?
7 }

```

Todas las clases tienen un constructor por defecto que inicializa los miembros de la clase en su valor por defecto:

```

1 let defaultVideoMode = VideoMode()

```

Es importante saber que si definimos un miembro que no tenga valor por defecto o no sea definido en todos los constructores de la clase, entonces tendremos un error de compilación.

```

1 class Animal {
2     let species: String
3
4     // COMPILATION ERROR: species not initialized
5 }

```

Esto también implica que al tener miembros sin valor por defecto ya no podemos utilizar el constructor por defecto.

También podemos definir nuestros propios constructores utilizando la palabra reservada `init`, podemos tener cuantos constructores queramos mientras la firma de cada constructor sea única y cada constructor inicialice todas las propiedades sin valor por defecto de la clase:

```

1 class Celsius {
2     var temperatureInCelsius: Double
3
4     init(fromFahrenheit fahrenheit: Double) {
5         temperatureInCelsius = (fahrenheit - 32.0) / 1.8
6     }
7     init(fromKelvin kelvin: Double) {
8         temperatureInCelsius = kelvin - 273.15
9     }
10 }
11 let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
12 // boilingPointOfWater.temperatureInCelsius is 100.0
13 let freezingPointOfWater = Celsius(fromKelvin: 273.15)
14 // freezingPointOfWater.temperatureInCelsius is 0.0

```

Swift también provee interacción entre los opcionales y los constructores, permitiéndonos crear *failable constructors* utilizando `init?`:

```

1 struct Animal {
2     let species: String
3
4     // Failable constructor
5     init?(species: String) {
6         if species.isEmpty { return nil }
7         self.species = species
8     }
9 }
10
11 let someCreature = Animal(species: "Giraffe")
12 // someCreature is of type Animal?, not Animal
13
14 if let giraffe = someCreature {
15     print("An animal was initialized with a species of \(giraffe.species)"
16 )
17 }
18 // Prints "An animal was initialized with a species of Giraffe"
19
20 let anonymousCreature = Animal(species: "")
21 // anonymousCreature is of type Animal?, not Animal
22
23 if anonymousCreature == nil {
24     print("The anonymous creature couldn't be initialized")
25 }
26 // Prints "The anonymous creature couldn't be initialized"

```

1.1.1.2. Propiedades de objetos

Existen varios tipos de definir propiedades para clases en swift. La forma mas simple de propiedades son los *stored properties*, estas son aquellas propiedades definidas con `var let` de la misma forma en la que se definen variables fuera del contexto de una clase:

```

1 struct FixedLengthRange {
2     var firstValue: Int
3     let length: Int
4 }
5 var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
6 // the range represents integer values 0, 1, and 2
7 rangeOfThreeItems.firstValue = 6
8 // the range now represents integer values 6, 7, and 8

```

Cualquier propiedad puede ser marcada como *lazy*, cuando se hace esto, esta propiedad es llamada *propiedad perezosa*. Las propiedades perezosas no se evalúan hasta que son necesitadas:

```

1 class DataImporter {
2     /*
3     DataImporter is a class to import data from an external file.
4     The class is assumed to take a nontrivial amount of time to initialize
5     .

```

```

5      */
6      var filename = "data.txt"
7      // the DataImporter class would provide data importing functionality
      here
8  }
9
10 class DataManager {
11     lazy var importer = DataImporter()
12     var data = [String]()
13     // the DataManager class would provide data management functionality
      here
14 }
15
16 let manager = DataManager()
17 manager.data.append("Some data")
18 manager.data.append("Some more data")
19 // the DataImporter instance for the importer property hasn't yet been
      created
20
21 print(manager.importer.filename)
22 // the DataImporter instance for the importer property has now been
      created
23 // Prints "data.txt"

```

El siguiente tipo de propiedades son las *computed properties*. Las *computed properties* son propiedades que no necesariamente se almacenan como *stored properties*, en su lugar se provee un *getter* y un *setter* para su valor:

```

1 struct Point {
2     var x = 0.0, y = 0.0
3 }
4 struct Size {
5     var width = 0.0, height = 0.0
6 }
7 class Rect {
8     var origin = Point()
9     var size = Size()
10
11     init(origin: Point, size: Size) {
12         self.origin = origin
13         self.size = size
14     }
15
16     var center: Point {
17         get {
18             let centerX = origin.x + (size.width / 2)
19             let centerY = origin.y + (size.height / 2)
20             return Point(x: centerX, y: centerY)
21         }
22         set(newCenter) {
23             origin.x = newCenter.x - (size.width / 2)
24             origin.y = newCenter.y - (size.height / 2)
25         }
26     }
27 }

```

```

26     }
27 }
28 var squareRect = Rect(origin: Point(x: 0.0, y: 0.0),
29                        size: Size(width: 10.0, height: 10.0))
30 let initialSquareCenter = squareRect.center
31
32 squareRect.center = Point(x: 15.0, y: 15.0)
33 print("square.origin is now at \(squareRect.origin.x), \(squareRect.
    origin.y)")
34 // Prints "square.origin is now at (10.0, 10.0)"

```

El cuerpo de los getters y setters son *clousures* de swift, lo cual nos permite escribirlos de forma compacta:

```

1 class CompactRect {
2     var origin = Point()
3     var size = Size()
4     var center: Point {
5         get {
6             Point(x: origin.x + (size.width / 2),
7                   y: origin.y + (size.height / 2))
8         }
9         set {
10            origin.x = newValue.x - (size.width / 2)
11            origin.y = newValue.y - (size.height / 2)
12        }
13    }
14 }

```

Una *computed properties* variante de las *computed properties* son los *read-only properties* que como su nombre lo dice, son solo de lectura, al no poseer setter, se pueden escribir de forma compacta:

```

1 class Cuboid {
2     var width = 0.0, height = 0.0, depth = 0.0
3     var volume: Double {
4         return width * height * depth
5     }
6
7     init(width: Double, height: Double, depth: Double){
8         self.width = width
9         self.height = height
10        self.depth = depth
11    }
12 }
13
14 let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
15 print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
16 // Prints "the volume of fourByFiveByTwo is 40.0"

```

Otro concepto interesante que tienen las clases de swift son los *property observers*, estos permiten observar cambios en alguna propiedad en particular. Tenemos dos tipos *willSet* y

didSet. *willSet* se llama justo antes de asignar el nuevo valor y *didSet* se llama justo despues de asignar el nuevo valor:

```
1 class StepCounter {
2     var totalSteps: Int = 0 {
3         willSet(newTotalSteps) {
4             print("About to set totalSteps to \(newTotalSteps)")
5         }
6         didSet {
7             if totalSteps > oldValue {
8                 print("Added \(totalSteps - oldValue) steps")
9             }
10        }
11    }
12 }
13 let stepCounter = StepCounter()
14 stepCounter.totalSteps = 200
15 // About to set totalSteps to 200
16 // Added 200 steps
17 stepCounter.totalSteps = 360
18 // About to set totalSteps to 360
19 // Added 160 steps
20 stepCounter.totalSteps = 896
21 // About to set totalSteps to 896
22 // Added 536 steps
```

En swift tambien existen los denominados *property wrappers*, estos surgen como la solucion a la necesida de permite añadir una capa de separacion entre el codigo que maneja como se almacena una propiedad y el codigo que define la propiedad. Para implementar un *property wrapper*, defines un struct, clase o enum que defina una propiedad llamada *wrappedValue*. Adicionalmente se debe colocar el decorador `@propertyWrapper` encima de la definicion de la clase. Para asignarle un *property wrapper* a una propiedad de una clase, se utiliza `@NombreDelWrapper(ARGUMENTOS DEL WRAPPER)` precediendo la definicion de la propiedad.

```
1 @propertyWrapper
2 class SmallNumber {
3     private var maximum: Int
4     private var number: Int
5
6     var wrappedValue: Int {
7         get { return number }
8         set { number = min(newValue, maximum) }
9     }
10
11     init() {
12         maximum = 12
13         number = 0
14     }
15     init(wrappedValue: Int) {
16         maximum = 12
17         number = min(wrappedValue, maximum)
18     }
19     init(wrappedValue: Int, maximum: Int) {
20         self.maximum = maximum
```

```

21         number = min(wrappedValue, maximum)
22     }
23 }
24
25 class NarrowRectangle {
26     @SmallNumber(wrappedValue: 2, maximum: 5) var height: Int
27     @SmallNumber(wrappedValue: 3, maximum: 4) var width: Int
28 }
29
30 var narrowRectangle = NarrowRectangle()
31 print(narrowRectangle.height, narrowRectangle.width)
32 // Prints "2 3"
33
34 narrowRectangle.height = 100
35 narrowRectangle.width = 100
36 print(narrowRectangle.height, narrowRectangle.width)
37 // Prints "5 4"
38
39
40 class MixedRectangle {
41     @SmallNumber var height: Int = 1
42     @SmallNumber(maximum: 9) var width: Int = 2
43 }
44
45 var mixedRectangle = MixedRectangle()
46 print(mixedRectangle.height)
47 // Prints "1"
48
49 mixedRectangle.height = 20
50 print(mixedRectangle.height)
51 // Prints "12"

```

El ultimo variante de propiedades que tenemos en swift son las *type properties*. Estas propiedades pertenecen al Tipo. Todas la propiedades anteriores pertenecian a las instancias de los tipos. Estas propiedades son el equivalente a las atribbutos estaticos de lenguajes como Java y C#. Para definir las propiedades de tipo se utiliza la palabra **static**. (Se usa la palabra *Tipo* para entender que se puede utilizar en **class**, **enum** y **struct**). Para las clases adicionalmente podemos utilizar la palabra **class** en lugar de **static**, la diferencia es que cuando se usa **class** la propiedad puede ser reescrita por sus sub-clases:

```

1 struct SomeStructure {
2     static var storedTypeProperty = "Some value."
3     static var computedTypeProperty: Int {
4         return 1
5     }
6 }
7 enum SomeEnumeration {
8     static var storedTypeProperty = "Some value."
9     static var computedTypeProperty: Int {
10         return 6
11     }
12 }
13 class GivenClass {
14     static var storedTypeProperty = "Some value."

```

```

15     static var computedTypeProperty: Int {
16         return 27
17     }
18     class var overrideableComputedTypeProperty: Int {
19         return 107
20     }
21 }

```

1.1.1.3. Metodos de objetos

Tenemos dos variantes de metodos en swift, los metodos de instancia (*instance methods*) y los metodos de tipo (*type methods*). Ambas variantes cumplen que son funciones de swift, con lo cual, pueden hacer cualquier cosa que se pueda hacer con las funciones en swift.

Los metodos de instancia pertenecen a las instancias de un tipo en particular. Dentro de metodos de instancias se tiene en espacio de nombre al nombre **self**, este nombre hace referencia a la instancia actual del tipo. La palabra self es opcional para acceder a propiedades de la clase, sin embargo se puede utilizar en el caso donde haya ambigüedad de nombres:

```

1 class Counter {
2     var count = 0
3     func increment() {
4         self.count += 1
5     }
6     func increment(by count: Int) {
7         self.count += count
8     }
9     func reset() {
10        self.count = 0
11    }
12 }
13
14 let counter = Counter()
15 // the initial counter value is 0
16 counter.increment()
17 // the counter's value is now 1
18 counter.increment(by: 5)
19 // the counter's value is now 6
20 counter.reset()
21 // the counter's value is now 0

```

Los metodos de tipo funcionan de forma identica que las propiedades de tipo:

```

1 class SomeBClass {
2     class func someTypeMethod() {
3         // type method implementation goes here
4     }
5 }
6 SomeBClass.someTypeMethod()

```


1.1.2. Parte a.ii

Swift es un lenguaje con recolección de basura. El recolector de basura de Swift es una variante de *reference counting*, este se llama **ARC**. **ARC** define dos tipos de referencias: referencias **fuertes** (**strong**) y referencias **débiles** (**weak**). El *reference counting ARC* se hace sobre las referencias **fuertes** que tenga un objeto, esto quiere decir, que aquellos objetos que tengan cero referencias **fuertes** serán de-referenciados. Las referencias son por defecto **fuertes**, sin embargo, el programador tiene acceso a la palabra reservada **weak** que permite definir una referencia como **débil**. La idea de las referencias débiles, es que el programador pueda evitar que ocurran referencias cíclicas. Las referencias **débiles** siempre deben ser de tipos opcionales.

En Swift, cada objeto tiene un destructor llamado **deinit**. Por defecto el destructor de un objeto no hace nada. El destructor se puede sobrescribir a través de la clase del objeto. El destructor se ejecuta justo antes de que el objeto sea de-referenciado por el recolector de basura.

```
1 class Employee {
2     public var name: String
3
4     public init(_ name: String) {
5         self.name = name
6         print("Employee \(name) initialised")
7     }
8     deinit {
9         print("Employee \(name) de-initialised")
10    }
11 }
12
13 public class Manager: Employee {
14     var reports: [Employee] = []
15 }
16
17 public class Worker: Employee {
18     // Adding weak to this property will remove the reference cycle
19     weak var manager: Employee?
20 }
21
22 // The do statement makes a scope and the scope exits once the do
23 // statement finishes.
24 do {
25     let manager = Manager("Manager")
26
27     let employee1 = Worker("Employee 1")
28     employee1.manager = manager
29
30     let employee2 = Worker("Employee 2")
31     employee2.manager = manager
32
33     let employee3 = Worker("Employee 3")
34     employee3.manager = manager
```

```

35     manager.reports = [employee1, employee2, employee3]
36 }
37
38
39 /*
40 Employee Manager initialised
41 Employee Employee 1 initialised
42 Employee Employee 2 initialised
43 Employee Employee 3 initialised
44 Employee Manager de-initialised
45 Employee Employee 1 de-initialised
46 Employee Employee 2 de-initialised
47 Employee Employee 3 de-initialised
48 */

```

1.1.3. Parte a.iii

Swift tiene asociacion dinamica de metodos y no tiene forma de proveer asociacion estatica de metodos, si bien existe la palabra reservada **final** para clases, esta lo que hace es no permitir que alguna clase herede la la clase marcada con **final**.

1.1.4. Parte a.iii

1.1.4.1. Jerarquia de tipos

En swift tenemos el tipo **Any**. **Any** es ancestro en herencia de todos los tipos de swift (tal como lo indica su nombre). Tambien tenemos el tipo **AnyObject**, este es hijo directo de **Any**, ademas, es super-clase de todos los las clases definidas con **class** que no heredan de otra clase. De esta forma, en swift tendremos dos ramificaciones en la jerarquia de tipos, la rama de **AnyObject** (en donde estan todas las tipos que construyen objetos) y la(s) rama(s) restante(s) (en donde se encuantran los enums, structs, funciones, el tipo unitario **Void**, y en general cualquier tipo que no construya objetos.)

Algo curioso sobre la gerarquia de tipos de swift, es que la gama de tipos opcionales parten de un enum generico con *associated values* (mas sobre enums y associated values **en el video de swift elaborado por mi equipo** 🎥) llamado **Optional**. Esto significa que cuando se tiene un valor de tipo **Int?** realmente se tiene una **Optional<Int>**, el valor **nil** es uno de los casos de dicho enum y los operadores **!**, **??** y **?** son operadores sobre el enum **Optional<Int>**.

A partir de la rama de **AnyObject** la jerarquia se construye a traves de la herencia de clases. En swift la herencia es simple, toda clase puede tener a lo sumo una super-clase. Algo importante que se debe destacar acerca de swift es que no soporta clases abstractas. En su lugar se recomienda utilizar implementaciones vacias y protocolos (interfaces).

```

1 class Vehicle {
2     var currentSpeed = 0.0
3     var description: String {
4         return "traveling at \(currentSpeed) miles per hour"
5     }

```

```

6     func makeNoise() {
7         // do nothing - an arbitrary vehicle doesn't necessarily make a
        noise
8     }
9 }
10
11 class Bicycle: Vehicle {
12     var hasBasket = false
13 }
14
15 let bicycle = Bicycle()
16 bicycle.hasBasket = true

```

Las sub-clases tiene la opcion de sobre-escribir metodos y propiedades del padre utilizando la palabra `override`. Esta palabra es necesaria y debe ser explicita, el compilador va a dar un error si se intenta sobrecribir un metodo sin utilizar esta palabra reservada.

```

1 // Ejemplo de override de metodos
2 class Train: Vehicle {
3     override func makeNoise() {
4         print("Choo Choo")
5     }
6 }
7
8 let train = Train()
9 train.makeNoise()
10 // Prints "Choo Choo"
11
12 // Ejemplo de override de properties
13 class Car: Vehicle {
14     var gear = 1
15     override var description: String {
16         return super.description + " in gear \(gear)"
17     }
18 }
19
20 let car = Car()
21 car.currentSpeed = 25.0
22 car.gear = 3
23 print("Car: \(car.description)")
24 // Car: traveling at 25.0 miles per hour in gear 3
25
26
27 // Ejemplo de override de property observers
28 class AutomaticCar: Car {
29     override var currentSpeed: Double {
30         didSet {
31             gear = Int(currentSpeed / 10.0) + 1
32         }
33     }
34 }
35
36 let automatic = AutomaticCar()
37 automatic.currentSpeed = 35.0

```

```

38 print("AutomaticCar: \(automatic.description)")
39 // AutomaticCar: traveling at 35.0 miles per hour in gear 4

```

Las super-clases puede evitar que se sobrescriban propiedades y metodos utilizando el modificador **final**. Esta palabra tambien se puede usar para definir clases finales, esto es, clases que no admiten que otras clases hereden de ella.

```

1 class Judge {
2     // statment no se puede sobre-escribir
3     final var statment: String {
4         "Case closed!"
5     }
6 }
7
8 // We do not want the virus to mutate, so we define it final
9 final class COVID19 {
10     let foo = "I caused a pandemic"
11 }

```

Las subclases tienen una propiedad que referencia a la super clase. Esta referencia se guarda bajo el nombre **super**.

```

1 class JetPack: Vehicle {
2     override init() {
3         super.init()
4         // Do JetPack stuff after initializing the boring part
5     }
6 }

```

Llamar **super.init** es necesesario solo si la super clase no tiene un constructor por defecto.

Las super-clases tambien definir constructores requeridos utilizando la palabra **required**. Todas las sub-clase de la super clase deben sobrescribir dicho contructor. En ese caso, no hace falta utilizar la palabra **override** en la sub-clase ya se infiere del **required**.

```

1 class SomeClass {
2     required init() {
3         // initializer implementation goes here
4     }
5 }
6
7 class SomeSubClass: SomeClass {
8     required init() {
9         // subclass implementation of the required initializer goes here
10    }
11 }

```

La herencia de swift cumple con el principio de sustitucion de Liskov:

```

1 class HAnimal {
2     func makeNoise() { print("?") }
3 }

```

```

4
5 class HCat : HAnimal {
6     override func makeNoise() {
7         print("Meow ...")
8     }
9 }
10
11 class HDog : HAnimal {
12     override func makeNoise() {
13         print("Woof!")
14     }
15 }
16
17 var animal: HAnimal
18 animal = HCat()
19 print(animal.makeNoise())
20
21 animal = HDog()
22 print(animal.makeNoise())

```

1.1.4.2. Generics

Swift permite la definicion de tipos genericos utilizando la notacion usual de `TipoGenerico<Tipo>`, por ejemplo:

```

1 class Stack<Element> {
2     var items = [Element]()
3     func push(_ item: Element) {
4         items.append(item)
5     }
6     func pop() -> Element {
7         return items.removeLast()
8     }
9 }
10
11 var s = Stack<String>()
12 s.push("Hola")
13 s.push("Mundo")
14
15 var si = Stack<Int>()
16 si.push(42)
17 si.push(69)

```

Tambien se pueden tener funciones genericas, las funciones genericas pueden especificar protocolos (interfaces) que deben implementar los tipos con los cual se usa la funcion generica:

```

1 func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {
2     for (index, value) in array.enumerated() {
3         if value == valueToFind {
4             return index
5         }
6     }
7 }

```

```

6     }
7     return nil
8 }
9
10 let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25])
11 // doubleIndex is an optional Int with no value, because 9.3 isn't in the
    array
12 let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm", "Andrea"
    ])
13 // stringIndex is an optional Int containing a value of 2

```

1.1.4.3. Manejo de varianzas

Supongamos que tenemos `Cat` que es subtipo de `Animal`. Swift maneja la varianza siguiente las siguientes reglas:

- `[Cat]` es subtipo de `[Animal]` (**Arreglos co-variantes**)
- `Set<Cat>` es subtipo de `Set<Animal>` (**Conjuntos co-variantes**)
- `[Cat:X]` es subtipo de `[Animal:X]` y `[X:Cat]` es subtipo de `[X:Animal]` (**diccionarios co-variantes**)
- `(Animal) -> X` es subtipo de `(Cat) -> X` (**argumentos de clausura contra-variantes**)
- `(X) -> Cat` es subtipo de `(X) -> Animal` (**argumentos de clausura co-variantes**)
- (Suponiendo que `Generic` es un generic no nativo del *Standard Library*) `Generic<Animal>` y `Generic<Cat>` son incompatibles (**generics invariantes**)
- (Suponiendo que `Animal` y `Cat` son generics, `Cat` se definio de la forma `class Cat<T>: Animal<T>` y `Cat` o `Animal` son definidos fuera del *Standard Library*) `Animal<X>` y `Cat<X>` son incompatibles (**generics invariantes**)

1.2. Parte b

En el archivo *Contents.swift*  se encuentran las respuestas a cada parte de esta pregunta.

2. Pregunta 2

Para esta pregunta se selecciono **Java**.

2.1. Parte a

2.1.1. Parte a.i

Desde el JDK 1.0, Java tiene soporte nativo para **Threads**, antes de comenzar la ejecucion de un thread tiene que especificar el codigo que sera ejecutado en el él:

```

1 Runnable task = () -> {
2     String threadName = Thread.currentThread().getName();
3     System.out.println("Hello " + threadName);
4 };
5
6 task.run();
7
8 Thread thread = new Thread(task);
9 thread.start();

```

Con la salida de JDK 5 se introdujo el API de concurrencia que se utiliza en los muchos de los usos modernos del lenguaje, esta es `java.util.concurrent` que ha sido mejorada con el pasar de los años para proveer un manejo mas comodo de los threads en el lenguaje.

2.1.2. Parte a.ii

Utilizando `java.util.concurrent`, la forma principal de crear tareas concurrentes a a traves de la clase `Callable` y la clase `Executor`. Un `Callable` representa un bloque de codigo que se puede ejecutar y retornar un valor, `Callable` se usa para especificar que es lo que se quiere ejecutar en el nuevo thread. Un `Executor` es un manejador de threads, el lenguaje provee distintas variantes de `Executors` la mayoría de estas basadas en pooling de threads. Los `Executors` proveen metodos comodios para iniciar procesos en threads, esto lo hace a traves de los `Callable`, por ejemplo, `executor.submit` recibe un `Callable` y lo ejecuta en alguno de los threads que maneja. `execute.invokeAll` recibe un `List` de `Callable` los ejecuta en paralelo en los threads que maneja. `execute.invokeAny` recibe un `List` de `Callable`, los coloca a ejecutar y cuando alguno de los `Callable` termine detiene a los demas. `execute.shutdown` bloquea la ejecucion principal, espera a que todos los threads activos finalicen y posteriormente continua con la ejecucion principal.

```

1 ExecutorService executor = Executors.newWorkStealingPool();
2
3 List<Callable<String>> callables = Arrays.asList(
4     () -> "task1",
5     () -> "task2",
6     () -> "task3");
7
8 executor.invokeAll(callables)
9     .stream()
10    .map(future -> {
11        try {
12            return future.get();
13        }
14        catch (Exception e) {
15            throw new IllegalStateException(e);
16        }
17    })
18    .forEach(System.out::println);
19
20 // Otro ejemplo
21 ExecutorService executor = Executors.newWorkStealingPool();

```

```

22
23 List<Callable<String>> callables = Arrays.asList(
24     callable("task1", 2),
25     callable("task2", 1),
26     callable("task3", 3));
27
28 String result = executor.invokeAny(callables);
29 System.out.println(result);
30
31 // => task2

```

En `java.util.concurrent` se utiliza manejo de memoria compartida. Este mismo no tiene limitaciones, por lo tanto el programador deber estar atento a colocar la sincronizacion correcta.

2.1.3. Parte a.iii

Los principales metodos de sincronizacion son a traves de la palabra reservada **synchronized**. Cuando un metodo o propiedad se marca como **synchronized** este se sincronizara automaticamente el uso de dicho metodo o propiedad entre los threads (a traves de monitores), evitando las condiciones de carrera:

```

1 synchronized void incrementSync() {
2     count = count + 1;
3 }
4
5 ExecutorService executor = Executors.newFixedThreadPool(2);
6
7 IntStream.range(0, 10000)
8     .forEach(i -> executor.submit(this::incrementSync));
9
10 stop(executor);
11
12 System.out.println(count); // 10000

```

synchronized tambien esta disponible en forma de bloque:

```

1 void incrementSync() {
2     synchronized (this) {
3         count = count + 1;
4     }
5 }

```

Otra de las variantes de sincronizacion que tenemos son los **locks**. Uno de los mas comunes es el **ReentrantLock**, este es bloqueo de exclusion mutua que logra los mismo que se logra utilizando **synchronized** pero de forma mas explicita:

```

1 ExecutorService executor = Executors.newFixedThreadPool(2);
2 ReentrantLock lock = new ReentrantLock();
3
4 executor.submit(() -> {
5     lock.lock();

```



```



6     try {
7         sleep(1);
8     } finally {
9         lock.unlock();
10    }
11 });
12
13 executor.submit(() -> {
14     System.out.println("Locked: " + lock.isLocked());
15     System.out.println("Held by me: " + lock.isHeldByCurrentThread());
16     boolean locked = lock.tryLock();
17     System.out.println("Lock acquired: " + locked);
18 });
19
20 stop(executor);

```



Existen variedad de locks, entre lo que destacan: `ReadWriteLock` y `StampedLock`. También tenemos otros mecanismos como los semáforos.

2.2. Parte b


2.2.1. Parte b.i

En el archivo *ConcurrentMatrixSum.java*  se encuentra la respuesta a esta pregunta. El archivo *MatrixSum.java*  es un main que se puede ejecutar para probar el programa.

2.2.2. Parte b.ii

En el archivo *ConcurrentDirCount.java*  se encuentra la respuesta a esta pregunta. El archivo *DirCount.java*  es un main que se puede ejecutar para probar el programa.

3. Pregunta 3

En la carpeta *pregunta-3*  se encuentra un archivo *.pdf* para cada parte de esta pregunta.

4. Pregunta 4

Para esta pregunta hemos seleccionado **Python** (version 3.8). Para la implementación, tendremos una clase `Clase` que representa una clase junto con su tabla de métodos virtuales. Esta clase recibe en su constructor el nombre de la clase, la lista de métodos que define y la super clase (si este es `None` se interpreta como que la clase no tiene superclase). El miembro `self.methods` representa la tabla de métodos virtuales.

Adicionalmente tendremos la clase `VirtualMethodsSystem`. Cada instancia de dicha clase guarda un diccionario de clases definidas (indexado por el nombre de dichas clase), cada ins-

tancia tambien provee metodos para *definir clases* (**define**), *describir clases* (**describe**), *validar definiciones de clases* (**validate**) y *parsear expresiones de definicion de clase* (**parse**).

En el archivo **system.py** [🔗](#) esta la implementacion de VirtualMethodsSystem. La definicion de Clase se encuentra en el archivo **clase.py** [🔗](#). Para las especificaciones de IO se implemento el archivo **main.py** [🔗](#). En el archivo **README.md** [🔗](#) se encuentran las **instrucciones para ejecutar main.py** [🔗](#).

Para las pruebas unitarias se decidio utilizar **unittest**, el modulo de pruebas unitarias por defecto de **Python**. Para la medicion del cobertura se utilizo **coverage**, que es un bastante sencillo de integrar con **unittest**. En el archivo **system_test.py** [🔗](#) se encuentran las pruebas unitarias del VirtualMethodsSystem. En el archivo **README.md** [🔗](#) se encuentran las **instrucciones para ejecutar** las pruebas unitarias. Se hace mucho enfasis en las **instrucciones para ejecutar** debido a que el codigo se organizo utilizando modulos de **Python**. Se hizo de esta manera para poder utilizar plenamente **unittest** y **coverage**.

```
pintojoao@Joaos-MacBook-Pro CI3641-exam-3 % coverage run -m unittest pregunta-4.tests.system_test
.
-----
Ran 1 test in 0.001s
OK
pintojoao@Joaos-MacBook-Pro CI3641-exam-3 % coverage report
Name                               Stmts   Miss  Cover
-----
pregunta-4/__init__.py              0      0  100%
pregunta-4/src/__init__.py          0      0  100%
pregunta-4/src/clase.py             14      1   93%
pregunta-4/src/system.py            32      0  100%
pregunta-4/tests/__init__.py        0      0  100%
pregunta-4/tests/system_test.py     21      0  100%
-----
TOTAL                               67      1   99%
pintojoao@Joaos-MacBook-Pro CI3641-exam-3 %
```

Aca se deja un screenshot de la salida de las pruebas unitarias y de la medicion de la cobertura.

5. Pregunta 5

Las respuestas a las distintas partes de esta pregunta se encuentran en el archivo **sandbox.hs** [🔗](#).

6. Pregunta 6

Para esta pregunta hemos seleccionado **Python** (version 3.8).

Para la implementacion, tenemos la clase **Exp** y sus subclases **Rule**, **Struct**, **Atom** y **Variable**, cada uno representa respectivamente un tipo de expresion dentro del interprete. Cada uno posee miembros que determinar su comportamiento e interaccion con otras expresiones dentro del interprete, propiedades como **is_unificable**, **unificate** y **textual_sub**







(sustitucion textual) son ejemplos de miembros esenciales para el interprete.



Adicionalmente, tenemos la clase **UEQT** (*Unification equation term*) que representa un binding entre una variable y valor producido por alguna unificacion.

El interprete tambien maneja el concepto de **scope**, un scope es un conjunto de instancias de **UEQT**. Un **scope** representa el conjunto de bindings que tiene una expresion, utilizamos este concepto para representar los resultados de las consultas al interprete, asi como tambien, para expresar los bindings producidos por una unificacion.

Para hacer parse de las expresiones que introduce el usuario, tenemos la clase **ExpParser**. Esta clase es un parser de *lark* .

Finalmente tendremos la clase **InterpreterDatabase**. Cada instancia de dicha clase guarda una lista de reglas definidas por el usuario (los hechos se interpretan como reglas **hecho :- True**), cada instancia tambien provee metodos para *definir reglas/hechos* (**define**), *parsear consultas a consultar* (**parse_ask**), *validar expresiones introducidas* (**validate**) y *obtener los resultados de una consulta* (**query**). Este ultimo metodo es el corazon del interprete, fue implementado utilizando DFS y iteradores de python.

En el archivo *datamodel.py*  estan las implementaciones de **Exp**, **Rule**, **Struct**, **Atom**, **Variable** y **UEQT**. La definicion de **ExpParser** se encuentra en el archivo *libparse.py* . La implementacion de **InterpreterDatabase** se encuentra en el archivo *database.py* . Para las especificaciones de IO se implemento el archivo *main.py* . En el archivo *README.md*  se encuentran las **instrucciones para ejecutar main.py** .

Para las pruebas unitarias se decidio utilizar **unittest**, el modulo de pruebas unitarias por defecto de **Python**. Para la medicion del cobertura se utilizo **coverage**, que es un bastante sencillo de integrar con **unittest**. En el archivo *system_test.py*  se encuentran las pruebas unitarias desarrolladas. En el archivo *README.md*  se encuentran las **instrucciones para ejecutar** las pruebas unitarias. Se hace mucho enfasis en las **instrucciones para ejecutar** debido a que el codigo se origanizo utilizando modulos de **Python**. Se hizo de esta manera para poder utilizar plenamente **unittest** y **coverage**.

```
pintojoao@Joaos-MacBook-Pro CI3641-exam-3 % coverage run -m unittest pregunta-4.tests.system_test
.
-----
Ran 1 test in 0.001s

OK
pintojoao@Joaos-MacBook-Pro CI3641-exam-3 % coverage report
Name                               Stmts  Miss  Cover
-----
pregunta-4/__init__.py              0      0  100%
pregunta-4/src/__init__.py          0      0  100%
pregunta-4/src/clase.py             14      1   93%
pregunta-4/src/system.py           32      0  100%
pregunta-4/tests/__init__.py        0      0  100%
pregunta-4/tests/system_test.py     21      0  100%
TOTAL                               67      1   99%
pintojoao@Joaos-MacBook-Pro CI3641-exam-3 %
```

Aca se deja un screenshot de la salida de las pruebas unitarias y de la medicion de la cobertura.