CONCEPTOS Y PARADIGMAS DE LENGUAJA DE PROGRAMACIÓN

TRABAJO INTEGRADOR 2018

Segunda Parte

INTEGRANTES:

- **SALGADO IVAN, Legajo 11823/6**
- **ALONSO DIAZ MARIANO HUGO IGNACIO, Legajo 11046/7**
- **VARELA JUAN MANUEL, Legajo 11997/9**

GRUPO: 43

LENGUAJES: Processing – Python

AYUDANTE: Viviana

TURNO: Viernes (M)

BIBLIOGRAFIA:

- https://www.python.org/
- https://processing.org/
- https://docs.oracle.com/javase/10/docs/api/overview-summary.html
- https://www.learnpython.org/es/
- https://www.tutorialspoint.com/index.htm (compiladores)
- Teorías de la catedra.

Correcciones de la entrega #1:

A. Desarrolle en EBNF el bloque “for” donde se puedan utilizar expresiones numéricas en los delimitadores. Estas expresiones pueden tener paréntesis.

- La expresión está mal definida

- Falta definir <bloque>

G = (N, T, S, P)
N = {
<sentencia_for>, <variable>, <inicializada>, <var>, <letra>, <alfanumérica>, <digito>, <real>, <declarada_inicializada>, <tipo_dato>, <comparación>, <signo_comparacion>, <calculo>, <incremento>, <decremento>, <expresión>, <fact>, <term>
}
T = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, .. , z, A, .. , Z, for, ;, (,), =, <, >, >=, <=, ==, !=, +, -, *, /, ++, - -}
S = { <sentencia_for> }
P = {
<sentencia_for> ::= ‘for’ ‘(‘ <variable> ‘;’ <comparación> ‘;’ <calculo> ‘)’ ‘{‘ <bloque> ‘}’
<variable> ::= (<declarada_inicializada> | <inicializada>)
<inicializada> ::= <var> ‘=’ <real>
<var> ::= <letra> {<alfanumérico>}*
<letra> ::= (‘a’ | .. | ‘z’ | ‘A’ | .. | ‘Z’ |)
<alfanumérica> ::= (<letra> | <digito>)
<digito> ::= (‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’)
<real> ::= [(+ | -)] {<digito>} + [‘.’{<digito>}+]
<declarada_inicializada> ::= <tipo_dato> <inicializada>
<tipo_dato> ::= (‘int’ | ‘float’ | ‘byte’)
<comparación> ::= (<var> | <real>) <signo_comparacion> (<var> | <real>)
<signo_comparación> ::= (‘<’ | ‘>’ | ‘>=’ | ‘<=’ | ‘==’ | ‘!=’)
<calculo> ::= (<incremento> | <decremento> | <expresión>)
<incremento> ::= <var> ‘++’
<decremento> ::= <var> ‘- -‘
<expresión> ::= <fact> { (‘+’ | ‘-‘) <fact> }*
<fact> ::= <term> { (‘*’ | ‘/’) <term> }*
<term> ::= (<var> | <real> | ‘(’ <expresión> ‘)’)
<bloque> ::= (<sentencia_if> | <sentencia_switch> | <sentencia_while> | <sentencia_do_while> | … todas las sentencias, declaraciones, llamadas a métodos, etc)
}

2da Parte (Fecha Final de entrega: 29/06):

D. Suponiendo que Ud. está definiendo un nuevo lenguaje de programación, defina y enuncie las características más importantes del manejo de los parámetros incorporando las características que Ud. considere necesarias de los lenguajes asignados a su grupo.

Los parámetros o argumentos son una forma de intercambiar información con el método. Pueden servir para introducir datos para ejecutar el método (entrada) o para obtener o modificar datos tras su ejecución (salida).

Parámetros en Processing:
En este lenguaje los parámetros funcionan de la misma forma que Java, siempre se pasan por valor.
No cuenta con pasaje por referencia, y tampoco pasa objetos como parámetros, si no copias de las referencias a esos objetos, esto quiere decir que cuando se manda la copia de una referencia de un objeto como parámetro, la original y la recibida por parámetro son iguales, es decir, apuntar al mismo objeto; entonces de esta forma cualquier cambio aplicado al objeto se va a ver reflejado una vez terminado el método llamado.
Todos los objetos wrappers (Byte, Short, Integer, Float, Double, Character, Boolean, String) son inmutables, esto quiere decir que no se puede enviar una copia de la referencia del objeto.
Los métodos en Java pueden devolver un valor con la palabra clave “return”, pueden devolver cualquier tipo primitivo, wrappers, objetos o void (vacio).
Ejemplos:

```
void funcion1(int a) {
	a = 25;
	System.out.println(a); // Imprime 25
}

void setup() {
	int x = 10;
	función1(x);
	System.out.println(x); // Imprime 10.
} // Su valor no cambio a pesar de que se le estaba sobrescribiendo.

void función2(Character car) {
	car = 'P';
	System.out.println(car.toString()); // Imprime ‘P’
}

void setup() {
	Character c = new Character('A');
	funcion2(c);
	System.out.println(c.toString()); // Imprime ‘A’
} // Los objetos de tipo wrappers son inmutables.

String función3(String[] s) {
	String temp = s[0] + " " + s[1];
	s[0] = "ABC";
	s[1] = "123";
	System.out.println(s[0]); // Imprime “ABC”
	System.out.println(s[1]); // Imprime “123”
	return temp;
}

void setup() {
	String[] s = new String[2];
	s[0] = "Hola";
	s[1] = "Mundo";
	System.out.println(función3(s)); // Imprime “Hola Mundo”
	System.out.println(s[0]); // Imprime “ABC”
	System.out.println(s[1]); // Imprime “123”
} // Muestra en pantalla lo mismo en cada posición porque los objetos
```

Parámetros en Python:
En este lenguaje cuando se envían los objetos a las funciones pueden ser inmutables y mutables.
Si es inmutable el pasaje de parámetro actuara por valor, es decir, si dentro de la función se modifica uno de estos parámetros para que apunte a otro valor estos cambios no se verán reflejados fuera de la función.
Si es mutable, como por ejemplo la listas, no se envía una copia, si no que se trabaja directamente sobre el objeto, por lo tanto, este cambio si se vera reflejado fuera de la función.

Ejemplos:

Inmutable:

```
def cadena(miStr):
    miStr = 'lala'
    print miStr                 # Imprime ‘lala’

s = 'xyz'
cadena(s)
print s                         # Imprime ‘xyz’
```

Cuando una variable es de un tipo inmutable, como por ejemplo una cadena, es posible asignar un nuevo valor a esa variable, pero no es posible modificar su contenido.
Esto se debe a que cuando se realiza una nueva asignación, no se modifica la cadena en sí, sino que la variable a pasa a apuntar a otra cadena.

Mutable:

```
def modificar_lista(miLista):
    for i in range(len(miLista)):
        miLista[i] = miLista[i] * 2

lista = [1, 2, 3, 4]
print lista                     # Imprime [1, 2, 3, 4]
modificar_lista(lista)
print "Lista modificada. "
print lista                     # Imprime [2, 4, 6, 8]
```

En nuestro lenguaje elegimos usar el manejo de parámetros que utiliza Processing, el pasaje por valor, ya que con el podemos simular el pasaje por referencia (como se explicó anteriormente), enviando por valor las copias de las referencias de los objetos instanciados y así poder modificar el estado de esos objetos.

E. Realice una tabla comparativa permitiendo visualizar las diferencias más relevantes respecto del sistema de tipos de los lenguajes asignados. Justifique las características mencionadas con ejemplos de código (al menos para 3 de las características).

Processing	Python
Fuertemente Tipado	Fuertemente Tipado
Según tipo de ligadura: Tipado Estático, pero también cuenta con tipado dinámico (casting).	Según tipo de ligadura: Tipado Dinámico
Tipos Primitivos: byte, short, int, long, float, doublé, char, boolean.	Tipos Primitivos: byte, char, boolean, float.
Tipos definidos por el usuario: clases (el programador puede crear sus propias clases e instanciar).	Tipos definidos por el usuario: clases.
Tipos Compuestos: arreglos.	Tipos Compuestos: listas
	Estos tipos de datos pueden ser mutables (ej. listas) o inmutables (ej. string).
Tipado débil	Tipado fuerte

Ejemplo de tipos definidos por el usuario:
La forma de definir una clase es diferente en ambos lenguajes.

-Processing:

```
class Persona {

  // Variables de instancia con especificadores de acceso.
  public String nombre;
  public Integer edad;

  public Persona(String nombre, Integer edad) {              // Constructor
    this.nombre = nombre;
    this.edad = edad;
  }

}

Persona p = new Persona("Iván Salgado", 24);               // Forma de instanciar
System.out.print(p.nombre);
```

-Python:

```
class Persona:

    def __init__(self, nombre, edad):  // Constructor
        self.nombre = nombre
        self.edad = edad

p = Persona('Pepe', 20)                // Forma de instancear
print (p.nombre)
```

Ejemplo de tipos mutables e inmutables:

-Processing:
Este lenguaje no presenta esta característica en sus tipos de datos.

-**Python:**
Los tipos de datos inmutables son aquellos que no es posible modificar el contenido, aunque somo cualquier variable si es posible asignarle un nuevo valor.
Los tipos inmutables son: string, int, long, float, complex, boolean, tuple

Ej:
```
cadena = “hola”
print cadena
```
Si se quiere cambiar la letra ‘h’ por ‘H’, no es posible con este tipo de dato, lo que se tiene que hacer es asignar la nueva cadena a la misma variable.
```
cadena  = “Hola”
print cadena
```

Los tipos de datos mutables son todos aquellos que es posible modificar su contenido. Estos son: listas, diccionarios y conjuntos que sirven para guardar colecciones de datos.

Ejemplo de tipado:
Se dice que el sistema de tipos es fuerte cuando especifica restricciones sobre como las operaciones que involucran valores de diferentes tipos pueden operarse.

-Processing: (debil)

```
Int a = 10;
String b = “hola”;
System.out.println(a + b);      // Imprime 10hola
```

-Python: (fuerte)
```
a = 10
b = “hola”
print str(a) + b          # Se produce un error.
```
Error: TypeError: cannot concatenate 'str' and 'int' objects

Processing				
Tipo	**Inicialización**		**Valores**	**Operaciones**
	Declarar	**Instancia**		
byte	Ninguna	0	-128 a 128	+,-,*,/,%,++,--
short	Ninguna	0	-32768 a 32767	+,-,*,/,%,++,--
int	Ninguna	0	-2147483648 a 2147483647	+,-,*,/,%,++,--
long	Ninguna	0	-9223372036854775808 a 9223372036854775807	+,-,*,/,%,++,--
float	Ninguna	0.0	$\pm 3{,}4 * 10^{-38}$ a $\pm 3{,}4 * 10^{38}$	+,-,*,/,%,++,--
double	Ninguna	0.0	$\pm 1{,}7 * 10^{-308}$ a $\pm 1{,}7 * 10^{308}$	+,-,*,/,%,++,--
boolean	Ninguna	false	true o false	==,!=,<,<=,>,>=, &&, ||, !
char	Ninguna	'\u0000'	'\u0000' a '\uffff'	+,-,*,/ (Con sus valores ASCII)
Objeto	-	null	-	

Python				
Tipo	**Inicialización**		**Valores**	**Operaciones**
	Declarar	**Instancia**		
Int (32 bits)	Ninguna	Ninguna	-2147483648 a 2147483647	+,-,*,/,%,**
Int (64 bits)	Ninguna	Ninguna	-9223372036854775808 a 9223372036854775807	+,-,*,/,%,**
Long	Ninguna	Ninguna	Numero de cualquier precisión, limitado por la memoria	+,-,*,/,%,**
Float	Ninguna	Ninguna	$\pm 2{,}22 * 10^{-308}$ a $\pm 1{,}79 * 10^{308}$	+,-,*,/,%,**
String	Ninguna	Ninguna	Caracteres del código ASCII	+,*

F. Suponiendo que Ud. está definiendo un nuevo lenguaje de programación, defina y enuncie las características más importantes del manejo de excepciones incorporando las características que usted considere necesarias de los lenguajes asignados a su grupo. Ejemplifique con código.

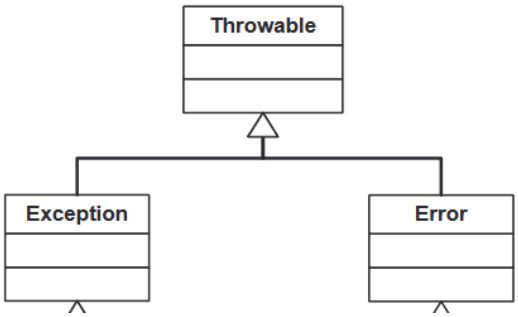
Una excepción es la indicación de que se produjo un error en el programa. Las excepciones, como su nombre lo indica, se producen cuando la ejecución de una instrucción no termina correctamente, sino que termina de manera excepcional como consecuencia de una situación no esperada.

Existen dos tipos de modelo de manejo de excepciones:

- Reasunción: se maneja la excepción y se devuelve el control al punto siguiente donde se invocó la excepción, permitiendo la continuación de ejecución de la unidad. El lenguaje que lo utiliza es PL/1.
- Terminación: se termina la ejecución de la unidad que alcanza la excepción y se transfiere el control al manejador. Los lenguajes que lo utilizan son ADA, CLU, C++, JAVA, PHP, etc

Processing:

El modelo de ejecución es por terminación.
Cuando se produce un error en un método, “se lanza” un objeto *Throwable*. Cualquier método que haya llamado al método puede “capturar la excepción” y tomar las medidas que estime oportunas. Tras capturar la excepción, el control no vuelve al método en el que se produjo la excepción, sino que la ejecución del programa continúa en el punto donde se haya capturado la excepción.
Jerarquía de clases:



- Throwable: Clase base que representa todo lo que se puede “lanzar” en Java.

    - Contiene una instantánea del estado de la pila en el momento en el que se creó el objeto (“stack trace” o “call chain").
    - Almacena un mensaje (variable de instancia de tipo String) que podemos utilizar para detallar qué error se produjo.
    - Puede tener una causa, también de tipo Throwable que permite representar el error que causó este error.
- Error: Subclase de Throwable que indica problemas graves que una aplicación no debería intentar solucionar. Este tipo de excepciones escapan a su control y, por lo general, tu programa no se ocupará de ellas.
- Exception: Exception y sus subclases indican situaciones que una aplicación debería tratar de forma razonable.
    - Los dos tipos principales de excepciones son:
        - *RuntimeException* (errores del programador, como una división por cero o el acceso fuera de los límites de un array)
        - *IOException* (errores que no puede evitar el programador, generalmente relacionados con la entrada/salida del programa)

Manejo:

El manejo de excepciones Java se gestiona a través de cinco palabras clave: *try*, *catch*, *throw*, *throws*,y *finally*. Forman un subsistema interrelacionado en el que el uso de uno implica el uso de otro.

Las declaraciones de programa que desea supervisar para excepciones están contenidas dentro de un bloque **try**. Si se produce una excepción dentro del bloque try, se lanza. El código puede atrapar esta excepción usando **catch** y manejarlo de una manera racional. Las excepciones generadas por el sistema son lanzadas automáticamente por el sistema de tiempo de ejecución de Java. Para lanzar manualmente una excepción, use la palabra clave **throw**. En algunos casos, una excepción arrojada por un método debe ser especificada como tal por una cláusula **throws**. Cualquier código que debe ejecutarse al salir de un bloque try se coloca en un bloque **finally**.

Uso del *Try y Catch:*

En el centro del manejo de excepciones están try y catch. Estas palabras clave trabajan juntas.

Ejemplo:

```
try{
  //bloque de código para monitorear errores
  }
catch (TipoExcepcion1 exOb){
  //Manejador para TipoExepción1
  }
catch (TipoExcepcion2 exOb){
 //Manejador para TipoExepción2
  }
```

Acá, *TipoExcepcion* es el tipo de excepción que ocurrió. Cuando se lanza una excepción, es atrapada por su instrucción catch correspondiente, que luego procesa la excepción. Como muestra la forma general, puede haber más de una declaración catch asociada con un try. El tipo de la excepción determina qué declaración de captura se ejecuta. Es decir, si el tipo de excepción especificado por una instrucción catch coincide con el de la excepción, entonces se ejecuta esa instrucción de catch (y todos los demás se anulan). Cuando se detecta una excepción, exOb recibirá su valor.

Lanzar una excepción:
Es posible lanzar manualmente una excepción utilizando la instrucción throw. Su forma general se muestra como “throw excepcOb;”. excepcOb debe ser un objeto de una clase de excepción derivada de Throwable. Ejemplo que ilustra la instrucción throw arrojando manualmente una ArithmeticException:

```
public class ThrowDemo {
   public static void main(String[] args) {
      try{
         System.out.println("Antes de lanzar excepción.");
         throw new ArithmeticException(); //Lanzar una excepción
      }catch (ArithmeticException exc){
         //Capturando la excepción
         System.out.println("Excepción capturada.");
      }
      System.out.println("Después del bloque try/catch");
   }
}
```

Uso de finally:
Cuando se quiere definir un bloque de código que se ejecutará cuando quede un bloque try/catch. Por ejemplo, una excepción puede causar un error que finaliza el método actual, causando su devolución prematura. Sin embargo, ese método puede haber abierto un archivo o una conexión de red que debe cerrarse.
Tales tipos de circunstancias son comunes en la programación, y hay una forma conveniente de manejarlos: *finally*. Para especificar un bloque de código a ejecutar cuando se sale de un bloque try/catch, incluya un bloque finally al final de una secuencia try/catch.
El bloque finally se ejecutará siempre que la ejecución abandone un bloque try/catch, sin importar las condiciones que lo causen. Es decir, si el bloque try finaliza normalmente, o debido a una excepción, el último código ejecutado es el definido por finally. El bloque finally también se ejecuta si algún código dentro del bloque try o cualquiera de sus declaraciones catch devuelve del método.

Uso de throws:
En algunos casos, si un método genera una excepción que no maneja, debe declarar esa excepción en una cláusula throws. Aquí está la forma general de un método que incluye una cláusula throws:

```
tipo-retorno nombreMetodo(lista-param) throws lista-excepc {
   // Cuerpo
   }
```

Python:

El modelo de ejecución es por terminación.

Los nombres de las excepciones estándar son identificadores incorporados al intérprete (no son palabras clave reservadas). Por ejemplo, *ZeroDivisionError*, *NameError* y *TypeError*.

Ejemplo:

```
While true:
        Try:
```

```
            X = int (input ("Dividendo:"))
            Y = int (input ("Divisor:"))
            If (X < 0) or (Y < 0):
                  myError = ValueError ('solo se aceptan valores positivos')
                  Raise myError
            Print X/Y
      Except ZeroDivisionError:
            Print "Error: Division por 0"
      Except ValueError:
            Print myError
      Else:                                    #solo si no se produjo una
excepción
            Print "dividiendo", X, "-divisor:", Y , "Resultado:" , X/Y
      Finally:
            Print "esto se ejecuta siempre"
```

Manejo de excepciones en Python:
La declaración **try** funciona de la siguiente manera:

- Primero, se ejecuta el bloque try (el código entre las declaraciones try y except).
- Si no ocurre ninguna excepción, el bloque except se saltea y termina la ejecución de la declaración try.
- Si ocurre una excepción durante la ejecución del bloque try, el resto del bloque se saltea. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada except, se ejecuta el bloque except, y la ejecución continúa luego de la declaración try.
- Si ocurre una excepción que no coincide con la excepción nombrada en el except, esta se pasa a declaraciones try de más afuera; si no se encuentra nada que la maneje, es una excepción no manejada, y la ejecución se frena con un mensaje como los mostrados arriba.

Una declaración **try** puede tener más de un **except**, para especificar manejadores para distintas excepciones. A lo sumo un manejador será ejecutado. Sólo se manejan excepciones que ocurren en el correspondiente try, no en otros manejadores del mismo try. Un except puede nombrar múltiples excepciones usando paréntesis

Las declaraciones **try... except** tienen un bloque **else** opcional, el cual, cuando está presente, debe seguir a los except. Es útil para aquel código que debe ejecutarse si el bloque try no genera una excepción.

Definiendo acciones de limpieza:
La declaración **try** tiene otra cláusula opcional que intenta definir acciones de limpieza que deben ser ejecutadas bajo ciertas circunstancias. Una cláusula **finally** siempre es ejecutada antes de salir de la declaración try, ya sea que una excepción haya ocurrido o no. Cuando ocurre una excepción en la cláusula try y no fue manejada por una cláusula except (u ocurrió en una cláusula except o else), es relanzada luego de que se ejecuta la cláusula finally. El Finally es también ejecutado "a la salida" cuando cualquier otra cláusula de la declaración try es dejada via break, continue or return.

En aplicaciones reales, la cláusula finally es útil para liberar recursos externos (como archivos o conexiones de red), sin importar si el uso del recurso fue exitoso.

Levantando excepciones:
La declaración **raise** permite al programador forzar a que ocurra una excepción específica.

Excepciones definidas por el usuario
Los programas pueden nombrar sus propias excepciones creando una nueva clase excepción. Las excepciones, típicamente, deberán derivar de la clase *Exception*, directa o indirectamente.
Las clases de Excepciones pueden ser definidas de la misma forma que cualquier otra clase, pero usualmente se mantienen simples, a menudo solo ofreciendo un número de atributos con información sobre el error que leerán los manejadores de la excepción. Al crear un módulo que puede lanzar varios errores distintos, una práctica común es crear una clase base para excepciones definidas en ese módulo y extenderla para crear clases excepciones específicas para distintas condiciones de error.
La mayoría de las excepciones son definidas con nombres que terminan en "Error", similares a los nombres de las excepciones estándar. Muchos módulos estándar definen sus propias excepciones para reportar errores que pueden ocurrir en funciones propias.

Nuestro lenguaje manejara las excepciones de la siguiente manera:

- Utilizará el criterio de terminación (al igual que nuestros lenguajes asignados), se finaliza el bloque donde se levanto y se ejecuta el manejador asociado.
- Para alcanzarlas se utilizará la palabra clave 'throw'
- Para definirlas se usan las palabras claves try, catch, throw
- Los bloques que pueden llevar a levantar excepciones van precedidos por la palabra clave try y al finalizar el bloque se detallan los manejadores usando la palabra catch (nombreExcepcion)

G. Conclusión sobre los lenguajes: Realice una conclusión exponiendo las diferencias, características, virtudes y defectos de los lenguajes asignados. Esta conclusión no debe superar una carilla

-Processing:

- **Características:**
  - Confiable: al ser fuertemente tipado y contar con un manejo de excepciones.
  - Abstracción: capacidad de definir y usar estructuras u operaciones complicadas ignorando muchos detalles. Por ejemplo, al realizar gráficos.
  - Soporte: es accesible para cualquiera que quiera usarlo.
  - Bindings: permite que el tipo de objeto se determine en ejecución.
- **Ventajas:**
  - Esta basado en Java
  - Fácil de utilizar.

    - Permite utilizar la orientación a objetos.
    - Se puede combinar con aplicación Java y exportar a proyectos web.
- **Desventajas:**
    - Esta específicamente orientado para la producción de proyectos interactivos de diseño digital
    - No permite el uso de variables y métodos estáticos (a menos que se indique que quiere el código en modo puro Java)

-Python:

- **Características:**
    - Simple y legible: tiene una sintaxis muy visual, ya que maneja una sintaxis identada.
    - Confiable: al ser fuertemente tipado y contar con un manejo de excepciones.
    - Ortogonal: permite combinar sus componentes.
- **Ventajas:**
    - Es de libre distribución.
    - Es multiparadigma y multiplataforma.
    - Cuenta con muchas liberarías para utilizar.
    - Desarrollo más rápido (se evitan los pasos de la compilación)
- **Desventajas:**
    - Al ser un lenguaje interpretado lo hace más lento.

H. Conclusión sobre el trabajo: Realice una conclusión mencionando los aportes que le generó la realización del trabajo como grupo.

Nuestra conclusión sobre el trabajo:

La cursada es bastante diferente a las materias que venimos cursando años anteriores, notamos que hay que realizar muchas investigaciones bibliográficas, ya sea por libros o a través internet. Nos resultó interesante ver diferentes tipos de lenguajes de programación, y no solamente uno como surge mayormente en las demás materias. También esta bueno conocer las distintas características que posee cada uno, y de algunos pudimos ver más en profundidad cómo es su sintaxis y semántica, como por ejemplo los lenguajes de Python y Processing que fueron los que nos designaron para el trabajo.
Con respecto al trabajo, nos costó entender algunas consignas ya que no especificaba muy claro lo que teníamos que realizar o nosotros no pudimos entenderlo correctamente, pero pudimos consultarlas en cualquier horario de consulta con los ayudantes y eso nos ayudo mucho en aclarar conceptos, como así también dudas sobre lo que realizamos.