Paradigmas de la Programación - Parcial 3

Temas:

- Frameworks
- Programación Lógica (ProLog)
- Lenguajes de Scripting
- Seguridad en lenguajes de programación: programación defensiva y ofensiva
- Programación Asistida con IA
- Programación distribuida y concurrencia: paradigma de actores

Índice:

Lenguajes de Scripting

→ Glue Languages

→ Domain Specific Language

Características:

- Uso de expresiones regulares
- Comunicación con el SO
- Poca verbosidad
- Construcciones de alto nivel
- Construcciones específicas de dominio
- Mucha abstracción
- No declaración de variables

Estrategias de Programación Defensiva:

- Reuso de Software inteligente (cuidado con el software legacy, o sea, con el uso de software "viejo". Puede haber funcionado antes pero ahora, con las modificaciones de la actualidad, ya no.).
- Sanitización de inputs (modificar los inputs para que tengan una forma que sabemos no romperá nada.)
  - Canonicalización.
  - Tratamiento con whitelist (protegerse de problemas conocidos) /blacklist (protegerse de todo lo desconocido).
- Autenticar/encriptar datos provenientes de la red.
- Asserts.
- Excepciones.

Estrategias de Programación Ofensiva:

- Eliminar todas las estrategias del código que "salvan" errores.

Vulnerabilidades:

- De memoria
- De tipo

Principios de diseño del paradigma de actores:

- Resiliencia.
- Responsividad/Reactividad.
- Flexibilidad/Elasticidad.
- Basado en el envío de mensajes.
- Elasticidad (capacidad de crecer o de mantenerse pequeño de ser necesario).

Características de los mensajes de los actores:

- Sin orden.
- Asíncronos.
- En buffer.

Highlights de clase (cosas que dijo la profe que puede preguntar y trolearte si no viste las clases):

- El modelo-vista-controlador que comunica el modelo y la vista a través de mensajes en un framework puede pensarse como la interacción entre agentes (MCP: Model Context Protocol), que también es a través de mensajes.
- En MCP, el context es lo que hay al rededor de los agentes.
- MCP: protocolo abierto que estandariza cómo las aplicaciones proporcionan contexto a los LLMs. Conecta asistentes con el mundo de los datos y herramientas.

Usa mensajes JSON para conectar Hosts, Clientes y Servidores
Hosts: LLM apps que inician las conexiones.
Clientes: Conectores dentro de los hosts.
Servidores: servicios que proveen el contexto.

- Futuros y Promesas en paradigma de actores son variables que tomarán valor en el futuro.
- Características de los glue language: no son DSL, su función principal es comunicar cosas.

Semánticas concurrentes

- Manejo explícito de threads
- Sincronización
- Exclusión mutua

Ejercicios:

Parcial 16/06/23

2. En el siguiente programa en Prolog, en los lugares del código que ocupan las letras A, B, C y D debería haber alguna de las variables que ya se encuentran en la regla. Escriban a cuál de las variables se corresponde cada una de las letras **[10 pt.]**

```
1  precio(manzana, 0.50).
2  precio(banana, 0.30).
3  precio(mango, 1.20).
4  precio(naranja, 0.40).
5
6  tieneStock(juan, manzana, 5).
7  tieneStock(juan, naranja, 3).
8  tieneStock(sara, mango, 2).
9  tieneStock(sara, banana, 1).
10
11 comprar(Item, Cantidad, Proveedor, Monto) :-
12     precio(A, B),
13     tieneStock(C, D, CantidadStock),
14     Cantidad <= CantidadStock,
15     Precio * Cantidad <= Monto.
```

A y D: Item; B: Precio; C: Proveedor

REPASAR ESTRATEGIAS DE PROGRAMACIÓN DEFENSIVA!!!!!

3. Estos dos fragmentos de código tienen casi la misma semántica, pero uno tiene una vulnerabilidad (vean una pista sobre qué vulnerabilidad en la página siguiente), mientras que el otro aplica una estrategia de programación defensiva para cubrirla.

   Identifiquen cuál es el código con una estrategia de programación defensiva y cuál el que tiene una estrategia de programación ofensiva **[10 pt.]** Nombren cuál es la estrategia de programación defensiva que se aplica y subrayenla en el código **[5 pt.]**

```
1  def f(input_string):
2      output_string = input_string.replace("'", "")
3      return output_string
4
5  def search_products(category):
6      conn = sqlite3.connect('products.db')
7      cursor = conn.cursor()
8
9      f_category = f(category)
10
11     query = "SELECT * FROM products WHERE category = ?"
12
13     cursor.execute(query, (f_category,))
14
15     results = cursor.fetchall()
16     for row in results:
17         print(row)
18
19     conn.close()
20
21 category = input("Enter the product category: ")
22
23 search_products(category)
```

→ Programación defensiva

```
1  def search_products(category):
2      conn = sqlite3.connect('products.db')
3      cursor = conn.cursor()
4
5      query = "SELECT * FROM products WHERE category = '" + category + "'"
6
7      cursor.execute(query)
8
9      results = cursor.fetchall()
10     for row in results:
11         print(row)
12
13     conn.close()
14
15 category = input("Enter the product category: ")
16
17 search_products(category)
```

→ Programación ofensiva

Se usa sanitización del input, particularmente tratamiento con blacklist: se eliminan las comillas del input, que serían algo "desconocido" para la query que estamos intentando llevar a cabo.

4. El siguiente programa es una instancia de *map - reduce*. Identifiquen cuál de las dos funciones, `functionA` o `functionB` se correspondería con *map*, cuál con *reduce* **[10 pt.]**

```
1  data = [
2      "manzana naranja banana",
3      "naranja mango manzana",
4      "banana manzana",
5      "mango naranja"
6  ]
7
8  def functionA(oracion):
9      palabras = oracion.split()
10     return [(palabra, 1) for palabra in palabras]
11
12 def functionB(item):
13     palabra, cuentas = item
14     total_cuenta = sum(cuentas)
15     return palabra, total_cuenta
16
17 with Pool() as pool:
18     intermediate_results = pool.map(functionA, data)
19
20 intermediate_results = [item for
21     sublist in intermediate_results for item in sublist]
22
23 palabra_cuentas = Counter()
24 for palabra, cuenta in intermediate_results:
25     palabra_cuentas[palabra] += cuenta
26
27 for palabra, cuenta in palabra_cuentas.items():
28     print(palabra, cuenta)
```

UO A UA TOMAR!!

7. Según el siguiente texto, las estrategias de Akka para el tratamiento de excepciones en streams, ¿son bloqueantes? **[10 pt.]** ¿con qué principio de diseño del paradigma de actores podrían relacionar el hecho de que sean o no sean bloqueantes? **[5 pt.]**

> Uno de los componentes clave para el manejo de excepciones en Akka Streams es la estrategia de supervisión definida para las etapas de procesamiento del flujo. La estrategia de supervisión determina cómo se gestionan y propagan los fallos dentro del flujo.
> Hay dos estrategias principales de supervisión disponibles en Akka Streams:
>
> **Resume:** Con esta estrategia, cuando se produce una excepción en una etapa del flujo, el elemento que falla se omite, y el procesamiento continúa con el siguiente elemento. El elemento problemático es efectivamente abandonado, y el procesamiento del flujo continúa sin interrupción.
>
> **Restart:** Esta estrategia consiste en reiniciar la etapa que ha fallado, permitiendo que se recupere y continúe el procesamiento desde un estado conocido. Cualquier estado o contexto acumulado se pierde, y la etapa se reinicia como si estuviera recién inicializada.

Sí, son bloqueantes, ya que antes de continuar con el flujo se lleva a cabo por completo alguna de las dos estrategias comentadas. O se omite el elemento y luego el procesamiento continúa ó se reinicia la etapa que falla, se espera que se recupere y luego continúa el procesamiento.

Esto podría relacionarse con los principios de resiliencia (tolerancia a los fallos) y flexibilidad (fácil adaptación a los cambios).

Parcial 16/06/22

2. **[15 pt.]** Lea el siguiente texto:

> *If Await.Result() is called at any point before the Future has completed, the Future becomes blocking. If you instead use onComplete, onSuccess, onFailure, map, or flatMap (and some other methods), you are registering a callback function that will occur when the Future returns. Thus, the Future is non-blocking. Use non-blocking Futures with callbacks whenever possible.*

Este texto nos habla de una propiedad interesante de los futuros. ¿Qué efectos tiene esta propiedad con respecto a la eficiencia de los programas orientados a actores? ¿Con qué decisión de diseño de la orientación a actores la pueden relacionar, y por qué?

Esta propiedad aumenta la eficiencia de los programas orientados a actores, ya que permite hacer no-bloqueantes a los futuros, permitiendo que el manejo de los mensajes por parte de los actores continúe aún cuando se ejecuta un futuro. Se relaciona con la responsividad, ya que, si los futuros son bloqueantes, no se podrá "atender" msjes hasta que termine de ejecutarse, mientras que si son no-bloqueantes sí.

5. **[15 pt.]** En el siguiente texto se explica el comportamiento de los streams en programación reactiva.

> *In computing, reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change. With this paradigm, it's possible to express static (e.g., arrays) or dynamic (e.g., event emitters) data streams with ease, and also communicate that an inferred dependency within the associated execution model exists, which facilitates the automatic propagation of the changed data flow.*
> *For example, in an imperative programming setting,* `a := b + c` *would mean that a is being assigned the result of* `b + c` *in the instant the expression is evaluated, and later, the values of* `b` *and* `c` *can be changed with no effect on the value of* `a`*. On the other hand, in reactive programming, the value of* `a` *is automatically updated whenever the values of* `b` *or* `c` *change, without the program having to explicitly re-execute the statement* `a := b + c` *to determine the presently assigned value of* `a`*.*

Según esta definición, explique qué se imprimirá al ejecutar el siguiente programa si el operador "=" se interpreta como el operador de asignación propio de la programación imperativa, o bien si se interpreta como un operador reactivo, que cambia el valor de la variable del lado izquierdo del operador no solamente cuando se hace la asignación explícitamente, sino también cuando las variables referenciadas en la parte derecha del operador cambian.

```
var b = 1
var c = 2
var a = b + c
b = 10
console.log(a)
```

= en prog. imperativa: imprime 3

= en prog. reactiva: imprime 12

7. **[5 pt.]** En el siguiente programa, identifique los mecanismos de programación defensiva, y **[10 pt.]** explique qué tipo de inseguridad se está tratando de cubrir con esos mecanismos. **[5 pt.]** ¿Cómo se escribiría este programa en programación ofensiva? ¿Cuál es el objetivo de seguridad de escribir un programa como este en su versión ofensiva?

```
static void CreateRandomPermutation(int numbers[], int nNumbers)
{
    assert(numbers != NULL);
    assert(nNumbers >= 0);

    for (int i=0; i<nNumbers; i++)
        numbers[i] = i;

    for (int src=1; src<nNumbers; src++)
    {
        const int dst = GetRandomNumberIn(0, src);
        SwapNumbers(numbers, src, dst);
    }
}
```

→ lleno numbers con nros del 0 a nNumbers-1

→ random nmbr del 0 a 1 / 2 / 3

↳ swap la pos src con la dst

Mecanismos de programación defensiva:

Aserciones

Se está tratando inseguridades de tipo, se chequea que las variables sean del tipo adecuado.

En programación ofensiva se escribiría sin ningún assert. El objetivo de este tipo de programación es que sucedan los errores para reconocerlos.

Recuperatorio 23/06/22

1. **[10 pt.]** ¿Qué características de los lenguajes de scripting se pueden observar en el siguiente programa? Siempre que sea posible, cite los fragmentos específicos de código en los que se observan las propiedades que mencione.

```
tell application "Finder"
    set passAns to "app123"
    set userAns to "John"
    if the text returned of (display dialog "Username" default answer "") is userAns then
        display dialog "Correct" buttons {"Continue"} default button 1
        if the text returned of (display dialog "Username : John" & return & "Password" default answer ""
buttons {"Continue"} default button 1 with hidden answer) is passAns then
            display dialog "Access granted" buttons {"OK"} default button 1
        else
            display dialog "Incorrect password" buttons {"OK"} default button 1
        end if
    else
        display dialog "Incorrect username" buttons {"OK"} default button 1
    end if
end tell
```

Construcciones de alto nivel, poca verbosidad

2. **[25 pt.]** El siguiente programa está escrito en Erlang. Encuentre en el texto del programa por lo menos dos características propias del paradigma de actores y relaciónelas con lo que conoce de Akka.

```
% Create a process and invoke the function web:start_server(Port, MaxConnections)
ServerProcess = spawn(web, start_server, [Port, MaxConnections]),

% Create a remote process and invoke the function
% web:start_server(Port, MaxConnections) on machine RemoteNode
RemoteProcess = spawn(RemoteNode, web, start_server, [Port, MaxConnections]),

% Send a message to ServerProcess (asynchronously). The message consists of a tuple
% with the atom "pause" and the number "10".
ServerProcess ! {pause, 10},

% Receive messages sent to this process
receive
        a_message -> do_something;
        {data, DataContent} -> handle(DataContent);
        {hello, Text} -> io:format("Got hello message: ~s", [Text]);
        {goodbye, Text} -> io:format("Got goodbye message: ~s", [Text])
end.
```

El programa se comunica con el ServerProcess a través de mensajes. (message driven)

El proceso reacciona ante los mensajes que recibe, dependiendo del contenido de dicho mensaje (responsividad)

Akka funciona sobre el paradigma de actores, recibe streams, que son como los mensajes y tmb responde a ellos dependiendo de su contenido.

3. **[10 pt.]** Dada la siguiente base de conocimiento, ¿qué va a contestar el intérprete si le preguntamos `digiriendo(rana,mosca).`, y cómo va a llegar a su respuesta?

```
digiriendo(X,Y) :- comio(X,Y).
digiriendo(X,Y) :-
        comio(X,Z),
        digiriendo(Z,Y).
comio(mosquito,sangre(juan)).
comio(rana,mosquito).
comio(gaviota,rana).
```

rana, Z => sí, rana, mosquito

mosquito, mosca

Responde: NO

Camino para llegar: 1- chequea digiriendo(rana, mosca) -> comio(rana, mosca), esto contesta NO.

2- chequea digiriendo(rana, mosca) -> comió(rana, Z) -> comió(rana, mosquito) -> digiriendo (mosquito, mosca) -> comió(mosquito, mosca) -> NO

digiriendo(mosquito, mosca) -> comió(mosquito, Z) -> comió(mosquito, sangre(juan)) -> digiriendo(sangre(juan), mosca) -> digiriendo(sangre(juan), mosca) -> NO

digiriendo(sangre(juan), mosca) -> comió(sangre(juan), Z) -> NO

4. **[30 pt.]** En el siguiente texto nos explican una característica muy interesante de Elm. Esta característica, ¿contribuye negativamente o positivamente a la seguridad del lenguaje? ¿por qué? ¿y a la robustez del lenguaje? ¿Qué mecanismos de programación defensiva podríamos aplicar en otro lenguaje para tener un comportamiento parecido al comportamiento que presenta Elm en este programa?

One of the guarantees of Elm is that you will not see runtime errors in practice. This is partly because Elm treats errors as data. Rather than crashing, we model the possibility of failure explicitly with custom types. For example, say you want to turn user input into an age. You might create a custom type like this:

```
type MaybeAge
  = Age Int
  | InvalidInput

toAge : String -> MaybeAge
toAge userInput =
  ...

-- toAge "24" == Age 24
-- toAge "99" == Age 99
-- toAge "ZZ" == InvalidInput
```

Instead of crashing on bad input, we say explicitly that the result may be an Age 24 or an InvalidInput. No matter what input we get, we always produce one of these two variants. From there, we use pattern matching which will ensure that both possibilities are accounted for. No crashing!

Contribuye positivamente ya que evita que el programa se detenga culpa de un error, además explicita el error tratándolo como data y pudiendo devolverlo luego de que finalice la ejecución de nuestro programa.
En otros lenguajes deberíamos aplicar excepciones, para detectar el error, "tomarlo" (catch) y hacer algo con él, como imprimirlo por pantalla, para que luego siga la ejecución del programa normalmente. Sino, podríamos utlizar if's.

5. **[25 pt.]** El siguiente texto explica los conceptos de *hot spot* y *frozen spot* en frameworks. También nos explica cómo funcionan los frameworks con orientación a objetos. Cuando instanciamos una aplicación con un framework basado en objetos, terminamos teniendo un sistema de objetos específico para esa aplicación, basado en el sistema de objetos provisto por el framework. Basándose en los conceptos de *concreto* y *abstracto* y *composición* y *subclases* que usa el texto, explique qué componentes de este sistema de objetos se podrían considerar *frozen spots* y cuáles se podrían considerar *hot spots*.

> *Software frameworks consist of frozen spots and hot spots. Frozen spots define the overall architecture of a software system, that is to say its basic components and the relationships between them. These remain unchanged (frozen) in any instantiation of the application framework. Hot spots represent those parts where the programmers using the framework add their own code to add the functionality specific to their own project.*
>
> *In an object-oriented environment, a framework consists of abstract and concrete classes. Instantiation of such a framework consists of composing and subclassing the existing classes.*

Frozen spots: clases concretas y abstractas.

Hot spots: composiciones y subclases de ellas.

Recuperatorio 22/06/2023

1. ¿Qué tipo de lenguaje de scripting observamos en el siguiente ejemplo? **[5 pt.]** Nombren por lo menos 2 características de este tipo de lenguajes de scripting **[10 pt.]**

```
1  // Player entity variables
2  entity player;
3  float playerSpeed = 100.0;
4
5  void main() {
6      // Initialize player entity
7      player = spawn();
8      setmodel(player, "player.mdl");
9      setsize(player, Vector(-16, -16, 0), Vector(16, 16, 72));
10     setorigin(player, Vector(0, 0, 0));
11     player.think = Player_Think;
12     player.movetype = MOVETYPE_WALK;
13     player.solid = SOLID_BBOX;
14
15     // Start the game loop
16     while (1) {
17         // Process keyboard input
18         if (key_down(K_LEFTARROW)) {
19             player.velocity_y = -playerSpeed;
20         } else if (key_down(K_RIGHTARROW)) {
21             player.velocity_y = playerSpeed;
22         } else {
23             player.velocity_y = 0;
24         }
25
26         // Update the player entity
27         setorigin(player, player.origin + player.velocity * frametime);
28
29         // Update the game state
30         progs_run();
31     }
32 }
```

Observamos un DSL (Domain Specific Language). Se caracterizan, entre otras cosas, por brindar herramientas para resolver problemas específicos de un área (en este caso, videojuegos) y, al igual que los demás lenguajes de scripting, por tener construcciones de muy alto nivel, que permiten abstraerse del código subyacente a dichas construcciones (en este caso, la entidad player y las instrucciones que "trae" como think, velocity, etc.)

3. En el siguiente programa en Prolog, en los lugares del código que ocupan las letras A, y C debería haber alguna de las variables que ya se encuentran en la regla. Escriban a cuál de las variables se corresponde cada una de las letras **[10 pt.]** y expliquen la función de la variable B **[5 pt.]**

```
1  package(p1, location(a)).
2  package(p2, location(b)).
3  package(p3, location(c)).
4  package(p4, location(d)).
5
6  transport(truck, a, b).
7  transport(plane, b, c).
8  transport(train, c, d).
9  transport(ship, d, a).
10
11 connected(X, Y) :-
12     transport(_, X, Y).
13 connected(X, Y) :-
14     transport(_, Y, X).
15
16 reachable(Package, Destination) :-
17     package(A, B),
18     find_route(B, C, [Package]).
19
20 find_route(Source, Destination, Visited) :-
21     connected(Source, Destination),
22     \+ member(Destination, Visited).
23
24 find_route(Source, Destination, Visited) :-
25     connected(Source, Interim),
26     \+ member(Interim, Visited),
27     find_route(Interim, Destination, [Interim|Visited]).
```

A: Package; C: Destination; B: Source

4. En el siguiente fragmento de código se está aplicando una estrategia de programación defensiva. Identifíquela **[10 pt.]** y describa cómo funciona **[5 pt.]**

```
1  def divide_numbers(dividend, divisor):
2      if divisor == 0:
3          raise ValueError("Cannot divide by zero.")
4      result = dividend / divisor
5      return result
6
7  def get_user_input():
8      try:
9          dividend = int(input("Enter the dividend: "))
10         divisor = int(input("Enter the divisor: "))
11         result = divide_numbers(dividend, divisor)
12         print(f"The result of the division is: {result}")
13     except ValueError as e:
14         print("Invalid input:", e)
15     except Exception as e:
16         print("An error occurred:", e)
17
18 get_user_input()
```

Estrategia: excepciones e if. En la función divide_numbers se chequea si el divisor es 0 usando un if. Si esto se cumple se levanta una excepción "ValueError" que será atrapada por el except correspondiente y el error será notificado al usuario por pantalla.

5. En el siguiente programa, identifique los mecanismos por los cuales se garantiza concurrencia declarativa **[10 pt.]** y explique cómo funcionan esos mecanismos para evitar la existencia de una sección crítica **[5 pt.]**

```
1  from actor import Actor, ActorSystem
2
3  # Actor implementation
4  class Counter(Actor):
5      def __init__(self):
6          self.count = 0
7
8      def receive(self, message, sender):
9          if message == 'increment':
10             self.count += 1
11             print(f'Counter: {self.count}')
12         elif message == 'get_count':
13             sender.send(self.count)
14
15 # Create an actor system
16 system = ActorSystem()
17
18 # Create an instance of the Counter actor
19 counter = system.create_actor(Counter)
20
21 # Send messages to the Counter actor
22 counter.send('increment')
23 counter.send('increment')
24 counter.send('get_count')
25
26 # Wait for the response
27 response = system.receive()
28
29 # Print the count
30 print(f'Final count: {response}')
31
32 # Shut down the actor system
33 system.shutdown()
```

Se garantiza concurrencia declarativa mediante el uso de actores que se comunican a través de mensajes que envían o reciben, no hay variables globales ni asignación destructiva, sino que el comporatmiento (qué devuelven) de los actores dependerá de sus parámetros (msjes que reciben).
La inexsistencia de variables globales implica la inexsistencia de variables compartidas y esto a su vez implica la inexsistencia de regiones críticas.