FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 1 — Scala

[TB025] Paradigmas de la Programación
Curso: 02 Brasburg, Raik
Primer cuatrimestre de 2025

Alumnos:	Tomás de Matos Cancina Matías Agustín Daniel Burgos Moreno Caropreso Alejandro
Números de padrón:	112277 111735 110486 111427
Email:	Tdematos@fi.uba.ar Mcancina@fi.uba.ar Dburgos@fi.uba.ar Acaropreso@fi.uba.ar

Índice

1. Copilacion del Programa **2**

2. Funcionalidades **2**

3. Decisiones tomadas respecto al procesamiento, modelado en ejecución y manipulación de los documentos XML y stylesheets XSLT **2**
3.1. Modelado . . . 2
3.2. Uso de datos inmutables . . . 3
3.3. Estructuración recursiva . . . 3
3.4. Modelado de etiquetas con `case class` . . . 3

4. Fragmentos del código donde se vio la necesidad de salirse de los conceptos de programación funcional **3**
4.1. Manejo de errores mediante excepciones . . . 4
4.2. Acceso a los parametros y paths para el procesamiento de archivos . . . 4

5. Hipótesis tomadas durante la realización del trabajo **4**
5.1. XML/XSLT bien formado . . . 4
5.2. Sin etiquetas auto-cerradas . . . 4
5.3. Variables globales en XSLT . . . 4
5.4. Estructura funcional del XML y XSLT . . . 4
5.5. Modelo funcional para instrucciones XSLT . . . 4
5.6. Formato de parámetros del programa . . . 5
5.7. Restricción a estructuras funcionales . . . 5
5.8. Múltiples raíces en XML . . . 5
5.9. Tratamiento del contenido textual . . . 5

6. Excepciones en el procesamiento del XML/XSLT **5**
6.1. Excepción por cierre inesperado de etiqueta . . . 5
6.2. Excepción por etiquetas abiertas sin cerrar . . . 6

7. Conclusión **6**

1. Copilacion del Programa

Compilacion:
La compilacion tendra que tener el siguiente formato:
scalac src/*.scala src/Transformaciones/*.scala

Ejecución:
Para la ejecucion del programa se requeriran los siguentes archivos que seran pasados por consola:

- Un archivo del tipo XML.
- Un archivo del tipo XSLT para la ejecución correcta.

La ejecucion sin parametros tendra que ser de la siguiente manera:
scala main rutaXsl RutaXml

La ejecucion con parametros tendra que ser de la siguiente manera:
scala main rutaXsl RutaXml –param1=pepe

Un ejemplo de compilacion y ejecucion simultanea seria de la siguente manera:
scala src/*.scala src/Transformaciones/*.scala – src/coso src/cosoXml –param=Juan –param=maria

2. Funcionalidades

En este proyecto, se implementaron varias funcionalidades que procesan documentos XML mediante transformaciones XSLT.
Las funciones implementadas son:

- Template
- Value-Of
- Copy-Of
- Copy
- Param
- Attribute-Of
- Sort-By-Attr

En la carpeta xsltTestFiles podran encontrar archivos *template* del tipo **xslt** para ser ejecutadas junto a un archivo *base* **XML** de testeo, esperando como la respueste de la ejecucion de estos 2 archivos el resultado dentro de **result.txt**

3. Decisiones tomadas respecto al procesamiento, modelado en ejecución y manipulación de los documentos XML y stylesheets XSLT

3.1. Modelado

La implementación del procesamiento y modelado de los documentos XML y XSLT se llevó a cabo siguiendo un enfoque funcional, aprovechando las características de Scala para trabajar con datos inmutables y funciones puras.

Durante el desarrollo del trabajo, consideramos un enfoque basado en una estructura de árbol/nodos, optamos a estos nodos representarlos como las etiquetas, creando nuestra estructura de **TAG**, que se componen de los siguientes elementos:

- **key:String**: El nombre de la etiqueta.
- **tagType: String**: El tipo de formato de la etiqueta **(XSLT - XML)**.
- **params: Map[String, String]**: Los parámetros de la etiqueta.
- **childs: Either[List[Tag], String]**: los hijos de la etiqueta.

Para asegurar la validez del XML, se procesan los archivos siguiendo las funcionalidades definidas en los archivos `Procesador.scala` y `utils.scala`, lo que nos permitió lograr un enfoque más organizado y preciso al resolver el problema.

En cuanto a los archivos `XML` y `XSLT`, se asume que **`"Variable"`** y **`"Param-Id"`** son variables globales en ambos contextos. Esto garantiza que estas variables puedan ser utilizadas en cualquier parte del código o de las transformaciones.

El archivo `Procesador.scala` maneja la lógica de las transformaciones de XSLT sobre el XML. Este archivo procesa las etiquetas siguiendo los comandos `psxl:`, aplicando las funciones correspondientes según el `pattern matching` implementado.

Por otro lado, el archivo `utils.scala` contiene funciones auxiliares para facilitar la implementación de las transformaciones, optimizando el proceso de desarrollo y asegurando que las reglas se puedan modificar fácilmente.

3.2. Uso de datos inmutables

No se utilizaron bibliotecas mutables, garantizando que todas las colecciones (como `List`, `Map`, `Either`, etc.) fueran **inmutables por defecto**. Esta decisión tomada en base a que todo el proyecto fue orientado al uso del Paradigma funcional. Por ejemplo, las etiquetas XML se modelan como `case class`, utilizando `List[Tag]` para almacenar los hijos de una etiqueta y `Map[String, String]` para sus parámetros, todo de manera inmutable.

3.3. Estructuración recursiva

El procesamiento de las líneas del XML se maneja mediante funciones recursivas, como `interpretarLineas`. Este enfoque evita el uso de bucles imperativos y favorece la claridad del código. Además, al utilizar `@tailrec` se previene el desbordamiento de pila al trabajar con grandes volúmenes de datos.

3.4. Modelado de etiquetas con `case class`

Las etiquetas XML se modelan utilizando `case class`, lo que permite aprovechar sus características de inmutabilidad y su capacidad para descomponer objetos de manera eficiente mediante **pattern matching**. Esta representación es ideal para el procesamiento funcional, ya que cada vez que se crea una nueva instancia de `Tag`, no se modifica la instancia original.

4. Fragmentos del código donde se vio la necesidad de salirse de los conceptos de programación funcional

A pesar de que la mayor parte del trabajo se realizó siguiendo principios funcionales, hubo situaciones donde se recurrió a enfoques más imperativos o se realizaron excepciones debido a requisitos específicos del problema:

4.1. Manejo de errores mediante excepciones

En la función `procesarCierre`, se optó por lanzar excepciones (`throw`) para manejar errores de formato en el XML, como etiquetas de cierre inesperadas. Aunque el enfoque funcional prefiere manejar errores mediante tipos como `Option` o `Either`, se eligió usar excepciones para simplificar el código y asegurar la correcta formación del XML, dado que se buscaba un manejo explícito de los errores.

4.2. Acceso a los parametros y paths para el procesamiento de archivos

Durante la ejecución del programa, se accede a los argumentos de entrada mediante su posición en el arreglo `args`, específicamente con expresiones como `args(0)`. Si bien este método es funcional desde el punto de vista práctico y ampliamente utilizado en programas de consola, representa una leve desviación del paradigma funcional estricto.

5. Hipótesis tomadas durante la realización del trabajo

Para facilitar el desarrollo y garantizar que el trabajo se realizara dentro de los límites establecidos, se asumieron algunas hipótesis que simplificaron el procesamiento del XML:

5.1. XML/XSLT bien formado

Se asumió que el XML/XSLT de entrada está correctamente formado, con todas las etiquetas de apertura y cierre balanceadas. Esta suposición permitió enfocarse en la correcta interpretación de las etiquetas.

5.2. Sin etiquetas auto-cerradas

Se asumió que el XML no contiene etiquetas auto-cerradas (como `<tag/>`). Aunque este tipo de etiquetas es válido en XML, no se consideraron para simplificar el modelo y evitar complicaciones adicionales en el análisis.

5.3. Variables globales en XSLT

Durante la implementación del XSLT, asumimos que las variables globales como `Variable` y `Param-Id` tienen un comportamiento consistente y pueden ser utilizadas globalmente en el documento. Esto simplifica el manejo de las transformaciones, sin necesidad de un contexto más complejo.

5.4. Estructura funcional del XML y XSLT

Se asume que tanto el archivo XML como el XSLT pueden representarse como estructuras en forma de árbol, donde cada nodo (`Tag`) contiene una clave, parámetros (atributos) opcionales y una lista de hijos o contenido textual. Este diseño permite una manipulación completamente recursiva y acorde a los principios de la programación funcional.

5.5. Modelo funcional para instrucciones XSLT

Se considera que las instrucciones XSLT pueden interpretarse mediante funciones puras, sin necesidad de un motor de transformación externo ni estructuras mutables. Cada regla se implementa como una función que transforma nodos del XML de forma determinista, manteniendo la inmutabilidad de los datos.

5.6. Formato de parámetros del programa

Los argumentos adicionales pasados al programa se asumen en formato `-clave=valor`, sin espacios ni caracteres especiales en las claves. Esta estructura permite extraer los parámetros fácilmente y representarlos como un `Map[String, String]` para su posterior uso funcional.

5.7. Restricción a estructuras funcionales

Se establece una restricción autoimpuesta al uso exclusivo de programación funcional: no se utilizan estructuras imperativas como `var`, `if-else` o `for`. El control de flujo se resuelve mediante `match`, recursión y uso de colecciones inmutables, en consonancia con las buenas prácticas del paradigma funcional en Scala.

5.8. Múltiples raíces en XML

El archivo XML puede contener múltiples etiquetas de nivel superior (sin un nodo raíz único). Por ello, la función `textoATag` devuelve una `List[Tag]` en lugar de una única raíz, lo cual habilita el procesamiento paralelo o secuencial de múltiples estructuras independientes.

5.9. Tratamiento del contenido textual

El contenido de texto dentro de un nodo XML se representa como `Right(String)` en el campo `childs`, diferenciándose explícitamente de los hijos estructurales, que se representan como `Left(List[Tag])`. Esta distinción facilita el pattern matching y evita ambigüedades durante las transformaciones.

6. Excepciones en el procesamiento del XML/XSLT

El procesamiento de los documentos XML requiere que estos estén correctamente estructurados. Si el XML no sigue un formato válido, el sistema lanzará excepciones indicando qué elementos faltan o están mal formados. Un formato considerado válido sigue las siguientes reglas:

- Cada línea debe contener solo una etiqueta de inicio, una de cierre o un valor.
- No se permiten etiquetas mixtas (etiquetas de inicio y cierre en una misma línea).
- Las etiquetas de comentario no se permiten.

Durante el desarrollo del procesamiento del XML, se manejan dos tipos principales de excepciones que son fundamentales para garantizar que el archivo XML o XSLT sea procesado correctamente y se manejen los casos de error de manera adecuada. A continuación se detallan las excepciones implementadas.

6.1. Excepción por cierre inesperado de etiqueta

Código relevante: $case_{=} > thrownewIllegalArgumentException(s"XMLmalformado : etiquetadecierreinesperada < /key > .")$

Formato de la excepción:

- **Contexto:** Esta excepción es lanzada en la función `procesarCierre`, que es responsable de procesar las etiquetas de cierre en el XML o XSLT. Si contiene una etiqueta de cierre sin una apertura correspondiente, se lanza esta excepción.

6.2. Excepción por etiquetas abiertas sin cerrar

Código relevante:

$caseNil if aperturas.nonEmpty => thrownewIllegalArgumentException("XMLmalformado: faltanetiquetasdecierre.")$

Formato de la excepción:

- **Contexto:** Esta excepción se lanza en la función `interpretarLineas`, cuando el procesamiento de todas las líneas del XML o XSLT ha finalizado, pero la lista de aperturas (`aperturas`) aún contiene elementos. Esto indica que algunas etiquetas de apertura no han sido cerradas adecuadamente.

7. Conclusión

Durante el desarrollo del trabajo práctico, logramos cumplir con el objetivo de implementar un procesador de transformaciones XSLT sobre documentos XML utilizando el paradigma funcional. A lo largo del proceso, se tomaron decisiones importantes que permitieron modelar los documentos como estructuras inmutables y procesarlos mediante funciones puras, respetando las restricciones del paradigma propuesto.

Enfrentamos varios desafíos, especialmente al momento de adaptar patrones comunes de programación imperativa —como ciclos, variables mutables o condicionales tradicionales— a estructuras funcionales. Esto nos llevó a repensar enfoques y rediseñar soluciones de manera más declarativa y recursiva. Uno de los mayores aprendizajes fue entender que muchos problemas pueden resolverse con estructuras similares a las conocidas (como árboles), pero adaptadas al contexto del paradigma funcional (como nuestro modelo de `Tag`).

Además, comprendimos que, en el contexto funcional, la claridad, inmutabilidad y la expresividad del código prevalecen sobre la eficiencia imperativa inmediata, favoreciendo un desarrollo más estructurado y predecible. Logramos construir un sistema que permite representar, interpretar y transformar XML mediante XSLT, utilizando un modelo interno que abstrae la complejidad del formato original de los archivos.

El trabajo nos permitió afianzar conceptos clave de la programación funcional, como el uso de `case class`, `Either`, `Map`, `List`, y el `pattern matching` para el flujo de control. También resaltó la importancia de asumir hipótesis claras y coherentes al momento de diseñar un sistema, para poder acotar correctamente el problema y enfocarse en su resolución efectiva.