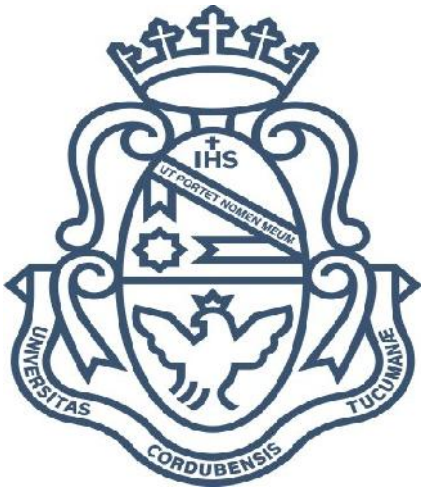
Mapeos de lenguaje de consulta a esquemas XML

Autor: Elisa Fabiana Montes
Director: Dr. Juan Eduardo Durán



Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba
Argentina

Trabajo especial para título de grado
Licenciada en Ciencias de la Computación
en la Universidad Nacional de Córdoba
· 2018 ·

A mi familia que siempre me apoya en cada decisión que he tomado.

AGRADECIMIENTOS

Agradezco principalmente a mi director, el Dr. Juan Durán, por haber depositado en mi su confianza en mi para realizar este trabajo, por haberme ayudado y acompañado en esta etapa y por hacerse un tiempo siempre para despejar mis dudas.

Al la Ing. Andrea Masuero por su apoyo en esta etapa.

A mis amigas, amigos y compañeras y compañeros de trabajo, gracias por estar y ser parte de mi vida.

Córdoba, 28 de marzo del 2018

Resumen: Recientemente se ha estudiado cómo transicionar o mapear modelos de requisitos junto con esquemas de datos a diagramas de interfaz de usuario (UI). Pero no se encontraron trabajos que partan de una etapa previa, o sea desde expresiones de consulta de lenguajes de consulta tradicioneales. Hacer esto requeriría mapear expresiones de consulta a esquemas de datos. Hasta el momento solo encontramos un trabajo que mapea consultas XQuery a esquemas XML pero este trabajo considera solo una parte de XQuery y solo da ideas de cómo mapear llamadas a funciones y expresiones XPath a pesar de que esos problemas son bastante complejos. Por lo tanto el problema considerado es la transformación de expresiones de consulta SQL a esquemas de datos y completar y expandir el trabajo previo sobre mapeo de consultas XQuery a esquemas XML. En este trabajo consideramos la transformación a XML esquemas de consultas XQuery, de consultas SQL y de esquemas de datos en SQL; para esto se trabajó con el espacio tecnológico de los lenguajes definidos por medio de gramáticas BNF y se expresaron transformaciones usando attribute grammars. Para implementar dichas transformaciones se trabajó con lex, yacc y el lenguaje de programación C.

Abstract: Recently it has been studied how to transition or map requirement models and schemas of data onto user interface diagrams; however, we did not find in the literature papers that make this transition from a previous stage starting with query expressions in traditional query languages. To deal with this problem we can map query expressions onto data schemas. We have only found a paper that maps queries in XQuery onto XML schemas, but this work only considers a part of XQuery and for the transformation of function call and of XPath expressions (these problems are rather complex) they only give some ideas and not a complete solution. Therefore, the problem considered in this report is the transformation of SQL query expressions into data schemas and to extend and complete the previous work mapping XQuery queries onto XML Schemas. In this work we considered the transformation to XML Schemas of XQuery, SQL queries and SQL data schemas; for this purpose we based this study on the approach of BNF grammars to define languages and the use of attribute grammars for defining transformations. To define these query languages and implement the transformations we worked with lex, yacc and the C programming language.

Palabras claves: consultas SQL, Xquery, XPath, esquemas XML.

Índice General

1. Introducción

Un tema poco estudiado es la transformación de expresiones de consulta en lenguajes de consulta a esquemas de datos. Hasta el momento solo hemos encontrado [1] que transforma consultas XQuery a esquemas XML.

El transformar expresiones de consulta a esquemas de datos tiene la utilidad de que esos esquemas de datos pueden usarse junto con descripciones de requisitos para generar una parte importante de la interfaz del usuario (UI) de la aplicación.

En la práctica se ha estudiado como hacer la transición de modelos de requisitos junto con esquemas de datos asociados a determinadas partes de requisitos a una parte importante de la UI (ver [2,3]). En particular este tipo de estudio se ha hecho con diagramas de tareas y esquemas de relación, y con diagramas de tareas y esquemas de datos orientados a objetos. Pero no hemos encontrado trabajos que partan desde un paso previo, o sea, desde expresiones de consulta de lenguajes de consulta tradicionales. El único caso que tenemos conocimiento es WebML (ver [4]) que define un lenguaje de consultas para modelado de entidad-relación y parte de consultas en ese lenguaje para generar elementos de UI; sin embargo, no se consideran lenguajes de consulta tradicionales y podría haber consultas en estos lenguajes si se encaró el diseño de los datos antes que el diseño de la aplicación o de su UI.

En la transformación de XQuery a esquemas XML de [1] se considera un subconjunto de XQuery bastante reducido y determinados problemas no se los resuelve en suficiente detalle y para ellos solo se dan algunas ideas (por ejemplo, esto pasa con el mapeo de expresiones XPath a esquemas XML y mapeo de declaraciones y llamadas de funciones a esquemas XML) y la solución a dichos problemas tienen bastante complejidad.

Por lo tanto el problema considerado es la transformación de expresiones de consulta SQL a esquemas de datos y completar y expandir el trabajo en [1].

Una pregunta es si conviene o no que los lenguajes de los esquemas de datos resultantes de las transformaciones para los distintos lenguajes de consulta sean iguales o no. Como al generar la UI a partir de esquemas de datos, no queremos tener que procesar esquemas de datos en más de un lenguaje de esquemas de datos, decidimos que todas las transformaciones de lenguajes de consultas sean al mismo esquema de datos.

Dada la expresividad de los esquemas XML para describir esquemas de datos decidimos tener tres transformaciones, una de SQL a esquemas XML y otra de XQuery a esquemas XML, y otra de esquemas SQL a esquemas XML.

Los objetivos de este trabajo son:

- Mapear consultas XQuery a esquemas XML de una manera más completa y abarcando un subconjunto mayor del lenguaje.
- Mapear consultas SQL a esquemas XML
- Mapear esquemas SQL a esquemas XML

Para hacer dichas transformaciones se trabajó con el espacio tecnológico de los lenguajes definidos por medio de gramáticas y se expresaron las transformaciones usando attribute grammar. Para implementar dichas transformaciones se usaron lex, yacc y el lenguaje de programación C.

El trabajo está organizado como sigue: en el capítulo 2 se definen conceptos básicos referidos a XQuery, esquemas XML; en el capítulo 3 se estudia el mapeo de las consultas y esquemas SQL a esquemas XML (primero se presenta una gramática para SQL, luego se hace mapeo de

esquemas de datos de SQL a esquemas XML y luego otro mapeo de consultas SQL a esquemas XML); en el capítulo 4 se presenta el mapeo de consultas XQuery a esquemas XML (primero se presenta una gramática para XQuery y luego por medio de attribute gramar se presenta el mapeo de consultas XQuery a esquemas XML; dos subproblemas complejos e importantes de este mapeo son el mapeo de expresiones XPath y de expresiones de declaración y llamada de funciones – sobre todo las definidas por el usuario).

2. Conceptos Básicos

2.1 Xquery

XQuery, también conocido como XML Query, es un lenguaje creado para buscar y extraer elementos y atributos de documentos XML. La mejor forma de entender este lenguaje es diciendo que Xquery es para XML lo que SQL es para las bases de datos relacionales. XQuery es un lenguaje de consulta para escribir consultas sobre colecciones de datos expresadas en XML.
Su principal función es extraer información de un conjunto de datos organizados como un árbol n-ario de etiquetas XML. En este sentido XQuery es independiente del origen de los datos.
XQuery es un lenguaje funcional, lo que significa que en vez de ejecutar una lista de comandos como un lenguaje procedimental clásico, cada consulta es una **expresión** que es evaluada y devuelve un resultado, al igual que en SQL. Diversas expresiones pueden combinarse de una manera muy flexible con otras expresiones para crear nuevas expresiones más complejas y de mayor potencia semántica.
En cada una de las siguientes subsecciones se va explicar una clase diferente de expresiones Xquery.

Algunos conceptos importantes

Para realizar los ejemplos de las terminologías que se van a definir y otros conceptos se va a utilizar el **ejemplo 1:** el documento XML de ejemplo.

Nodo: Hay siete tipos de nodos: elemento, atributo, texto, espacio de nombres, instrucciones de procesamiento, comentario y nodos de documento. Los nodos son usados para representar constructores XML como elemento o atributo.
Los documentos XML se tratan como árboles de nodos. El elemento superior del árbol se llama elemento raíz. En el ejemplo 2 se muestra ejemplos de nodo elemento, atributo y raíz.

Ejemplo 1:el documento XML de ejemplo. anterior:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
 <book>
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
        <year>2005</year>
  <price>29.99</price>
 </book>
</bookstore>
```

Ejemplo 2: nodos en el documento XML

```
<bookstore> (nodo elemento raiz)
<author>J K. Rowling</author> (nodo elemento)
   lang="en" (nodo atributo)
```

Valores atómicos: Los valores atómicos son nodos sin hijos o padres.
Ejemplo 3: valores atómicos:
J K. Rowling
"en"
Items: Items son valores atómicos o nodos
Secuencia (sequence) : una lista ordenada de cero, uno o mas items.

Relación de Nodos

Padre (Parent): Cada elemento y atributo tiene un padre. Por ejemplo del archivo XML del ejemplo 1, el elemento book es el padre de title, author, year y price; también tenemos el elemento raíz bookstore es el padre de book

Hijos (Children): Los nodos de elementos pueden tener cero, uno o más hijos. Por ejemplo del archivo XML del ejemplo 1; tenemos a los elementos title, author, year y price son todos hijos del elemento book.
Hermanos (Siblings): Nodos que tienen el mismo padre. Por ejemplo del archivo XML del ejemplo 1; tenemos a los elementos title, author, year y price son todos hermanos.
Ancestros (Ancestors): El padre de un nodo, el padre de un padre, etc. Por ejemplo del archivo XML del ejemplo 1; tenemos los ancestros del elemento title son el elemento book y el elemento bookstore.
Descendientes (Descendants): Los hijos de un nodo, los hijos de un hijo, etc. Por ejemplo del archivo XML del ejemplo 1; tenemos los descendientes del elemento bookstore son los elementos book, title, author, year y price.

Expresiones XPath

XPath es un lenguaje que permite definir un subconjuntos de nodos de un documento XML para su posterior procesamiento e identificar partes específicas de un documento XML.
Una expresión Xpath sirve para localizar nodos en un árbol de nodos de un documento XML.

XPath usa expresiones de ruta para seleccionar nodos o conjuntos de nodos en un documento XML.

Ejemplo 4 : catalog.xml

```
<catalog>
    <product dept="WMN">
      <number>557</number>
      <name language="en">Fleece Pullover</name>
      <colorChoices>navy black</colorChoices>
   </product>
   <product dept="ACC">
      <number>563</number>
      <name language="en">Floppy Sun Hat</name>
   </product>
   <product dept="ACC">
     <number>443</number>
     <name language="en">Deluxe Travel Bag</name>
    </product>
    <product dept="MEN">
       <number>784</number>
       <name language="en">Cotton Dress Shirt</name>
        <colorChoices>white gray</colorChoices>
        <desc>Our <i>favorite</i> shirt!</desc>
    </product>
</catalog>
```

Expresion de ruta de ubicación

Cada expresión XPath consta de varios pasos,
ruta: paso1/paso2/paso3/…/… que se separan por "/" o "//".
"/", significa que el siguiente paso se encuentra adyacente en la jerarquía de nodos, es decir, es un hijo directo del nodo. "//", permite localizar cualquier nodo que sea descendiente sin importar el nivel.
Por ejemplo el camino o ruta:

```
doc("catalog.xml")/catalog/product
```

Selecciona todos los elementos product desde el documento catalog.xml.
Las expresiones de ruta se utilizan para atravesar un árbol XML para seleccionar elementos y atributos de interés. Son similares a las rutas utilizadas para los nombres de archivo en muchos

sistemas operativos.Consisten en una serie de pasos, separados por barras, que atraviesan los elementos y atributos en los documentos XML. En este ejemplo, hay tres pasos:
1. **doc ("catalog.xml")** llama a una función XQuery llamada doc, pasándole el nombre del archivo para abrir.
2. **catalog** selecciona el elemento catalog, el elemento más externo del documento
3. **product** selecciona todos los elementos product hijos del elemento catalog
El resultado de la consulta serán los cuatro elementos **product**, exactamente como aparecen (con los mismos atributos y contenidos) en el documento de entrada. El ejemplo 5 muestra:
Ejemplo 5 Cuatro elementos product seleccionados desde el elemento catalog

```
<product dept="WMN">
      <number>557</number>
      <name language="en">Fleece Pullover</name>
      <colorChoices>navy black</colorChoices>
  </product>
  <product dept="ACC">
     <number>563</number>
     <name language="en">Floppy Sun Hat</name>
  </product>
  <product dept="ACC">
    <number>443</number>
    <name language="en">Deluxe Travel Bag</name>
  </product>
  <product dept="MEN">
     <number>784</number>
     <name language="en">Cotton Dress Shirt</name>
      <colorChoices>white gray</colorChoices>
      <desc>Our <i>favorite</i> shirt!</desc>
  </product>
```

Expresiones XPath y contexto

Una expresión de ruta siempre se evalúa en relación con un elemento de contexto particular, que sirve como punto de partida para la ruta relativa. Algunas expresiones de ruta comienzan con un paso que establece el elemento de contexto, como en

```
doc("catalog.xml")/catalog/product/number
```

La función llamada doc("catalog.xml") devuelve el nodo documento del documento catalog.xml, que se convierte en el item de contexto. Cuando el item de contexto es un nodo (como opuesto a un valor atómico), se llama nodo de contexto. El resto de la ruta se evalúa relativo a eso. Otro ejemplo es:

```
$catalog/product/number
```

donde el valor de la variable $catalog establece el contexto. La variable debe seleccionar cero, uno o más nodos, que se convierten en los nodos de contexto para el resto de la expresión.

Las rutas en XPath pueden ser:

1. **Absolutas**, empiezan con barra y hacen referencia al nodo raiz. Ejemplo: /bookstore/book, en el ejemplo *bookstore* tiene que ser el nodo raíz,tal como se muestra, de otra forma no podría ser encontrado.
2. **Relativas** la expresión de ruta se evaluará en relación con el nodo de contexto actual, que debe haberse determinado previamente fuera de la expresión. Puede haber sido establecido por el procesador fuera del alcance de la consulta, o en una expresión externa. Ejemplo: book/title, en el ejemplo partimos del nodo contexto en este caso *bookstore* que tendría un nodo hijo llamado *book* y *book* un nodo hijo llamado *title* para que la selección nos devolviera un valor.

Pasos y contexto cambiado

El item de contexto cambia con cada paso. Un paso devuelve una secuencia de cero, uno o más nodos que sirven como items de contexto para evaluar el próximo paso. Por ejemplo, en:
doc ("catalog.xml") / catalog / product / number
el paso **doc("catalog.xml")** devuelve un nodo documento que sirve como item contexto al evaluar el paso **catalog**. El paso del **catalog** se evalúa utilizando el nodo documento como nodo de contexto actual, devolviendo una secuencia de un elemento **catalog** hijo del nodo documento. Este elemento **catalog** sirve entonces como el nodo de contexto para la evaluación del paso **product**, que devuelve la secuencia de hijos **product** de los elementos **catalog** obtenidos previamente.
El último paso, **number**, se evalúa a su vez para cada hijo producto en esta secuencia.
Durante este proceso, el procesador realiza un seguimiento de tres cosas:
• El nodo de contexto en sí, por ejemplo, el elemento **product** que está actualmente siendo procesado
• La secuencia de contexto, que es la secuencia de elementos que se están procesando actualmente, por ejemplo, todos los elementos **product**
• La posición del nodo de contexto dentro de la secuencia de contexto, que puede ser utilizado para recuperar nodos según su posición

Pasos

Ejemplos de pasos en una ruta pueden ser llamadas a funciones (doc ("catalog.xml")),
y referencias de variables ($catalog).
Cualquier expresión que devuelve nodos puede estar en el lado izquierdo del operador de barra inclinada "/".
Otro tipo de paso es el *paso eje*, que permite navegar por la jerarquía de nodos XML.
Cada paso tiene un *eje*, que define la dirección y la relación de los nodos seleccionados Por ejemplo, se puede usar el eje **child ::** (un eje hacia adelante) para indicar que solo deben seleccionarse nodos hijos, mientras que el eje **parent ::** (a eje inverso) se puede usar para indicar que solo se debe seleccionar el nodo padre.
8 de los 12 ejes que existen se enumeran en la Tabla 4-2.

Ejes	Eje abreviado	significado
self::		El nodo de contexto en si mismo. Selecciona del nodo actual.
child::		Hijos del nodo de contexto. Los atributos no se consideran hijos de un elemento. Este es el eje predeterminado si no se especifica ninguno.
descendent::		Todos los descendientes del nodo de contexto (hijos, hijos de hijos, etc.). Los atributos no son descendientes considerados.
descendent-or-self::	//	El nodo contexto y sus descendientes.
attribute::		Selecciona atributes. Atributos del nodo de contexto.(si hay alguno)
parent::	..	El padre del nodo de contexto (si hay alguno). Este es el elemento o el nodo del documento que lo contiene El padre de un atributo es su elemento, aunque no se lo considera hijo de ese elemento.
ancestor::		Todos los ancestros del nodo de contexto. (padre, padre del padre,

		etc.)
ancestor-or-self::		El nodo de contexto y todos sus ancestros.

Además de tener un eje, cada paso eje tiene un *nodo test*. El nodo test indica cuál de los nodos (por nombre o clase o tipo de nodo) seleccionar, a lo largo del eje especificado. Por ejemplo, **child :: product** solamente selecciona todos los elementos product que son hijos del nodo de contexto corriente.

Los *comodines XPath* se pueden usar para seleccionar nodos XML desconocidos.
* machea con cualquier nodo elemento.
@* machea con cualquier nodo atributo
node() Coincide con cualquier clase o tipo de nodo

child::* selecciona todos los nodos elemento hijo del nodo de contexto corriente.
attribute::* selecciona todos los nodos atributo hijo del nodo de contexto corriente
ancestor :: ***** selecciona todos los elementos ancestros del nodo de contexto corriente
child::node() selecciona todos los hijos del nodo de contexto corriente
ancestors::node() selecciona todos los nodos ancestros del nodo de contexto corriente
attribute::node() selecciona todos los nodos atributo del nodo de contexto corriente (es lo mismo que @*)
También hay cuatro clases de nodo test adicionales: text( ), comment( ), processing-instruction( ), and document-node() - ver página 281 cap 21 del libro de xquery

Otras expresiones que son pasos

Además de los pasos eje, otras expresiones también se pueden usar como pasos.
Por ejemplo:
doc ("catalog.xml") / catalog / product / (number | name)
que usa la expresión entre paréntesis (número | nombre) para seleccionar todos los elementos number y elementos name. El | el operador es un operador de unión; selecciona la unión de dos conjuntos de nodos, más expresiones como pasos pag 60 del libro de xquery.

Predicados

Los predicados se utilizan en una expresión de ruta para filtrar los resultados y contener solo nodos
que cumplen con criterios específicos. Usando un predicado, puede, por ejemplo, seleccionar solo el
elementos que tienen un cierto valor para un atributo o elemento hijo.

Ejemplo 6:Todos los elementos de product que tienen un atributo dept cuyo valor es ACC
product[@dept=”ACC”]
Resultado

```
<product dept="ACC">
    <number>563</number>
    <name language="en">Floppy Sun Hat</name>
</product>
<product dept="ACC">
    <number>443</number>
    <name language="en">Deluxe Travel Bag</name>
</product>
```

Ejemplo 7 Todos los elementos product que tienen un hijo name cuyo valor es igual a Floppy Sun Hat
product[name = "Floppy Sun Hat"]

Resultado

```
<product dept="ACC">
      <number>563</number>
      <name language="en">Floppy Sun Hat</name>
 </product>
```

Entonces un paso consiste de : un eje, un nodo test y cero o más predicados.

/libros/libro[titulos='El Zorro']/following-sibling::*/titulo

```
                  ↑                    ↑          ↑
            Predicado                 Eje       Filtro
Paso1 /      paso2            / paso3            / paso4
```

Ejemplo 8

En el ejemplo 8 tenemos una expresión de camino que consta de cuatro pasos. En el segundo paso tiene un predicado que se evalúa que los títulos seleccionados sea el zorro. En el tercer paso tenemos tanto un eje, following-siblings, con un filtro , *.

Expresiones FLWOR

La estructura básica de muchas (pero no todas) consultas es la expresión FLWOR. FLWOR (pronunciado "flor") significa "for, let, where, order by, return", las palabras clave usado en la expresión.
Los FLWOR, a diferencia de las expresiones de ruta, le permiten manipular, transformar y clasificar tus resultados. Clausula **for** tiene una variable cuyo valor es una secuencia y el for significa recorrer esa secuencia. Para filtrar elementos de esa secuencia se usa el **where** y el where usa una condición. El **return** sirve para construir el resultado para cada elemento de la secuencia que pasó el filtrado. **Order by** ordena los elementos de la secuencia.

El ejemplo 4 muestra un FLWOR simple que devuelve los nombres de todos productos en el departamento de ACC.

Ejemplo 4. Consulta FLWOR

```
for $prod in doc("catalog.xml")/catalog/product
where $prod/@dept = "ACC"
order by $prod/name
return $prod/name
```

Ejemplo 5 Sumando una cláusula let

```
for $product in doc("catalog.xml")/catalog/product
let $name := $product/name
where $product/@dept = "ACC"
order by $name
return $name
```

Results

```
<name language="en">Deluxe Travel Bag</name>
<name language="en">Floppy Sun Hat</name>
```

Como puede ver en el ejemplo 4:
Con la cláusula **for** establece una iteración a través de los nodos product, y el resto del FLWOR se evalúa una vez para cada uno de los cuatro productos. Cada vez, una variable $prod está vinculado a un elemento product diferente. Los signos de dólar $ están acostumbrados a indicar nombres de variables en XQuery. Con la cláusula **where** se selecciona elementos product donde el atribute dept es igual a "ACC". Esto tiene el mismo efecto como predicado ([@dept = "ACC"]) en una expresión de ruta. Con la cláusula **order by** ordena los resultados del los elementos name hijos del elemento product, algo que no es posible con expresiones de ruta. Con la cláusula **return** indica que se debe devolver, en este caso el nodo hijo name de cada uno de los dos elementos product que devuelve la cláusula where.
La cláusula **let** (la L en FLWOR) se utiliza para establecer el valor de una variable. A diferencia de una cláusula for, no configura una iteración.

El ejemplo 5 muestra un FLWOR que devuelve el mismo resultado que en el Ejemplo 4. La segunda línea es una cláusula **let** que asigna el nombre del producto del elemento hijo a una variable llamada $name. La variable $name es luego referenciada más adelante en el FLWOR, tanto en la cláusula order by como en la cláusula return.La cláusula let evita repetir la misma expresión varias veces.

Constructores

Constructores directos

¿Como podemos crear nuevos elementos enteramente y atributos y incluirlos en los resultados de una consulta xquery?

Se puede insertar sus propios elementos y atributos XML en los resultados de la consulta usando constructores XML. Hay dos tipos de constructores XML: constructores directo, que usan una sintaxis familiar similar a XML y constructores programados, que le permiten generar dinámicamente los nombres XML utilizados en los resultados.

Un constructor de elemento directo es un constructor del primera clase; especifica un elemento XML (opcionalmente con atributos) usando sintaxis similar a XML, como se muestra en el Ejemplo 7 se construye elementos html, hl, ul y li.

El resultado de la consulta es un fragmento XHTML que presenta los datos seleccionados.

Example 7.Construcción de elementos usando sintaxis como XML

```
<html>
   <h1>Product Catalog</h1>
   <ul>{
         for $prod in doc("catalog.xml")/catalog/product
         return <li>number: {data($prod/number)}, name: {data($prod/name)}</li>
   }</ul>
</html>
```

Results

```
<html>
   <h1>Product Catalog</h1>
     <ul>
       <li>number: 557, name: Fleece Pullover</li>
       <li>number: 563, name: Floppy Sun Hat</li>
       <li>number: 443, name: Deluxe Travel Bag</li>
      <li>number: 784, name: Cotton Dress Shirt</li>
    </ul>
</html>
```

Expresiones englobadas

Las expresiones englobadas son expresiones que están dentro de llaves pueden incluir más de una subexpresión dentro de llaves, usando comas como separadores. En el ejemplo 5-7, la expresión englobada. El constructor li contiene cuatro subexpresiones diferentes, separadas por comas.

Example 5-7. Expresión englobada con multiples sub expresiones.

```
for $prod in doc("catalog.xml")/catalog/product
return <li>{$prod/@dept,"string",5+3,$prod/number}</li>
```

Results

```
<li dept="WMN">string 8<number>557</number></li>
<li dept="ACC">string 8<number>563</number></li>
<li dept="ACC">string 8<number>443</number></li>
<li dept="MEN">string 8<number>784</number></li>
```

La primera subexpresión, $prod/@dept, se evalúa como un atributo, y por lo tanto se convierte en un atributo de li.

Las siguientes dos subexpresiones, "string" y 5 + 3, evalúan a sus valores atómicos: una cadena y un entero, respectivamente. Tenga en cuenta que están separados por un espacio en los resultados.

La subexpresión final, $prod/number, es un elemento que no está separado de los valores atómicos por un espacio en el resultado.
Observar que si no se usara una expresión englobada, o sea ponemos:
return <li> $prod/@dept**,**"string"**,**5+3**,**$prod/number
Entonces obtenemos:
Results
 $prod/@dept**,**"string"**,**5+3**,**$prod/number

Expresiones Cuantificadas

Una expresión cuantificada determina si algunos o todos los elementos en una secuencia cumplen con una condición particular. Por ejemplo, si desea saber si alguno de los los items en un orden tienen como atributo dept = “ACC”, puede usar la expresión mostrado en el ejemplo 9. Esta expresión devolverá verdadero.

Example 9.Expresión cuantificada usando la palabra clave **some**
some $dept **in** doc("catalog.xml")//product/@dept
satisfies ($dept = "ACC")

Alternativamente, si quiere saber si cada item en una orden tienen como atributo dept = “ACC”, simplemente puede cambiar la palabra un poco de **some** a **every**, como se muestra en Ejemplo 10. Esta expresión devolverá falso.

Example 10.Expresión cuantificada usando la palabra clave **every**
every $dept **in** doc("catalog.xml")//product/@dept
satisfies ($dept = "ACC")

Una expresión cuantificada siempre evalúa a un valor booleano (verdadero o falso). Como tal, no es útil para seleccionar los elementos o atributos que cumplen ciertos criterios, pero si para determinar si alguno existe. Las expresiones cuantificadas generalmente pueden se pueden volver a escribir fácilmente como FLWOR o incluso como simples expresiones de ruta. sin embargo, la expresión cuantificada puede ser más compacta y más fácil para optimizar implementaciones.

Una expresión cuantificada está compuesta de varias partes:
• Un cuantificador (la palabra clave **some** o **every**)
• Una o más cláusulas en las que unen variables a secuencias
• cláusula **satisfies** que contiene la expresión de prueba

El procesador prueba la expresión de **satisfies** (usando su valor booleano efectivo) para cada items en la secuencia. Si el cuantificador es **some**, devuelve verdadero si la expresión de satisfies es verdadera para cualquiera de los items. Si el cuantificador es every, devuelve verdadero solo si la expresión de satisfies es verdadera para todos los items. Si no hay elementos en la secuencia, una expresión con **some** siempre devuelve falso, mientras que una expresión con **every** devuelve verdadero.

Expresiones condicionales (if-then-else)

XQuery permite expresiones condicionales utilizando las palabras clave if, then, and else. La expresión después de la palabra clave **if** se conoce como la **expresión de test**. Debe ser encerrado entre paréntesis. Si la expresión de test se evalúa como verdadera, el valor de toda expresión condicional es el valor de la expresión **then**. De lo contrario, es el valor de la expresión **else**. En el ejemplo 11: muestra una expresión condicional (incrustada en un FLWOR).

Example 11.consulta de la expresión condicional.

```
for $prod in (doc("catalog.xml")/catalog/product)
return if ($prod/@dept = 'ACC')
        then <accessoryNum>{data($prod/number)}</accessoryNum>
        else <otherNum>{data($prod/number)}</otherNum>
```

Resultado

```
<otherNum>557</otherNum>
<accessoryNum>563</accessoryNum>
<accessoryNum>443</accessoryNum>
<otherNum>784</otherNum>
```

Comparaciones generales

Las comparaciones generales se utilizan para comparar valores atómicos o nodos que contienen valores atómicos. La Tabla 1 muestra algunos ejemplos de comparaciones generales. Usan el operadores = (igual a), ! = (no igual a), < (menor que), <= (menor o igual que), > (mayor que), y > = (mayor que o igual a).

Tabla 1: comparaciones generales

Ejemplo	Valor
doc("catalog.xml")/catalog/product[2]/name = 'Floppy Sun Hat'	true
doc("catalog.xml")/catalog/product[4]/number < 500	false
1 > 2	false
( ) = (1, 2)	false
(2, 5) > (1, 3)	true
(1, "a") = (2, "b")	Type error

Si cualquiera de los operandos es la secuencia vacía, la expresión se evalúa como falsa.

Comparaciones generales en secuencias de múltiples elementos

Las comparaciones generales pueden operar en secuencias de más de un elemento, así como secuencias vacías. Si uno o ambos operandos es una secuencia de más de un item, la expresión se evalúa como verdadera si la comparación del valor correspondiente es verdadera para cualquier combinación de dos elementos de las dos secuencias. Por ejemplo, la expresión (2, 5) <(1, 3) devuelve verdadero si una o más de las siguientes condiciones es verdadera:

- 2 es menor que 1
- 2 es menor que 3
- 5 es menor que 1
- 5 es menor que 3

Este ejemplo devuelve verdadero porque 2 es menor que 3. La expresión (2, 5)> (1, 3) también devuelve verdadero porque hay valores en la primera secuencia que son mayores que valores en la segunda secuencia.

Las comparaciones generales son útiles para determinar si algún valor en una secuencia cumple una
criterio particular. Por ejemplo, si desea determinar si alguno de los productos están en el departamento ACC, puede usar la expresión:

```
doc ("catalog.xml") / catalog / product / @ dept = 'ACC'
```

Esta expresión es verdadera **si al menos uno** de los cuatro atributos de departamento es igual a ACC.

Comparaciones generales y tipos

Al comparar dos valores, se tienen en cuenta sus tipos. Valores de tipos similares (por ejemplo, ambos numéricos, o ambos derivados del mismo tipo primitivo) siempre pueden ser probado para la igualdad usando los operadores = y! =. El procesador puede generar un error de tipo si los dos operandos contienen un valores incomparable, como se muestra en la última fila de la Tabla 1.

Al comparar dos valores atómicos en cada operando, si se escribe un valor, y el otro está sin tipo, el valor sin tipo se convierte al tipo del otro valor (o a xs:doble si el tipo específico es numérico). Por ejemplo, puede comparar el valor sin tipo de un elemento number con xs: entero 500, siempre que el contenido del elemento number puede ser convertido a xs: double. Si ambos operandos están sin tipo, se comparan como strings.

Expresiones aritméticas

Las expresiones aritméticas son valores numéricos . cuando se realizo una operacion aritmetica sobre dos valores que son del mismo tipo el resultado es el del mismo tipo, si tiene distintos tipo se tiene que convertir uno al otro usando **cast**. Tenemos los operadores +, - , *, div, mod.

Para mas información ver [15].

2.2 Esquema XML

En esta sección se asume conocimiento de XML y de espacios de nombres.

Los XML Schemas son una notación usada para describir cómo debe estar organizado el contenido dentro de un documento XML.

Cuando se define un XML Schema la sintaxis es XML.
Un documento XML que adhiere a un esquema XML particular se llama documento instancia de esquema XML.

<xs:schema>

El elemento raíz dentro de un XML schema es un elemento <xs:schema>.
Los esquemas XML pueden declarar vocabularios, los cuales pueden ser identificados por el espacio de nombres que es especificado usando el atributo targetNameSpace.
Ejemplo 1:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.wrox.com/name"
 xmlns="http://www.wrox.com/name">
```

Las *declaraciones globales* son declaraciones que aparecen como hijos directos del elemento <xs:schema>. Las declaraciones globales van a reutilizarse a lo largo del XML Schema. Las *declaraciones locales* no tienen al elemento <xs:schema> como su padre directo y se usan en un contexto específico.

<xs:element>

Para declarar elementos se usa <xs:element>
La declaración de un elemento puede ser local o global.
Cuando se declara un elemento se realizan dos tareas: se especifica el nombre del elemento y se define el contenido permitido que es determinado por su tipo.
Ejemplo 2:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="first" type="xs:string"/>
```

Además el tipo de un elemento puede ser un tipo simple (para esto se usa <xs:simpleType>) o un tipo complejo (para esto se usa <xs:complexType>).
Ese tipo puede ser localmente declarado (i.e. se pone como hijo del elemento <xs:element>) o globalmente declarado (en cuyo caso va a ser reutilizado junto con la declaración del elemento).

Se pueden *reutilizar elementos* refiriéndose a un elemento global existente. En este caso se incluye una referencia a la declaración del elemento global. En ese caso se usa el atributo ref.

Ejemplo 3: para reutilizar el elemento del ejemplo 2 usamos:

```
<element ref=”first”/>
```

Cardinalidad representa el número de ocurrencias de un elemento específico dentro de un modelo de contenido. Para expresar la cardinalidad podemos usar los atributos minOccurs y maxOccurs dentro de la declaración del elemento. Estos atributos no se permiten en declaraciones de elementos globales.
Ejemplo 4:

```
<element ref="element2" maxOccurs="10"/>
<element name="element3" type="string" minOccurs="0“ maxOccurs="unbounded"/>
```

El valor por default para minOccurs es 1 y el valor por default para maxOccurs es 1 (en ese caso no hace falta poner los atributos minOccurs y maxOccurs).

<xs: attribute>

Para declarar un atributo se usa <xs:attribute>
La declaración de un atributo puede ser local o global.
Cuando se declara un atributo se realizan dos tareas: se especifica el nombre del atributo y se define el contenido permitido que es determinado por su tipo.
Ejemplo 5: <xs:attribute name="title" type="xs:string"/>

Además el tipo de un atributo puede ser un tipo simple (para esto se usa <xs:simpleType>). Ese tipo puede ser localmente declarado (i.e. se pone como hijo del elemento <xs:attribute>) o globalmente declarado (en cuyo caso va a ser reutilizado junto con la declaración del atributo).
Se pueden *reutilizar atributos* refiriéndose a un atributo global existente. En este caso se incluye una referencia a la declaración del atributo global. En ese caso se usa el atributo ref.
Ejemplo 6: Si la declaración del ejemplo 5 es global, puedo reutilizar ese atributo mediante:
<attribute ref="title"/>

Cuando se declara un atributo se puede especificar que es requerido, opcional (i.e. el atributo puede aparecer o no en un documento instancia), o prohibido (i.e. el atributo no se puede usar en un documento instancia); para indicar esto usar el atributo use en la declaración de un atributo local.
Ejemplo 7: <xs:attribute name="lang" type="xs:string" use="optional"/>

Se pueden declarar valores default para atributos.
Ejemplo 8: <attribute name="title"type="string"default="Mr."/>

<xs:sequence>

El elemento <xs:sequence> especifica que los elementos hijo deben aparecer en secuencia (o sea en orden). Puede mantener sus atributos minOccurs y maxOccurs para definir una cantidad de ocurrencias del total de la secuencia.
Ejemplo 9:

```
<xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
    <xs:element name="address" type="xs:string"/>
    <xs:element name="city" type="xs:string"/>
    <xs:element name="country" type="xs:string"/>
</xs:sequence>
```

Ejemplo 10:

```
<xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element name="dog" type="xs:string"/>
    <xs:element name="cat" type="xs:string"/>
  </xs:sequence>
```

<xs: choice>

El element <xs:choice> permite solo uno de los elementos contenidos en la declaración <xs:choice> estar presente en el elemento que lo contiene.
Ejemplo 11:

```
<xs:choice>
    <xs:element name="employee" type="employee"/>
```

```
    <xs:element name="member" type="member"/>
</xs:choice>
```

La cantidad de apariciones del choice en sí mismo está controlado por sus atributos minOccurs y maxOccurs, mientras que el número de ocurrencias de cada elemento dentro de una simple ocurrencia de *xs: choice* se puede controlar mediante los atributos *minOccurs* y *maxOccurs* de los elementos del choice.

<xs:restriction>

Un tipo derivado declarado usando la declaración <xs:restriction> es un subconjunto de su tipo base. Una faceta es una propiedad simple de un tipo. Restringiendo las facetas de tipos existentes podemos crear nuestros tipos propios más restrictivos. Los tipos pueden ser restringidos por más de una faceta.
Ejemplos de facetas son: minExclusive (el menor valor para el tipo el cual excluye el valor que uno especifica), minInclusive (el menor valor para el tipo el cual incluye al valor que uno especifica), maxExclusive (el mayor valor para el tipo el cual excluye el valor que uno especifica), maxInclusive (el mayor valor para el tipo el cual incluye el valor que uno especifica), totalDigits (el número total de dígitos en un tipo numérico), fractionDigits (el número de dígitos fraccionarios en un tipo numérico), lenght (número de ítems en un tipo lista o número de caracteres en un tipo string), minLength (número mínimo de ítems en un tipo lista o mínimo número de caracteres en un tipo string), maxLenght (número máximo de ítems en un tipo lista o máximo número de caracteres en un tipo string), Enumeration (permite especificar un valor permitido en una lista enumerada), Pattern (permite restringir tipos string usando expresiones regulares)

Ejemplo 12:

```
<restrictionbase="nonNegativeInteger">
<maxInclusivevalue="59"/>
</restriction>
```

Ejemplo 13:

```
<xs:restriction base="xs:string">
  <xs:pattern value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>
</xs:restriction>
```

Ejemplo 14:

```
 <xs:restriction base="xs:string">
    <xs:minLength value="5"/>
    <xs:maxLength value="8"/>
  </xs:restriction>
```

<xs:list>

El elemento <xs:list> define un elemento como una lista de valores de un tipo de datos especificado.

Ejemplo 15: <xs:list itemType="xs:integer"/>

<xs:simpletype>

Los esquemas XML permiten especificar el tipo de datos textuales permitidos dentro de atributos y elementos usando declaraciones de tipos simples.
Un elemento tipo simple especifica las restricciones y la información acerca de los valores de los atributos y los elementos de solo texto.

Dentro de un tipo simple se puede usar <xs:restriction>, <xs:list> o <xs:unión>.
Las declaraciones de tipos simples pueden ser locales o globales.
Ejemplo 16: el elemento age tiene un tipo simple cuyo valor no puede ser menor a 0 ni mayor a 100. Este es un ejemplo de declaración local.

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Se pueden reutilizar tipos simples dentro de definiciones de elementos. Para esto se usa el atributo type dentro de <element>.
Ejemplo 17: Este es ejemplo de tipo simple declarado globalmente y usado.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="stringvalues" type="valuelist"/>
<xs:simpleType name="valuelist">
  <xs:list itemType="xs:string"/>
</xs:simpleType>
</xs:schema>
```

<xs:unión>

El elemento union define un tipo como una colección que es una unión de valores de tipos de datos especificados.

Ejemplo 17:

```
<xs:element name="jeans_size">
  <xs:simpleType>
    <xs:union memberTypes="sizebyno sizebystring" />
  </xs:simpleType>
</xs:element>

<xs:simpleType name="sizebyno">
  <xs:restriction base="xs:positiveInteger">
    <xs:maxInclusive value="42"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="sizebystring">
  <xs:restriction base="xs:string">
    <xs:enumeration value="small"/>
    <xs:enumeration value="medium"/>
    <xs:enumeration value="large"/>
  </xs:restriction>
</xs:simpleType>
```

<xs:complextype>

El element complexType permite definir un tipo complejo; un tipo complejo contiene otros elementos y/o atributos.
Dentro de complextype se puede usar <xs:sequence>, <xs:choice>, <xs:group>, <xs:attributeGroup>

Las declaraciones de tipos complejos pueden ser locales o globales.
Ejemplo18: la siguiente es una declaración local

```
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
   <xs:element name="to" type="xs:string"/>
   <xs:element name="from" type="xs:string"/>
   <xs:element name="heading" type="xs:string"/>
   <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Ejemplo 19: la siguiente es una declaración global.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
<elementname="first" type="string"/>
<elementname="middle" type="string"/>
<elementname="last" type="string"/>
<complexTypename="NameType">
<sequence>
<elementref="first"/>
<elementref="middle"/>
<elementref="last"/>
</sequence>
<attributename="title" type="string"/>
</complexType>
<elementname="name" type="NameType"/>
</schema>
```

<xs:group>

Se pueden definer grupos reutilizables de elementos creando una declaración <xs: group> global. Todas las declaraciones group globales deben tener un nombre asociada al atributo name.
Dentro de un elemento group se puede usar <xs:sequence> y <xs:choice>
Para reutilizar un grupo se usa el elemento group con atributo ref igual al nombre del grupo definido globalmente.

Ejemplo 20:

```
<group name="NameGroup">
<sequence>
<element name="first" type="string"/>
<element name="middle" type="string"/>
<element name="last" type="string"/>
</sequence>
</group>
<element name="name">
<complexType>
<group ref="NameGroup"/>
<attribute name="title" type="string"/>
</complexType>
</element>
```

<xs: attributeGroup>

Se pueden definir grupos de atributos globales para poder usar el mismo conjunto de atributos para varios elementos. Todas las declaraciones globales attributeGroup deben tener un nombre asociado al atributo name.
Dentro de un elemento attributeGroup se pueden usar elementos attribute y attributeGroup.
Para reutilizar un attributeGroup se usa el elemento attributeGroup con atributo ref igual al nombre del grupo de atributos definido globalmente.

Ejemplo 21:

```
<xs:attributeGroup name="personattr">
  <xs:attribute name="complete name" type="string"/>
  <xs:attribute name="age" type="integer"/>
</xs:attributeGroup>

<xs:complexType name="person">
  <xs:attributeGroup ref="personattr"/>
</xs:complexType>
```

<xs:extension>

El elemento extensión extiende un elemento simpleType o complexType. Para indicar el tipo extendido se usa el atributo base.

Dentro de un elemento extensión se puede usar: choice, sequence, group, attibute, attributeGroup

Ejemplo 22:

```
<xs:simpleType name="size">
  <xs:restriction base="xs:string">
    <xs:enumeration value="small" />
    <xs:enumeration value="medium" />
    <xs:enumeration value="large" />
  </xs:restriction>
</xs:simpleType>
…
<xs:extension base="size">
     <xs:attribute name="sex">
       <xs:simpleType>
         <xs:restriction base="xs:string">
           <xs:enumeration value="male" />
           <xs:enumeration value="female" />
         </xs:restriction>
       </xs:simpleType>
     </xs:attribute>
   </xs:extension>
```

<xs: complexContent>

Un elemento complexContent define extensiones o restricciones en un complexType que contiene elementos solamente. Por lo tanto dentro de un complexContent debe haber uno de los siguientes elementos: restriction, extensión.
Ejemplo 23:

```
<xs:element name="employee" type="fullpersoninfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:sequence>
        <xs:element name="address" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

<xs:simpleContent>

El element simpleContent contiene extensiones o restricciones en un complexType de solo texto o en un simpleType como contenido y no contiene elementos. Por lo tanto dentro de un complexContent debe haber uno de los siguientes elementos: restriction, extension.
Ejemplo 24:

```
<xs:element name="shoesize">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="country" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

El siguiente es un ejemplo de contenido que respeta esta definicion:

```
<shoesize country="france">35</shoesize>
```

Para mas información ver [14].

3. Mapeo de consultas y esquemas SQL a esquemas XML

3.1 Gramática de SQL

(Ver [13])

```
sql_list:   sql ';'
   |   sql_list sql ';'
   ;
```

Iniciamos con la no terminal **sql_list**, una lista de consultas sql, cada una de las cuales es un tipo de declaración. Hemos puesto la no terminal **sql** al comienzo de cada una de las secciones de la gramática.

Definición de Tablas en SQL (ver [13])

```
sql:   create_table
   |  create_domain
   ;

create_table:     CREATE TABLE table '(' base_table_element_commalist ')'  ;
```

Una definición de tabla base es **create_table**. Con **table,** el nombre de la tabla que puede ser un simple nombre o un nombre calificado por un nombre de usuario, y con **base_table_element_commalist** una lista de elementos de la tabla base entre paréntesis.

```
table: NAME
   |   NAME '.' NAME
   ;

column_ref:  NAME
   |   NAME '.' NAME    /* needs semantics */
   |   NAME '.' NAME '.' NAME
   ;

base_table_element_commalist:  base_table_element
    |    base_table_element_commalist ',' base_table_element
         ;

base_table_element:   column_def
    |     table_constraint_def
         ;
column_def:     column data_type column_def_opt_list;
column:      NAME ;
```

Cada elemento de tabla base con la no terminal **base_table_element** es una definición columna o una definición de restricción de tabla.
Una definición de columna **column_def** es un nombre de columna, su tipo de datos y un lista opcional de definiciones opcionales de columna.

```
data_type: CHARACTER
   |   CHARACTER '(' INTNUM ')'
   |   NUMERIC
   |   NUMERIC '(' INTNUM ')'
   |   NUMERIC '(' INTNUM ',' INTNUM ')'
   |   DECIMAL
   |   DECIMAL '(' INTNUM ')'
```

```
   |   DECIMAL '(' INTNUM ',' INTNUM ')'
   |   INTEGER
   |   SMALLINT
   |   FLOAT
   |   FLOAT '(' INTNUM ')'
   |   REAL
   |   DOUBLE PRECISION
   |  domain_name
   ;
column_def_opt_list: /* empty */
   |   column_def_opt_list column_def_opt
   ;
column_def_opt:  NOT NULLX
   |   NOT NULLX UNIQUE
   |   NOT NULLX PRIMARY KEY
   |   PRIMARY KEY
   |   DEFAULT literal
   |   DEFAULT NULLX
   |   DEFAULT USER
   |   CHECK '(' search_condition ')'
   |   REFERENCES table
   |   REFERENCES table '(' column_commalist ')'
   ;

table_constraint_def: UNIQUE '(' column_commalist ')'
   |   PRIMARY KEY '(' column_commalist ')'
   |   FOREIGN KEY '(' column_commalist ')'
         REFERENCES table
   |   FOREIGN KEY '(' column_commalist ')'
         REFERENCES table '(' column_commalist ')'
   |   CHECK '(' search_condition ')'
   ;

column_commalist: column
   |   column_commalist ',' column
   ;

literal: STRING
   |   INTNUM
   |   APPROXNUM
   ;
```

Hay largas listas de posibles tipos de datos, **data_type** y de opciones de definición de columna, **column_def_opt_list**.

Create_domain

```
create_domain:  CREATE DOMAIN domain_name data_type constraint
domain_name: NAME ;
constraint :    /* empty */
   |   CONSTRAINT constraint_name column_def_opt
   |  constraint ',' CONSTRAINT constraint_name column_def_opt
   ;

constraint_name: NAME ;
```

Expresiones select-from-where (ver [12])

```
sql: select_stmt ;
select_stmt: SELECT select_opts select_expr_list
        |  SELECT select_opts select_expr_list
                  FROM table_references
                    opt_where opt_groupby opt_having opt_orderby opt_limit
                    opt_into_list

opt_where: /* empty */
           | WHERE expr ;

opt_having: /* empty */ | HAVING expr ;
```

Cláusula **order by** y **group by**
De manera predeterminada la cláusula **order by** lista los elementos en orden ascendente. Para especificar el tipo de ordenación se puede incluir la cláusula **desc** para orden descendente o **asc** para orden ascendente. Además, se puede ordenar con respecto a más de un atributo.

```
opt_orderby: /* empty */ | ORDER BY groupby_list ;
groupby_list: expr opt_asc_desc
              | groupby_list ',' expr opt_asc_desc
              ;
opt_groupby: /* empty */
             | GROUP BY groupby_list
             ;

opt_asc_desc: /* empty */
              | ASC
              | DESC
         ;

opt_limit: /* empty */ | LIMIT expr
          | LIMIT expr ',' expr
          ;

opt_into_list: /* empty */
  | INTO column_list
  ;

column_list: NAME
  | column_list ',' NAME
  ;
```

Opciones Select y cláusula **distinct**
En aquellos casos donde se quiera forzar la eliminación de duplicados, se insertará la palabra clave **distinct** después de **select**.

```
select_expr_list: select_expr
    | select_expr_list ',' select_expr
    | '*'
    ;
```

select_opts: /* empty */ | select_opts DISTINCT ;

Cláusula as

La operación renombramiento *nombre-antiguo* **as** *nombre-nuevo.* Al escribir expresiones la **cláusula as** puede aparecer tanto en **select** como en **from**.

select_expr: expr opt_as_alias ;

opt_as_alias: AS NAME
  | NAME
  | /* empty */
  ;

SELECT table_references (ver [12])

La parte más compleja y poderosa de SELECT, y la parte más poderosa de SQL, es la forma en que puede referirse a múltiples tablas. En un SELECT, puede decirle que cree conceptual
tablas unidas creadas a partir de datos almacenados en muchas tablas reales, ya sea por uniones explícitas o por instrucciones SELECT recursivas. Como las tablas pueden ser bastante grandes, también hay formas pensar sobre cómo hacer la unión de manera eficiente.

table_references: table_reference
    | table_references ',' table_reference
    ;

table_reference: table_factor
  | join_table
;

Una tabla no es tan compleja, consiste principalmente en listas de items y una gran cantidad de cláusulas opcionales. Cada **table_reference** puede ser un **table_factor** (cual es una tabla, un anidamiento SELECT, o una lista entre paréntesis),o sino **join_table**, un join explícito. Una referencia a tabla es el nombre de la tabla con o sin el nombre de la base de datos que lo contiene, una cláusula opcional AS que da el nombre alias.

table_factor:NAME
  | table_subquery opt_as NAME
  | '(' table_references ')'
  ;

table_subquery: '(' select_stmt ')';

Tenga en cuenta las reglas separadas **table_factor** y **table_reference**. Están separados para establecer la asociatividad de los operadores JOIN y resolver la ambigüedad en una expresión como **a JOIN b JOIN c**, lo que significa **(a JOIN b) JOIN c** en lugar de **a JOIN (b JOIN c)**. En la regla **join_table**, hay un **table_references** en el lado izquierdo de cada join y **table_factor** a la derecha, haciendo que la sintaxis sea asociativa.

opt_as: AS | /* empty */ ;

Un SELECT anidado son sentencias SELECT entre paréntesis, que debe tener un nombre asignado, aunque el AS antes del nombre es opcional. Un **table_factor** también puede ser una lista **table_references** entre paréntesis, que puede ser útil al crear combinaciones.

Operaciones de reunión o combinación

Tenemos reunión Interna y Externa.
Tipos de reunión: inner join, left outer join y right outer join
Condiciones de reunión: natural, on <predicado>, using (A 1 , A 2 , ..., A n)

***Condiciones de reunión*: natural**

E1 natural [inner] join E2

join_table: table_reference NATURAL opt_inner JOIN table_factor

E1 natural {left [outer] | right [outer] } join E2

join_table: table_reference NATURAL opt_left_or_right_outer JOIN table_factor

Condiciones de reunión: on <predicado>

La cláusula ON se utiliza para unir tablas donde los nombres de las columnas no coinciden en ambas tablas. Las condiciones de unión se retiran de las condiciones de filtro en la cláusula WHERE

E1 [inner] join E2 **on** P

join_table :table_reference opt_inner JOIN table_factor ON expr

E1 {left [outer] | right [outer] } join E2 **on** P

join_table: table_reference left_or_right opt_outer JOIN table_factor ON expr

Condiciones de reunión: using (A 1 , A 2 , ..., A n)

La condición de reunión **using** (*A*1, *A*2, … , *An*) es similar a la condición de reunión natural, salvo en que los atributos de reunión en este caso son *A*1, *A*2,…,*An*, en lugar de todos los atributos comunes de ambas relaciones y aparecen sólo una vez en el resultado de la unión.

E1 [inner] join E2 **using (A1,A2,..,An)**

join_table:table_reference opt_inner JOIN table_factor USING '(' column_list ')'

E1 {left [outer] | right [outer] } join E2 **using (A1,A2,...,An)**

join_table: table_reference left_or_right opt_outer JOIN table_factor USING '(' column_list ')'

La producción join_table nos queda,

```
join_table:
    |   table_reference NATURAL opt_inner JOIN table_factor
    |   table_reference NATURAL opt_left_or_right_outer JOIN table_factor
    |  table_reference opt_inner JOIN table_factor
    |  table_reference opt_inner JOIN table_factor ON expr
    |  table_reference opt_inner JOIN table_factor USING '(' column_list ')'
    |  table_reference left_or_right opt_outer JOIN table_factor ON expr
    |  table_reference left_or_right opt_outer JOIN table_factor USING '(' column_list ')'
    ;

opt_inner: /* nil */ | INNER ;

opt_outer: /* nil */  | OUTER;

left_or_right: LEFT | RIGHT ;

opt_left_or_right_outer: LEFT opt_outer | RIGHT opt_outer | /* empty */ ;
```

Un join especifica la forma de combinar dos grupos de tablas. join vienen en una variedad de sabores que cambian el orden en que se combinan las tablas, especifica qué hacer con registros

en un grupo que no coinciden con ningún registro en el otro grupo, y especifica otros detalles. Cada unión también especifica explícita o implícitamente los campos a usar para hacer coincidir en las tablas, en una variedad de sintaxis.

En NATURAL join, join machea sobre campos con el mismo nombre, y en una join regular, si no hay campos en la lista, crea un producto cruzado, uniendo cada registro en el primer grupo con cada registro en el segundo grupo. En este último caso, el resultado suele ser reducido por una cláusula WHERE o HAVING. Para todos los diversos tipos de join, tenemos el operador JOIN con subcampos que describan el tipo exacto de unión.

El siguiente grupo de operadores usa listas de expresiones de longitud variable (llamadas listas de valores o **val_list**). Las reglas de bison para analizar las listas de longitud variable solo necesitan mantener un conteo de cantidad de expresiones que ha parseado, que conservamos como el valor de la lista del lado derecho del lado del símbolo, en este caso **val_list**. Una lista de elementos individuales tiene longitud 1, y en cada etapa, una lista de elementos múltiples tiene un elemento más que su sublista. Hay algunos constructores donde la lista de valores es opcional, por lo que una **opt_val_list** está vacía, con un recuento valor de cero,o **val_list** con un valor de conteo de lo que sea que tenga **val_list**.

Definición de Lista de Expresiones (ver [12])

```
sql : val_list

val_list: expr
  | expr ',' val_list
  ;
opt_val_list: /* empty */
  | val_list
  ;
```

Subconsultas anidadas

- Pertenencia a conjuntos **not in**, **in**

La conectiva **in** comprueba la pertenencia a un conjunto, donde el conjunto es la colección de valores resultado de una cláusula **select**. La conectiva **not in** comprueba la no pertenencia a un conjunto.

```
expr: expr IN '(' val_list ')'
  | expr NOT IN '(' val_list ')'
  | expr IN '(' select_stmt ')'
  | expr NOT IN '(' select_stmt ')'
  | EXISTS '(' select_stmt ')'
  ;
```

Una vez que tenemos las listas, podemos analizar los operadores IN y NOT IN que testea una expresión está o no en una lista de valores. Tenga en cuenta que el código emitido incluye el recuento de valores. Sql también tiene una forma variante donde los valores vienen desde sentencias SELECT.

like

```
expr: expr LIKE expr
    | expr NOT LIKE expr
    ;
```

funciones regulares (ver [12])

```
expr: NAME '(' opt_val_list ')' ;
```

Con la expresión anterior realizamos llamadas a funciones.

Funciones son sintáxis especial

```
expr: FCOUNT '(' '*' ')'
  | FCOUNT '(' expr ')'
  | MAX '(' expr ')'
  | MIN '(' expr ')'
  | SUM '(' expr ')'
  | AVG '(' expr ')'

expr: FSUBSTRING '(' val_list ')'
  | FSUBSTRING '(' expr FROM expr ')'
  | FSUBSTRING '(' expr FROM expr FOR expr ')'
  | FTRIM '(' val_list ')'
  ;

expr: FDATE_ADD '(' expr ',' interval_exp ')'
  |  FDATE_SUB '(' expr ',' interval_exp ')'
  ;

interval_exp: INTERVAL expr DAY_HOUR
  | INTERVAL expr DAY_MICROSECOND
  | INTERVAL expr DAY_MINUTE
  | INTERVAL expr DAY_SECOND
  | INTERVAL expr YEAR_MONTH
  | INTERVAL expr YEAR
  | INTERVAL expr HOUR_MICROSECOND
  | INTERVAL expr HOUR_MINUTE
  | INTERVAL expr HOUR_SECOND
```

Manejamos cinco funciones con sintaxis especial aquí, COUNT, SUBSTRING, TRIM, DATE_ADD,
y DATE_SUB. COUNT tiene una forma especial, COUNT (*), que se utiliza para contar eficazmente el número de registros devueltos por una instrucción SELECT, así como una forma normal que cuenta el número de valores diferentes de una expresión.
SUBSTRING es un operador de subcadena normal que toma el original cadena final, dónde comenzar y cuántos caracteres tomar. Puede usar la sintaxis de llamada regular o usar palabras reservadas FROM y FOR para delimitar los argumentos. Hay una regla para cada forma, todos generan código similar ya que son los mismos dos o tres argumentos.

Expresiones binarias y unarias (ver [12])

```
expr: expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| expr '%' expr
| expr MOD expr
```

```
expr: '(' expr ')'
   | NAME
   | USERVAR
   | NAME '.' NAME
   | STRING
   | APPROXNUM
   | BOOL
   | TIMESTAMP
   | INTNUM
   | DECIMALNUM
   | FLOATNUM
   | DOUBLENUM
   | REALNUM
   ;
```

Otras expresiones

```
expr: CURRENT_TIMESTAMP
      | CURRENT_DATE
      | CURRENT_TIME
      ;
```

3.2 Definición del mapeo de SQL a XML Schemas usando un attribute gramar

3.2.1 Mapeo de esquemas SQL

```
sql_list:   sql ';'.{sql_list.schema = sql.schema}
   |    sql_list sql ';'.{sql_list[1].schema = sql_list[2].schema + sql.schema}
   ;


sql:   create_table.{ sql.schema = create_table.schema }
   |  create_domain.{ sql.schema = create_domain.schema }
   ;


create_table:     CREATE TABLE table '(' base_table_element_commalist ')'
   { create_table.schema =
         "<xsd:element name = "'+ table.getString()  +"' >" +
      "<xsd:complexType>
         <sequence>" +
         base_table_element_commalist.schema +
        "</xsd:sequence>
       </xsd:complexType >
          </xsd:element >" ;
           schema_of_table(table.getstring(), create_table.schema);
   }
```

Recordar que manejamos un mapeo para guardar los esquemas asociados a los nombres de las tablas. Esto va a ser útil cuando tengamos que obtener esquemas XML asociados a consultas. Para asociar al nombre de una tabla su esquema XML usamos la función schema_of_table.

```
base_table_element_commalist:
       base_table_element.{ base_table_element_commalist.schema =
base_table_element.schema; }
   |   base_table_element_commalist ',' base_table_element.
          { base_table_element_commalist[1].schema =
         base_table_element_commalist[2].schema + base_table_element.schema; }
   ;


base_table_element:
              column_def. { base_table_element.schema = column_def.schema; }
    |     table_constraint_def.{ base_table_element.schema = table_constraint_def.schema; }
    ;
```

Ahora hay que procesar columnas; al describir una columna se da su tipo que en nuestro caso es un data_type de sql.

Los data_type de sql se pueden clasificar en dos grupos:

1. Aquellos que incluyen una restricción en su declaración (p.ej. varchar(n) restringe las cadenas a aquellas de longitud menor o igual que n, numeric(n,m) restrige números a aquellos de punto fijo con n dígitos de parte entera y m de parte decimal, etc.).
2. Aquellos que no incluyen restricción en su declaración (p.ej. Integer, real, etc.)

Los data_type de sql del segundo grupo se los traduce a un tipo básico perteneciente a xml_schema.
Los data_type de sql del primer grupo se los traduce a una restricción de un tipo base de xml schema usando la etiqueta <xsd:restriction>.
Por ejemplo CHARACTER '(' INTNUM ')' se traduce a xml_schema:

```
 <xsd:restriction base="xsd:string">
       <xsd:length value = "INTNUM"/>
</xsd:restriction>.
```

Se consideró la información provista por la tabla en tipos de datos de SQL a esquemas XML para hacer estas traducciones**.**

Aplicando lo que acabamos de decir, el procesamiento de una columna queda:

```
column_def := column data_type column_def_opt_list .
  { if data_type is (INTEGER,DECIMAL,DOUBLE,REAL,TIME)
       column_def.schema =
          "<xsd:element name = "'+ column.schema +"' 
           type ="+ "data_type.schema"; +"' >"
       + "</xsd:element>" ;
           else
      column_def.schema = "<xsd:element name = "'+ column.schema +"' >"
"<xsd:simpleType +"' >" + data_type.schema + </xsd:simpleType >"
+"</xsd:element>" ; }
```

Definición de data_type.schema

Los tipos de sql sin restricción en su declaración que se llaman igual que en xml schema. se traducen con el mismo nombre.

```
data_type:= DECIMAL . { data_type.schema="decimal";}
data_type:= INTEGER . { data_type.schema="integer";}
data_type:= DOUBLE . { data_type.schema="double";}
data_type:= REAL . { data_type.schema="real";}
data_type:= TIME . { data_type.schema="time";}
```

Ahora se explica mapeo para tipos de sql que tienen restricciones en su declaración.

```
data_type:= FLOAT '(' INTNUM ')' .
   { data_type.schema=
          "<xsd:restriction base = float >
         <maxLength value = "' + INTNUM.getString + "'> "' +
     "</xsd:restriction>" ;
   }
data_type:= DECIMAL '(' INTMUM ')' .
   { data_type.schema=
       "<xsd:restriction base = decimal >
      <xsd:maxLength value = "' + INTNUM.getString + "'/> "' +
       "</xsd:restriction>";}
data_type:= CHARACTER . {data_type.schema=
                "<xsd:restriction base = "String' >
      <xsd:length value = '1'> " +
         "</xsd:restriction>" ; }

data_type := CHARACTER '(' INTNUM ')' .
    {data_type.schema=
```

```
   “<xsd:restriction base = ”String’ >
          <xsd:length value = "' + INTNUM.getString + "' /> " +
   “</xsd:restriction>" ;}

data_type := VARCHAR '(' INTNUM ')' .
    { data_type.schema=
   “<xsd:restriction base = ”String’ >
           <xsd:maxLength value = "' + INTNUM.getString + "' /> " +
   “</xsd:restriction>" ;}

data_type := NUMERIC '(' INTNUM ',' INTNUM ')' .
      { data_type.schema=
          “<xsd:restriction base=”decimal’>
                +   <xsd:precision value="' + INTNUM.getString + INTNUM.getString "' /> "
                + <xsd:scale value="' + INTNUM.getString +  "' /> "
            + </xsd:restriction> “;;}

data_type:= DATE . { data_type.schema =
                              "<xs:restriction base="xs:date">
                                 <xs:pattern value="\p{Nd}{2}-\p{Nd}{2}-\p{Nd}{4}"/>
                               </xs:restriction>" ; }
```

Ahora ilustramos el mapeo anterior por medio de un ejemplo.
Ejemplo 1: mapeo de tabla de empleados:

CREATE TABLE Empleados (id INTEGER, nombre CHARACTER(7), apellido DECIMAL);
El resultado que da bison se lo guarda en **create_table.xml**:

```
<schema>
<xsd:element name="Empleados">
    <xsd:complexType>
       <xsd:sequence>
       <xsd:element name = "id" type = "integer”/>
<xsd:element name = "nombre">
<xsd:simpleType >
<xsd:restriction base ="String">
    <xsd:length value = "7">
    </xsd:restriction>
</xsd:simpleType >
<xsd:element>
    <xsd:element name = "apellido" type = "decimal”/>
    <xsd:/sequence>
    <xsd:/complexType>
<xsd:/element >
</schema>
```

3.2.2 Mapeo de consultas select-from-where

Al definir el attribute grammar para consultas select-from-where vamos a hacer uso de XQuery para definir algunas funciones que se usan. Esto permite ser más compacto en la especificación que simplemente programar en un lenguaje de más bajo nivel como C.

```
select_stmt ::= SELECT select_opts select_expr_list
    FROM table_references
    opt_where opt_groupby opt_having opt_orderby opt_limit
    opt_into_list .
{ select_stmt.schema = obtain_schema( stringtolist (select_expr_list), table_references.schema) }
```

Para entender esta definición vamos a explicar primero los argumentos de la función. obtain_schema.

Stringtolist recibe un string que contiene varias “,” y devuelve una secuencia formada por las palabras separadas por las “.” . stringtolist se define en XQuery de la siguiente forma:

```
declare function local:stringtolist($palabra as xs:string?) as xs:string* {
    let $lista_string := tokenize ($palabra, ",")
    return $lista_string
  };
```

Por ejemplo: si tenemos el string "apellido, nombre, dni, telefono, sueldo", entonces: stringtolist("apellido, nombre, dni, telefono, sueldo") = ("apellido”, “nombre”, “dni”, “telefono”, “sueldo")

En el attribute grammar de select_stmt aplicamos stringtolist a los ítmes que acompañan un select (atributos, funciones de agregaciòn y expresiones). Estos se definen a nivel de gramática mediante:

```
select_expr_list: select_expr
 | select_expr_list ',' select_expr
 | '*
 ;
select_expr: expr opt_as_alias ;
```

El siguiente argumento de obtain_schema es el esquema asociado a una lista que contiene tablas y subconsultas anidadas en la parte from del select-from-where. En la gramática de sql esta lista se expresa mediante el no terminal table-references.

```
table_references:  table_reference . {table_references.schema = table_reference.schema}
    | table_references ',' table_reference.
            {table_references[1].schema = table_references[2].schema +
table_reference.schema};

table_reference: table_factor.{table_references.schema = table_factor.schema}
     | join_table {table_references.schema=join_table.schema }

table_factor= NAME { table_factor.schema= getTableSchema(NAME.getStgring()) }
 | table_subquery opt_as NAME .
      {  table_factor.schema=   “<xsd:element name = ”’+ NAME.getString()  +”’ >” +
                    table_subquery.schema
                        </xsd:element >” ;    }
 | '(' table_references ')'.{ table_factor.schema= table_references.schema   }
```

Recordar que tenemos un mapeo que asocia a cada tabla su esquema. Para obtener el esquema de la tabla de un cierto nombre se usa la funciòn getTableSchema.

table_subquery::= '(' select_stmt ')'. { table_subquery.schema = select_stmt.schema }

Ahora pasamos a especificar la función obtain_schema.
obtain_schema : String x xml schema de tablas del from → xml_schema del select

```
fun obtain_schema(s:array [1..N] of string, table_references.schema string) return string
{ int i
    for(i=0;i<N;i++)
    {
     if(s[i]==atribute) procesar_atributo(s[i], table.references.schema)
     if(s[i]==funcion_agregacion) procesar_funciónAgregación(s[i], table.references.schema)
     if(s[i]==expresion) procesar_expresion(s[i], table.references.schema )
    }
}
end
```

Obtatin schema procesa uno a uno los elementos de las lista de elementos del select. El resultado de procesar un elemento es un fragmento del XML schema hace referencia al nombre del elemento del select y de su tipo.

Hay 3 tipos de elemento a procesar:

1. atributo
2. función de agregación
3. expresión

Para cada tipo de elemento se define una función de procesamiento.

Caso procesamiento de atributo

El esquema de la parte del from consiste de una lista de esquemas de datos; cada uno de estos esquema de datos es una lista de nombres de campos y sus tipos; todo esto expresado en XML Schema.

La función procesar_atributo devuelve para un atributo: el nombre del atributo seguido del tipo que tiene ese atributo en la tabla del from correspondiente (se supone que tiene que haber una tabla con ese atributo porque se toma una consulta sql bien formada).

```
Declare function procesar_atributo($atributo as xs:string, $tableReferencesLista as xs:string?)
as xs:string* {
      let $aut1 :=   for $x in doc($tableReferencesLista)//element
                       where $x/@name=$atributo
                  return
                       if ((string($x/@type) = "integer") or (string($x/@type) = "decimal") or
                           (string($x/@type) = "double") or (string($x/@type) = "real") or
                           (string($x/@type) = "time") or (string($x/@type) = "date"))
                       then  <element>{  $x/@name,  $x/@type  }</element>
                       else  <element>{  $x/@name,  $x/simpleType  }</element>
  return (<schema> {$aut1} </schema>)
};
```

Ejemplo 2: Ilustramos la definición anterior por medio de un ejemplo:

```
let $selectOps := " nombre, apellido "
let $tableReferences := "create_table.xml"
    return (local:obtain_schema($selectOps, $tableReferences))
```

Aquí se tomó la tabla de empleados definida en el ejemplo 1 (del attribute grammar de create_table) y se la guardó en el archivo create_table.xml.
El resultado de esa consulta XQuery va a dar:

```
<schema>
 <element name="nombre">
  <simpleType>
   <restriction base="String">
      <length value="7"/>
   </restriction>
  </simpleType>
</element>
<element name="apellido" type="decimal"/>
</schema>
```

Caso funciones de agregación luego de lista de atributos
Procesamiento de funcion de agregación.

La función procesar_funciónAgregación va a retornar para una función de agregación *f(a)* un campo de nombre *f_a* y un tipo que se calcula de acuerdo a la función de agregación *f* y el tipo del atributo *a* en el esquema de la parte del from.

Si la función de agregación es *count*, el tipo es integer.
Si la función de agregación es *avg*, el tipo es el tipo asociado al atributo al cual se aplica avg.
Para las funciones *min*, *max* y *sum*, el tipo se calcula como en el caso *avg*.

```
Declare function procesar_funciónAgregación($funagreg as xs:string, $tableReferencesLista
as xs:string?) as xs:string* {
 let $aux :=    if ( matches( $funagreg, '[A-Za-z0-9_]') == true )   /* caso es un atribute */
                for $x in doc($tableReferencesLista)//element
                  where $x/@name=$funagreg
                  return
                       if((string($x/@type) = "integer") or (string($x/@type) = "decimal") or
                          (string($x/@type) = "double") or (string($x/@type) = "real") or
                          (string($x/@type) = "time") or (string($x/@type) = "date"))
                       then     <element>{    $x/@name, $x/@type   }</element>
                       else     <element>{    $x/@name, $x/simpleType   }</element>
                 else if ( contains( $funageg,"count") == true ) then
                     return  <element name = "?column?" type ="integer" />
    else if ( contains( $funagreg, "avg" ) == true or
              contains( $funagreg, "max" ) == true or
              contains( $funagreg, "min" ) == true or
              contains( $funagreg, “sum” ) == true )
       for $x in doc($tableReferencesLista)//element
       where $x/@name=$funagreg
       return     if((string($x/@type) = "integer") or (string($x/@type) = "decimal") or
                     (string($x/@type) = "double") or (string($x/@type) = "real") or
                     (string($x/@type) = "time") or (string($x/@type) = "date"))
                  then      <element>{   $x/@name, $x/@type   }</element>
                  else        <element>{   $x/@name,   $x/simpleType  }</element>
```

return (<schema> {$aux} </schema>)

Caso de expresión

Aquí procesar_expresión va a retornar para una expresión E y su nombre asociado (con as N) el esquema XML que se refiere a N y al tipo de E obtenido a partir del esquema XML de la parte del from.
Una expresión de select se puede pensar como un árbol binario, donde el padre son las operaciones y los hijos son los operandos.
Se puede generar el árbol de una expresión haciendo parsing top-down de la expresión.

Una idea para obtener el tipo de una expresión E es a partir del árbol de E inferir el tipo de E por medio de una función recursiva que recorre el árbol de E en orden in-order. Sin embargo está el problema saber cuáles son los tipos de los atributos y constantes de E; sin esta información no se puede inferir el tipo de E. El lugar más sencillo para calcular esos tipos es con attribute grammar de las expresiones. Por lo tanto el arbol de la expresion va a tener tipos como hojas.

```
expr: '(' expr ')'
  | NAME .{ expr.schema= procesar_atributo(NAME, tablereferences.schema)}
  | STRING.{expr.scehma="string"}
  | APPROXNUM.{expr.schema="float"}
  | BOOL.{expr.schema="boolean"}
  | TIMESTAMP.{expr.schema="dateTime"}
  | INTNUM.{expr.schema="integer"}
  | DECIMALNUM.{expr.schema="decimal"}
  | FLOATNUM.{expr.schema="float"}
  | DOUBLENUM.{expr.schema="double"}
  | REALNUM.{expr.schema="real"}
  ;
```

Como consecuencia de esto el árbol de una expresión E va tener como nodos nombres de operación y nombres de tipo.

fun procesar_expresion(elem: string, table_references:string) return string
arbolbinario:bintree;
tipo: string;
arbol_binario := parser_topdown (expresión:string, table_references:string);
tipo:= ObtenerTipo(árbol_binario);
return tipo;
end

Generar fragmento XML schema que tiene nombre(t) y tipo.

Vamos a ir desarrollando esta idea gradualmente.

TAD arbol binario

TAD bintree[elem]
constructores
<> : bintree
< _ , _ , _ > : bintree × elem × bintree → bintree
operaciones
root : bintree → elem {se aplica sólo a un árbol no vacío}

```
left : bintree → bintree                {se aplica sólo a un árbol no vacío}
right : bintree → bintree               {se aplica sólo a un árbol no vacío}
is_empty : bintree → bool
ecuaciones
root( < l,e,r > ) = e
left( < l,e,r > ) = l
right( < l,e,r > ) = r
is_empty( <> ) = true
is_empty( < l,e,r > ) = false
```

Pseudocódigo de TAD árbol binario

```
type tnode = tuple
                lft: pointer to tnode
                value: elem
                rgt: pointer to tnode
               end
type bintree = pointer to tnode

fun empty() return t:bintree
  t:=null
end
fun node(l:bintree,e:elem,r:bintree) return t:bintree
  alloc(t)
  t->lft:=l
  t->value:=e
  t->rgt:=r
end
fun root(t:bintree) return e:elem
   e:=t->value
end
fun left(t:bintree) return l:bintree
   e:=t->lft
end
fun right(t:bintree) return r:bintree
   e:=t->rgt
end
fun is_empty(t:bintree) return b:bool
  b:=(t==null)
end
proc destroy(in/out t:bintree)      {libera todo el espacio de memoria ocupado por t}
   if ¬is_empty(t) then destroy(left(t))
                        destroy(right(t))
                        free(t)
                        t:=null
  fi
end
fun Error(a1:bintree) return b:bool
   b:=(t->value==error)
end
```

Ejemplo de árbol binario de expresión

Dada la expresión:

(nota_teóricoPráctico * 0,6) + (nota_taller * 0,4)

El árbol binario de la misma es:

<< integer,*, decimal>,+,<integer,*, decimal>>

Definición de Parser Top-down

Una forma de implementar parser top-down non-backtracking llamado parser predictive. Un parser predictivo se caracteriza por la habilidad de elegir la producción a aplicar solamente sobre la base del siguiente símbolo de entrada y actual no terminal que se está procesando. Para habilitar esto, la gramática debe tener una particular forma.

La llamaremos LL(1). La primera "L" significa que escanea la entrada desde izquierda a derecha; la segunda "L" significa que creamos una derivación a izquierda; y el 1 significa un símbolo de entrada de lookahead. Informalmente, un LL(1) no tiene producciones recursivo a izquierda y tiene factorización a izquierda(left-factored).

Recursive Descent

La primera técnica para implementar un parse predictivo llamado recursive-descent. Un parser recursive-descent consiste de varias pequeñas funciones, una para cada terminal en la gramática. A medida que parseamos una sentencia, llamamos a las funciones que corresponda desde el lado izquierdo de la noterminal de la producción.Si estas producciones son recursivas, terminamos llamando a las funciones recursivamente.

Para poder aplicar parser top-down la gramática de las expresiones tiene que ser una gramática LL(1).

La gramática de expresiones original ha sido definida en la sección de gramática de SQL; leer las subsecciones tituladas: funciones regulares y expresiones unarias y binarias. Se transformó dicha gramática en una gramática LL(1) equivalente.

Gramática LL(1) de expresiones

Tenemos esta gramática de expresiones

```
expr: expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| expr '%' expr
| expr MOD expr
| '(' expr ')'
| "string"
| "decimal"
| "float"
| "double"
| "real"
| "integer"
| "dateTime"
| "date"
| "datatype"
```

| "duration"
| "time"
| "boolean"
;

se traduce a:
expr: "string" expr'| "decimal" expr'| "float" expr'| "double" expr'| "real" expr'| "integer" expr'
| "dateTime" expr'| "date" expr'| "datatype" expr'| "duration" expr'| "time" expr'| "boolean" expr'
| (expr) expr' | empty
expr' : "+" expr expr' | "-" expr expr' | "*" expr expr' | "/" expr expr' | "%" expr expr' | "MOD"
expr expr' | /*empty*/

Pseudocódigo del parser top-down

```
arbol ParseError(){
return node(empty(),"error",empty());
}

bintree parseExpr(){
if(token == "string") then
  sigtoken(token);
  return parseExpr'( node(empty(),"string",empty()));
} else if(token = "decimal"){
  sigtoken(token);
  return parseExpr'( node(empty(),"decimal",empty()));
} else if(token == "float"){
  sigtoken(token);
  return parseExpr'( node(empty(),"float",empty()));
} else if(token == "double"){
  sigtoken(token);
  return parseExpr'( node(empty(),"double",empty()));
} else if(token == "real"){
  sigtoken(token);
  return parseExpr'( node(empty(),"real",empty()));
} else if(token == "integer"){
  sigtoken(token);
  return parseExpr'( node(empty(),"integer",empty()));
} else if(token == "dateTime"){
  sigtoken(token);
  return parseExpr'( node(empty(),"dataTime",empty()));
} else if(token == "date"){
  sigtoken(token);
  return parseExpr'( node(empty(),"date",empty())) ;
}  else if(token == "datetype"){
  sigtoken(token);
  return parseExpr'( node(empty(),"datatype",empty()))
} else if(token == "duration"){
  sigtoken(token);
  return parseExpr'( node(empty(),"duration",empty()));
} else if(token == "time"){
  sigtoken(token);
  return parseExpr'( node(empty(),"time",empty())) ;
} else if(token == boolean){
```

```
  sigtoken(token);
  return parseExpr'( node(empty(),"boolean",empty()));
} else if (token = '(') {
  sigtoken (token);
  return parseExpr'(parseExpr())
}  else{    ParseError(); }

bintree parseExpr'(a: bintree){
if (token == '+' ) {
  sigtoken(token);
  a1 = node(a,"+",parseExpr());
  if(token == ')'){ sigtoken(token); }
  return parseExpr'(a1);
} else if (token == "-" ) {
  sigtoken(token);
  a1 = node(a,"-",parseExpr());
  if(token == ')'){ sigtoken(token); }
  return parseExpr'(a1);
} else if (token == "*" ) {
  sigtoken(token);
  a1 = node(a,"*",parseExpr());
  if(token == ")"){ sigtoken(token); }
  return parseExpr'(a1);
} else if (token == "/" ) {
  sigtoken(token);
  a1 = node(a,"/",parseExpr());
  if(token == ")"){ sigtoken(token); }
  return parseExpr'(a1);
} else if (token == "%" ) {
  sigtoken(token);
  a1 = node(a,"%",parseExpr());
  if(token == ")"){ sigtoken(token); }
  return parseExpr'(a1);
} else  if (token == "MOD"){
  sigtoken(token);
  a1 = node(a,"MOD",parseExpr());
  if (token == ")" {  sigtoken(token); }
    return parseExpr'(a1);
 }
 else // caso empty
    return a;
}

string sigToken(buffer: string)
{ tok: estring;
 if (buffer == "string") {
    tok  = buffer;
 } else if (buffer == "decimal") {
    tok  = buffer;
 } else if (buffer == "float") {
    tok  = buffer;
 } else if(buffer == "double") {
    tok  = buffer;
 } else if (buffer == "real") {
    tok  = buffer;
```

```
 }else if (buffer == "integer") {
   tok  = buffer;
 } else if (buffer == "dateTime") {
   tok  = buffer;
 } else if (buffer == "date") {
   tok  = buffer;
 } else if (buffer == "datatype") {
   tok  = buffer;
 } else if (buffer == "duration") {
   tok  = buffer;
 } else if (buffer == "time") {
   tok  = buffer;
 } else if (buffer == "boolean") {
   tok  = buffer;
 } else {
   return NULL;
 }
  return tok;
}
```

Función recursiva para inferir el tipo a partir del árbol de la expresión

ObtenerTipo:(n:árbol) : Type
/* Caso Base (caso no recursivo)
If empty(right(n)) and empty(left(n)) and
(root(n) == “integer” or root(n) == “decimal” or root(n) == “double” or
root(n) == “real” or root(n) == “time” or root(n) == date or root(n) == “float”)
then root(n)

/*Caso recursivo*/
*/*Casos !empty(left(n)) and empty(right(n))*/*
/(funciones de fechas)*/*
if root(n)= 'CURRENT_TIMESTAMP' then **Datetime**
if root(n)= 'CURRENT_DATE' then **Date**
*if root(n)= 'CURRENT_TIME' then **Time***
if root(n)='DAY' then **Integer**
if root(n)='MONTH' then **Integer**
if root(n)='YEAR' then **Integer**
if root(n)='WEEKDAY' then **Integer**
if root(n)='EXTEND' then **String**

/*(funciones de conversión)*/
if root(n)='DATE' then **Date**
if root(n)='TO_CHAR' then **String**
*if root(n)='TO_DATE' then **Date***

/(funciones con string)*/*
if root(n)='**ltrim**' then String
if root(n)='**rtrim**' then String
if root(n)='**upper**' then String
if root(n)='**lower**' then String
if root(n)='**reverse**' then String
*if root(n)='**lenght**' then Integer*

/*Casos !empty(left(n)) and !empty(right(n))*/

/(funciones de string)*/*
if root(n)='**left**' then String
if root(n)='**right**' then String
if root(n)='**replicate**' then String
if root(n)='**substring**' then String,una funcion de par ordenado
if root(n)='**concat**' then String

if root(n)= **'*'** then
   obtenerTipoOperacion(**ObtenerTipo**((left(n),x)), **ObtenerTipo**(right(n), x)) ;
if root(n)= '/' then
   obtenerTipoOperacion(**ObtenerTipo**((left(n),x)), **ObtenerTipo**(right(n), x)) ;
if root(n)= **'+,-'** then
   obtenerTipoOperacion**(ObtenerTipo**((left(n),x)), **ObtenerTipo**(right(n), x)) ;

obtenerTipoOperacion : Type x Type → Type

Esta función tiene dos nombres de tipos como argumento, uno de ellos contiene al otro y retorna el nombre del tipo con más valores.

obtenerTipoOperacion (t1,t2)
t1yt2 son tipos de xml_schema pq son el resultado de la funcion ObtenerTipo
if(t1== t2) then t1
else
si son numericos es el mayor
if(t1==integer and t2=double) then double
if(t1==decimal and t2=double) then double
if(t1==float and t2=double) then double
if(t1==integer and t2=decimal) then decimal
if(t1==char and t2=string) then string
if(t1==datetime and t2=duration) then datetime
if(t1==datetime and t2=datetime) then duration
if(t1==datetime and t2=duration) then datetime
if(t1==duration and t2=duration) then duration
if(t1==duration and t2=integer) then duration

where
opt_where: /* empty */
        | WHERE expr .{opt_where.schema= {} };

opt_orderby: /* empty */ | ORDER BY groupby_list ;

3.2.4 Mapeo de consultas con subconsultas anidadas.

table_factor= table_subquery opt_as NAME. { table_factor.schema = table_subquery.schema }
table_subquery::= '(' select_stmt ')'.{ 'table_subquery.schema= select_stmt.schema }

Join

```
Join_table:
        table_reference NATURAL opt_inner JOIN table_factor
              { join_table.schema =
                        table_reference.schema + ( table_factor.schema - table_references.schema) }
    |     table_reference NATURAL opt_left_or_right_outer JOIN table_factor.
              { join_table.schema =
                        table_reference.schema + ( table_factor.schema - table_references.schema) }

    |   table_reference opt_inner JOIN table_factor.
              { join_table.schema= table_reference.schema + table_factor.schema }
    |   table_reference opt_inner JOIN table_factor ON expr .
              { join_table.schema= table_reference.schema + table_factor.schema }
    |   table_reference left_or_right opt_outer JOIN table_factor ON expr.
              { join_table.schema= table_reference.schema + table_factor.schema }

    |   table_reference opt_inner JOIN table_factor USING '(' column_list ')'.
             { join_table.schema =   column_list.schema +
                                    (table_reference.schema - column_list.schema)+
                                    (table_factor.schema - column_list.schema) }
    |   table_reference left_or_right opt_outer JOIN table_factor USING '(' column_list ')'
              { join_table.schema = column_list.schema +
                                     (table_reference.schema - column_list.schema) +
                                   + (table_factor.schema - column_list.schema) }
    ;

opt_inner: /* nil */ | INNER ;
opt_outer: /* nil */  | OUTER ;
left_or_right: LEFT | RIGHT ;
opt_left_or_right_outer: LEFT opt_outer | RIGHT opt_outer | /* empty */  ;
```

4. Mapeo de consultas XQuery a esquemas XML

4.1. Gramática de XQuery (ver [7])

Una consulta XQuery consiste de un prólogo y del cuerpo de la consulta.

MainModule ::= Prolog QueryBody

Prólogo

El prólogo contiene una lista de declaraciones de variables y declaraciones de funciones.

Prolog ::= (AnnotatedDecl “;”)*
AnnotatedDecl ::= “declare” (VarDecl | FunctionDecl)

Declaración de variables

La declaración de una variable contiene el nombre de la variable precedido por
$, opcionalmente una declaración del tipo de la variable y luego la asignación de una expresión.
VarDecl ::= "variable" "$" VarName TypeDeclaration? ":=" ExprSingle

Declaración de funciones

La declaración de una función consiste del nombre de la función, una lista opcional de parámetros, opcionalmente un tipo de retorno y luego el cuerpo de la función que es una expresión entre llaves.
La lista de parámetros contiene parámetros donde cada parámetro es un nombre precedido por $ y opcionalmente puede tener un tipo.

FunctionDecl ::= **'declare' 'function'** FunctionName '(' ParamList? ')' (“as” SequenceType)? (EnclosedExpr)
ParamList ::= Param (',' Param)*
Param ::= '$' QName TypeDeclaration?
EnclosedExpr ::= '{' Expr '}'
Ejemplo

```
   declare function local:summary($emps)
  {
      for $d in fn:distinct-values($emps/deptno)
         let $e := $emps[deptno = $d]
      return
        <dept>
          <deptno>{$d}</deptno>
          <headcount> {fn:count($e)} </headcount>
          <payroll> {fn:sum($e/salary)} </payroll>
        </dept>
  };
```

local:summary(fn:doc("acme_corp.xml")//employee[location = "Denver"]

Declaraciones de tipos

TypeDeclaration ::= “as” SequenceType
SequenceType ::= ("empty-sequence" "(" ")") | (ItemType OccurrenceIndicator?)
OccurrenceIndicator ::= "?" | "*" | "+"

Expresiones

Esta sección se presentan cada uno de los tipos básicos de expresión. Cada tipo de expresión tiene un nombre como **PathExpr**, que se introduce en el lado izquierdo de la producción de gramática que define la expresión. Como XQuery es un lenguaje composable, cada tipo de expresión se define en términos de otros expresiones cuyos operadores tienen una precedencia más alta. De esta manera, la precedencia de los operadores se representa explícitamente en la gramática.
El orden en que se presentan las expresiones en este documento no refleja el orden de precedencia del operador. En general, presentaremos los tipos más simples de expresiones primero, seguidas por expresiones más complejas. La gramática completa se puede consultar en [A XQuery Grammar].
Una expresión está representada en la gramática XQuery por símbolo **Expr**.

Construyendo Secuencias

Expr ::= ExprSingle (',' ExprSingle)*

Una forma de construir una secuencia es usando el operador de coma, que evalúa cada uno de sus operandos y concatena las secuencias resultantes, en orden, en una única secuencia de resultados. Los paréntesis vacíos se pueden usar para denotar una secuencia vacía.

ExprSingle puede evaluar una secuencia que contenga más de un elemento y se usa en varios lugares de la gramática donde una expresión no puede contener una coma. El operador que tiene la prioridad más baja es el operador de coma.

Una secuencia puede contener valores o nodos atómicos duplicados, pero una secuencia es nunca un artículo en otra secuencia. Cuando se crea una nueva secuencia concatenando dos o más secuencias de entrada, la nueva secuencia contiene todas las items de las secuencias de entrada y su longitud es la suma de las longitudes de la entrada secuencias.
Ejemplo: El resultado de esta expresión es una secuencia de cinco enteros: (10, 1, 2, 3, 4)

```
ExprSingle ::= FLWORExpr
             | QuantifiedExpr
             | IfExpr
             | OrExpr
             | PrimaryExpr
```

Las expresiones que tienen la siguiente prioridad más baja son FLWORExpr, QuantifiedExpr, IfExpr y OrExpr. Cada una de estas expresiones se describe en las secciones siguientes.

Expresiones Flwor

```
FLWORExpr      ::= ( ForClause | LetClause )+ WhereClause? OrderByClause? 'return'
ExprSingle
ForClause      ::= 'for' '$' VarName  'in' ExprSingle ( ',' '$' VarName  'in' ExprSingle )*
LetClause      ::= 'let' '$' VarName  ':=' ExprSingle ( ',' '$' VarName  ':=' ExprSingle )*
WhereClause    ::= 'where' ExprSingle
OrderByClause  ::= ( 'order' 'by' ) OrderSpecList
OrderSpecList  ::= OrderSpec ( ',' OrderSpec )*
OrderSpec      ::= ExprSingle OrderModifier
OrderModifier  ::= ( 'ascending' | 'descending' )?
```

FLWORExpr se usa para definir expresiones FLWOR completas.
ForClause sirve para definir una o más relaciones entre variables y secuencias de valores (esas variables van a tomar uno a uno los valores de las secuencias).
LetClause sirve para definir una o más asignaciones de secuencias de valores a variables.
WhereClause es opcional y se usa para definir una condición usada para filtrar las tuplas de valores generadas por el for.
OrderByClause es opcional y se usa para indicar cómo se van a ordenar las tuplas generadas por el for (que cumplen la WhereClause si la hay)
Para cada tupla (que cumple la WhereClause si la hay y en el orden dado por la clausula OrderByClause si la hay) se evalúa la cláusula return la cual construye los resultados de la expresión FLWOR.
El resultado de la expresión FLWOR es una secuencia ordenada que contiene los resultados de estos evaluaciones, concatenadas usando operador de coma.

Expresiones Quantificadas

Las expresiones cuantificadas respaldan la cuantificación existencial y universal. El valor de una expresión cuantificada siempre es verdadero o falso.
QuantifiedExpr ::=
('**some**' | '**every**') '**$**' VarName '**in**' ExprSingle ('**,**' '**$**' VarName '**in**' ExprSingle)* '**satisfies**' ExprSingle
Una expresión cuantificada comienza con un cuantificador, que es la palabra clave **some** or **every**, seguido de una o más cláusulas internas que se usan para vincular variables, seguidas por la palabra clave **satisfies** y una expresión de prueba. Cada cláusula asocia a una variable una expresión que devuelve una secuencia de elementos, llamada secuencia de enlace para esa variable.
Ejemplo: **every** $part **in** /parts/part **satisfies** $part/@discounted
Esta expresión es verdadera si cada elemento de *part* tiene un atributo *discounted* (independientemente de los valores de estos atributos).
Ejemplo: **some $**emp **in** /emps/employee **satisfies** ($emp/bonus > 0.25 * $emp/salary)
Esta expresión es verdadera si al menos un elemento employee satisface el expresión de comparación.

Expresiones If

XQuery admite una expresión condicional basada en las palabras clave if, then, y else.

IfExpr ::= '**if**' '**(**' Expr '**)**' '**then**' ExprSingle '**else**' ExprSingle

La expresión que sigue a la palabra clave if se llama expresión de prueba, y las expresiones que siguen a las palabras clave then y else son expression sigles.
En este ejemplo, la expresión de prueba es una expresión de comparación:
if ($widget1/unit-cost < $widget2/unit-cost**)**
then $widget1
else $widget2
En este ejemplo, la expresión de prueba se fija en la existencia de un atributo nombrado como discounted, independientemente de su valor:
if ($part/@discounted**)**
then $part/wholesale
else $part/retail

Expresiones Primarias

Las expresiones primarias son las primitivas básicas del lenguaje. Incluyen literales, referencias de variables, expresiones de elementos de contexto, constructores y llamadas a funciones.
También se puede crear una expresión primaria al encerrar cualquier expresión entre paréntesis, lo que a veces es útil para controlar la precedencia de los operadores.

PrimaryExpr ::= Literal | VarRef | ParenthesizedExpr | ContextItemExpr | FunctionCall
| OrderedExpr | UnorderedExpr | Constructor

Literales

Un literal es una representación sintáctica directa de un valor atómico. XQuery admite dos tipos de literales: literales numéricos y literales de cadenas.

```
Literal          ::= NumericLiteral
NumericLiteral   ::= IntegerLiteral | DecimalLiteral | DoubleLiteral
IntegerLiteral   ::= Digits
DecimalLiteral   ::= ('.' Digits) | (Digits '.' [0-9]*)
DoubleLiteral      ::= ( ('.' Digits) | (Digits ( '.' [0-9]* )? )) [eE] [+-]? Digits
Digits           ::= [0-9]+
```

Referencias a variables

Una referencia de variable es un QName precedido por un el simbolo $.

```
VarRef    ::= '$' VarName
VarName  ::= QName
```

Una variable puede estar referenciada por una expresión XQuery. Los clases de expresiones que pueden vincular variables son expresiones FLWOR, expresiones cuantificadas.Las llamadas a funciones también se unen valores a los parámetros formales de las funciones antes de ejecutar la cuerpo de la función.

Expresiones entre paréntesis

ParenthesizedExpr ::= '(' Expr? ')'

Los paréntesis se pueden usar para hacer cumplir una orden de evaluación particular en expresiones que contienen múltiples operadores. Por ejemplo, la expresión (2 + 4) * 5 evalúa a treinta, ya que la expresión entre paréntesis (2 + 4) se evalúa primero y su resultado se multiplica por cinco. Sin paréntesis, la expresión 2 + 4 * 5 evalúa a veintidós, porque el operador de multiplicación tiene mayor precedencia que el operador de suma.
Los paréntesis vacíos se utilizan para denotar una secuencia vacía se describe más adelante.

Expresión item de contexto

ContextItemExpr :: = "."

Una expresión item de contexto evalúa el item de contexto, como se vio antes que puede ser un nodo (como en la expresión fn: doc ("bib.xml") / books / book [fn: count (./ author)> 1]), o un valor atómico o función (como en la expresión (1 a 100) [. mod 5 eq 0]).
Si el item de contexto está ausente, una expresión de item de contexto genera un error dinámico [err: XPDY0002].

Llamadas de función

Las funciones built-in soportadas por Xquery se definen en [XQuery 1.0 and XPath 2.0 Functions and Operators]. XQuery permite a los usuarios declarar funciones propias. En cualquiera de los dos casos Una llamada de función consiste en un QName seguido de una lista entre paréntesis de cero o más expresiones, llamadas argumentos.

FunctionCall ::= FunctionName '(' (ExprSingle (',' ExprSingle)*)? ')'

Expresiones ordenadas

OrderedExpr ::= 'ordered' '{' Expr '}'
El propósito de las expresiones ordenadas y desordenadas es establecer el modo de ordenación en el contexto estático ordenado o desordenado para una determinada región en una consulta. El modo de ordenamiento especificado se aplica a la expresión anidada dentro de las llaves.

Expresiones no ordenadas

UnorderedExpr ::= 'unordered' '{' Expr '}'
ejemplo

```
unordered {
   for $p in fn:doc("parts.xml")/parts/part[color = "Red"],
       $s in fn:doc("suppliers.xml")/suppliers/supplier
   where $p/suppno = $s/suppno
  return
    <ps>
    { $p/partno, $s/suppno }
    </ps>
}
```

Constructor

Constructor ::= DirectConstructor
DirectConstructor ::= DirElemConstructor
DirectConstructor ::= “<” QName DirAtributeList “/>”
| “<” QName DirAtributeList “>” DirElemContent* “</” QName S? “>”
DirAttributeList ::= (S (Qname S?) “=” S? (’**”** ’(“**{**” ExprSingle ”**}**”)* ’**”** '))?)*
DirAttributeList ::= (S (Qname S?) “=” S? (“ ' ” (“**{**” ExprSingle ”**}**”)* “ ' ”))?)*
S::= son espacios en blanco
DirElemContent::= DirectConstructor* | (“**{**” ExprSingle ”**}**”)*
Veamos algunos ejemplos:

```
<shoe size="7"/>
```

Añadimos el valor 7 al atributo size del elemento shoe, de manera idéntica podemos incluir expresiones anidadas:

```
<shoe size="{7}"/>
```

Expresiones Lógicas

Una expresión lógica es una expresión *and* o una expresión *or*. Si una expresión lógica no genera un error, su valor es siempre uno de los booleanos valores verdadero o falso.
OrExpr ::= AndExpr ('or' AndExpr)*
AndExpr ::= ComparisonExpr ('and' ComparisonExpr)*
El primer paso para evaluar una expresión lógica es evaluar cada uno de sus operandos.
En el ejemplo retorna true :
1 eq 1 **and** 2 eq 2
1 eq 1 **or** 2 eq 3

Como se verá, en el caso de no haber conectivos, una OrExpr puede tomar muchas formas, por ejemplo: expresiones aritméticas, expresiones de camino.

Expresiones de comparación
ComparisonExpr ::= AdditiveExpr ((ValueComp | GeneralComp) AdditiveExpr)?
ValueComp ::= '**eq**' | '**ne**' | '**lt**' | '**le**' | '**gt**' | '**ge**'
GeneralComp ::= '**=**' | '**!=**' | '<' | '<=' | '>' | '>='

Expresiones Aritméticas

XQuery proporciona operadores aritméticos para suma, resta, multiplicación división y módulo, en sus formas binarias y unarias usuales.

AdditiveExpr ::= MultiplicativeExpr (('+' | '-') MultiplicativeExpr)*
MultiplicativeExpr ::= UnionExpr (('*' | 'div' | 'idiv' | 'mod') UnionExpr)*

Si un **AdditiveExpr** contiene más de dos **MultiplicativeExprs**, se agrupan de izquierda a derecha. Entonces, por ejemplo, A - B + C - D es equivalente a ((A - B) + C) - D. La no terminal **UnionExpr** se explica en la siguiente sección.

Aquí hay algunos ejemplos de expresiones aritméticas:

La primera expresión a continuación devuelve el valor de xs: decimal -1.5, y la segunda expresión devuelve el valor de xs: integer -1:

```
-3 div 2
-3 idiv 2
```

La resta de dos valores de fecha da como resultado un valor de tipo xs: dayTimeDuration:

```
$emp/hiredate - $emp/birthdate
```

Este ejemplo ilustra la diferencia entre un operador de resta y un guión:

```
$ precio unitario - $ unit-discount
```

Combinación de secuencias de nodos

UnionExpr ::= IntersectExceptExpr (('union' | '|') IntersectExceptExpr)*
IntersectExceptExpr ::= UnaryExpr (("intersect" | "except") UnaryExpr)*
UnaryExpr ::= ('-' | '+')* PathExpr

XQuery proporciona los siguientes operadores para combinar secuencias de nodos:
En la no terminal **UnionExpr**, la **'union'** y **'|'** los operadores son equivalentes; toman dos secuencias de nodos como operandos y devolverá una secuencia que contiene todos los nodos de ambas secuencias.
En la no terminal **IntersectExceptExpr**, el operador de **"intersect"** toma dos secuencias de nodo como operandos y devuelve una secuencia que contiene todos los nodos que ocurren en ambos operandos. El operador **except** toma dos secuencias de nodos como operandos y devuelve un secuencia que contiene todos los nodos que ocurren en el primer operando pero no en el segundo operando. La no terminal **PathExpr** se explica en la siguiente sección.

Expresiones de ruta

PathExpr ::= ('/' RelativePathExpr ?) |('//' RelativePathExpr) | RelativePathExpr

RelativePathExpr ::= StepExpr (('/' | '//') StepExpr)*
Una expresión de ruta se puede usar para localizar nodos dentro de los árboles.Una expresión de ruta consiste en una serie de uno o más pasos, separados por "/" o "//", y opcionalmente comenzando con "/" o "//". Un "/" inicial o "//" es una abreviatura de uno o más pasos iniciales que se agregan implícitamente al comienzo de la expresión de ruta, como se describe a continuación.
Una expresión de ruta que consta de un solo paso se evalúa como se describe en la siguiente sección Pasos.

Pasos

StepExpr ::= FilterExpr | AxisStep
AxisStep ::= (ReverseStep | ForwardStep) PredicateList
ForwardStep ::= ForwardAxis NodeTest | AbbrevForwardStep

ReverseStep ::= ReverseAxis NodeTest | AbbrevReverseStep
PredicateList ::= Predicate*

Un paso es una parte de una expresión de ruta que genera una secuencia de items y luego filtra la secuencia por cero o más predicados. El valor del paso consiste en aquellos items que satisfacen los predicados, trabajando de izquierda a derecha. Un paso puede ser un paso de eje o una expresión de filtro. Las expresiones de filtro son descrito en 3.3.2 Expresiones de filtro.

Expresiones de filtro

FilterExpr ::= PrimaryExpr PredicateList
PredicateList ::= Predicate*

Una expresión de filtro consiste simplemente en una expresión primaria seguida por cero o más predicados. El resultado de la expresión del filtro consiste en los elementos devuelto por la expresión primaria, Items filtrados que se obtienen aplicando cada predicado de izquierda a derecha. Si no se especifican predicados, el resultado es simplemente el resultado de la expresión primaria. El orden de los items devueltos por una expresión de filtro es lo mismo que su orden en el resultado de la expresión primaria.
Las posiciones de contexto se asignan a los elementos en función de su posición ordinal en la secuencia resultado. La primera posición de contexto es 1.
Por ejemplo, la descripción formal de las expresiones de filtro sugiere que $s[1] debe evaluarse examinando todos los elementos en secuencia $s, y seleccionando todos aquellos que satisfacen el predicado position() = 1.

Node Tests

Un **Node Tests** que consta únicamente de un QName o un Comodín se llama name test.
Un **QName** en un name test que se resuelve en un QName expandido usando el estáticamente espacios de nombres conocidos en el contexto de la expresión.

NodeTest ::= KindTest | NameTest
NameTest ::= QName | Wildcard
Wildcard ::= '*'
KindTest ::= ElementTest | AttributeTest | TextTest | AnyKindTest
AnyKindTest ::= 'node' '(' ')'
TextTest ::= 'text' '(' ')'
AttributeTest ::= 'attribute' '(' (AttribNameOrWildcard (',' TypeName)?)? ')'
AttribNameOrWildcard ::= AttributeName | '*'

ElementTest ::= 'element' '(' (ElementNameOrWildcard (',' TypeName '?'?)?)? ')'
ElementNameOrWildcard ::= ElementName | '*'
AttributeName ::= QName
ElementName ::= QName
TypeName ::= QName

Forward Axis

ForwardAxis ::= 'child' '::' | 'descendant' '::' | 'attribute' '::' | 'self' '::' | 'descendant-or-self' '::'
| 'following-sibling' '::' | 'following' '::'

Reverse Axis

ReverseAxis ::= 'parent' '::' | 'ancestor' '::' | 'preceding-sibling' '::' | 'preceding' '::
| 'ancestor-or-self' '::'

AbbrevForward Step
AbbrevForwardStep ::= '@'? NodeTest

AbbrevReverse Step
AbbrevReverseStep ::= '..'

Predicate
Predicate ::= '[' Expr ']'

4.2. Definición del mapeo de consultas XQuery a esquemas XML usando un attribute grammar.

4.2.1 Mapeo de XML Schemas de expresiones FLWOR

Tenemos el attribute grammar de expresiones simples, englobadas, FLRExpr,forClause, letClause, primaryExpr, ifExpr de [1] **:**

Construyendo Secuencias

ExprSingle = "(" ExprSingle ("," ExprSingle)* ")".
{ ExprSingle[1].schema =
"<xsd:sequence>" +
ExprSingle[2].schema + ... +ExprSingle[i].schema
+ "</xsd:sequence>"); }

Expresiones englobadas

ExprSingle = "{" ExprSingle "}". { ExprSingle[1].schema = ExprSingle[2].schema; }

La expresión FLRExpr

La expresión FLRExpr es una ExprSingle

ExprSingle = FLRExpr. { ExprSingle.schema = FLRExpr.schema; }

FLRExpr

Una cláusula for itera sobre un conjunto de nodos y para cada paso de iteración la cláusula return es evaluada. Por lo tanto si existe al menos una cláusula for, nosotros declaramos los nodos del esquema de salida de la cláusula return están dentro de una secuencia.

FLRExpr = (ForClause | LetClause)+ "return" ExprSingle.
{ if(there exists at least one ForClause)
FLRExpr.schema = "<xsd:sequence minOccurs=0" + " maxOccurs=’unbounded>’"
+ ExprSingle.schema + "</xsd:sequence>";
else FLRExpr.schema = ExprSingle.schema; }

El esquema de salida de la cláusula return para un ForClause sus ítems están dentro de una secuencia.

ForClause

ForClause = "for" "$" VarName "in" ExprSingle.
{ *schemaOfVariable*(VarName.getString(), ExprSingle.schema); }

Usamos la función schemaOfVariable para almacenar el esquema de salida de la declaración de la variable. Requerimos la información del esquema de salida de la variable más tarde cuando el contenido de la variable es accedida.

PrimaryExpr:=VarRef.{PrimaryExpr.schema=varRef.schema}
VarRef ::= '$' VarName. {VarRef.schema = getVarSchema(QName.getString()); }}

Si se recupera el contenido de una variable, determinamos el esquema de la variable por la función getVarSchema. Esta función getVarSchema devuelve el esquema de la variable. Si la variable es un parámetro de una función (en el esquema de declare function) devuelve el nombre de la variable , en este caso, el nombre de la variable será reemplazada por su correspondiente esquema cuando la recuperemos en el esquema de la llamada a función.

LetClause

LetClause = "let" "$" VarName ":=" ExprSingle.
{ *schemaOfVariable*(VarName.getString(), ExprSingle.schema); }

Expresion if

ExprSingle = IfExpr. { ExprSingle.schema = IfExpr.schema; }

IfExpr = "if" "(" ExprSingle ")" "then" ExprSingle "else" ExprSingle.
{ IfExpr.schema = "<xsd:choice>" +
ExprSingle[2].schema +
ExprSingle[3].schema +

"</xsd:choice>"); }

xs: choice es un compositor que define un grupo de partículas mutuamente excluyentes, y con un máximo de un grupo de dos partículas podemos representar el if .

4.2.2 Mapeo de XML Schemas de Constructor Directo

DirectConstructor ::= <QName DirAtributeList > ({ExprSingle})* </QName>**.**
{ DirectConstructor.schema =
"<element name="+ QName +">"
<complexType>"+
DirAtributeList.schema +
"<sequence>" +
ExprSingle[1].schema + ExprSingle[2].schema + ... + ExprSingle[i].schema +
"</sequence>
<complexType>" **}**

DirAtributeList = (QName = "{ ExprSingle }")* **. {**
DirAtributeList.schema =
"<attribute name ="+ QName[1] + " type= " + obtenerTipo(expreSingle[1].schema) + ">"
<attribute name ="+ QName[2] + "type=" + obtenerTipo(expreSingle[2].schema) + ">"
+ ... + "<attribute name = "+QName[i]+" type=" + obtenerTipo(expreSingle[j].schema)+">"**}**

4.2.3 Especificación de tipos de datos usados

En esta sección vamos a definir tad mapeo, stack y árbol general que serán usados en las siguientes secciones.
Vamos a necesitar especificar una estructura que represente un conjunto de claves y una colección de valores donde cada clave tiene asociado un valor. Más adelante se verá que las claves serán nombres de variables o nombres de funciones o nombres de parámetros y necesitamos asociarlas a su esquema correspondiente.

TAD mapeo[elem]

constructores
empty : mapeo
mapear : elem × mapeo → mapeo

operaciones
contieneclave : clave x mapeo → bool
obtenervalor : clave x mapeo→ valor si contieneclave(clave,mapeo)=true

ecuaciones
sea m: mapeo, c,k: clave, s: valor
contieneclave(c, empty) = false
contieneclave(c,mapear(k,s,m))= c=k or contieneclave(c,m)
obtenervalor(c, mapear(k,s,m))= if c=k then s else obtenervalor(c,m)

Los elementos del TAD mapeo son pares (clave, valor), lo elementos se localizan mediante su clave.
En el caso de mapeo de variables con su esquema, clave=nombre de la variable, valor= esquema de la variable
schemaofvariable=mapear getvarschema=obtenervalor

Pseudocódigo de implementación de mapeo

Una forma sencilla de implementar mapeo es con un arreglo donde se van alojando los elementos. Como los arreglos tienen tamaño fijo y el mapeo no, hace falta un indicador de cuántos elementos hay en el arreglo.

```
type mapeo = tuple
                elems: array [1..N] of elem;
                int size;
             end
```

Con esta representación, el campo elems es el arreglo en el que se alojan los elementos del mapeo y el campo size indica cuantos elementos se encuentran alojados actualmente en el mapeo. El mapeo m se inicializa como mapeo vacío, asignando 0 al campo size de m.

```
proc empty(out m:mapeo)
  m->size = 0;
end
```

El primer elemento que se agrega al mapeo se aloja en m.elems[1], el segundo en m.elems[2], etc. A medida que se agregan dichos elementos, el campo m.size adopta los valores 1,2, etc. Es fácil observar que el campo size indicará la última celda del arreglo elems ocupada por el mapeo. En efecto, los elementos del mapeo m se encontrarán en las posiciones m.elems[1], m.elems[2], . . . , m.elems[m.size] en el orden en que ingresaron a m.
Implementada el mapeo de esta manera, para agregarle un elemento es necesario que quede espacio en el arreglo. Eso se indica a continuación con la precondición ¬ is_full(mapeo). Al agregar un elemento al mapeo, su número de elementos se incrementa en 1, lo que debe consignarse incrementando el campo size. Como el campo size indicaba la última celda

del arreglo elems ocupada por el mapeo, ahora que fue incrementada indica la primera celda libre. Allí es donde se aloja el nuevo elemento e.

```
proc mapear( in e:elem, in / out m:mapeo)
   m->size = m->size+1;
   strcpy(m->elems[m->size].clave, e.clave);
   strcpy(m->elems[m->size].valor, e.valor);
end

fun contieneclave( var:Clave, m:mapeo) return bool
  int cont;
  for(cont = 0; cont < N; cont++){
     if( strcmp( m->elems[cont].clave, var) == 0){
       return 1;
     }
  }
  return 0;
end

fun obtenervalor( in var: Clave, in m:mapeo, out val:valor)
  int cont; bool b;
  b=contieneclave( var, m);
  if(b==true){
      for(cont = 0; cont < N; cont++){ // Se recorre la matriz 'm'
         if( strcmp( m->elems[cont].clave, var) == 0){
       return m->elems[cont].valor;
    }
      }
   }
end

fun is_full(m:mapeo) return b:bool
   b:=(m->size==N)
end
```

Necesitaremos poder procesar funciones anidadas donde el resultado de cada función que es un parámetro de la función que estamos procesando se tendrá que obtener previamente para obtener su resultado y usaremos una pila para este proceso. La pila se lo va usar para obtener el esquema de una llamada a función.

TAD stack[elem]

constructores
empty : stack
push : elem × stack → stack

operaciones
top : stack → elem {se aplica sólo a una pila no vacía}
pop : stack → stack {se aplica sólo a una pila no vacía}
is_empty : stack → bool

ecuaciones
top(push(e,s)) = e
pop(push(e,s)) = s

```
is_empty(empty) = true
is_empty(push(e,s)) = false

type node = tuple
                value: elem
                next: pointer to node
              end
type stack = pointer to node
proc empty( out p:stack)
   p:= null
end

proc push( in e:elem, in / out p:stack)
  var   q: pointer to node
    alloc(q)
    q->value:= e
    q->next:= p
    p:= q
end

fun top(p:stack) return e:elem          {se aplica a p sólo cuando ¬ is_empty(p)}
 e:= p->value
end

proc pop( in / out p:stack)    {se aplica a p sólo cuando ¬ is_empty(p)}
  var q: pointer to node
  q:= p
  p:= p->next
  free(q)
end

fun is_empty(p:stack) return b: Bool
  b:= (p == null )
end

proc destroy( in / out p:stack)          {libera todo el espacio de memoria ocupado por p}
   while ¬ is_empty(p) do pop(p) od
end
```

Un árbol general es un árbol que tiene un padre, hijos y hermanos que también son arboles.
El árbol general se lo va usar para obtener el esquema de una expresión XPath.

TAD arbolgeneral[elem]

constructores
empty: → arbolgeneral
node : arbolgeneral x elem x arbolgeneral x arbolgeneral → arbolgeneral

operaciones
padre: arbolgeneral → arbolgeneral {se aplica sólo a un árbol no vacío}
raiz: arbolgeneral → elem {se aplica sólo a un árbol no vacío}
hijo_izq: arbolgeneral → arbolgeneral {se aplica sólo a un árbol no vacío}
hermano_der: arbolgeneral → arbolgeneral {se aplica sólo a un árbol no vacío}

ecuaciones
raiz(node(p,e,h1,hd))== e
padre(node(p,e,h1,hd))== p
hijo_izq(node(p,e,h1,hd))== h1
hermano_der(node(p,e,h1,hd))== hd
is_empty(empty)=true
is_empty(node(p,e,h1,hd))=false

Pseudocódigo de arbolgeneral

```
type tnode = tuple
                padre: pointer to tnode
                elemento: string
                primer_hijo: pointer to tnode
                hermano_derecho: pointer to tnode
             end
type arbolgeneral = pointer to tnode;
AG(padre,elemento,primer_hijo,hermano_derecho)

fun empty return t:arbolgeneral
  t:=null;
end

fun node(p:arbolgeneral, e: string, h1:arbolgeneral, hd:arbolgeneral ) return t:arlbolgeneral
   alloc(t)
   t->padre:=p
   t->elemento:=e
   t->primer_hijo:=h1
   t->hermano_derecho:=hd
end

fun raiz( t:arbolgeneral) return  e:string          { se aplica sólo cuando ¬is_empty(t) }
    if ( is_empty(t->padre) and is_empty(t->hermano_derecho) )
      e:=t->elemento;
    fi
end

fun padre(t:arbolgeneral) return p:arbolgeneral    { se aplica sólo cuando ¬is_empty(t) }
   p:=t->padre
end

fun hijo_izq(t: arbolgeneral) return p:arbolgeneral    { se aplica sólo cuando ¬is_empty(t) }
```

```
   p:=t->primer_hijo;
end

fun hermano_der(t:arbolgeneral) return hd:arbolgeneral{ se aplica sólo cuando ¬ is_empty(t) }
   hd:=t->hermano_derecho
end

fun is_empty(t: arbolgeneral) return b:bool
  b:=(p==null);
end
proc  destroy( in / out t:arbolgeneral)  {libera todo el espacio de memoria ocupado por t}
   if ¬ is_empty(t)
        then destroy(padre(t))
           destroy(hijo_izq(t))
           destroy(hermano_der(t))
           free(t)
           t:= null
   fi
end

fun crear_arbolgeneral(e:string) return t:arbolgeneral
    t->elemento:=e
    t->padre:=null
   t->primer_hijo:=null
   t->hermano_derecha:=null
end

fun insertar padre(p:arbolgeneral, t:arbolgeneral) return r:arbolgeneral
     if(is_empty(t->padre))
       t->padre:=p
     fi
     r:=t
end

La función inserta un hijo a un árbol que no tiene hijos.
fun insertar_hijo_izq(in e:string, in t:arbolgeneral) return r:arbolgeneral
       var nuevo,r: pointer to tnode
       alloc(nuevo)
       if(  ¬ is_empty(t) and is_empty(t->primer_hijo) )
          nuevo<-crear_arbolgeneral(e:string)
           t->primer_hijo:=nuevo
       fi
      r:=t
end

Inserta un hermano a la derecha en un árbol que no tiene hermanos a la derecha
fun insertar_primer_hermano_der(in e:string, in t:arbolgeneral) return r:arbolgeneral
    var q,r: pointer to tnode
       alloc(q)
       if(  ¬ is_empty(t) and is_empty(t->hermano_der) )
          q<-crear_arbolgeneral(e:string)
           t->hermano_der:=q
       fi
      r:=t
end
```

Inserta un hijo a un árbol.

```
fun insertar_hijo(in e:string, in t:arbolgeneral) return r:arbolgeneral
   var a:arbolgeneral
     a ← empty()
   q ← crear_arbol_general(e)
   if( ¬ is_empty(t) )
         if( is_empty( t->primer_hijo) )
            q ← insertar_hijo_izq( e, t )
            r ← q
         else
            a := t->primer_hijo
            if ( ¬ is_empty( a->hermano_der ))
               r:=a->hermano_der        {r realiza la búsqueda a partir del primer hermano derecho
nodo}
               while r->hermano_der  != null do          {mientras ? r no sea el último nodo}
                     r:=r->hermano_der                   {que r pase a señalar el nodo siguiente}
               od
              r->hermano_der := q
            else
                r:= insertar_hermano_der(e,a)
            fi
   fi
end
```

4.2.4. Mapeo a esquemas XML de declaraciones y llamadas de funciones.

Attribute grammar para declaraciones de funciones definidas por el usuario
XQuery les permite a los usuarios declarar funciones propias, el attribute grammar para hacerlo es el siguiente:

FunctionDecl ::= "declare" "function" QName "(" ("$" QName ("," "$" QName)*)? ")" "{" ExprSingle "};".
{ FunctionDecl.schema =
"<xsd:group name="+**QName+"** >
<xsd:sequence>
<xsd: element ref="+**QName[1]+"**>
<xsd: element ref="+**QName[2]+"** >
…
<xsd: element ref="+**QName[i]+"** >"+
asociarParametroFuncion(QName,QName[1],..,QName[i]);+
asociar(QName, ExprSingle[2].schema); +
"</xsd:sequence>
</xsd:group>"}

En el attribute grammar de declaración de función necesitamos saber cual son los parámetros y el esquema de la expresión de la función y para eso utilizo las funciones siguientes:
La función ***asociarParametroFuncion(QName,QName[1],..,QName[i])*** crea una asignación, nombre de la función aquí QName con la lista de parámetros de la función QName[1],..,QName[i]

asociar(QName, ExprSingle[2].schema)
asocia el nombre de función con el esquema del contenido de su declaración de función.

Ahora hacemos el attribute grammar de la llamada de función (puede ser built-in o definida por el usuario).

FunctionCall ::= QName "(" (ExprSingle ("," ExprSingle)*)? ")".
{
eval(getString(QName) **,** ExprSingle[1].schema + ";" + ExprSingle[2].schema + ";"
+ … + ";" + ExprSingle[i].schema)
}

En la llamada de función eval usamos los esquemas de los argumentos y el nombre de la función de la declaración de función.

Si *E* es el esquema asociado a la función de nombre *functName*, *F* es el esquema que resulta de reemplazar en *E* los nombres de los parámetros de la función de nombre *QName* por los esquemas de los argumentos respectivos (o sea, ExprSingle[1].schema,…, ExprSingle[i].schema), entonces eval(functName, argSchemas) retorna el resultado de evaluar la expresión *F*. *F* va a tener la forma strings de texto intercalados con llamadas a funciones y en las llamadas de funciones puede haber funciones anidadas. Llamemos a un anidamiento de funciones un *término*. Entonces evaluar *F* significa evaluar todos los términos en *F* dejando todos los strings como estaban. La función que evalúa un término se llama *evalgenérico* y la describimos a continuación.

eval generico

```
/* Algoritmo para evaluar expresiones que tienen llamadas a funciones, funciones anidadas
fun evalgenerico(expresión:string) return string
var p: stack of N
var i: integer; /* usado para recorrer expresión */
var j integer /* usado para guardar longitud de string expresión */
var k integer /* usado para guardar parámetros */
var nombre_funcion, parámetro :string
string parámetros array[10]; /* contendrá parámetros de una función */
j :=  longitud(expresion)
empty(p)
i:= 0;
mientras i <j
  i  =  reconocer(i, expresión);
 c := expresión[i];
 /* reconoce carácter significativo después de posicion i  e
    i pasa a ser posicion de ese carácter significativo*/
  switch c
    case '(':  nombre_funcion<-guardar_nombre_función(expresion, i)
              /* como no hay manera de distinguir nombre de función de nombre de parámetro
                 sintácticamente, tenemos que indicar en la pila si algo es función o parámetro
               push(<nombre_funcion, “function”>,p);

    case ','    /* como encontre ',' , si antes de él no hay ‘)’ entonces llegué  parámetro
                   de una funcion y hay que agregarlo a ese parámetro  a la pila*/
                if  not cierreParentesisAntes(s, i)  then
                    parámetro <- guardar_parámetro(expresión, i);
                    push(<parámetro, “param”>, p)
                fi
                /* como encontre ',' , si antes de él hay ‘)’ entonces llegué  parámetro
                   de una funcion y no hay que agregar a ese parámetro  a la pila porque ya está allí*/

    case )':  /* como encontre ')' , si antes de él no hay ‘)’ entonces llegué al último parámetro
                   de una funcion y hay que agregarlo a ese parámetro  a la pila*/

                if  not cierreParentesisAntes(s, i)  then
                    parametro <- guardar_parametro(expresión, i);
                    push(<parametro, “param”>, p);
                fi
                 /* como encontre ')' , si antes hay ‘)’ entonces llegué al último parámetro
                   de una función y no hay que agregar ese parámetro a la pila porque ya está allí*/
               /* ahora junto todos os parámetros en un array */
                 k := 0;
                 mientras type(top (p)) == “param”
                    parámetros[k] := first(pop(p));
                   k ++;
                 fin mientras;
               parámetros[k] := “/0”;  /* para indicar que aquí se acaban los parámetros */
               nombre_función := first(pop(p));
               parametro_=  toString(evaluar (nombre_funcion, parametros ));
               /* evaluar llama la funciòn de nombre en nombre_función con los parámetros */
               push(<parametro, “param”>, p);
 fin mientras
end
```

```
fun cierreParentesisAntes(s: string, i: integer) return Boolean
    k: integer; /* usada para recorrer s*/
   /* antes de posicion i puede haber espacios en blanco  y hay que saltearlos*/
   for (j:= i-1 ; whitespace(s[j])  ; j--) do
      k := j;
  od
  return (s[k] == ‘)’);
end

fun reconocer(i: integer, s: string) return integer
  integer n:= length(s);
  for (j:= i+1 ; j <= n ; j++)
    if s[j]  in {‘(‘, ’)’, ’,’, ’/0’} then return j fi
  od
end

fun guardar_nombre_función(s: string, i: integer) return string
  integer  k; /* usado para encontrar donde comienza nombre de función */
  integer t; /* usado para generar resultado */
  string r; /* para guardar nombre de función resultado */
  /* identifico posicion donde debería comenzar nombre de function */
  for (j:= i-1 ; (s[j]  != ‘,’) and (j != -1) ; j--) do
      k := j;
  od
  /* k termina en posicion luego de coma o en posicion 0 si no hay coma */
  /* pero a partir de k puede haber espacios en blanco;  me salto esos espacios en blanco si los
hay */
  for (j = k; whiteSpace(s[j]);  j++)  k++ od
  /* ahora sí k comienza donde comienza nombre function; ahora construyo resultado; asumo
que en nombre de función no hay espacios en blanco; por lo tanto, llegar a espacio en blanco
significa terminar nombre de función*/
  t := 0;
  for (j = k;  j < i and not whitespace(s[j]), j++) r[t] := s[j]; t++;  od
  r[t] = ‘/0’;
  return r;
end
```

```
fun guardar_parametro(s: string, i: integer) return string
  integer  k; /* usado para encontrar donde comienza nombre de parámetro */
  integer t; /* usado para generar resultado */
  string r; /* para guardar nombre de parámetro */
  /* identifico posición donde debería comenzar nombre de parámetro */
  for (j:= i-1 ; (s[j]  != ‘,’) and (s[j]) != ‘(‘ ; j--) do
      k := j;
  od
  /* k termina en posicion luego de coma o de paréntesis que abre */
  /* pero a partir de k puede haber espacios en blanco; me salto esos espacios en blanco si los
hay */
  for (j = k; whiteSpace(s[j]);  j++)  k++ od
  /* ahora sí k comienza donde comienza nombre de parámetro; ahora construyo resultado;
asumo que en nombre de parámetro puede haber espacios en blanco*/
  t := 0;
  for (j = k; j < i, j++) r[t] := s[j]; t++; od
  r[t] = ‘/0’;
  return r;
end
```

Si *E* es el esquema asociado a la función de nombre *functName*, *F* es el esquema que resulta de reemplazar en *E* los nombres de los parámetros de la función de nombre *functName* por los esquemas de los argumentos respectivos en *argSchemas*, entonces eval(functName, argSchemas) retorna evalgenerico(*F*).

En otras palabras: en *F* podemos tener strings que son parte del esquema xml y otros strings que son llamadas a función; dichas llamadas a función pueden tener a su vez funciones anidadas; o sea que en *F* podemos tener anidamientos de funciones. Se utilizará un eval genérico para resolver funciones anidadas.

En el ejemplo de local:sumaNueva aqui solamente tenemos una llamada a función, y al ejecutar el eval solo llamara a typemasgeneral obteniendo el esquema xml de local:sumaNueva , en este ejemplo, no tuvimos que usar el eval genérico.

¿Cómo obtener *E*? hay que usar devolver(nombre_funcion).

¿Cómo obtener *F*? los parámetros de la función de nombre *functName* se obtienen por medio de *NP* = devolverParametrosFuncion(functName) ; a esos nombres de parámetros se asignan los esquemas se los argumentos *argSchemas.* Así para obtener *F* hay que reemplazar en *E* todas las ocurrencias de los elementos de *NP* por sus esquemas de argumentos respectivos en *argSchemas*.

```
fun eval( functName: string,  argSchemas: string) return string
   string fs = devolver(functName);
  /* fs al final va  a contener esquema de functName donde nombres de parametros han sido
     sustituidos por esquemas de argumentos */
  string ass = argSchemas; /*esquemas de argumentos*/
  string paramNames = devolverParametrosFunción(functName); /* parámetros de funcName */
  string PN; /* nombre de parámetro */
  string schema; /*esquema de argumento */
  int k;  /* para recorrer esquemas de argumentos */
  int i;   /* para recorrer ass y para recorrer fs*/
  int r; /* para recorrer paramNames y marcar posicion en fs de fin de string */
  int j /* guarda longitude de fs al final */
  string t; /* para almacenar términos y strings */
  string result; /* para construir resultado */

  if argSchemas <> “” then
    /* hay esquemas de argumentos */
    /* Se reemplazan en fs esquemas de argumentos por nombres de parámetros respectivos */
    k := 0;  i := 0; r:= 0;
    repeat
       /* calcula esquema de argumento siguiente */
       i = reconocer2(ass, i);
       schema := guardarEsquemaArgumento(ass, i)
      /* calcula siguiente nombre de parámetro de función */
      r := reconocer3(paramNames, r);
      PN :=   guardarParámetro(paramNames, r);
      /* se reemplaza en fs  nombre de parámetro de función obtenido por nombre de esquema
        de argumento calculado */
       string-replace(fs, paramName, schema);
       k++;
    until (pss[i] == ‘/0’)
  fi
  /* fs contiene strings y terminos intercalados. El resultado va a tener copiados los string tal
    cual y en lugar de los términos el resultado de su evaluación */
  i:= 0;
  j :=  longitud(fs);
  mientras i < j
     i  =  reconocer4(fs, i);
     /* a partir de posición i obtiene posición de primer ‘(‘ de próximo término
        y si no hay dicho término entonces obtiene la posición del final de fs */
    if (fs[i] == ‘(‘) then
         /* antes de ese paréntesis hay un nombre de función y antes de dicho nombre de función
            puede haber string. */
        /* Ese string si lo hay se calcula y añade al resultado */
           /* va a posición inicio de  nombre de función */
         r := comienzo_nombre_función(fs, i);
         if (r <> 0) then /* función no comienza en posicion 0*/
           if (fs[r-1] <> ´)´) then t :=  guardar_string(fs, r -1) fi
           /* en posicion r-1 no finalize término, por lo que hay string previo */
         fi
           /* guarda string que finaliza en posición r-1 */
           result := result + t;
        fi
        /* Luego se calcula el término, y su evaluación se añade al resultado */
           /* retorna posición de último carácter de término que incluye posición i */
```

```
          i := ir_fin_término(fs,i);
             /* guarda termino que inicia en posición r y termina en posicion i */
          t := guardar_término(fs, r, i);
          result := result + evalgenerico(t);
     else
         t := guardar_string(fs,i); /* guarda string que finaliza en posición i */
         result := result +  t;
     fi
  fin mientras
  return result;
end
```

Ahora explicamos las funciones en las que se apoya eval.

devolver(nombre_funcion:string) return string
retorna el esquema de la función de nombre dado por el parámetro nombre_funcion, y se trata del esquema para la declaración de la función de nombre nombre_función.

devolverParametrosFuncion**(** QName **) = (** QName[1], ... ,QName[i])
devolverParametrosFuncion va devolver los nombres de los parámetros que tiene el nombre función QName que toma de una estructura de datos que contiene toda la información necesaria y en la cual va a buscar la respuesta.(i.e. saca la información de asociarParametroFuncion)

Hay un módulo se generan todas las informaciones para todas las funciones built- in (o sea, para cada función built-in: los nombres de sus parámetros y el esquema asociado a la función built-in).

```
fun reconocer2(s: string , i: integer) return integer
  integer n:= length(s);
  for (j:= i+1 ; j <= n ; j++)
    if s[j]  in {';', '/0'} then return j fi
  od
end
```

```
fun guardarEsquemaArgumento(s: string, i: integer) return string
   /* a la izquierda de i hay esquema de argumento */
  integer  k; /* usado para encontrar donde comienza esquema de argumento*/
  integer t; /* usado para generar resultado */
  string r; /* para guardar nombre de argumento */
  /* identifico posicion donde debería comenzar esquema de argumento */
  for (j:= i-1 ; (s[j]  != ';') and j >= 0 ; j--) do
      k := j;
  od
  /* k termina en posicion luego de ';' anterior o  k = 0*/
  /* pero a partir de k puede haber espacios en blanco; me salto esos espacios en blanco si los
hay */
  for (j = k; whiteSpace(s[j]);  j++)  k++ od
  /* ahora sí k comienza donde comienza esquema de argumento; ahora construyo resultado;
asumo que en esquema de argumento puede haber espacios en blanco*/
  t := 0;
  for (j = k; j < i, j++) r[t] := s[j]; t++; od
  r[t] = '/0';
  return r;   end
```

```
fun reconocer3(s: string , i: integer) return integer
  integer n:= length(s);
  for (j:= i+1 ; j <= n ; j++)
    if s[j]  in {',', '/0'} then return j fi
  od
end

fun guardarParametro(s: string, i: integer) return string
   /* a la izquierda de i hay nombre de parámetro */
  integer  k; /* usado para encontrar donde comienza parámetro*/
  integer t; /* usado para generar resultado */
  string r; /* para guardar nombre de parámetro */
  /* identifico posicion donde debería comenzar nombre de parámetro */
  for (j:= i-1 ; (s[j]  != ',') and j >= 0 ; j--) do
      k := j;
  od
  /* k termina en posicion luego de ',' anterior o  k = 0*/
  /* pero a partir de k puede haber espacios en blanco; me salto esos espacios en blanco
     si los hay */
  for (j = k; whiteSpace(s[j]);  j++)  k++ od
  /* ahora  k comienza donde comienza nombre de parámetro; ahora construyo resultado; asumo
que en nombre de parámetro no puede haber espacios en blanco */
  t := 0;
  for (j = k; j < i and not whitespace(s[j]), j++) r[t] := s[j]; t++; od
  r[t] = '/0';
  return r;
end

fun reconocer4(s: string , i: integer) return integer
  integer n:= length(s);
  for (j:= i+1 ; j <= n ; j++)
    if s[j]  in {'(', '/0'} then return j fi
  od
end

fun comienzo_nombre_función(s, i) return integer
  int j; /* para recorrido de s */
  /*en s[i] hay '('  y antes hay nombre de función, y antes del nombre de función puede haber o
    no más caracteres (si no hay más caracteres, se llega a posición 0 de s) */
  /*por lo tanto, vamos a ir a posición donde comienza nombre de función*/
  /*Asumimos que nombre de función no contiene espacios en blanco y antes de nombre de
    función si hay caracteres  va a haber algún carácter especial (espacio en blanco,
    retorno de carro, ';', etc.) hay que entender bien qué posibilidades hay en total; yo pondría
    conjunto de  posibilidades y lo iria aumentando conforme a las necesidades */
  /* se va a posición donde comienza nombre de función */
  for (k = i-1; not s[k] in {' ', ';', \n}´and k >= 0 , k--) j: = k;
  return j;
end
```

```
fun guardar_string(s: string, i: integer) return string
    /* a la izquierda de i hay string */
   integer  k; /* usado para encontrar donde comienza string*/
   integer t; /* usado para generar string del resultado */
   string r; /* para guardar el string */
   /* identifico posicion donde debería comenzar el string*/
   /* La idea es que el string llega hasta donde finaliza término (o sea, carácter ´)' o hasta
        posicion 0 si antes no hay término) */
   for (j:= i ; (s[j]  <> ')') and j >= 0 ; j--) do
        k := j;
   od
   /* k termina en posicion luego de ')' anterior o  k = 0 */
   If k <> 0 then k:= k +1; fi
   /* ahora  k comienza donde comienza string; ahora construyo resultado */
   t := 0;
   for (j = k; j < i, j++) r[t] := s[j]; t++; od
   r[t] = '/0';
   return r;
end

fun guardar_termino(s: string, init: integer, end: integer) return string
    /* en posición init  de s inicia término y en posicion end de s se finaliza término*/
   integer t; /* usado para generar término del resultado */
   string r; /* para guardar el término */
  /*ahora construyo resultado */
   t := 0;
   for (j = init; j < end + 1, j++) r[t] := s[j]; t++; od
   r[t] = '/0';
   return r;
end

fun ir_fin_término(s,i) return integer
   /* en posicion i de s hay '(' que es el primer paréntesis del término a saltar */
   /* la idea es ir al paréntesis que cierra ese paréntesis que abre. Asumimos que va a existir
      dicho paréntesis; y antes de él puede haber varios paréntesis más */
   int cantCerrar = 1;  /* cantidad de paréntesis a cerrar */
   int k; /* para reorrer s*/
   for (j := i + 1;  cantCerrar <> 0; j++ ) do
      if s[j] == ´(' then cantCerrar++;  fi
      if s[j] == ´)' then cantCerrar—; fi
     k = j;
   od
   return k;
end
```

Ejemplo
declare function local:sumaNueva($a as integer, $b as decimal) as decimal { $a + $b };
<group name = “sumaNueva”>
<sequence>
<group ref=”a” />
<group ref=”b” />
asociarParametroFuncion(sumaNueva,a,b);
asociar(sumaNueva, typemasgeneral(a,b));
</sequence>
</group>
asociarParametroFuncion(sumaNueva, a, b) crea asignación del estilo sumaNueva → la lista de a y b. Se puede hacer esto porque esta función se usa en el attribute grammar para la producción de declaración de función.
sumaNueva(4,3.5).schema =
eval(devolver(suma nueva),” <element name=”nombreTipo” type=”integer”>,
<element name=”nombreTipo” type=”decimal”> “)

= eval(typemasgeneral(a,b), ” <element name=”nombreTipo” type=”integer”>,
<element name=”nombreTipo” type=”decimal”> “

=
eval (typemasgeneral (
<element name=”nombreTipo” type=”integer”>,
<element name=”nombreTipo” type=”decimal”>
)
)
= <element name=”nombreTipo” type=”decimal”> ?

devolverParametrosFuncion**(** sumaNueva **) = (a, b)**
asociarParametroFuncion**(**sumaNueva**,a**, **b)**
asociarParametroFuncion asocia el nombre de la función sumaNueva con sus parámetros

Esquemas asociados a funciones built-in

Para las funciones built-in descritas en [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)], no va haber declaración de función, va haber un módulo donde para cada función built-in se genera todas las informaciones para todas las built- in , va ser un módulo que va tener toda la información necesaria de las built-in.

Definición de typemasGeneral
TYPE es un tipo numérico
fun typemasgeneral(s :string) return string
local:*typemasGeneral*(
<xs:element name="nombreType" type="xs:TYPE"/>) = <xs:element name="nombreType" type="xs:TYPE"/>

fun typemasgeneral(s :string, s1:string) return string
local:*typemasGeneral*(
<xs:element name="nombreType" type="xs:TYPE"/> ,<xs:element name=”nombreType" type="xs:TYPE"/>
) = <xs:element name="nombreType" type="xs:TYPE"/>

local:*typemasGeneral*(
<xs:element name="nombreType" type="xs:decimal"/> ,<xs:element name=”nombreType"
type="xs:integer"/>
) = <xs:element name="nombreType" type="xs:decimal"/>
local:*typemasGeneral*(
<xs:element name="nombreType" type="xs:integer"/> ,<xs:element name=”nombreType"
type="xs:decimal"/>
) = <xs:element name="nombreType" type="xs:decimal"/>

local:*typemasGeneral*(
<xs:element name="nombreType" type="xs:float"/> ,<xs:element name=”nombreType"
type="xs:integer"/>
) = <xs:element name="nombreType" type="xs:float"/>
local:*typemasGeneral*(
<xs:element name="nombreType" type="xs:integer"/> ,<xs:element name=”nombreType"
type="xs:float"/>
) = <xs:element name="nombreType" type="xs:float"/>

local:*typemasGeneral*(
<xs:element name="nombreType" type="xs:double"/> ,<xs:element name=”nombreType"
type="xs:integer"/>
) = <xs:element name="nombreType" type="xs:double"/>
local:*typemasGeneral*(
<xs:element name="nombreType" type="xs:integer"/> ,<xs:element name=”nombreType"
type="xs:double"/>
) = <xs:element name="nombreType" type="xs:double"/>

local:*typemasGeneral*(
<xs:element name="nombreType" type="xs:double"/> ,<xs:element name=”nombreType"
type="xs:float"/>
) = <xs:element name="nombreType" type="xs:double"/>
local:*typemasGeneral*(
<xs:element name="nombreType" type="xs:float"/> ,<xs:element name=”nombreType"
type="xs:double"/>
) = <xs:element name="nombreType" type="xs:double"/>

CASO un choice y un tipo
* *local: typemasGeneral(*
“<xsd:choice>
<element name=” name of element ” type=”decimal”>
<element name=” name of element ” type=”integer”>
</xsd:choice>”, “ <element name=” name of element ” type=”decimal”>”)
= *local: typemasGeneral(*
<element name=” name of element ” type=”decimal”>”, “ <element name=” name of element ”
type=”decimal”> “)
* *local: typemasGeneral(*
“<xsd:choice>
<element name=” name of element ” type=”float”>
<element name=” name of element ” type=”integer”>
</xsd:choice>”, “ <element name=” name of element ” type=”float”> “)
= *local: typemasGeneral(*

“ <element name=” name of element ” type=”float”>”, “ <element name=” name of element ” type=”float”>”)
* *local: typemasGeneral(*
“<xsd:choice>
<element name=” name of element ” type=”double”>
<element name=” name of element ” type=”integer”>
</xsd:choice>”, “ <element name=” name of element ” type=”double”>”)
= *local: typemasGeneral(*
“<element name=” name of element ” type=”double”>”, “ <element name=” name of element ” type=”double”> “)
* *local: typemasGeneral(*
“<xsd:choice>
<element name=” name of element ” type=”float”>
<element name=” name of element ” type=”double”>
</xsd:choice>”, “ <element name=” name of element ” type=”float”>”)
= *local: typemasGeneral(*
“<element name=” name of element ” type=”double”>”, “ <element name=” name of element ” type=”float”>”)
* *local: typemasGeneral(*
“<xsd:choice>
<element name=” name of element ” type=”double”>
<element name=” name of element ” type=”float”>
</xsd:choice>”, “ <element name=” name of element ” type=”float”> “)
= *local: typemasGeneral(*
<element name=” name of element ” type=”double”>”, “ <element name=” name of element ” type=”float”>”)
* *local: typemasGeneral(*
“<xsd:choice>
<element name=” name of element ” type=”double”>
<element name=” name of element ” type=”float”>
</xsd:choice>”, “<element name=” name of element ” type=”integer”>”)
= *local: typemasGeneral(*
“<element name=” name of element ” type=”double”>”, “ <element name=” name of element ” type=”integer”>”)
* *local: typemasGeneral(*
“ <xsd:choice>
<element name=” name of element ” type=”TYPE”>
<element name=” name of element ” type=”TYPE”>
</xsd:choice>”, “ <element name=” name of element ” type=”TYPE”> “)
= *local: typemasGeneral(*
“<element name=” name of element ” type=”TYPE”>, “<element name=” name of element ” type=”TYPE”>”)
=”<element name=” name of element ” type=”TYPE”>”

Definiciones de esquemas de algunas funciones built-in

Funciones generales y operadores en secuencias
fn:boolean($arg as item()***) as xs:boolean**. { fn:boolean($arg as item()*) .schema = <xsd:element name="nombreType" type="xs:boolean"/> }

fn:empty($arg as item()***) as xs:boolean** . { fn:empty($arg as item()*) .schema = <xsd:element name="nombreType" type="xs:boolean"/> }

fn:exists($arg as item()***) as xs:boolean** . { fn:exists($arg as item()*).schema = <xsd:element name="nombreType" type="xs:boolean"/> }
fn:index-of($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType**) as xs:integer*** . {
fn:index-of($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType).schema = <xsd:element name="nombreType" type="xs:integer"/> }
fn:index-of($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType, $collation as xs:string**) as xs:integer*** . { fn:index-of($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType, $collation as xs:string).schema = <xsd:element name="nombreType" type="xs:integer"/> }

fn:name Devuelve el nombre de un nodo, como una xs: cadena que es la cadena de longitud cero, o tiene la forma léxica de una xs: Qname. Si se omite el argumento, el valor predeterminado es el **elemento de contexto (.)**. El comportamiento de la función si se omite el argumento es exactamente el mismo que si el elemento de contexto se hubiera pasado como argumento.
fn:name Devuelve el nombre del nodo de contexto o el nodo especificado como una xs: cadena.
fn:name() **as xs:string** . { fn:name( )**.schema = <xsd:element name="nombreType" type="xsd:string"/>** }

fn:name($arg as node()?**) as xs:string** .
{ fn:name($arg as node()?)**.schema = <xsd:element name="nombreType" type="xsd:string"/>** }
fn:local-name Devuelve el nombre local del nodo de contexto o el nodo especificado como un xs: NCName.
fn:local-name() **as xs:string** .
{ fn:local-name($arg as node()?)**.schema = <xsd:element name="nombreType" type="xsd:string"/>** }

fn:local-name($arg as node()?**) as xs:string .**
{ fn:local-name($arg as node()?).**schema = <xsd:element name="nombreType" type="xsd:string"/>** }

fn:node-name Devuelve un QName expandido para tipos de nodos que pueden tener nombres. Para otros tipos de nodos, regresa la secuencia vacía. Si $ arg es la secuencia vacía, se devuelve la secuencia vacía.
fn:node-name($arg as node()?) as xs:QName? .
{ **node-name**(**$arg as node()?**)**.schema = <xsd:element name="nombreType" type="xs:QName"/>** }

Funciones sobre strings
fn:concat($arg1 as xs:anyAtomicType?, $arg2 as xs:anyAtomicType?, ...) **as xs:string .**
{ fn.concat($arg1 as xs:anyAtomicType?, $arg2 as xs:anyAtomicType?, ...) **.schema =** <xsd:element name="nombreType" type="xsd:string"/> }

fn:string-length() **as xs:integer** . { fn:string-length().schema = <xs:element name="nombreType" type="xs:integer"/> }

fn:string-length($arg as xs:string?**) as xs:integer** .
{ fn:string-length($arg as xs:string?) .schema = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:string-join($arg1 as xs:string*, $arg2 as xs:string) **as xs:string .**

{ fn:string-join($arg1 as xs:string*, $arg2 as xs:string).schema = <xs:element name="nombreType" type="xs:string"/> }

fn:substring($sourceString as xs:string?, $startingLoc as xs:double**) as xs:string** .
{ fn:substring($sourceString as xs:string?, $startingLoc as xs:double).schema = <xs:element name="nombreType" type="xs:string"/> }

fn:substring($sourceString as xs:string?, $startingLoc as xs:double, $length as xs:double**) as xs:string .**
{ fn:substring($sourceString as xs:string?, $startingLoc as xs:double, $length as xs:double).schema **=** <xs:element name="nombreType" type="xs:string"/> **}**

fn:upper-case($arg as xs:string?**) as xs:string. {** fn:upper-case($arg as xs:string?) .schema **=** <xs:element name="nombreType" type="xs:string"/> **}**

fn:lower-case($arg as xs:string?**) as xs:string.{** fn:lower-case($arg as xs:string?**)**.schema **=** <xs:element name="nombreType" type="xs:string"/> **}**

Funciones de extracción de componentes en duraciones, fechas y horas
fn:years-from-duration($arg as xs:duration?**) as xs:integer? . {** fn:years-from-duration($arg as xs:duration?).**schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:months-from-duration($arg as xs:duration?**) as xs:integer? . {** fn:months-from-duration($arg as xs:duration?).**schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:days-from-duration($arg as xs:duration?) **as xs:integer**?.{**fn:days-from-duration**($arg as xs:duration?).schema = <xs:element name="nombreType" type="xs:integer"/>}
fn:hours-from-duration($arg as xs:duration?**) as xs:integer**? **. {** fn:hours-from-duration($arg as xs:duration?)**.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:minutes-from-duration($arg as xs:duration?**) as xs:integer**? **. {** fn:minutes-from-duration($arg as xs:duration?) **.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**
fn:seconds-from-duration($arg as xs:duration?**) as xs:decimal**? **. {** fn:seconds-from-duration($arg as xs:duration?)**.schema** = <xs:element name="nombreType" type="xs:**decimal**"/> **}**

fn:year-from-dateTime($arg as xs:dateTime?**) as xs:integer**?**. {** fn:year-from-dateTime($arg as xs:dateTime?)**.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:month-from-dateTime($arg as xs:dateTime?**) as xs:integer**? **. {** fn:month-from-dateTime($arg as xs:dateTime?)**.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:day-from-dateTime($arg as xs:dateTime?**) as xs:integer**? **. {** fn:day-from-dateTime($arg as xs:dateTime?) **.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:hours-from-dateTime($arg as xs:dateTime?**) as xs:integer**? **. {** fn:hours-from-dateTime($arg as xs:dateTime?)**.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:minutes-from-dateTime($arg as xs:dateTime?**) as xs:integer**? **. {** fn:minutes-from-dateTime($arg as xs:dateTime?) **.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:seconds-from-dateTime($arg as xs:dateTime?**) as xs:decimal**? **. {** fn:seconds-from-dateTime($arg as xs:dateTime?)**.schema** = <xs:element name="nombreType" type="xs:**decimal**"/> **}**

fn:year-from-date($arg as xs:date?**) as xs:integer**? **. {** fn:year-from-date($arg as xs:date?)**.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:month-from-date($arg as xs:date?**) as xs:integer**?**. {** fn:month-from-date($arg as xs:date?)**.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:day-from-date($arg as xs:date?**) as xs:integer**?**. {** fn:day-from-date($arg as xs:date?)**.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:hours-from-time($arg as xs:time?**) as xs:integer**? **. {** fn:hours-from-time($arg as xs:time?)**.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:minutes-from-time($arg as xs:time?**) as xs:integer**? **. {** fn:minutes-from-time($arg as xs:time?)**.schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:seconds-from-time($arg as xs:time?**) as xs:decimal**? **. {** fn:seconds-from-time($arg as xs:time?)**.schema** = <xs:element name="nombreType" type="xs:**decimal**"/> **}**

Funciones y operadores en duraciones, fechas y horas
op:yearMonthDuration-less-than($arg1 as xs:yearMonthDuration, $arg2 as xs:yearMonthDuration**) as xs:boolean . {**
op:yearMonthDuration-less-than($arg1 as xs:yearMonthDuration, $arg2 as xs:yearMonthDuration) **.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:yearMonthDuration-greater-than($arg1 as xs:yearMonthDuration, $arg2 as xs:yearMonthDuration**) as xs:boolean . {** op:yearMonthDuration-greater-than($arg1 as xs:yearMonthDuration, $arg2 as xs:yearMonthDuration) **.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:dayTimeDuration-less-than($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration**) as xs:boolean . {** op:dayTimeDuration-less-than($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration) **.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:dayTimeDuration-greater-than($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration**) as xs:boolean . {** op:dayTimeDuration-greater-than($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:duration-equal($arg1 as xs:duration, $arg2 as xs:duration**) as xs:boolean . {** op:duration-equal($arg1 as xs:duration, $arg2 as xs:duration) **.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**
op:dateTime-equal($arg1 as xs:dateTime, $arg2 as xs:dateTime**) as xs:boolean . {**
op:dateTime-equal($arg1 as xs:dateTime, $arg2 as xs:dateTime)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:dateTime-less-than($arg1 as xs:dateTime, $arg2 as xs:dateTime**) as xs:boolean . {** op:dateTime-less-than($arg1 as xs:dateTime, $arg2 as xs:dateTime)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:dateTime-greater-than($arg1 as xs:dateTime, $arg2 as xs:dateTime**) as xs:boolean . {** op:dateTime-greater-than($arg1 as xs:dateTime, $arg2 as xs:dateTime)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:date-equal($arg1 as xs:date, $arg2 as xs:date**) as xs:boolean . {** op:date-equal($arg1 as xs:date, $arg2 as xs:date)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:date-less-than($arg1 as xs:date, $arg2 as xs:date**) as xs:boolean . {** op:date-less-than($arg1 as xs:date, $arg2 as xs:date)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:date-greater-than($arg1 as xs:date, $arg2 as xs:date**) as xs:boolean . {** op:date-greater-than($arg1 as xs:date, $arg2 as xs:date)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:time-equal($arg1 as xs:time, $arg2 as xs:time**) as xs:boolean . {** op:time-equal($arg1 as xs:time, $arg2 as xs:time) **.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:time-less-than($arg1 as xs:time, $arg2 as xs:time**) as xs:boolean . {** op:time-less-than($arg1 as xs:time, $arg2 as xs:time)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:time-greater-than($arg1 as xs:time, $arg2 as xs:time**) as xs:boolean . {** op:time-greater-than($arg1 as xs:time, $arg2 as xs:time)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:gYearMonth-equal($arg1 as xs:gYearMonth, $arg2 as xs:gYearMonth**) as xs:boolean . {** op:gYearMonth-equal($arg1 as xs:gYearMonth, $arg2 as xs:gYearMonth) **.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:gYear-equal($arg1 as xs:gYear, $arg2 as xs:gYear**) as xs:boolean . {** op:gYear-equal($arg1 as xs:gYear, $arg2 as xs:gYear)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:gMonthDay-equal($arg1 as xs:gMonthDay, $arg2 as xs:gMonthDay**) as xs:boolean . {** op:gMonthDay-equal($arg1 as xs:gMonthDay, $arg2 as xs:gMonthDay)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:gMonth-equal($arg1 as xs:gMonth, $arg2 as xs:gMonth**) as xs:boolean . {** op:gMonth-equal($arg1 as xs:gMonth, $arg2 as xs:gMonth)**.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

op:gDay-equal($arg1 as xs:gDay, $arg2 as xs:gDay**) as xs:boolean . {** op:gDay-equal($arg1 as xs:gDay, $arg2 as xs:gDay) **.schema** = <xs:element name="nombreType" type="xs:**boolean**"/> **}**

Funciones de agregación

fn:**min**($arg as xdt:anyAtomicType*) as xdt:anyAtomicType?
$arg, secuencia de elementos de los cuales se devolverá el valor mínimo
fn:min($arg as xdt:anyAtomicType***) as xdt:anyAtomicType**? **.{**
fn:min($arg as xdt:anyAtomicType*)**.schema** = *typemasGeneral(* arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema) **}**
fn:min($arg as xs:anyAtomicType*, $collation as string**) as xs:anyAtomicType**? **. {**
fn:min($arg as xs:anyAtomicType*, $collation as string).**schema** = *typemasGeneral(* arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema)
}

fn:max($arg as xs:anyAtomicType*) as xs:anyAtomicType?
$arg, secuencia de elementos de los cuales se devolverá el valor máximo
fn:max($arg as xs:anyAtomicType***) as xs:anyAtomicType**? **. {**
fn:max($arg as xdt:anyAtomicType*).**schema** = *typemasGeneral(* arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema) **}**

fn:max($arg as xs:anyAtomicType*, $collation as string**) as xs:anyAtomicType**?**.{**fn:max($arg as xs:anyAtomicType*, $collation as string).**schema** = *typemasGeneral(* arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema) **}**

fn:avg($arg as xs:anyAtomicType*) as xs:anyAtomicType?
$arg, secuencia de elementos de los cuales se devolverá el valor promedio
fn:avg($arg as xs:anyAtomicType* **) as xs:anyAtomicType**? **. {** fn:avg($arg as xs:anyAtomicType*).**schema** =
typemasGeneral(arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema) **}**

fn:count($arg as item()*) as xs:integer
$arg, secuencia de elementos de los cuales se devolverá el número de items de la secuencia.
Returns the number of items in the value of $arg.
fn:count($arg as xs:anyAtomicType* **) as xs:integer . {**
fn:*count* ($arg as xs:anyAtomicType*).**schema** = <xs:element name="nombreType" type="xs:integer"/> **}**

fn:sum($arg as xs:anyAtomicType*) as xs:anyAtomicType
$arg, secuencia de elementos de los cuales se devolverá el número de ítems de la secuencia.
fn:sum($arg as xs:anyAtomicType* **) as xs:anyAtomicType . {**
fn:*sum* ($arg as xs:anyAtomicType*).**schema** =
typemasGeneral(arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema **) }**

fn:sum($arg as xs:anyAtomicType*, $zero as xs:anyAtomicType?**) as xs:anyAtomicType**? **.{**
fn:sum($arg as xs:anyAtomicType*, $zero as xs:anyAtomicType?).**schema** =
typemasGeneral(arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema **) }**

Con los esquemas de las funciones builtin armamos un modulo utilisando las funciones asociar y asociarParametroFuncion.

Para las funciones built-in tenemos Funciones generales y operadores en secuencias
asociar("fn:boolean","<xsd:element name="nombreType" type="xs:boolean"/>")
asociarParametroFuncion("fn:boolean", "arg");

asociar("fn:empty","<xsd:element name="nombreType" type="xs:boolean"/> ")
asociarParametroFuncion("fn:empty", "arg");

asociar("fn:exists","<xsd:element name="nombreType" type="xs:boolean"/>")
asociarParametroFuncion(“fn:exist”, “arg”);

asociar("fn:index-of_2","<xsd:element name="nombreType" type="xs:integer"/>")
asociarParametroFuncion(“fn:index-of_2”, “seqParam*”,”*srchParam*”);*

asociar("fn:index-of_3","<xsd:element name="nombreType" type="xs:integer"/>")
asociarParametroFuncion(“fn:index-of_3”, “seqParam*”,”*srchParam*”,”collation”);*

asociar("fn:name_0","<xsd:element name="nombreType" type="xsd:string"/>")
asociarParametroFuncion(“fn:name_0”, “ ”);

asociar("fn:name_1","<xsd:element name="nombreType" type="xsd:string"/>")
asociarParametroFuncion(“fn:name_1”, “arg”);

asociar("fn:local-name_0","<xsd:element name="nombreType" type="xsd:string"/>")
asociarParametroFuncion(“fn:local-name_0”, “”);

asociar("fn:local-name_1","<xsd:element name="nombreType" type="xsd:string"/>")
asociarParametroFuncion(“fn:local-name”, “arg*”);*

asociar("fn:node-name","<xsd:element name="nombreType" type="xsd:QName"/>")
asociarParametroFuncion(“fn:node-name”, “arg*”);*

Funciones sobre strings
asociar("fn:concat","<xsd:element name="nombreType" type="xsd:string"/>")
asociarParametroFuncion(“fn:concat”, “arg1*”,”arg2”, ...);*

asociar("fn:string-length_0","<xs:element name="nombreType" type="xs:integer"/>")
asociarParametroFuncion(“fn:string-length_0”, “ ”);

asociar("fn:string-length_1","<xs:element name="nombreType" type="xs:integer"/>")
asociarParametroFuncion(“fn:string-length_1”, “arg*”);*

asociar("fn:string-join","<xs:element name="nombreType" type="xs:string"/>")
asociarParametroFuncion(“fn:string-join*”,* “arg1*”,”arg2”);*

asociar("fn:substring_2","<xs:element name="nombreType" type="xs:string"/>")
asociarParametroFuncion(“fn:substring_2*”,* “sourceString*”,”*startingLoc*”);*

asociar("fn:substring_3","<xs:element name="nombreType" type="xs:string"/>")
asociarParametroFuncion(“fn:substring_3*”,* “sourceString*”,”*startingLoc*”,”*length *”);*
asociar("fn:upper-case","<xs:element name="nombreType" type="xs:string"/>")
asociarParametroFuncion(“fn:upper-case*”,* “arg*”);*

asociar("fn:lower-case","<xs:element name="nombreType" type="xs:string"/>")
asociarParametroFuncion(“fn:lower-case*”,* “arg*”);*

Funciones de extracción de componentes en duraciones, fechas y horas
asociar("fn:years-from-duration","<xs:element name="nombreType" type="xs:integer"/>")
asociarParametroFuncion(“fn:years-from-duration*”,* “arg*”);*

asociar("fn:months-from-duration"," <xs:element name="nombreType" type="xs:integer"/>")

*asociarParametroFuncion(“*fn:months-from-duration*”, “*arg*”);*

asociar("fn:hours-from-duration","<xs:element name="nombreType" type="xs:integer"/>")
*asociarParametroFuncion(“*fn:hours-from-duration*”, “*arg*”);*

asociar("fn:minutes-from-duration","<xs:element name="nombreType" type="xs:integer"/>")
*asociarParametroFuncion(“*fn:minutes-from-duration*”, “*arg*”);*

asociar("fn:seconds-from-duration","<xs:element name="nombreType" type="xs:decimal"/>")
*asociarParametroFuncion(“*fn:seconds-from-duration*”, “*arg*”);*

asociar("fn:year-from-dateTime","<xs:element name="nombreType" type="xs:integer"/>")
*asociarParametroFuncion(“*fn:year-from-dateTime*”, “*arg*”);*

asociar("fn:month-from-dateTime","<xs:element name="nombreType" type="xs:integer"/> ")
*asociarParametroFuncion(“*fn:month-from-dateTime*”, “*arg*”);*

asociar("fn:day-from-dateTime","<xs:element name="nombreType" type="xs:integer"/>")
*asociarParametroFuncion(“*fn:day-from-dateTime*”, “*arg*”);*

asociar("fn:hours-from-dateTime","<xs:element name="nombreType" type="xs:integer"/>")
*asociarParametroFuncion(“*fn:hours-from-dateTime*”, “*arg*”);*

asociar("fn:minutes-from-dateTime","<xs:element name="nombreType" type="xs:integer"/>")
*asociarParametroFuncion(“*fn:minutes-from-dateTime*”, “*arg*”);*

asociar("fn:seconds-from-dateTime","<xs:element name="nombreType" type="xs:decimal"/>")
*asociarParametroFuncion(“*fn:seconds-from-dateTime*”, “*arg*”);*

asociar("fn:year-from-date","<xs:element name="nombreType" type="xs:integer"/>")
*asociarParametroFuncion(“*fn:year-from-date*”, “*arg*”);*

asociar("fn:month-from-date","<xs:element name="nombreType" type="xs:integer"/>")
*asociarParametroFuncion(“*fn:month-from-date*”, “*arg*”);*

asociar("fn:days-from-duration","<xs:element name="nombreType" type="xs:integer"/>")
*asociarParametroFuncion(“*fn:days-from-duration*”, “*arg*”);*

asociar("fn:day-from-date","<xs:element name="nombreType" type="xs:integer"/>")
*asociarParametroFuncion(“*fn:day-from-date*”, “*arg*”);*

asociar("fn:hours-from-time","<xs:element name="nombreType" type="xs:integer"/>")
asociar("fn:minutes-from-time","<xs:element name="nombreType" type="xs:integer"/>")
asociar("fn:seconds-from-time","<xs:element name="nombreType" type="xs:decimal"/>")

Funciones y operadores en duraciones, fechas y horas
asociar("op:yearMonthDuration-less-than","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:yearMonthDuration-greater-than","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:dayTimeDuration-less-than","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:dayTimeDuration-greater-than","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:duration-equal","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:dateTime-equal","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:dateTime-less-than","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:dateTime-greater-than","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:date-equal","<xs:element name="nombreType" type="xs:boolean"/>")

asociar("op:date-less-than","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:date-greater-than","<xs:element name="nombreType" type="xs:boolean"/> ")
asociar("op:time-equal","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:time-less-than","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:time-greater-than","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:gYearMonth-equal"," <xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:gYear-equal","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:gMonthDay-equal","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:gMonth-equal","<xs:element name="nombreType" type="xs:boolean"/>")
asociar("op:gDay-equal","<xs:element name="nombreType" type="xs:boolean"/>")

Funciones de agregación

asociar("fn:min_1","typemasGeneral(arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema)")
asociar("fn:min_2","typemasGeneral(arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema)")
asociar("fn:max_1","typemasGeneral(arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema)")
asociar("fn_max_2","typemasGeneral(arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema)")
asociar("fn:avg","typemasGeneral(arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema)")
asociar("fn:count"," <xs:element name="nombreType" type="xs:integer"/> ")
asociar("fn:sum_1","typemasGeneral(arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema) ")
asociar("fn:sum_2","typemasGeneral(arg[1].schema, arg[2].schema, arg[3].schema, ..., arg[i].schema) ")

4.2.5. Mapeo a esquemas XML de expresiones XPath.

Para obtener el esquema xml de una expresión XPath, vamos a tener como contexto un esquema xml de los datos (con datos me refiero al documento xml) donde la expresión xpath se aplica. El esquema de los datos sobre la que se va hacer la búsqueda, no es el documento xml de los datos, es el esquema de los datos de ese documento xml, los datos sobre que hacemos la consulta respeta un esquema xml, ese esquema xml lo vamos a tener que usar para devolver los resultados de la expresión XPath.Se supone que está definido un esquema xml del documento xml de entrada para procesar la expresión XPath; entonces tenemos un archivo del esquema xml como dato de entrada (.xsd).
Para poder procesar una expresión xpath se va necesitar acceder al archivo para procesarlo tantas veces como se necesite y esto es costoso en tiempo. Por eso se decidió guardar el contenido de cada línea del archivo en un arreglo, donde cada fila del arreglo va contener la línea del archivo.xsd (es mucho más rápido el acceso a memoria que a disco), al arreglo lo vamos llamar arreglo1: rray[1..n] of string.
Existe un problema: sucede que nada impide escribir expresiones de camino que no tienen sentido porque contradicen el esquema XML de los datos; en esos casos, va a ser necesario descubrir esta inconsistencia y para hacer esto sería necesario recorrer tanto del arreglo1 y como de la expresión XPath siendo considerada.
Debido al uso de referencias y construcciones como <group>, <extension>, etc. en el esquema XML del documento, puede ser necesario recorrer varias veces el arreglo1 para encontrar inconsistencia entre el esquema XML y la expresión de camino. El recorrido del arreglo 1 va a ser intrincado y el algoritmo que lo hace va a ser complicado.

En caso de no haber inconsistencias en la expresión XPath y el esquema XML de los datos, también va a ser necesario recorrer el arreglo1 para generar el esquema XML resultante de la expresión XPath. Debido al uso de referencias y construcciones como <group>, <extension>, etc. en el esquema XML del documento, puede ser necesario recorrer varias veces el arreglo1 para generar el esquema XML correspondiente a la expresión XPath. Nuevamente, el recorrido del arreglo 1 va a ser intrincado y el algoritmo que lo hace va a ser complicado.

Además el esquema XML resultante de procesar la expresión XPath se espera que sea usado para generar elementos de interfaz de usuario y por ese motivo debe ser fácil de procesar, y para ello debe ser lo más sencillo posible; con esto queremos decir, que el esquema XML resultante de la expresión XPath no puede tener referencias, ni etiquetas como <group>, <attributegroup>, <extension>.

Para facilitar tanto el recorrido del esquema XML de los datos como la generación del esquema XML resultante de la expresión XPath decidimos generar el árbol del esquema XML de los datos. Entonces se usará este árbol para ver si la expresión XPath es consistente con el esquema XML de los datos y para generar el esquema XML resultante de la expresión XPath.

Entonces a partir del arreglo1 se va a generar el árbol del esquema XML; para esto vamos a usar un tipo abstracto de datos llamado árbol general; un árbol general es un árbol n-ario, o sea, de un nodo pueden colgar *n* subárboles. Un elemento del esquema XML puede tener n subelementos anidados dentro del mismo; por eso hablamos de árbol n-ario.
Entonces, si mi expresión de caminos termina por ejemplo con un /c y tiene sentido lo que escribo, una idea es buscar directamente en el arreglo1 un elemento de nombre c y doy como salida el esquema XML asociado a c suponiendo que ese elemento se encuentra completo en forma contigua; entonces para hacer eso no necesitaría trabajar con un árbol general; pero la realidad es mucho más complicada, porque no siempre se va tener un resultado para una expresión de camino; el resultado puede dar vacio porque mi expresión de caminos no tiene sentido; además el elemento c puede estar definido no contiguamente debido a referencias, etiquetas <group>, <extension>, etc.

Se puede generar el árbol del esquema XML (cuando se recorre el arreglo1) de más de una manera: en profundidad, en amplitud, etc.
Aquí se lo hace en profundidad; además una optimización es que cuando se genere un nodo *n* se generarán a continuación todos sus hijos de *n* (antes de continuar con la generación en profundidad).

Para generar el árbol del esquema XML de los datos vamos a necesitar buscar muchos objetos en el arreglo1 (i.e. elementos, atributos, definiciones simpleType, definiciones complexType, etc.) de un modo que va a obligar a recorrer muchas veces el arreglo1 (debido a que durante la generación en profundidad necesito buscar un nodo en el arreglo 1 más de una vez y a que en el arreglo 1 puede haber referencias y etiquetas <group>, <extend>, etc. que hacen que necesite ir de un lugar a otro muy distante en el arreglo 1).
Para evitar (i.e. minimizar) este problema decidimos utilizar un segundo arreglo, llamado arreglo 2. Dicho arreglo se va a usar como un índice alfabético a las definiciones de elementos, de atributos, y de tipos en el XML esquema de los datos (i.e. definiciones de simpleType y de complexType) presentes en el arreglo 1.
Por ejemplo, si quiere encontrar la definición de un elemento *E* cuyo nombre comienza con la letra 'b' entonces hay que buscar array2[b] y me va a dar la lista de nombres que comienzan con 'b'; cada uno de esos nombres va a estar seguido del índice o posición en el arreglo 1 donde están las definiciones correspondientes. En el arreglo 2 se distinguen atributos de otro tipo de objetos (i.e. elementos, simpleType, complexType) y los nombres de atributo se preceden de '@'.
El arreglo 2 no solo va a servir para generar el árbol del esquema XML de datos, sino que también va a servir para construir el esquema XML asociado a una expresión XPath. Con el arreglo 2 no es necesario recorrer el arreglo1 para encontrar un elemento o atributo en el arreglo 1. Solo es necesario recorrer el arreglo 1 una vez para construir el arreglo 2.
Entonces para generar el árbol general vamos a utilizar el arreglo1 y el arreglo2.

arreglo2: array[a..z] of string.

El siguiente ejemplo ilustra los contenidos del arreglo 1 y el arreglo 2 para un esquema XML de datos.

El arreglo1

```
1<?xml version="1.0"?>
2<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:simpleType name="string255">
4      <xs:restriction base="xs:token">
5         <xs:maxLength value="255"/>
6      </xs:restriction>
7   </xs:simpleType>
8   <xs:simpleType name="string32">
9      <xs:restriction base="xs:token">
10        <xs:maxLength value="32"/>
11     </xs:restriction>
12  </xs:simpleType>
13  <xs:simpleType name="isbn">
14     <xs:restriction base="xs:NMTOKEN">
15        <xs:length value="10"/>
16     </xs:restriction>
17  </xs:simpleType>
18  <xs:simpleType name="supportedLanguages">
19     <xs:restriction base="xs:language">
20        <xs:enumeration value="en"/>
21        <xs:enumeration value="es"/>
```

```
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="name" type="string32"/>
  <xs:element name="qualification" type="string255"/>
  <xs:element name="born" type="xs:date"/>
  <xs:element name="dead" type="xs:date"/>
  <xs:element name="isbn" type="isbn"/>
  <xs:element name="nombrecompleto" type="string"/>
  <xs:element name="correo" type="string"/>
  <xs:element name="ubicacion" type="string"/>
  <xs:element name="telefono" type="string"/>
  <xs:element name="clasificacion" type="string"/>
  <xs:element name="edad" type="int"/>
  <xs:attribute name="id" type="xs:ID"/>
  <xs:attribute name="available" type="xs:boolean"/>
  <xs:attribute name="lang" type="supportedLanguages"/>
  <xs:element name="title">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="string255">
          <xs:attribute ref="lang"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="book" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="author">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="persona"/>
        <xs:element ref="born"/>
        <xs:element ref="dead" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute ref="id"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="isbn"/>
        <xs:element ref="title"/>
        <xs:element ref="author" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="character" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute ref="id"/>
      <xs:attribute ref="available"/>
    </xs:complexType>
  </xs:element>
```

```
77   <xs:element name="character">
78      <xs:complexType>
79         <xs:sequence>
80            <xs:element ref="name"/>
81            <xs:element ref="born"/>
82            <xs:element ref="qualification"/>
83         </xs:sequence>
84         <xs:attribute ref="id"/>
85      </xs:complexType>
86   </xs:element>
87   <xs:group name="elementosPersona">
88    <xs:sequence>
89       <xs:element ref="nombrecompleto"/>
90       <xs:element ref="correo" minOccurs="0" maxOccurs="unbounded"/>
91       <xs:element ref="ubicacion"/>
92       <xs:element ref="telefono" minOccurs="0" maxOccurs="2"/>
93       <xs:element ref="clasificacion"/>
94       <xs:element ref="edad"/>
95    </xs:sequence>
96   </xs:group>
97   <xs:complexType name="datosDePersona">
98     <xs:group ref="elementosPersona"/>
99     <xs:attribute name="id"  type="xs:integer" use="required"/>
100     <xs:attribute name="visible"  type="xs:boolean" use="optional"/>
101   </xs:complexType>
102   <xs:element name="persona" type="datosDePersona"/>
103 </xs:schema>
```

El arreglo2

a author 54 @available 36
b born 26
c character 71 correo 30 clasificación 33
d dead 27 datosDePersona 97
e elementosPersona 87 edad 34
f
g
h
i isbn 28, simpleType 13 @id 35
j
k
l lang 37 library 47
m
n nombrecompleto 29 name 24
o
p persona 102
q qualification 25
r
s string255 3 string32 supportedLanguages 18
t telefono 32 title 38
u ubicacion 31
v
w
x
y
z

4.2.5.1. Generación del árbol del esquema XML de datos

Ahora vamos a dar un algoritmo que computa el árbol del esquema XML de datos. Pero antes vamos a especificar algunas funciones usadas por dicho algoritmo.

Algunas funciones usadas en el algoritmo

crear_arbolgeneral(e:string) return t:arbolgeneral, construye un árbol a partir de un nombre de elemento o atributo.
insertar_hijo_izq(h1:arbolgeneral, t:arbolgeneral) return r:arbolgeneral
inserta un hijo a un árbol que no tiene hijos
insertar_hermano_derecho(hd:arbolgeneral, t:arbolgeneral) return r:arbolgeneral
Inserta un hermano a la derecha en un árbol que no tiene hermanos a la derecha
fun concatenar (string1:string, string2:string) return string1:string
  string1=string1+string2
end
Concatenación de strings (function strcat en C).

convertir_int_astring(i:int) return string. Convierte un número entero a formato string (función itoa() en C).

buscar(nombre_nodo: string) **return integer**
Retorna la posición en el arreglo 1 donde se encuentra la definición de nombre_nodo
(Para esto hace uso del arreglo 2).

Queremos una función que busca valores de atributos en una línea del arreglo 1. La signatura de la misma es:
devolverNombre(s: string, pos: integer) return string
Esta función toma un string que hace referencia a un nombre de atributo válido de un elemento de XML Schema de datos, una posición del arreglo 1 y retorna el string asociado al atributo de ese nombre en la línea pos.
Por ejemplo:
devolverNombre(“name”, i) busca el atributo de nombre “name” en la línea i del arreglo 1 y devuelve el valor de ese atributo en dicha línea.
Otros ejemplos de uso son:
devolverNombre(“ref”, i) busca el atributo de nombre “ref” en la línea i del arreglo 1 y devuelve el valor de ese atributo.
devolverNombre(“type”, i)
devolverNombre(“base”, i)

```
fun devolverNombre(s: string, pos: integer) return string
  int i; /*para recorrer arreglo1[pos]*/
  int k; /* para recorrer result mientras se lo construye */
  string result; /*resultado*/
  /*retorna posición luego de donde termina s en arreglo1[pos]*/
   i := search(s, arreglo1[pos], 0);
   /*busca posición de ‘“’ que abre luego de s en arreglo1[pos] */
   i := reconocer(i);
   /* construye el resultado */
  k:= 0;
  for (j:= i+1; arreglo1[j] <> ‘””; j++) do
     result[k] := arreglo1[i];
     k++;
  od
```

```
  result[k] := '/0';
  return result;
end

fun reconocer(i: int) return int
   int n:= length(arreglo1[pos]);
   for (j:= i ; j <= n ; j++)
      if  arreglo1[j] == '"' then return j fi
   od
end

fun search (subString: string, Str: string, start: int) return int
    int flag /* bandera para saber si encontré substring */
    int indexOf = -1; /* para guardar resultado */
    int loopEnd = (int)(length(Str)-length(subString)); /*posición última donde buscar*/
    for (int i = start; i <= loopEnd; i++)
    {
       flag = 1;
       /* ahora averigua si subString aparece en Str desde posición i */
       for (int index = 0; index < length(subString); index++){
          if (string[i] != subString[index]){
             flag = 0;
             break;
          }
          else{ i++; }
       }
       if (flag == 1) { break; }
    }
    return indexOf;
end
```

Queremos una función que dada una etiqueta en la línea i del arreglo 1 devuelve la posición siguiente en el arreglo 1 a la posición donde se encuentra la etiqueta de cierre correspondiente. La signatura de dicha función es:
jump(objeto: string, pos: integer) return integer

```
fun jump(objeto: string, pos: integer) return integer{
   /* asumimos que pos es índice en arreglo1 y que objeto aparece en arreglo1 y que cierre de
      objeto también aparece en arreglo1*/
   int cantCerrar = 1; \* representa cantidad de strings objeto a cerrar */
   for( i=pos; cantCerrar <> 0 ; i++ ){
      c2 ← devolver_etiqueta (arreglo1[i] );
      if( c2 == "<" + objeto + ">"){  cantCerrar = cantCerrar +1;  }
      if( c2 == "</" + objeto + " >" ){  cantCerrar = cantCerrar - 1;  }
   }
   return i+1;
}
end
```

Devolver_etiqueta recibe un string que comienza con una etiqueta y retorna esa etiqueta

```
fun devolver_etiqueta (s: string ) return string
/* assume que s comienza con etiqueta XML */
  int i = 0; /* para recorrer s */
  int k; /* para recorrer el resultado */
  string result; /* para guardar el resultado */
  /*encuentra primer '<' */
  for (int j := 0; s[j] <> '<'; j++ ) do i++; od
  /* computa el resultado, teniendo en cuenta que s[i] = '<' */
  k := 0;
  /* copia en result desde s[i] hasta cierre de etiqueta sin incluir */
  for (int j := i ; s[j] <> '>'; j++) do result[k] = s[j]; k++; od
  /* agrega a result cierre de etiqueta y cierre de string */
   result[k] := '>';
   result[k+1] := '/0';
   return result;
end
```

devolver_tipo_de_linea(s: string) return string
Esta función recibe una línea válida de la sintaxis de XML schema que comienza con una etiqueta de XML schema y devuelve un string que indica el nombre dentro de esa etiqueta. Particularmente puede devolver los strings: "element", "attribute", "group", "complextype", "simpletype", "extension", "restriccion", "sequence", "choice"

```
fun devolver_tipo_de_línea(s: string) return string
  /* assume que s comienza con etiqueta XML */
  int i = 0; /* para recorrer s */
  int k; /* para recorrer el resultado */
  string result; /* para guardar el resultado */
  /*encuentra primer '<' */
  for (int j := 0; s[j] <> '<'; j++ ) do i++; od
  /* computa el resultado */
  k := 0;
  for (int j := i + 1; s[j] <> '>'; j++) do result[k] = s[j]; k++; od
   result[k] := '/0';
   return result;
end
```

Definición del algoritmo que genera el árbol general

Recordemos que de acuerdo a la definición del tipo árbol general un nodo puede contener un primer hijo, un hermano a la derecha, y un padre (todos ellos son árboles generales).
Un nodo hoja es aquél donde no hay primer hijo (pero puede haber hermano a la derecha y padre).

La función tratar_nodo toma un nodo hoja *N* del árbol general que está siendo generado y primero expande *N*; esto significa que añade hijos a *N* de acuerdo con el esquema XML presente en el arreglo 1; a continuación tratar_nodo busca el siguiente nodo a expandir y una vez que lo encuentra, lo retorna. En otras palabras se expande *N* si se puede y si no se va al nodo siguiente a expandir.

Ahora bien como dijimos que la generación del árbol general es en profundidad, ir al siguiente nodo a expandir es buscar ese próximo nodo en profundidad. Esto quiere decir que si expandimos N y le agregamos hijos a *N* el próximo nodo va a ser el hijo más a la izquierda

agregado. Si no se agregaron hijos a *N* y *N* tiene hermano a la derecha, entonces el hermano a la derecha de *N* es el próximo nodo; en caso contrario se va a buscar el primer ancestro *A* de *N* que tiene hermano a la derecha *M* y *M* va a ser el nodo siguiente a expandir.

```
fun tratar_nodo( t:arbolgeneral, a:arreglo1 ) return p:arbolgeneral
  expandir_nodo( t, a );
  p ← hijo_izq(t)
  if ¬ is_empty(p)
     then p ← hijo_izq(p)
  else
    if ¬ is_empty( hermano_der( t ) ) //si tiene hermano a la derecha
       p ← hermano_der( t )
    else
       Mientras is_empty( hermano_der( t) ) y not raiz(t)
            t ← padre( t)
       fin mientras
       If not is_empty( hermano_der( t) ) then p ← hermano_der( t )
       else p <- t
    fi
end
```

La generación del árbol general es sencilla: simplemente comenzamos con el elemento raíz cuyo nombre viene dado como parámetro. A partir de ese nombre generamos el nodo raíz del árbol general (es un nodo sin padre ni hermano a la derecha). Luego a partir de ahí se trata el nodo obtenido y se siguen tratando los nodos siguientes hasta volver al nodo raíz del árbol general.

```
proc generacion_arbol (nombre_raíz; in arreglo1: array[1..n] of string,
                              in arreglo2: array[a..z] of string) return nodo: arbolgeneral{
    Nodo := crear_arbolgeneral(nombre_raíz);
    repeat
        nodo ← tratarNodo ( nodo, arreglo1 );
    until  esnodoraiz( nodo )
    return nodo;
end
```

La función expandir_nodo expande un nodo hoja *N* del árbol general actual. Esto significa que busca los hijos de *N* en el arreglo 1 y los agrega como hijos de *N* (para esto se construyen los nodos que hagan falta). La forma en que se expande *N* va a depender del tipo de nodo que es *N* . Por ejemplo, no es lo mismo la expansión de un elemento que de un atributo.

```
int expandir_nodo( nodo: arbolgeneral, arreglo1: array[1..n] of string ){
 if ( esnodoraiz ( nodo ) ){ /* la raíz del árbol general siempre es un elemento */
    m    ← buscar(nodo);
    m1 ← procesarelement( arreglo1,m,nodo);
 } else {
   m    ←  buscar(nodo);
   linea  ←  devolver_tipo_de_linea (arreglo 1[i+1]);
   switch(linea)
     Case linea = element
        m1 ← procesarelement( arreglo1, m, nodo );
     Case linea = attribute
        m1 ← procesarAttribute( arreglo1, m, nodo );
```

```
      Case linea = sequence
         m1 ← procesarSequence( arreglo1, m, nodo );
      Case linea = choice
         m1 ← procesarChoice( arreglo1, m, nodo );
      Case linea = extension
         m1 ← procesarExtension(arreglo,m,nodo);
  return m1;
}
```

Definición de funciones auxiliares para generación del árbol general

```
int procesarAttribute( arreglo1: array[1..n] of string, i:int, t:arbolgeneral){
  var arroba,nombre_nombre,nombre_ref:string
   arroba="@"

  nombre_name ← devolvernombre( " name" , arreglo1[i]);
   if (nombre_name != vacio)  {
      nombre_nombre ←  concatenar(arroba, nombre_name)
      insertar_hijo(nombre_name,t)
    }

   nombre_ref ← devolvernombre( " ref" );
   if (nombre_ref != vacio)  {
   nombre_ref ←  concatenar(arroba, nombre_ref)
    insertar_hijo(nombre_ref, t)
   } return i+1;
}

int procesarelement( arreglo1: array[1..n] of string, i:int, t:arbolgeneral ){ /*estoy buscando los
hijos de element */
m:int
    nombre_type←devolvernombre("type", arreglo[i]);
   If (nombre_type == vacio) {
       c2 ←  devolver_tipo_de_linea ( arreglo1[i+1]);
       switch(c2)
       Case c2="simpleType"
          m ←procesarsimpleType( arreglo1, i+1 );
       Case c2="complexType"
         m ←procesarComplexType( arreglo1, i+1,t );
     } else {
          if(nombre_type == type_basico){
                    m=i+1;
          }
          if( nombre_type == type_def_usuario){
              p1 ←  buscar( nombre_type );
             c2 ←  devolver_tipo_de_linea ( arreglo1[p1] );
       switch(c2)
          Case c2="simpleType"
            m ←procesarsimpleType( arreglo1, i+1 );
          Case c2="complexType"
            m ←procesarComplexType( arreglo1, i+1,t );
          Case c2="element"
            m ←procesarElement( arreglo1, i+1,t );
          }
     }
```

```
    return m;
}

Int procesarSimpleType( arreglo1: array[1..n] of string, i:int ){
 /*Case  s1 =SimpleType :si es el nodo raíz aquí significa que solo tendríamos este nodo en el arbolgeneral. aqui en
este caso si tenemos simpleType con name o sin name:  no tiene hijos para el nodo que estamos buscando , al tener
restricción sin base o restricción con base es solo para nombrar el type base, básico que se le va poner alguna
restricción.*/
  M ← jump (“simpleType”, i);
  return M;
}

int procesarComplexType( arreglo1: array[1..n] of string, i:int, t:arbolgeneral ) {
x,m:int
s:string
x=i+1
Mientras (  devolver_tipo_de_linea (arreglo 1[x]) != </complexType>)  {
c2 ←  devolver_tipo_de_linea ( arreglo 1[x]);
switch(c2)
Case c2 = “attribute”
  x← procesarAttribute(arreglo1, x,t);
Case c2 = “atributeGroup”
  x← procesaratributeGroup(arreglo1,x);
Case c2 = “group”
  x← procesarGroup(arreglo1, x);
Case c2 = “choice”
  x← procesarChoice(arreglo1, x,t);
Case c2 = “sequence”
 x← procesarSequence(arreglo1,x,t);
Case c2 = “simpleContent” /* simpletype se define para agregar atributos a element, en este
caso el complextype esta dentro de un elemento element no tiene attribute name el complextype.
El elemento extensión con el attribute base, donde base es de type basico */
 c3 ←  devolver_tipo_de_linea ( arreglo 1[x+1]);
  switch(c3)
  Case c3= “extension”
    c4 ←  devolver_tipo_de_linea ( arreglo 1[x+2]);
    switch(c4)
    Case c4=”attribute”
    x←procesarAttribute( arreglo1, x+2,t ); x=x+2;
Case c2 = “xs:complexContent”
    c3 ←  devolver_tipo_de_linea ( arreglo1[x+1]);
    switch(c3)
    Case c3 =” extension”
     x = x+2;
    s ← convertir_int_astring(x)
   c3←concatenar(c3,s)
   insertar_hijo(c3,t)
      x=x+1;
     x←jump(extension, x,arreglo1, N);
     Break;
  }
 return x + 1;
}
```

```
int procesar_atributeGroup(arreglo1: array[1..n] of string, i:int,t:arbolgeneral) {
 Mientras (  devolver_tipo_de_linea (arreglo 1[i+1]) != </attributeGroup>)  {
   C1 ←  devolver_tipo_de_linea (arreglo 1[i+1]) ;
   Swith(C1)
    Case C1= “attribute”:
      m ← procesarAttribute( arreglo1, i+1,t  );
     break;
    i++
  }
  return i + 1;
}

int procesar_atributeGroup_ref( arreglo1: array[1..n] of string, i:int, t:arbolgeneral  ) {
nombre_ref ← devolvernombre( “ref” , i);
p1 ←  buscar( nombre_ref );
C1 ←  devolver_tipo_de_linea ( arreglo1[p1]);
Switch (C1) :
  Case  C1 = “atributeGroup” con  name: j=p1
  Mientras (  devolver_tipo_de_linea (arreglo 1[j+1]) != </attributeGroup>) {
    C2 ←  devolver_tipo_de_linea (arreglo1[j+1]) ;
    Switch(C2)
     Case C2 = “attribute”:
        m ← procesarAttribute( arreglo1, j+1,t );
      break;
    j++;
   }
  return i+1;
}

int procesarSequence( arreglo1: array[1..n] of string, i:int, t:arbolgeneral ) {
s:string, x:int
Mientras (  devolver_tipo_de_linea (arreglo 1[i+1]) != </sequence>)  {
 c2 ←  devolver_tipo_de_linea ( arreglo 1[i+1]);
 switch(c2)
 case c2 = “element”:
    nombre_type ← devolvernombre(“type”);
    if(nombre_type == vacio){
      nombre_name ← devolvernombre( “name” );
       if (nombre_name != vacio) {
           insertar_hijo(nombre_name,t)
          i=i+2;
      i=x ← jump(element, i + 1) else  i++;
    } else { /*aqui element es el hijo que estoy buscando, no me importa si el type basico o def por usuario, solo
quiero el nombre del elemento ya que encontre un hijo*/

        nombre_name ← devolvernombre( “name” );
       if (nombre_name != vacio) {
          insertar_hijo(nombre_name,t)
          i=i+2;

       nombre_ref ← devolvernombre( “ ref” );
       if (nombre_ref != vacio)  {
            insertar_hijo(nombre_ref, t)
         i=i+2;
      }
```

```
   }
   break;
Case c2 = “sequence”
   s ← convertir_int_string(i+1)
   c2 ← concatenar(c2,s)
     insertar_hijo(c2,t)
     k ← jump(sequence, arreglo1, i+1);
     break;
Case c2 = “choice”
     s ← convertir_int_string(i+1)
     c2 ← concatenar(c2,s)
     insertar_hijo(c2,t)
    k ← jump(choice, arreglo1, i+1);
 break;
 Case c2 = “group”
  x ← procesarGroup_ref( arreglo1, i+1 )
  }
 return i + 2
}

int procesarChoice( arreglo1: array[1..n] of string, i:int, t:arbolgeneral  ){
s:string
x:int
 Mientras (  devolver_tipo_de_linea (arreglo 1[i+1]) != </choice>)  {
  S:string, x:int
  c3 ←  devolver_tipo_de_linea ( arreglo 1, i);
  switch(c3)
  Case c3=”element”
    nombre_type ← devolvernombre(“type”);
    if(nombre_type == vacio){
      x ← jump(element, i + 1) else  i++;
    } else {
        nombre_name ← devolvernombre( “name” );
       if (nombre_name != vacio) {
        insertar_hijo(nombre_name, t)
          i=i+2;

       nombre_ref ← devolvernombre( “ ref” );
       if (nombre_ref != vacio)  {
            insertar_hijo(nombre_ref, t)
         i=i+2;
       }
    }
    break;
    Case c3=”sequence”
      s ← convertir_int_astring(i+1)
      c3 ← concatenar(c3,s)
       insertar_hijo(c3, t)
      m ← jump(sequence,arreglo1, i);
       break;

  Case c3=”choice”
      s ← convertir_int_astring(i+1)
      c3 ← concatenar(c3,s)
       insertar_hijo(c3, t)
```

```
    pos←jump(choice, arreglo1, i);
    break;

 Case c3 = "group"
  x ← procesarGroup_ref( arreglo1, i+1 )
 }
return i+2
}

int  procesarExtension(arreglo1: array[1..n] of string, i:int, t:arbolgeneral){
m: int;
   nombre_base ← devolvernombre( " base" );
   if(nombre_base == type_basico){
            m=i+1;
   }
   if( nombre_type == type_def_usuario){
     p1 ←  buscar( nombre_type );
     c2 ←  devolver_tipo_de_linea ( arreglo1[p1] );
     switch(c2)
     Case c2="simpleType"
        m ←procesarsimpleType( arreglo1, i+1 );
     Case c2="complexType"
        m ←procesarComplexType( arreglo1, i+1,t );
    }
}

int procesarGroup_name( arreglo1: array[1..n] of string, i:int, t:arbolgeneral ){
 x:int
 c2 ←  devolver_tipo_de_linea ( arreglo1[i+1]);
 switch(c2)
 Case c2="choice":
   x←procesarChoice( arreglo1, i+1, t );
  Case c2="sequence":
  x← procesarSequence( arreglo1, i+1, t ) ;
}

int procesarGroup_ref( arreglo1: array[1..n] of string, i:int  ){
 m,x:int
 nombre_ref ←devolvernombre(" ref");
 m <-buscar( nombre_ref );
  x← procesarGroup_name( arreglo1,m  );
}
```

4.2.5.2 Procesamiento de una expresión XPath

Dado una expresión XPath deseamos obtener los nodos del árbol general que se corresponden con dicha expresión XPath; a esto le llamamos *procesamiento de una expresión XPath.*
Al procedimiento que dado el conjunto de nodos del árbol general que se corresponde con una expresión XPath y el árbol general calcula el esquema XML de datos asociado a dicho conjunto de nodos le llamamos *generación del esquema XML resultante*.

1ra etapa encontrar los nodos que cumplen los pasos de la expresión xpath
Procesar una expresión XPath consiste en procesar los pasos de la expresión XPath de izquierda a derecha. Hay 6 tipos de pasos que vamos a considerar, los cuales están en la siguiente tabla.

Tabla X: Tipos de pasos considerados

//	c
//	@c
/	*
/	@*
/	c
/	@c

Consideramos que la expresión XPath a procesar va a tener la siguiente forma:
/ o // $step_1$ / o // $step_2$ … / o // $step_n$
Paso 1 Paso 2 Paso n
Dicho camino comienza con nodo raíz del árbol.

Procesar una expresión XPath consiste en procesar sus pasos de izquierda a derecha. Al procesar un paso tenemos un conjunto de nodos que es el resultado de procesar el paso anterior. El resultado de procesar un paso puede ser vacío si la expresión XPath contradice esquema XML de los datos. Si el resultado de procesar un paso es vacío entonces no tiene sentido seguir procesando los pasos siguientes. La función que procesa un paso tiene la siguiente signatura:

Usar función procesar_paso(**in** p:paso, **in** t: arbolgeneral, **in** nodos: set of nodos) **return** nodos: set of nodos

Si se procesó el último paso y se obtuvieron nodos resultantes, entonces para esos nodos va a ser necesario generar un esquema XML.

Antes de definir el algoritmo que procesa una expresión XPath y la función procear_paso vamos a definir algunas funciones auxiliares que serán usadas en esos algoritmos.

Funciones auxiliares

Para usar procesar_paso necesitamos extraer los pasos de la expresión XPath uno a uno. Para no hacer esto a cada rato cada vez que se llama procesar_paso vamos a guardar los pasos de una expresión XPath en un arreglo. Para esto usamos la funciòn:
guardo_pasos_reconocidos_enArreglo. El algoritmo de dicha función tiene en cuenta que los pasos pueden estar separados por / o // y que el último paso termina en '/0'.

fun guardo_paso_reconocidos_enArreglo (in expresion: expresionXpath) **return** A:arreglo
var i: int, /* variable auxiliar que me indica la posición en el arreglo A donde se va guardar cada paso */
var p:int, /* variable de posición que me indica el inicio del paso */
var p1:int /* variable de posición que me indica la finalización del paso */
var expr: string /* */
var A:array[50] of string
expr:=expresion
t← longitud(expresion)
j← cantidad_pasos_expresion(expresion)
i=0 p=0 p1=0
Mientras (i<j) {
p←buscar_caracter(expr, p, '/')
p1←buscar_caracter(expr, p,'/')
/* si la función buscar_caracter me devuelve -1 pq no existe una posición para el p1 porque no encontró hay '/' */
if (p1 == -1) **then** { p1 := buscar_caracter(expr, p, '/0') } /*caso el paso comienza / y es el último paso y buscar_caracter no encontro /, entonces busco la posicion del caracter vacio para p1*/
else { if(p1==p+1) then { /*caso reconozco que el paso comienza con doble barra y necesito una nueva posición para p1*/
p1←buscar(expr, p+1,'/')
if (p1 == -1) then { p1 := buscar_caracter(expr, p+1, '/0') } /* caso el paso comienza con doble barra y llego al último paso, no existe una posición para p1, no hay barra para p1, buscar_caracter me devuelve -1, entonces busca la posición del caracter vacio para p1 */
}
}
for(k=p;k<p1; k++)
{
A[i] = A[i] + expr[k];
}
A[i] ='\0';
i++
}
return A;
end

cantidad_pasos_expresion(expr: string) return integer
Devuelve la cantidad de pasos que contienen la expresión XPath.

fun cantidad_pasos_expresion(in expresion: expresionXPath) return j:int
var t:int
expr[]:=expresion
t = longitud(expresion);
j=0
for(i=0;i<t;i++)
if(expresion[i]=='/' and expresion[i+1]!='/')
/* Con este if se verifica que el elemento que sigue no es / es decir si aparece doble barra cuenta como una por ejemplo, la siguiente cadena "//name/@ooooo//@*" a simple vista tenemos 3 pasos, y si se lo analiza cuenta hasta 3

```
por que el índice i se incrementa de a uno y si bien en la primera posición (i=0) no se incrementa j, cuando se pasa a
la siguiente (i=1) si la cuenta.  */
  j++
   fi
  end
  return j;
end
```

Para buscar un carácter en un string a partir de una posición en el string se usa la función buscar_caracter definida abajo.

```
fun buscar_caracter(expr:string, pos:int, chr:character) return int
{   /* las posiciones de mi string, expr comienzan desde cero. busca el caracter chr a partir de la posición pos en la
expresión expr */
 var t:int
 t:=0
 t:=longitud(expresion)
 for(i=pos;i<=t;i++)
 {
     if(chr != '/0')
    {
        if(expr[i] == chr ) then { return i; }
        else{
            if(i==t-1) return (-1)
        }
    } else {  return i; }
 }
}
```

4.2.5.3 Algoritmo de procesamiento de una expresión XPath

Para procesar una expresión XPath se usa una función que recibe una expresión XPath y un árbol general y retorna el conjunto de nodos en el árbol general que respetan la expresión XPath. Dicha función se basa en la función procesar_paso; o sea, los pasos de la expresión XPath se procesan de izquierda a derecha hasta encontrar que una llamada a procesar un paso retorna el conjunto vacío o que se procesó el último paso y se obtuvieron nodos de resultado.

```
fun  procesar_expresion_xpath( in expresion: string, in T: arbolgeneral ) return nodos: set of
nodos
   var i:int
   var nodos: set of nodos
   nodos:=root(T)
   A ← guardo_paso_reconocidos_enArreglo( expresion )
    j ← cantidad_pasos_expresion(expresion)
   for ( i=0; i<j and nodos != empty ; i++)
      nodos ← procesar_paso ( A, i, T, nodos )
   fin for
   return nodos
end
```

Ahora vamos a definir la función procesar paso. Como hay 6 tipos de paso y cada tipo de paso se procesa de una manera diferente, necesitamos identificar el tipo de un paso antes de procesarlo. Para ello vamos a usar la función tipo_del_paso que recibe el string de un paso y retorna su tipo de acuerdo con la tabla de tipos de pasos definida arriba.

```
fun tipo_del_paso(A[i]) return tipo:string    /* siendo el primer elemento de A[i][0]=/  */
{
   if A[i][1] == *  then tipo:=1
   if A[i][1] ==@ and A[i][2]== *   then tipo:=2
   if A[i][1] ==letra(A[i][1]) then tipo:=3    (ejemplo /n donde n pertenece al string /nombre)
   if A[i][1]==@ and A[i][2]==letra(A[i][2]) then tipo:=4
   if A[i][1]==/ and A[i][2]==letra(A[i][2]) tipo:=5
   if A[i][1]==/ and A[i][2]==@ and A[i][3]==letra(A[i][3])  tipo:=6
}
```

La función letra me da la primera letra de un string que comienza con una letra.

```
fun letra(s:string) return character
{    switch(first(s))
          Case first(s)==’a’: return ‘a’
                    ...
          Case first(s)==’z’: return ‘z’
          Case first(s)==’A’: return ‘A’
                    …
          Case first(s)==’Z’: return ‘Z’
          else return ‘/0’
}
```

La función procesar_paso primero reconoce el tipo del paso y luego para cada tipo de paso llama una función de procesamiento de paso de ese tipo (esas funciones comienzan con la palabra *slash*).

```
proc procesar_paso ( in A:arreglo, in i:int, in T:arbolgeneral, in/out nodos: set of nodos )
{  tipo ← Tipo_del_paso (A[i]) ;
   Switch( tipo )
      Case tipo=1   : slash_asterisco (nodos, A[i], i ); break;
      Case tipo=2   :  slash_asterisco_attribute (  nodos, A[i], i ); break;
      Case tipo=3   : slash_nombre_nodo ( nodos, A[i], i); break;
      Case tipo=4   : slash_attribute_nombre_nodo ( nodos, A[i], i); break;
      Case  tipo=5  : slash_slash_nombre_nodo (  nodos, A[i], i); break;
      Case  tipo=6  : slash_slash_attribute_nombre_nodo ( nodos, A[i], i ); break
}
```

Antes de definir las funciones cuyo nombre comienza con slash vamos a definir algunas funciones auxiliares que serán empleadas.

Funciones auxiliares para definir funciones que comienzan con slash

Vamos a necesitar distinguir los nodos que contienen información que son nodos elemento o atributo de los nodos restantes que son nodos choice, sequence y extension; para esto definimos una funciòn booleana que recibe un nodo y retorna verdadero si y sólo si dicho nodo es de información.

```
fun nodo_informacion(t:arbol_general) return b:bool
   if( t->elemento != choice and  t->elemento != sequence and t->elemento != extension )
      b:=true
   else
     b:=false
   fi
end
```

La siguiente función recibe un string que se supone que va a corresponder el nombre de un nodo (i.e. en el campo elemento de dicho nodo) y va a devolver verdadero si dicho nombre corresponde al nombre de un atributo, o sea, comienza con '@'.

```
fun is_atribute( e:string ) return b:bool
   if( e comienza con @)
      b:=true
   else
     b:=false
   fi
end
```

La función siguiente da como resultado la negación del resultado de la anterior.

```
fun is_elemento( e:string ) return b:bool
   if( e not comienza con @ )
      b:=true
   else
     b:=false
   fi
end
```

Una vez que sabemos el tipo del paso para procesar el paso necesitamos el nombre del elemento o atributo que viene luego de los slash. Para eso se usa la función quitar slash que recibe un paso y devuelve el nombre de atributo o elemento en el paso.

```
fun quitar_slash(paso:string) return string
{    expr, var_aux:array[0..r]of character
   s,i,j:int
   j=0
   expr := paso
    s=longitud(expr);
   for(i=0;i<s;i++){
       if(expr[i]!='/')
       {   var_aux[j]:=expr[i]
           j++;
        }
   }
  var_aux[j]:='\0'
    return var_aux
}
```

Funciones que comienzan con slash

Podemos pensar que tenemos un conjunto *S* de nodos resultantes de procesar el paso anterior y ahora queremos procesar un nuevo paso *P*. Si pensamos en un nodo *t* de *S*, entonces procesar el paso *P* para *t* sería buscar comenzando desde *t* en el árbol general nodos que son descendientes de *t* y que respetan *P* - hablamos de descendientes, porque en el árbol general puede haber elementos sequence, choice, extension (por lo tanto, aun cuando se trate de procesar /c con c nombre de elemento, no necesariamente los nodos resultados van ha ser hijos directos de t en el árbol general). Para hacer dicha búsqueda hay que hacer un recorrido dentro del subárbol *T* del árbol general con raíz *t*. Este recorrido se va a hacer en profundidad.

En general una función slash va a procesar todos los nodos resultantes del paso anterior y para cada uno de ellos va a buscar los nuevos nodos que respetan el paso actual. Procesar un nodo *n*

resultante del paso anterior significa hacer una búsqueda en profundidad de nodos que respetan el paso actual en el subárbol *T* del árbol general con raíz *n*. Para hacer dicha búsqueda en profundidad nos vamos a servir de una función dfs_slash que recibe como argumentos un nodo t dentro del subárbol *T* e información de un paso c y devuelve el siguiente nodo del recorrido en profundidad a procesar; la función dfs-slash consiste de dos partes: primero procesa el nodo *t* añadiéndolo a los resultados si respeta el paso c y luego si es necesario busca en profundidad el siguiente nodo a procesar (a veces esto no es necesario, porque con encontrar un resultado alcanza).

Cada función slash tiene particularidades que dependen del tipo de paso. Lo mismo sucede con las funciones dfs_slash.

Para los tipos de pasos /c, /@c, //c, //@c nos basamos en la siguiente restricción de los esquemas XML: para un código xml schema válido debe satisfacer la restricción "Declaración consistente de Elementos": las declaraciones de dos elementos con el mismo nombre ocurren pero tienen que tener el mismo tipo.
Esta restricción implica que dado un nodo *n* a procesar resultante del paso anterior, para dichos tipos de pasos basta con encontrar el primer nodo resultado que respeta el paso y no aporta nada buscar otros nodos que respetan el paso. Esto implica que las funciones dfs-slash terminan cuando encontraron un resultado y no van a buscar el siguiente nodo a procesar (en otras palabras, solo se busca en profundidad el siguiente nodo a procesar si el nodo procesado no es un resultado - i.e. respeta el paso siendo considerado).

Además para las funciones slash hay que distinguir dos casos: si se está en el paso que comienza de la expresión de caminos (paso cero) u otro paso posterior. Los dos casos se tratan de diferente manera, porque si estamos en primer paso de la expresión de camino, entonces solo el nodo raíz del árbol general es parámetro de función slash y dicho nodo puede o no ser resultado de la expresión de camino; en cambio sí estamos en un paso posterior al primer paso, entonces los nodos parámetro de la función slash son nodos que satisfacen el paso anterior de la expresión de camino, y por lo tanto no van a satisfacer el nuevo paso que se está procesando. Estas diferencias hacen que el tratamiento sea muy diferente: si estamos en el primer paso, no se va a usar la función dfs-slash porque solo hay que procesar el nodo raíz del árbol general y no hay que recorrer nada más. En cambio, si estamos en un paso que no es el primero, entonces vamos a tener que procesar un conjunto de nodos *S* y haremos como explicamos previamente (para cada nodo *t* de *S* se va a hacer búsqueda en profundidad en el subárbol de raíz *t* de nodos que respetan el paso actual).

Las funciones dfs_slash para procesar pasos del tipo /c y /@c si al procesar el nodo actual *t* no encontraron un resultado, entonces al buscar el nodo siguiente no van a ir a un hijo izquierdo de t que sea de tipo elemento o atributo, porque si no al hacer eso estaremos considerando //c o //@c, lo que está mal; por lo tanto, habrá que continuar la búsqueda con el hermano derecho de t si lo hay o sino hacer backtracking para encontrar un ancestro t (debajo del nodo en cuyo subárbol estoy buscando nodos resultado que respetan el paso) que tenga hermano derecho (y ese hermano derecho será el siguiente nodo a procesar).

```
fun slash_nombre_nodo ( in nodos: set of nodos, in c:string, stepNr: integer ) return s:set of nodos
{      /* paso="/N" seleccionaría todos los elementos llamados N. Me da el nodo N si es hijo de los nodos */
   c ← quitar_slash(c)
   if  StepNr == 0   then {
      n := choice (nodos);   /* nodos solo tiene el nodo raíz porque estamos en el primer paso */
      if n-> elemento == c then s:={n}; return s;
      else s:= empty; return s;
   } else {
```

```
   for cada  n  ∈ nodos do
       if hijo_izq(n) == null then {s:= empty; return s;}
       else{
            n := hijo_izq(n);
            repeat
                n ← dfs_slashnombrenodo(n,c);
            until (n == null or n ->elemento == c)
            if n!= null then { s:={n}; return(s); }
       }
 od
 s := empty; return s;
}

Se utiliza la abreviatura dfs para Búsqueda en profundidad
fun dfs_slashnombrenodo( t:arbolgeneral, c:string  ) return p:arbolgeneral
{  /*   función para el caso '/c'  */
    if ( nodo_informacion( t ))
    {     if (t->elemento == c ) then
             {p ← t;  return p;}
          else{
               p← hermano_der( t )
               if  not is_empty(p) then return p;
               else{
                  p := t;
                  Mientras  is_empty(  hermano_der(p )  )  and  (not is_raiz(p)) and
                            not nodo_informacion (padre(p)))
                       p← padre( p)
                  fin mientras
                 if  ¬ is_empty( hermano_der(p) )  then  p←  hermano_der( p ); return p;
                 else  p ← null; return p;
                 fi
            }
    }
    if ( not nodo_informacion( t )) then
    {  p ←hijo_izq( t ); return p; }
}

fun slash_attribute_nombre_nodo ( in nodos: set of nodos, in c:string, in stepNr: integer )
     return set of nodos
{  /* << /@N >>  */
   c←quitar_slash(c)
  if stepNr ==0 {
     n=choice(n);
     if ( is_attribute( n->elemento ) ) and ( n->elemento ==c ) ) then s:={n}; return s;
     else s:={} ; return;
  }else{
      for cada n ∈ nodos do
      {
        if is_empty (hijo_izq( n )) then s:= {};
        else {
             n :=  hijo_izq( n );
            repeat
              n ← dfs_slash_attribute_nombre_nodo( n, c )
            until  ( ( n==null ) or ( is_attribute( n->elemento ) and ( n->elemento==c ) )
```

```
            if( n!=null ) then { s:={n}; return s;  }
          }
       }
       s:={}; return s;
}
```

Se utiliza la abreviatura dfs para búsqueda en profundidad

```
fun dfs_slash_attribute_nombre_nodo( t:arbolgeneral, c:string ) return p:arbolgeneral
/*  función para la búsqueda de '/@N', donde c=@N */
 if ( nodo_informacion( t ))
    if  (t->elemento == c ) then p ← t ; return p;
    else{
      p←hermano_der( t)
      if not  is_empty( p ) then return p;
      else
          p := t;
          Mientras is_empty(  hermano_der(p )  )  and  (not is_raiz(p) and not
nodo_información(padre(p)))
                  p← padre( p)
          fin mientras
          if ¬ is_empty( hermano_der( p)  )  then p←  hermano_der( p );
          else p ← null  return (p); /* no se puede seguir recorriendo y no encontré resultado */
          fi
     fi
 }
fi
if (¬nodo_de_informacion(t))
   then p ← hijo_izq(p)
fi
end

fun slash_slash_nombre_nodo ( in nodos: set of nodos, c:string, stepNr: integer) return set of
nodos
   /* << //c >>*/
   c←quitar_slash(c)
 if stepNr == 0  then { /* el paso 0 se llama con nodos igual a la raíz del árbol general*/
      n := choice (nodos); /* nodos solo tiene el nodo raíz porque estamos en el primer paso */
      if n-> elemento == c  then s:={n}; return s;
      else{
         if hijo_izq  (n) == null then s:= {};   /* si no se puede bajar, entonces no hay resultados
         else{
            n := hijo_izq( n);
            repeat
                    n ← dfs_slash_slash_nombre_nodo( n, c)
            until (n == null or  n ->elemento == c )
            if n != null then  {  s:={n}; return(s);  }
         }
         s:= {}; return s;  /* porque se recorrió todo lo que se podía y no se encontró resultado */
      }
 } else {  /* no es el primer paso y nodos contiene resultados del paso anterior */
     for cada n ∈ nodos  do
       if hijo_izq  (n) == null then s:= {};
       else{
            n := hijo_izq( n);
            repeat
```

```
            n ← dfs_slash_slash_nombre_nodo( n, c )
          until (n == null or n ->elemento == c )
          if n != null then { s:={n}; return(s); }
       }
    }
    od
     s := {}; return s;
}

Se agrega el nodo si está en el resultado
fun dfs_slash_slash_nombre_nodo( t:arbolgeneral, c:string ) return p:arbolgeneral
  /* función búsqueda en profundidad de '//c', c=c */
    if ( nodo_informacion( t ) and t->elemento == c )
    then{ p ← t return p; }

   if ( nodo_informacion( t ) and t->elemento != c ) {
    if ( is_atribute( t->elemento ) ) {
       if ¬is_empty( hermano_der( t ) ) { //si tiene hermano a la derecha
          p← hermano_der( t ) ;   return p;
       } else {
            p := t;
            Mientras is_empty( hermano_der( p ) ) and not is_raiz(p)
              p←padre( p)
            fin mientras
            if ¬is_empty( hermano_der( p ) ) then p← hermano_der( p ); return p;
            else p ← null ; return p;
            fi
       }
    }
    if ( is_elemento(t->elemento) ){
         p←hijo_izq(t)
      if is_empty(p) {
        if ¬is_empty( hermano_der( t ) ) { //si tiene hermano a la derecha
           p← hermano_der( t ) ; return p;
        } else {
            p:=t
            Mientras is_empty( hermano_der( p) ) and not is_raiz(p)
              p ←padre( p)
            fin mientras
            if ¬is_empty( hermano_der( p ) ) then p← hermano_der( p ); return p;
            else p ← null; return p;
            fi
        }
      }
   }
}
if ( ¬nodo_de_informacion( t ) )
then p←hijo_izq( t ); return(p);

end
```

```
fun slash_slash_atribute_nombre_nodo( in T:arbolgeneral, in nodos: set of nodos, c:string)
return set of nodos
{  /*  << //@c  >>*/
   c ← quitar_slash(c)
   if stepNr ==0 { /* como es primer paso, nodos es la raíz, que no puede ser atributo
      n=choice(n);
     if hijo_izq  (n) == null then s:= {};   /* si no se puede bajar, entonces no hay resultados
       else{
             n := hijo_izq( n );
             repeat
                n ← dfs_slash_slash_atribute_nombrenodo( n, c )
             until (n == null or  ( is_attribute( n->elemento ) and ( n->elemento==c ) )
             if n != null then  {  s:={n}; return(s);  }
        }
       s:= empty; return s;  /* porque se recorrió todo lo que se podía y no se encontró resultado */
   }else{ /* no es el primer paso y en nodos hay resultados del paso anterior */
      for cada n ∈ nodos do
      {
        if is_empty (hijo_izq( n )) then s:= {}; /* como no se puede bajar, no hay resultados */
        else {
            n :=  hijo_izq( n );
            repeat
                n ← dfs_slash_slash_atribute_nombrenodo(n, c )
            until  ( ( n==null ) or ( is_attribute( n->elemento ) and ( n->elemento==c ) )
            if( n!=null ) then { s:={n}; return s;  }
         }
      }
      s:={}; return s;
}

fun dfs_slash_slash_atribute_nombrenodo( t:arbolgeneral,c:string  ) return p:arbolgeneral

/*  '//@N' , c=@N */
   if ( nodo_informacion( t ) and  t->elemento == c )
      then p ← t  return p
   fi
   if ( nodo_informacion( t ) and  t->elemento != c )
     if ( is_atribute( t->elemento ) )
        if ¬is_empty( hermano_der( t ) ) //si tiene hermano a la derecha
           p← hermano_der( t )
         else
            p := t
            Mientras is_empty( hermano_der( p ) ) and not raiz(p)
              p←padre( p)
            fin mientras
            if ¬is_empty( hermano_der( p) ) then p← hermano_der( p ); return p;
            else p ← null; return p; /* no se puede seguir recorriendo más y no llegué al resultado */
            fi
        fi
     fi
     if ( is_elemento(t->elemento) )
         p←hijo_izq(t)
       if  is_empty(p)
```

```
        if ¬is_empty( hermano_der( t ) ) /*si tiene hermano a la derecha ie hijo derecho t e
hermano derecho de p */
            p← hermano_der( t )
         else
            p := t;
            Mientras is_empty( hermano_der( p ) ) and not raiz(p)
              p ←padre( p)
            fin mientras
            if ¬is_empty( hermano_der( p ) ) then p← hermano_der( p ); return p;
            else p <- null; return p; /* no se puede seguir recorriendo más y no llegué al resultado */
            fi
       fi
     fi
  fi
fi
if ( ¬nodo_de_informacion( t ) )
     then  p←hijo_izq( t )
fi
end
```

Para los pasos del tipo /* y /@* al procesarse por una función slash un nodo *n* del conjunto de nodos argumento de la función slash, se hará una búsqueda en profundidad en el subárbol del árbol general de raíz *n* de aquellos nodos que respetan el paso y dicha búsqueda no se va a detener cuando se encuentre un resultado (i.e. un nodo que respeta el paso), sino que se va a continuar buscando resultados nuevos diferentes a los anteriores (o sea nodos resultado cuyo campo element tiene nombres diferentes a los campos element de los resultados anteriores).

Para los pasos del tipo /* y /@* una función dfs_slash que tiene como argumento un nodo *t* va a agregar dicho nodo a los resultados si respeta el paso; aun cuando *t* pertenece a los resultados va a buscar en profundidad el siguiente nodo a procesar (i.e. a ver si es resultado).

```
fun slash_asterisco( in nodos: set of nodos, in c:string, in stepNr:integer )  return set of nodos

/* <<  /*  >>  */
var s: set of nodos
var r: set of nodos
var f: tupla(set of nodos, arbolgeneral )
   c ← quitar_slash(c)
if( c == " * ")
{
   if ( stepNr == 0 ) then { /* es primer paso y nodos es nodo raíz */
      n:=choice(nodos);
      if (is_element( n->elemento) ) then s:={n}; return s; /* como n es raiz terminó */
      else s:={} ;  return s;
    /* solo por completitud, pero  en este else no se va a entrar porque la raíz es un elemento */
   }  else { /* no es primer paso y elementos de nodos matchean con paso anterior */
      for cada n ∈ nodos do
      {
        if  is_empty( hijo_izq( n ) ) then s:={}; /* de n no se puede obtener resultado */
        else
        {
            n:= hijo_izq( n );
             repeat
```

```
            ( r, n ) ← dfs_slash_asterisco( n, c )
            f:=( r, n )
            s:=s U f.1
          until ( f.2==null or is_attribute( f.2->elemento ) )
        }
      }
       return s;
    }
     s:= {} return s ;
}

fun dfs_slash_asterisco( t:arbolgeneral,c:string ) return (r:set of nodos, p:arbolgeneral)
p:arbolgeneral
if ( c == * )
{
     r ← { }
     if ( nodo_informacion( t ) )
    {    if ( is_elemento(t->elemento) )
         then {
             if ( t ∉ r ) then  r ← r U t;  return ( r, empty)
                 else return ( r, empty)
           }
     }
     if( nodo_informacion(t) and is_atribute(t->elemento ) )
     then  if  ¬ is_empty( hermano_der( t ) ) //si tiene hermano a la derecha
           then { p ← hermano_der( t ); return ( {}, p) }
           else {
              p := t;
              Mientras (is empty (hermano_der(p)) and (not is_raiz(p) and
                          not nodo_informacion(padre(p))))
                    p ← padre( p)
              fin mientras
              if  ¬ is_empty( hermano_der( p) )
              then { p ← hermano_der( p ); return ({}, p) ;}
              else { p := null /* ya se recorrió todo y no se halló resultado y no se puede seguir */
             return ( {}, p )
           }
      if ( ¬ nodo_informacion( t ))
      {       p ← hijo_izq(t)
       return ( {}, p )
      }
}

fun slash_atribute_asterisco( in nodos: set of nodos, in c:string, stepNr:integer )
     return set of nodos
{   /* << /@N >>  */
    var p:arbolgeneral
    var s:set of nodos
    var r:set of nodos
    var f:tupla(set of nodos, arbolgeneral)
    c ← quitar_slash(c)
   if( c == @* )
   {
```

```
    if( stepNr == 0 ) then { /* primer paso y nodos es raíz */
      s:= {}; return s;  /* como la raíz no es atributo retorna vacío si es el primer paso */
    }else{  no es primer paso y en nodos hay resultados de paso anterior
          for cada n ∈ nodos do
          {
            if( is_empty ( hijo_izq( n ) ) ) then s:={}
            else {
                  p<- hijo_izq(n)
                   repeat
                        f ← dfs_slash_atribute_asterisco(p, c );
                        s:=sU f.1
                   until ( f.2==null )
                 }
             }
           }
           return s;
    }
    s:={}; return s;
  }
}

fun dfs_slash_atribute_asterisco( t:arbolgeneral, c:string ) return( r:set of nodos, p:arbolgeneral
){    /*  '/@*' , 'c=@*'        */
  p :arbolgeneral
  r ← { }
   if ( nodo_informacion( t ) )
     if ( is_atribute(t->elemento) )
       then  r ← r U t; return ( r, empty)
     fi
  if( nodo_informacion(t) and is_elemento(t->elemento ) )
    then if ¬ is_empty( hermano_der( t ) ) //si tiene hermano a la derecha
          then{   p ← hermano_der( t )
             return ( { }, p ) }
          else{
            p := t;
            Mientras (is empty (hermano_der(p)) and (not is_raiz(p) and
                        not nodo_informacion(padre(p))))
                 p ← padre( p)
            fin mientras
           if ¬ is_empty( hermano_der( p) )
           then { p ← hermano_der( p )
                 return ( { }, p) }
           else {p ← null; return ( { }, p)}
          /* se recorrió todo y no se encontraron resultados */
    fi
fi
else if ( ¬ nodo_informacion( t ))
      p ← hijo_izq(t)
      return( { }, p)
fi
end
```

4.2.5.4 Generación del esquema XML resultante a partir de un conjunto de nodos y del árbol general

La función generar_esquema_XML recibe un conjunto de nodos en el árbol general correspondientes a atributos o elementos y para cada nodo del argumento retorna el esquema XML asociado al nodo de acuerdo al esquema XML de los datos guardado en el arreglo 1. El esquema XML asociado a un nodo *N* para un elemento *E* consiste no sólo de la información de *E* presente en *N*, sino también de la información de todos los elementos y atributos anidados dentro del elemento *E*. Una forma de obtener el esquema XML asociado a un nodo *N* para un elemento *E* es recorrer el subárbol de raíz *N* del árbol general generando para cada nodo de dicho subárbol la información del esquema XML del resultado que corresponda. Ese recorrido se puede hacer en profundidad.

Tratar un nodo *M* del subárbol de raíz *N* consiste de generar en notación de esquema XML la información asociada a *M* y luego ir al nodo siguiente de acuerdo al recorrido en profundidad (del subárbol de raíz *N*) y este comportamiento es expresado por medio de la función tratar_nodo_a_xml_schema.

Entonces generar_esquema_XML para un nodo *N* va a consistir en llamar tratar_nodo_a_xml_schema comenzando con *N* y seguir llamando esta función hasta que no haya más nodos que recorrer del subárbol de raíz *N*.

```
fun generar_esquema_XML( in nodos: set of nodos,
                                    in arreglo1: array[1..n] of string,
                                    in arreglo2: array[a..z] of string,
                                   ) return xml_schema:string
 s: string.
 xml_schema := “”;
  for n ∈ nodos do
      p := n;
      repeat
          (s, p) :=  tratar_nodo_a_xml_schema(p, arreglo1, arreglo2, n) ;
           xml_schema := Concatenar (Xml_schema, s);
      until (p == null)
      /* Se aplica tratar nodo a xml esquema hasta recorriendo todo lo que cuelga de n
hasta  regresar a n */
    od
   return xml_shema;
end
```

La función *tratar_nodo_a_xml_schema* aplicada a un nodo *t* del subárbol de raíz *n* (presente en el árbol general) consiste de generar en notación de esquema XML la información asociada a *t* y luego ir al nodo siguiente de acuerdo al recorrido en profundidad del subárbol de raíz *n*. La información a generar para el nodo *t* va a depender de su tipo que puede ser un nodo de información (i.e. elemento, o atributo) o no (i.e. choice, sequence o extension); además para generar esa información puede suceder que no se encuentre toda en el nodo t y haya que acceder a la información completa en el arreglo 1; para acceder a la información en el arreglo 1 basta con acceder en el arreglo 2 al nombre presente en el el campo elemento del nodo *t*. Hay que tener cuidado en la parte de ir al nodo siguiente que si se tiene que hacer backtracking se genere la información de cierre de etiquetas de los nodos ancestros de *t* durante el backtracking.

La función *tratar_nodo_a_xml_schema* la función lo que hace es imprimir lo que corresponda e ir al nodo siguiente.

```
fun tratar_nodo_a_xml_schema( in  t:arbolgeneral
                              in arreglo1: array[1..n] of string,
                              in arreglo2: array[a..z] of nombrenodoenlineas,
                              in n:arbolgeneral /* en n comenzó toda la generación de XML
schema */
                             ) return xml_schema:string, p:arbolgeneral
xml_schema, a: string
xml_schema = “”;
/* AHORA SE TRATA NODO t */
a ← t->elemento
if( nodo_informacion( a ) )
    if( is_element(a) ) then{
      if (hijo_izq(t) == null) then { /* Si t no tiene hijos, entonces es un elemento sin atributos ni
subelementos anidados */
          n := buscar (a);
          if (not  typeIn (n)) then {/* caso simpleType */
            xml_schema := concatenar ( xml_schema, arreglo1[n])  ;
            xml_schema := concatenar (xml_schema, simpleTypeFragment(n+1)) ;
          }
          else  xml_schema := concatenar ( xml_schema, arreglo1[n])  ;
      }
      else xml_schema = < element name = a > fi /* t tiene hijos, por lo que hay que seguir
recorriendo */
    fi
    if( is_attribute( a ))
       n ←  buscar( a );
       if (not  typeIn (n)) then /* caso simpleType */
          then  xml_schema := concatenar (xml_schema, simpleTypeFragment(n+1) ;
          else  xml_schema := concatenar ( xml_schema, arreglo1[n])  ;
                /* aquí tengo tengo un atributo con name y type */
   fi
else
  if ( a == choice ) then
     if(nodo_informacion(t->padre))
          xml_schema := concatenar(xml_schema, <xsd:complexType>)
          xml_schema := concatenar(xml_schema, <a>)
       else
          if( t->padre == sequence)
             xml_schema := concatenar(xml_schema, <a>)fi
          if( t->padre == choice)
             xml_schema := concatenar(xml_schema, <a>)fi
          if( t->padre == extension)
              xml_schema := concatenar(xml_schema, <xsd:complexType>)
              xml_schema := concatenar(xml_schema, <a>) fi
        /* no se usa la etiqueta extensión porque no se usa reutilización */
  fi
  if ( a == sequence ) then
       if(nodo_informacion(t->padre))
          xml_schema :=  concatenar(xml_schema, <xsd:complexType>)
          xml_schema := concatenar(xml_schema, <a>)
       else
          if( t->padre == sequence)
              xml_schema := concatenar(xml_schema, <a>)
           if( t->padre == choice)
             xml_schema := concatenar(xml_schema, <a>)
```

```
            if( t->padre == extension)
               concatenar(xml_schema, <xsd:complexType>)
               concatenar(xml_schema, <a>)
   fi
   if ( a == extension)
       if nodo_informacion (t->primer_hijo->element )
          if (is_attribute(t->primer_hijo->elemento) ) then
             a ← t-> primer_hijo -> elemento
             n ←  buscar( a );
             if (not  typeIn (n)) then{ /* caso simpleType */
                xml_schema := concatenar ( xml_schema, arreglo1[n])  ;
                xml_schema := concatenar (xml_schema, simpleTypeFragment(n+1) ;
             }
             else  xml_schema := concatenar ( xml_schema, arreglo1[n])
          fi
       else
          if (t->primer_hijo->element == sequence ) then
            xml_schema := concatenar(xml_schema,<xsd:sequence>) fi
           if (t->primer_hijo->element == choice ) then  xml_schema :=
concatenar(xml_schema,<xsd:choice>) fi
       fi
   fi
fi
/* AHORA SE VA AL NODO SIGUIENTE */
if( not is_empty (hijo_izq(t)))
     p ← hijo_izq(t);
 else
    if ¬is_empty( hermano_der( t ) ) and (t != n)
      /* t no es nodo donde comenzó todo y tiene hermano a derecha //
       p ← hermano_der(t);
    Else
    /* EN BACKTRACKING SE CIERRAN ETIQUETAS QUE CORRESPONDAN */
       p := t;
       Mientras (is_empty( hermano_der( p) ) and (p != n)) do
            /* no debe subir más allá de nodo donde comenzó toda la generación! */
           p←padre( p)
           if nodo_informacion(p)
              /* is_element( p->element ) */
            xml_schema :=  concatenar( xml_schema, </xsd: element> ) /* elemento con hijos */
           else  ¬nodo_informacion(p)
                if (p->elemento == choice)
                   if ( nodo_informacion( p->padre ) )
                       xml_schema :=  concatenar( xml_schema, </ p->elemento > )
                      concatenar( xml_schema, </xsd: complextype > )
                   else
                       if(p->padre == sequence)
                          xml_schema :=  concatenar( xml_schema, </ p->elemento > )

                       fi
                       if(p->padre == choice)
                          xml_schema :=  concatenar( xml_schema, </ p->elemento > )
                       fi
                       if(p->padre == extension)
                            xml_schema :=  concatenar( xml_schema, </ p->elemento > )
```

```
                        xml_schema :=  concatenar( xml_schema, </xsd: complextype > )
                    fi
                fi
            fi
            if (p->elemento == sequence)
                if( nodo_informacion ( p->padre ) )
                    concatenar( xml_schema, </ p->elemento > )
                    concatenar( xml_schema, </ complexType > )
                else
                    if(p->padre == sequence)
                        xml_schema :=   concatenar( xml_schema, </xsd: p->elemento > )
                    fi
                    if(p->padre == choice)
                        xml_schema :=   concatenar( xml_schema, </xsd: p->elemento > )
                    fi
                fi
            fi
        fin mientras
        if  ¬ is_empty( hermano_der( p) )  then  p ← hermano_der( p )
        else p :=  null /* ya se terminó de recorrer todo! */
        fi
    fi
fi
end
```

Conclusiones y trabajo futuro

En este trabajo consideramos el problema de transformar consultas en un lenguaje de consultas a esquemas XML. En particular se construyeron soluciones para los lenguajes SQL y XQuery.

Las transformaciones de lenguajes de consulta a esquemas XML definidas solo consideraron las construcciones más usadas de XQuery y SQL.

Además se consideró el mapeo de esquemas en SQL a esquemas XML; este mapeo tiene la limitación que no se considera la sentencias create domain como parte de esos esquemas SQL.

En particular para el caso de XQuery se resolvieron en detalle los mapeos de expresiones XPath a esquemas XML y de las llamadas de funciones a esquemas XML. Estos temas tienen una solución compleja y solo se dio la idea de la solución a los mismos en [1].

Además de considerarse mapeos de esquemas SQL a esquemas XML se podría haber considerado por completitud mapeos de otros lenguajes a esquemas XML como esquemas JSON de datos, e incluso mapear DTDs (i.e. es un lenguaje para definir esquemas de documentos XML más sencillo que esquemas XML que usa otra sintaxis) a esquemas XML.

Como trabajo futuro se puede considerar la generación de parte de la UI de una aplicación a partir de modelos de requisitos, esquemas XML de datos generados a partir de expresiones de consulta (en SQL, XQuery y otros lenguajes de consulta) y la asociación de dichos esquemas XML de datos a acciones de output de información en los requisitos. Particularmente se puede atacar este problema para sistemas de información de la web y móviles.

Otro problema que se podría considerar para el futuro es el mapeo de mensajes SOAP de respuesta a esquemas XML (SOAP es uno de los protocolos usados para servicios web).

Bibliografía

[1] Groppe, S., Groppe, J., Linnemann, V. How to Determine Output Schemas of XQuery Queries. In 24th British National Conference on Databases, IEEE, 2007.

[2] Vi Tran, Jean Vanderdonckt, Manuel Kolp, Vi Tran, Jean Vanderdonckt, Manuel Kolp. Generating User Interface from Task, User and Domain Models. ICSEA 2009.

[3] Vi Tran , Jean Vanderdonckt , Ricardo Tesoriero , François Beuvens, Systematic generation of abstract user interfaces, Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems, June 25-26, 2012, Copenhagen, Denmark.

[4] Ceri, S., Fraternaly, P., Bongio, A., Brambilla, M., Comai, S., Matera, M. Designing Data-Intensive Web Applications, Morgan Kaufmann, 2003.

[5] SQL tutorial in http://www.w3schools.com/sql/

[6] http://www.w3.org/TR/xquery/

[7] https://www.w3.org/TR/2003/WD-xquery-20030502/

[8] http://www.w3.org/XML/Schema

[9] http://www.w3.org/TR/soap/

[10] http://www.json.org

[11] Tutorial de XML en http://www.w3schools.com/xml/

[12] Flex &bison John Levine O'Reilly

[13] Lex&yacc John Levine O´Reilly, Tony Mason E Doug Brown

[14] XML Schema, Eric van der Vlist, O'Reilly

[15] Xquery, Priscilla Walmsley, O'Reilly