

Capítulo 8

XML: Parte 1

Introduction

- XML: Extensible Markup Language
- Defined by the WWW Consortium (W3C)
- Documents have tags giving extra information about sections of the document
 - E.g. `<title> XML </title> <slide> Introduction ...</slide>`
- **Extensible**, unlike HTML
 - Users can add new tags, and *separately* specify how the tag should be handled for display

Introduction

- The ability to specify new tags, and to create nested tag structures make XML a great way to exchange **data**, not just documents.
 - Much of the use of XML has been in data exchange applications, not as a replacement for HTML
- Tags make data (relatively) self-documenting
 - E.g.

```
<bank>
  <account>
    <account_number> A-101  </account_number>
    <branch_name>    Downtown </branch_name>
    <balance>        500      </balance>
  </account>
  <depositor>
    <account_number> A-101  </account_number>
    <customer_name> Johnson </customer_name>
  </depositor>
</bank>
```

Introduction

- **Motivation**: sharing of *documents* among systems and databases.
- *Well-Formed XML* allows you to invent your own tags.
- *Valid XML* involves a DTD (*Document Type Definition*), a grammar for tags.

XML Motivation

- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
 - Similar in concept to email headers
 - Does not allow for nested structures, no standard “type” language
 - Tied too closely to low level document structure (lines, spaces, etc)
- Each XML based standard defines what are valid elements, using
 - XML type specification languages to specify the syntax
 - DTD (Document Type Descriptors)
 - XML Schema
 - Plus textual descriptions of the semantics
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

Comparison with Relational Data

- **Inefficiency:** tags, which in effect represent schema information, are repeated
- **Better than relational tuples as a data-exchange format**
 - Unlike relational tuples, XML data is self-documenting due to presence of tags
 - Non-rigid format: tags can be added
 - Allows nested structures
 - Wide acceptance, not only in database systems, but also in browsers, tools, and applications

XML bien formado

- Start the document with a **declaration**, surrounded by `<?xml ... ?>` .
 - Se usan atributos `version`, `encoding` y `standalone` y deben estar en ese orden.
 - El atributo `version` es obligatorio
 - Los atributos `encoding` y `standalone` son opcionales
 - Normal declaration is:

```
<?xml version = "1.0" standalone = "yes" ?>
```
 - “Standalone” = “no DTD provided.”
- Si el atributo `standalone` es incluido en la declaración XML, debe ser fijado en `yes` o `no`.
 - `Yes` especifica que el documento existe enteramente en sí mismo sin depender de otros archivos.
 - `No` indica que el documento puede depender en un DTD externo.

XML bien formado

- **Tag**: label for a section of data
 - XML tags are case sensitive.
- **Element**: section of data beginning with `<tagname>` and ending with matching `</tagname>`
- Elements must be properly **nested**
 - Proper nesting
 - `<account> ... <balance> </balance> </account>`
 - Improper nesting
 - `<account> ... <balance> </account> </balance>`
 - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.
- Every document must have a single **top-level element**

XML bien formado

```
<bank-1>
  <customer>
    <customer_name> Hayes </customer_name>
    <customer_street> Main </customer_street>
    <customer_city>   Harrison </customer_city>
    <account>
      <account_number> A-102 </account_number>
      <branch_name>    Perryridge </branch_name>
      <balance>        400 </balance>
    </account>
    <account>
      ...
    </account>
  </customer>
  .
</bank-1>
```

XML bien formado

- Los **comentarios** proveen una manera de insertar en un documento XML texto que no es parte del documento.
 - Los comentarios son destinados a la gente que está leyendo el código fuente XML.
 - Los comentarios comienzan con `<!--` y terminan con `-->`.
 - No se puede poner un comentario dentro de una etiqueta.

Motivation for Nesting

- Nesting of data is useful in data transfer
 - Example: elements representing *customerId*, *customerName*, and *address* nested within an *order* element
- Nesting is not supported, or discouraged, in relational databases
 - With multiple orders, *customerName* and *address* are stored redundantly
 - normalization replaces nested structures in each order by foreign key into table storing *customerName* and *address* information
- But nesting is appropriate when transferring data
 - External application does not have direct access to data referenced by a foreign key

XML bien formado

- Mixture of text with sub-elements is legal in XML.
 - Example:

```
<account>
  This account is seldom used any more.
  <account_number> A-102</account_number>
  <branch_name> Perryridge</branch_name>
  <balance>400 </balance>
</account>
```
 - Useful for document markup, but discouraged for data representation

XML bien formado

- Elements can have **attributes**

```
<account acct-type = "checking" >  
  <account_number> A-102 </account_number>  
  <branch_name> Perryridge </branch_name>  
  <balance> 400 </balance>  
</account>
```

- Attributes are specified by **name = value** pairs inside the starting tag of an element
- An element may have several attributes, but **each attribute name can only occur once**

```
<account acct-type = "checking" monthly-fee="5">
```

XML bien formado

- El orden en el cual los atributos son incluidos en un elemento no es considerado relevante.
 - Si hay información en un documento XML que debe venir en un cierto orden se debe poner esa información en elementos en lugar de en atributos.
- Elements without subelements or text content can be abbreviated by ending the start tag with a `/>` and deleting the end tag
 - `<account number="A-101" branch="Perryridge" balance="200 />`

Attributes vs. Subelements

- Distinction between subelement and attribute
 - In the context of data representation, the difference is unclear and may be confusing
 - Same information can be represented in two ways
 - `<account account_number = "A-101"> </account>`
 - `<account>`
`<account_number>A-101</account_number> ...`
`</account>`
 - **Suggestion:** use attributes for identifiers of elements, and use subelements for contents

Espacios de Nombres

- XML data has to be exchanged between organizations
- **Problem:** Same tag name may have different meaning in different organizations, causing confusion on exchanged documents
- **Solution 1:** Specifying a unique string as an element name avoids confusion
- **Solution 2:** Better solution: use **unique-name: element-name**
 - Avoid using long unique names all over document by using **XML Namespaces**

```
<bank xmlns:FB='http://www.FirstBank.com'>
```

```
...
```

```
<FB:branch>
```

```
    <FB:branchname>Downtown</FB:branchname>
```

```
    <FB:branchcity> Brooklyn </FB:branchcity>
```

```
</FB:branch>
```

```
...
```

```
</bank>
```


XML Document Schema

- Database schemas constrain what information can be stored, and the data types of stored values
- XML documents are not required to have an associated schema
- However, schemas are very important for XML data exchange
 - Otherwise, a site cannot automatically interpret data received from another site
- Two mechanisms for specifying XML schema
 - **Document Type Definition (DTD)**
 - Widely used
 - **XML Schema**
 - Newer, increasing use

DTD

- The schema of an XML document can be specified using a **DTD**
- DTD constraints structure of XML data
 - What elements can occur
 - What attributes can/must an element have
 - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
 - All values represented as strings in XML

DTD

- Hay dos opciones para definir un DTD:
 - a) Incluir el DTD como un preámbulo del documento XML.
 - b) Seguir DOCTYPE por SYSTEM y un camino al archivo donde el DTD puede ser encontrado.
- la palabra clave **SYSTEM** es seguida por una referencia URI a un documento con una localización física.
- Ejemplo:
 - <!DOCTYPE name SYSTEM "name.dtd" [...]>
 - <!DOCTYPE name SYSTEM "file:///c:/name.dtd" [...]>
 - <!DOCTYPE name SYSTEM "http://sernaferna.com/hr/name.dtd" [...]>

DTD

```
<?xml version = "1.0" standalone = "no" ?>
```

```
<!DOCTYPE BARS [  
  <!ELEMENT BARS (BAR*)>  
  <!ELEMENT BAR (NAME, BEER+)>  
  <!ELEMENT NAME (#PCDATA)>  
  <!ELEMENT BEER (NAME, PRICE)>  
  <!ELEMENT PRICE (#PCDATA)>  
>  
<BARS>
```

The DTD

The document

```
  <BAR><NAME>Joe's Bar</NAME>  
    <BEER><NAME>Bud</NAME> <PRICE>2.50</PRICE></BEER>  
    <BEER><NAME>Miller</NAME> <PRICE>3.00</PRICE></BEER>  
  </BAR>  
  <BAR> ...  
</BARS>
```

DTD

- Assume the BARS DTD is in file bar.dtd.

```
<?xml version = "1.0" standalone = "no" ?>
```

```
<!DOCTYPE BARS SYSTEM "bar.dtd">
```

```
<BARS>
```

```
  <BAR><NAME>Joe's Bar</NAME>
```

```
    <BEER><NAME>Bud</NAME>
```

```
      <PRICE>2.50</PRICE></BEER>
```

```
    <BEER><NAME>Miller</NAME>
```

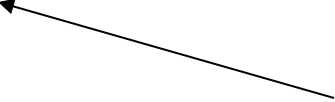
```
      <PRICE>3.00</PRICE></BEER>
```

```
  </BAR>
```

```
  <BAR> ...
```

```
</BARS>
```

Get the DTD
from the file
bar.dtd



DTD

- **Estructura de un DTD**

```
<!DOCTYPE root tag name [  
    <!ELEMENT element (subelements-spec)>  
    . . . more elements . . .  
    <!ATTLIST element (attributes)>  
    . . .  
>
```

- The **description of an element** consists of its name (tag), and a parenthesized description of any nested tags.

DTD: Elements

- **Subelements** can be specified as
 - names of elements, or
 - #PCDATA (parsed character data), i.e., character strings
 - EMPTY (no subelements) or
 - ANY (anything can be a subelement)
- **Example**
 - <! ELEMENT depositor (customer_name account_number)>
 - <! ELEMENT customer_name (#PCDATA)>
 - <! ELEMENT account_number (#PCDATA)>
- Un tag puede ser seguido de un símbolo para indicar su multiplicidad.
 - * : zero or more.
 - + : one or more.
 - ? : zero or one.

DTD: Elements

- Cuando no se usa indicador de multiplicidad significa que el elemento debe aparecer una y solo una vez.
 - Este es el comportamiento por default para elementos usados en modelos de contenido.
- El símbolo | se usa para conectar **secuencias alternativas de tags** (actua como un or exclusivo).
- La especificación de subelementos puede tener **expresiones regulares**

```
<!ELEMENT bank ( ( account | customer | depositor)+)>
```


DTD: Elements

```
<!DOCTYPE bank [  
  <!ELEMENT bank ( ( account | customer | depositor)+)>  
  <!ELEMENT account (account_number branch_name balance)>  
  <! ELEMENT customer(customer_name customer_street customer_city)>  
  <! ELEMENT depositor (customer_name account_number)>  
  <! ELEMENT account_number (#PCDATA)>  
  <! ELEMENT branch_name (#PCDATA)>  
  <! ELEMENT balance(#PCDATA)>  
  <! ELEMENT customer_name(#PCDATA)>  
  <! ELEMENT customer_street(#PCDATA)>  
  <! ELEMENT customer_city(#PCDATA)>  
>
```

DTD: Attributes

- **Especificación de Atributo:**

- Nombre
- Tipo del atributo
 - CDATA: datos de caracteres
 - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
- Un atributo puede ser
 - Obligatorio (#REQUIRED)
 - Tener un valor por default (value),
 - o ninguno de los anteriores (#IMPLIED)

- **Ejemplos**

- `<!ATTLIST account acct-type CDATA "checking">`
- `<!ATTLIST customer`
 `customer_id ID # REQUIRED`
 `accounts IDREFS # REQUIRED >`

IDs and IDREFs

- Un elemento puede tener a lo más un atributo de tipo ID.
- *Los valores del atributo de tipo ID de cada elemento en un documento XML deben ser distintos*
 - El valor del atributo de tipo ID es un identificador de objeto.
- Un atributo de tipo IDREF debe contener el valor ID de un elemento en el mismo documento
- Un atributo de tipo IDREFS contiene un conjunto de (0 o más) valores ID.
 - Cada valor ID debe contener el valor ID de un elemento en el mismo documento.

IDs and IDREFs

- Atributos pueden ser punteros de un objeto a otro.
- Esto permite que la estructura de un documento XML sea un grafo general en lugar de solo un árbol.
- *Para permitir que objetos de tipo F se refieran a otro objeto con un atributo ID, dar a F un atributo de tipo IDREF.*
- O permita que el atributo tenga tipo IDREFS, así el objeto F puede referirse a cualquier número de otros objetos.

IDs and IDREFs

- Bank DTD with ID and IDREF attribute types.

```
<!DOCTYPE bank-2[
  <!ELEMENT account (branch, balance)>
  <!ATTLIST account
    account_number ID      # REQUIRED
    owners          IDREFS # REQUIRED>
  <!ELEMENT customer(customer_name, customer_street,
    customer_city)>
  <!ATTLIST customer
    customer_id      ID      # REQUIRED
    accounts         IDREFS # REQUIRED>
  <!ELEMENT loan (branch, amount)
  <!ATTLIST loan
    loan_id          ID      #REQUIRED
    borrower         IDREF  #REQUIRED
  ... declarations for branch, balance, customer_name,
    customer_street and customer_city
]>
```

IDs and IDREFs

```
<bank-2>
  <account account_number="A-401" owners="C100 C102">
    <branch_name> Downtown </branch_name>
    <balance>      500 </balance>
  </account>
  <customer customer_id="C100" accounts="A-401">
    <customer_name>Joe      </customer_name>
    <customer_street> Monroe </customer_street>
    <customer_city>  Madison</customer_city>
  </customer>
  <customer customer_id="C102" accounts="A-401 A-402">
    <customer_name> Mary    </customer_name>
    <customer_street> Erin   </customer_street>
    <customer_city>  Newark </customer_city>
  </customer>
</bank-2>
```

DTD: Attributes

- **Example:** SELLS elements could have attribute `price` rather than a value that is a price.
- In the DTD, declare:
 <!ELEMENT SELLS EMPTY>
 <!ATTLIST SELLS theBeer IDREF #REQUIRED>
 <!ATTLIST SELLS price CDATA #REQUIRED>
- **Example** use:
 <SELLS theBeer = "Bud" price = "2.50"/>

DTD: Attributes

- **ENUMERATED LIST:** Además de usar los tipos default se puede declarar una lista enumerada de posibles valores para el valor de un atributo.
- **Ejemplo:**
`<!ATTLIST name title (Mr. | Mrs. | Ms. | Miss | Dr. | Rev) #IMPLIED>`
- Se han declarado los valores permitidos entre paréntesis. Todos los posibles valores son separados por '|'.
- Algunos valores válidos del atributo title son:
`<name title="Mr.">`
`<name title="Miss">`
- **Especificar un atributo por default** es fácil; simplemente incluya el valor entre comillas luego del tipo del atributo.
- **Ejemplo:**
`<!ATTLIST name title (Mr. | Mrs. | Ms. | Miss | Dr. | Rev) "Mr.">`

Limitations of DTDs

- No typing of text elements and attributes
 - All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
 - Order is usually irrelevant in databases (unlike in the document-layout environment from which XML evolved)
 - $(A \mid B)^*$ allows specification of an unordered set, but
 - Cannot ensure that each of A and B occurs only once
- IDs and IDREFs are untyped
 - The *owners* attribute of an account may contain a reference to another account, which is meaningless
 - *owners* attribute should ideally be constrained to refer to customer elements

Ejemplo de Referencia

```
<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>
```

Ejemplo de referencia

```
<order num="00299432" date="2006-09-15" cust="0221A">  
  <item dept="WMN" num="557" quantity="1" color="navy"/>  
  <item dept="ACC" num="563" quantity="1"/>  
  <item dept="ACC" num="443" quantity="2"/>  
  <item dept="MEN" num="784" quantity="1" color="white"/>  
  <item dept="MEN" num="784" quantity="1" color="gray"/>  
  <item dept="WMN" num="557" quantity="1" color="black"/>  
</order>
```

Ejemplo de Referencia

```
<prices>
  <priceList effDate="2006-11-15">
    <prod num="557">
      <price currency="USD">29.99</price>
      <discount type="CLR">10.00</discount>
    </prod>
    <prod num="563">
      <price currency="USD">69.99</price>
    </prod>
    <prod num="443">
      <price currency="USD">39.99</price>
      <discount type="CLR">3.99</discount>
    </prod>
  </priceList>
</prices>
```

Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data
- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
 - Element nodes have child nodes, which can be attributes or subelements
 - Text in an element is modeled as a text node child of the element
 - Children of a node are ordered according to their order in the XML document
 - Element and attribute nodes (except for the root node) have a single parent, which is an element node
 - The root node has a single child, which is the root element of the document

XPath

- **XPath** is used to address (select) parts of documents using path expressions
- A **path expression** is a sequence of steps separated by “/”
 - Think of file names in a directory hierarchy
- **Result of path expression:** set of values that along with their containing elements/attributes match the specified path
- **Ejemplo:** obtener los nombres de clientes del banco junto con sus etiquetas contenedoras. (pag. 29)

XPath

- **XPath** is used to address (select) parts of documents using path expressions
- A **path expression** is a sequence of steps separated by “/”
 - Think of file names in a directory hierarchy
- **Result of path expression:** set of values that along with their containing elements/attributes match the specified path
- **Ejemplo:** obtener los nombres de clientes del banco junto con sus etiquetas contenedoras. (pag 29)
 - `/bank-2/customer/customer_name`
 - evaluated on the [bank-2 data](#) we saw earlier returns:
`<customer_name>Joe</customer_name>`
`<customer_name>Mary</customer_name>`

XPath

- `text()` se usa para obtener el **valor texto de un elemento** sin sus etiquetas contenedoras y las de sus subelementos.
- **Ejemplo**: obtener los nombres de los clientes del banco sin sus etiquetas contenedoras

XPath

- `text()` se usa para obtener el **valor texto de un elemento** sin sus etiquetas contenedoras y las de sus subelementos.
- **Ejemplo:** obtener los nombres de los clientes del banco sin sus etiquetas contenedoras
 - `/bank-2/customer/customer_name/text()`

XPath

- Una expresión camino es siempre evaluada relativa a un **ítem de contexto** particular, el cual sirve como el punto de partida para un camino relativo.
 - Algunas expresiones camino comienzan con un paso que fija el ítem de contexto, como en:
`doc("catalog.xml")/catalog/product/number`
 - La llamada de función `doc("catalog.xml")` retorna el nodo documento de `catalog.xml`, el cual se convierte en el ítem de contexto.
- En `$catalog/product/number` el valor de la variable `$catalog` fija el contexto.
 - La variable debe elegir cero, uno o más nodos, los cuales se convierten en ítems de contexto para el resto de la expresión.

XPath

- El ítem de contexto cambia con cada paso.
 - Un paso retorna una secuencia de cero, uno o más nodos que sirven como los ítems de contexto para evaluar el paso siguiente.
 - P.ej. En: `doc("catalog.xml")/catalog/product/number`
 - El paso `doc("catalog.xml")` retorna un nodo documento que sirve como el ítem de contexto cuando se evalúa el paso `catalog`.
 - El paso `catalog` es evaluado usando el nodo documento como el ítem de contexto corriente, retornando una secuencia de un elemento `catalog` hijo del nodo documento.
 - Ese elemento `catalog` sirve como el ítem de contexto para la evaluación del paso `product`, el cual retorna la secuencia de hijos `product` de `catalog`.
 - El paso final, `number` es evaluado para cada hijo `product` en la secuencia.

XPath

- If the descriptor begins with `//`, then the path can start anywhere.
 - `“//”` can be used to skip multiple levels of nodes
- **Ejemplo:** encontrar todos los elementos `customer_name` *en todos lados* debajo del elemento `/bank-2`, sin importar los elementos en los cuales están contenidos.

XPath

- If the descriptor begins with `//`, then the path can start anywhere.
 - `“//”` can be used to skip multiple levels of nodes
- **Ejemplo:** encontrar todos los elementos `customer_name` *en todos lados* debajo del elemento `/bank-2`, sin importar los elementos en los cuales están contenidos.
 - `/bank-2//customer_name`

XPath

- A star (*) in place of a tag represents any one tag.
- **Ejemplo:** encontrar todos los objetos price en el tercer nivel de anidamiento: (pag. 20)

XPath

- A star (*) in place of a tag represents any one tag.
- **Ejemplo:** encontrar todos los objetos price en el tercer nivel de anidamiento: (pag. 20)
 - `/*/*/PRICE`
- **Selection Predicates:**
 - A condition inside [...] may follow a tag.
 - If so, then only paths that have that tag and also satisfy the condition are included in the result of a path expression.
 - Selection predicates may follow any step in a path, in []
 - Boolean connectives **and** and **or** and function **not()** can be used in predicates

XPath

- **Ejemplo:** encontrar elementos account con un valor de balance mayor que 400

XPath

- **Ejemplo:** encontrar elementos account con un valor de balance mayor que 400
 - `/bank-2/account[balance > 400]`
- **Ejemplo:** encontrar elementos account conteniendo un subelemento balance
 - `/bank-2/account[balance]`

XPath

- Los pasos en un camino pueden ser:
 - expresiones primarias como llamadas a función (p.ej. `doc("catalog.xml")`) o
 - referencias a variables (`$catalog`).
 - Toda expresión que retorna nodos puede estar en el lado izquierdo del operador `/`.
- `"."` representa el nodo de contexto corriente en si mismo, sin consideración de su tipo de nodo.
- `".."` representa el nodo padre, el cual puede ser un nodo elemento o el nodo documento.

XPath

- “@” es usada para referirse a atributos.
 - p.ej. @dept elige los atributos de dept .
 - @* elige todos los atributos.
- **Ejemplo:** encontrar los números de cuenta de las cuentas con balance mayor que 400

XPath

- “@” es usada para referirse a atributos.
 - p.ej. @dept elige los atributos de dept .
 - @* elige todos los atributos.
- **Ejemplo:** encontrar los números de cuenta de las cuentas con balance mayor que 400
 - /bank-2/account[balance > 400]/@account_number

XPath

- “@” es usada para referirse a atributos.
 - p.ej. @dept elige los atributos de dept .
 - @* elige todos los atributos.
- **Ejemplo:** encontrar los números de cuenta de las cuentas con balance mayor que 400
 - /bank-2/account[balance > 400]/@account_number

XPath

- The function `count()` at the end of a path counts the number of elements in the set generated by the path
- **Ejemplo:** encontrar las cuentas con más de 2 clientes

XPath

- The function `count()` at the end of a path counts the number of elements in the set generated by the path
- **Ejemplo:** encontrar las cuentas con más de 2 clientes
 - `/bank-2/account[count(./customer) > 2]`
 - recordar que `.` denota el nodo de contexto actual.
- Function `id()`
 - `fn:id($arg as xs:string*) as element()*`
 - Returns the sequence of element nodes that have an ID value matching the value of one or more of the IDREF values supplied in `$arg`
 - `id()` can be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
- **Ejemplo:** encontrar todos los clientes referenciados por el atributo `owners` de los elementos `account`.

XPath

- The function `count()` at the end of a path counts the number of elements in the set generated by the path
- **Ejemplo:** encontrar las cuentas con más de 2 clientes
 - `/bank-2/account[count(./customer) > 2]`
 - recordar que `.` denota el nodo de contexto actual.
- Function `id()`
 - `fn:id($arg as xs:string*) as element()*`
 - Returns the sequence of element nodes that have an ID value matching the value of one or more of the IDREF values supplied in `$arg`
 - `id()` can be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
- **Ejemplo:** encontrar todos los clientes referenciados por el atributo *owners* de los elementos *account*.
 - `/bank-2/account/id(@owner)`

XPath

- El operador “|” es usado para implementar unión
- **Ejemplo:** encontrar los clientes con cuentas o préstamos
 - `/bank-2/account/id(@owner) | /bank-2/loan/id(@borrower)`
- However, “|” cannot be nested inside other operators.

XQuery

- **XQuery (2007)** es un estándar de la W3C que extiende XPath a un lenguaje de consultas que tiene un poder similar a SQL.
- XQuery es un **lenguaje de expresiones**.
 - Una expresión XQuery puede ser una expresión para otra expresión XQuery.
 - XQuery tiene un sistema de tipos sutil.

XQuery: Modelo de Datos

- El **modelo de datos** de XQuery (XDM) tiene las siguientes componentes básicas:
 - **Nodo**: es una construcción XML tales como un elemento o un atributo.
 - **Valor atómico**: es un valor de datos simple.
 - Un **tipo atómico** es un **tipo primitivo simple** o un **tipo derivado** por una restricción de un tipo primitivo simple.
 - Un **valor atómico** es un valor de un tipo atómico.
 - Puede estar etiquetado con ese tipo atómico
 - **Item**: un nodo o un valor atómico.
 - **Secuencia**: una lista ordenada de cero, uno o más ítems.

XQuery: Modelo de Datos

- **Nodos.**
 - **Nodos Elemento** son elementos descritos por declaraciones !ELEMENT en DTDs.
 - **Nodos Atributo** son atributos descritos por declaraciones !ATTLIST en DTDs.
 - **Nodos Texto** = #PCDATA.
 - **Nodos Documento** representan archivos.
- **Valores atómicos:** strings, integers, etc.
 - Algunos **tipos primitivos simples** son:
`xs:string`, `xs:boolean`,
`xs:decimal`, `xs:float`, `xs:double`,
`xs:duration`, `xs:dateTime`, `xs:time`, `xs:date`,
`xs:hexBinary`, `xs:base64Binary`, `xs:anyURI`, etc.
 - También algunos valores contruidos como `true()`,
`date("2004-09-30")`.

XQuery: Valores para un nodo

- Hay dos tipos de **valores para un nodo**: string y tipado.
- Todos los nodos tienen un **valor string**.
 - El **valor string de un nodo elemento** es su contenido de datos carácter y los contenidos de datos carácter de todos sus subelementos descendientes concatenados entre sí.
 - El **valor string de un nodo atributo** es simplemente el valor del atributo.
 - El **valor string de un nodo** puede ser accedido usando la función:
 - `dm:string-value($n as Node) as xs:string`
 - `dm:string-value` retorna una representación string del nodo.
 - Para algunos tipos de nodo, esta es una parte del nodo;
 - para otros tipos de nodos, es computada por medio de `dm:string-value` de sus nodos descendientes.

XQuery: Valores para un nodo

- Nodos de elementos y atributos tienen un **valor tipado** que toma en cuenta su tipo si lo hay.
 - El **valor tipado de un nodo** puede ser accedido usando la función:
 - `dm:typed-value($n as Node) as AtomicValue*`
 - `dm:typed-value` retorna el **valor tipado de un nodo**, el cual es una secuencia de cero o más valores atómicos.
 - Cuando el tipo es un tipo atómico, el valor tipado es siempre el valor atómico construido del valor string y del tipo.
 - En el caso general, `dm:typed-value` construye una secuencia de valores atómicos. Esos valores son derivados del valor string del elemento y su tipo.

XQuery: Items y valores

- **Item** = nodo o valor atómico.
- **Valor** = secuencia ordenada de cero o más ítems.
 - **Ejemplos:**
 1. `()` = secuencia vacía.
 2. `("Hello", "World")`
 3. `("Hello", <PRICE>2.50</PRICE>, 10)`
 - Las estructuras de lista anidadas son expandidas.
 - **Ejemplo:** `((1,2),(),(3,(4,5))) = (1,2,3,4,5) = 1,2,3,4,5.`

XQuery: Items y valores

- En la **comparación de valores** se usan los **operadores de comparación de Fortran**.
 - eq, ne, gt, ge, lt, le.
 - Cada operando debe ser un valor atómico, un nodo que contiene un valor atómico, o la secuencia vacía.
 - Si un operando es una secuencia de más de un item se emite un error.
 - Si cualquier operando es la secuencia vacía se devuelve la secuencia vacía.
 - Valores no tipados son tratados como strings por comparaciones de valores.
 - Los dos operandos cuando son tipados deben tener tipos comparables.

XQuery: Expresiones de Comparación

- Ejemplos:

- 3 gt 4:
- “abc” lt “def”:
- <a>3 gt <z>2</z>:
- <a>03 gt <z>2</z>:
- (1,2) eq (1,2):
- “4” gt 3:
- Xs:integer(“4”) gt 3:

XQuery: Expresiones de Comparación

- Ejemplos:

- 3 gt 4: false
- “abc” lt “def”: true
- <a>3 gt <z>2</z>: true
- <a>03 gt <z>2</z>: false (a y z son no tipados)
- (1,2) eq (1,2): Type error.
- “4” gt 3: Type error
- Xs:integer(“4”) gt 3: true

XQuery: Expresiones

- **Operadores aritméticos:** +, - , *, div, idiv, mod.
 - Aplicarlos a cualquier expresión que devuelve valores aritméticos o de fecha/hora.

XQuery: Expresiones de Comparación

- Las **comparaciones generales** son usadas para comparar
 - valores atómicos o nodos que contienen valores atómicos.
 - secuencias de más de un ítem, así como secuencias vacías.
- Las comparaciones generales usan los operadores
 - =, !=, <, <=, >, >=.
- Cuando se comparan dos valores, sus tipos son tenidos en cuenta.
- **¿Cómo se pueden comparar valores de tipos parecidos?**
 - Valores de tipos parecidos (p.ej. ambos numéricos) pueden siempre ser testeados por igualdad usando los operadores = y !=.

XQuery: Expresiones de Comparación

- **¿Cómo se comparan dos valores no tipados?**
 - Son comparados como strings.
- **¿Cómo se comparan dos valores atómicos, uno tipado y el otro no?**
 - El valor no tipado es convertido al tipo del otro valor.
 - La comparación general trata de hacer una coersión a un tipo “requerido apropiado” para hacer que la comparación trabaje.

XQuery: Expresiones de Comparación

Semántica de comparación general:

- Comparaciones generales son comparaciones existencialmente cuantificadas que pueden ser aplicadas a secuencias de operandos de cualquier longitud.
 - El resultado de una comparación general que no emite un error es siempre `true` o `false`.
- Para comparar dos secuencias de valores atómicos, la comparación general retorna `true` si hay un valor en la izquierda que se corresponde con un valor en la derecha, usando la comparación apropiada.
 - **Ejemplo:** $c = c'$ retorna `true` si y solo si hay un valor en c que se corresponde con un valor en c' .

XQuery: Expresiones de Comparación

- Los operandos antes de ser comparados deben ser **atomizados**.
 - **Atomización** es aplicada a un valor cuando el valor es usado en un contexto en el cual una secuencia de valores atómicos es requerida. El resultado de la atomización es una secuencia de valores atómicos o el [type error](#).
- Después de la atomización y coersión los valores atómicos son comparados usando uno de los operadores de comparación de valores `eq`, `ne`, `lt`, `le`, `gt`, o `ge`, dependiendo de si el operador de comparación general fue `=`, `!=`, `<`, `<=`, `>`, o `>=`.
- Una comparación general puede dar lugar a un [error](#) tan pronto como la misma encuentra un error en evaluar cada operando, o en comparar un par de ítems de los dos operandos.

XQuery: Expresiones de Comparación

- Ejemplos:
 - $1 > 2$:
 - $() = (1, 2)$:
 - $(2, 5) > (1, 3)$:
 - $(1, "a") = (2, "b")$:

XQuery: Expresiones de Comparación

- Ejemplos:

- $1 > 2$: true
- $() = (1, 2)$: false
- $(2, 5) > (1, 3)$: true
- $(1, "a") = (2, "b")$: type error

XQuery: Valor efectivo booleano

- El **valor booleano efectivo** (EBV) de una expresión es:
 1. El valor actual si la expresión es de tipo booleano.
 2. FALSE si la expresión evalúa a 0, "" [string vacío], o () [secuencia vacía].
 3. TRUE de otro modo.
- **Ejemplos:**
 1. `@name="JoesBar"` tiene EBV TRUE o FALSE, dependiendo de si el atributo `name` es "JoesBar".
 2. `/BARS/BAR[@name="GoldenRail"]` tiene EBV TRUE si algún bar se llama Golden Rail, y FALSE si no hay tal bar.

XQuery: Valor Efectivo Booleano

- **Conectivos booleanos:** E_1 and E_2 , E_1 or E_2 , not(E)
- if (E_1) then E_2 else E_3
 - se aplica a cualesquiera expresiones.
- Tomar el EBV's de la expresión primero.
 - **Ejemplo:** not (3 eq 5 or 0) tiene valor TRUE.
- También: true () y false () son funciones que retornan valores TRUE y FALSE.

XQuery: Secuencias

- Una **secuencia** puede ser creada explícitamente usando un **constructor de secuencia**.
 - serie de valores delimitados por comas y encerrada entre paréntesis.
 - **Ejemplo:** (1, 2, 3) crea secuencia consistente en 3 valores atómicos.
 - También se pueden usar expresiones dentro de constructores de secuencia.
- Las secuencias son ordenadas y el orden no es necesariamente el mismo que el orden del documento.
- Además las secuencias pueden contener nodos duplicados.

XQuery : Secuencias

- Las secuencias no tienen nombres, aunque pueden estar ligadas a una variable nombrada.
 - Una secuencia con solo un ítem se llama **secuencia singleton**.
 - No hay diferencia entre una secuencia singleton y el ítem que contiene.
 - Todas las funciones u operadores que operan en secuencias también pueden operar en ítems, los cuales son tratados como secuencias singleton.
- Una secuencia con **0** ítems se llama **secuencia vacía**.
 - En XQuery la secuencia vacía es distinta del string vacío ("") o de un valor **0**.
 - Muchas de las funciones y operaciones built-in aceptan la secuencia vacía como argumento y han definido un comportamiento para manejarla.

XQuery : Secuencias

- Las secuencias no pueden ser anidadas dentro de otras secuencias; hay solo un nivel de ítems. Las secuencias pueden ser insertadas en otras secuencias
- **Ejemplo:** (10, (20, 30), 40) es equivalente a (10, 20, 30, 40).
- Algunas de las funciones en secuencia más usadas son las **funciones de agregación** (min, max, avg, sum).
- union, intersect, except operan en secuencias de nodos.
 - Los significados son análogos que en SQL.
 - El resultado elimina duplicados.
 - El resultado aparece en el orden del documento.

XQuery : Secuencias

- Hay también un número de funciones que operan genéricamente en toda secuencia tales como:
 - **index-of**: encuentra un ítem en una secuencia.
 - **insert-before**: inserta ítems en una secuencia.
 - **Ejemplo**: `insert-before((1, 2, 3, 4), 2, (5, 6)) => (1, 5, 6, 2, 3, 4)`
 - **count**: la longitud de la secuencia.
 - **distinct-values**: remueve todos los valores duplicados
 - **remove**: remueve ítems de la secuencia.
 - **Ejemplo**: `remove($seq, 2)`
 - **subsequence**: selecciona una subsecuencia.
 - **Ejemplo**: `subsequence($seq, 1, 3)`
 - **exists**: true si la secuencia no es vacía.
 - **empty**: true si la secuencia es vacía

XQuery: Literales

- **Literales** son simplemente valores constantes que son representados directamente en una consulta.
 - **Ejemplo:** “BBC” y 23,45.
 - Los literales pueden usarse en expresiones en todo lugar donde un valor constante es necesitado.
- Hay dos tipos de literales:
 - **literales string** que deben estar englobados entre comillas o apóstrofes, y
 - **literales numéricos** que pueden tomar la forma de enteros simples, números decimales o números en punto flotante.
 - El procesador hace su-posiciones acerca del tipo de un literal numérico basado en su formato.
- Se pueden usar **constructores de tipo** para convertir sus valores literal al tipo deseado.
 - **Ejemplo:** `xs:date("2007-12-04")` .

XQuery: Funciones

- Las **funciones** son definidas usando **declaraciones de funciones**.
 - Las funciones pueden aparecer en el prólogo o en una biblioteca externa.
- Una **declaración de función** consiste de varias partes:
 - La palabra clave **declare function** seguida por un nombre de función.
 - Una lista de parámetros entre paréntesis y separados por coma.
 - El tipo de retorno de la función.
 - Un cuerpo de función entre llaves y seguido de ‘;’.
- El **cuerpo de una función** puede contener cualquier expresión XQuery válida, incluyendo FLWORS, expresiones camino y cualquier otra expresión XQuery. No necesita contener una cláusula `return`; el valor de retorno es simplemente el valor de la expresión.

XQuery : Funciones

- **Ejemplo:** declare function local:get-pi() {3.141592653589};
- **Ejemplo:** la siguiente es una declaración de función:
declare function local:discountPrice(\$price as xs:decimal?,
 \$discount as xs:decimal?,
 \$maxDiscountPct as xs:integer?) as xs:decimal? {
 let \$maxDiscount := (\$price * \$maxDiscountPct) div 100
 let \$actualDiscount := min((\$maxDiscount, \$discount))
 return (\$price - \$actualDiscount)
};

XQuery : Funciones

- Dentro del cuerpo de una función una función puede invocar otras funciones, sin importar el orden de sus declaraciones.
- Si no se especifica el tipo de retorno se asume que es **item*** (esto es, una secuencia de cero o más valores atómicos y nodos).
- Cada **parámetro** en una lista de parámetros tiene un nombre único y opcionalmente un tipo.
 - El nombre es expresado como un nombre de variable, precedido por **\$**.
 - Cuando se llama una función, la variable especificada es ligada al valor que es pasado a la misma.
 - Si no se especifica un tipo para un parámetro especificado, permite cualquier argumento.
 - Sin embargo, es lo mejor especificar el tipo, para chequeo de errores y claridad.

XQuery: Funciones

- **Llamadas a función** son otro bloque de construcción para consultas.
 - Una llamada a función típica tiene un nombre de función y argumentos entre paréntesis separados por coma.
 - Los argumentos pueden ser referencias a variables, literales, etc.
- XQuery tiene alrededor de 100 **funciones built-in**.
- Para comparar los contenidos y atributos de dos nodos se puede usar la función built-in **deep-equal**.

XQuery: Variables

- **Variables** pueden opcionalmente declaradas en un prólogo de consulta.
- Crear variables en el prólogo puede ser también una forma útil de definir constantes o valores que pueden ser calculados antes y usados a lo largo de la consulta.
- Hay que recordar que variables globales son inmutables tal como las otras variables XQuery.
- La sintaxis de una declaración de variable es:

```
declare variable $ <variable-name> := <expr> ;
```

XQuery: Variables

- **Ejemplo:** `declare variable $maxItems := 12;`
- Si un cuerpo de función referencia una variable que es declarada en el prólogo, la declaración de función debe aparecer después de la declaración de variable.
- Si una variable global referencia a otra, la variable referenciante debe venir después que la variable referenciada.
- Una variable puede ser ligada solo una vez.
- La expresión que especifica el valor de una variable se conoce como la **expresión de inicialización**.
 - La misma no tiene porque ser una constante y puede ser cualquier expresión XQuery válida.
 - La expresión de inicialización puede llamar una función.

XQuery : Variables

- Las **variables** en XQuery son identificadas por nombres que son precedidos por '\$'.
- Cuando se evalúa una consulta una variable está ligada a un valor particular.
 - Ese valor puede ser cualquier secuencia, incluyendo un nodo simple, un valor atómico simple, la secuencia vacía o varios nodos y/o valores atómicos.
- Una vez que la variable está ligada a un valor su valor no cambia.
 - Una consecuencia de esto es que no se puede asignar un valor nuevo a una variable. En lugar de eso se usa una nueva variable.

XQuery: Prólogo

- El **prólogo** consiste de una serie de declaraciones terminadas por ';'. Hay 3 secciones distintas en un prólogo.
- La primera declaración a aparecer en el prólogo es la **declaración de versión** si existe. La sintaxis es:

```
XQuery version "<version>" encoding "<character encoding>" ;
```

- La última sección del prólogo contiene declaraciones de funciones y de variables.
 - **Declaraciones de funciones**: declaran funciones definidas por el usuario.
 - **Declaraciones de variables**: declara variables globales.