4 Dati semistrutturati - XML

Registrazione 341

Link XML sito Franceschet

Affrontiamo questo argomento perché ci stiamo muovendo sempre più verso dati poco strutturati. Abbiamo visto fino ad ora dati che possono esser rappresentati su una struttura "rettangolare". Con i network abbiamo visto sempre una struttura tabellare, dove però le colonne non erano le variabili e le righe non erano le osservazioni. Ci stiamo allora sgretolando sempre di più, come si suol dire "decomponendo" nell'ambito psicologico. Dobbiamo esser in grado alla fine di questo corso di analizzare si adatti con una struttura che è senza struttura, perché è facile analizzare dati strutturati, risulta più difficile farlo con dati meno strutturati.

Vedremo ora come analizzare una pagina web, per la quale è da matti analizzarla in una tabella. Vedremo che ci sono molte varianti: assenza di informazione, informazione ripetuta, struttura gerarchica annidata. Ci sono due formalismi utilizzati per rappresentare tali dati: JSON e XML. Noi vedremo XML.

4.2 Extensible Markup Language

XML è un acronimo e sta per Extensible Markup Language e quindi è:

- Language (linguaggio). Linguaggio formale. Cioè un documento XML è definito in base a delle regole formali, cioè una grammatica che dice esattamente come strutturare un documento
- Markup (annotazione): i dati o il testo all'interno del documento sono annotati con delle annotazioni (tag), che danno semantica al testo
- Extensible (estensibile): dice che l'insieme dei tag non è fissato a priori, ma definito dal progettista. A differenza dell'HTML dove i tag sono prefissati e non possono essere inventati.

Un documento XML è un documento di testo con delle annotazioni che servono per dare una semantica, cioè un significato all'informazione. Non è un linguaggio di presentazione come HTML, anche se usa i tag come HTML, in particolare il significato dei tag non è di fare una formattazione (tipo questa parola è in grassetto), ma di dare un significato (questa parola è il prezzo del libro). Non è un linguaggio come java. Non è un protocollo e neanche un DBase. È semplicemente un file di testo per rappresentare dati semistrutturati.

Vediamo ora la struttura di un documento XML.
Parleremo innanzitutto e soprattutto del testo. Un testo sarà delimitato con dei tag o etichette come quelle che si usa per HTML. Per dire che Luca Buratto è una persona scrivo:

```
<person>
  Luca Buratto
</person>
```

Apro il tag persone inserisco contenuto e lo chiudo. Si nota quindi la differenza tra il tag di apertura e il tag di chiusura. Sto dicendo che la stringa Luca Buratto si riferisce ad una persona. Questo è un elemento, Luca Buratto e il suo contenuto e il nome dell'elemento è person. In questo modo sto dando semantica, cioè do un significato alla stringa Luca Buratto.

I tag servono a descrivere il tipo del contenuto. Vediamo un elemento più complesso.

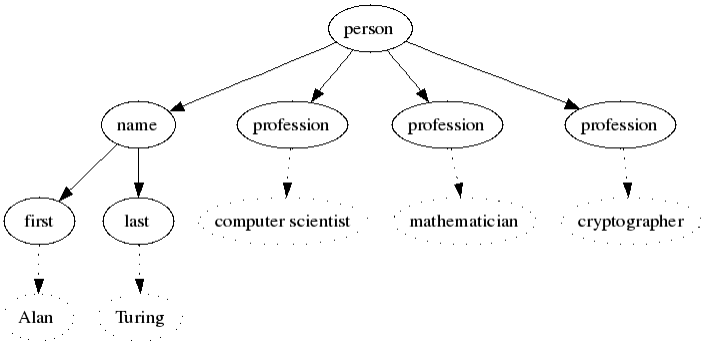
```
<person>
   <name>
      <first>Alan</first>
      <last>Turing</last>
   </name>
   <profession>computer scientist</profession>
   <profession>mathematician</profession>
   <profession>cryptographer</profession>
</person>
```

Gli elementi profession, first e last contengono solo del testo. Altri elementi come name o person sono invece **elementi strutturati**, cioè contengono altri elementi. Questa è una differenza rispetto all'HTML: XML è un modello gerarchico, un elemento può contenere altri elementi che a loro volta ne contengono altri.

Ma.. gli elementi non possono sovrapporsi. Ci sono due regole:

- **Gli elementi non possono sovrapporsi**, cioè se apro person e poi apro name, non posso chiudere person prima di name. La relazione tra gli elementi può essere o di inclusione, come name e person, oppure disgiunti come name e professione. Ma non possono essere intersecanti.
- **Ci deve essere un elemento contenitore**, cioè un elemento radice che contiene tutti gli altri. Questo fa si che la struttura è una struttura ad albero (grafo diretto privo di cicli).

Nell'esempio di prima l'albero è: in questo albero abbiamo i così detti ***nodi figli*** e ***nodi padre***. Person è nodo padre di profession. Profession è nodo padre di computer scientist, con contorno punteggiato perché indica un nodo testo. La relazione elemento-sottoelemento corrisponde alla relazione padre e figlio nell'albero.



Posso anche mescolare testo ed elementi. Ad esempio

```
<person>
   <first-name>Alan</first-name>
   <last-name>Turing</last-name>
   is mainly known
   as a <profession>computer scientist</profession>.

However, he was also an accomplished <profession>mathematician</profession>
   and a <profession>cryptographer</profession>.
</person>
```

Posso anche scrivere anche degli **elementi vuoti** abbreviando in questo modo:

```
<address/>
```

Significa che adress non ha alcun contenuto.

Si noti che XML è case sensitive.

Attributi

Ci sono due componenti in un documento XML,

- uno sono gli **elementi** e
- gli altri sono gli **attributi**.

Gli attributi sono proprietà che valgono per l'intero elemento e si scrivono con questa sintassi. Ipotizziamo di voler aggiungere due attributi all'elemento person: *Born* e *died*, che contengono la data di nascita e di morte di Alan Turing. Nel tag iniziale dell'elemento scrivo gli attributi.
La struttura quindi è name = "value" (singoli apici o doppi, indifferente). Gli attributi all'interno di uno stesso elemento devono avere nomi distinti.

```
<person born="23/06/1912" died="07/06/1954">
   <name>
      <first>Alan</first>
      <last>Turing</last>
   </name>
   <profession>computer scientist</profession>
   <profession>mathematician</profession>
```

```
   <profession>cryptographer</profession>
</person>
```

Ora vediamo lo stesso documento in cui si usa in modo smodato gli attributi, cioè si rappresenta tutta la informazione mediante gli attributi.

```
<person born="23/06/1912" died="07/06/1954">
  <name first="Alan" last="Turing"/>
  <profession value="computer scientist"/>
  <profession value="mathematician"/>
  <profession value="cryptographer"/>
</person>
```

Gli attributi hanno sempre una struttura atomica, mentre gli elementi possono essere strutturati in modo arbitrario. Ci son altri elementi che non sono importanti per noi, salto fino a qui.

Storia XML
XML è nato negli anni'60 presso IBM. Negli anni '80 implementato, ma poco usato ed implementato perché complicato. Finché Tim Berners-Lee propose di fissare il vocabolario, ossia i tag. L'applicazione più famosa di SGML è stata HTML. Poi salto a qui.

Schema
Dobbiamo definire uno **schema per il documento XML**, innanzitutto dobbiamo capire che significa schema per un documento XML. Schema è lo stesso 'schema' che conosciamo nelle basi di dati. Nelle base di dati avevamo detto che c'è una tabella, c'era una istanza che ha pure uno schema, che era il nome delle colonne e dei relativi tipi.
Avevamo detto che lo schema definisce la tabella e l'istanza definisce il contenuto, stessa definizione che intercorre tra classi ed oggetti nella programmazione orientata agli oggetti. Anche qui c'è questa differenza: i documenti XML sono le istanze, cioè i dati, gli schemi specificano questa semistruttura dei dati, nel senso che si tratta di dati semistrutturati i quali devono essere posti nel documento secondo certe regole, le quali sono definite in uno *schema*.

Ci sono due **linguaggi per definire gli schemi XML**:
1. ***DTD*** Document Type Definition (che vedremo)
2. *XML Schema* (non lo vedremo)

4.3 Document Type Declaration (DTD)

DTD, che una definizione di tipo del dato, cioè di classe dei nostri oggetti, è interessante perché ci introduce le espressioni irregolari (argomento successivo del corso). Vedremo come è possibile scrivere degli schemi. DTD non è un'applicazione XML, quindi non sarà un linguaggio XML, ma si usano queste clausole. Potremmo scrivere lo schema così. Vogliamo dare uno schema all'esempio di prima:

```
<!ELEMENT person     (name, profession*)>
<!ATTLIST person      born CDATA #REQUIRED died CDATA #IMPLIED>
<!ELEMENT name       (first,last)>
<!ELEMENT first      (#PCDATA)>
<!ELEMENT last       (#PCDATA)>
<!ELEMENT profession (#PCDATA)>
```

Simbolo	Senso	Esempio
`#PCDATA`	Contiene dati di carattere o testo analizzati	`<elemento (#PCDATA)>`
`#PCDATA, nome-elemento`	Contiene testo e un altro elemento; #PCDATA è sempre elencato per primo in una regola	`<element (#PCDATA, child) *>`
`, (comma)`	Deve usare in questo ordine	`<elemento (child1, child2, child3)>`
`| (pipe bar)`	Usa solo un elemento delle scelte fornite	`<elemento (child1 | child2 | child3)>`
`nome-elemento (di per sé)`	Utilizzare solo una volta	`<elemento (figlio)>`
`Elemento-name?`	Utilizzare una volta o non utilizzarlo	`<elemento (child1, child2 ?, child3?)>`
`elemento nome +`	Utilizzare una o più volte	`<elemento (child1 +, child2 ?, child3)>`
`elemento nome *`	Utilizzare una volta, molte volte o non farlo affatto	`<elemento (child1 *, child2 +, child3)>`
`( )`	Indica gruppi; può essere annidato	`<elemento (#PCDATA | figlio) *> o <elemento ((figlio1 *, figlio2 +, figlio3) * | figlio4)>`

Element person ha l'attributo born, died.

- profession***** si mette la stella per indicare che può avere più determinazioni (maggiore o uguale di 0)
- **(#PCDATA)>** principalmente significa che sono elementi atomici e possono **contenere solo testo**, cioè sono foglie del nostro albero.
- L'elemento person ha una lista di attributi born died
- L'elemento name è strutturato come first e last. Se non metto alcun quantificatore significa che deve apparire una sola volta.
- Gli elementi first, last e profession contengono solo testo, e non contengono altri elementi

Il documento che abbiamo visto prima risulta essere valido secondo questo schema, cioè soddisfa questo schema, ma anche altri documenti potrebbero esser validi. Ad esempio, se togliessimo le tre professioni esso soddisferebbe ancora questo schema, anche se ne aggiungessimo ancora una. Ma togliessimo l'elemento name non soddisfarebbe lo schema.
Uno schema è come se fosse un insieme infinito di documenti. Diremo che un documento/istanza soddisfa uno schema se fa parte di quella collezione (insieme infinito di documenti).

Andiamo nel dettaglio. La prima cosa che dobbiamo scrivere in un documento XML e che vogliamo convalidare secondo uno schema è la cosiddetta ***Document Type Declaration***, cioè la dichiarazione della DTD. È semplicemente una riga che dice qual è la radice del nostro documento, nel nostro caso *person*, e dove si trova la DTD.

Dove si trova la DTD?

- La DTD si può trovare in un file esterno tipo url,
- oppure può essere scritta all'interno del documento, ma quest'ultimo caso non è una buona idea perché una DTD viene usata per più documenti ossia corrisponde a più documenti.

Conviene quindi separare la DTD dal documento XML mettendo nel documento XML un puntatore riferito alla DTD che rappresenta lo schema del documento XML.

Validity

Diremo che un documento XML è valido (secondo uno schema) se soddisfa le regole di quell'espressione regolare, ossia se fa parte dell'insieme delle istanze che soddisfano quello schema. È importante tenere presente che, quando si carica un documento XML con un Browser, non viene fatta la validazione; viene fatto solo il controllo che il documento sia ben formato, cioè che rappresenti l'albero (tag chiusi, non intersezione, …). Il browser controlla solo la 'grammatica'. Per il controllo 'completo' si usano software appositi.

Elementi principali della DTD

Vediamo le componenti principali della DTD. La DTD serve per vedere lo schema. Sapendo che XML è definito in base agli elementi e agli attributi, mi serviranno delle regole per definire il contenuto degli elementi, e la loro struttura, e delle regole, che saranno un po' più semplici, per definire il contenuto degli attributi, in quanto contengono solo testo.

La definizione/dichiarazione *element* serve per dichiarare un elemento di nome *name* con un contenuto.

```
<!ELEMENT name content>
```

Vediamo degli esempi:

```
<!ELEMENT email (#PCDATA)>
```

ho un solo elemento email che contiene del testo. Non ci sono altri elementi

```
<!ELEMENT contact (e-mail | phone)>
```

l'elemento contact contiene un solo altro elmento: *e-mail* ossia la email oppure contiene element phone, ma non entrambi. **La barra verticale significa o esclusivo**.

```
<!ELEMENT name (first,last)>
```

L'elemento name deve contenere l'elemento *first*, seguito dall'elemento *last*. **La virgola significa composizione**.

```
<!ELEMENT image EMPTY>
```

Posso dire che ho **un elemento vuoto**.

```
<!ELEMENT image EMPTY>
```

L'elemento ha una forma qualsiasi.

Infine, la parte più interessante è **l'iterazione**, perché si può iterare la composizione con la scelta con questi tre operatori:

```
*, + e ?
```

Vediamo alcuni esempi: supponiamo che l'elemento name è strutturato in questo modo:

```
<!ELEMENT name (first*, middle?, last+)>
```

I seguenti esempi sono tutti dei documenti che soddisfano tale schema:
La prima è ok. La seconda anche, perché middle è facoltativo. La terza anche perché middle facoltativo, almeno un last.

```
<name>
        <first>Samuel</first>
        <middle>Lee</middle>
        <last>Jackson</last>
</name>
<name>
        <first>Samuel</first>
        <first>Michael</first>
```

```
        <last>Jackson</last>
</name>
<name>
        <last>Jackson</last>
        <last>Keaton</last>

</name>
```

Se mettessimo due elementi middle o zero di last non sarebbero sai validi, in quanto non soddisfano la definizione.

Contenuto misto
S'intende sia testo che elementi. La prossima definizione dice che il nome può avere qualsiasi numero di first, middle e last in qualsiasi ordine:
```
<!ELEMENT name (#PCDATA | first | middle | last)*
```

Ad esempio: elemento testo, elemento middle, elemento last, elemento testo, elemento testo, elemento last

Certe definizioni sono **vietate**, in particolare la prossima definizione che dice che il nome potrebbe essere o il nome di battesimo o un cognome o un nome di battesimo seguito da un cognome. Questo è sbagliato perché il contenuto non è deterministico.
```
<!ELEMENT name (first | last | (first,last))>
```

Analizziamo il motivo: una DTD corrisponde ad una *espressione regolare*, alcune espressioni regolari possono corrispondere ad automi non deterministici. Perché non deterministici? Perché se il validatore incontra un elemento first ha due possibilità: o l'elemento name è finito o si aspetta un elenco last, riferendo name quell'ultimo elemento. Poiché il validatore ha letto lo stesso simbolo in due modi diversi, allora siamo in presenza di un comportamento non deterministico. Non ammesso per motivi computazionali. Lo vedremo meglio quando analizzeremo gli automi.
Ci **si ricordi che le DTD devono essere deterministiche**: ad ogni elemento letto deve esserci un'unica uscita.

8/5/2018
registrazione 342

Video

Definizione degli attributi
per adesso abbiamo visto come scrivere delle istanze con XML e degli schemi con DTD. Ora vediamo come definire gli attributi Link.

Con la parola chiave ATTLIST dichiariamo l'elemento, l'attributo, il suo tipo ed eventualmente il suo valore di default:
```
<!ATTLIST element attribute TYPE DEFAULT>
```

In realtà il tipo non il tipo nel senso che conosciamo, ma possiamo dire semplicemente che l'attributo contiene del testo (CDATA) o contenga dei numeri (attributo enumerativo) o, ciò che ci interessa di più, un ID. Quando un attributo è di tipo ID corrisponde ad una chiave nel senso delle base di dati: l'attributo ID deve essere univoco, nel senso che il valore dell'attributo ***ID*** deve essere univoco per tutti gli attributi di tipo ID all'interno del documento XML. In più abbiamo l'***IDREF*** che corrisponde alla chiave esterna. In questo caso il vincolo deve soddisfare la richiesta per la quale ci deve essere un altro attributo di tipo ID che contiene tali valori (classico vincolo di chiave esterna).

Oltre a definire il tipo per attributo bisogna definire anche un default definition. Ci sono quattro possibilità per questi default:

1. **#REQUIRED** per il quale il valore dell'attributo deve essere definito
2. #IMPLIED per il quale il valore dell'attributo è opzionale

Supponiamo di avere una relazione banking tra due entità, un customer e un account. E una relazione *Bank*. Supponiamo che un cliente possa avere più conti correnti e che un account possa essere associato ad un unico cliente. Non ci possono essere conti condivisi. Se fossimo in un Database relazionale avremmo 2 sole tabelle, poiché è una relazione di tipo uno a molti. Nel modello relazionale useremmo una chiave esterna che ogni account associa il relativo customer. Come modellare in XML questa informazione? In XML modelliamo la relazione in questo modo: link

Creiamo un elemento contenitore banking che contiene una lista di customer. Ogni customer ha al suo interno un attributo nome del cliente e una lista di account che lui possiede. Inoltre ogni customer ha un attributo ID obbligatorio. Invece ogni account ha solo il nome della banca, il numero di conto corrente e un attributo ID obbligatorio.

```
<?xml version="1.0"?>

<!DOCTYPE banking [
  <!ELEMENT banking    (customer*)>
  <!ELEMENT customer   (name, account*)>
  <!ATTLIST customer    id ID #REQUIRED>
  <!ELEMENT account    (bank,number)>
  <!ATTLIST account     id ID #REQUIRED>
  <!ELEMENT name       (#PCDATA)>
  <!ELEMENT bank       (#PCDATA)>
  <!ELEMENT number     (#PCDATA)>
]>
```

Vediamo due istanze valide:
- c'è un cliente C1 che ha gli account A1 e A2 e un altro cliente A2 che ha un unico conto

```
<banking>
   <customer id="C1">
   <name>Massimo Franceschet</name>
   <account id="A1">
      <bank>Fineco</bank>
      <number>34567</number>
   </account>

   <account id="A2">
      <bank>ABN AMRO</bank>
      <number>98672</number>
   </account>
   </customer>
</banking>
```

Come vediamo non si ha usato il meccanismo delle chiavi esterne perché non serve, in quanto si può usare il fatto che XML sia gerarchico per annidare un elemento dentro l'altro, per cui il fatto che il conto corrente A2 sia del customer C1 lo vedo perché sta dentro l'elemento con id C1. La gerarchia padre - figlio è usata per modellare la relazione che mi dice che un account\conto corrente è di proprietà di certo customer. Questo è comodo perché evita di inserire il vincolo di chiave esterna. Si nota che tutto funziona nell'esempio perché la relazione è uno a molti (1 account no 2 proprietari).

Supponiamo ora di cambiare ora la semantica: un account può avere anche più di un solo proprietario. Prendiamo lo stesso documento. L'unica differenza ora è che il conto è cointestato. Vediamo che il seguente codice non va bene:

```
<banking>
   <customer id="C1">
      <name>Massimo Franceschet</name>
      <account id="A1">
         <bank>Fineco</bank>
         <number>34567</number>
      </account>
      <account id="A2">
         <bank>ABN AMRO</bank>
         <number>98672</number>
      </account>
   </customer>

<customer id="C2">
      <name>Enrico Zimuel</name>
      <account id="A2">
         <bank>ABN AMRO</bank>
         <number>98672</number>
      </account>
   </customer>
</banking>
```

Questo schema NON va bene perché c'è una violazione del vincolo di chiave primaria. Ho due attributi di tipo ID con un valore duplicato (un ID duplicato "A2" che va bene per entrambi i prof.), quindi non posso fare questa cosa. Allora è fondamentale la soluzione di prima che funzionava perché ad ogni account era associato un unico cliente e quindi non potevo avere duplicati.

A questo punto se vogliamo tenere la relazione molti a molti devo passare ad un approccio relazionale: devo creare un elemento customer e uno ID in questo modo:

```
<!DOCTYPE banking [
  <!ELEMENT banking    (customer | account)*>
  <!ELEMENT customer   (name, accounts?)>
  <!ATTLIST customer    id ID #REQUIRED>
  <!ELEMENT name       (#PCDATA)>
  <!ELEMENT accounts   EMPTY>
  <!ATTLIST accounts   idrefs IDREFS #REQUIRED>

  <!ELEMENT account    (bank,number,owners)>
  <!ATTLIST account     id ID #REQUIRED>
  <!ELEMENT bank       (#PCDATA)>
  <!ELEMENT number     (#PCDATA)>
  <!ELEMENT owners     EMPTY>
  <!ATTLIST owners     idrefs IDREFS #REQUIRED>
]>
```

Il Banking ora è una lista arbitraria di elementi di tipo customer e account. L'elemento *customer* ha un elemento names e uno facoltativo account, infatti un customer può avere zero conti correnti. E ha sempre un ID. L'elemento *accounts* ha un attributo idrefs di tipo IDREFS che è la lista dei conti posseduti da quel customer. In modo simmetrico account avrà un elemento bank, poi number, e infine owners. Quest'ultimo sarà un elemento vuoto con un attributo idrefs che punterà ai rispettivi client. Vediamo un'istanza per capire meglio che sta succedendo:

Il cliente C1 Massimo Franceschet ha conti identificati dalle chiavi A1 e A2 che sono il conto Fineco e ABN AMRO. Il Cliente C2 ha il conto A2. Il conto A2 quindi è cointestato, ma non è un problema ora perché non ho una duplicazione di chiavi. Vediamo che ora le chiavi sono tutte univoche e quindi l'istanza è valida: vincoli di chiave primaria e di chiave esterna è soddisfatto, perché ogni elemento degli attributi di tipo IDREFS corrisponde ad un valore dell'attributo ID (controllare frase).

```
<banking>
   <customer id="C1">
        <name>Massimo Franceschet</name>
        <accounts idrefs="A1 A2"/>
   </customer>
   <customer id="C2">
        <name>Enrico Zimuel</name>
        <accounts idrefs="A2"/>
   </customer
   <account id="A1">
        <bank> Fineco </bank>
        <number>34567</number>
        <owners idrefs="C1"/>
        </account>
   <account id="A2">
        <bank>ABN AMRO</bank>
        <number>98672</number>
        <owners idrefs="C1 C2"/>
   </account>
```

In generale quando voglio modellare delle relazioni uno a uno o uno a molti posso sfruttare l'annidamento, mentre quando abbiamo relazioni di tipo molti a molti di deve sostanzialmente simulare il modello relazionale creando due elementi e connettendoli attraverso le chiavi esterne. Nel nostro esempio abbiamo connesso le entità da entrambi i versi, per ogni cliente abbiamo una lista di account e per ogni account abbiamo fatto una lista di clienti: è ridondante. Basterebbe da relazione da un solo verso. Ma ciò ci viene comodo per applicare delle query, per interrogare i dati: dato un cliente so i suoi conti, dato un conto so i suoi proprietari.

C'è un altro schema che è W3C che non vedremo perché è più complesso e formale, esso è utilizzato in particolare per i documenti XML più complessi. Noi andremo ad analizzare documenti semplici e ci limiteremo a tale livello. Vedremo

W3C: questa parte di XML non la vedremo (link). Fine parte di DTD.

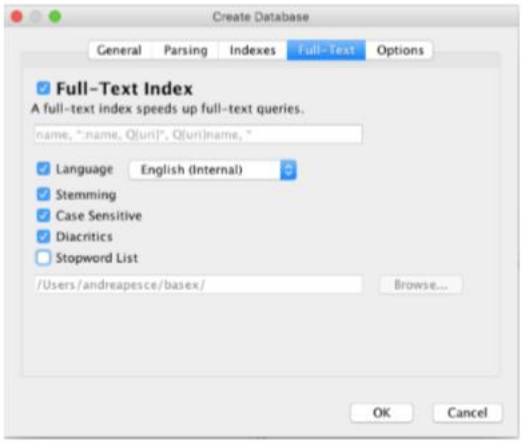
4.4 Query languages for XML: XPath e XQuery

Vedremo ora alcuni **tipi di linguaggi di interrogazione di XML** che ci permettono di estrarre informazione, un po' come avveniva con SQL. Questi linguaggi si chiamano **XPath** e XQuery che ci permettono di estrarre informazioni da un documento XML.

Per vedere questi linguaggi useremo ***BaseX***, framework per elaborare e interrogare file XML e che permette di fare validazione. È scritto in java. Scaricalo da qui. Vediamo un esercizio: XML incontra R esercizio.

Scaricati i due file e messi in una cartella, apriamo BaseX si vada su DataBase -per creare un DataBase nuovo, e si carichi il file voluto, prima di mettere ok, si facciamo come nella figura seguente, in modo tale da creare

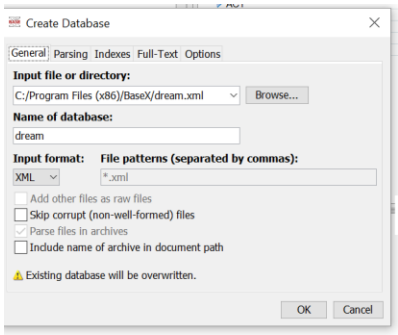
degli indici per velocizzare le query. E da full text barrate stemming, language, case sensitive e diacritics (volendo possiamo aggiungere un file .text con la lista delle stop words).

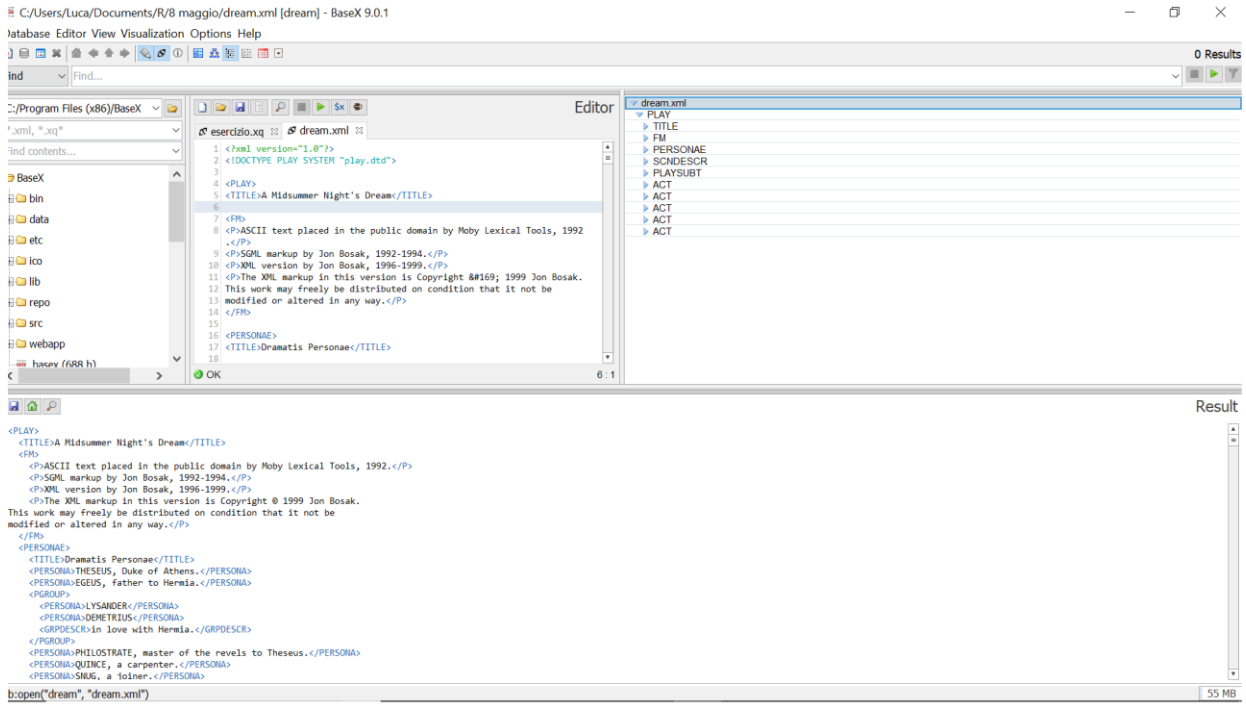


Stiamo creando un DBase nativo XML, quindi non relazionale che non si basa su SQL, che va ad archiviare document XML e che userà un linguaggio di interrogazione che si chiama XQUERY (che vedremo).

Come tutti i DBase in fase di creazione posso creare degli indici per velocizzare le interrogazioni. Come per i DBase relazionali.

Nell'interfaccia BaseX: in alto a destra c'è lo spazio dove immettere i comandi. Si possono cambiare le visualizzazioni del nostro schema andando in alto a sinistra (ad albero, a folder). Sceglo folder.

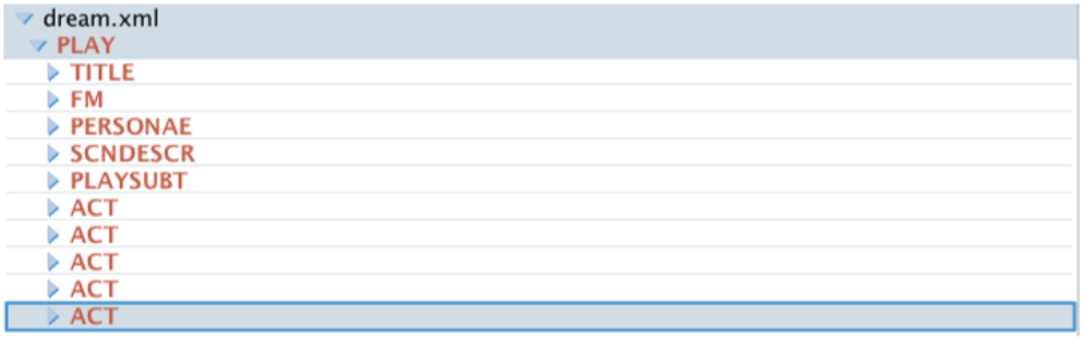


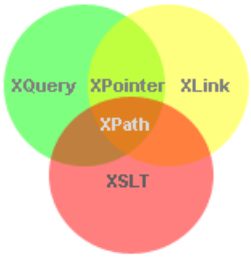


Qua sotto abbiamo la figura che ci rappresenta la struttura del nostro XML, che è un'opera di Shakespeare sogno di una notte di mezza estate. Opera, titolo, persone, atti, scene: ogni scena ha degli speaker, e degli speech.

Questa struttura è data grazie alla semantica di XML, dai tag insomma, che ci permettono anche di interrogare la struttura. Lo scopo di XML è questo, fornire una notazione semantica (parziale) del contenuto. Molliamo

BaseX per ora e andiamo su XPath. Vediamo ora la struttura del primo linguaggio che vedremo che si chiama XPath.





XPath è un semplice linguaggio di recupero di elementi e attributi all'interno di un documento XLM,è una sorta di tecnologia jolly. Ogni nodo del nostro albero può essere o la radice dell'albero oppure è un nodo di tipo elemento o di tipo attributo o altri nodi più specifici (nodi che contengono, testo o istruzioni o commenti). A noi interessano solo i nodi elemento e i nodi attributo.

Nel documento dream.XML troviamo il codice:
prima riga: versione XML che programmatore usa
seconda riga: dichiarazione delle DTD, cioè il documento è validabile secondo uno schema DTD

```
<?xml version="1.0"?>
<!DOCTYPE PLAY SYSTEM "play.dtd">

<PLAY>
<TITLE>A Midsummer Night's Dream</TITLE>
```

```
<?xml version="1.0"?>
<!DOCTYPE person SYSTEM "Turing.dtd">
<?xml-stylesheet type="text/css" href="Turing.css"?>
<!-- Alan Turing was the first computer scientist
<person born="23/06/1912" died="07/06/1954">
   <name>
      <first>Alan</first>
      <last>Turing</last>
   </name>
   <profession>computer scientist</profession>
   As a computer scientist, he is best-known for the Turing Test
   and the Turing Machine.
   <profession>mathematician</profession>
   <profession>cryptographer</profession>
</person>
<!-- He committed suicide on June 7, 1954. -->
```

- Prima riga: dichiarazione della dvd: il documento è validabile rispetto a schema turing DTD
- Seconda riga: istruzione: se voglio visualizzare documento con un browser posso usare Turin.css x fare rendering
- terza riga: file css che caratterizza il layout

- Quarta riga: è comment

4.4.1 XML Path Language

XPath **è una sorta di espressione regolare su alberi, non su stringhe**. Il risultato di una espressione regolare in XPATH su XLM è un sottoinsieme di nodi dell'albero. Abbiamo quindi il nostro albero XML e vogliamo scrivere un'espressione che caratterizzi un insieme di nodi.

Teniamo a mente che i nodi in un documento son ordinati così come appaiono nel documento, quindi se l'elemento A precede B nel documento, allora allora lo precederà anche nell'ordinamento. È un ordine totale dal nodo radice ai figli, e da sinistra a destra nell'albero XML. Si chiama ***ordinamento anticipato*** nel senso che la radice anticipa tutti. Corrisponde all'ordinamento dei tag all'interno del documento. È importante perché le interrogazioni che scriveremo sfrutteranno questo ordinamento. Per comprenderlo meglio conviene andare a vedere nelle slide in quanto spiegarlo a parole risulta complicato.

Siccome ogni documento XML corrisponde ad un nodo dell'albero. Ogni nodo (PER NOI!) può essere o **nodo elemento** o **nodo attributo**.

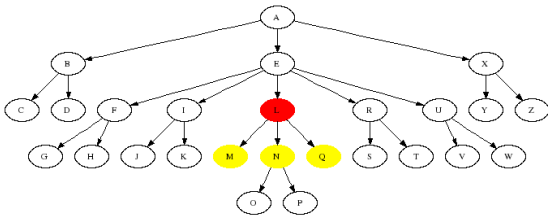
XPath location steps

Tratto da qui. Ogni espressione XPath (o query XPath) viene valutata in un nodo ad albero XML (*chiamato **nodo di contesto***) e restituisce un oggetto di uno dei quattro tipi:

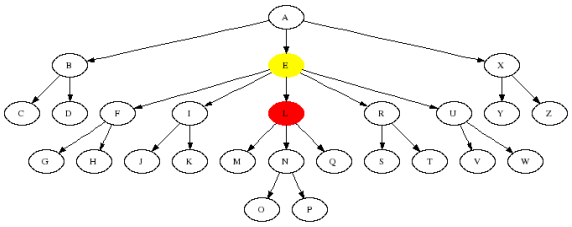
1) La prima componente di una query XPath è una **location step**, cioè un passo che ha una forma del tipo: `axis::test[filter]`. L'asse è una modalità per accedere ai nodi. Il test e il filtro sono delle modalità per filtrare le parti che abbiamo trovato. Il filtro è opzionale.

Vediamo la parte asse. Party dal nodo rosso.

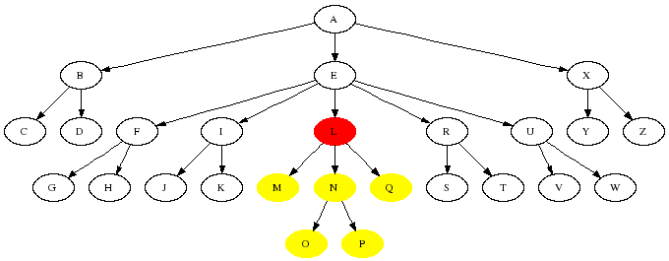
`child::*` Asse child l'asse più semplice sarà l'asse child (rosso), cioè che seleziona tutti i nidi figli (giallo).



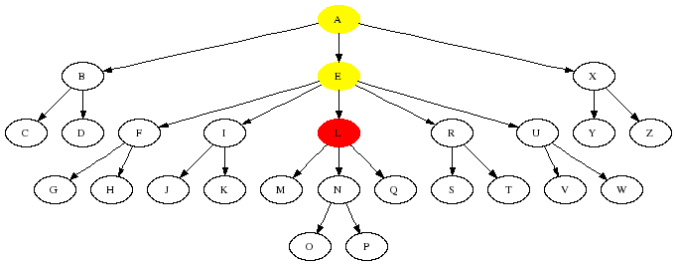
`parent::*` Asse: PARENT di un nodo: assegnerà il nodo padre. Tutti i nodi a parte la radice hanno un padre.



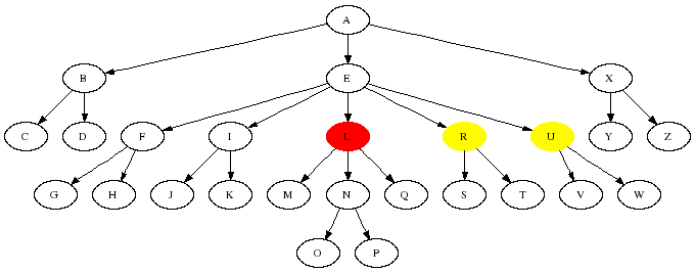
`descendant::*` Asse DESCENDENT: nodi della discendenza. Nodi a cui arrivo potendo solo scendere (in XML sono tutti gli elementi all'interno dell'elemento)



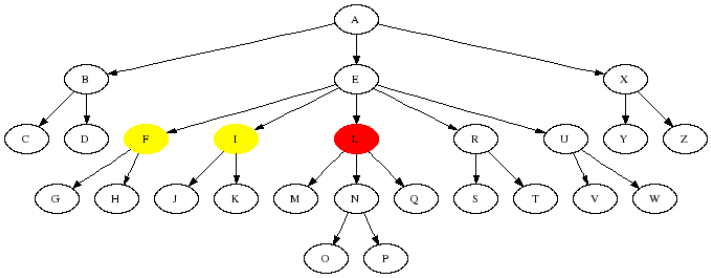
`ancestor::*` Asse ANCESTOR: (antenati) seleziona il nodo padre, il nodo nonno e via su.



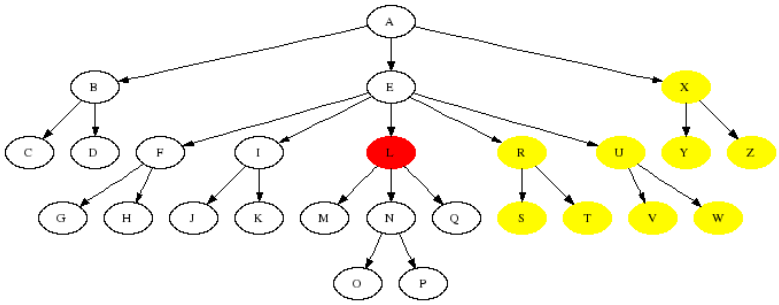
following-sibling::* (ossia asse RIGHT SIBLING) nodi che stanno a dx (mio pdv) su stesso livello. I fratelli o sorelle di un nodo sono i nodi che hanno lo stesso padre



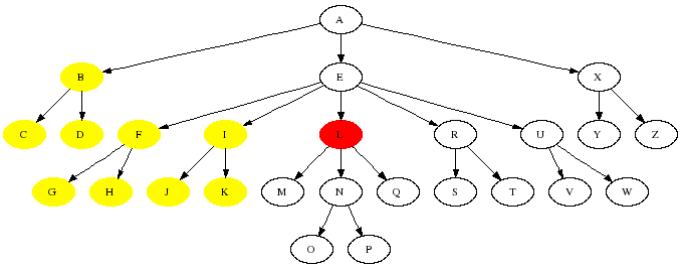
preceding-sibling::* (ossia LEFT SIBLING nodi che stanno a sx (mio pdv) su stesso livello. I fratelli o sorelle di un nodo sono i nodi che hanno lo stesso padre



`following::*` Asse FOLLOW prende tutti I nodi che precedono il nodo a parte il nodo stesso (si è giusto)



`preceding::*` Asse PRECEDE



Il carattere star che rappresenta qualsiasi nodo di quel elemento ma potrebbe essere qualcosa di diverso, ad esempio un elemento specifico (ad esempio voglio tutti i figli che hanno quel nome) oppure altre cose se posse text() prendo tutti i figli di tipo testo, ecc.

Cerchiamo di impostare il nodo di contesto sul nodo della persona. Quindi

- child :: node () corrisponde a tutti i nodi figli, ovvero i nodi figlio nome e elemento professione e l'unico nodo figlio di testo,
- child :: * corrisponde solo ai nodi figlio elemento,
- child :: profession corrisponde solo a nodi figlio elemento professione,
- child :: text () corrisponde al nodo text,

- attribute :: * seleziona entrambi gli attributi
- attribute :: born corrisponde all'attributo di tipo born.

Inoltre, se il nodo di contesto è la radice,

- child :: comment () seleziona i due commenti dell'elemento del documento e
- child :: processing-instruction ('xml-stylesheet') seleziona l'istruzione valutando la query dalla radice

Per esempio prendiamo il documento

```
<?xml version="1.0"?>
<!DOCTYPE person SYSTEM "Turing.dtd">
<?xml-stylesheet type="text/css" href="Turing.css"?>
<!-- Alan Turing was the first computer scientist -->
<person born="23/06/1912" died="07/06/1954">
   <name>
      <first>Alan</first>
      <last>Turing</last>
   </name>
   <profession>computer scientist</profession>
   As a computer scientist, he is best-known for the Turing Test
   and the Turing Matchine.
   <profession>mathematician</profession>
   <profession>cryptographer</profession>
</person>
<!-- He committed suicide on June 7, 1954. -->
```

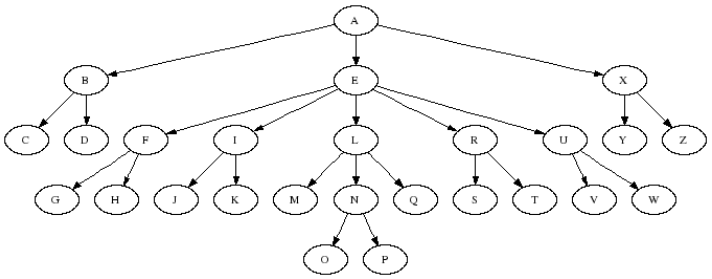
Supponiamo che il nodo con testo da cui valutiamo la nostra valutazione sia person e ci chiediamo se *child::node()*, che seleziona tutti i figli di qualsiasi tipo, verranno selezionati name, i 3 profession, ma anche il figlio di tipo testo “as computer….”. Se invece scriviamo *child::** selezionato solo i figli di tipo elemento: name e i tre profession. Se scriviamo *child::profession* selezioniamo solo i figli di tipo profession e se scriviamo solo *child::text()* selezioniamo solo i figli di tipo testo. Se scriviamo *attributi::** esso accede agli attributi.

(4.4.1) XPath location paths

I location steps possono essere combinati con un location path. Una location path è una sequenza di steps.
Sematica di un PATH:

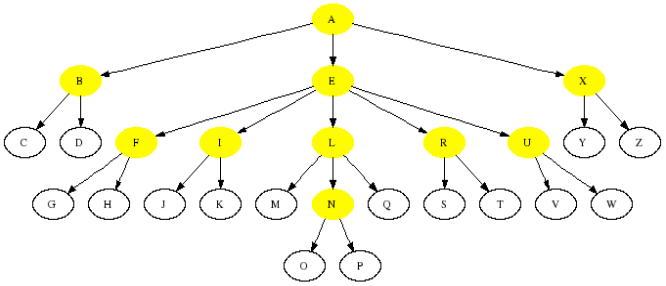
`/descendant::L/child::*/child::*`

Slash **/** significa **partire dalla radice**, applico il primo poi il secondo ed il terzo location steps: sta dicendo parti dalla radice e prendi i figli con qualsiasi nome, poi prendi al loro interno i figli con qualsiasi nome. La soluzione è data dalla unione dai nodi selezionati alla fine degli steps. Nell'esempio il risultato è O e P: prendi i figli di L, prendi i loro figli e poi i nipoti.



Filtro

È una combinazione booleana di Location path. Lo scopo è filtrare nodi che soddisfano delle condizioni. Supponiamo di aver selezionato dei nodi con una interrogazione, a questo punto applico il **filtro** a quei nodi, ossia quanto contenuto nelle parentesi quadre: ad ogni nodo risultato dell'interrogazione applico il filtro, se il risultato è insieme vuoto escludo quel nodo dal risultato altrimenti tengo il nodo come risultato.



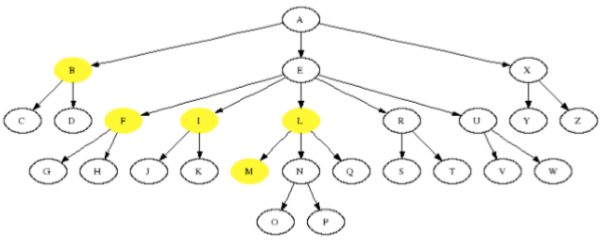
Facciamo un esempio:
/descendant::*[child::*] Il filtro è *child::**. Il primo location step dice di selezionare tutti i nodi, poi il filtro dice di selezionare tutti i **nodi interni** cioè tutti i nodi che hanno almeno un nodo figlio.

Connettivi boolenani: *and*, *or*, *not*.
Ad esempio,

```
/descendant::*[child::* and preceding-sibling::*]
/descendant::*[ancestor::B or preceding::B]
```

Altro esempio:
`/descendant::*[not(child::*)]`
Seleziona I nodi che non hanno un figlio, cioè le foglie

Esempio
`/descendant::*[following-sibling::*/following-sibling::*]`
Prendiamo i nodi che hanno un figlio e almeno un fratello a sinistra, come il nodo R.



Andiamo su Basex, riga di comando Find. E copia e incolla le istruzioni del file play.dtd:

```
<!ELEMENT TITLE     (#PCDATA)>
<!ELEMENT FM        (P+)>
<!ELEMENT P         (#PCDATA)>
<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
```

I comandi seguenti non so da dove vengano ne dove vadano inseriti

All'interno dei filtri posso usare una funzione di nome *Contains*. Lei Verifica se nel primo argomento è incluso la stringa del secondo argomento, il punto è una abbreviazione per un altro asse self star che è un asse che sta fermo, non si muove. Ad esempio

```
/PLAY/PERSONAE/PERSONA[contains(., "Theseus")
```

Dice che voglio tutte le persone il cui valore stringa, ossia il nodo testo, valga Theseus. In maiuscolo (case sensitive).

Non so ove inserire i cmd: Basex?

Il doppio slash // è un'abbreviazione per l'asse descendant. Ad esempio,

```
//SCENE [parent::ACT[TITLE = "ACT I"]]
```

Da come risultato tutte le scene dell'opera. Va bene anche `\descendant` al posto di \\.

Registrazione 343

4.4.2 XQuery (o XML Query Language)

Istruzioni tratte da qui Ora andremo ad analizzare XQuery. L'input di ogni interrogazione è una sequenza dove ogni elemento della sequenza o sono elementi XML oppure valori atopici come stringhe e, ecc. Il legame tra XPath e XQuery è che XPath è una parte di XQuery, cioè XQuery usa XPath. Tipicamente il processo di un XQuery è il seguente:

1. Carico uno o più documenti XML che voglio interrogare
2. Uso XPath per recuperare sequenze di nodi da questi documenti
3. Uso delle primitive, cioè delle clausole specifiche di XQuery per elaborare questo insieme di nodi
4. Costruisco un risultato
5. Restituisco il risultato della mia interrogazione

Alcuni di questi passi sono facoltativi e si può vedere XQuery come un linguaggio di programmazione su sequenze, quindi è possibile scrivere qualsiasi funzione computabile. Le interrogazioni in XQuery si chiamano **FLWOR expression** (leggi flower come fiore)., il cui nome è acronimo delle clausole che definiscono l'interrogazione che sono al pari di select, group by, ecc. di SQL. Vediamo le clausole nello specifico:

FLWOR expressions

- **For: l**a clausola for associa una o più variabili ad una espressione XQuery, ricordando che il risultato di una espressione è una sequenza di nodi XML. Questa clausola itera sul risultato di questa sequenza, proprio come il for nel linguaggio di programmazione (cicla al solito modo)
- **Let**: La clausola let assegna sempre una variabile ad una espressione, però con una semantica diversa, perché valuta l'espressione e assegna il risultato ad una variabile, non esegue l'istruzione in modo iterativo.

Alla fine di queste due clausole l'assegnamento variabile e valore prende il nome di **tupla**, come per in SQL. a. A questo punto le prossime clausole servono per far qualcosa con queste tuple.

- **where**: filtra le tuple
- **order by**: ordina le tuple
- **return**: restituire un risultato

Le uniche obbligatorie sono una tra *for* e *let,* e *return,* mentre le altre sono opzionali. Vediamo tale linguaggio tramite un esempio su BaseX. Su opera di Shakespeare.

```
for $act in //ACT
return <act>
         {$act/TITLE}
         <count>{count($act/SCENE)}</count>
       </act>
```

Tutte le variabili sono prefissate dal dollaro: *$variabile*. L'istruzione dice di assegnare in modo iterativo la variabile act a tutti i nodi restituiti dalla query *//ACT*. Per ogni assegnamento chiamo la clausola return che costruisce un elemento XML detto act che al suo interno ha un primo elemento che è il titolo degli atti: `{$act/TITLE}` fra parentesi graffe nel senso che voglio valutare l'espressione e mettere risultato nell'output. Il primo elemento richiesto quindi è il titolo degli atti. {count($act/SCENE)} restituisce nivee il numero di scene in tali atti. La prossima query è equivalente:

```
for $act in //ACT
let $scene := $act/SCENE
return <act>
         {$act/TITLE}
         <count>{count($scene)}</count>
       </act>
```

Usiamo la clau

sola Let che associa una variabile ad un risultato intero. Si noti il **:=** per l'assegnazione e l'in per l'iterazione del form (ocio).

Ora vogliamo tutti i titoli e il conto degli speach per gli atti che hanno più di 100 speech:

```
for $act in //ACT
let $speech := $act//SPEECH
where count($speech) > 100
return <act>
         {$act/TITLE}
         <count>{count($speech)}</count>
        </act>
```

Vediamo ora tutti i titoli e le battute (speech) ordinate in ordine decrescente rispetto al numero di battute:

```
for $act in //ACT
let $line := $act//LINE
order by -count($line)
return <act>
         {$act/TITLE}
         <count>{count($line)}</count>
       </act>
```

Order by è la sola differenza, ordina il numero di battute. Si noti che il risultato non è esattamente un documento XML, pur avendo comunque gli elementi con medesima definizione di validità.

Vediamo le ultime due, più complicate. Per commentare XQuery si faccia il sorriso :). Vogliamo ora trovare l'atto con il massimo numero di battute:

```
let $count := for $act in //ACT
              return count($act//LINE)
let $max := max($count)
for $act in //ACT
where count($act//LINE) = $max
return <act lines="{$max}">{$act/TITLE}</act>
```

Con Let assegno un'espressione che può essere XPATH o anche XQUERY. La prossima è simile alla precedente:

```
let $act := for $a in //ACT
            order by -count($a//LINE)
            return $a
return $act[1]/TITLE
```

La differenza è che la prima query restituisce anche un numero maggiore di due atti se hanno medesima lunghezza, mentre la seconda solo uno.

Infine XQuery ha una funzione *fulltext*, cioè fa delle interrogazioni orientate al testo, ad esempio tutte le battute che contengono la parola *hath*:

Validazione XML contro uno schema DTD

Una query che si ha saltato prima è la validazione, è possibile anche validare documenti rispetto a schemi XML, si mette il documento in una variabile doc, lo schema in una variabile schema e applico tale query:

```
# XML document in dream.xml
let $doc := doc('dream.xml')
let $schema := 'play.dtd'
return validate:dtd($doc, $schema)

# current XML database
let $schema := 'play.dtd'
return validate:dtd(., $schema)
```

Il risultato è ok. E' valido rispetto allo schema play.dtd

XQuery Full-Text

Infine XQuery ha una funzione fulltext, cioè fa delle interrogazioni orientate al testo, ad esempio il comando che restituisce tutte le linee che contengono la parola *hath* è:

```
//LINE[ . contains text "hath" ]
```

Se lo scrivessi in XPath avrei tutte quelle contengono la stringa *hath* e non la parola:

```
//LINE[contains(., "hath")]
```

C'è una differenza perché quando utilizziamo una query di tipo fulltext all'inizio c'è un processo che si chiama tokenziation in cui andiamo a prendere le parole e faccio un certo numero di trasformazioni. Ad esempio si mette tutto in minuscolo, si tolgono gli accenti, faccio lo stemming, etc. . L'effetto è che la prima query è esattamente uguale alla prossima query:

```
//LINE[ . contains text "...Hath!" ]
```

Vediamo alcuni esempi di combinazioni:
la prossima query restituisce le linee che contengono *death* e *Abjure* (case sensitive)

```
//LINE[ . contains text "death" ftand "Abjure" ]
```

La seguente query fa lo stemming in lingua inglese (es. restiuisce la riga per parole tipo *hated*

```
//LINE[ . contains text "hate" using stemming using language "en"]
```

Così eliminiamo le stop words, inserendo il file testo ove è contenuta la lista

```
//LINE[ . contains text "my very love" using stop words at
"http://files.basex.org/etc/stopwords.txt"]
```

Utilizziamo i caratteri speaciali (o wildecards):

```
//LINE[ . contains text ".*love.+" using wildcards]
```

Restituisce le battute che contengono qualsiasi cosa anche vuoto prima di love, e che contengono love, e che contengono qualsiasi cosa che segue love, purché abbia almeno un carattere

. corrisponde a un singolo carattere arbitrario
.? corrisponde a zero o a un carattere
. * corrisponde a zero o più caratteri
. + corrisponde a uno o più caratteri
. {min, max} corrisponde al numero min-max di caratteri

Restiuisce le linee che contengono *love* o uno dei suoi sinomini contenuti nel dizionario dei sinonimi

```
//LINE[ . contains text "love"  using thesaurus at "thesaurus.xml"]
```

Le righe contenenti la parola *hate* usando una ricerca non esatta (fuzzy), restituisce parole che sono vicine a hate, sfrutta la particolare distanza Levenshtein.

```
//LINE[ . contains text "hate" using fuzzy]
```

Le righe contenenti la parola *love* almeno 2 volte

```
//LINE[ . contains text "love" occurs at least 2 times ]
```

Le righe contenenti la parola *death* e *abjure* ad una distanza di almeno 2 parole

```
//LINE[ . contains text "death" ftand "abjure" distance at least 2 words ]
```

Per analizzare la rilevanza della parola, cioè un fattore che ha a che fare con il numero di volte in cui si presenta tale parola:

```
for $hit score $score in //LINE[ . contains text "love" ]
order by $score descending
return <hit score='{format-number($score, "0.00")}'>{$hit}</hit>
```

Le istruzioni dicono: quante volte la parola occorre nella battuta? Calcola una sorta di frequenza di presenza di una parola nella battuta. Poi viene assegnata un punteggio per ogni *love* nella battuta e ordinare in modo decrescente.

Si guardi attentamente l'esercizio, noi concludiamo qui tale trattazione perché non abbiamo il tempo di analizzare altro.

Se facciamo delle modifiche al documento XML dopo aver importato il DB in Basex tale modifica non si presenta, c'è tutta una parte di XQuery per modificare i documenti XML Xquery Update.

Registazione 344

4.7. make XML meets R

Esercitazione

A questo punto si è svolto l'esercizio XML makes R. In R si può eseguire XPath, ma non XQuery.

http://users.dimi.uniud.it/~massimo.franceschet/ds/r4ds/syllabus/learn/xml/xmlr.html

1. con BaseX, esegui un XQuery che estrae, per ogni atto, il numero di scene, discorsi e linee. Il risultato dovrebbe essere XML ben formato --> va aggiunto un elemento contenitore

Installare pacchetti XML and xml2

Registazione 345