Caffè XML

Sommario

Un bel trailer da iniziare: il Web 2.0 ... The Machine is Us it di Michael Wesch, Assistant Professor di Antropologia culturale presso la Kansas State University.

Il linguaggio XML

XML è l'acronimo di **Extensible Markup Language** . Ciò significa che:

- XML è un **linguaggio** formale . Ciò significa che XML è definito da un insieme di regole formali (una grammatica) che dicono esattamente come comporre un documento XML.
- XML consente di **contrassegnare i** tuoi dati. I dati sono inclusi nei documenti XML come stringhe di testo e sono circondati da una marcatura di testo che descrive i dati.
- XML è **estensibile** . La lingua consente un set estensibile di tag di markup che possono essere adattati per soddisfare molte esigenze diverse.

XML è semplicemente un formato flessibile per contrassegnare i dati con tag leggibili dall'uomo. Vale la pena sottolineare che un documento XML è un **documento di testo** e può essere letto e modificato con qualsiasi editor di testo. In particolare, **XML non è** un:

- **linguaggio di presentazione** come HTML. Il markup XML definisce il significato dei dati. Non dice nulla sullo stile dei dati.
- **linguaggio di programmazione** come Java. Un documento XML non calcola nulla.
- **protocollo di trasporto di rete** come HTTP. XML non trasferisce i dati attraverso la rete.
- **sistema di gestione di database** come Oracle. XML non memorizza e recupera i dati.

Un **parser XML** è un software che legge il documento XML e determina se è ben formato. Un documento **ben formato**aderisce alle regole grammaticali XML. Vale la

pena notare che un parser XML non può accettare un documento non valido e non è consentito provare a correggere il documento. E 'necessario segnalare gli errori.

XML è utile sia per gli esseri umani che per i computer. Gli scenari comuni in cui XML può essere utilizzato dalle persone includono:

- **crivere un libro** usando DocBook . Docbook è un'applicazione XML progettata per il markup di nuovi testi. È particolarmente comune nella documentazione del computer. Gran parte del corpus del Linux Documentation Project è scritto in DocBook. I vantaggi dell'utilizzo di DocBook includono: non è proprietario, portatile, modulare e facile da usare con qualsiasi editor di testo. Inoltre, è possibile formattare la versione finale in base alle proprie esigenze: PostScript per la stampa, DVI per visualizzare, HTML per la pubblicazione sul Web;
- **scrivere una pagina Web** in XHTML . XHTML definisce una versione di HTML compatibile con XML. A differenza dei documenti HTML, un documento XHTML valido può essere manipolato con gli stessi strumenti utilizzati per lavorare con XML. Inoltre, il documento XHTML può contenere altri markup appartenenti a una diversa applicazione XML, come un'immagine SVG o un'equazione MathML . I vantaggi dell'utilizzo di XHTML includono: ha una sintassi ben definita, puoi lavorare con qualsiasi strumento XML (per analizzare, convalidare, collegare e interrogare il tuo documento) e i motori di ricerca web alla fine capiranno il tuo documento e lo indicheranno correttamente.

I documenti risultanti sono chiamati documenti **incentrati sul testo** . Questi sono documenti XML solitamente scritti dagli umani per essere letti da altri umani. Sono documenti XML semipermanenti con molto testo e una struttura scadente.

Gli scenari comuni in cui XML può essere utilizzato dai computer includono:

- **scambio di dati** . Le informazioni provengono da fonti diverse (relazioni, oggetti, documenti) e devono essere scambiate tra queste fonti. L'XML potrebbe fungere da comune datacle;
- **database semistrutturati** . I dati semistrutturati non hanno schema regolare. Spesso include informazioni duplicate e mancanti. Quindi, non si adatta naturalmente ai database relazionali. XML è stato proposto come modello di dati per i dati semistrutturati.

I documenti risultanti sono chiamati documenti **incentrati sui dati** . Questi sono documenti XML solitamente scritti da computer per essere letti da altri computer. Sono documenti XML transitori con una struttura ricca e molti dati grezzi.

Per ragioni di interoperabilità, le organizzazioni possono accettare di utilizzare solo determinati tag. Questi set di tag sono chiamati **applicazioni XML** . Gli esempi sono:

- XHTML : definisce una versione XML compatibile di HTML.
- Scalable Vector Graphics (SVG): è un linguaggio per descrivere grafica bidimensionale in XML.
- Mathematical Markup Language (MathML): è un linguaggio per descrivere la notazione matematica.

I namespace sono un meccanismo che consente a un documento XML di combinare diverse applicazioni XML. Hanno due scopi in XML:

1. Per distinguere tra elementi e attributi da diversi vocabolari con significati diversi e che capita di condividere lo stesso nome.
2. Raggruppare tutti gli elementi e gli attributi correlati da un'unica applicazione XML in modo che il software possa riconoscerli.

Esempio

Il seguente documento XML contiene una piccola bibliografia:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bibliography>
  <article key="G03">
    <author>
      <name>Georg</name>
      <surname>Gottlob</surname>
    </author>
    <title>XPath processing in a nutshell</title>
    <year>2003</year>
    <journal>SIGMOD Records</journal>
  </article>
  <book key="HM04" isbn="0-596-00764-7">
    <author>
      <name>Elliotte</name>
      <name>Rusty</name>
      <surname>Harold</surname>
    </author>
    <author>
      <surname>Means</surname>
    </author>
    <title>XML in a nutshell</title>
    <year>2004</year>
    <cite item="G03"/>
    <publisher>O'Reilly</publisher>
  </book>
</bibliography>
```

Ecco una versione estesa che include l'applicazione XTHML e che utilizza gli spazi dei nomi.

Il modo più semplice per analizzare un documento è caricarlo in un browser Web che conosce XML. Il browser mostrerà il documento ogni volta che è ben formato o segnalerà gli errori in caso contrario. In alternativa, si può usare un parser XML standalone come lo strumento da riga di comando xmllint , che fa parte della libreria

XML libxml sviluppata per il progetto Gnome (ma utilizzabile al di fuori della piattaforma Gnome).

Lingue dello schema: DTD

Il markup permesso in una particolare applicazione XML può essere documentato in uno **schema** . Il linguaggio di schema più ampiamente supportato e l'unico definito dalla specifica XML 1.0 è il **Document Type Definition** (DTD).

Un DTD consente di posizionare alcuni vincoli sulla **struttura di** un documento XML. Elenca tutti gli elementi, gli attributi e le entità utilizzati dal documento e il contesto in cui vengono utilizzati. I DTD non dicono mai nulla sul tipo di contenuto di un elemento o sul valore di un attributo. Ad esempio, non puoi dire che il prezzo è un numero reale, o il nome è una stringa, o nato è una data.

Un documento XML è ritenuto **valido** se aderisce alle definizioni della DTD associata. Come regola generale, i browser Web non convalidano i documenti ma li controllano solo per la loro idoneità. Se si sta sviluppando un'applicazione, è possibile utilizzare l'API del parser per convalidare il documento. Se stai scrivendo documenti a portata di mano, puoi utilizzare un validatore online o scaricare ed eseguire un programma locale.

Example

The following DTD contains a DTD for the bibliography example:

```
<!ELEMENT bibliography     (article | book)*>

<!ELEMENT article          (author+, title, year, cite*, journal)>
<!ATTLIST article          key ID #REQUIRED>

<!ELEMENT book             (author+, title, year, cite*, publisher)>
<!ATTLIST book             key ID #REQUIRED
                           isbn CDATA #REQUIRED>

<!ELEMENT cite             EMPTY>
<!ATTLIST cite             item IDREF #REQUIRED>

<!ELEMENT author           (name*, surname+)>

<!ELEMENT name             (#PCDATA)>
<!ELEMENT surname          (#PCDATA)>
<!ELEMENT title            (#PCDATA)>
<!ELEMENT year             (#PCDATA)>
<!ELEMENT journal          (#PCDATA)>
<! ELEMENT editore (#PCDATA)>
```

Ecco una DTD per la versione estesa .

Un esempio di validatore online è il modulo di convalida XML del Brown University Scholarly Technology Group. Un parser e validatore XML della riga di comando

è xmllint . Ad esempio, puoi validare il nostro esempio di bibliografia`bib.xml`contro una DTD `bib.dtd` con la seguente sintassi:

```
xmllint --dtdvalid bib.dtd bib.xml
```

Schema languages: W3C XML Schema

La DTD presenta alcuni gravi inconvenienti, in particolare:

1. non ha idea di **tipo** . Il contenuto degli elementi fogliari o degli attributi può essere qualsiasi dato carattere o nessuno. L'assegnazione di un tipo a un elemento o un attributo aggiunge semantica a quell'elemento o attributo;
2. il **meccanismo di riferimento** è troppo semplice, ad esempio non è possibile limitare l'ambito dell'unicità per gli attributi ID a un frammento dell'intero documento. Inoltre, solo i singoli attributi possono essere usati come chiavi;
3. non è descritto nella **notazione XML** , che sarebbe stata utile per manipolare gli schemi con strumenti XML, ad esempio per verificare che un DTD sia ben formato o per interrogare gli schemi.

XML Schema è una proposta del W3C che risolve questi problemi. In particolare, contiene un potente **sistema di tipi** che consente di definire tipi semplici e complessi e anche di ereditare tipi di altri tipi nello stile dei linguaggi di programmazione orientati agli oggetti. I tipi possono essere associati ad elementi e attributi, aggiungendo significato alle loro interpretazioni. Sfortunatamente, questo ha un prezzo: lo schema XML è generalmente **complicato da comprendere** e **difficile da usare** per i non esperti (infatti, le specifiche del W3C sono difficili da leggere anche per gli esperti di XML!).

Esempio

Il seguente schema XML contiene uno schema XML per l'esempio bibliografico:

```
<? xml version = "1.0"?>
<xs: schema xmlns: xs = "http://www.w3.org/2001/XMLSchema">

  <! - Dichiarazioni elemento ->
  <xs: element name = "author" type = "authorType" />
  <xs: element name = "name" type = "xs: string" />
  <xs: element name = "cognome" type = "xs: string" />
  <xs: element name = "title" type = "xs: string" />
  <xs: element name = "year" type = "xs: gYear" />
  <xs: element name = "cite" type = "citeType" />
  <xs: element name = "journal" type = "xs: string" />
  <xs: element name = "publisher" type = "xs: string" />
  <xs: element name = "coreItem" type = "coreItemType" />
  <xs: element name = "article" type = "articleType" />
  <xs: element name = "book" type = "bookType" />
  <xs: element name = "bibliography" type = "bibliographyType">
    <! - Vincoli chiave ->
    <xs: key name = "primaryKey">
      <xs: selector xpath = "*" />
      <xs: campo xpath = "@ chiave" />
    </ Xs: key>
```

```
  <xs: keyref name = "foreignKey" refer = "primaryKey">
    <xs: selector xpath = "* / cite" />
    <xs: field xpath = "@ item" />
  </ Xs: keyref>
</ Xs: element>

<! - Dichiarazioni degli attributi ->
<xs: attribute name = "isbn" type = "isbnType" />
<xs: attribute name = "key" type = "xs: string" />
<xs: attribute name = "item" type = "xs: string" />

<! - Definizioni di tipo ->
<xs: simpleType name = "isbnType">
  <xs: restriction base = "xs: string">
    <xs: pattern value = "\ d- \ d \ d \ d- \ d \ d \ d \ d \ d- \ d" />
  </ Xs: restrizione>
</ Xs: simpleType>

<xs: complexType name = "citeType">
  <xs: attribute ref = "item" />
</ Xs: complexType>

<xs: complexType name = "authorType">
  <xs: sequence>
    <xs: element ref = "name" minOccurs = "0" maxOccurs = "illimitato" />
    <xs: element ref = "cognome" maxOccurs = "non limitato" />
  </ Xs: sequence>
</ Xs: complexType>

<xs: complexType name = "coreItemType">
  <xs: sequence>
    <xs: element ref = "author" maxOccurs = "non limitato" />
    <xs: element ref = "title" />
    <xs: element ref = "year" />
    <xs: element ref = "cite" minOccurs = "0" maxOccurs = "non limitato" />
  </ Xs: sequence>
  <xs: attribute ref = "key" />
</ Xs: complexType>

<xs: complexType name = "articleType">
  <Xs: complexContent>
    <xs: extension base = "coreItemType">
      <xs: sequence>
        <xs: element ref = "journal" />
      </ Xs: sequence>
    </ Xs: estensione>
  </ Xs: complexContent>
</ Xs: complexType>

<xs: complexType name = "bookType">
  <Xs: complexContent>
    <xs: extension base = "coreItemType">
      <xs: sequence>
        <xs: element ref = "publisher" />
      </ Xs: sequence>
      <xs: attribute ref = "isbn" />
    </ Xs: estensione>
  </ Xs: complexContent>
</ Xs: complexType>

<xs: complexType name = "bibliographyType">
  <xs: choice minOccurs = "0" maxOccurs = "illimitato">
```

```
      <xs: element ref = "article" />
      <xs: element ref = "book" />
    </ Xs: scelta>
  </ Xs: complexType>

</ Xs: schema>
```

Ecco uno schema per la versione estesa .

La convalida di un documento con uno schema XML richiede un **parser di convalida** che supporti lo schema XML come il parser Xerces open source dal progetto Apache Xerces. Questo è scritto in Java e include, nell'archivio`xercesSamples.jar`, un programma da riga di comando `jaxp.SourceValidator`che può essere usato per convalidare. La sintassi per`jaxp.SourceValidator` segue:

```
java jaxp.SourceValidator -i bib.xml -a bib.xsd
```

Query languages: XPath

XML path language (**XPath**) è un linguaggio semplice per recuperare elementi XML da un *singolo* documento XML. XPath può essere sfruttato in diverse tecnologie XML: da solo come semplice linguaggio di query per XML, in XQuery per recuperare elementi XML che possono essere ulteriormente elaborati per risolvere una query, in XSLT per recuperare gli elementi a cui vengono applicate le regole del modello in per trasformare un documento XML, in W3C XML Schema per individuare chiavi e riferimenti chiave e infine in XPointer per puntare a particolari elementi XML nel documento XML collegato.

XPath visualizza un documento XML come una **struttura ad albero** : gli elementi sono mappati ai nodi e i sottoelementi corrispondono ai nodi figli. Puoi convertire il tuo documento XML nella sua rappresentazione ad albero usando lo strumento XMLTree combinato con GraphViz . Ad esempio, il nostro documento bibliografico è mappato a questo albero .

Seguono alcuni esempi di query XPath:

All books in the bibliography

```
/bibliography/book
```

All books published in 2004

```
/descendant::book[year = "2004"]
```

All articles written by Georg Gottlob

```
/descendant::article[author[name = "Georg" and surname = "Gottlob"]]
```

All articles that follows in the document the one written by Georg Gottlob

```
/descendant::article[author[name = "Georg" and surname = "Gottlob"]]/following-
sibling::article
```

The first author of the last bib item

```
/bibliography/*[position() = last()]/author[position() = 1]
```

The number of authors of the bib item with key HM04

```
count(/id("HM04")/author)
```

The bib items cited in bib item with key HM04

```
id(/id("HM04")/cite/@item)
```

Puoi provare tutte le query precedenti con qualsiasi processore XPath. Un **processore XPath** è un software che valuta le query XPath. BaseX è un completo processore XPath basato su Java con una bella interfaccia grafica web. Mostra risultati in diversi formati tra cui testo, albero e mappa ad albero. Saxon è un processore da riga di comando XSLT e XQuery. Poiché XPath è utilizzato sia in XSLT che in XQuery, puoi provare anche le query XPath con Saxon. Ad esempio, per valutare la query `/ /descendant::article` sul documento XML `bib.xml`, esegui il seguente comando:

```
java net.sf.saxon.Query -s bib.xml "{/ descendant :: article}"
```

L'opzione `-S` imposta il nodo di contesto iniziale sulla radice del documento XML specificato. Se preferisci memorizzare la query nel file `articles.xpl`, quindi puoi digitare il seguente comando:

```
java net.sf.saxon.Query -s bib.xml articles.xpl
```

Query languages: XQuery

Il **linguaggio di query XML** (**XQuery**) è un linguaggio di query completo per i database XML. Si distingue per i database XML in quanto SQL corrisponde a quelli relazionali. Un **database XML** è una raccolta di documenti XML (correlati).

XQuery funziona su **sequenze** , non su set di nodi come XPath. Una sequenza contiene elementi che sono o elementi XML o valori atomici (come una stringa o un numero). La relazione tra XPath e XQuery consiste nel fatto che le espressioni XPath vengono utilizzate nelle query XQuery. Quindi, possiamo considerare XPath come un frammento sintattico di XQuery.

Un'espressione tipica in XQuery funziona come segue:

1. **caricamento** : uno o più documenti XML vengono caricati dal database;

2. **recupero** : le espressioni XPath vengono utilizzate per recuperare sequenze di nodi ad albero dai documenti caricati;
3. **processo** : le sequenze di nodi recuperate vengono elaborate con operazioni XQuery come il filtraggio (creando una nuova sequenza selezionando alcuni degli elementi di quella originale) e ordinando (ordinando gli elementi della sequenza in base ad alcuni criteri);
4. **costrutto** : nuove sequenze possono essere costruite e combinate con quelle recuperate;
5. **output** : una sequenza finale viene restituita come output.

Seguono alcuni esempi di istruzioni XQuery:

Il titolo e l'anno di pubblicazione di tutti gli articoli bib con più di un autore ordinati per anno

```
per $ item in doc ("bib.xml") / bibliography / *
where count($item/author) > 1
order by $item/year
return <item key = "{$item/@key}">
         {$item/title}
         {$item/year}
       </item>
```

The bib items that are cited by at least one other item

```
let $doc := doc("bib.xml")
for $item in $doc/bibliography/*
let $citation := for $c in $doc/descendant::cite
                 where $c/@item = $item/@key
                 return $c
where count($citation) > 0
return <item key = "{$item/@key}"/>
```

Un **processore XQuery** è un software che valuta le query in XQuery. Vedi la pagina delle risorse XQuery per un elenco di processori XQuery. Saxonè un buon esempio. Al fine di valutare con Saxon la query contenuta nel file`xquery.xql`, digitare quanto segue:

```
java net.sf.saxon.Query xquery.xql
```

Sfortunatamente, al momento non esiste una lingua standard per aggiornare un documento XML.

Stylesheet languages: XSLT

Extensible Stylesheet Language Transformations (XSLT) è un'applicazione XML per **trasformare** un documento XML in un altro documento in un formato (come XML, HTML, testo normale). Poiché una tipica applicazione di XSLT è quella di rendere le

informazioni contenute nel documento XML mappando il documento XML in uno HTML, i documenti XSLT sono anche chiamati **fogli di stile XSLT** .

Esempio

Ecco un esempio di foglio di stile XSLT per mappare il nostro documento XML di bibliografia in un documento HTML:

```
<? xml version = "1.0"?>
<xsl:      stylesheet      version      =      "2.0"      xmlns:      xsl      =
"http://www.w3.org/1999/XSL/Transform">

  <xsl: template match = "/">
    <Html>
      <Head>
        <title> Una piccola bibliografia </ title>
      </ Head>
      <Body>
        <h1> Una piccola bibliografia </ h1>

        <xsl: if test = "bibliography / article">
          <H2> Articoli </ h2>
          <Ol>
            <xsl: apply-templates select = "bibliografia / articolo">
              <xsl: sort select = "year" order = "descending" />
            </ Xsl: apply-templates>
          </ Ol>
        </ Xsl: if>

        <xsl: if test = "bibliografia / libro">
          <H2> Libri </ h2>
          <Ol>
            <xsl: apply-templates select = "bibliografia / libro">
              <xsl: sort select = "year" order = "descending" />
            </ Xsl: apply-templates>
          </ Ol>
        </ Xsl: if>

      </ Body>
    </ Html>
  </ Xsl: template>

  <xsl: template match = "bibliography / article">
    <a name="{@key}"/>
    <Li>
      <xsl: apply-templates select = "author">
        <xsl: sort select = "cognome"> </ xsl: sort>
      </ Xsl: apply-templates>
      <b> <xsl: apply-templates select = "title" /> </ b>.
      <xsl: apply-templates select = "journal" />,
      <xsl: apply-templates select = "year" />.
      <xsl: if test = "cite">
        Riferimenti: <xsl: apply-templates select = "cite" />
      </ Xsl: if>
    </ Li>
  </ Xsl: template>

  <xsl: template match = "bibliography / book">
    <a name="{@key}"/>
```

```
    <Li>
      <xsl: apply-templates select = "author">
        <xsl: sort select = "cognome"> </ xsl: sort>
      </ Xsl: apply-templates>
      <b> <xsl: apply-templates select = "title" /> </ b>,
      <xsl: apply-templates select = "year" />.
      <xsl: apply-templates select = "editore" />.
      <xsl: apply-templates select = "@ isbn" />.
      <xsl: if test = "cite">
        Riferimenti: <xsl: apply-templates select = "cite" />
      </ Xsl: if>
    </ Li>
  </ Xsl: template>

  <xsl: template match = "author">
    <xsl: apply-templates select = "nome" />
    <xsl: apply-templates select = "cognome" />
    <xsl: if test = "position ()! = last ()">, </ xsl: if>
    <xsl: if test = "position () = last ()">. </ Xsl: if>
  </ Xsl: template>

  <xsl: template match = "name">
    <Xsl: apply-templates />
    <xsl: value-of select = "string ('')" />
  </ Xsl: template>

  <xsl: template match = "cite">
    <a href="#{@item}">
      <xsl: apply-templates select = "@ item" />
    </a>
  </ Xsl: template>

</ Xsl: stylesheet>
```

Ecco un XSLT che funziona per la versione estesa .

Un foglio di stile XSLT è associato a un documento XML utilizzando l'istruzione di elaborazione con il **xml-stylesheet** destinazione . Ad esempio, possiamo associare il foglio di stile contenuto nel file`bib.xsl` al nostro documento XML di bibliografia aggiungendo le seguenti istruzioni al documento XML:

```
<? xml-stylesheet type = "application / xml" href = "bib.xsl"?>
```

Un **processore XSLT** è un software che immette un documento XML e un corrispondente foglio di stile XSLT e genera il documento risultato applicando il foglio di stile al documento XML. Un processore XSLT può essere integrato in un browser web, come Mozilla TransforMiiX . Oppure può essere un programma autonomo come Saxon . Con Saxon, possiamo applicare`bib.xsl` a `bib.xml` con la seguente sintassi:

```
java net.sf.saxon.Transform bib.xml bib.xsl
```

Se il documento XML contiene le istruzioni di elaborazione del foglio di stile, è possibile utilizzare questa sintassi:

```
java net.sf.saxon.Transform -a bib.xml
```

Inoltre, utilizzare l'opzione `-snone` per preservare i nodi di testo degli spazi bianchi nel documento XML e l'opzione `archiviarlo` per inviare l'output al file indicato.

Stylesheet languages: CSS

I nomi degli elementi XML descrivono il **significato** dei loro contenuti. Tuttavia, non dicono nulla sulla presentazione del contenuto. Il CSS è un linguaggio per descrivere l' **aspetto** degli elementi in un documento. Non modifica il markup di un documento XML ma applica semplicemente le regole di presentazione al contenuto esistente.

Esempio

Ecco un esempio di documento XML in stile CSS. Infatti, più fogli di stile sono stati associati al documento. Se si utilizza Mozilla Firefox, è possibile alternare fogli di stile diversi con l'opzione di menu Visualizza / Stile pagina.

Il foglio di stile per un documento XML viene specificato con l' **istruzione di elaborazione** `xml-stylesheet`nel prologo del documento XML. Ad esempio, un foglio di stile CSS per la nostra bibliografia può essere associato al documento XML di bibliografia con le seguenti istruzioni di elaborazione:

```
<? xml-stylesheet type = "text / css" href = "bib.css"?>
```

Extensible Markup Language

XML è l'acronimo di **Extensible Markup Language** . Ciò significa che:

- XML è un **linguaggio** formale . Ciò significa che XML è definito da un insieme di regole formali (una grammatica) che dicono esattamente come comporre un documento XML.
- XML consente di **contrassegnare i** tuoi dati. I dati sono inclusi nei documenti XML come stringhe di testo e sono circondati da una marcatura di testo che descrive i dati.
- XML è **estensibile** . La lingua consente un set estensibile di tag di markup che possono essere adattati per soddisfare molte esigenze diverse.

XML è semplicemente un formato flessibile per contrassegnare i dati con tag leggibili dall'uomo. Vale la pena sottolineare che un documento XML è un **documento di testo** e può essere letto e modificato con qualsiasi editor di testo. In particolare, **XML non è** un:

- **linguaggio di presentazione** come HTML. Il markup XML definisce il significato dei dati. Non dice nulla sullo stile dei dati.

- **linguaggio di programmazione** come Java. Un documento XML non calcola nulla.
- **protocollo di trasporto di rete** come HTTP. XML non trasferisce i dati attraverso la rete.
- **sistema di gestione di database** come Oracle. XML non memorizza e recupera i dati.

XML è l'acronimo di **Extensible Markup Language** . Ciò significa che:

- XML è un **linguaggio** formale . Ciò significa che XML è definito da un insieme di regole formali (una grammatica) che dicono esattamente come comporre un documento XML.
- XML consente di **contrassegnare i** tuoi dati. I dati sono inclusi nei documenti XML come stringhe di testo e sono circondati da una marcatura di testo che descrive i dati.
- XML è **estensibile** . La lingua consente un set estensibile di tag di markup che possono essere adattati per soddisfare molte esigenze diverse.

XML è semplicemente un formato flessibile per contrassegnare i dati con tag leggibili dall'uomo. Vale la pena sottolineare che un documento XML è un **documento di testo** e può essere letto e modificato con qualsiasi editor di testo. In particolare, **XML non è** un:

- **linguaggio di presentazione** come HTML. Il markup XML definisce il significato dei dati. Non dice nulla sullo stile dei dati.
- **linguaggio di programmazione** come Java. Un documento XML non calcola nulla.
- **protocollo di trasporto di rete** come HTTP. XML non trasferisce i dati attraverso la rete.
- **sistema di gestione di database** come Oracle. XML non memorizza e recupera i dati.

Un documento XML è costruito dal **testo** . Il testo è contrassegnato con **tag di** testo . I tag di testo sono racchiusi tra parentesi angolari. Ecco un documento XML molto semplice ma completo:

```
<Persona>
  Alan Turing
</ Persona>
```

Nel documento sopra c'è un singolo **elemento di** nome person. L'elemento è delimitato dal **tag di inizio**`<Persona>`e il **tag di fine** `</ Persona>`. I tag sono una forma di markup. Tutto ciò che si trova tra il tag di inizio e il tag di fine è chiamato il **contenuto** dell'elemento. In questo caso, il contenuto del`persona` elemento è la stringa `Alan Turing`.

I tag XML sono simili a quelli HTML. Tuttavia, ci sono due grandi differenze. In XML puoi inventare i tuoi tag, mentre il set di tag HTML è fisso. Ancora più importante, i tag XML hanno lo scopo di descrivere il **tipo di contenuto** piuttosto che la formattazione o le informazioni di layout. In altre parole, in XML non si dice che qualcosa è in corsivo o rientrato, si dice che qualcosa è una persona o un nome di persona o un indirizzo di persona.

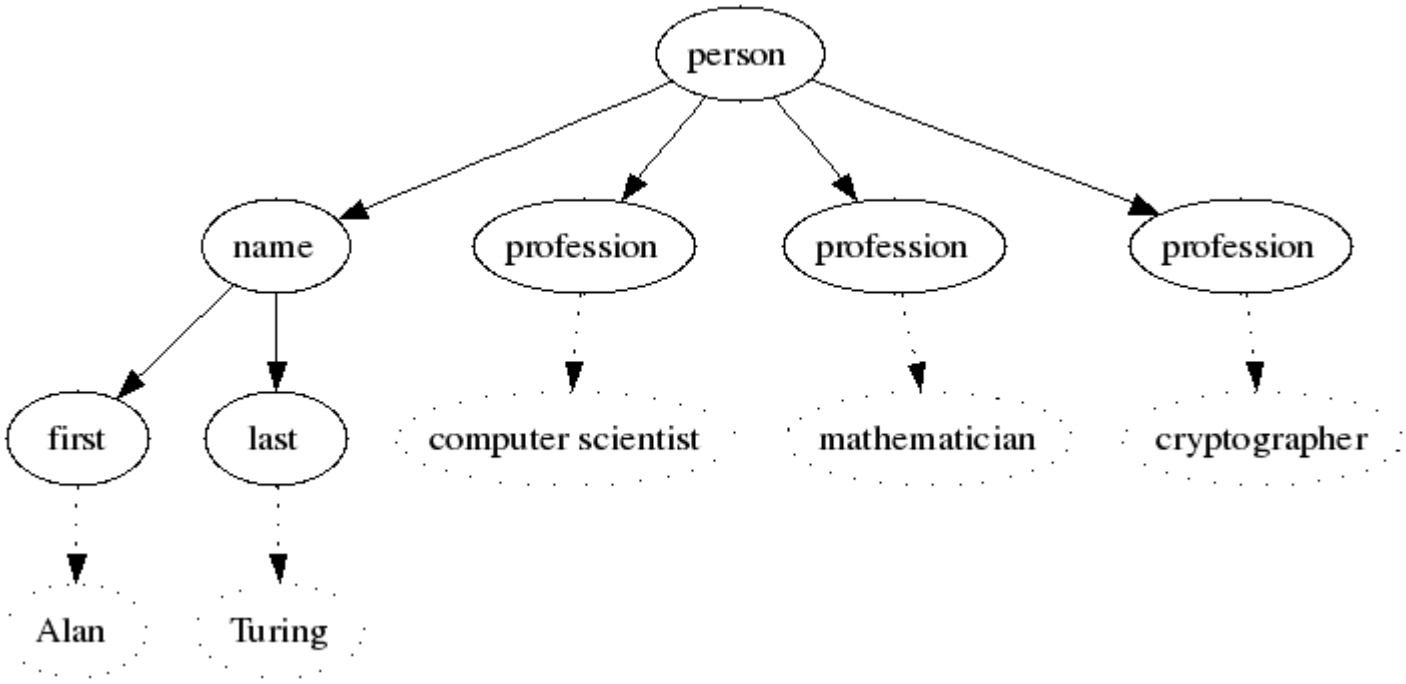
Gli elementi XML possono contenere testo non contrassegnato (chiamato dati carattere) o altri elementi XML. Ecco un esempio più coinvolgente:

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

Il `persona` l'elemento contiene a `nome` sottoelemento e tre `professione`sottoelementi. Inoltre, il`nome` l'elemento contiene a `primo` e a `scorso`elementi figlio. Il`professione`, `primo` e `scorso`gli elementi contengono solo dati di carattere (cioè testo senza markup). Un documento XML deve rispettare le seguenti regole:

- **gli elementi non possono sovrapporsi** : se il tag di inizio dell'elemento B segue nel documento il tag di inizio dell'elemento A, allora il tag di fine dell'elemento B deve precedere nel documento il tag di fine dell'elemento A;
- deve esistere un **elemento documento** univoco che racchiuda tutti gli altri elementi.

Di conseguenza, un documento XML può essere rappresentato come una **struttura ad albero** : gli elementi sono mappati ai nodi (l'elemento del documento corrispondente alla radice dell'albero) e i sottoelementi corrispondono ai nodi figli. Ad esempio, il documento XML sopra riportato è mappato all'albero seguente:



Gli elementi possono anche combinare dati di carattere e sottoelementi come nell'esempio seguente:

```
<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>
```

Elementi vuoti, cioè elementi senza contenuto, come anche possibile. Un elemento vuoto chiamato`indirizzo` può essere abbreviato come segue: `<Indirizzo />`. Infine, ricorda che l'XML è case sensitive, quindi`indirizzo` e `Indirizzo` sono tag diversi.

Gli elementi XML possono avere **attributi** . Queste sono proprietà che valgono per l'intero elemento. Un attributo ha la sintassi`nome = "valore"`, dove `nome` è il nome dell'attributo e `valore`è una stringa. Le virgolette che racchiudono il valore possono essere singole o doppie. Un elemento può avere qualsiasi numero di attributi, ma i loro nomi devono essere distinti. L'ordine degli attributi non è significativo. Un esempio segue:

```
<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>
```

Alcune informazioni potrebbero essere ugualmente codificate come valori di attributo o contenuto di elementi. L'esempio seguente mostra fino a che punto puoi andare con gli attributi:

```
<persona nata = "23/06/1912" è morta = "07/06/1954">
  <name first = "Alan" last = "Turing" />
  <valore value = "informatico" />
  <valore value = "matematico" />
  <valore value = "cryptographer" />
</ Persona>
```

Quindi, qual è la differenza tra l'uso di attributi e elementi per contenere le informazioni? La scelta spetta a te, ma ricorda che ogni elemento non può avere più di un attributo con un nome specifico. Inoltre, gli attributi hanno una struttura piuttosto limitata: il valore di un attributo è semplicemente una stringa. Una buona regola è usare gli attributi per i metadati sull'elemento mentre gli elementi per l'informazione stessa.

Quali sono i nomi validi per gli elementi e gli attributi XML? Un nome valido per loro è chiamato un **nome XML** e si è formato come segue:

- può contenere qualsiasi carattere alfanumerico, incluse lettere, numeri e ideogrammi non inglesi;
- può includere solo tre caratteri di punteggiatura: underscore (_), trattino (-), punto (.). Il colon (:) è possibile ma è riservato.
- non può contenere alcun tipo di spazio (come spazio, ritorno a capo, avanzamento riga);
- non può iniziare con un numero, un trattino o un punto;
- non può iniziare con la stringa riservata `xml` (in qualsiasi combinazione di casi).

Gli elementi e gli attributi sono i principali componenti XML. Tuttavia, è possibile inserire altro in un documento XML, vale a dire:

- **commenti** . Un commento inizia con`<!--` e finisce con `-->`. I commenti possono essere utilizzati ovunque, anche al di fuori dell'elemento del documento, ma potrebbero non essere visualizzati all'interno dei tag e all'interno di altri commenti. I commenti sono destinati ai lettori umani. Non scrivere documenti che dipendono dal contenuto dei commenti;
- **istruzioni di elaborazione** . Le istruzioni di elaborazione contengono informazioni per applicazioni che possono leggere il documento, non per i lettori umani. Iniziano con`<?` e finisci con `?>`. Seguendo il`<?`è un nome XML chiamato target, probabilmente il nome dell'applicazione a cui è destinata l'istruzione. Il resto delle istruzioni contiene testo in un formato appropriato per l'applicazione. Possono essere utilizzati ovunque, anche al di fuori dell'elemento del documento, ma potrebbero non essere visualizzati all'interno dei tag. L'istruzione di elaborazione più comune è`xml-stylesheet`, utilizzato per allegare fogli di stile ai documenti XML. Ecco un esempio:

```
<? xml-stylesheet type = "text / css" href = "recipe.css"?>
```

- **sezioni di dati di carattere** . Le sezioni dei Character data (CDATA) sono blocchi di dati dei caratteri trattati come dati di testo non elaborati. Cioè, se markup è presente è una sezione CDATA, quindi non viene trattata come markup ma come dati carattere. Inizia una sezione CDATA con`<! [CDATA [` e finisce con `] ]>`. Le sezioni CDATA possono essere posizionate ovunque sia possibile posizionare un elemento. Esistono per comodità di autori umani, non per programmi. Un esempio segue:

```
<paragraph>
SVG is an XML encoding of line art. For instance, the
following encodes an ellipse and a rectangle:
</paragraph>

<![CDATA[
<svg width="12cm" height="10cm">
<ellipse rx="110" ry="130"/>
<rect x="4cm" y="1cm" width="3cm" height="6cm"/>
</svg>
]]>
```

- **riferimenti di entità** . Un'entità in XML è un nome per una porzione di testo. Alcune entità sono predefinite, altre potrebbero essere definite dall'utente. Esistono cinque riferimenti di entità XML predefiniti:
  - `& Lt;` per il segno di meno di (<);
  - `& Gt;` per il segno maggiore di (>);
  - `& Amp;` per la e commerciale (&)
  - `& Quot;` per le virgolette doppie diritte (")
  - `& APOS;` per le virgolette semplici (')

  Questi caratteri sono utilizzati nella sintassi XML. Quindi, le entità predefinite sono utili quando vogliamo scrivere questi caratteri con un significato diverso da quello dato dalla sintassi XML, come nell'esempio seguente:

```
<tutorial>
<name>&lt;Caffè XML/&gt;</name>
</tutorial>
```

  Il contenuto del `nome` elemento è la stringa <Caffè XML /> e non un elemento vuoto.

riferimenti di carattere . Se un carattere non è accessibile dall'editor che stai utilizzando, puoi scriverlo come riferimento di carattere. Un riferimento di carattere fornisce un numero del particolare carattere Unicode che rappresenta. Il numero dovrebbe essere preceduto da un segno # e può essere in entrambi i decimali (ad es `&#1114;` per il carattere ћ) o esadecimale (es. `& # X45A;` per lo stesso personaggio). È possibile

utilizzare riferimenti di caratteri nel contenuto di elementi, valori di attributo e commenti, ma non nei nomi di elementi e attributi. Ecco un esempio:

```
<paragraph>
The following is a Greek maxim saying: "The wise man knows himself".
</paragraph>

<maxim>
&#x3C3;&#x3BF;&#x3C6;&#x3CC;&#x3C2;
&#x3AD;&#x3B1;&#x3C5;&#x3C4;&#x3CC;&#x3BD;
&#x3B3;&#x3B9;&#x3B3;&#x3BD;&#x3CE;&#x3C3;&#x3Ba;&#x3B5;&#x3B9;
</maxim>
```

I documenti XML possono avere una **dichiarazione XML** . Se presente, la dichiarazione XML deve essere la prima cosa nel documento (nemmeno uno spazio bianco può precederlo). Inizia con`<? xml`, finisce con `?>`e deve avere lo pseudo-attributo `versione` e possibilmente `codifica` e `indipendente, autonomo`. Il significato degli pseudo-attributi è il seguente:

- la **versione** pseudo-attributo contiene la versione di XML in cui il documento è scritto;
- lo pseudo-attributo **codifica** specifica il set di caratteri usato nel documento. Se il server del file system fornisce meta-informazioni sulla codifica, questa codifica esterna ha la priorità sulla dichiarazione XML. Per impostazione predefinita, si presume che i documenti XML siano codificati nella codifica UTF-8 del set di caratteri **Unicode** . Un **set di caratteri** mappa particolari caratteri, come Z, a numeri particolari, come 90. Questi numeri sono chiamati punti codice. Una **codifica di caratteri**determina come quei punti di codice sono rappresentati in byte. Unicode è un set di caratteri standard internazionale che può essere utilizzato per scrivere documenti in quasi tutte le lingue che è probabile che parlino. UTF-8 è probabilmente la codifica dei caratteri supportata più ampiamente di Unicode. È una codifica a lunghezza variabile. Ad esempio, i caratteri da 0 a 127 sono codificati in 1 byte ciascuno (come in ASCII) e quelli da 128 a 2047 usano 2 byte ciascuno;
- lo pseudo-attributo **standalone** può avere i valori booleani`sì` o `no`. Se il valore è `sì`, quindi il documento è auto-contenente, il che significa che tutti i suoi valori sono presenti nel documento. Se il valore è`no`, quindi alcuni valori del documento sono specificati in un DTD esterno (ad esempio un valore predefinito per un attributo).

Ad esempio, la seguente riga dichiara un documento XML standalone nella versione 1.0 con una codifica ISO-8859-1 (ASCII più i caratteri per la maggior parte delle lingue dell'Europa occidentale)

```
<? xml version = "1.0" encoding = "ISO-8859-1" standalone = "yes"?>
```

Poiché i documenti XML sono scritti in Unicode, puoi usarli per scrivere documenti multilingue. In tali documenti, è utile identificare in quale lingua è scritta una particolare sezione. Ad esempio, un correttore ortografico multilingue potrebbe controllare l'ortografia delle sezioni in diverse lingue. Ogni elemento XML può avere un attributo **xml: lang** che specifica la lingua del contenuto dell'elemento. I codici di lingua sono definiti in ISO-639 . Un esempio segue:

```
<paragraph xml:lang="en">
The following is a Greek maxim saying: "The wise man knows himself".
</paragraph>

<maxim xml:lang="el">
&#x3C3;&#x3BF;&#x3C6;&#x3CC;&#x3C2;
&#x3AD;&#x3B1;&#x3C5;&#x3C4;&#x3CC;&#x3BD;
&#x3B3;&#x3B9;&#x3B3;&#x3BD;&#x3CE;&#x3C3;&#x3Ba;&#x3B5;&#x3B9;
</maxim>
```

Namespaces

Per ragioni di interoperabilità, le organizzazioni possono accettare di utilizzare solo determinati tag. Questi set di tag sono chiamati **applicazioni XML** . Gli esempi sono:

- XHTML : definisce una versione HTML compatibile di HTML.
- Scalable Vector Graphics (SVG): è un linguaggio per descrivere grafica bidimensionale in XML.
- Mathematical Markup Language (MathML): è un linguaggio per descrivere la notazione matematica.

I namespace hanno due scopi in XML:

1. Per distinguere tra elementi e attributi da diversi vocabolari con significati diversi e che capita di condividere lo stesso nome. Ad esempio, considera questo documento XML . Diversi elementi (titolo, autore, anno) sono stati sovraccaricati di significati diversi in diverse parti del documento. Questo non è semanticamente corretto. Inoltre, ciò potrebbe causare problemi di convalida, rendering, interrogazione e altro. Ad esempio, l'elemento`autore` ha diversi tipi in diverse parti del documento.
2. Raggruppare tutti gli elementi e gli attributi correlati da un'unica applicazione XML in modo che il software possa riconoscerli. Ad esempio, considera questo documento XML . Combina due applicazioni XML: XHTML e MathML. Devono essere resi in diversi modi. Come può un browser riconoscerli?

In entrambi i casi la soluzione è di utilizzare diversi spazi dei nomi per diverse applicazioni XML.

I namespace vengono implementati allegando un prefisso a ciascun elemento e attributo. Ogni prefisso è mappato a un URI di un`xmlns:prefisso`attributo. L'associazione ha ambito all'interno dell'elemento in cui il`xmlns`l'attributo è dichiarato e all'interno del suo contenuto. Gli elementi e gli attributi associati allo stesso URI appartengono allo stesso spazio dei nomi. È l'URI che conta, non il prefisso.

Ad esempio, ecco una soluzione per il problema di sovraccarico degli elementi sopra descritto. Si noti che nella soluzione proposta l'autore del CV non è più confuso con gli autori delle pubblicazioni. Lo stesso per titolo e anno.

Quando gli spazi dei nomi vengono utilizzati solo per identificare gli elementi e gli attributi di una particolare applicazione XML e non per distinguere elementi diversi con lo stesso nome, gli **spazi dei nomi predefiniti**sono migliori. Gli spazi dei nomi predefiniti possono essere utilizzati in due modi. Utilizzando un attributo xmlns senza prefisso e dichiarando un attributo xmlns fisso nella DTD del documento. Alcune applicazioni hanno gli URI dello spazio dei nomi standard. Per esempio:

- XHTML ha un URI dello spazio dei nomi standard `http://www.w3.org/1999/xhtml`;
- MathML ha un URI dello spazio dei nomi standard `http://www.w3.org/1998/Math/MathML`;
- SVG ha un URI dello spazio dei nomi standard `http://www.w3.org/2000/svg`.

Ad esempio, ecco una soluzione per il problema di riconoscimento dell'applicazione sopra descritto.

Un **parser XML** è un software che legge il documento XML e determina se è ben formato. Un documento **ben formato** aderisce alle regole grammaticali XML, in particolare obbedisce alle seguenti regole:

1. ogni tag di inizio deve avere un tag di chiusura corrispondente;
2. gli elementi non possono sovrapporsi;
3. ci deve essere esattamente un elemento del documento;
4. i valori degli attributi devono essere quotati;
5. i nomi degli attributi per lo stesso elemento devono essere diversi.

Vale la pena notare che un parser XML non può accettare un documento non valido e non è consentito provare a correggere il documento. E 'necessario segnalare gli errori.

Il modo più semplice per analizzare un documento è caricarlo in un browser Web che conosce XML. Il browser mostrerà il documento ogni volta che è ben formato o segnalerà gli errori in caso contrario. In alternativa, si può usare un parser XML standalone come lo strumento da riga di comando xmllint .

Lo spettro di applicazioni

XML è utile sia per gli esseri umani che per i computer. Gli scenari comuni in cui XML può essere utilizzato dalle persone includono:

- scrivere un documento usando DocBook . Docbook è un'applicazione XML progettata per il markup di testi principalmente su hardware e software.
- scrivere un documento in TEI . TEI è un'applicazione XML per la marcatura di testi leggibili meccanicamente principalmente nelle discipline umanistiche, delle scienze sociali e della linguistica.

I documenti risultanti sono chiamati documenti **incentrati sul testo** . Questi sono documenti XML solitamente scritti da umani per essere letti da umani o computer. Sono documenti XML semipermanenti con molto testo e una struttura scadente.

Gli scenari comuni in cui XML può essere utilizzato dai computer includono:

- **scambio di dati** . Le informazioni provengono da fonti diverse (relazioni, oggetti, documenti) e devono essere scambiate tra queste fonti. L'XML potrebbe fungere da comune datacle;
- **database semistrutturati** . I dati semistrutturati non hanno schema regolare. Spesso include informazioni duplicate e mancanti. Quindi, non si adatta naturalmente ai database relazionali. XML è stato proposto come modello di dati per i dati semistrutturati.

I documenti risultanti sono chiamati documenti **incentrati sui dati** . Si tratta di documenti XML generalmente generati da computer per la lettura da parte di altri computer. Sono documenti XML transitori con molti dati grezzi e una struttura ricca.

Storia XML

Ecco alcuni **punti** di **riferimento storici** nell'evoluzione dell'XML:

1960

L'Agenzia di ricerca avanzata (ARPA) del Dipartimento della Difesa degli Stati Uniti inizia un progetto per vedere se i computer ampiamente separati potrebbero essere collegati tra loro. La rete di computer risultante si chiama **ARPANET** . Il nome della rete si evolve gradualmente in **Internet** .

1970

Charles Goldfarb, Edward Mosher e Raymond Lorie di IBM inventano il **Generalized Markup Language** (GML).

1986

GML si sviluppa in **Standard GML** (SGML), uno standard ISO.

1989

Tim Berners-Lee propone l' **HyperText Markup Language** (HTML), l'applicazione SGML di maggior successo. La motivazione originale di HTML era quella di tenere traccia dei dati sperimentali presso l'Organizzazione europea per la ricerca nucleare (CERN).

1996

Jon Bosak, Tim Bray, CM Sperberg-McQueen, James Clark e molti altri iniziano a lavorare su una **versione lite di SGML**che conserva la maggior parte del suo potere durante il taglio di molte funzioni inutili.

1998

XML 1.0 diventa uno standard del World Wide Web Consortium (W3C), un consorzio internazionale guidato da Tim Berners-Lee che produce specifiche tecniche per le tecnologie web.

Il modello di dati XML è anche strettamente correlato al **modello di dati gerarchici** , che era piuttosto popolare durante gli anni '60 prima dell'avvento del modello di dati relazionali.

Esecuzione di esempio

La DBLP Computer Science Bibliography (DBLP) è la più grande bibliografia digitale per l'informatica. La bibliografia è disponibile in formato XML (questo è un grande documento XML compresso). Ecco alcuni brevi estratti XML della bibliografia:

- Tutti gli articoli pubblicati da Massimo Franceschet [xml ; piccolo 14 KB]
- Tutti gli articoli pubblicati sulla rivista Communications of the ACM [xml ; media 3,4 MB]
- Tutti gli articoli pubblicati nel 2000 [xml ; grande 24 MB]

L'obiettivo dell'esempio corrente è lo sviluppo di un pool di tecnologie XML che funzionano sulla bibliografia DBLP.

XML resources

- Home page at W3C
- W3C specifications
- The XML FAQ

Document Type Definition

Il markup permesso in una particolare applicazione XML può essere documentato in uno **schema** . Il linguaggio di schema più ampiamente supportato e l'unico definito dalla specifica XML 1.0 è il **Document Type Definition** (DTD). Un altro linguaggio di schema comune è XSchema .

Un DTD consente di posizionare alcuni vincoli sulla **struttura di** un documento XML. Elenca tutti gli elementi, gli attributi e le entità utilizzati dal documento e il contesto in cui vengono utilizzati. I DTD non dicono mai nulla sul tipo di contenuto di un elemento o sul valore di un attributo. Ad esempio, non puoi dire che il prezzo è un numero reale, o il nome è una stringa, o nato è una data.

Richiama il documento XML su Alan Turing. Ecco una DTD per questo documento:

```
<!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)>
```

La prima riga dichiara che l'elemento `persona` deve contenere esattamente un elemento figlio `nome` e zero o più elementi figlio `professione`, in questo ordine. La seconda riga dice che lo stesso elemento deve avere a`Nato` attributo e potrebbe avere a `morto`attributo (l'ordine non è rilevante). La terza riga dichiara che l'elemento`nome` deve avere due bambini chiamati `primo` e `scorso`in questo ordine. Le ultime tre righe specificano che gli elementi`primo`, `scorso`, e `professione` deve contenere dati di carattere analizzati (PCDATA), ovvero testo non elaborato che contiene eventualmente riferimenti di entità e carattere ma che non contiene alcun tag.

Dichiarazione del tipo di documento

Un documento XML è associato a un DTD con una **dichiarazione del tipo di documento** . Questa è una dichiarazione che dovrebbe seguire la dichiarazione XML nel documento XML. Ha la seguente forma:

```
<! DOCTYPE root [DTD]>
```

dove `DOCTYPE` è la parola chiave per la dichiarazione del tipo di documento, `radice` è il nome dell'elemento del documento del documento XML e `DTD` è l'insieme effettivo di regole che definisce la DTD.

Un documento XML può includere un riferimento alla DTD, invece della DTD stessa. Ciò è utile quando lo stesso DTD è condiviso da documenti diversi. La dichiarazione del tipo di documento cambia come segue:

```
<! DOCTYPE root KEYWORD "URI">
```

dove `PAROLA CHIAVE` può essere o `SISTEMA` o `PUBBLICO`. Se la parola chiave è`SISTEMA`quindi il DTD viene specificato al di fuori del documento tramite un URL (**Uniform Resource Locator**). L'URL può essere assoluto o relativo. Ad esempio, la seguente dichiarazione specifica che l'elemento del documento è`persona` e che il DTD è contenuto in un file locale chiamato `Turing.dtd` che è memorizzato nella stessa cartella del documento XML:

```
<! DOCTYPE person SYSTEM "Turing.dtd">
```

Se la parola chiave è `PUBBLICO` quindi il DTD viene specificato al di fuori del documento tramite un URN (**Uniform Resource Name**). L'URN è un nome che identifica in modo univoco l'applicazione XML. Ad esempio, la seguente dichiarazione specifica che l'elemento del documento è`html` e che il DTD è la versione rigorosa di XHTML 1.0, di cui è l'URN – `// W3C // DTD XHTML 1.0 Strict // IT`. Si noti che viene aggiunto anche un URL di backup nel caso in cui il DTD dichiarato non sia disponibile sul file system locale:

```
<! DOCTYPE html PUBLIC "- // W3C // DTD XHTML 1.0 Strict // IT"
"Http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Infine, puoi combinare sottoinsiemi DTD interni ed esterni usando la seguente sintassi:

```
<! DOCTYPE root KEYWORD "URI" [DTD]>
```

Validità

Un documento XML è ritenuto **valido** se aderisce alle definizioni della DTD associata. Come regola generale, i browser Web non convalidano i documenti ma li controllano solo per la loro idoneità. Se si sta sviluppando un'applicazione, è possibile utilizzare l'API del parser per convalidare il documento. Se stai scrivendo documenti a portata di mano, puoi utilizzare un validatore online o scaricare ed eseguire un programma locale.

Un esempio di validatore online è il modulo di convalida XML del Brown University Scholarly Technology Group. Un parser e validatore XML da riga di comando è xmllint , che fa parte della libreria XML libxmlsviluppata per il progetto Gnome (ma utilizzabile al di fuori della piattaforma Gnome). Ad esempio, puoi convalidare`Turing.xml` contro `Turing.dtd` con la seguente sintassi:

```
xmllint --dtdvalid Turing.dtd Turing.xml
```

Se `Turing.xml` contiene la dichiarazione del tipo di documento, è possibile utilizzare la seguente sintassi:

```
xmllint - Turing.xml valido
```

Usa l'opzione `--noout` per evitare l'output del documento XML.

Si può anche convalidare un documento con una DTD usando la funzione BaseX validate:dtd come nell'esempio seguente:

```
let $ doc: = doc ('bancario.xml')
let $ schema: = 'banking.dtd'
return validate: dtd ($ doc, $ schema)
```

Definizioni degli elementi

Ogni elemento utilizzato in un documento valido deve essere dichiarato nella DTD con una **definizione di elemento** come segue

```
<! ELEMENT nome contenuto>
```

dove `nome` è il nome dell'elemento e `contenuto` specifica quali sono i bambini che l'elemento può o deve avere in quale ordine. Il contenuto della definizione dell'elemento può essere:

Dati dei caratteri analizzati

Questa è la specifica di contenuto più semplice che dice che un elemento può contenere solo testo inclusi riferimenti a entità e caratteri, ma non può contenere elementi secondari:

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

Elemento figlio

Questo dice che un elemento può contenere solo un elemento figlio di un dato tipo:

```
<! ELEMENT contatto (e-mail)>
```

Scelta

Questo dice che un elemento può contenere un tipo di elemeto figlio o un altro, ma non entrambi:

```
<! ELEMENT contatto (e-mail | telefono)>
```

Sequenza

Questo dice che un elemento può contenere più elementi figlio nell'ordine specificato:

```
<! ELEMENT nome (primo, ultimo)>
```

Contenuto vuoto

Questo dice che un elemento non deve avere alcun contenuto (ma potrebbe avere attributi):

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

Qualsiasi contenuto

Questo dice che un elemento può avere qualsiasi contenuto (tuttavia i suoi figli, se ce ne sono, devono essere definiti):

```
<! ELEMENT immagine ANY>
```

Iterazione

Esistono tre suffissi che possono essere apposti dopo nomi, sequenze e scelte per specificare quanti elementi figlio sono previsti. Questi sono:

* Sono consentite zero o più istanze
+ Sono consentite una o più istanze
? Zero o una istanza sono consentiti

Ad esempio, la seguente definizione lo dice `nome` deve avere zero o più `primo` bambini, eventualmente seguiti da a `mezzo`bambino, seguito da uno o più `scorso` bambini:

```
<! ELEMENT nome (primo *, medio?, Ultimo +)>
```

Data questa definizione, tutto quanto segue `nome` gli elementi sono validi:

```
<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>
```

La seguente definizione dice questo `nome` può avere qualsiasi numero di `primo`, `mezzo`, e `scorso`bambini *in qualsiasi ordine* :

```
<! ELEMENT nome (first | middle | last) *>
```

Di seguito è riportato un esempio notevole di **contenuto misto** , ovvero testo interlacciato con markup, che è il tipico contenuto dei documenti narrativi. Lo specifica che `name` può avere qualsiasi numero di childern: `primo`, `mezzo`, e `scorso` in qualsiasi ordine possibilmente interlacciati con dati di carattere analizzati:

```
<! ELEMENT nome (#PCDATA | first | middle | last) *>
```

Data questa definizione, la seguente `nome` elemento è valido:

```
<name>
   First comes the first name: <first>Samuel</first>
   Then the middle one: <middle>Lee</middle>
   Last comes the last name: <last>Jackson</last>
   Not very surprising indeed!
</name>
```

Vale la pena notare che questo è l'unico modo per indicare il contenuto misto: si può solo dire che un elemento contiene un numero qualsiasi di elementi da una lista in qualsiasi ordine, così come dati di carattere analizzati. Inoltre, la parola

chiave#PCDATA deve essere il primo della lista. Infine, considera la seguente definizione alternativa pernome:

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

Questa definizione apparentemente innocua è in realtà invalida. L'errore che si ottiene se si tenta di convalidare un documento con questa definizione è simile al seguente: *Content model of name is not determinist*

In effetti, l'errore è nella DTD, non nell'XML. Il modello di contenuto generato da qualsiasi DTD deve essere **deterministico** . In questo caso non lo è, da quando il validatore legge a primo elemento nell'XML ci sono *due* possibili strade da percorrere: una è la definizione per nome è finito (il primo disgiunto nella definizione), il secondo è che a last l'elemento è previsto (l'ultimo disgiunto nella definizione). Quindi, leggendo lo stesso simbolo, il validatore può raggiungere due stati diversi. Questo è un tipico comportamento non deterministico. La logica alla base di questa limitazione è che i processori DTD dovrebbero essere più facili da implementare.

Definizioni di attributi

Oltre a definire i suoi elementi, un documento valido deve definire tutti gli attributi degli elementi. Questo viene fatto con la seguente dichiarazione:

```
<! ATTLIST element attribute TYPE DEFAULT>
```

dove ATTLIST è la parola chiave per la definizione dell'elenco di attributi, elemento è il nome dell'elemento che contiene gli attributi, attributo è il nome dell'attributo da definire, GENERE è il tipo dell'attributo e PREDEFINITOè un valore predefinito per l'attributo. Puoi definire molti attributi per lo stesso elemento.

Questi sono i **tipi di attributi** più rilevanti :

CDATA

Il valore di un attributo CDATA è qualsiasi stringa di testo. Ad esempio, il seguente definisce il CDATANato e morto attributi per a persona elemento:

```
<!ATTLIST person born CDATA #REQUIRED
                died CDATA #IMPLIED>
```

Enumerazione

Un'enumerazione è un elenco di valori possibili per un attributo, separati da una barra verticale (|). Ad esempio, il seguente definisce il sesso attributo per a elemento persona. I valori dell'attributo sesso sono male o female:

```
<!ATTLIST person sex (male | female) #REQUIRED>
```

ID

Il valore di un attributo ID è un nome XML (in particolare non può iniziare con un numero) in modo tale che nessun altro attributo ID (con qualsiasi nome, di alcun elemento) nello stesso documento abbia lo stesso valore. Gli attributi ID assegnano identificatori univoci agli elementi e corrispondono a **chiavi** in database relazionali.

IDREF

Il valore di un attributo IDREF è un nome XML che corrisponde al valore di alcuni attributi ID nel documento. Gli attributi IDREF sono riferimenti a elementi identificati da identificatori univoci. Gli attributi IDREF corrispondono a **chiavi esterne** in database relazionali.

IDREFS

Il valore di un attributo IDREFS è un elenco separato da spazi bianchi di valori IDREF.

Oltre a fornire un tipo di dati, ciascuna definizione di attributo include una **definizione predefinita** per l'attributo. Ci sono quattro possibilità per questo default:

 # REQUIRED (NECESSARIO)

L'attributo è obbligatorio e deve essere fornito.

#IMPLIED

L'attributo è facoltativo e può o non può essere fornito.

"valore"

L'attributo ha il valore specificato predefinito. L'attributo può o non può essere fornito. Se è incluso, potrebbe avere un valore legale. Se non è incluso, il validatore fornirà uno con il valore specificato. Ad esempio, il seguente definisce il`sposato` attributo per a `persona` elemento i cui valori possono essere entrambi `sì` o `no` e, per impostazione predefinita, il valore `no` è fornito:

```
<!ATTLIST person married (yes | no) "no">
```

#FIXED "value" (Valore fisso")

L'attributo ha il valore specificato costante e immutabile. L'attributo può o non può essere fornito. Se è incluso, deve avere il valore specificato. Se non è incluso, il validatore fornirà uno con il valore specificato. Questo valore predefinito viene in genere utilizzato per definire gli **spazi dei nomi predefiniti** , come nell'esempio seguente:

```
<! ATTLIST html xmlns CDATA #FIXED "http://www.w3.org/1999/xhtml">
```

In questo caso, non è necessario fornire l'attributo dello spazio dei nomi predefinito nel documento XML. Il validatore lo farà per te.

Si noti che il processo di convalida genera un documento XML che potrebbe essere diverso da quello di input. Puoi usare xmllint con l'opzione`--dtdattr` per popolare il documento XML con attributi ereditati. Ricordiamo che, come regola generale, i browser Web non convalidano i documenti. Tuttavia, possono leggere la DTD (interna) e possono fornire attributi che sono stati definiti nella DTD.

Attributi ID e IDREF

Gli attributi ID e IDREF sono le controparti XML di chiavi e chiavi esterne nei database relazionali. Di seguito mostreremo alcuni casi d'uso tipici per questi attributi.

Immagina uno scenario bancario in cui ci sono clienti e account. I clienti possono avere zero o più account e account sono associati a un cliente unico. Ecco una soluzione valida che utilizza gli attributi ID ma evita quelli IDREF:

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

<! DOCTYPE banking [
  <! ELEMENT banking (cliente *)>
  <! ELEMENT cliente (nome, account *)>
  <! ID ID cliente ATTLIST #REQUIRED>
  <! ELEMENT conto (banca, numero)>
  <! ID ID account ATTLIST #REQUIRED>
  <! Nome ELEMENT (#PCDATA)>
  <! ELEMENT bank (#PCDATA)>
  <! ELEMENT numero (#PCDATA)>
]>

<Banking>
   <customer id = "C1">
   <nome> Massimo Franceschet </ name>
   <account id = "A1">
      <Banca> Fineco </ bank>
      <Numero> 34567 </ numero>
   </ Account>
   <ID account = "A2">
      <bank> ABN AMRO </ bank>
      <Numero> 98672 </ numero>
   </ Account>
   </ Cliente>

   <customer id = "C2">
   <name> Enrico Zimuel </ name>
   <ID account = "A3">
      <bank> ING Bank </ bank>
      <Numero> 8909 </ numero>
   </ Account>
   </ Cliente>
</ Banking>
```

Ogni cliente e ogni account ha un `id` attributo di tipo ID. Poiché non esiste un account di proprietà di più di un cliente, possiamo elencare gli account del cliente come figli di`cliente` elemento senza duplicare alcun valore ID.

Supponiamo ora che uno scenario diverso in cui i clienti possano avere zero o più account e account possa essere di proprietà di uno o più clienti. La seguente è una soluzione non valida :

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

<! DOCTYPE banking [
  <! ELEMENT banking (cliente )>
  <! ELEMENT cliente (nome, account *)>
  <! ID ID cliente ATTLIST #REQUIRED>
  <! ELEMENT conto (banca, numero)>
  <! ID ID account ATTLIST #REQUIRED>
  <! Nome ELEMENT (#PCDATA)>
  <! ELEMENT bank (#PCDATA)>
  <! ELEMENT numero (#PCDATA)>
]>

<Banking>
   <customer id = "C1">
      <nome> Massimo Franceschet </ name>
      <account id = "A1">
         <Banca> Fineco </ bank>
         <Numero> 34567 </ numero>
      </ Account>
      <ID account = "A2">
         <bank> ABN AMRO </ bank>
         <Numero> 98672 </ numero>
      </ Account>
   </ Cliente>

   <customer id = "C2">
      <name> Enrico Zimuel </ name>
      <ID account = "A2">
         <bank> ABN AMRO </ bank>
         <Numero> 98672 </ numero>
      </ Account>
   </ Cliente>
</ Banking>
```

La soluzione non è valida poiché l'account identificato da `A2`è di proprietà di entrambi i clienti e quindi il suo valore ID è duplicato. Una soluzione valida consiste nell'elencare separatamente clienti e account e rappresentare le loro relazioni utilizzando gli attributi IDREF (S) (si noti che non è necessario denominare gli attributi di tipo ID`id` e gli attributi di tipo IDREF (S) non hanno bisogno di essere nominati `IDREF (s)`):

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

<! DOCTYPE banking [
  <! ELEMENT banking (cliente | account) *>
  <! ELEMENT cliente (nome, account?)>
  <! ID ID cliente ATTLIST #REQUIRED>
  <! Nome ELEMENT (#PCDATA)>
  <! ELEMENT account VUOTO>
  <! ATTLIST account idrefs IDREFS #REQUIRED>

  <! ELEMENT conto (banca, numero, proprietari)>
  <! ID ID account ATTLIST #REQUIRED>
  <! ELEMENT bank (#PCDATA)>
```

```
  <! ELEMENT numero (#PCDATA)>
  <! ELEMENT owner VUOTO>
  <! ATTLIST owner idrefs IDREFS #REQUIRED>
]>

<Banking>
   <customer id = "C1">
      <nome> Massimo Franceschet </ name>
      <account idrefs = "A1 A2" />
   </ Cliente>

   <customer id = "C2">
      <name> Enrico Zimuel </ name>
      <account idrefs = "A2" />
   </ Cliente>

   <account id = "A1">
      <Banca> Fineco </ bank>
      <Numero> 34567 </ numero>
      <owner idrefs = "C1" />
   </ Account>

   <ID account = "A2">
      <bank> ABN AMRO </ bank>
      <Numero> 98672 </ numero>
      <owner idrefs = "C1 C2" />
   </ Account>
</ Banking>
```

In generale, le **relazioni n: m** devono essere codificate usando gli attributi ID e IDREFS, mentre le **relazioni 1: n** possono evitarle. Tuttavia, nelle relazioni n: m non è necessario rappresentare sia le relazioni dirette che quelle inverse (`conti` e `proprietari`nel nostro esempio). Uno di questi è sufficiente. Tuttavia, rappresentare entrambe le relazioni potrebbe essere utile per semplificare le query. Ad esempio, supponiamo che rappresentiamo solo la relazione `accounts` nel nostro esempio bancario. In tal caso, recuperare gli account di proprietà di un determinato cliente è un compito facile. Tuttavia, il contrario (selezionando i proprietari di un account) è più complicato. Vale il viceversa se rappresentiamo solo la relazione `owners`.

Vale la pena notare che non è valido utilizzare lo stesso valore per due attributi ID. Inoltre, non è valido utilizzare un valore per un attributo IDREF (S) che non è un valore ID nel documento. Tuttavia, tutto il resto è valido, anche se è **logicamente non valido** . In particolare, il validatore non si cura di verificare se l' `IDREFS` valori di `proprietari` gli elementi sono in effetti identificativi del cliente o se il `IDREFS` valori di `conti` gli elementi sono in effetti identificatori di account. Inoltre, il validatore non controlla il vincolo della relazione inversa. Questo vincolo specifica che, ad esempio, se il cliente `C1` possiede un account `A1`, quindi account `A1` deve essere di proprietà del cliente `C1`.

Definizioni di entità

L' **entità** è una scorciatoia per un pezzo di dati. I dati non sono necessariamente in formato XML. Possiamo avere diversi tipi di entità: interne o esterne, predefinite o definite dall'utente. Abbiamo già discusso le entità predefinite nel capitolo XML. Qui, introduciamo entità definite dall'utente. **Le entità interne definite dall'utente** sono definite nella DTD come segue:

```
<! ENTITY name "text">
```

dove `ENTITÀ` è la parola chiave definition, `nome` è il nome dell'entità e `testo`è il testo di sostituzione dell'entità. Il valore del testo è una stringa che probabilmente contiene un markup ben formato. Qui ci sono un paio di esempi:

```
<! ENTITY XML "Extensible Markpup Language">
<! ENTITY footer "<author> Massimo Franceschet </ author>
                  <data> 16 febbraio 2005 </ data> ">
```

Le entità definite dall'utente esterne sono definite nella DTD come segue:

```
<! ENTITY name SYSTEM "URI">
```

dove `URI` è un documento contenente il testo di sostituzione dell'entità dell'entità `nome`. Per esempio:

```
<! ENTITY footer SYSTEM "footer.xml">
```

Il testo di sostituzione dell'entità deve essere ben formato, ma non deve essere racchiuso in un elemento radice univoco. Quando è esterno, il testo di sostituzione dell'entità può avere una dichiarazione di testo. Questo è simile a una dichiarazione XML, ma l' l'attributo `versione` è facoltativo e il `encoding` uno è richiesto. Infine, il testo sostitutivo può contenere altri riferimenti di entità, ma riferimenti auto-referenziali e circolari sono vietati. Le entità definite dall'utente sono utilizzate come predefinite. Ad esempio, per invocare l'entità chiamata`XML` devi scrivere `& XML;` nel documento XML. Il parser sostituirà la chiamata dell'entità con il suo valore di testo.

Un uso delle entità è di evitare di digitare lo stesso testo nel documento XML molte volte. Ecco un esempio :

```
<? xml version = "1.0" encoding = "ISO-8859-1" standalone = "no"?>

<!DOCTYPE slides [
   <!ELEMENT slides     (slide*)>
   <!ELEMENT slide     (course,university,image,title,content,author,date,about)>
   <!ELEMENT title      (#PCDATA)>
   <!ELEMENT content    (#PCDATA)>
   <!ELEMENT author     (#PCDATA)>
   <!ELEMENT date       (#PCDATA)>
   <!ELEMENT about      (#PCDATA)>
   <!ELEMENT course     (#PCDATA)>
   <!ELEMENT university (#PCDATA)>
   <!ELEMENT image      EMPTY>
   <!ATTLIST image source CDATA #REQUIRED>
```

```
   <!ENTITY  XML "Extensible Markpup Language">
   <!ENTITY  UdA "Università &quot;G. D'Annunzio&quot;">
   <!ENTITY  footer "<author>Massimo Franceschet</author>
                     <date>16 February 2005</date>
                     <about>&XML;</about>">
   <!ENTITY  prolog SYSTEM "prolog.txt">
]>

<slides>
   <slide>
      &prolog;
      <title>&XML; fundamentals</title>
      <content>The fundamentals about &XML;</content>
      &footer;
   </slide>
   <slide>
      &prolog;
      <title>&XML; schema languages</title>
      <content>The schema languages for the &XML;</content>
      &footer;
   </slide>
</slides>
```

dove prolog.txt contiene il seguente testo:

```
<?xml encoding="ISO-8859-1"?>
<course>Semistructured data: representation and query languages.</course>
<university>&UdA;</university>
<image source="logo.jpg"/>
```

Il documento XML analizzato dovrebbe assomigliare a questo:

```
<? Xml version = "1.0" encoding = "ISO-8859-1" standalone = "no"?>
<slides>
   <Slitta>
      <course> Dati semistrutturati: linguaggio di rappresentazione e di query.
</ course>
      <university>Università "G. D'Annunzio"</university>
      <image source = "logo.jpg" />
      <title> Fondamenti di Extensible Markpup Language </ title>
      <content> I fondamenti di Extensible Markpup Language </ content>
      <autore> Massimo Franceschet </ author>
      <data> 16 febbraio 2005 </ data>
      <about> Extensible Markpup Language </ about>
   </ Slide>
   <Slitta>
      <course> Dati semistrutturati: linguaggio di rappresentazione e di query.
</ course>
      <university>Università "G. D'Annunzio"</university>
      <image source = "logo.jpg" />
      <title> Lingue dello schema di Extensible Markpup Language </ title>
      <contenuto> Le lingue dello schema per l'Extensible Markpup Language </
content>
      <autore> Massimo Franceschet </ author>
      <data> 16 febbraio 2005 </ data>
      <about> Extensible Markpup Language </ about>
   </ Slide>
</ Slides>
```

Infine, **un'entità parametro** è una scorciatoia per un pezzo di DTD (non per un pezzo di dati XML). Come entità generali, possono essere interne ed esterne. Ecco un esempio:

```
<! ENTITY% body "<! ELEMENT title (#PCDATA)>
                 <! ELEMENT content (#PCDATA)> ">
<! ENTITY% prolog SYSTEM "prolog.dtd">
```

Per richiamare l'entità parametro chiamata `corpo` devi scrivere `%corpo;` nella DTD. Una caratteristica interessante è che le entità parametro possono essere definite nel sottoinsieme DTD esterno e ridefinite nel frammento DTD interno. In questo caso, la definizione interna conta.

Le entità dei parametri sono spesso utilizzate per la **modularizzazione** dei DTD. Quando un DTD è grande, l'insieme di dichiarazioni è tipicamente separato in moduli diversi e solo i moduli utili possono essere inclusi in un altro DTD. Ad esempio, le seguenti definizioni includono nel DTD corrente le definizioni contenute nel`body.dtd` documento:

```
<! ENTITY% body SYSTEM "body.dtd">
%corpo;
```

La modularizzazione spesso sfrutta il `IGNORARE` e `INCLUDERE`Direttive XML. Il primo è utilizzato per commentare una sezione di dichiarazioni:

```
<! [IGNORE [
  <! ELEMENT note (#PCDATA)>
]]>
```

L'effetto è di ignorare la definizione dell'elemento nella DTD corrente. La seconda direttiva è utile per includere una sezione di dichiarazioni:

```
<! [INCLUDE [
  <! ELEMENT note (#PCDATA)>
]]>
```

L'effetto è di usare la definizione dell'elemento è la DTD corrente. Le entità parametriche e le direttive precedenti possono essere utilizzate per implementare un'inclusione condizionale delle dichiarazioni. Supponiamo di definire la seguente entità parametro:

```
<! ENTITY% switch_note "INCLUDE">
```

Ora possiamo sostituire la parola chiave `INCLUDERE` con l'entità parametro definita:

```
<! [% Switch_note; [
  <! ELEMENT note (#PCDATA)>
]]>
```

L'entità parametro `switch_note` può quindi essere ridefinito in frammenti DTD interni, consentendo di accendere e spegnere il `Nota` definizione dell'elemento.

Come ultima osservazione, si noti che Mozilla non carica entità esterne (e DTD). Puoi usare xmllint con l'opzione`--noent`analizzare un documento XML con qualsiasi tipo di entità.

Database XML nativi

Molte informazioni in giro in questi giorni sono **semistrutturate** , **gerarchiche** e **ibride** .

Dati semistrutturatiha una struttura allentata (schema): un nucleo di attributi è condiviso da tutti gli oggetti associati a uno schema semistrutturato, ma sono possibili molte varianti individuali. Ad esempio, considera una bibliografia contenente riferimenti a pubblicazioni accademiche. Tutti i riferimenti hanno in comune un piccolo nucleo di attributi, come autori, titolo e anno di pubblicazione. Diversi tipi di riferimento, tuttavia, hanno molti attributi specifici. Ad esempio, i libri hanno editori, i giornali hanno numeri di volume, i contributi alle conferenze hanno titoli degli atti e degli indirizzi di conferenze, e le tesi hanno istituti ospitanti. Inoltre, alcuni di questi attributi potrebbero essere strutturati in modi diversi. Ad esempio, potremmo specificare il nome di un autore come un token univoco o come una lista arbitrariamente lunga di token, incluso lo spazio per possibilmente multipli prima, mezzo,

I dati gerarchici sono composti da elementi atomici ed elementi composti. Gli elementi atomici hanno un semplice contenuto piatto. Al contrario, gli elementi composti contengono sotto-elementi nidificati, sia atomici che composti. Non c'è limite al livello di nidificazione delle informazioni. La struttura risultante è una gerarchia di informazioni, probabilmente rappresentabile come un albero di oggetti. Ad esempio, una pagina Web scritta in XHTML ha una struttura gerarchica in cui gli elementi di tag potrebbero contenere altri elementi di tag. Come altro esempio, si consideri la gerarchia tassonomica biologica in cui una specie, il grado più elementare, annida all'interno di un genere, che a sua volta annida all'interno di una famiglia, e così via.

Il termine **dati ibridi si** riferisce al fatto che le informazioni spesso mescolano sia i dati che il testo. Ad esempio, un riferimento bibliografico potrebbe contenere record, come autori, titolo, anno, nonché informazioni narrative, come l'abstract di carta. Allo stesso modo, gli oggetti che rappresentano le persone in un social network mescolano attributi, come nome e professione, con descrizioni possibilmente lunghe e strutturate, come un CV.

Molti ricercatori hanno proposto XML come il formalismo più appropriato per lavorare con questo tipo di informazioni. XML ha i seguenti punti di forza unici come formato di dati:

1. **Sintassi semplice** . XML è un formato ben definito i cui documenti sono facili da creare, manipolare, analizzare tramite software e leggere da umani dotati di un

editor di testo di base. Inoltre, XML è portabile su diverse architetture di computer e linguaggi di programmazione;
2. **Dati semistrutturati** . La flessibilità del modello di dati XML consente di rappresentare informazioni non strutturate e dati con uno schema libero;
3. **Supporto per la nidificazione** . La natura gerarchica di XML rende possibile rappresentare strutture complesse in modo naturale;
4. **Supporto per informazioni ibride** . Sia le informazioni incentrate sui dati che quelle incentrate sul testo possono essere facilmente rappresentate all'interno dello stesso documento XML.

Un **database XML** è un sistema di persistenza dei dati che consente di gestire i dati in formato XML. Deve garantire quelle caratteristiche di un sistema di database tradizionale che sono vitali per le applicazioni del mondo reale, tra cui un linguaggio di query universale, efficiente e scalabile, vincoli di integrità dei dati, gestione delle transazioni, privacy dei dati, backup e ripristino. Esistono due principali classi di database XML:

1. **Database abilitati per XML** . Questi sistemi associano i dati XML a un database tradizionale, in genere un database relazionale. I dati XML possono essere archiviati in diversi modi: un documento XML può essere distrutto in campi relazionali, record e tabelle, può essere interamente memorizzato come dati carattere nei record, oppure può essere salvato come oggetti di carattere esterno di grandi dimensioni.
2. **Database XML nativi** . Un database XML nativo definisce un modello logico per un documento XML e memorizza e recupera i documenti in base a tale metodo. Esempi di tali modelli sono il modello di dati XPath, l'Infoset XML e i modelli sottostanti DOM e SAX.

Un esempio notevole di database XML, che implementa la maggior parte delle funzionalità menzionate, è BaseX .

A differenza di XML, il modello relazionale consente di memorizzare le informazioni in modo strutturato e piatto. Ne consegue che anche le informazioni semplici e altamente correlate, come una fattura con data e numero accompagnate da un elenco di voci della fattura (ognuna con descrizione, quantità e attributi di prezzo), devono essere suddivise su più tabelle e query potenzialmente costose, unendo le informazioni sparse attraverso i meccanismi chiave e chiave esterna, devono essere utilizzate per ricostruire le informazioni al momento della query.

Quando è un database XML una soluzione migliore per un database relazionale? Non c'è una risposta chiara. Tuttavia, Ronald Bourret dà la seguente regola euristica:

> *Se ti trovi essenzialmente a costruire un database relazionale all'interno del tuo database XML nativo, potresti chiederti perché non stai utilizzando un database relazionale in primo luogo.*

La progettazione di un database segue una metodologia consolidata comprendente la modellazione concettuale, logica e fisica dei dati. Di seguito diamo un contributo allo sviluppo di metodologie di progettazione e strumenti per database XML nativi. Adottiamo il **modello di Entity Relationship** ben compreso, esteso con specializzazione (ER in breve), come modello concettuale per i database XML nativi. Inoltre, scegliamo il W3C XML Schema Language (XML Schema, in seguito), come linguaggio dello schema per XML. Proponiamo una mappatura da ER a XML Schema con le seguenti proprietà: le informazioni e i vincoli di integrità del modello ER vengono preservati, non viene introdotta ridondanza, sono consentite diverse visualizzazioni gerarchiche delle informazioni concettuali, la struttura risultante è altamente connessa e il design è reversibile.

XML Path Language

XML path language (**XPath**) è un linguaggio semplice per recuperare elementi XML da un *singolo*documento XML. XPath può essere sfruttato in diverse tecnologie XML: da solo come semplice linguaggio di query per XML, in XQuery per recuperare elementi XML che possono essere ulteriormente elaborati per risolvere una query, in XSLT per recuperare gli elementi a cui vengono applicate le regole del modello per trasformare un documento XML, in W3C XML Schema per individuare chiavi e riferimenti chiave e infine in XPointer per puntare a particolari elementi XML nel documento XML collegato.

Per comprendere la semantica XPath è necessario comprendere il modello di dati XPath, ovvero, in che modo XPath visualizza un documento XML. Il **modello di dati XPath** rappresenta ogni documento XML come un albero di nodi. Ogni nodo ha uno di questi tipi:

Radice
: Il nodo radice è un *nodo virtuale* che non corrisponde a nessun componente nel documento XML. Ha un commento figlio per ogni commento all'esterno dell'elemento del documento, un istruzione figlio di elaborazione per ogni istruzione di elaborazione all'esterno dell'elemento del documento e un elemento figlio univoco che corrisponde all'elemento del documento. Il nodo radice non ha genitore. Il suo **valore di stringa** è il valore stringa del nodo dell'elemento del documento.

Elemento
: Un nodo elemento corrisponde a un elemento XML ed è etichettato con il tag dell'elemento XML. Ha sempre un genitore (il nodo radice o un nodo elemento) e può avere figli di tipo elemento, commento, istruzione di elaborazione e testo. Gli attributi e gli spazi dei nomi associati all'elemento non sono figli del nodo dell'elemento. Il valore stringa di un nodo elemento è il testo contenuto tra i tag di inizio e di fine dell'elemento, esclusi tutti i tag, i commenti e le istruzioni di elaborazione.

Attributo

Un nodo attributo corrisponde a un attributo. Il suo genitore è il nodo dell'elemento che contiene l'attributo (tuttavia l'attributo non è figlio del suo genitore!). Il valore stringa di un nodo attributo è il valore dell'attributo.

Spazio dei nomi

Un nodo dello spazio dei nomi rappresenta uno spazio dei nomi. Come i nodi degli attributi, i nodi dello spazio dei nomi hanno un genitore ma non sono figli del genitore. Il valore di stringa di un nodo dello spazio dei nomi è l'URI dello spazio dei nomi.

Testo

Un nodo di testo rappresenta una stringa di testo massima tra tag, commenti e istruzioni di elaborazione. Ha un nodo genitore ma nessun figlio. Il suo valore di stringa è il testo del nodo.

Istruzioni di lavorazione

Un nodo di istruzioni di elaborazione rappresenta un'istruzione di elaborazione. Ha un genitore e nessun figlio. Il valore di stringa di un nodo di istruzioni di elaborazione è il contenuto dell'istruzione che esclude la destinazione.

Commento

Un nodo di commento rappresenta un commento. Ha un genitore e nessun figlio. Il valore di stringa di un nodo di commento è il contenuto del commento.

In particolare, il modello di dati XPath considera le dichiarazioni XML e DTD. Inoltre, tutti i riferimenti di entità e carattere e le sezioni CDATA vengono risolti prima che venga creato l'albero XML. Ad esempio, prendere in considerazione il seguente documento XML :

```
<?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. -->
```

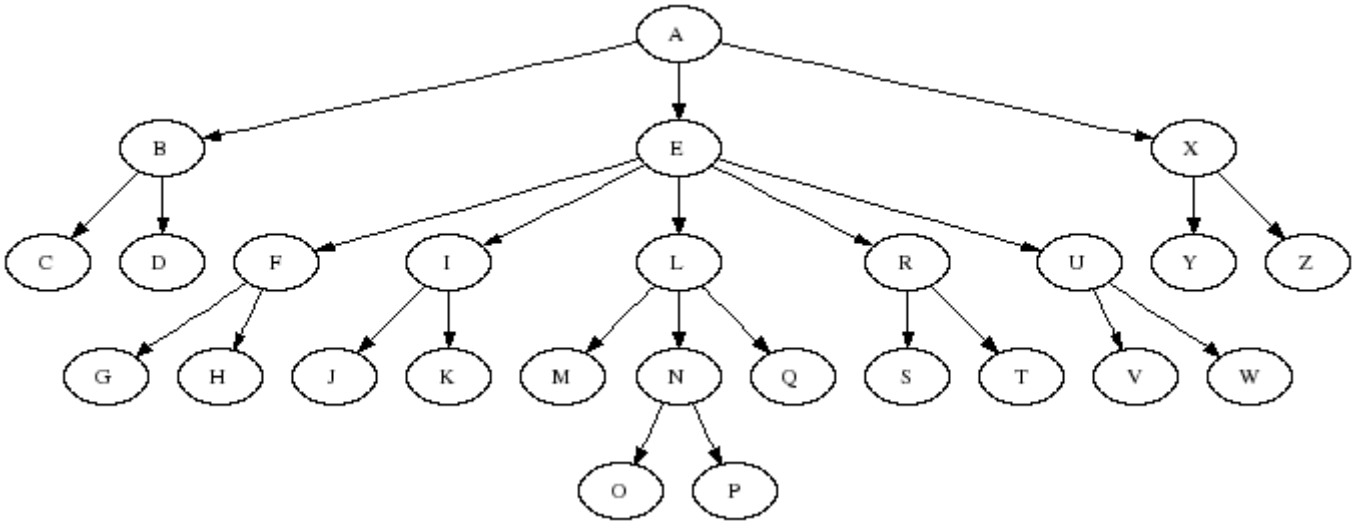
Per semplicità, in questo esempio, ignoriamo i nodi di testo i cui valori sono stringhe di spazi. Il nodo radice per questo documento ha due commenti figli corrispondenti ai due commenti del documento, un figlio di istruzioni di elaborazione per l'istruzione di elaborazione del foglio di stile e un elemento figlio per il`persona`elemento. Non è un bambino associato alle dichiarazioni XML e DTD. Il`persona` l'elemento ha come genitore il nodo radice e ha 5 figli in questo ordine: il `nome` nodo elemento, il primo nodo

elementoprofessione , il nodo di testo con valore tutto il testo tra il primo e il secondo elemento professione e gli ultimi due elemento figli professione. I nodi dell'attributoNato e mortonon sono figli del nodo persona. Tuttavia, il nodo persona è il genitore di loro. Ilnome l'elemento ha come genitore il persona elemento nodo e da bambini il primoe scorsonodi elemento. Il suo valore di stringa èAlnturidag. Il genitore diprimo elemento è il nomenodo dell'elemento. Ha un testo figlio con valorezona.

Passi di posizione XPath

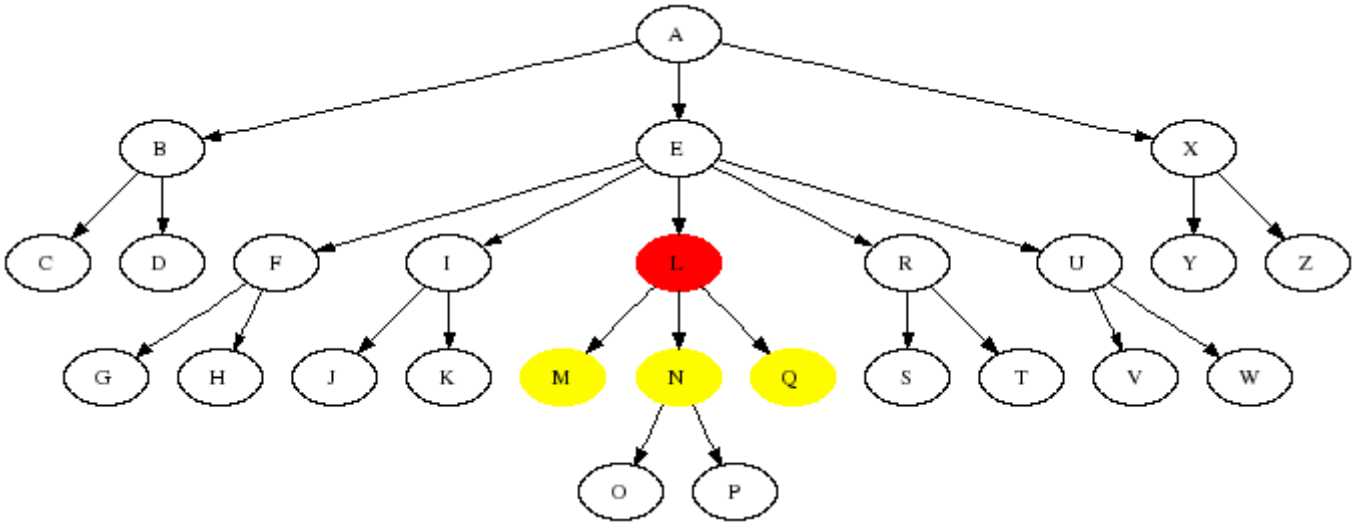
Ogni espressione XPath (o query XPath) viene valutata in un nodo ad albero XML (chiamato nodo di contesto) e restituisce un oggetto di uno dei quattro tipi: Booleano, numero, stringa, insieme di nodi (non a caso, un insieme di nodi). La più importante espressione XPath viene chiamata (posizione) **percorso** , il cui valore è sempre un insieme di nodi. Un percorso è composto da passaggi (posizione). Ogni **passaggio** ha la forma axis::test[filter]. La parte [filtro] è facoltativo.

Impareremo XPath con l'esempio sfruttando il seguente albero dell'alfabeto che corrisponde al documento XML dell'alfabeto :

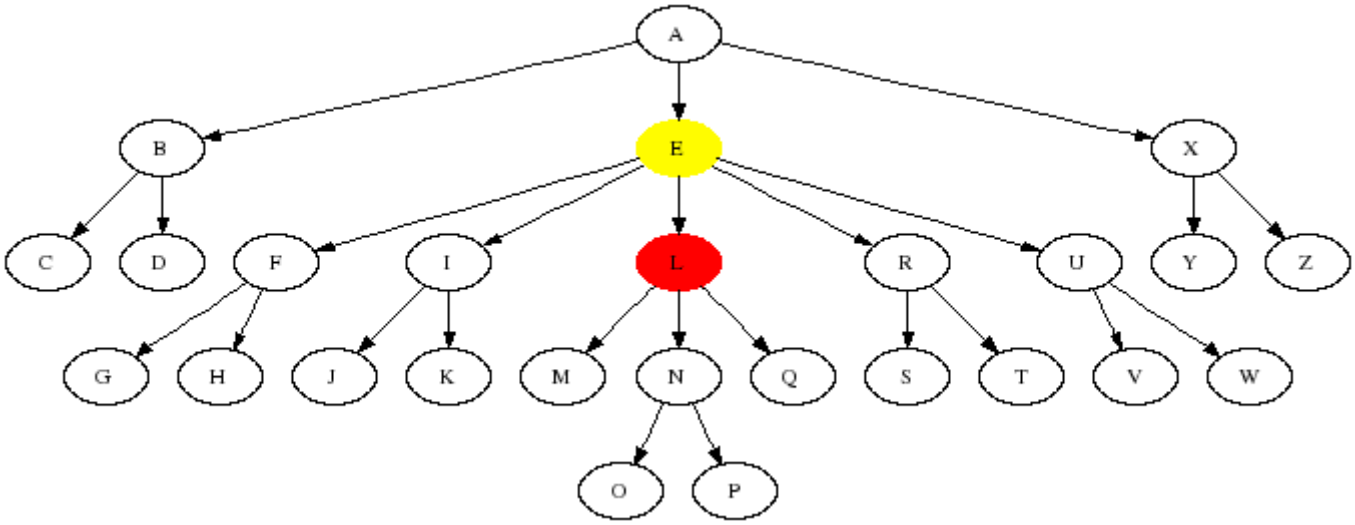


L' **ordine del documento** è un ordine totale definito sugli elementi XML di un documento. Un elemento A precede un elemento B nell'ordine del documento se il tag iniziale di A viene prima del tag di partenza di B che legge il documento dall'alto verso il basso. Si noti che l'ordine del documento corrisponde al **preordine** sull'albero XML. Ad esempio, se si legge il documento alfabetico nell'ordine del documento o l'albero dell'alfabeto in preordine, si ottiene l'alfabeto inglese in ordine decrescente.

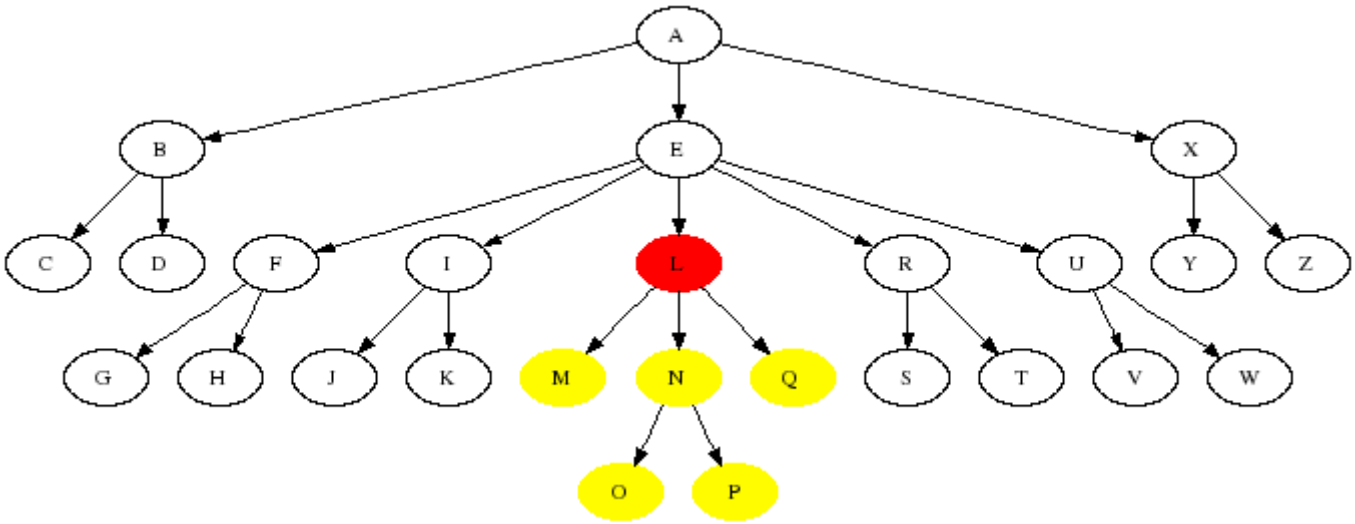
Ecco il primo esempio di percorso di posizione (in effetti, un passaggio di posizione): `child::*`. Il risultato di questa espressione è l'insieme di nodi **figlio** elemento del nodo contesto indipendentemente dai nomi (tag). Ecco un esempio grafico: il nodo di contesto è il nodo rosso e il set di nodi di risultato contiene i nodi gialli:



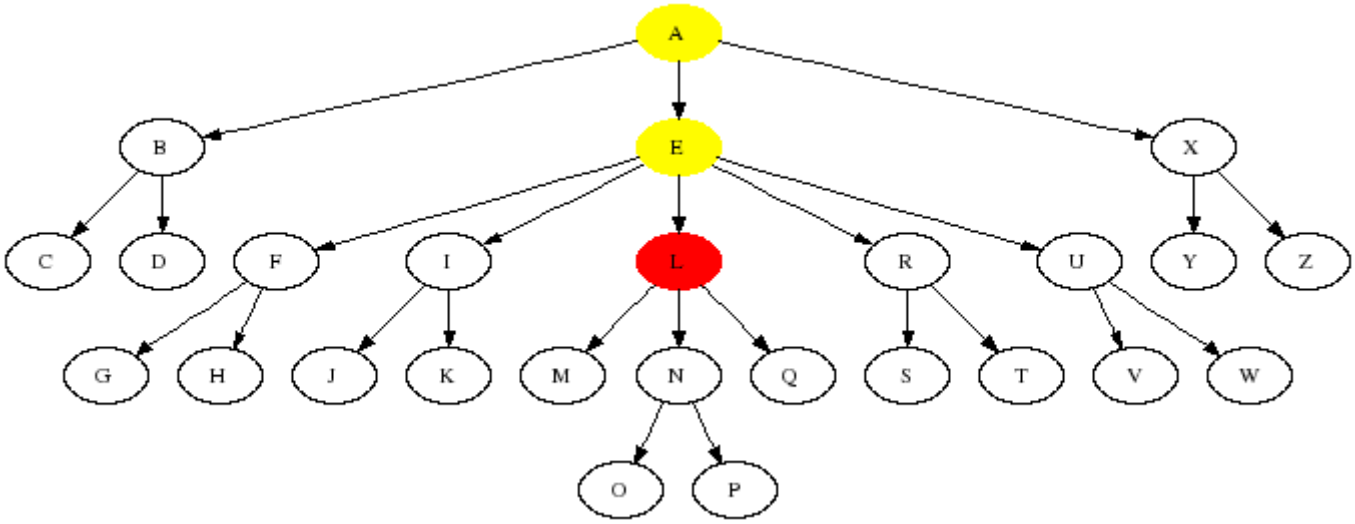
Il gradino `parent::*` recupera il nodo **genitore** dell'elemento del nodo di contesto. Segue un esempio grafico (il nodo di contesto è sempre il nodo rosso e il set di nodi risultato contiene sempre i nodi gialli):



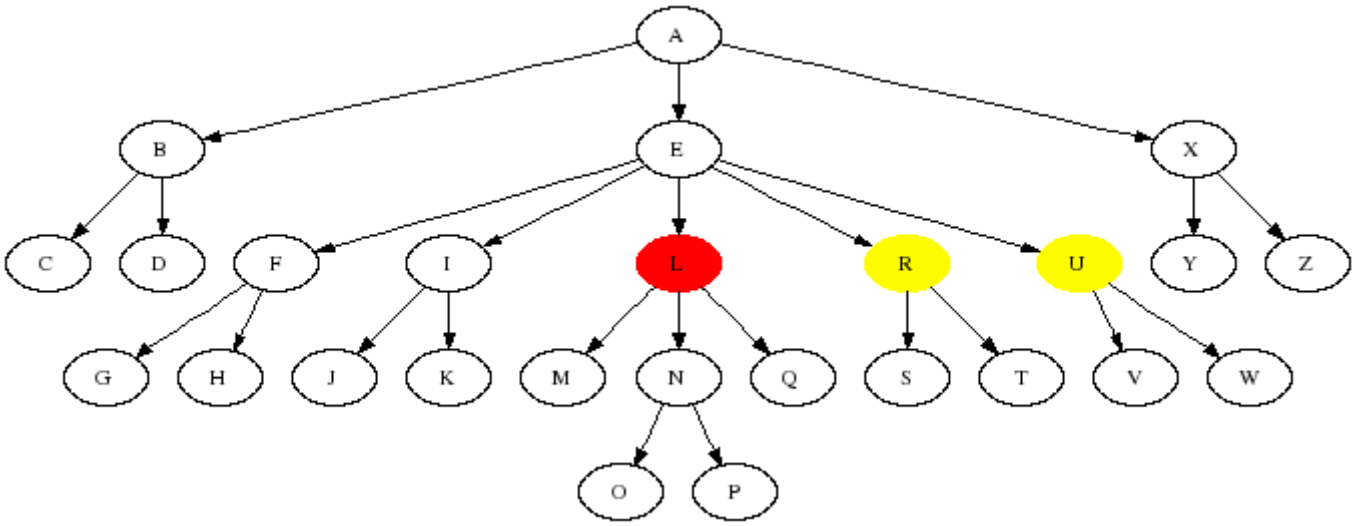
Il gradino `descendant::*` recupera i nodi **discendenti** dell'elemento del nodo di contesto. Segue un esempio grafico:



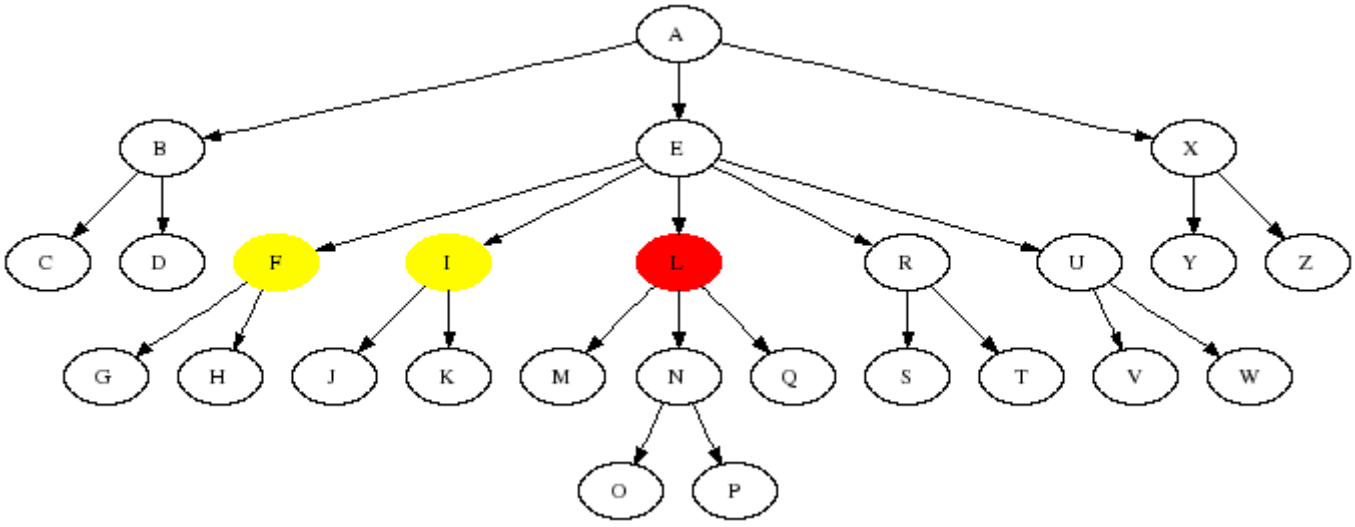
Il gradino `ancestor::*` recupera i nodi degli **antenati** dell'elemento del nodo di contesto. Segue un esempio grafico:



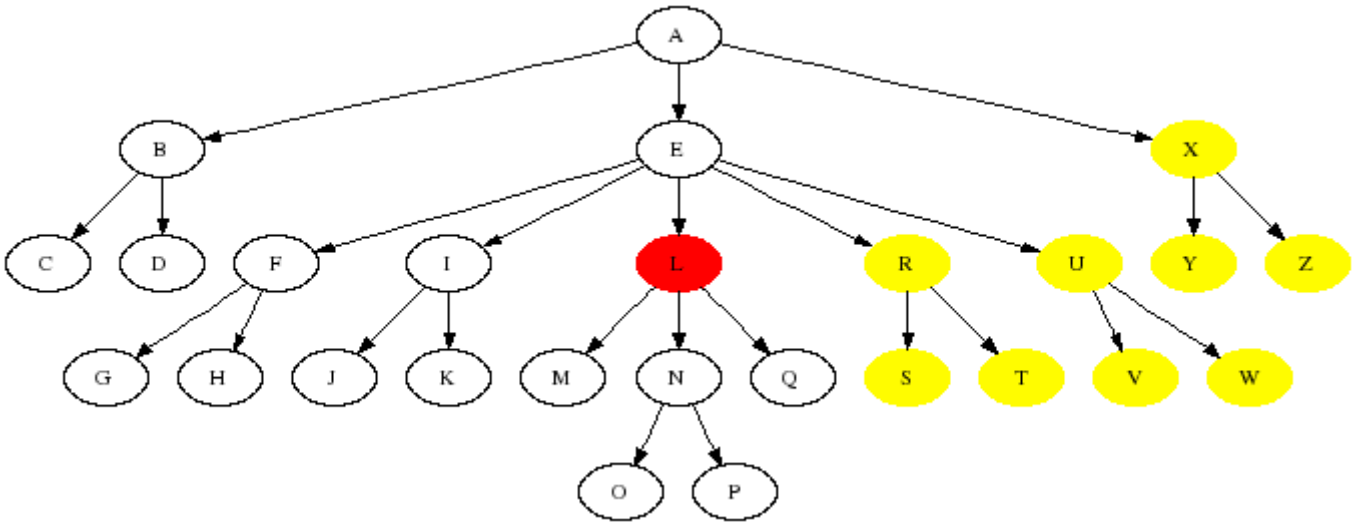
Il gradino `following-sibling::*` (significa fratello). recupera l'elemento nodi **fratelli giusti del** nodo di contesto. Segue un esempio grafico:



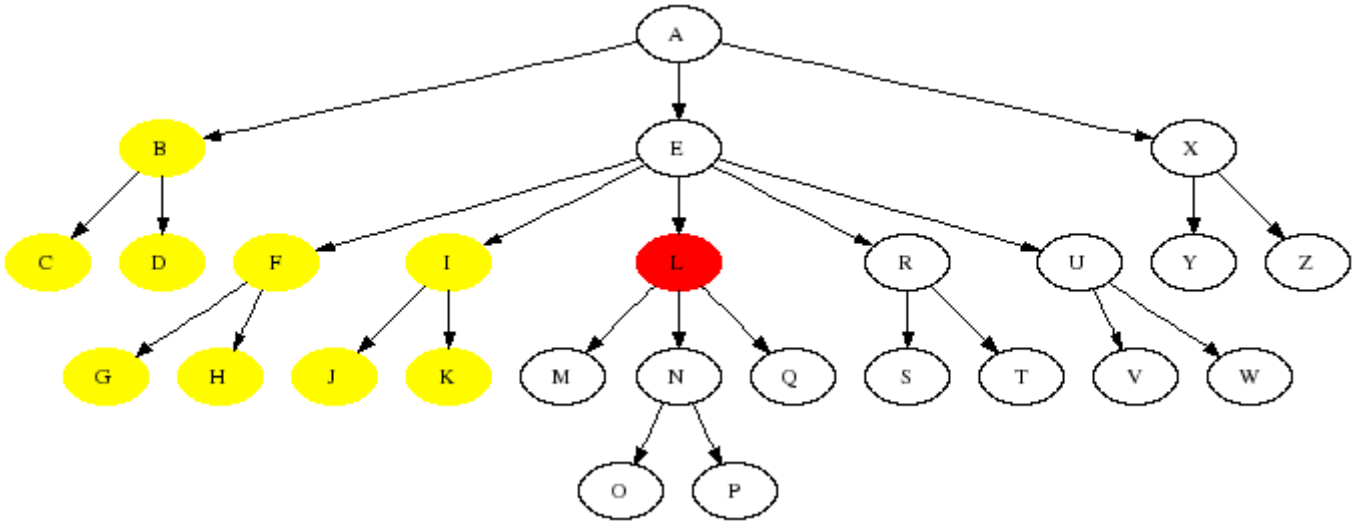
Il gradino `preceding-sibling ::*` (significa fratello). recupera l'elemento nodi **fratelli** di pari livello del nodo di contesto. Segue un esempio grafico:



Il gradino `a following::*` recupera i nodi elemento che **seguono** il nodo di contesto rispetto all'ordine del documento, escludendo i nodi discendenti. Segue un esempio grafico:



Il gradino `preceding::*` recupera i nodi elemento che **precedono** il nodo di contesto rispetto all'ordine del documento, escludendo i nodi antenati. Segue un esempio grafico:



Inoltre, il passo `descendant-or-self::*` recupera i nodi discendenti dell'elemento del nodo di contesto più il nodo di contesto stesso,

`antenato-or-self::*` recupera i nodi dell'elemento antenato del nodo di contesto più il nodo di contesto stesso,

`self::*` recupera il nodo del contesto stesso,

`attribute::*` recupera i nodi di attributo del nodo di contesto

e infine

`namespace :: *` recupera i nodi dello spazio dei nomi del nodo di contesto.

Finora abbiamo usato solo il test * che soddisfa qualsiasi nodo elemento indipendentemente dal nome. Altri test rilevanti includono:

nome
: Lungo tutti gli assi ma attributo e spazio dei nomi, abbina tutti gli elementi con il tag specificato. Lungo l'asse degli attributi, corrisponde a tutti gli attributi con il nome specificato. Lungo l'asse dello spazio dei nomi, corrisponde a tutti gli spazi dei nomi con il prefisso specificato.

nodo()
: Corrisponde a tutti i nodi indipendentemente dal tipo. Tuttavia, notalo`child :: node ()` seleziona tutti i figli del nodo di contesto ma gli attributi e gli spazi dei nomi, poiché, in base al modello di dati XPath, non sono figli del nodo di contesto. Per selezionare tutti gli attributi del nodo di contesto, utilizzare `attribute::*`, o `namespace ::*` per gli spazi dei nomi.

testo()
: Corrisponde a tutti i nodi di testo.

commento()
: Corrisponde a tutti i nodi dei commenti.

processing-instruction ()
: Corrisponde a tutti i nodi di istruzioni di elaborazione. Con una stringa come argomento, seleziona tutte le istruzioni di elaborazione con quell'obiettivo.

Ad esempio, considera nuovamente il seguente esempio:

```
<?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. -->
```

Cerchiamo di impostare il nodo di contesto sul nodo della persona. Poi `child :: node ()` corrisponde a tutti i nodi figli, cioè il `nome` e `professione` elemento figlio nodi e l'unico nodo figlio di testo, `bambino::*` corrisponde solo ai nodi figlio

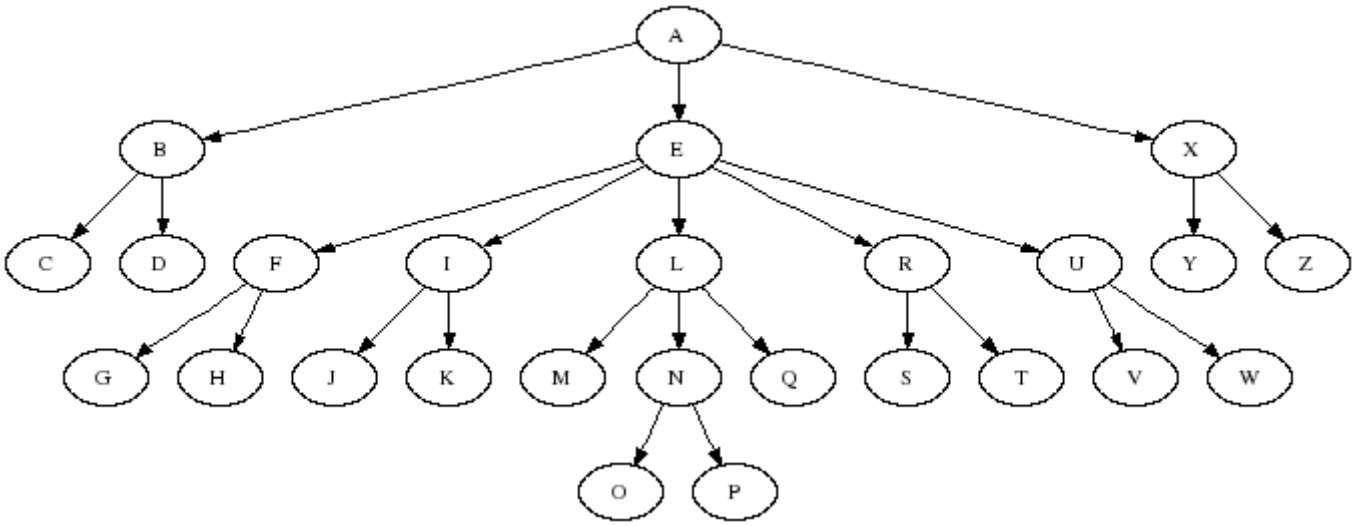
elemento, `child :: profession` matcha solo `profession` elemento figlio nodi, `child :: text ()` corrisponde al nodo di testo, `attribute::*` corrisponde a entrambi gli attributi e `attribute :: born` corrisponde al `born` attributo. Inoltre, se il nodo di contesto è la radice, allora `child::comment()` seleziona i due commenti dell'elemento del documento e `child::processing-instruction('xml-stylesheet')` seleziona l'istruzione di elaborazione del foglio di stile.

L'ultimo componente di un passo di posizione è il filtro opzionale. Poiché i filtri possono contenere percorsi di localizzazione, rimandiamo la loro introduzione dopo la discussione dei percorsi di localizzazione.

Percorsi di localizzazione XPath

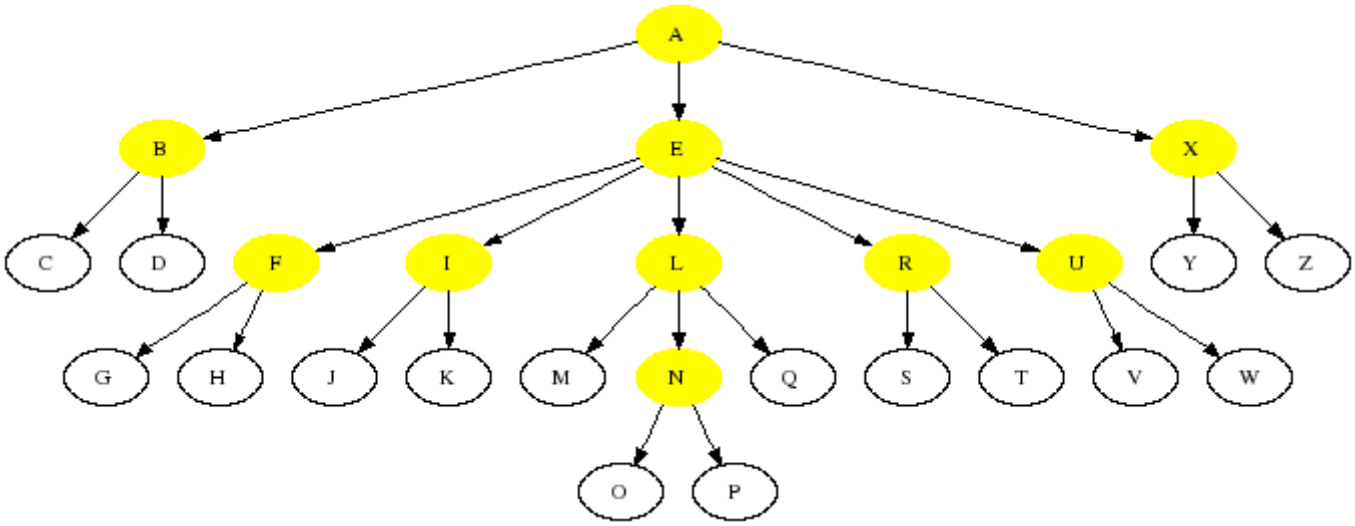
I passaggi di posizione possono essere combinati in **percorsi di posizione** . Un percorso inizia opzionalmente con un segno di barra e continua con una sequenza di almeno un passo di posizione. I passaggi sono separati da segni di barra. Un sentiero ben formato è `/ /descendant::L/child::*/child::*`

Un percorso dovrebbe essere valutato come segue. Se il percorso inizia con /, il nodo contesto viene impostato sul nodo radice, altrimenti un nodo contesto dovrebbe essere fornito come input. Il primo passaggio viene valutato nel nodo contesto risultante in un *insieme* di nodi di contesto, denominato *set* di nodi di **contesto** . Quindi, il secondo passo viene valutato in ciascun nodo del contesto risultante dalla valutazione del primo passo. I gruppi di nodi risultanti vengono uniti per formare il risultato del secondo passo. E così via. Il risultato dell'ultimo passo è anche il risultato del percorso. Ad esempio, considera nuovamente l'esempio di alfabeto:

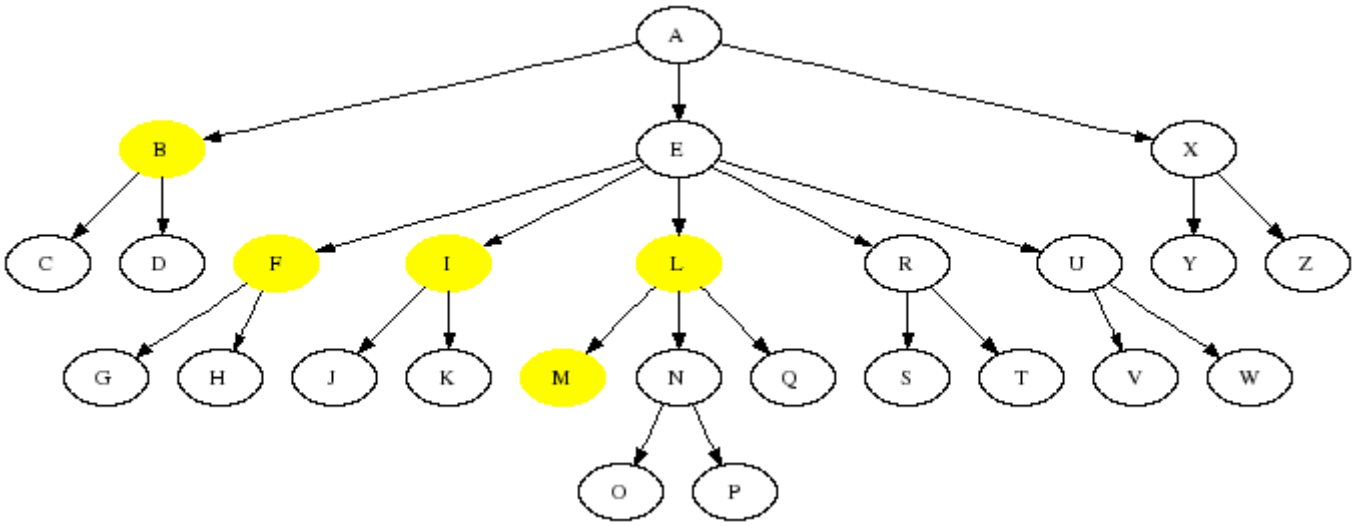


Il sentiero `/descendant::L/child::*/child::*` seleziona i nodi etichettati con O e P. In effetti, la radice è il nodo di contesto iniziale e il primo passo seleziona tutti i discendenti di radice etichettati con L. Il risultato contiene solo il nodo L. Il secondo passo recupera tutti i figli di L. Questi sono i nodi M , N e Q. Infine, il terzo passo seleziona tutti i figli di M, N e Q e prende l'unione dei set risultanti. Il risultato sono i nodi O e P.

Ricorda che un passaggio ha la forma `axis::test[filter]` dove `[filter]`è facoltativo. I percorsi di posizione possono essere utilizzati nei **filtri** dei passaggi di posizione. La parte del filtro, come suggerisce il nome, viene utilizzata per filtrare l'insieme di nodi selezionati dal percorso. Il processo di filtraggio è il seguente: il filtro viene applicato a ciascun nodo nel set corrente di nodi. Se il risultato non è vuoto, il nodo viene lasciato nel set di nodi corrente, altrimenti il nodo viene cancellato dal set di nodi corrente. Ad esempio, con riferimento all'esempio alfabetico, il passo `/descendant::*[child::*]` seleziona tutti i nodi interni (nodi con almeno un figlio): tutti i discendenti di root vengono prima selezionati e quindi i discendenti che hanno almeno un figlio sono lasciati nel set. Il risultato è colorato in giallo:



Come altro esempio, il passaggio successivo seleziona tutti i nodi che hanno almeno due fratelli a destra: `/descendant::*[following-sibling::*/following-sibling::*]`. Il risultato è colorato in giallo:



I percorsi di posizione nei filtri possono essere combinati con i collegamenti booleani `e`, `o`, e `non`. Ad esempio, il percorso `/descendant::*[not(child::*)]` seleziona tutti i nodi foglia (nodi senza figli), il percorso `/descendant::*[child::* and preceding-sibling::*]` seleziona tutti i nodi interni con almeno un fratello sinistro e il percorso `/descendant::*[ancestor::B or preceding::B]` seleziona tutti i nodi che seguono B nell'ordine del documento.

Operatori XPath e funzioni

Il frammento di XPath che abbiamo descritto finora è solitamente chiamato **XPath di** navigazione o **core** . In sostanza fornisce tutte le funzionalità per navigare 0 XML. Tuttavia, XPath completo offre di più, tra cui:

- Operatori di confronto, come =,! =, <, <=,>,> =;
- Operatori numerici, come +, -, *, div (divisione), mod (resto);
- funzioni.

Considera il seguente documento XML contenente un estratto di una bibliografia:

```
<biblio>
   <inproceedings key="M4M05" cite="JLLI05">
      <author>M. Franceschet</author>
      <author>E. Zimuel</author>
      <title>Modal logic and navigational XPath: an experimental
comparison</title>
      <booktitle>Workshop Methods for Modalities</booktitle>
      <pages>156-172</pages>
      <year>2005</year>
      <url>http://www.sci.unich.it/~francesc/pubs/m4m05.pdf</url>
   </inproceedings>

   <article key="JLLI05" cite="M4M05">
```

```
      <author>M. Franceschet</author>
      <author>B. ten Cate</author>
      <title>Guarded fragments with constants</title>
      <journal>Journal of Logic, Language and Information</journal>
      <volume>14</volume>
      <number>3</number>
      <pages>281-288</pages>
      <year>2005</year>
      <url>http://www.sci.unich.it/~francesc/pubs/jlli05.pdf</url>
      <price>15</price>
   </article>
</biblio>
```

Segue una DTD semplice per il documento sopra riportato:

```
<!ELEMENT biblio (article | inproceedings)*>

<!ELEMENT article (author+,title,journal,volume,number,pages,year,url,price)>
<!ATTLIST article key ID #REQUIRED
                  cite IDREFS #IMPLIED>

<!ELEMENT inproceedings (author+,title,booktitle,pages,year,url)>
<!ATTLIST inproceedings key ID #REQUIRED
                        cite IDREFS #IMPLIED>

<!ELEMENT author    (#PCDATA)>
<!ELEMENT title     (#PCDATA)>
<!ELEMENT booktitle (#PCDATA)>
<!ELEMENT pages     (#PCDATA)>
<!ELEMENT year      (#PCDATA)>
<!ELEMENT journal   (#PCDATA)>
<!ELEMENT volume    (#PCDATA)>
<!ELEMENT number    (#PCDATA)>
<!ELEMENT url       (#PCDATA)>
<!ELEMENT price     (#PCDATA)>
```

Gli operatori di confronto possono essere usati per confrontare il valore di stringa di un nodo. Ad esempio, la query:

```
/child::biblio/child::*[child::author = "E. Zimuel"]
```

recupera tutti gli articoli bibliografici con un certo autore chiamato `E. Zimuel`. Le stringhe devono essere citate singole o doppie. La seguente query seleziona tutti gli articoli pubblicati dopo l'anno 2000:

```
/child::biblio/child::article[child::year > 2000]
```

Siccome `2000` non è quotato, è considerato come un numero e il segno> è interpretato come l'operatore maggiore di numeri. Se scrivi `child::year > "2000"`, allora `2000` è considerato come una stringa, e il segno> è interpretato come un operatore più grande di quello su stringhe (l'ordine lessicografico).

XPath definisce un numero di funzioni che è possibile utilizzare nei filtri (di solito) o nelle espressioni non elaborate. Ogni funzione restituisce uno dei quattro tipi di base: stringa,

numero, booleano, set di nodi. L'unica funzione booleana rilevante è `not()` che completa la sua argomentazione. Altre funzioni rilevanti seguono:

Funzioni del set di nodi

Queste funzioni operano o restituiscono informazioni sui set di nodi. I più rilevanti sono:

`position ()`

Restituisce la posizione, rispetto all'ordine del documento, del nodo di contesto nel set di nodi di contesto. Ad esempio, la query successiva restituisce il primo autore della voce bibliografica con chiave`M4M05`:

```
/child::biblio/child::*[attribute::key = "M4M05"]/child::author[position() = 1]
```

`last ()`

Restituisce la posizione dell'ultimo elemento nel set di nodi di contesto, ovvero la cardinalità del set di nodi di contesto. Ad esempio, la query successiva restituisce l'ultimo autore della voce bibliografica con chiave`M4M05`:

```
/child::biblio/child::*[attribute::key = "M4M05"]/child::author[position()
= last()]
```

`count(path)`

Restituisce la cardinalità del set di nodi di contesto risultante dalla valutazione dell'argomento `sentiero`. Ad esempio, la query successiva restituisce il numero di autori della voce bibliografica con chiave`M4M05`:

```
count(/child::biblio/child::*[attribute::key = "M4M05"]/child::author)
```

mentre il prossimo seleziona i documenti della conferenza con più di 3 autori:

```
/child::biblio/child::inproceedings[count(child::author) > 3]
```

`id(string)`

Prende come input una stringa contenente uno o più ID separati da spazi bianchi e restituisce un set di nodi contenente tutti i nodi nel documento che hanno quegli ID. La stringa di input può essere una stringa statica o una stringa dinamica ottenuta come risultato della valutazione di una query XPath. Ad esempio, la query successiva restituisce la voce con ID`M4M05`:

```
id ( "M4M05")
```

mentre il prossimo seleziona le voci che vengono citate dalla voce con ID `M4M05`:

```
id(/child::biblio/child::*[attribute::key = "M4M05"]/attribute::cite)
```

Puoi anche scrivere l'ultimo come:

```
/child::biblio/child::*[attribute::key = "M4M05"]/id(attribute::cite)
```

Si noti che il `id()` le funzioni funzionano solo se sono stati dichiarati correttamente gli attributi ID e IDREF nel DTD del documento.

Funzioni di stringa

Le funzioni più rilevanti che operano sulle stringhe sono:

`contains(string1,string2)`

Restituisce vero se la prima stringa contiene il secondo. Entrambe le stringhe possono derivare dalla valutazione di un'espressione XPath. Ad esempio, la query che segue recupera tutte le voci che contengono nel titolo la stringa `logica`:

```
/child::biblio/child::*[contains(child::title,"logic")]
```

`starts-with(string1,string2)`

Restituisce vero se la prima stringa inizia con il secondo. Entrambe le stringhe possono derivare dalla valutazione di un'espressione XPath.

Funzioni numeriche

La funzione più rilevante che opera sui numeri è `sum (percorso)` che accetta un nodo impostato come argomenti e somma i valori di tutti i nodi nell'insieme dopo la conversione in numeri. Ad esempio, la query successiva restituisce il prezzo totale di tutti gli articoli pubblicati nel 2005:

```
sum (/ child :: biblio / child :: article [year = "2005"] / price)
```

Sintassi abbreviata XPath

Alcuni assi sono usati più di altri. Per loro, sono disponibili le seguenti **scorciatoie** :

- `child :: un` può essere abbreviato come `un`
- `/ descendant :: un` può essere abbreviato con `//a`
- `attribute::a un` può essere abbreviato con `@a`
- `self::*` può essere abbreviato con as `.`
- `parent::*` può essere abbreviato con `..`

Ad esempio, la query:

```
/descendant::a/child::b[attribute::c]/parent::*[self::* = "w"]
```

può essere scritto come segue:

```
//a/b[@c]/..[. = "w"]
```

Si noti che la query `//un[//b]` si espande a `/descendant::a[/descendant::b]`. Quest'ultimo seleziona tutti i discendenti di radice etichettati con `un` *se* esiste un discendente *radice* etichettato con `b` e restituisce il set vuoto altrimenti. In particolare, non è equivalente a `/descendant::a[descendant::b]`, che seleziona tutti i discendenti di radice etichettati con `a` avere un discendente etichettato con `b`. Non esiste alcuna abbreviazione per `descendant::b`.

Puoi provare tutte le query precedenti con qualsiasi processore XPath. Un **processore XPath** è un software che valuta le query XPath. BaseX è un completo processore XPath basato su Java con una bella interfaccia grafica web. Mostra risultati in diversi formati tra cui testo, albero e mappa ad albero. Saxon è un processore da riga di comando XSLT e XQuery. Poiché XPath è utilizzato sia in XSLT che in XQuery, puoi provare anche le query XPath con Saxon. Ad esempio, per valutare la query `/descendant::article` sul documento XML `biblio.xml`, esegui il seguente comando:

```
java net.sf.saxon.Query -s biblio.xml "{/ descendant :: article}"
```

L'opzione `-S` imposta il nodo di contesto iniziale sulla radice del documento XML specificato. Se preferisci memorizzare la query nel file `articles.xpl`, allora puoi digitare il seguente comando:

```
java net.sf.saxon.Query -s biblio.xml articles.xpl
```

Esecuzione di esempio

Scrivi in XPath le seguenti domande per l'applicazione DBLP ed eseguine:

1. La tesi di dottorato di Andreas Neumann

```
/dblp/phdthesis[author = "Andreas Neumann"]
```

2 Tutti gli articoli di riviste con almeno quattro autori

```
/dblp/article[count(author) >= 4]
```

3 Tutti gli articoli pubblicati nel 2000 in una conferenza o in una rivista che contengono la parola *XML* nel titolo

```
/ dblp / * [anno = 2000 e contiene (titolo, "XML") e
      (auto :: inproceedings o self :: article)]
```

oppure

```
/dblp/inproceedings[year = 2000 and contains(title, "XML")] |
/dblp/article[year = 2000 and contains(title, "XML")]
```

4La percentuale di articoli di giornale nella bibliografia

```
count(/dblp/article) div count(/dblp/*)
```

5 La percentuale di giornali di riviste individuali (articoli con un solo autore)

```
count (/ dblp / article [count (author) = 1]) div count (/ dblp / article)
```

6 Il numero medio di autori di documenti per conferenze

```
count(/dblp/inproceedings/author) div count(/dblp/inproceedings)
```

7 Il primo autore di tutti i lavori scritti da Massimo Franceschet

```
/dblp/*[author = "Massimo Franceschet"]/author[position() = 1]
```

8 I primi e gli ultimi autori di carta con riviste chiave / ci / CervesatoFM00

```
/dblp/*[@key = "journals/ci/CervesatoFM00"]/author[position() = 1] |
/dblp/*[@key = "journals/ci/CervesatoFM00"]/author[position() = last()]
```

9 Tutti gli autori che seguono Massimo Franceschet in formato cartaceo con riviste chiave / ci / CervesatoFM00

```
dblp/*[@key = "journals/ci/CervesatoFM00"]
/author[preceding-sibling::author[. = "Massimo Franceschet"]]
```

oppure

```
/dblp/*[@key = "journals/ci/CervesatoFM00"]
/author[. = "Massimo Franceschet"]/following-sibling::author
```

10. Tutti gli atti che seguono la carta con le principali riviste / ci / CervesatoFM00 nella bibliografia

```
dblp/*[@key = "journals/ci/CervesatoFM00"]/following::proceedings
```

11. Tutti `i` elementi nel titolo di un documento (a qualsiasi profondità)

```
/dblp/*/title//i
```

XML Query Language

Il **linguaggio di query XML** (**XQuery**) è un linguaggio di query completo per i database XML. Si distingue per i database XML in quanto SQL corrisponde a quelli relazionali. Un **database XML** è una raccolta di documenti XML (correlati).

XQuery funziona su **sequenze** , non su set di nodi come XPath. Una sequenza contiene elementi che sono nodi XML (come elementi e attributi) o valori atomici (come stringhe e numeri). La relazione tra XPath e XQuery consiste nel fatto che le espressioni XPath vengono utilizzate nelle query XQuery (*xqueries*). Quindi, possiamo considerare XPath come un frammento sintattico di XQuery.

Un tipico xquery funziona come segue:

1. **caricamento** : uno o più documenti XML vengono caricati dal database;
2. **recupero** : le espressioni XPath vengono utilizzate per recuperare sequenze di nodi ad albero dai documenti caricati;
3. **processo** : le sequenze di nodi recuperate vengono elaborate con operazioni XQuery come il filtraggio (creando una nuova sequenza selezionando alcuni degli elementi di quella originale) e ordinando (ordinando gli elementi della sequenza in base ad alcuni criteri);
4. **costrutto** : nuove sequenze possono essere costruite e combinate con quelle recuperate;
5. **output** : una sequenza finale viene restituita come output.

Alcuni dei passaggi precedenti sono opzionali. Ad esempio, un xquery può evitare il caricamento di un documento e il recupero di sequenze con espressioni XPath e, invece, può creare, elaborare e generare le proprie sequenze. Infatti, XQuery può essere usato come un *linguaggio di programmazione* (completo di Turing) che manipola sequenze di elementi, possibilmente utilizzando funzioni definite dall'utente. In queste pagine ci concentreremo su XQuery come linguaggio di query XML.

Un **processore XQuery** è un software che valuta xqueries. Vedi la pagina delle risorse XQuery per un elenco di processori XQuery. Saxon è un buon esempio. Al fine di valutare con Saxon la query contenuta nel fil e`exquery.xql`, digitare quanto segue:

```
java net.sf.saxon.Query xquery.xql
```

XQuery: muovendo i primi passi

Impareremo XQuery con l'esempio. Proponiamo una serie di query da interpretare su documenti generati da XMark .

Il più semplice xquery carica solo un documento XML: il `doc()` funzione carica il documento XML dato come argomento, come nell'esempio seguente:

```
doc ( "auction.xml")
```

Il risultato della query è l'intero documento caricato. Come detto sopra, le query XPath sono xqueries valide. In genere, il documento su cui deve essere valutata la query XPath viene caricato e quindi la query viene applicata a questo documento. Ecco un esempio:

```
doc("auction.xml")/site/people/person[not(watches/watch)]
```

Il risultato è una sequenza di `persona` elementi, ordinati in base all'ordine del documento, che corrisponde alle persone che non guardano alcuna asta aperta.

Espressioni FLWOR

Le espressioni **FLWOR** (pronunciato *fiore*) sono le espressioni più comuni in XQuery. Sono simili alle istruzioni select-from-where in SQL. Il nome FLWOR è un acronimo, che rappresenta la prima lettera delle clausole che possono verificarsi in una tale espressione:

for

`le clausole for` associano una o più variabili alle espressioni XQuery. Ogni variabile *iterativamente* è associata a un valore dell'espressione corrispondente.

let

`le clausole let` associano le variabili *all'intero* risultato di un'espressione XQuery. Una sequenza di associazioni di variabili create da For e `Let` di un'espressione FLWOR sono chiamate **tuple** .

where

le clausole `dove` filtrano le tuple mantenendo solo quelle che soddisfano una data condizione;

order by

le clausole `order by` ordinano le tuple secondo i criteri dati;

return

le clausole `return` costruiscono il risultato dell'espressione.

Un'espressione FLWOR inizia con uno o più clausole `per` o `for` in qualsiasi ordine, seguito da una clausola opzionale, seguita da una clausola opzionale `order by`, seguita da una clausola obbligatoria `return`.

La clausola per

La clausola Per associa iterativamente una variabile ai valori di un'espressione. Ecco un esempio:

```
per $ i in (1,2,3)
return <tuple> {$ i} </ tuple>
```

In XQuery, le variabili sono precedute dal prefisso `$`. Il `for` associa la variabile `i` agli elementi della sequenza `(1,2,3)`. Per ogni assegnazione di variabile, il`ritorno` la clausola restituisce il valore contenuto nella variabile `io` racchiuso in a `tuple`elemento. Si noti che la variabile`io` è racchiuso tra parentesi graffe quando usato in `tuple` costruttore di elementi della clausola `return`. Ciò significa che il *valore* dell'espressione, e non l'espressione stessa, deve essere stampato. Il risultato della query è il seguente:

```
<Tuple> 1 </ tuple>
<Tuple> 2 </ tuple>
<Tuple> 3 </ tuple>
```

Più di una variabile può essere assegnata in una clausola `per`, come nell'esempio seguente:

```
for $i in (1,2), $j in (1,2)
return <tuple>
         <i>{$i}</i>
         <j>{$j}</j>
       </tuple>
```

che produce il seguente output:

```
<tuple>
   <i>1</i>
   <j>1</j>
</tuple>
<tuple>
   <i>1</i>
   <j>2</j>
</tuple>
<tuple>
   <i>2</i>
   <j>1</j>
</tuple>
<tuple>
   <i>2</i>
   <j>2</j>
</tuple>
```

Infine, ecco un esempio sui documenti di XMark. La query restituisce il nome e l'indirizzo email di tutte le persone nel documento `auction.xml`. Si noti che i commenti devono essere introdotti tra `(:` e `:)` segni:

```
(: The name and email address of all persons :)

for $p in doc("auction.xml")/site/people/person
return <person>
         {$p/name}
         {$p/emailaddress}
       </person>
```

La clausola let

La clausola `let` assegna una variabile all'intero risultato di un'espressione. Ecco un esempio:

```
let $i := (1,2,3)
return <tuple> {$ i} </ tuple>
```

Il risultato è un *unico* elemento `tupla` contenente l'intera sequenza assegnata nella clausola `let`:

```
<tuple> 1 2 3 </ tuple>
```

Per concludere, ecco un esempio più coinvolgente che utilizza entrambi `for` e `let`clausole. La query restituisce tutte le aste aperte con il numero corrispondente di offerenti. La variabile`un` è assegnato a tutte le aste aperte dal `per`clausola. Per ogni asta aperta, il`permettere` la clausola assegna la variabile `b`a tutti gli offerenti dell'attuale asta aperta. Infine, per ogni asta aperta, il`ritorno` clausola emette l'asta `id` attributo purché il numero di offerenti dell'asta:

```
per $ a in doc ("auction.xml") / site / open_auctions / open_auction
lascia $ b: = $ a / offerente
return <auction id = "{$ a / @ id}">
            <Bidder_count> {count ($ b)} </ bidder_count>
         </ Asta>
```

Le clausole where e order by

Un incarico delle variabili usate nel `for` e `let`le clausole di una query sono chiamate **tuple** . Le tuple generate da una query possono essere filtrate con`dove` clausola e ordinati con il `ordinato da`clausola. Ad esempio, la seguente query restituisce solo le aste aperte con un numero di offerenti compreso tra 1 e 3:

```
for $a in doc("auction.xml")/site/open_auctions/open_auction
```

```
let $b := $a/bidder
where count($b) >= 1 and count($b) <= 3
return <auction id = "{$a/@id}">
         <bidder_count>{count($b)}</bidder_count>
       </auction>
```

La query successiva ordina l'asta aperta in relazione al numero di offerenti dell'asta:

```
for $a in doc("auction.xml")/site/open_auctions/open_auction
let $b := $a/bidder
order by count($b)
return <auction id = "{$a/@id}">
         <bidder_count>{count($b)}</bidder_count>
       </auction>
```

È possibile utilizzare più di una chiave di ordinamento, separata da una virgola. Per impostazione predefinita, l'ordine è crescente. Un ordine decrescente può essere ottenuto aggiungendo la parola chiave`discendente`dopo la corrispondente chiave di ordinamento. Segue un'istanza:

```
for $a in doc("auction.xml")/site/open_auctions/open_auction
let $b := $a/bidder
order by count($b) descending, number($a/current)
return <auction id = "{$a/@id}">
         <bidder_count>{count($b)}</bidder_count>
         {$a/current}
       </auction>
```

Query di annidamento

In tutti i nostri esempi, `for e let` le clausole contengono espressioni XPath. Tuttavia, possono contenere anche espressioni FLWOR più complesse, che possono contenere altre espressioni FLWOR e così via. Ricorda che un'espressione FLWOR valida deve iniziare con almeno una clausola `for o let` e deve terminare con la clausola `return` .

La seguente query recupera il nome e l'indirizzo e-mail di tutte le persone che hanno acquistato più di un oggetto e li stampa ordinati per nome. Il secondo`permettere`la clausola usa un'espressione FLWOR nidificata per *unirsi* al tipo IDREF`persona` attributo di a `acquirente` elemento di un'asta chiusa con il tipo ID `id` attributo di a `persona` elemento:

```
let $doc := doc("auction.xml")
for $p in $doc/site/people/person
let $a := for $b in $doc/site/closed_auctions/closed_auction
          where $b/buyer/@person = $p/@id
          return $b
where count($a) >= 2
order by $p/name
return <person>
          {$p/name}
          {$p/emailaddress}
          <items>{count($a)}</items>
```

```
       </person>
```

Si noti che possiamo evitare le clausole di nidificazione di for utilizzando un filtro XPath come segue:

```
let $doc := doc("auction.xml")
for $p in $doc/site/people/person
let $a := $doc/site/closed_auctions/closed_auction[buyer/@person = $p/@id]
where count($a) >= 2
order by $p/name
return <person>
          {$p/name}
          {$p/emailaddress}
          <items>{count($a)}</items>
       </person>
```

La prossima query restituisce le persone, ordinate in base al loro reddito in ordine decrescente, accompagnate con le annotazioni che hanno fatto ad un'asta chiusa. Le persone che non hanno fatto annotazioni non vengono stampate (sono escluse dall'esterno tramite clausola `dove`). Ancora una volta, la sottoquery viene utilizzata per unire un attributo di tipo IDREF a uno di tipo ID.

```
let $doc := doc("auction.xml")
for $p in $doc/site/people/person
let $a := for $b in $doc/site/closed_auctions/closed_auction
          where $b/annotation/author/@person = $p/@id
          return $b/annotation
where count($a) > 0
order by $p/profile/number(@income) descending
return <person id="{$p/@id}" income="{$p/profile/@income}">
          <annotations>{$a}</annotations>
       </person>
```

Il prossimo esempio seleziona le persone e gli oggetti europei che hanno acquistato (vengono stampate solo le persone che hanno acquistato almeno un oggetto europeo):

```
let $doc := doc("auction.xml")
for $p in $doc/site/people/person
let $item := for $a in $doc/site/closed_auctions/closed_auction
             where         ($a/buyer/@person         =         $p/@id)         and
$a/itemref/id(@item)/parent::europe
             return $a/itemref
where count($item) > 0
return <person id="{$p/@id}" items="{$item/@item}
```

Funzioni definite dall'utente

In XQuery puoi definire le tue funzioni da utilizzare nelle query. Ecco un esempio di una funzione *ricorsiva*che restituisce le categorie che sono raggiungibili da category0 attraverso un percorso *arbitrario* nel grafico della categoria:

```
declare function local:closure($input as node()*, $result as node()*) as node()*
{
```

```
   let $current := //edge[@from = $input]/@to
   let $new := $current except $result
   let $all := ($result, $new)
   return
      if(exists($new))
      then ($new, local:closure($new,$all))
      else ()
};

for $c in local:closure(id("category0")/@id,())
return <category id="{$c}"/>
```

La definizione di cui sopra merita qualche commento:

- Per distinguere le funzioni integrate (come il conteggio e la somma) da quelle definite dall'utente, le funzioni devono essere definite all'interno di uno spazio dei nomi. È possibile utilizzare il built-in`Locale` namespace senza dichiararlo;
- il nome della funzione deve essere un nome XML valido. Possono esserci più di una dichiarazione di funzione con lo stesso nome qualificato a condizione che il numero di parametri sia diverso;
- per ciascun parametro di funzione è necessario specificare un nome di variabile, un tipo e un indicatore di occorrenza. Per il risultato della funzione è necessario specificare un tipo e un indicatore di ricorrenza. Il tipo è un tipo di nodo (come node () ed element ()) o un tipo XSchema (come xs: string e xs: date). L'indicatore di occorrenza è sia ? (zero o uno), + (uno o molti) o * (zero, uno o molti). Ad esempio, la definizione `$input as node()*` specifica che l'input del parametro può essere una sequenza eventualmente vuota di nodi di qualsiasi tipo;
- il corpo della funzione è un'espressione racchiusa tra parentesi graffe, che può essere qualsiasi espressione XQuery valida. Non deve contenere una clausola di ritorno; il valore di ritorno è semplicemente il valore dell'espressione. All'interno del corpo della funzione, una funzione può chiamare altre funzioni incluso se stessa (funzioni ricorsive).

Come secondo esempio, considera la bibliografia e migliorata con le citazioni. Segue una semplice istanza XML con 6 pubblicazioni:

```
<CITEX>
<article cites = "6">
  <Author> M. Franceschet </ author>
</ Article>
<article cites = "5">
  <Author> M. Franceschet </ author>
</ Article>
<article cites = "4">
  <Author> M. Franceschet </ author>
</ Article>
<article cites = "3">
  <Author> M. Franceschet </ author>
</ Article>
<article cites = "2">
  <Author> M. Franceschet </ author>
</ Article>
```

```
<article cites = "1">
  <Author> M. Franceschet </ author>
</ Article>
<CITEX>
```

L'indice h di un insieme di pubblicazioni è il numero più alto di pubblicazioni nel set che hanno ricevuto almeno quel numero di citazioni. Ad esempio, l'indice h del gruppo di pubblicazioni di cui sopra in 3, poiché ci sono al massimo 3 pubblicazioni che hanno ricevuto almeno 3 citazioni. La seguente funzione XQuery calcola l'indice h per l'insieme di pubblicazioni di un determinato autore:

```
declare function local: h ($ doc as node () *, $ author as xs: string) as xs:
intero
{
  lascia $ pub: = per $ x in $ doc / citex / * [autore = $ autore]
              ordina per xs: intero ($ x / @ cita) decrescente
              ritorno $ x
  $ cita: = per $ n in (1 per contare ($ pub))
                dove xs: intero ($ pub [$ n] / @ cita)> = $ n
                restituisce $ pub [$ n] / @ cita
  conteggio di ritorno ($ cita)
};
```

L'indice g è una variante dell'indice h menzionato. Dato un insieme di articoli classificati in ordine decrescente del numero di citazioni che hanno ricevuto, l'indice g di un insieme è il più grande numero di pubblicazioni in modo tale che gli articoli g migliori ricevano insieme almeno ^{2} citazioni. Ad esempio, l'indice g del gruppo di pubblicazioni di cui sopra è 4, perché la somma delle citazioni delle pubblicazioni top-4 è 18, che è maggiore di $4^2 = 16$, mentre la somma delle citazioni dei primi 5 le pubblicazioni sono 20, che è inferiore a $5^2 = 25$. La prossima funzione XQuery è un primo tentativo di calcolare l'indice g per l'insieme di pubblicazioni di un determinato autore:

```
declare function local:g($doc as node()*, $author as xs:string) as xs:integer
{
  let $pub := for $x in $doc/citex/*[author=$author]
              order by xs:integer($x/@cites) descending
              return $x
  $sum := 0
  let $cites := for $n in (1 to count($pub))
                let $sum := $sum + $pub[$n]/@cites
                where $sum >= ($n * $n)
                return $pub[$n]/@cites
  return count($cites)
};
```

La soluzione non è corretta poiché un ciclo for in XQuery viene eseguito in parallelo e non in modo sequenziale; quindi, la variabile $sum non contiene la somma delle citazioni fino alla pubblicazione corrente. Contiene le citazioni della pubblicazione corrente. Segue *una* soluzione corretta ma *inefficiente* :

```
declare function local:g($doc as node()*, $author as xs:string) as xs:integer
{
  let $pub := for $x in $doc/citex/*[author=$author]
              order by xs:integer($x/@cites) descending
```

```
              return $x
  let $cites := for $n in (1 to count($pub))
                let $seq := for $i in (1 to $n)
                            return $pub[$i]/@cites
                let $sum := sum($seq)
                where $sum >= ($n * $n)
                return $pub[$n]/@cites
  return count($cites)
};
```

Segue una soluzione *ricorsiva* efficiente :

```
declare function local:gRec($cites as xs:integer*,
                            $g as xs:integer,
                            $sum as xs:integer) as xs:integer {
   let $sumCites := $sum + $cites[1]
   let $new := remove($cites, 1)
   return
      if (exists($cites))
      then (if ($sumCites >= $g * $g)
            then local:gRec($new, $g+1, $sumCites)
            else $g - 1
           )
      else ($g - 1)
};

declare function local:g($doc as node()*, $author as xs:string) as xs:integer
{
  let $pub := for $x in $doc/citex/*[author=$author]
              order by xs:integer($x/@cites) descending
              return $x
  return local:gRec($pub/@cites, 1, 0)
};
```

Usiamo le funzioni locali definite come segue:

```
let $doc := /.
let $author := "M. Franceschet"
return
<bibliometrics author = "{$author}"
                  h = "{local:h($doc, $author)}"
                  g = "{local:g($doc, $author)}"/>
```

XQuery Full Text

XQuery Full Text estende XQuery con ricerche full text. I seguenti esempi possono essere eseguiti sulla commedia di Shakespeare A Midsummer Night's Dream, messa in formato XML da Jon Bosak .

Le righe che contengono la **parola** *hath*:

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

D'altra parte, la query seguente recupera le righe che contengono la **stringa** *hath* :

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

Si noti che la stringa di ricerca viene tokenizzata prima di essere confrontata con la stringa di input con token. Nel processo di tokenizzazione avvengono diverse normalizzazioni: ad esempio, vengono rimossi il maiuscolo / minuscolo e i segni diacritici (dieresi, accenti, ecc.) E viene applicato un algoritmo di derivazione opzionale dipendente dalla lingua. Inoltre, i caratteri speciali come gli spazi bianchi e i segni di punteggiatura saranno ignorati. Quindi la seguente ricerca produce lo stesso risultato:

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

Le righe contenenti sia le parole *morte* che *abjure* :

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

Le righe contenenti la parola *morte* o la parola *abjure* :

```
//LINE[ . contains text "death" ftand ftnot "abjure" ]
```

Le righe contenenti la parola *morte* ma non la parola *abjure* :

```
// LINE [. contiene testo "morte" ft e ftnot "abjure"]
```

Le righe contenenti la parola *Demetrius* in questo caso:

```
//LINE[ . contains text "Demetrius" using case sensitive ]
```

Le righe contenenti la parola *Orléans* con questo accento:

```
//LINE[ . contains text "Orléans" using diacritics sensitive]
```

Le righe contenenti la parola *odio* o parole correlate da stemming

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

Le righe contenenti la parola *amore* evitano parole di stop:

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

Le linee contenenti la parola *amore* usando i caratteri jolly:

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

- . 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

Le linee contenenti la parola *amore* usando un determinato dizionario:

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

Le righe contenenti la parola *odiano* usando la ricerca fuzzy (approssimativa):

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

La ricerca fuzzy si basa sulla distanza di Levenshtein . Il numero massimo di errori consentiti viene calcolato dividendo la lunghezza del token di un termine di query specificato per 4, mantenendo un minimo di 1 errori.

Le righe contenenti la parola *amore* almeno due volte:

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

Le righe contenenti entrambe le parole *morte* e *abiura* a una distanza di almeno due parole:

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

Altre varianti di distanza sono: al massimo, esattamente, da x a y

Le righe contenenti entrambe le parole *odiano* e *amano* nella stessa frase (o paragrafo):

```
//LINE[ . contains text "hate" ftand "love" same sentence ]
```

Le frasi sono delimitate da marcatori di fine riga (.,!,?, Ecc.) E i caratteri di nuova riga vengono considerati come delimitatori di paragrafo.

Le linee che contengono la parola *amore* ordinate in ordine decrescente di rilevanza:

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

Il punteggio di parole chiave introduce una variabile che riceve il valore del punteggio. Questo valore è garantito tra 0 e 1: un valore più alto significa un colpo più rilevante. Il calcolo della pertinenza dipende dall'implementazione. Seguono alcune linee di uscita:

```
<hit score="0.62">
  <LINE>Sweet love,--</LINE>
</hit>
<hit score="0.45">
  <LINE>my sweet love?</LINE>
</hit>
<hit score="0.45">
  <LINE>Asleep, my love?</LINE>
</hit>
<hit score="0.36">
  <LINE>Love takes the meaning in love's conference.</LINE>
</hit>
```

XQuery Update

XQuery Update Facility è una piccola estensione di XQuery per l'aggiornamento di un documento XML. Cerchiamo di lavorare sul seguente documento semplice:

```
<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>
```

Eseguiamo le seguenti operazioni di aggiornamento:

- **Elimina**

Rimuovi tutti gli elementi della professione

```
delete node //profession
```

```
<person born="23/06/1912" died="07/06/1954">
  <name>
```

```
    <first>Alan</first>
    <last>Turing</last>
  </name>
</person>
```

- **Inserire**

Inserisci un nodo professione nel nodo persona

```
insert node <profession/> into /person
```

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

Inserisci un nodo di testo nel nodo professione

```
insert node "computer scientist" into /person/profession
```

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

Inserire due nodi attributo nel nodo professione

```
insert node (attribute start {1936}, attribute end {1954}) into
/person/profession
```

```
<person born="23/06/1912" died="07/06/1954">
  <name>
    <first>Alan</first>
    <last>Turing</last>
  </name>
  <profession start="1936" end="1954">computer scientist</profession>
</person>
```

Inserisci un nodo professione come primo nodo nel nodo persona

```
insert node <profession/> as first into /person
```

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

```
  </name>
  <profession start="1936" end="1954">computer scientist</profession>
</person>
```

Uso `come last` (impostazione predefinita) per inserire il nodo come ultimo elemento in un nodo.

Inserisci un nodo professione prima del nodo del nome

```
insert node <profession/> before /person/name
```

```
<person born="23/06/1912" died="07/06/1954">
  <profession/>
  <profession/>
  <name>
    <first>Alan</first>
    <last>Turing</last>
  </name>
  <profession start="1936" end="1954">computer scientist</profession>
</person>
```

Uso `after` inserire il nodo dopo un altro nodo.

- La sequenza di aggiornamenti è incorporata nelle transazioni: vengono eseguite atomicamente (commit o rollback) e in parallelo (non in sequenza). Prova questa sequenza di due aggiornamenti (il secondo è illegale e l'intera transazione viene annullata):

  ```
  (insert node attribute nato {1936} into /person,
  insert node attribute died {1954} into /person)
  ```

  Prova questo:
  ```
  let $ att: = / person / attribute :: born return
  (inserisci nodo <born> {string ($ att)} </ born>
  come prima volta in $ to / .., elimina node $ att)
  ```

```
<person died="07/06/1954">
  <born>23/06/1912</born>
  <profession/>
  <profession/>
  <name>
    <first>Alan</first>
    <last>Turing</last>
  </name>
  <profession start="1936" end="1954">computer scientist</profession>
</person>
```

Ora prova questo:

```
let $att := /person/attribute::died return (delete node $att,
insert node <died>{string($att)}</died> as first into $att/..)
```

```
<person>
  <died>07/06/1954</died>
  <born>23/06/1912</born>
  <profession/>
  <profession/>
  <name>
    <first>Alan</first>
    <last>Turing</last>
  </name>
  <profession start="1936" end="1954">computer scientist</profession>
</person>
```

Sostituire

Sostituisci il nome del nodo con un altro nodo

```
replace node //name with <name>Alan Turing</name>
```

```
<person>
  <died>07/06/1954</died>
  <born>23/06/1912</born>
  <profession/>
  <profession/>
  <name>Alan Turing</name>
  <profession start="1936" end="1954">computer scientist</profession>
</person>
```

- Sostituisci il valore del nodo del nome con un altro valore

```
replace value of node //name with "A. Turing"
<person>
  <died>07/06/1954</died>
  <born>23/06/1912</born>
  <profession/>
  <profession/>
  <name>A. Turing</name>
  <profession start="1936" end="1954">computer scientist</profession>
</person>
```

- **Rinominare**

Rinominare il nome del nodo con un altro nome

```
rename node //name as "full-name"
```

```
<person>
  <died>07/06/1954</died>
  <born>23/06/1912</born>
  <profession/>
  <profession/>
  <full-name>A. Turing</full-name>
  <profession start="1936" end="1954">computer scientist</profession>
</person>
```