Azzolini Riccardo 2020-04-21

XPath 2.0 e XQuery

1 XPath 2.0

La versione 2.0 di XPath aggiunge numerose funzionalità che, in linea generale, permettono di esprimere computazioni su sequenze di elementi.

1.1 Elementi delle sequenze

Gli elementi di una sequenza possono essere:

- **valori atomici**: numeri, booleani, stringhe, e tipi di dato definiti in XML schema;
- **nodi** di un albero XML.

Si può effettuare una conversione da un nodo a un valore atomico, detta **atomizzazione**, che restituisce una stringa formata dai valori contenuti nel nodo e nei suoi eventuali sottoelementi.

1.2 Espressioni

XPath 2.0 prevede molti tipi diversi di espressioni:

- **Espressioni letterali**: denotano i valori atomici.
- **Variabili**, indicate con il prefisso `$` (ad esempio `$var`), che possono riferirsi a valori arbitrari (non solo atomici).
- **Espressioni aritmetiche** (ad esempio `$var + 17`).
- **Espressioni di confronto** (simili a quelle presenti nei comuni linguaggi di programmazione).
- **Espressioni di sequenza**: denota la costruzione di una sequenza formata dalla concatenazione dei valori di più espressioni, con la sintassi

  $$espr_1\texttt{,}\ espr_2\texttt{,}\ \dots\texttt{,}\ espr_n$$

  Inoltre, `()` indica la sequenza vuota, e

  $$espr_1\ \texttt{to}\ espr_2$$

denota la sequenza di numeri interi compresi tra i numeri (che devono essere anch'essi interi) risultanti dalla valutazione di $espr_1$ e $espr_2$.

- **Espressioni di percorso**: i percorsi di locazione, come in XPath 1.0.
- **Espressioni filtro**: generalizzano i predicati dei percorsi di locazione, permettendone l'applicazione a sequenze qualsiasi. La sintassi è la stessa dei predicati,

  $$espr_1\texttt{[}espr_2\texttt{]}$$

  dove $espr_1$ è un'espressione che produce una sequenza, della quale vengono selezionati ("filtrati", appunto) gli elementi per i quali l'espressione booleana $espr_2$ assume valore vero.
- **Espressioni `for`**:

  `for $`*var* `in` $espr_1$ `return` $espr_2$

  per ogni elemento della sequenza ottenuta dall'espressione $espr_1$, esso viene associato (*legato*) alla variabile `$`*var*, si valutata l'espressione $espr_2$ (che può contenere riferimenti a `$`*var*), e viene infine restituita la sequenza data dalla concatenazione dei risultati di tutte queste valutazioni.

  Ad esempio, l'espressione

  ```
  for $r in //rcp:ricetta
    return fn:count($r//rcp:ingrediente[fn:not(rcp:ingrediente)])
  ```

  restituisce il numero di ingredienti "semplici" (che non hanno sottoingredienti) di ciascuna ricetta (questo esempio è riferito al solito albero delle ricette, supponendo che gli elementi appartengano a un namespace identificato dal prefisso `rcp`).

  Questo esempio fa anche uso di due **funzioni**:

  - `fn:count` conta il numero di elementi della sequenza passata come argomento.
  - `fn:not` corrisponde all'operazione logica NOT; qui essa riceve come argomento una sequenza di elementi `rcp:ingrediente` (gli eventuali sottoingredienti), e assume valore vero se e solo se tale sequenza è vuota.
- **Espressioni `if`**:

  `if (`$espr_1$`) then` $espr_2$ `else` $espr_3$

  in base al valore dell'espressione booleana $espr_1$, restituisce il valore dell'espressione $espr_2$ oppure di $espr_3$.
- **Espressioni quantificate**: determinano se alcuni/tutti gli elementi di una sequenza soddisfano una determinata condizione. Ne esistono due tipi:

– `some $`*var* `in` $espr_1$ `satisfies` $espr_2$
  è vera se e solo se *almeno uno* degli elementi della sequenza $espr_1$ soddisfa l'espressione booleana $espr_2$. Un'espressione equivalente può essere scritta usando `for`, `if`, e alcune funzioni:

  ```
  fn:exists(
    for $var in espr1
      if (espr2) then fn:true() else ()
  )
  ```

– `every $`*var* `in` $espr_1$ `satisfies` $espr_2$
  assume valore vero se e solo se *tutti* gli elementi di $espr_1$ soddisfano $espr_2$. Anche in questo caso esiste un'espressione equivalente:

  ```
  fn:empty(
    for $var in espr1
      if (espr2) then () else fn:false()
  )
  ```

  (perché controllare che *non* esistano elementi della sequenza che *non* verificano la condizione è equivalente a controllare che tutti gli elementi la verifichino).

Riassumendo, quindi, XPath 2.0 introduce la possibilità di definire computazioni sui dati e sugli elementi di un albero XML, utilizzando strutture di controllo analoghe a quelle della programmazione imperativa.

2 XQuery

Nonostante le aggiunte della versione 2.0, il linguaggio XPath non ha alcune importanti funzionalità:

- la capacità di estrarre, confrontare ed elaborare sottoalberi di alberi XML diversi;
- la capacità di creare un nuovo albero XML, contenenti il risultato di un'interrogazione, cioè di effettuare una **ristrutturazione del risultato**.

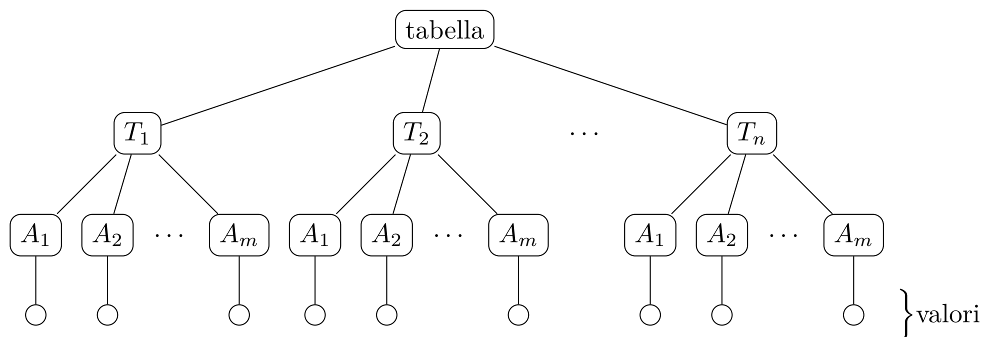
Il linguaggio di interrogazione **XQuery**, basato su XPath 2.0, assolve a questi (e altri) compiti.

2.1 Corrispondenza tra relazioni e alberi

Per rendere più intuitive le interrogazioni XQuery, è utile pensare a un albero XML come a una generalizzazione di una tabella relazionale. Infatti, una generica tabella

A_1	A_2	...	A_m	
				T_1
				T_2
				$\vdots$
				T_n

è equivalente a un albero come il seguente,



nel quale la radice corrisponde alla tabella stessa, i figli della radice corrispondono alle tuple, e ciascuno di essi ha, a sua volta, un figlio per ciascun attributo della tupla.

Viceversa, non tutti gli alberi corrispondono direttamente a tabelle (perché, come già detto, si tratta di un modello semi-strutturato), ma l'analogia è comunque utile a livello intuitivo.

2.2 Espressioni XML

Una delle funzionalità aggiuntive di XQuery (rispetto a XPath), che costituisce la base delle capacità di ristrutturazione del risultato, è la possibilità di creare nuovi nodi XML.

In generale, per fare ciò si usa direttamente la sintassi di XML, ma in più, nel contenuto di un elemento (o nel valore di un attributo), è possibile specificare delle espressioni XQuery, racchiuse tra parentesi graffe: il risultato della valutazione di tali espressioni verrà inserito nel contenuto, al posto delle espressioni stesse.

Ad esempio, le seguenti espressioni sono tutte equivalenti:

```
<numbers>1 2 3 4 5</numbers>

<numbers>{1, 2, 3, 4, 5}</numbers>

<numbers>{1 to 5}</numbers>

<numbers>1 {1 + 1} {3} {4 to 5}</numbers>
```

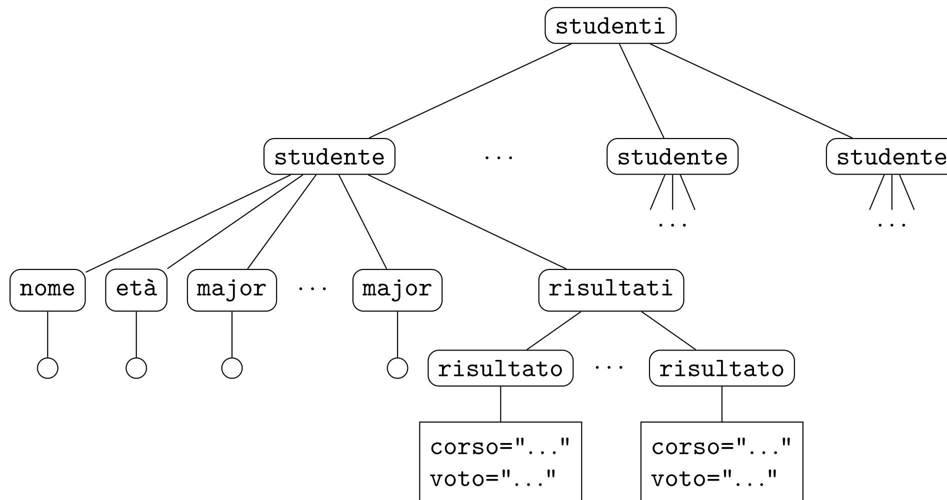
2.3 Espressioni FLWOR

La principale funzionalità messa a disposizione da XQuery per le interrogazioni sono le **espressioni FLWOR**. FLWOR (pronunciato “flower”) è l’abbreviazione di *For, Let, Where, Order by, Return*, i nomi delle clausole che compongono l’espressione.

1. La clausola `for` esegue un’iterazione su una sequenza di elementi del documento (tipicamente individuati mediante un’interrogazione XPath), legando (assegnando) ciascuno di essi, a turno, a una variabile. Essa è del tutto analoga all’espressione `for` già vista nel contesto di XPath (della quale le espressioni FLWOR possono infatti essere considerate una generalizzazione).
2. La clausola `let` permette di dichiarare altre variabili, legando a esse dei valori che possono essere calcolati anche in base all’elemento corrente dell’iterazione del `for`, eventualmente mediante interrogazioni XPath.
3. La sequenza di legami alle variabili generata da `for` e `let` viene filtrata da un’espressione booleana contenuta nella clausola `where`.
4. La sequenza viene ordinata mediante la clausola `order by`.
5. Infine, il risultato viene calcolato valutando, per ciascuno dei valori legati alle variabili, l’espressione specificata nella clausola `return`.

2.3.1 Esempio

Questo albero XML rappresenta dati relativi a degli studenti universitari:



Si suppone che la serializzazione dell'albero sia contenuta in un file `studenti.xml`. Allora, la seguente interrogazione XQuery estrae tutti gli studenti con un doppio "major", cioè quelli che hanno almeno due orientamenti di studio principali ("major"), e produce, come risultato, una sequenza di elementi `double`, contenenti i nomi di tali studenti:

```
for $s in fn:doc("studenti.xml")//studente
let $m := $s/major
where fn:count($m) ge 2
order by $s/@id
return <double>{$s/name/text()}</double>
```

1. La clausola `for` itera su tutti gli elementi `studente` presenti nel file `studenti.xml`. Infatti, la funzione predefinita `fn:doc` accede alla radice dell'albero contenuto nel file individuato dall'URI passato come argomento, e il resto del percorso di locazione sfrutta l'asse `descentant-or-self` (abbreviato a `//`) per individuare tutti gli elementi `studente` tra i discendenti della radice.

   Così, a ogni iterazione, si genera un nuovo legame della variabile `$s` a uno degli elementi `studente`.

2. Nella clausola `let` si usa un altro percorso di locazione per estrarre la sequenza di tutti gli elementi `major` figli dello studente corrente (`$s`), e l'intera sequenza viene legata alla variabile `$m`.

   A questo punto, allora, per ogni iterazione del `for` si ottiene una coppia di legami alle variabili `$s` e `$m`: il valore legato a `$s` è un singolo elemento `studente`, mentre quello legato a `$m` è la sequenza dei suoi `major`.

3. La clausola `where` filtra le coppie di legami (`$s`, `$m`), selezionando solo quelle tali che la sequenza di `major` legata a `$m` contenga almeno 2 elementi (ovvero le coppie corrispondenti agli studenti con almeno due major).

La condizione specificata in questa clausola è un'espressione XPath scritta con la funzione `fn:count`, introdotta in precedenza, e l'operatore di confronto `ge` (*greater or equal*, maggiore o uguale).

4. Con la clausola `order by`, le coppie (`$s`, `$m`) rimaste dopo il `where` vengono ordinate in base al valore dell'attributo `id` dell'elemento `studente` legato a `$s`.

5. Infine, per ogni coppia (`$s`, `$m`), viene costruito un elemento `double`, contenente il valore testuale del nodo `nome` figlio del nodo `studente` legato a `$s`.

Come già detto, il risultato di quest'interrogazione è una sequenza di elementi `double`, ma una sequenza di elementi *non* è un albero XML. Se si vuole ottenere un albero, è sufficiente racchiudere tale sequenza in un altro elemento, chiamato ad esempio `doubles`:

```
<doubles>{
  for $s in fn:doc("studenti.xml")//studente
  let $m := $s/major
  where fn:count($m) ge 2
  order by $s/@id
  return <double>{$s/name/text()}</double>
}</doubles>
```

Si ottiene così un albero con la seguente struttura: