1 INTRODUZIONE

L'oggetto di questa tesi è la realizzazione di un preprocessore per la validazione di documenti XML utilizzando il linguaggio DTD++.

La maggior parte dei dati informatici degli ultimi 40 anni è andata persa [Har99]. Questo non per disastri naturali o decadimento delle unità di backup, ma semplicemente per il fatto che i programmi odierni non sono in grado di interpretare i dati del passato. Tale problematica non si pone generalmente per i principali software commercial,i che sono spesso ben supportati dalle case di produzione, grazie a filtri che permettono l'importazione delle vecchie versioni dei documenti. Tuttavia, ciò comporta il doversi sempre affidare ad un dato programma. Il problema nasce, invece, nel momento in cui si desidera cambiare completamente il modello del software.Per fare un esempio, se si desiderasse passare da un software per il cad prodotto da una software house che non opera più sul mercato ad una delle ultime release di Autocad, si rischierebbe di spendere ingenti quantità di tempo e di denaro per trasformare i vecchi documenti, situazione quest'ultima non sempre possibile.

Negli ultimi anni la larga diffusione del WEB ha ulteriormente accentuato questo problema. Milioni di pagine HTML popolano la rete contenendo informazioni sempre meno strutturate.

L'esplosione di questo fenomeno è dovuta principalmente alla grande semplicità che ha permesso praticamente a chiunque di realizzare siti Internet.

La mancanza di regole precise per l'utilizzo, e l'assenza nei browser di un meccanismo di convalida, ha portato però il web ad uno stato di anarchia sintattica e semantica dei documenti. I tag, inizialmente pensati con valenza strutturale, sono stati utilizzati per dare una rappresentazione grafica: la differenza si è così lentamente persa.

Ciò è rimasto completamente invisibile per l'utente finale, ma una tale crescita del web ha reso necessaria anche una crescita dell'automazione. I motori di ricerca, per esempio, hanno un compito sempre più arduo nel reperire le corrette informazioni in una tale mole di dati.
Una divisione fra struttura e rappresentazione consente, invece, di risolvere questo problema. Separando le cose non ci si deve più porre il problema di perdere il significato dei documenti, in quanto esso è ben definito altrove. Ciò permette quindi una scelta di ciò che si vuole reperire in base all'utenza. Il motore di ricerca si limiterà quindi a esaminare la parte relativa alla struttura, scartando tutto ciò che è relativo alla rappresentazione grafica.
I limiti del web e i problemi relativi al trasferimento dei dati tra applicazioni diverse potranno essere risolti grazie all'introduzione di linguaggi diversi. XML (eXtensible Markup Language) viene proposto dal W3C (World Wide Web Consortium) [BPSM98][BPSM00] come uno di questi linguaggi. XML permette di gestire i dati in maniera più efficiente, separandoli su più file e dividendoli dalla loro definizione di struttura e dalla loro eventuale rappresentazione. In questo modo l'informazione può essere gestita con la massima interscambiabilità. È possibile creare il file dei dati, validarlo con la relativa struttura ed eventualmente visualizzarlo utilizzando il file relativo alla rappresentazione grafica.
La diffusione di questi nuovi linguaggi costituisce un primo passo nella risoluzione delle problematiche attuali della rete e in generale dell'interscambio dei dati.

XML è un insieme di regole per la definizione semantica di tag i quali suddividono e identificano le diverse parti di un documento. La prima cosa da capire è che XML non è un altro linguaggio di markup come HTML. I linguaggi di markup, infatti, definiscono un fissato insieme di tag che descrivono un fissato numero di elementi, mentre XML è un

meta-markup-linguaggio, cioè, un linguaggio col quale si possono creare i tag di cui si ha bisogno.
Questi tag devono essere organizzati seguendo certi principi generali, ma sono comunque molto flessibili. Per esempio, se si dovesse lavorare sull'archiviazione di un campionato di calcio, si potrebbero creare tag riguardanti i dati anagrafici dei giocatori, delle squadre, i calendari delle partite e tutto quello di cui si ha bisogno per l'applicazione.

L'utilizzo di XML porta sicuramente ad avere dei vantaggi:

Design dei markup language specifici per dominio
XML permette ai professionisti di crearsi un markup language per le proprie esigenze. Ciò permette di inviare dati, senza domandarsi se il ricevente possieda o meno il software necessario per leggerli.

Dati autodescrittivi
XML è un formato dati incredibilmente semplice. Può essere scritto interamente in ASCII: questo, oltre a permetterne la lettura immediata da parte dell'utente con un semplice programma di visualizzazione testi, rende il file leggibile anche in caso di corruzione parziale.
Inoltre, i tag utilizzati, se ben fatti, sono autodescrittivi. Sarà quindi possibile interpretare il contenuto del documento senza l'utilizzo di alcun programma.

Interscambio dati
Usando XML, al posto di un formato dati proprietario, è possibile utilizzare qualsiasi tool che capisca XML per lavorare con i propri dati.

Convalida della struttura del documento.
XML permette di forzare la convalida del documento imponendo dei vincoli sulla presenza, o meno, degli elementi di markup e sui valori che

essi possono avere. Ciò avviene grazie all'utilizzo di particolari forme di dichiarazione che rispettano determinate forme sintattiche tipo DTD (Document Type Definition) e XML Schema. DTD esprime vincoli in una sintassi tipica dei DTD SGML, mentre XML Schema si attiene ad una sintassi che rispetta quella XML. Una volta specificate le regole è possibile, tramite tool specifici, determinare se un dato documento le rispetti.

Tra i vantaggi citati sull'utilizzo di XML, risalta particolarmente quello relativo alla possibilità di convalidare il documento. XML fornisce due differenti nozioni di correttezza, documenti ben formati e documenti validi.

Avere documenti ben formati significa avere documenti comprensibili, che seguono una serie di regole strutturali. Queste regole valgono per tutti i documenti XML e non necessitano di essere specificate, perché parte integrante del linguaggio:

1. I tag devono essere nidificati correttamente. Ogni coppia di tag (di apertura e di chiusura) deve contenere qualunque altra coppia di tag che inizia al proprio interno.

2. Il nome del tag di chiusura di un elemento deve corrispondere al relativo tag di apertura.

3. Nessun nome di attributo può apparire più di una volta all'interno dello stesso tag di apertura o in quello relativo ad un elemento vuoto.

4. I valori degli attributi non possono contenere riferimenti, diretti o indiretti, ad entità esterne.

5. Il testo sostitutivo di un’entità riferita nel valore di un attributo non deve contenere il carattere “<”.

6. La dichiarazione di un’entità parametrica deve precedere qualunque riferimento ad essa.

7. La dichiarazione di un’entità deve precedere qualunque riferimento ad essa che appaia sotto forma di valore predefinito all’interno della dichiarazione di una lista di attributi. In altre parole non è possibile mischiare i dati binari in mezzo al testo senza un meccanismo di gestione dichiarato.

8. I riferimenti alle entità parametriche possono apparire soltanto nella descrizione della struttura del documento. Ciò significa che non è possibile riportare istruzioni di elaborazione nell’istanza del documento, aspettandosi che queste assumano un qualche significato.

La nozione di documento valido è invece limitata ad ogni singolo documento. Un documento si definisce valido quando sottostà a tutte le regole contenute nell’insieme che ne definisce la struttura. Per verificare queste regole esistono tool di analisi automatica chiamati validatori, che leggono il documento e ne verificano la corrispondenza con la sua descrizione strutturale.
Attraverso l’uso di un DTD (Document Type Definition) o di un XML Schema è possibile definire le regole di una determinata classe di documenti. Vengono specificati gli elementi che devono o meno esistere e dove si trovano.
Se è importante che i documenti siano strutturato in un certo modo conviene scrivere un DTD o un XML schema che ne verifichino la correttezza. Questo darà coerenza ai documenti. Rendere pubblico il

DTD permetterà a chiunque di scrivere strutturalmente corretti pronti per essere inseriti nel sistema informativo. Infatti, grazie ai validatori è possibile, per l'utente, controllare il documento prima di renderlo pubblico.

DTD e XML Schema sono i due linguaggi forniti dal W3C per la definizione delle regole e delle strutture dei documenti. La differenza tra loro sta nella semplicità d'uso e nella potenza espressiva.

DTD è molto semplice ma poco espressivo, non è capace di dare forti restrizioni e porre vincoli precisi. In DTD, infatti, non esiste un vero e proprio concetto di tipo, applicabile alle strutture del documento. Non è possibile derivazione dei tipi ne di dichiaratre dei namespace a cui si possano associare le dichiarazioni degli elementi.

XML Schema permette di porre forti vincoli alle definizioni, possiede una gestione dei tipi molto complessa, che permette di definirli con precisione, ma non supporta i concetti di entità parametrica e generale. Inoltre, scrivere un documento XML schema risulta un operazione molto complessa.

A partire dal 1998 ad oggi sono stati proposti all'incirca una dozzina di linguaggi di definizione dei tipi, tra cui: SOX, XDR, Schematron, DSD e Relax.

Schema for Object-Oriented XML 2.0 (SOX)[DFM+99] è un linguaggio di descrizione alternativa allo Schema, pensato per la definizione delle strutture sintattiche e di parte delle strutture semantiche. SOX estende DTD in forma orientata agli oggetti consentendo l'uso dei tipi di dati estensibili, l'erediarietà fra i tipi degli elementi e supporta le dichiarazioni di namespace.

XDR (XML-Data Reduced)[FrTh98] era precedentemente conosciuto come XML-Data[LJM+98] ed è nato da una collaborazione di Microsoft con altri. XDR è usato nel framework Microsoft BizTalk. È notevolmente influenzato da un'altra proposta sviluppata da Microsoft e IBM cn la quale condivide molte delle funzionalità.

Schematron[Jel] è un linguaggio che si distingue da tutti gli altri in quanto orientato alla validazione degli schemi mediante *patter* piuttosto che definizione degli schemi stessi. La cosa innovativa è la possibilità di poter creare degli schemi relativamente brevi e di specificare potenti vincoli tramite l'uso di XPath.
La collaborazione dei laboratori AT&T e BRICS da alla luce un nuovo linguaggio in grado di descrivere elementi e attributi dipendentemente dal contesto, offrendo al tempo stesso meccanismi flessibili di inserimento dati. La potenza espressiva di questo linguaggio, chiamato DSD[RaB], fornisce un potente meccanismo per l'introduzione dei vincoli, così come avviene per Schematron.
In ultimo RELAX (REgular LAnguage description for XML)[JI] rappresenta una tappa fondamentale nella migrazione dal DTD all'XML Schema ed offre tutte le caratteristiche tipiche del DTD.

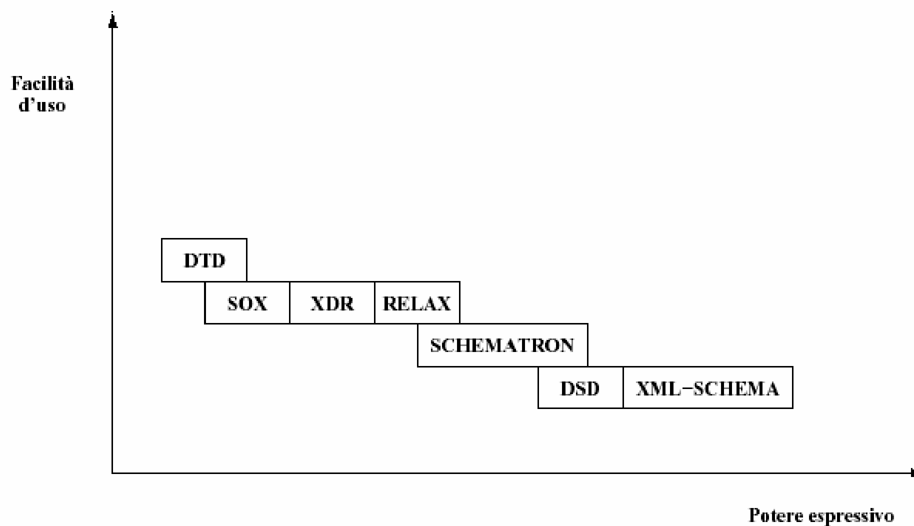

Figura 1.1 Potere espressivo e facilità d'usodei linguaggi di definizione dei tipi [Amo02]

Il grafico in figura 1.1 mostra come l'evolversi di questi linguaggi cerchi di colmare il divario che c'è tra XML Schema e DTD.

DTD può essere comunque considerato un buon compromesso tra espressività e semplicità. DTD è essenziale, compatto, semplice da imparare e interpretare direttamente a occhio nudo, senza utilizzare tool appositi.
Seguendo questa linea, presso la facoltà di Informatica dell'Università di Bologna, è stata sviluppata un'estensione sintattica al linguaggio DTD che ne aumenta le funzionalità, potenziandolo e rendendolo equivalente ad un sottoinsieme significativo di XML Schema: DTD++ [VAG03].
Tramite questo linguaggio è quindi possibile definire documenti molto complessi e porre dei vincoli ben precisi sui tipi dei documenti, senza rinunciare ai pregi di DTD rispetto a XML schema: la semplicità d'uso e la possibilità di definire entità parametriche.

Gli obbiettivi di DTD++ sono:

- Creare un linguaggio che sia un'estensione diretta di DTD. Questo significa che:
  - Ogni DTD è anche un DTD++ valido;
  - Ogni DTD++ può essere privato tipograficamente della sintassi DTD++ per ottenere un DTD valido, utilizzabile per validare gli stessi tipi di documenti;
  - Le estensioni alla sintassi del DTD sono state ridotte al minimo.
- Utilizzare XML Schema come fonte delle caratteristiche aggiuntive. Questo significa che:
  - Il maggior numero di funzionalità presenti in Schema è direttamente disponibile in DTD++;
  - Ogni DTD++ valido è semanticamente equivalente ad un XML Schema valido;
  - La validazione di documenti XML per mezzo di DTD++ è identica alla validazione con XML Schema, a meno dei messaggi di errore.

In DTD++ sono quindi stati introdotti alcuni costrutti al fine di realizzare i suddetti obbiettivi.

Tipi semplici

I tipi semplici corrispondono ai sympleType di XML Schema. Un tipo semplice può contenere solamente dati di tipo carattere.

DTD++ implementa tutti tipi predefiniti di XML Schema. Ad ogni tipo è stato dato un nome simile al rispettivo Schema: #STRING, #INTEGER, #FLOAT e così via.

Alla sintassi del costrutto ENTITY sono state apportate delle modifiche per permettere all'utente di definire dei tipi semplici derivando quelli predefiniti. È stato aggiunto un carattere "#" tra "<!ENTITY" e il nome dell'entità.

I tipi semplici definiti dall'utente possono essere anonimi, utilizzati direttamente nelle definizioni degli elementi e degli attributi, o nominati, con una definizione separata ed un riferimento al nome dagli elementi e dagli attributi.

La derivazione può avvenire per restrizione, per lista e per unione.

Un tipo semplice è derivato per restrizione da un tipo predefinito, o da un altro tipo semplice. A ciò si aggiungono ulteriori costrutti limitativi chiamati 'facet'.

XML Schema fornisce un numero di facet, alcuni dei quali sono validi per tutti i tipi predefiniti, altri solo per alcuni. Tra questi ci sono length, minlength, minInclusive, maxExclusive, totalDigits, pattern, ecc.

Il tipo semplice lista è una collezione di valori, dello stesso tipo, separati da spazi bianchi. Il tipo union, invece, è derivato da due o più tipi semplici. Un tipo union valido deve essere valido per uno dei tipi che lo definiscono.

Tipi complessi
I tipi complessi corrispondono ai complexType di XML Schema.
Un tipo complesso può contenere sia elementi che attributi, oltre ai dati di tipo carattere.
Come i tipi semplici, i tipi complessi possono essere sia anonimi che nominati. I tipi anonimi sono equivalenti ai costrutti degli ELEMENT e delle ATTLIST. I tipi complessi nominati, invece, utilizzano un'ulteriore variante del costrutto ENTITY del DTD. E' stato aggiunto un carattere "@" tra "<!ENTITY" e il nome dell'entità.
Al content model di tipo *choice* e *sequence* è stato aggiunto quello di tipo *all* e ampliata la definizione del content model di tipo *any*.
XML Schema fornisce un controllo molto fine per la ripetitività e l'obbligatorietà degli elementi rispetto a DTD, consentendo specificazioni numeriche del minimo e del massimo per ogni struttura. Anche DTD++ fornisce un costrutto per specificare queste restrizioni.
Anche i tipi complessi, come quelli semplici, possono essere derivati. La derivazione avviene per restrizione o per restrizione.

Namespace
Alla sintassi originale del DTD è stato aggiunto un nuovo tag "TARGETNS" che definisce il namespace del DTD++. Il target name space può essere anche definito con un prefisso. Ciò permette di definire più namespace all'interno dello stesso DTD++.

Pregi e vantaggi di DTD++
L'utilizzo di DTD++ permette di avere alcuni evidenti vantaggi:

- La similitudine con DTD permette un facile apprendimento del linguaggio, con una conseguente riduzione dei costi per l'aggiornamento del personale.

- La sintassi concisa aiuta a diminuire i tempi di realizzazione dei documenti di validazione.

- L'espressività del linguaggio permette comunque di creare dei vincoli molto forti sui documenti da validare.

La definizione di un nuovo linguaggio non offre, però, una soluzione completa al problema della validazione dei documenti. L'approcciare nuove tecnologie è sempre una scelta che va ben ponderata. Bisogna considerare i costi che comporta, siano essi economici o in utilizzo delle risorse umane. Inoltre, la scelta fatta potrebbe essere per una tecnologia destinata all'insuccesso. Mesi utilizzati per la formazione del personale, o ancor peggio progetti sviluppati utilizzando quella tecnologia andrebbero persi. Sia DTD che XML Schema sono affermati come standard e decidere per linguaggi alternativi potrebbe risultare una scelta sbagliata.
L'adozione di tecnologie non standard creerebbe problemi di condivisione dei documenti con strutture esterne e impedirebbe l'utilizzo di una vasta gamma di applicazioni ben testate e funzionanti.

L'obiettivo è quello di realizzare un software che, sfruttando la semplicità di DTD++, ne permetta l'utilizzo limitandone i rischi.

La soluzione sarebbe quella di un preprocessore per la validazione di documenti XML con documenti DTD++.
Il preprocessore deve:

- processare il documento DTD++ realizzandone una versione XML Schema equivalente;

- apportare le modifiche al documento XML inserendo i riferimenti allo Schema generato;

- richiamare un validatore Schema esterno affinché validi il documento XML con lo schema generato;

- intercettare gli errori restituiti dal validatore e riportarli all'utente con i riferimenti al documento DTD++.

Il preprocessore non deve effettuare controlli sulla validità dello Schema, ma delegare interamente il compito al validatore esterno.

Pregi e vantaggi

Ai vantaggi guadagnati dall'adozione di DTD++ si aggiungono quelli forniti dall'utilizzo del preprocessore.

- Apportando semplici modifiche al preprocessore è possibile utilizzare qualsiasi software che preveda un documento XML Schema come file di input.

- La possibilità di trasformare il documento DTD++ in XML Schema lascia sempre una porta aperta verso lo standard, offrendo la possibilità di scambiare documenti con terzi e di abbandonare l'utilizzo di DTD++ in qualsiasi momento.

- Il processo di validazione viene delegato a software già largamente utilizzati e testati.

La realizzazione di questo tool permetterà quindi di utilizzare DTD++ ad un livello produttivo fornendo un sistema di validazione semplice e allo stesso tempo molto potente.

La tesi è così strutturata:

- **Capitolo 2 - DTD**
  Viene introdotto il DTD. Si è cercato di fare un riassunto completo dei costrutti e delle strutture che è possibile definire utilizzando questa sintassi. Vengono riportati anche alcuni esempi sull'utilizzo del linguaggio.

- **Capitolo 3 - XML Schema**
  Viene fatta una panoramica sul linguaggio XML Schema. Come nel capitolo precedente, si è cercato di fare un riassunto completo di tutti i costrutti e delle strutture che si possono definire con la sua sintassi. Vengono riportati anche alcuni esempi sull'utilizzo del linguaggio.

- **Capitolo 4 - DTD++**
  Vengono introdotte ed esaminate nel dettaglio tutte le innovazioni sintattiche proposte da DTD++. Vengono mostrati i costrutti per la definizione dei tipi semplici e complessi, l'utilizzo dei facet, la definizione dei content model e la dichiarazione dei namespace.

- **Capitolo 5 - PreValidator for DTD++**
  Viene presentato il preprocessore partendo dall'esame dei motivi che ne hanno richiesto la realizzazione. Successivamente viene fatta una panoramica sul funzionamento, con alcune considerazioni sui pregi e i difetti. Si procede poi con una breve

introduzione di DTD++ for Xerces, un applicazione che utilizza Xerces per validare i documenti con DTD++ e si confrontano le 2 architetture. Viene inoltre presentata la versione DTD++ di MathML come esempio pratico delle potenzialità di DTD++. Il capitolo si conclude con gli ipotetici sviluppi futuri del preprocessore e del linguaggio DTD++.

- **Capitolo 6 - PreValidator for DTD++: dettagli implementativi.**
  Vengono affrontati in questo capitolo i dettagli implementativi. Vengono descritte le classi e il flusso del programma sono elencate le teconologie utilizzate, e riportate le istruzioni per l'uso e l'installazione.

- **Capitolo 7 - Conclusioni**
  Vengono riportate le osservazioni conclusive di questo progetto.

2 DTD

Il *Document Type Definition* (DTD) specifica la grammatica di un documento XML, definendone la struttura fisica e logica, attraverso *dichiarazioni di markup* (markup declaration). Queste dichiarazioni fissano i vincoli di validità per una classe di documenti. In pratica, un DTD stabilisce quali elementi possono comparire in una classe di documenti e in che ordine; inoltre formalizza la lista dei possibili attributi per ciascun elemento.
Sono quattro i tipi di dichiarazioni di markup che possono comparire in un DTD: dichiarazione del tipo di elemento, dichiarazione della lista degli attributi, dichiarazione di entità, dichiarazione di notazione.
In un DTD, un documento complesso viene ridefinito mediante la scomposizione in parti più semplici, secondo il paradigma del *divide et impera.* Inizialmente, si specifica la composizione dell'elemento gerarchicamente superiore, quello che a sua volta contiene tutti i sotto-elementi. Di seguito vengono formalizzati gli elementi gerarchicamente inferiori, uno per uno, in maniera ricorsiva, fino ad esaurire la complessità del documento.
Il DTD non compare obbligatoriamente in un documento XML, ma è necessario per stabilire la rispondenza formale di un documento alle caratteristiche di una determinata classe. Infatti, solo dal raffronto di un DTD con la struttura di un documento è possibile stabilirne la validità.

Seguendo le specifiche di un DTD è possibile creare documenti appartenenti ad una determinata classe, con la certezza che tutte le istanze così create godano delle stesse proprietà.

Un esempio classico è quello della descrizione di una scheda anagrafica:

```
<! DOCTYPE persona [
```

```
<!ELEMENT persona (nominativo, indirizzo, nascita)>
<!ELEMENT nominativo (nome,cognome)>
<!ELEMENT indirizzo (via,comune,provincia,cap)>
<!ELEMENT nascita (data,comune, provincia)>
<!ELEMENT nome (#CDATA)>
<!ELEMENT cognome (#CDATA)>
<!ELEMENT via (#CDATA)>
<!ELEMENT comune (#CDATA)>
<!ELEMENT cap (#CDATA)>
<!ELEMENT provincia (#CDATA)>
<!ELEMENT data (#CDATA)>
]>
```

A partire da questo DTD si può realizzare questa istanza di documento XML valido:

```
<persona>
      <nominativo>
            <nome>Davide</nome>
            <cognome>Fiorello</cognome>
      </nominativo>
      <indirizzo>
            <via></via>
            <comune></comune>
            <provincia></provincia>
            <cap></cap>
      </indirizzo>
      <nascita>
            <data></data>
            <comune></comune>
            <provincia></provincia>
      </nascita>
</persona>
```

2.1 Doctype

Il primo elemento che può comparire in un documento XML è il prologo, a sua volta composto di due elementi opzionali: la dichiarazione XML e la dichiarazione del tipo di documento.

La dichiarazione XML fornisce tre tipi di informazione: specifica la versione di XML utilizzata, la codifica dei caratteri e se si tratta di un documento standalone o meno.

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

La dichiarazione del tipo di documento, invece, specifica il DTD di riferimento per il documento attuale. Il DTD può essere specificato in due modi: internamente oppure esternamente al documento. Nel primo caso, il document type declaration avrà questa forma:

```
<!DOCTYPE RootName [
<!ELEMENT RootName()
]>
```

Nel secondo caso, invece la dtd sarà contenuta in un documento esterno, del quale la document type declaration fornisce l'uri:

```
<!DOCTYPE RootName SYSTEM "file.dtd">
```

È possibile anche una terza via, che unisce le due precedenti. Si può in sostanza fornire una dtd mista, che unisca le definizioni interne e quelle esterne in un unico spazio.

2.2 Elementi del DTD

Gli elementi di un DTD vengono definiti allo stesso modo, indipendentemente dal fatto che si tratti di un DTD interno o esterno. La sintassi di base per definire un elemento è la seguente:

```
<!ELEMENT ElementName (Content model)>
```

Gli aspetti notevoli sono due: il primo consiste nell'unicità di ciascun elemento, che può essere definito una sola volta. Il secondo è invece il Content Model, attraverso il quale vengono formalizzati gli elementi

che possono comparire all'interno dell'elemento ElementName. Il Content Model stabilisce non solo cosa è permesso all'interno dell'elemento, ma anche la posizione e il numero. A livello sintattico, il Content Model si trova racchiuso tra parentesi tonde e fornisce un elenco dei possibili modelli di contenuto, separati da caratteri speciali. Questi caratteri fungono da separatori e assumono significati ben precisi, determinando ordine, obbligatorietà e ripetizione degli elementi di un content model.
Possono comparire all'interno del content model elementi di rango gerarchico inferiore, che vengono definiti ricorsivamente più avanti nel DTD, oppure la specifica particolare #PCDATA (che introduce dati nel formato di caratteri parsati, *parsed characer data*).
I separatori disponibili sono due:

- (a,b): a e b devono comparire obbligatoriamente, nell'ordine indicato;

- (a|b): può comparire solamente uno dei due elementi specificati.

La ripetibilità degli elementi viene espressa attraverso un altro set di caratteri speciali:

- nessun operatore di ripetizione: l'elemento deve comparire esattamente una volta;

- ?: l'elemento può comparire una volta sola o essere assente;

- +: l'elemento deve comparire almeno una volta;

- *: l'elemento può comparire zero o più volte.

Esistono poi due particolari tipologie di content model, ANY e EMPTY, dichiarati diversamente già a livello sintattico:

```
<!ELEMENT ElementNameAny ANY>
<!ELEMENT ElementNameEmpty EMPTY>
```

Il primo ammette qualsiasi modello di contenuto, mentre il secondo richiede un elemento vuoto. Nel documento XML, l'elemento vuoto si tradurrà in un tag di inizio, privo di quello di chiusura. Entrambe le tipologie speciali di Content Model vengono dichiarate senza essere racchiuse tra parentesi tonde.
Un'altra situazione particolare è quella in cui compaiano content model misti, formati cioè da *character data* mescolati ad altri elementi. In questi casi si rende necessario elencare la specifica #PCDATA prima di ogni altro elemento.

2.3 Gli attributi

Il DTD consente di specificare dichiarazioni di liste di attributi (*attribute list declarations*) per gli elementi. Queste dichiarazioni contengono il nome, il tipo di dato ed eventualmente il valore di default per ciascun attributo associato ad un elemento specifico (http://www.w3.org/TR/2004/REC-xml-20040204/#attdecls). Questi attributi possono essere definiti nel tag di inizio oppure in tag vuoti. La definizione di una lista di attributi per un determinato elemento ha la seguente sintassi:

```
<!ATTLIST ElementName
AttributeName-1 Type-1 default-1
AttributeName-2 Type-2 default-2
...
>
```

Nel dettaglio, i documenti XML ammettono tre tipi di attributo: stringa, tokenized ed enumerati.

Nel primo caso (attributi di tipo stringa), si dichiara il tipo CDATA, attraverso il quale si ammette qualsiasi valore letterale.
Nel secondo caso (*tokenized*), invece, presenta una serie di casi, ognuno con vincoli specifici:

- ID: l' identificativo unico per quell'elemento. Ciò significa che il nome assegnato non può essere ripetuto e che ogni elemento può avere un solo attributo di tipo ID. Non solo: gli attributi di questo tipo deve avere un valore di default, dichiarato come #IMPLIED o #REQUIRED.

- IDREF: è un riferimento ad un elemento identificato univocamente da un attributo di tipo ID. Il valore di IDREFS deve essere uguale a quello dell'attributo ID dell'elemento cui si riferisce.

- IDREF: una lista di IDREF separati da spazi;

- ENTITY: è il nome di un'entità *unparsed* dichiarata nel documento;

- ENTITIES: è una lista di ENTITY separate da spazi;

- NMTOKEN: una stringa composta di una sola parola;

- NMTOKENS: una lista di NMTOKEN separati da uno spazio bianco

Il terzo caso è quello degli attributi enumerati, che possono presentarsi sotto due diverse forme. La prima è quella di un elenco di attributi:

```
<!ATTLIST ElementName
AttributeName-1 (value-1 | value-2 | value-3) default-1
>
```

La seconda è un riferimento ad una notazione (*notation*), definita nella dtd. Le notazioni vengono definite nella dtd con questa sintassi:

```
<!NOTATION Name SYSTEM "value">
```

Il vincolo di validità per questo genere di attributi è proprio la dichiarazione della notazione di riferimento all'interno della dtd, oltre al fatto che un attributo di questo tipo può far riferimento ad una sola notazione.

Attraverso la Attribute List Declaration è possibile sapere se un elemento avrà o meno attributi, e quale comportamento dovrà tenere il processore XML nell'incontrarli.
Le possibilità sono 4:

- Valore di default #REQUIRED: in questo caso è sempre necessario specificare il valore dell'attributo;

  ```
  <!ATTLIST termdef id ID #REQUIRED>
  ```

- Valore di default #IMPLIED: il valore dell'attributo non deve essere espresso obbligatoriamente, nel caso in cui non venga espresso non verrà assegnato alcun valore di default;

  ```
  <!ATTLIST termdef name CDATA #IMPLIED>
  ```

- Valore di default #FIXED “value”: il valore dell’attributo non deve essere espresso obbligatoriamente. Se però viene espresso può essere unicamente il valore specificato;

```
<!ATTLIST form method CDATA #FIXED "POST">
```

- Valore di default letterale: l’attributo assume il valore espresso. Se il valore viene omesso il processore XML si comporta come se il valore fosse uguale a quello di default.

```
<!ATTLIST list type (bullets|ordered|glossary)
"ordered">
```

2.4 Sezioni condizionali

Il DTD può contenere sezioni condizionali, ovvero sezioni che vengono ignorate o meno a seconda della parola chiave con cui sono chiamate.

```
<![INCLUDE[
      …
      …
]]>
```

Ciò che compare all’interno di una sezione INCLUDE è considerato come parte del DTD a tutti gli effetti.
Al contrario, le sezioni IGNORE sono trattate come non facenti parte del DTD.

```
<![IGNORE[
      …
      …
]]>
```

2.5 Le entità

Le entità sono le strutture fisiche di cui consiste un documento XML: unità di memorizzazione, identificate univocamente da un nome e provviste di un contenuto. Possono essere *parsed* o *unparsed.* Per quanto riguarda le prime, nel momento in cui il documento viene processato, il loro contenuto è rimpiazzato da un testo sostitutivo, parte del documento a tutti gli effetti.
Le entità *unparsed,* invece, sono risorse, non necessariamente testuali, a cui è associata una notazione. Queste risorse vengono passate all'applicazione senza alcuna sostituzione.

Esistono tre tipi di entità, interne, esterne e parametriche. In generale, la sintassi per dichiarare un'entità è la seguente:

```
<!ENTITY EntityName “EntityValue”>
```

2.5.1 Entità interne

Le entità interne sono entità *parsed,* definite nel DTD con questa notazione:

```
<!ENTITY EntityName “InternalEntityValue”>
```

Quando il processore incontra il riferimento ad un'entità all'interno di un documento XML effettua la sostituzione con la stringa. Esistono cinque entità interne predefinite:

- < che il parser sostituisce con il simbolo <;
- >, che il parser sostituisce con il simbolo >;
- & che il parser sostituisce con il simbolo &;

- " che il parser sostituisce con il simbolo “;
- ' che il parser sostituisce con il simbolo ‘;

2.5.2 Entità esterne

Nella categoria delle entità esterne rientrano tutti i tipi di entità, ad esclusione di quelle interne. Le entità esterne fanno riferimento a definizioni non contenute in seno al DTD, ma a documenti esterni. Tuttavia, esattamente come le entità interne, vengono sostituite con il loro referente nel momento in cui il documento XML cui viene processato.
Per definire entità esterne si utilizza la seguente notazione:

```
<!ENTITY EntityName-1 SYSTEM “referente-esterno.xml”>
<!ENTITY EntityName-2 “http://www.dominio.it/referente-
esterno.xml”>
```

Le entità esterne possono essere definite come SYSTEM, se puntano fisicamente ad un determinato file, oppure come PUBLIC, nel caso in cui il riferimento al file sia contenuto in appositi registri di sistema.

Infine, le entità esterne possono essere *unparsed,* nel caso in cui venga specificato l’identificatore NDATA. In questo caso, l’entità fa riferimento ad una risorsa esterna, generalmente un file binario.

```
<!ENTITY image SYSTEM “image.gif” NDATA GIF>
```

Come accennato precedentemente, le entità *unparsed* sono risorse cui deve essere associata una notazione. Perciò, per poter utilizzare la dichiarazione di entità *image* appena introdotta, è necessario che questa sia preceduta all’interno della dtd da una notazione simile a questa:

```
<!NOTATION GIF SYSTEM “./ImageViewer.exe”>
```

2.5.3 *Entità parametriche*

Le entità parametriche sono assimilabili alle entità interne, con una differenza notevole. Infatti, le entità parametriche vengono espanse all'interno del DTD, e non all'interno del documento XML o dell'applicazione. A livello sintattico, questa differenza è evidenziata dall'uso del simbolo "%" come prefisso per referenziare entità parametriche.

```
<!ENTITY % EntityName Value>
```

All'atto pratico, le entità parametriche trovano applicazione nella definizione di attributi e valori comuni tra più elementi.

Questo capitolo raccoglie gli aspetti principali della sintassi del DTD. Per un'analisi più approfondita dell'argomento basta riferirsi alla "Raccomandazione" del W3C "*Extensible Markup Language (XML) 1.0 (Second Edition)*" [BPSM00]

3 XML SCHEMA

XML Schema nasce con uno scopo preciso: quello di estendere l'applicazione di XML al trasferimento dei dati tra applicazioni diverse. Alla base di questo passo risiede la concezione di XML come un linguaggio non limitato alla descrizione di documenti, ma anche come efficace strumento di interscambio dati, una sorta di lingua franca per il web. Proprio in questo genere di applicazioni XML Schema va ad affiancarsi e a completare DTD, perché aggiunge potenti strumenti per il controllo e la validazione dei dati e delle relative strutture.
Rispetto al DTD, XML Schema è molto più complesso e potente: supporta la programmazione modulare, può gestire strutture come i gruppi, gestisce i namespace e consente di stabilire vincoli precisi sul contenuto di elementi e attributi.
Le differenze tra XML Schema e DTD sono evidenti già dalla sintassi: mentre il DTD ha una sintassi tutta sua, XML Schema utilizza una sintassi XML. La principale differenza tra i due sta nella gestione degli elementi: infatti, XML Schema consente di specificare il tipo di ogni elemento, a partire da tipi semplici predefiniti, che possono essere combinati per creare tipi complessi.

3.1 Sintassi

Un documento XML Schema è un documento ben formato e valido, che esprime la struttura di un documento XML. Questa struttura viene espressa a partire da un elemento radice, <schema> appunto, all'interno del quale sono dichiarate le definizioni dei sotto-elementi. I documenti generati in maniera rispondente a queste definizioni vengono chiamate istanze del documento.
Con XML Schema emerge il concetto di validità rispetto ad uno schema, che si affianca a quello di validità e *well-formedness*. Ogni

elemento di uno schema XML è associato ad un namespace, in particolare a quello http://www.w3.org/-2001/XMLSchema/. È possibile associare a qualsiasi elemento il namespace di appartenenza, anche se per comodità ciò viene generalmente effettuato una volta per tutte nell'elemento radice. Gli attributi per assegnare uno schema ad un elemento sono *SchemaLocation* e *NoNamespaceSchemaLocation.*
Questa è la sintassi:

```
<my:doc xmlns:my="http://www.sito.it"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-
      instance"
      xsi:SchemaLocation="http://ww.miosito.org
      http://ww.miosito.org/files.xsd">
</my:doc>
```

Un documento XML Schema può contenere diversi elementi:

- `<element>`
- `<attribute>`
- `<import>`
- `<include>`
- `<simpleType>`
- `<complexType>`
- `<attibuteGroup>`
- `<group>`
- `<notation>`
- `<annotation>`

Come esempio la versione in XML Schema della scheda anagrafica utilizzata nel capitolo precedente, in una versione ridotta:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema elementFormDefault="qualified"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="persona" >
            <xsd:complexType >
                  <xsd:sequence >
                        <xsd:element ref="nominativo" />
                        <xsd:element ref="indirizzo" />
```

```
                  <xsd:element ref="nascita" />
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>

<xsd:element name="nominativo" >
      <xsd:complexType >
            <xsd:sequence >
                  <xsd:element ref="nome" />
                  <xsd:element ref="cognome" />
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>
. . .
<xsd:element name="nome" >
      <xsd:simpleType>
            <xsd:restriction base="xsd:string">
            </xsd:restriction>
      </xsd:simpleType>
</xsd:element>
. . .
</xsd:schema>
```

3.2 I tipi

A differenza del DTD, XML Schema utilizza il concetto di tipo di dato, sia per gli elementi, che per i loro attributi. Sono due le macrocategorie in cui si dividono i tipi: semplici e complessi. Mentre i primi possono essere usati sia per gli elementi che per gli attributi, i secondi possono essere usati solo per gli elementi. In ogni caso, ogni elemento ed ogni attributo deve sempre essere associato ad un tipo.

3.2.1 Tipi semplici

XML Schema definisce una grande quantità di tipi semplici, attraverso i quali si possono definire i tipi complessi. I tipi semplici si dividono in tre categorie fondamentali: predefiniti, anonimi e nominati. I tipi semplici predefiniti sono stati formalizzati dal W3C e vengono dettagliati nel documento "*Xml Schema part 2: Datatypes*"[BM01]. A questi si affiancano i tipi semplici anonimi, che vengono introdotti

all’interno di una dichiarazione di un attributo o di un elemento, e i tipi semplici nominati, che sono dichiarati globalmente dall’autore come *simple type:*

```
<xsd:simpleType name="etaUmana">
      <xsd:restriction base="xsd:integer">
            <xsd:minInclusive value="0"/>
            <xsd:maxInclusive value="125"/>
      </xsd:restriction>
</xsd:simpleType>
<xsd:element name="eta" type="etaUmana"/>
```

```
<xsd:element name="grado">
      <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="tenente"/>
                  <xsd:enumeration value="caporale"/>
                  <xsd:enumeration value="colonnello"/>
            <xsd:restriction>
      <xsd:simpleType>
<xsd:element>
```

3.2.2 Derivazione dei tipi semplici

XML Schema consente di derivare nuovi tipi semplici dai tipi semplici pre-esistenti. Questo accade in tre modi:

- **Restrizione**: la derivazione per restrizione crea un nuovo tipo semplice applicando dei vincoli ad un tipo semplice pre-esistente. Questi vincoli vengono chiamati *facet.*

```
<xsd:simpleType name="etaBambino">
      <xsd:restriction base="etaUmana" >
            <xsd:minInclusive value="0" />
            <xsd:maxInclusive value="12" />
      </xsd:restriction>
</xsd:simpleType>
```

- **Unione**: un tipo derivato per unione è una semplice lista di valori di altri tipi semplici, come in questo esempio:

```
<xsd:simpleType name="peso">
      <xsd:union>
            <xsd:simpleType>
                  <xsd:restriction base="xsd:float">
            </xsd:restriction>
            </xsd:simpleType>
            <xsd:simpleType>
                  <xsd:restriction base="#unit;" >
                  </xsd:restriction>
            </xsd:simpleType>
      </xsd:union>
</xsd:simpleType>
```

- **Lista:** i tipi derivati per lista sono una lista di valori di un determinato tipo atomico, come nell'esempio:

```
<xsd:simpleType name=”MyInteger”>
      <xsd:restriction base=”xsd:integer”>
            <xsd:minInclusive value=”10”/>
            <xsd:maxInclusive value=”99”/>
      </xsd:restriction>
</xsd:sympleType>

<xsd:simpleType name=”listOfMyInteger”>
      <xsd:list itemType=”MyInteger”/>
</xsd:simpleType>
```

Nel documento, questo tipo verrà istanziato in questo modo:

```
<MyIntegerList>11 23 77 89</MyIntegerList>
```

3.2.3 Facet

Il *facet* esprime particolari vincoli applicabili sui tipi semplici, in sostanza rappresenta particolari caratteristiche di un determinato tipo. In particolare, attraverso *facet* è possibile rappresentare vincoli rispetto l'ordine, la cardinalità, i valori e i limiti numerici.

- *length, minLength, maxlength:* definiscono o limitano la lunghezza di un tipo, e si applicano principalmente a stringhe e loro derivati.

- *minInclusive, minExclusive, maxInclusive, maxExclusive:* si applicano a tipi ordinati, come “number” o “duration”, per definirne i limiti massimi e minimi;

- *totalDigits, fractionDigits:* stabiliscono il numero delle cifre o dei decimali, per i tipi numerici e i derivati da decimal.

- *enumeration:* limita l’insieme dei valori possibili ad una lista predefinita;

- *whitespace:* consente di esprimere una politica nella gestione degli spazi bianchi;

- *pattern:* limita l’insieme dei valori possibili a quelli che soddisfano un’espressione regolare definita.

Chiaramente non tutti i *facet* possono essere applicati a tutti i tipi semplici: l’applicabilità o meno di determinati *facet* dipende essenzialmente dal tipo cui devono essere applicati.

```
<xsd:simpleType name=”valuta”>
      <xsd:restriction base=”xsd:decimal”>
            <xsd:totalDigits value=”5”>
            <xsd:fractiondigits value=”2”>
      </xsd:restriction>
<xsd:simpleType>
```

Non sarà possibile, per esempio applicare questi *facet* ad un tipo *integer*.

3.2.4 Tipi complessi

I tipi complessi si definiscono con *ComplexType* e possono contenere sia elementi che attributi. Possono essere di due tipi, nominati e anonimi: mentre i primi vengono definiti globalmente nello schema, i secondi sono definiti all'interno di una dichiarazione di elemento, senza che ne venga specificato il nome. Il content model di un *ComplexType* può essere di tipo empty, eny, strutturato e misto.

3.2.4.1 Empty content model

È possibile definire un content model per gli elementi vuoti, utilizzando un tipo complesso. In questo modo si può ottenere un risultato analogo a quello che si otterrebbe con una dichiarazione EMPTY nel DTD.

```
<xsd:element name="empty">
      <xsd:complexType/>
</xsd:element>
<xsd:complexType name="emptyAttr">
      <xsd:attribute name="attribute" type="xsd:string"/>
</xsd:compleType>
<xsd:element name="emptyElement" type="emptyAttr"/>
```

3.2.4.2 Any content model

Il contenuto di un elemento definito attraverso *anyType* non sottosta ad alcun tipo di vincolo. Questo perché il tipo *anyType* rappresenta l'astrazione del tipo *ur-type*, ovvero il tipo base dal quale derivano per astrazione o restrizione tutti gli altri tipi, semplici o complessi. Un tipo complesso può essere ereditato restringendo esplicitamente da *anyType* ma la restrizione può anche essere omessa.

Di seguito due esempi di come il tipo *anyType* venga ereditato in maniera implicita e esplicita:

```
<xsd:complexType name="nominativo">
      <xsd:sequence>
```

```
            <xsd:element name="nome" type="xsd:string">
            <xsd:element name="cognome" type="xsd:string">
      </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="nominativo">
      <xsd:complexContent>
            <xsd:restriction base="xsd:anyType">
                  <xsd:sequence>
                  . . .
                  </xsd:sequence>
            </xsd:restriction>
      </xsd:complexContent>
</xsd:complexType>
```

3.2.4.3 Content model strutturato

È possibile strutturare un content model, specificando in che modo gli elementi che lo popolano debbano apparire. Per questo si utilizzano i tag <sequence>, <choice> ed <all>, congiuntamente agli attributi *minOccurs* e *maxOccurs*, attraverso i quali si ottiene una funzione comparabile, ma più precisa, di quella offerta nella dtd da "+", "?", "*". Nel dettaglio:

- *sequence:* specifica che quanto è compreso tra i tag di inizio e fine deve rispettare l'ordine sequenziale espresso nello schema:

```
<xsd:element name="indirizzo">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="via"
                              type="xsd:string"/>
                  <xsd:element name="comune"
                              type="xsd:string"/>
                  <xsd:element name="cap"
                              type="xsd:string"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>
```

- *choice:* specifica che almeno uno degli elementi elencati deve comparire. Può essere comparato al “|” del DTD.

```
<xsd:element name="peso">
      <xsd:complexType>
            <xsd:choice>
                  <xsd:element name="grammi"
                              type="xsd:decimal"/>
                  <xsd:element name="libbre"
                              type="xsd:decimal"/>
            </xsd:choice>
< /xsd:complexType>
</xsd:element>
```

- *all:* specifica che tutti gli elementi elencati devono comparire, ma non ne vincola l’ordine.

```
<xsd:element name="pentagramma">
      <xsd:complexType>
            <xsd:all>
                  <xsd:element name="nota"
                              type="noteType"/>
                  <xsd:element name="pausa"
                              type="pauseType"/>
            </xsd:all>
      </xsd:complexType>
</xsd:element>
```

3.2.4.4 Content model misto

Il content model misto può contenere, oltre ad elementi, dati di tipo carattere. Poiché i tipi complessi possono contenere per definizione solo elementi, è stato introdotto un attributo “mixed” per introdurre dati di tipo carattere in questo contesto.

```
<xsd:element name="nominativo">
      <xsd:complexType mixed="true">
            <xsd:sequence>
                  <xsd:element name="nome"
                              type="xsd:string"/>
```

```
                <xsd:element name="cognome"
                            type="xsd:string"/>
          </xsd:sequence>
     </xsd:complexType>
</xsd:element>

<nominativo>
     Dati di tipo carattere
     <nome>Davide</nome>
     <cognome>Fiorello</cognome>
</nominativo>
```

3.2.5 Derivazione dei tipi complessi

I tipi, siano essi semplici o complessi, possono essere usati come atomi per la derivazione di nuovi tipi, che indipendentemente dal tipo originale, saranno tipi complessi. XML Schema fornisce meccanismi molto potenti per derivare tipi complessi. I metodi di derivazione sono i seguenti:

- **restrizione***:* questo metodo consente di derivare un nuovo tipo attraverso l'applicazione di particolari restrizioni ad un tipo base.

  Tipo base:
  ```
  <xsd:complexType name="nominativo">
       <xsd:sequence>
            <xsd:element name="nome"
                        type="xsd:string"/>
            <xsd:element name="cognome"
                        type="xsd:string"/>
       </xsd:sequence>
  </xsd:complexType>
  ```

  Derivato:
  ```
  <xsd:complexType name="nominativoRistretto">
       <xsd:complexContent>
            <xsd:restriction base="nominativo">
                 <xsd:sequence>
                      <xsd:element name="nome"
                                  type="xsd:string"/>
  ```

```
                </xsd:sequence>
            </xsd:restriction>
        </xsd:complexContent>
    </xsd:complexType>
```

- **estensione**: questo metodo consente di creare un nuovo tipo complesso, aggiungendo elementi o attributi ad un tipo precedentemente creato. Non esistono limiti agli elementi aggiuntivi, purché vengano inseriti alla fine del content model del tipo base. Il tipo complesso derivato è considerato come una sequenza composta dal tipo base e dalla sua estensione. Utilizzando il tipo complesso "personnel" definito precedentemente, ecco un esempio di tipo complesso derivato per estensione:

```
<xsd:complexType name="nominativoEsteso">
<xsd:complexContent>
        <xsd:extension base="nominativo">
            <xsd:sequence>
                <xsd:element name="indirizzo"
                    type="xsd:string"/>
            </xsd:sequence>
        </xsd:extension>
        <xsd:attribute name="eta"
                    type="xsd:integer"/>
    </xsd:complexContent>
</xsd:complexType>
```

3.3 Elementi e attributi

Gli elementi e gli attributi vengono definiti attraverso un tag, che presenta due attributi di base: *name* e *type.*

```
<xsd:element name="nome" type="xsd:string"/>
<xsd:attribute name="altezza" type="xsd:decimal"/>
```

Il tipo può essere specificato anche ricorrendo ad una dichiarazione anonima di un compelxType direttamente all'interno dell'elemento.

Si possono specificare altri attributi, oltre agli obbligatori *name* e *type.* In particolare, è possibile definire valori di default o valori fissi:

- *default*: specifica il valore di default, che deve essere obbligatoriamente di tipo *character data*. Può essere usato sia con gli attributi che con gli elementi. In quest'ultimo caso il valore di default viene applicato nei casi di elemento vuoto, mentre per gli attributi, il valore di default viene applicato in mancanza di una sua esplicitazione nell'istanza di documento.

- *fixed*: specifica un valore fisso, sia per gli attibuti che per gli elementi, anche in questo caso obbligatoriamente di tipo *character data*.

- *minOccurs*: Specifica il numero minimo di occorrenze dell'elemento. Il valore di default di *minOccurs* è uguale a 1.

- *maxOccurs*: Specifica il numero massimo di occorrenze dell'elemento. Il valore di default di *minOccurs* è uguale a 1. Per un numero infinito di occorrenze è necessario assegnare a *maxOccurs* il valore *unbounded*.

- *use*: Specifica il numero di occorrenze dell'attributo. Può assumere i valori: *optional*, *required* e *prohibited*.

3.3.1 Definizione globale e locale

XML Schema dà la possibilità di definire gli elementi sia localmente che globalmente, allo scopo di rendere il codice modulare.

Definizione locale. Sono definiti localmente tutti gli elementi (o gli attributi) dichiarati all'interno di un tipo complesso. Questo significa che l'elemento (o l'attributo) in questione esiste solo internamente all'elemento che assume quel tipo particolare di cui fa parte.

```
<xsd:complexType name="nominativo">
      <xsd:sequence>
            <xsd:element name="nome" type="xsd:string"/>
            <xsd:element name="cognome"
                              type="xsd:string"/>
      </xsd:sequence>
</xsd:complexType>
```

Definizione globale. Sono definit globalmente, invece, tutti gli elementi, o gli attributi, dichiarati al *top-level,* ovvero come figli diretti dell'elemento schema. Solo gli lementi definiti globalmente possono essere usati come elementi radice dell'istanza di documento. Una particolarità degli elementi definit globalmente è la loro immutabilità: non se ne può specificare la cardinalità, né l'opzionalità.
Xml Schema offre la possibilità ad elementi e attributi non definiti globalmente di fare riferimento a questi: esiste a questo scopo l'attributo *ref* , da utilizzare in sostituzione di *name.*

```
<xsd:element name="nome" type="xsd:string"/>
<xsd:element name="cognome" type="xsd:string"/>

<xsd:complexType name="nominativo">
      <xsd:sequence>
            <xsd:element ref="nome"/>
            <xsd:element ref="cognome"/>
      </xsd:sequence>
</xsd:complexType>
```

3.3.2 Valore NIL

XML Schema consente di definire elementi *nillable*, che possono cioè, essere assenti nell'istanza di un documento.

```
<xsd:element name="nominativo" type="xsd:string"
nillable="true"/>
```

L'istanza del documento riporterà un elemento come questo:

```
<nominativo xsi:nil="true"></nominativo>
```

L'attributo *nil* appartiene al *namespace* xsi:http://www.w3.org/2001/XML-Schema-istance, e deve essere specificato nel documento XML. Gli elementi che presentano l'attributo *nil,* possono avere altri attributi, ma non possono mai contenere sotto-elementi.

3.3.3 Raggruppare elementi e attributi

Può risultare conveniente raggruppare elementi o attributi, anche quando non è possibile ricorrere a tipi complessi. Per questo xml schema offre due attributi, *group* e *elementGroup*:

```
<xsd:group name="personale">
      <choice>
            <xsd:element name="programmatore"
                  type="persona"/>
            <xsd:element name="designer"
                  type="persona"/>
      </choice>
</xsd:group>

<xsd:attributeGroup name="personaleAttr">
      <xsd:attribute name="reparto" type="codice-
                                                     reparto"/
                                              >
</xsd:attributeGroup>
```

I gruppi, definiti globalmente, possono essere utilizzati localmente attraverso il meccanismo del riferimento, ovvero con l'attributo *ref*, come nell'esempio:

```
<xsd:complexType name="team-di-coppia">
```

```
      <sequence>
            <xsd:group ref="personale"/>
      </sequence>
      <xsd:attributeGroup ref=" personaleAttr"/>
</xsd:complexType>
```

3.3.4 Unicità degli elementi e degli attributi

All'interno di uno stesso *scope,* si può specificare che un determinato elemento (o un attributo) debba risultare unico. Per questo si utilizzano gli elementi *unique, selector e field*. Prima si identifica il campo degli elementi e poi si selezionano gli attributi o gli elementi che in quel campo debbano risultare unici:

```
<xsd:simpleType name="codiceProdotto">
      <xsd:restriction base="xsd:decimal">
            <xsd:minExclusive value="1000"/>
            <xsd: minExclusive value="9999"/>
      </xsd:restriction>
</xsd:simpleType>

<xsd:element name="magazzino">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="prodotto"
                              minOccurs="0"
                              maxOccurs="unbounded"/>
                        <xsd:complexType>
                              <xsd:sequence>
                                    <xsd:element name="nome"
                                          type="xsd:string"/>
                                    <xsd:element name="code"
                                          type="codiceProdotto
                                    "/>
                              </xsd:sequence>
                        </xsd:complexType>
                        <xsd:unique name="UNIQ">
                              </xsd:selector
xpath="prodotto"/>
                              </xsd:field
                                    xpath="codiceProdotto"/>
                        </xsd:unique>
```

```
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>
```

3.4 Definizione di key

Gli elementi *key* e *keyref* svolgono in XML Schema la funzione di *ID* e *IDREF* nel DTD: consentono, all'interno di uno stesso documento, di fare riferimento ad alcune sue parti.
Il costrutto *key* presenta questi vincoli:

- deve essere nidificato all'interno di un documento;
- deve trovarsi alla fine di un elemento, dopo la dichiarazione del contenuto e gli attributi;
- deve usare l'elemento figlio <selector> per selezionare un insieme di elementi;
- deve usare l'elemento figlio <field> per identificare quale elemento o attributo funziona come *key*.

3.5 La documentazione

XML Schema, introduce una possibilità molto interessante per quanta riguarda la documentazione. L'elemento *annotation*, infatti, offre la possibilità di inserire commenti, utili e leggibili non solo all'uomo, ma addirittura parsabili e interpretabili dalle macchine. Si apre la strada ha possibilità molto promettenti, come la generazione di opportuna documentazione grazie a fogli di stile dedicati. *Annotation* può essere usato in combinazione con gli elementi *documentation* e *appinfo* e deve sempre comparire per primo in una dichiarazione:

```
<xsd:annotation>
      <xsd:documentation xml:lang="en">
```

```
          This element stores a value
      </xsd:documentation>
</xsd:annotation>

<xsd:annotation>
      <xsd:appinfo source="anyURI"/>
</xsd:annotation>
```

3.6 Namespace e gestione

Parser e Processor devono essere in grado di gestire le ambiguità che dovessero presentarsi qualora un documento avesse al suo interno dichiarazioni diverse della stessa struttura. Grazie ai namespace è possibile gestire senza ambiguità tag con lo stesso nome ma con significati differenti, perché appartenti ad ambienti diversi. XML Schema gestisce i namespace in maniera estremamente sofisticata, a differenza del DTD.
Un namespace è una raccolta di nomi identificati da un uri. È composto di due parti: il nome del namespace e la parte locale, che corrisponde al nome dell'elemento o dell'attributo.

- *targetNamespace:* questo attributo, che va associato all'elemento <schema>, specifica il namespace di riferimento per lo schema. Gli elementi e gli attributi dichiarati al *top level* sono assegnati a questo namespace.

- *elementFormDefault, attributeFormDefault:* questi attributi dell'elemento <schema> danno due tipi di informazione. Se sono impostati a "*qualified*", gli elementi e gli attributi all'interno dei tipi complessi vengono assegnati al targetNamespace; se, invece, sono settati a "*unqualified*", gli elementi (e gli attributi) definit localmente non vengono associati ad alcun namespace.

- *form:* questo attributo consente di dichiarare localmente le proprietà specifiche di ciascun elemento.

3.7 Any ed AnyAttribute

XML Schema offre la possibilità di estendere i documenti con elementi appartenenti a namespace specifici, attraverso l'uso di *wildcard*, nello specifico grazie a *any* e *anyAttribute*. Il primo consente di inserire all'interno di un dato content model qualsiasi elemento, mentre il secondo, similmente, consente di inserire qualsiasi tipo di attributo.
In aggiunta, si può specificare il namespace al quale questi elementi appartengono, usando l'attributo *namespace.*

- *##targetNamespace*: in questo punto possono apparire solo elementi che appartengono al targetNamespace.

- *##other*: in questo punto possono apparire solo elementi che appartengono ad un altro namespace, esclusi quelli appartenenti al targetNamespace;

- *##local*: in questo punto possono apparire solo elementi non appartenenti ad alcun namespace;

- *##any*: in questo punto possono comparire elementi e attributi appartenenti a qualsiasi namespace, senza restrizioni;

- valore: possono comparire solo elementi e attributi del namespace specificato.

Si può specificare l'attributo *processContent* per stabilire il tipo di validazione da effettuare sulle *wildcard:*

- *strict*: forza una validazione che controlli la validità rispetto allo schema;

- *lax*: richiede un controllo dello schema, e se non viene trovato, verifica esclusivamente la wellformedness del documento;

- *skip*: accetta qualsiasi documento ben formato senza validarlo.

3.8 Include, redefine e import

Per mantenere il codice leggibile, XML Schema offre strumenti per modularizzarlo, in modo da poter gestire file relativamente brevi. In particolare, è possibile includere porzioni di codice, attraverso l'istruzione *include;* ridefinirne delle parti con *redefine;* importarne da altri namespace con *import.* Vediamo nel dettaglio, con alcuni esempi:

- *include*: permette di includere schemi che condividono il namespace con quello corrente.

```
<xsd:include
      schemaLocation="http://www.temp.org/foo.xsd"/>
```

- *redefine*: permette di includere uno schema dello stesso namespace, ma in più offre la possibilità di modificare i valori inseriti, di *simpleType*, *complexType*, *group* ed *attributeGroup.* Nell'esempio, il tipo complesso *test* viene ridefinito:

```
<xsd:redefine
      schemaLocation="http://www.temp.org/foo.xsd">
      <xsd:complexType name="test">
      </xsd:complexType>
</xsd:redefine>
```

- *import*: permette di importare strutture da namespace differenti rispetto a quello di origine. In questi casi è fondamentale l'uso corretto dei prefissi, che consentono di discriminari rispetto a quali namespace vengono effettuate le dichiarazioni.

```
<xsd:schema ... xmlns:temp=”http://www.temp.org”>
     <xsd:import namespace=”http://www.temp.org”

     schemaLocation=”http://www.temp.org/foo.xsd”/>
</xsd:schema>
```

3.9 XML Schema e le entità

La gestione delle entità rappresenta il principale limite di XML Schema, rispetto al DTD. Infatti, non è possibile dichiararle internamente, prima di tutto per ragioni storiche. XML Schema non punta a sostituire il DTD, quanto piuttosto a fornire il modo di dichiarare particolari strutture, impossibili con i DTD.

Per dichiarare entità all’interno di un documento da validare con uno schema è necessario anteporre alla dichiarazione dello schema la dichiarazione del DTD contenente le entità:

```
<!DOCTYPE person [
      <!ENTITY euro   "&#8364;">
]>
<prodotto … SchemaLocation=…>
      <prezzo>
            25&euro;
      </prezzo>
</prodotto>
```

Questo capitolo raccoglie la parte degli aspetti della sintassi di XML Schema che sono serviti per poter creare l’estensione della sintassi dei DTD. Per un’analisi più approfondita dell’argomento riferirsti alla “Raccomandazione” del W3C: *XML Schema Part 0: Primer*[Fal01], *XML Schema Part 1: Structures*[TBMM01], *XML Schema Part 2: Datatypes*[BM01].

4 DTD++

Il DTD++, realizzato da Nicola Amorosi[VAG03], rappresenta un'estensione sintattica del DTD, che si propone di colmare, anche se parzialmente, il divario funzionale che lo separa da XML Schema. Alla base di DTD++ sta il principio di non stravolgere la sintassi del DTD, integrandola con il minor numero di cambiamenti possibili: rimangono possibili tutti i costrutti tipici di DTD, in più vengono introdotte una serie di estensioni.
DTD++ consente di dichiarare entità che hanno lo stesso valore dei tipi semplici in XML Schema: a questo scopo sono stati introdotti alcuni tipi predefiniti, con le stesse caratteristiche di quelli espressi nella *recommendation Schema Part 2*[BM01].
Si è supposto che alcune stringhe, che possono comparire tra i valori delle entità, nei content model degli elementi e tra le liste degli attributi, abbiano un valore particolare e debbano perciò essere usate di conseguenza. Questi valori equivalgono ai tipi predefiniti in XML Schema:

4.1 Tipi semplici

Sintassi:

```
<!ENTITY # nome "(tipobasevincoli)">
<!ELEMENT nomeElemento (tipobasevincoli)>
<!ATTLIST nomeElemento nomeAttributo
(tipobasevincoli) default>
```

Esempio:

```
<!ENTITY # newInteger "(#INTEGER)">
```

I tipi predefiniti non possono essere ri-dichiarati: questo signfica che l'utente non può definire alcun tipo semplice con lo stesso nome.
I tipi semplici nominati si dichiarano attraverso un'entità in cui i valori tra parentesi sono sempre tipi semplici, o loro restrizioni; si possono

usare tipi semplici definiti dall'utente racchiudendone il nome tra i simboli "#" e ";".

```
<!ENTITY # myInteger "(#newInteger;)">
```

I tipi semplici definiti in questo modo possono essere usati anche nel dichiarare elementi o attributi:

```
<!ELEMENT salary (#myInteger;)>
<!ELEMENT name (#STRING)>
```

Gli aspetti fondamentali sono due: i nomi dei tipi devono sempre essere racchiusi tra parentesi e preceduti dal simbolo #, i tipi definiti dall'utente, invece, oltre ad essere racchiusi tra parentesi e ad essere preceduti dal simbolo "#", devono anche essere seguiti da ";". Lo stesso vale per gli attributi che, oltre all'insieme di tipi già esistenti nel DTD, possono assumere nuovi valori.

```
<!ATTLIST salary value (#newInteger;) #IMPLIED>
```

La sintassi per definire i tipi degli attributi è rimasta quella del DTD:

```
<!ATTLIST salary value (#CDATA) #IMPLIED>
```

È equivalente a:

```
<!ATTLIST salary value CDATA #IMPLIED>
```

#STRING e #CDATA	xsd:string
#BOOLEAN	xsd:boolean
#DECIMAL	xsd:decimal
#FLOAT	xsd:float
#DOUBLE	xsd:double
#DURATION	xsd:duration
#DATETIME	xsd:datetime
#TIME	xsd:time
#DATE	xsd:date
#GYEARMONTH	xsd:gYearMonth
#GYEAR	xsd:gYear
#GMONTHDAY	xsd:gMonthDay
#GDAY	xsd:gDay
#GMONTH	xsd:gMonth
#HEXBINARY	xsd:hexBinary
#BASE64BINARY	xsd:base64Binary
#ANYURY	xsd:anyURI
#QNAME	xsd:QName
#NOTATION	xsd:NOTATION
#NORMSTRING	xsd:normalizedString
#TOKEN	xsd:TOKEN
#NMTOKEN	xsd:NMTOKEN
#NMTOKENS	xsd:NMTOKENS
#NAME	xsd:Name
#NCNAME	xsd:NCName
#ID	xsd:ID
#IDREF	xsd:IDREF
#IDREFS	xsd:IDREFS
#ENTITY	xsd:ENTITY
#ENTITIES	xsd:ENTITIES
#INTEGER	xsd:integer
#NONPOSINTEGER	xsd:nonPositiveInteger
#NEGINTEGER	xsd:negativeInteger
#LONG	xsd:long
#INT	xsd:int
#SHORT	xsd:short
#BYTE	xsd:byte
#NONNEGINTEGER	xsd:nonNegativeInteger
#ULONG	xsd:unsignedLong
#UINT	xsd:unsignedInt
#USHORT	xsd:unsignedShort
#UBYTE	xsd:unsignedByte
#POSINTEGER	xsd:positiveInteger

Tabella 4.1: Corrispondenza tra tipi predefiniti nei DTD++ e XML Schema

4.1.1 Derivazione per restrizione

La derivazione per restrizione avviene applicando al tipo base dei vincoli, quelli che in XML Schema sono chiamati *facet.*

Sintassi:

```
<!ENTITY # nome "(tipobase{}[]//()\\)">
<!ELEMENT nomeElemento (tipobase{}[]//()\\)>
<!ATTLIST nomeElemento nomeAttributo
 (tipobase{}[]//()\\) default>
```

{lunghezza}
[range]
/pattern/
(enumerazione)
whitespace\\

Esempi:

```
<!ENTITY # myInteger "(#INTEGER[10,100])">
```

Da un punto di vista sintattico, è importante notare che non vanno inseriti spazi tra tipobase ed i vincoli. Allo stesso modo, non vanno inseriti spazi nemmeno tra un vincolo e l'altro.

Definizione della lunghezza. La lunghezza di un tipo viene definita ricorrendo a questi valori:

- ***length*****:** identifica l'unità di lunghezza, variabile a seconda del tipo di dato. Può essere esclusivamente un intero positivo, racchiuso tra parentesi graffe.
- ***minlength*****:** identifica la lunghezza minima che può assumere un valore di un dato tipo. Deve essere obbligatoriamente un intero non negativo, seguito da una virgola.
- ***maxlength*****:** esprime la lunghezza massima che può assumere un valore di un dato tipo.
- Deve essere obbligatoriamente un intero non negativo, preceduto da una virgola.

- ***totalDigits:*** è un numero che identifica il numero di cifre di un elemento. Questo valore viene espresso con un intero positivo racchiuso tra parentesi graffe e seguito da un punto. Se combinato con *fractiondigits,* quest'ultimo sarà il valore dopo il punto.
- ***fractionDigits:*** è un numero che identifica il numero delle cifre della parte decimale di un elemento. Questo valore viene espresso con un intero positivo, racchiuso tra parentesi graffe e preceduto da un punto. Se combinato con totalDigits, quest'ultimo sarà il numero prima del punto.

Definizione del range. Il range viene definito ricorrendo a questi valori:

- ***maxInclusive:*** è un numero che varia a seconda del tipo cui è applicato e indica un limite superiore all'insieme di valori di quel tipo. Può essere combinato con minExclusive o minInclusive. Questo valore viene espresso come un numero preceduto da una virgola e racchiuso tra parentesi quadre.
- ***maxExclusive:*** è un numero che impone un limite superiore all'insieme dei valori possibili, escludendo il numero specificato. Si può combinare con MinExclusive o MinInclusive. Questo valore viene espresso come un numero preceduto da una virgola e racchiuso tra parentesi quadre.
- ***minInclusive:*** è un numero che indica il limite inferiore all'insieme dei valori possibili, includendo il numero specificato. Può essere combinato con maxExclusive o MaxInclusive. Questo valore viene espresso come un numero seguito da una virgola e racchiuso tra parentesi quadre.
- ***maxExclusive:*** è un numero che indica il limite inferiore all'insieme dei valori possibili, escludendo il numero specificato. Può essere combinato con maxExclusive o MaxInclusive. Questo valore viene espresso come un numero seguito da una virgola e racchiuso tra parentesi quadre

Definizione del pattern. Il pattern restringe lo spazio dei valori per un tipo di dato, attraverso la definizione di un'espressione regolare. L'espressione regolare individua un insieme di valori validi. Un elemento che ha un tipo derivato per restrizione con questo *facet* potrà contenere solo valori che combaciano con quelli espressi dal vincolo.

Definizione dell'elenco (enumeration). Questo *facet* vincola lo spazio dei valori ad un determinato insieme, ma non pone relazioni d'ordine tra gli elementi dell'insieme. Questo genere di vincolo viene espresso attraverso un elenco di valori, racchiusi tra parentesi tonde e divisi tra loro dal simbolo "|".

Definizione del whitespace. Questo vincolo dà all'autore la possibilità di specificare come debbano essere trattati gli spazi bianchi (andate a capo, spazi, tabulatore). Il vincolo può essere espresso con questi valori:

- **p:** specifica al parser di non effettuare alcuna normalizzazione;
- **r:** rimpiazza con uno spazio bianco (#x20) tutte le occorrenze di tabulatore (#x9), interlinea (#xa) e ritorno a capo (#xD);
- **c:** esegue lo stesso processo di "r", in più, sostiutisce gli spazi bianchi contigui con uno spazio bianco unico.

Il *facet* whitespace può essere utilizzato con tutti i tipi atomici e i derivati per lista. In particolare, per tutti i tipi atomici diversi dalle stringhe il valore di questo vincolo è "c"; per le stringhe, invece, il vincolo ha il valore di default "p".

Tutti questi *facet* possono venire applicati nelle dichiarazioni di attributi, in maniera analoga a quanto avviene con gli elementi.
I tipi CDATA, ID, IDREF, IDREFS, NMTOKEN, NMTOKENS, ENTITY e ENTITIES richiedono una sintassi leggermente diversa da

quella usata per esprimere i *facet*. Più precisamente, bisogna esprimere i vincoli come in una dichiarazione di attributo:

```
<!ATTLIST name position (CDATA{3}) #IMPLIED>
```

La seguente dichiarazione è invece errata:

```
<!ATTLIST name position CDATA{3} #IMPLIED>
```

Rimane possibile la dichiarazione di valori di default o prefissati:

```
<!ATTLIST name position (CDATA{3}) #FIXED "uno">
```

Si possono applicare anche più *facet* contemporaneamente, rispettando un ordine sequenziale. Chiaramente l'applicabilità di determinati facet rimane vincolata al tipo di dato cui si riferiscono.

4.1.2 Derivazione per lista

La derivazione per lista si ottiene aggiungendo al tipo di base il simbolo "+".

Sintassi:

```
<!ENTITY # nome "tipobase+{}[]//()\\">
<!ELEMENT # nomeElemento (tipobase+{}//()\\)>
<!ATTLIST # nomeElemento nomeAttributo
(tipobase+{}[]//()\\) default>
```

Esempi:

```
<!ENTITY # ListOfName "(#STRING+{5}/[A-Z]/)">
<!ELEMENT # Name "(#STRING+{5}/[A-Z]{5}/)">
```

Gli elementi il cui tipo è stato derivato per lista conterranno una lista di valori del tipo atomico derivato. Vediamo un esempio:

```
<!ENTITY # newDecimal "(#DECIMAL)">

<!ELEMENT elem (#newDecimal;+)">
```

L'istanza di questa dichiarazione potrà avere la seguente forma:

```
<elem>15 345 94</elem>
```

I tipi base derivati per lista possono essere vincolati ulteriormente, ricorrendo a derivazioni per restrizione. Per convenzione, il simbolo "+" è il primo ad apparire dopo il tipo semplice. I nuovi tipi derivati per lista funzioneranno come tipi semplici e potranno a loro volta essere derivati per restrizione. In questo caso, possono applicarsi unicamente questi vincoli: length, minLength, maxLength, pattern, enumeration, whitespace.

4.1.3 Derivazione per unione

La derivazione per unione genera un nuovo tipo semplice, che è l'unione dei tipi derivati. Questa derivazione si esprime ponendo i tipi base coinvolti tra parentesi, separati tra loro dal simbolo "|".

Sintassi:
```
<!ENTITY # nome "union">
<!ELEMENT # nomeElemento (union)>
<!ATTLIST # nomeElemento  nomeAttributo union
default>
```

union: `(tipo base+ vincoli) | (tipo base+ vincoli)`

Esempi:
```
<!ENTITY # newDecimal "(#STRING{4})|(#FLOAT)">
<!ELEMENT salary ((#STRING)|(#DECIMAL{3.}))>
```

Si possono definire tipi derivati per unione sia nelle dichiarazioni di tipo, sia in quelle di elemento o attributo.

```
<!ENTITY # salaryType "(#DECIMAL[0,100])|(#STRING{3})">

<!ELEMENT salary ((DECIMAL[0,100])|(STRING{3}))>
```

```
<!ATTLIST newsalary value (#DECIMAL[0,100])|(#STRING{3})
#IMPLIED>
```

Le derivazioni per unione effettuate direttamente nelle dichiarazioni di content model dell'elemento, o in quelle di tipo degli attributi, devono essere racchiuse tra paretesi tonde. I tipi definiti in questo modo potranno essere derivati per restrizione, attraverso *facet* di tipo pattern e enumeration. L'applicazione di alcuni vincoli limita di fatto l'applicazione di altri: è compito dell'utente porre la dovuta attenzione nell'applicare particolari restrizioni.

L'utilizzo del tipo union all'interno delle ATTLIST ha subito una modifica rispetto alla precedente versione. Al fine di dare coerenza alla sintassi sono state eliminate le parentesi che comprendevano la definizione di tipo dell'attributo.
Quindi, quello che originariamente era:

```
<!ATTLIST newsalary value ((#DECIMAL)|(#STRING)) #IMPLIED>
```

diventa

```
<!ATTLIST newsalary value (#DECIMAL)|(#STRING) #IMPLIED>
```

4.2 I tipi complessi

4.2.1 Definizione dei tipi complessi.

I tipi complessi vengono dichiarati usando una nuova definizione di entità, attraverso il carattere @. Possono contenere al loro interno sia elementi che attributi. La sintassi richiede alcune specificazioni:

- un tipo complesso contiene una struttura simile al content model di un elemento, racchiusa tra le prime virgolette, che sono sempre obbligatorie;

- se presenti, gli attributi devono essere racchiusi tra virgolette, altrimenti le virgolette non sono necessarie;

- i valori delle dichiarazioni devono essere separati da spazi, gli elementi del content model sono sempre racchiusi tra parentesi tonde e gli attributi vanno separati tra loro con uno spazio.

Sintassi: `<!ENTITY @ nome "contentmodel" "listaAttributi">`

Esempi: `<!ENTITY @ personType "(nome, email)" "age CDATA #IMPLIED">`

Si possono definire tipi complessi che contengono solo elementi o solo attributi, con le seguenti sintassi:

```
<!ENTITY @ personType "(firstname | secondname)">
<!ENTITY @ personType2 "" "age CDATA #IMPLIED year CDATA
#REQUIRED">
```

Per utilizzare un tipo complesso nel content model, è necessario racchiuderlo tra parentesi e tra i simboli "@" e ";", come in questo esempio:

```
<!ELEMENT person (@persontype;)>
```

4.2.2 Derivazione per estensione

DTD++ consente di derivare i tipi complessi per estensione, aggiungendo valori a quelli definiti. L'estensione si effettua attraverso

una dichiarazione di entità complessa, in cui appaiono in sequenza il tipo base da estendere e gli elementi aggiuntivi.
Ci sono alcune cose da ricordare:

- il tipo complesso è sempre racchiuso tra “@” e “;” ed è separato dal content model che lo estende attraverso una virgola;

- I tipi base possono venire estesi anche con attributi. Per effetuare l’estensione solo attraverso attributi bisognerà porre il tipo base tra le prime virgolette e gli attributi tra le seconde.

```
<!ENTITY @ personType "(firstname, secondname)">
<!ENTITY @ elemType1 "@personType;" "year CDATA
#REQUIRED">
```

Sintassi:
```
<!ENTITY @  nome "tipoBaseExt,contentmodel"
listaAttr>
```

tipoBaseExt: Tipo da derivare per estensione;

Esempio:
```
<!ENTITY @ personType "(firstname)">
<!ENTITY @ elemType
"@personType;,(firstname,secname)" "age CDATA
#IMPLIED">
```

4.2.3 Derivazione per restrizione

La derivazione per restrizione consente di introdurre vincoli ai tipi complessi definiti dall’utente. Per farlo, si dichiara un nuovo tipo complesso come restrizione di un tipo già esistente.

```
<!ENTITY @ personType "(firstname,name*)" "age CDATA
#REQUIRED year CDATA #REQUIRED">

<!ENTITY @ elemType1 @personType; "(firstname,name)" "age
CDATA #REQUIRED">
```

Sintassi: `<!ENTITY @ nome tipoBaseRestr "contentModel" listaAttrib>`

tipoBaseRestr: tipo da derivare per restrizione;

Esempio:
```
<!ENTITY @ personType "(firstname,secondname)">
<!ENTITY @ newpersonType @personType;
"(firstname)">
```

4.3 Content Model

DTD++ mira ad importare le funzionalità avanzate di XML Schema nella dichiarazione del content model. Gli strumenti attraverso i quali si è perseguito l'obiettivo sono nuove forme di vincoli, un nuovo content model group e un nuovo tipo di content model misto.

4.3.1 Content model misto

I DTD classici permettono di dichiarare un solo tipo di content model misto in cui la parola chiave "#PCDATA" compare in prima posizione all'interno di un costrutto di tipo *choice.*

```
<!ELEMENT elem (#PCDATA | a | b | c)*>
```

DTD++, invece, cerca di emulare la funzionalità di XML Schema, che consente ai caratteri di essere presenti in ogni posizione, e non solo nella prima.
La sintassi classica di DTD è stata modificata solo leggermente, spostando #PCDATA al di fuori dell'espressione.
Perciò:

```
<!ELEMENT elem (#PCDATA | a | b | c)*>
```

Diventa:

```
<!ELEMENT elem (#PCDATA ( a | b | c)*)>
```

Sintassi: <!ELEMENT nome (#PCDATA *contentmodel*)

Esempio: <!ELEMENT person (#PCDATA (firstname,secondname))>

4.3.2 Content model group AND

I DTD++ permettono di instanziare un nuovo tipo di content model, oltre ai classici sequence e choice: si tratta del costrutto AND, che mira alla integrazione del content model complesso *all* di Xml Schema. Viene dichiarato attraverso il connettore “&”:

```
<!ELEMENT elem (a & b & c)>
```

Gli elementi del content model possono comparire in qualsiasi posizione dell’istanza di documento, nel rispetto dell’indicatore di occorrenza “?”.

4.3.3 Vincoli di occorrenza

I vincoli di occorrenza sono stati introdotti nei DTD++ prendendo come spunto gli attributi *minOccurs* e *maxOccurs:* infatti all’interno di un content model è possibile dichiarare la presenza di un particolare elemento inserendovi accanto due numeri compresi tra parentesi quadre.

```
<!ELEMENT elem (a[0,3]b+,c)?>
```

Questi due numeri specificano il numero minimo e massimo di volte in cui l’elemento può essere ripetuto. Possono essere usati anche per specificare solo uno dei valori:

```
<!ELEMENT elem (a[1,],b+,c)?>
```

L’elemento *a* deve comparire minimo una volta.

```
<!ELEMENT elem (a[,5],b+,c)?>
```

L’elemento *a* deve comparire al massimo 5 volte.

Questa particolare sintassi non esclude l’uso del metodo classico di DTD per specificare l’occorrenza. È possibile, infatti, utilizzare gli operatori “+”,”?”,”*”): perciò la dizione [1,] equivale a “+”; [0,] equivale a “*”; [0,1] equivale a “?”.

4.4 Namespace

DTD++ introduce delle dichiarazioni per riproporre i meccanismi offerti da Xml schema nel trattare le dichiarazioni di namespace.

4.4.1 Dichiarazione di targetNamespace

Un DTD++ può essere visto come una collezione di dichiarazioni di elementi, i cui nomi appartengono ad un determinato *namespace,* il *targetNamespace.* Questo concetto proviene da XML Schema, con una modifica sostanziale. In DTD++ il *targetNamespace* non è limitato ad un unico valore e può essere specificato attraverso una forma di dichiarazione introdotta nell’estensione dei DTD. La nuova funzionalità di DTD++ consente di dichiarare una serie di *namespace* diversi, ognuno dei quali identificherà un insieme di elementi a seconda del prefisso associato. Gli elementi sono gli unici valori cui potrà essere associato un prefisso che ne caratterizza il *namespace* di appartenenza.

Sintassi: `<!TARGETNS prefisso "URI">`
prefisso: prefisso associato al namespace

Esempio: `<!TARGETNS ns "http://www.namespace.it">`

Questo costrutto ha le seguenti particolarità:

- in ogni DTD++ può essere dichiarato, ma non è obbligatorio;
- in ogni DTD++ può esserci una sola dichiarazione di TARGETNS senza prefisso;
- quando vengono specificati altri namespace, attraverso TARGETNS, dovranno venire associati prefissi identificativi a questi valori: in questo modo ogni elemento apparterrà al namespace dichiarato nel targetNamespace con il prefisso corrispindente;
- gli elementi senza prefisso apparterranno al namespace senza prefisso;

Questo costrutto consente di dichiarare nello stesso DTD elementi appartenenti a namespace differenti, cosa impossibile sia in schema che in DTD classico. Il DTD++ non è più vincolato ad un'unica radice: qualsiasi elemento potrà essere radice del documento XML.

4.4.2 Content ANY

La dichiarazione di content model *ANY* proviene da XML schema, più precisamente dall'omonimo elemento *any.* La sintassi è quella classica dei DTD, a cui si aggiungono le dichiarazioni relative al numero di elementi presenti e il namespace da cui derivano, che vengono espresse attraverso i vincoli di occorrenza visti precedentemente.

Sintassi:
```
<!ELEMENT nome ANY[,]{}>
<!ENTITY @ nome "ANY[,]{}" listaAttr>
```

Esempio:
```
<!ELEMENT person ANY[1,]{##any}>
```

```
<!ENTITY @ personType "ANY[1,]{##any}"
"age (CDATA) #IMPLIED">
<!ELEMENT person ANY{"http://www.target.it"}>
```

Non tutti i valori sono possibili tra parentesi graffe, ecco quelli ammessi:

- ##any: ogni elemento che appartiene ad uno dei namespace dicharati nel DTD++;
- ##other: ogni elemento che appartiene a tutti i namespace tranne il targetNamespace dell'elemento in cui è stato definito;
- ##local: sono ammessi solo elementi che non appartengono ad alcun namespace;
- ##targetNamespace: ciascun element che appartiene al targetNamespace dell'elemento in cui è stato definito;
- valore: la lista dei namespace, separati da uno spazio, da cui vengono importati gli elementi.

4.4.3 Document Type Declaration

Queste nuove dichiarazioni sono state pensate per essere racchiuse, insieme alle dichiarazioni classiche, in file con estensione ".dpp"; a tali file ci si riferisce nel documento con la stessa dichiarazione di DOCTYPE usata per i DTD classici:

```
<!DOCTYPE root SYSTEM "file.dpp">

<!DOCTYPE ns:root SYSTEM "file.dpp">
```

5 PREVALIDATOR FOR DTD++

La scelta tra l'utilizzo di DTD o di Schema va ponderata con molta attenzione. DTD non permette di dichiarare molti dei costrutti tipici di XML Schema, limitandone quindi la capacità espressiva, fornisce invece i costrutti per la dichiarazione di entità generali e parametriche, non consentite in XML schema.
D'altro canto, se è necessaria una certa espressività nel linguaggio, la scelta di utilizzare XML Schema diventa obbligatoria. XML Schema, però, obbliga ad un lavoro maggiore dovuto alla complessità del linguaggio. Infatti, un documento XML Schema può essere cinque/dieci volte più lungo del corrispettivo documento DTD.
La definizione di un nuovo linguaggio, con i pregi di entrambi, è sicuramente il punto di partenza per risolvere il problema.
DTD++ è un'estensione sintattica di DTD che ne estende l'espressività andando a colmare il divario con XML Schema.
Un nuovo linguaggio, però, è sicuramente necessario ma non sufficiente. C'è generalmente una certa reticenza nell'abbandonare le vecchie tecnologie per abbracciarne di nuove. La scelta di investire del tempo, e di conseguenza del denaro, nello studio e nell'utilizzo di un nuovo linguaggio è una cosa che bisogna ponderare sempre molto attentamente. Sia DTD che XML Schema si sono affermati ormai come standard e non è facile abbandonarli per un nuovo linguaggio la cui adozione, nel caso non dovesse avere il successo sperato, porterebbe ad una spesa non indifferente.
L'utilizzo di tecnologie non convenzionali porterebbe inoltre ad aver problemi di interscambio documenti con strutture esterne. Se nella mia azienda decido di utilizzare DTD++ non è detto che le aziende con cui lavoro decidano di farne altrettanto. Se avviene uno scambio di documenti XML che devono essere validati, non si possono imporre le

proprie tecnologie e i propri metodi di lavoro a terzi: perciò si sarà costretti a riscrivere in XML Schema l'intero documento di validazione.
Inoltre, sul mercato ci sono già numerosi validatori, testati e ben funzionanti e numerose librerie che si interfacciano con DTD e XML Schema.

5.1 Soluzioni

La soluzione definitiva sta nella realizzazione di un programma che utilizzi le tecnologie che già sono in circolazione, che permetta di recuperare, eventualmente, il lavoro fatto, e di risparmiare tempo nella realizzazione di documenti di validazione.
Sfruttando la semplicità del DTD, DTD++ permette, con poche righe, di scrivere regole di validazione molto complesse e con un espressività equivalente a quella di XML schema.
Da queste considerazioni nasce l'idea di realizzare un preprocessore che trasformi il DTD++ in un XML Schema equivalente e utilizzi poi un validatore schema per validare il documento XML.
Il programma deve inoltre avere la possibilità di esportare lo schema generato in un formato ben leggibile e facilmente modificabile. Ciò permette di utilizzare lo schema per scopi diversi da quello della validazione: per esempio per condividere la struttura di documenti XML con persone che non utilizzano DTD++.

5.2 PreValidator for DTD++

PreValidator for DTD++ è un preprocessore per la validazione di documenti XML con documenti DTD++. Come suggerisce il nome, esegue le operazioni precedenti alla fase di validazione del documento.
Il flusso del programma è il seguente:

- Prende in input i nomi dei file dei documenti XML e DTD++;

- processa il documento DTD++ realizzandone una versione XML Schema equivalente e salvandola sul file system in un file temporaneo;

- crea, se necessario, un documento XML temporaneo modificato opportunamente, inserendo i riferimenti allo Schema e le dichiarazioni d'entità;

- richiama un validatore schema esterno affinché validi il documento XML con lo schema generato;

- intercetta gli errori restituiti dal validatore ed effettua un'operazione di trasformazione dei riferimenti dal documento Schema generato, e utilizzato dal validatore, al documento DTD++ originale.

Il preprocessore non effettua controlli sulla validità dello schema generato, che vengono, invece, lasciati interamente al validatore esterno. Questo permette la massima flessibilità d'utilizzo. L'unico vincolo sta nell'obbligo di definizione delle entità parametriche che vengono processate dal preprocessore in quanto non previste da XML Schema.
Il preprocessore è stato scritto interamente in java, per permetterne la massima portabilità, e senza appoggiarsi a tecnologie esterne, se non quelle per la realizzazione del parser, per poter avere un controllo completo del flusso di esecuzione ed alcuna limitazione sull'implementazione delle funzionalità.
PreValidator for DTD++ può anche essere utilizzato senza l'ausilio di un validatore. Le funzionalità, in questo caso, si limitano alla generazione di uno Schema corrispondente al documento DTD++.

5.3 Pregi e vantaggi

Sposare una nuova tecnologia, come già detto, comporta sempre una serie di valutazioni su quelli che potrebbero essere i vantaggi e gli svantaggi che accompagnano la scelta.
Le esigenze variano in base alla situazione e al campo di applicazione, ciò che per qualcuno è di fondamentale importanza, per esempio la condivisione dei documenti con terzi, per altri è completamente irrilevante.
L'utilizzo del preprocessore e di DTD++ porta sicuramente ad una serie di vantaggi in grado di soddisfare le esigenze più varie:

- **Un apprendimento veloce del linguaggio.**
  Essendo DTD++ un'estensione sintattica di DTD è sufficiente imparare i nuovi costrutti per utilizzare immediatamente il linguaggio e creare documenti di validazione.

- **Velocità nello scrivere documenti.**
  La sintassi essenziale permette di scrivere documenti DTD++ molto velocemente senza perdere tempo inutile nella ripetitiva apertura e chiusura dei tag XML Schema.

- **Semplicità e brevità nei tag.**
  Questa caratteristica permette di visualizzare sempre un maggior numero di tag rispetto ad un equivalente documento scritto in XML Schema. Ciò rende più facile la correzione di eventuali errori e la possibilità apportare velocemente delle modifiche al documento.

- **Espressività.**
  La grande capacità espressiva di DTD++ permette di scrivere documenti di validazione molto dettagliati e con un concetto di

tipizzazione molto forte. La possibilità di utilizzare e derivare i tipi predefiniti di XML Schema permette di definire con grande precisione qualsiasi tipo.

- **Utilizzare tutti i software in cui viene utilizzato Schema.**
  Apportando semplici modifiche al preprocessore è possibile utilizzare qualsiasi software che utilizza un documento XML Schema come file di input. Si possono, per esempio, utilizzare software per la generazione automatica di classi C++.

- **Possibilità di esportare il documento in XML Schema.**
  La possibilità di trasformare il documento DTD++ in XML Schema lascia sempre una porta aperta verso lo standard. Ciò permette per esempio di utilizzare DTD++ per un uso interno ma di pubblicare per un utilizzo pubblico una versione in XML Schema.

- **Validazione effettuata con programmi testati.**
  Il processo di validazione viene delegato a software già largamente utilizzati e testati. Inoltre l'eventuale aggiunta di costrutti nel DTD++ viene risolta semplicemente con la generazione dello schema corrispondente, senza doversi preoccupare del complesso processo di validazione.

5.4 Limiti

PreValidator for DTD++ è alla sua prima versione e ciò ne comporta alcuni limiti sull'utilizzo. Si cercherà di porre rimedio a queste lacune con le prossime versioni del software.

- **Gestione cache.**
  La mancanza di una gestione cache dei documenti processati crea la possibilità di perdita di tempo nella validazione dei documenti nel caso di DTD++ molto complessi. Per l'utilizzo da parte di utenti, il problema non è molto rilevante in quanto il processore, per un documento molto complesso, impiega qualche secondo per generare lo schema. L'utilizzo su sistemi automatici che effettuano continuamente delle validazione potrebbe, invece, portare velocemente alla saturazione del processore.

- **Sistema d'interfaccia per la chiamata ai validatori.**
  Il preprocessore può utilizzare qualsiasi validatore esterno ma, per evocarlo, utilizza una sintassi predefinita (*nomeProgramma file.xml file.xsd*). Per utilizzare il software esterno aggiungendo, per esempio, dei parametri è necessario intervenire direttamente nel codice.

- **Sistema flessibile per il ritorno degli errori.**
  Lo stesso problema riscontrato nella gestione delle chiamate ai software esterni lo ritroviamo nella gestione degli errori. Attualmente gli errori vengono interpretati e riportati al DTD++ originale solo se in un particolare formato. Se il formato di risposta cambia è necessario intervenire sul codice.

5.5 DTD++ for Xerces

Un parser validante per DTD++ è stato già realizzato[Amo02]. Il parser è stato creato avvalendosi del parser Xerces Java 2. Xerces Java è un progetto open-source che nasce come diretto discendente di XMLJ4 di IBM e si prefigge l'obiettivo di creare un parser validante per documenti

XML. Xerces Java fa parte del progetto Apache[1]. La prima versione di Xerces Java, la 1.x, forniva i primi supporti di base per XML Schema e un completo meccanismo di validazione dei DTD, basato sull'utilizzo di DOM e SAX. La versione 2.x è in grado di validare qualsiasi vincolo, o struttura, espressa in uno schema.
Per la realizzazione del parser sono stati modificati i vari moduli di Xerces dedicati allo scanning del DTD classico per dare spazio alle nuove dichiarazioni del DTD++. In questo modo si arriva a generare una grammatica DTD++, e da essa lo schema per validare il documento XML istanza del DTD++.
Le modifiche apportate al programma sono sostanzialmente di due tipi:

- per analizzare le nuove dichiarazioni;
- per salvare i dati.

In più è stato creato il parser vero e proprio che, a partire dai dati salvati nelle strutture, genera l'intero DOM dello schema corrispondente alla grammatica estesa. Le strutture dichiarate con i DTD++ sono equivalenti alle dichiarazioni di XML Schema e da ciascuna collezione di dati si genera la corrispettiva dichiarazione XML Schema.
Una volta generato il documento XML Schema il programma è pronto per effettuare la validazione del documento XML.

5.6 Confronto tra le architetture

La differenza sostanziale tra DTD++ for Xerces e PreValidator for DTD++ consta nelle tecnologie utilizzate per la realizzazione.

[1] http://xml.apache.org

DTD++ for Xerces, come detto in precedenza, è stato realizzato apportando delle modifiche al parser Xerces Java 2 mentre PreValidator for DTD++ non si appoggia su validatori preesistenti.
Il flusso delle due applicazioni è molto simile in quanto, anche se con processi diversi, entrambi effettuano un parsing del DTD++ per poter poi realizzare una trasformazione XML Schema equivalente.
Entrambi sono stati scritti interamente in java per potersi avvalere della massima compatibilità fra i vari sistemi operativi.
La scelta di modificare Xerces ha lo svantaggio di dover sottostare a quelle che sono le limitazioni poste da un'architettura già esistente, obbligando alcune scelte implementative per non perdere i vantaggi forniti dall'architettura stessa. Inoltre le modifiche effettuate sul codice sorgente di Xerces non saranno disponibili per le versioni successive del parser.
Resta comunque l'indiscutibile vantaggio di aver utilizzato un'architettura testata e ben funzionante, supportata da una vasta comunità di programmatori open-source.
La realizzazione ex-novo del software, invece, trova proprio in questo suoi punti deboli. Il fatto di dover affrontare da zero certe problematiche rende sicuramente il software meno esente da bug.
I vantaggi di PreValidator for DTD++, però, sono notevoli. Innanzi tutto un completo controllo sul flusso di esecuzione, che permette di gestire al meglio gli errori e ottimizzare il codice. Inoltre, l'utilizzo di un validatore esterno da la possibilità di utilizzare le migliori tecnologie disponibili senza dover apportare delle pericolose modifiche al codice.

5.7 MathML

Al fine di avere un applicazione pratica d'esempio per il preprocessore è stata realizzata una versione di MathML scritta in DTD++.
MathML (Mathematical Markup Language) è la prima applicazione rilasciata come raccomandazione dal W3C [ABCD03].

MathML è una specifica di basso livello per descrivere espressioni matematiche per la comunicazione tra processi. Il suo scopo è quello di facilitare l'uso di contenuti matematici e scientifici sul web. Con un adeguato supporto degli style sheet, sarà possibile per i browser effettuare nativamente il rendering delle espressioni matematiche.
La realizzazione del documento DTD++ è stata effettuata prendendo come riferimento XML Schema del MathML 2.0 realizzata da Stephane Dalmas presso l'INRIA[2].

Il documento originale è composto da 29 file XML Schema suddivisi nelle tre categorie common, presentation, e content. Questi file vengono inclusi nel documento principale mathml.xsd grazie all'utilizzo del tag schema <xsd:include>.
Il documento che contiene l'elemento di root è il file common/math.xsd, l'elemento math.
Per realizzare il documento sono state trasformate le definizioni in Schema nelle rispettive DTD++.
Lo Schema originale fa un largo uso dell'elemento group che non ha un corrispettivo diretto nel DTD++. Per ovviare a questo problema sono state utilizzate le entità parametriche. L'uso delle entità parametriche associato ad una non completa implementazione delle entità globali nel preprocessore ha impedito di realizzare la versione DTD++ del MathML rispettando la struttura originale.
Il MathML è stato quindi riscritto in un unico file contenente tutte le dichiarazioni.

Di seguito alcuni esempi su come le trasformazioni sono state applicate.
L'esempio si basa sulla codifica dell'elemento math e comprende tutte le tipologie di elementi riscontrati.

---
[2] http://www.w3.org/Math/XMLSchema/mathml2/mathml2.xsd

Per ogni element *nomeElement* definito all'interno dello schema sono state seguite le seguenti direttive:

- definizione degli attributi tramite un elemento di tipo attributeGroup con nome *nomeElement.attlist*,

- definizione del content model, se presente, per mezzo di un elemento di tipo group con nome *nomeElement.content*,

- definizione del tipo complesso con un elemento di tipo complexType con nome *nomeElement.type* che fa riferimento a *nomeElement.attlist* e *nomeElement.content*,

- definizione dell'elemento element di nome *nomeElemento* type *nomeElemento.type*.

Per rispettare la struttura originale dello schema, che associa l'attribute group al complex type e non all'element è necessario definire delle entità parametriche al posto delle attlist.

```
<xs:attributeGroup name="math.attlist">
  <xs:attributeGroup ref="Browser-interface.attrib"/>
  <xs:attribute name="macros" type="xs:string"/>
  <xs:attribute name="display" default="inline">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="block"/>
        <xs:enumeration value="inline"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attributeGroup ref="Common.attrib"/>
</xs:attributeGroup>
```

```
<!ENTITY % math.attlist "
  %Browser-interface.attrib;
```

```
  macros  (#STRING)
  display (#STRING(block|inline)) 'inline'
  %Common.attrib;
  ">
```

Come già detto in precedenza DTD++ non prevede un tag corrispondente al group dell'XML Schema, l'elemento group, come per le attlistgroup, diventa quindi un'entità parametrica.

```
<xs:group name="math.content">
  <xs:choice>
    <xs:group ref="PresExpr.class"/>
    <xs:group ref="ContExpr.class"/>
  </xs:choice>
</xs:group>

<!ENTITY % math.content
"%PresExpr.class;|%ContExpr.class;">
```

I tipi complessi vengono invece trasformati utilizzando il tipo complesso di DTD++.

```
<xs:complexType name="math.type">
  <xs:group ref="math.content" minOccurs="0"
                         maxOccurs="unbounded"/>
  <xs:attributeGroup ref="math.attlist"/>
</xs:complexType>

<!ENTITY @ math.type "(%math.content;)*" "%math.attlist;">
```

Come per il tipo complesso anche l'element viene trasformato utilizzando il rispettivo tipo del DTD++

```
<xs:element name="math" type="math.type"/>

<!ELEMENT math (@math.type;)>
```

Nello schema vengono definiti anche alcuni sympleType con facet di tipo enumeration e pattern. I rispettivi elementi DTD++ vengono implementati utilizzando i costrutti dei tipi semplici.

```
<xs:simpleType name="thickness">
  <xs:restriction base="xs:string">
    <xs:enumeration value="thin"/>
    <xs:enumeration value="medium"/>
    <xs:enumeration value="thick"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="length-with-unit">
  <xs:restriction base="xs:string">
    <xs:pattern
     value="(-?([0-9]+|[0-9]*\.[0-
           9]+)*(em|ex|px|in|cm|mm|pt|pc|%))|0"/>
   </xs:restriction>
</xs:simpleType>
```

```
<!ENTITY # thickness "(#STRING(thin|medium|thick))">
<!ENTITY # length-with-unit "(#STRING/(-?([0-9]+|[0-
9]*\.[0-9]+)*(em|ex|px|in|cm|mm|pt|pc|%))|0/)" >
```

Unica differenza strutturale tra la versione Schema e la versione DTD++ è dovuta ad una limitazione di quest'ultimo. In DTD++, infatti, non è possibile definire elementi di tipo anyAttribute. Questi vengono utilizzati dal MathML nella definizione dell'attributeList `common.attrib`.

```
<xs:attributeGroup name="Common.attrib">
  <xs:attribute name="class" type="xs:NMTOKENS"/>
  <xs:attribute name="style" type="xs:string"/>
  <xs:attribute name="xref" type="xs:IDREF"/>
  <xs:attribute name="id" type="xs:ID"/>
  <xs:anyAttribute namespace="##other"
                     processContents="skip"/>
</xs:attributeGroup>
```

In questo paragrafo sono state presentate le implementazioni di alcuni dei costrutti del MathML. La versione completa del DTD++ si trova nell'appendice B.

5.8 Limiti del linguaggio

La scrittura del documento DTD++ relativa al MathML ha fatto emergere delle lacune nella definizione del linguaggio.

Alcune di queste sono aggirabili utilizzando altri costrutti del DTD++, altre invece impongono delle limitazioni all'espressività del linguaggio.

Abbiamo visto come la mancanza di un elemento di tipo group sia facilmente risolvibile utilizzando le entità parametriche. Questa soluzione, però, si ripercuote sulle dimensioni del documento XML Schema generato. La versione Schema di MathML generata dal preprocessore ha dimensioni tre/quattro volte superiori a quella originale, e questa espansione è dovuta quasi integralmente all'assenza dell'elemento group.

Altra limitazione, che viene risolta con l'utilizzo delle entità parametriche, è l'impossibilità di associare un attributeGroup ad un tipo complesso o ad un element in maniera esplicita. Il DTD++ associa implicitamente, come viene definito nel linguaggio DTD, un attributeGroup ad un element che possiede lo stesso nome.

Ciò permette inoltre di associare ad un element solamente un attributeGroup.

La mancanza di un elemento di tipo anyAttribute, invece, pone una grande limitazione espressiva al linguaggio. Non c'è modo, con gli attuali costrutti del DTD++, di aggirare questo problema.

5.9 Funzioni non implementate nel preprocessore

Alcune funzionalità non sono state implementate in maniera completa.

Il preprocessore non gestisce al meglio le entità parametriche esterne. Il problema si pone nel momento in cui all'interno del file importato ci siano delle definizioni di entità parametriche, queste non vengono elaborate dal parser e quindi inserite nella struttura dati.
Inoltre non vengono gestite le entità esterne e il tag NOTATION.
La gestione del file XML è al momento ridotta al minimo indispensabile. Il preprocessore si limita, nel qual caso fosse richiesto, ad aggiungere un riferimento all'XML Schema generato all'interno del primo nodo del file XML. Ciò avviene solamente se all'interno del tag già non esiste un proprietà di tipo *xmlns:xsi*

5.10 Implementazioni future

La realizzazione di un sistema di caching dei documenti XML Schema generati è sicuramente la più importante delle funzionalità da implementare. Ciò non si deve limitare ad un salvataggio sul file system del file XML Schema, ma anche ad una serializzazione della struttura dati, necessaria per la gestione dei messaggi d'errore.
Lo studio, e la realizzazione, di un sistema programmabile per l'interfacciamento con i validatori esterni renderebbe PreValidator for DTD++ un software utilizzabile con qualsiasi applicazione che faccia uso di XML Schema. Ciò porterebbe il preprocessore ad un utilizzo che va oltre la validazione dei documenti e permetterebbe di utilizzare DTD++ come perfetto sostituto di XML Schema.
Si potrebbero, inoltre, aggiungere altre funzioni per la generazione del codice, che permetterebbero, quindi, di ottenere documenti in un formato diverso da XML Schema. Sarebbe possibile generare i documenti per la validazione in altri linguaggi e ottenere, con piccole modifiche, una versione DTD del documento DTD++. Oltre a documenti per la validazione, si potrebbe utilizzare come generatore di codice automatico per java o c++, tanto per fare un esempio.

6 PREVALIDATOR FOR DTD++: DETTAGLI IMPLEMENTATIVI

In questo capitolo viene descritta in dettaglio l'implementazione del preprocessore per la validazione di documenti XML con documenti DTD++.
Il preprocessore è stato sviluppato interamente utilizzando il linguaggio Java per permetterne la portabilità.
Non effettua la validazione del documento ma si limita a generare XML Schema relativo al DTD++ e a richiamare un validatore esterno riportandone poi gli errori. Come validatore esterni sono stati utilizzati MSXML e Xerces, ma con semplici modifiche è possibile utilizzare qualsiasi altro validatore.

6.1 Flusso del programma

Il file DTD++ viene caricato dal disco e passato alla classe DppDocManager che indicizza il file per righe. Il documento viene successivamente elaborato dal lexer che ne estrae le entità parametriche.
Le entità parametriche vengono risolte internamente e successivamente vengono sostituite nel documento. Ci si trova quindi con un documento privo di entità parametriche che viene passato al parser. Al termine del parsing, il parser restituisce un vettore contenente la lista dei tag.
Dalla lista dei tag vengono estratti i tipi e le attributeList. Quindi vengono associate le attributelist agli elementi relativi.
A questo punto la struttura è pronta per emettere la trasformazione in schema del documento.
La lista dei tag viene scorsa e ogni singolo tag emette il proprio schema.

Una volta che lo schema è stato generato viene chiamato il validatore esterno e validato. Gli errori generati dal validatore vengono intercettati dal preprocessore che, tramite il DppDocumentManager risolve i numeri di linea e restituisce gli errori sullo standard output con i numeri di riga relativi al dpp.

6.2 Tecnologie utilizzate

Per la realizzazione del preprocessore sono stati utilizzate tecnologie esterne per il parsing del documento e per la validazione. Per il parsing sono stati utilizzati Jflex e JavaCup, per la validazione MSXML e Xerces.

6.2.1 JFlex

JFlex[3] è un generatore di analizzatore lessicale per java, scritto in java.
È una reimplementazione di *jlex*, un tool molto comune realizzato da Elliot Berk dell'Università di Princeton.
JFlex è stato realizzato per lavorare con Java Cup.
Inizialmente il lexer scelto per la realizzazione del preprocessore era stato Jlex ma le features di JFlex, in particolar modo la miglior gestione delle espresioni regolari e degli errori, lo hanno fatto preferire.

6.2.2 JavaCup

JavaCup[4] (Constructor of Useful Parsers) è un sistema per la generazione di parser LARL. Serve lo stesso ruolo del largamente usato YACC e ne offre la maggior parte delle features.
Cup è scritto in java e produce un parser implementato in java. Cup utilizza JFlex per creare i token.

---

[3] http://jflex.de/
[4] http://www.cs.princeton.edu/~appel/modern/java/CUP/

6.2.3 MSXML

MSXML[5] è una API di Microsoft basata su COM per parsare e processare documenti XML. La versione corrente di questa libreria include il supporto per DOM Level 2.0, SAX 2.0, XPath, XSLT, e XSD Schema.
Per utilizzare MSXML è stata realizzata una semplice applicazione in C# che effettua l'operazione di validazione.
Prende in input un file XML e un file XML Schema, effettua la validazione utilizzando le classi XmlSchemaCollection e XmlValidatingReader e restituisce sullo standard error gli eventuali errori.

6.2.4 Xerces

Xerces Java è un progetto open-source che nasce come diretto discendente di XMLJ4 di IBM e si prefigge l'obiettivo di creare un parser validante per documenti XML. Xerces Java fa parte del progetto Apache[6]. La prima versione di Xerces Java, la 1.x, forniva i primi supporti di base per XML Schema e un completo meccanismo di validazione dei DTD, basato sull'utilizzo di DOM e SAX. La versione 2.x è in grado di validare qualsiasi vincolo, o struttura, espressa in uno schema. Per utilizzare Xerces come validatore è stato utilizzato il GTRI Validation Tool 1.1. Il tool è stato realizzato dalla Georgia Tech Research Corporation scritto interamente in java. È fondamentalmente un programma che si limita a rendere Xerces un validatore XML stand-alone.

6.3 Il programma

Il programma è strutturato all'interno del package preValidator che contiene la main class PreValidator.class e sei package:

---

[5] http://msdn.microsoft.com/library/en-us/dnanchor/html/anch_xmlprod.asp
[6] http://xml.apache.org

- preValidator.documents: contiene le classi che gestiscono i file DTD++ e XML
- preValidator.types: contiene le classi che gestiscono i tipi del documento DTD++.

- preValidator.errors: contiene le classi che gestiscono I messaggi di errore

.

- preValidator.parsers: contiene le classi che effettuano lo scanning e il parsing.

- preValidator.parserStruct: contiene le classi che gestiscono la strutturazione e rappresentazione del documento da generare

- preValidator.utils: contiene una serie di classi di varia utilità.

6.3.1 preValidator

Il package principale contiene la main class PreValidator.

6.3.1.1 PreValidator

PreValidator è la main class del programma. Il metodo pubblico *start*, che prende in input la lista degli argomenti, effettua tutte le chiamate alle operazioni necessarie al processo di trasformazione e validazione dei documenti. Una volta processati gli argomenti, per mezzo del metodo *getArgs,* inizializza il DppDocumentManager e le struttura dati relativa alla gestione degli errori Quindi richiama il metodo *process*, il quale effettua l'operazione di parsing e generazione dello schema. Una volta generato lo Schema prepara, se necessario, una versione modificata del documento XML e chiama il metodo *Validate* per la

validazione del documento. Al termine elimina i file temporanei e stampa sullo stdout il documento XML Schema, se richiesto.

6.3.2 preValidator.documents

Il package preValidator.documents contiene le classi che gestiscono i documenti.

6.3.2.1 DppDocManager

Realizza la gestione del file dpp. Preso in input il documento lo separa, tramite il metodo *process*, per linee utilizzando un vettore di LineItem. Con il metodo *substitute* effettua per ciascuna linea la sostituzione delle entità parametriche. Al termine di queste operazioni il documento è pronto per essere passato al parser che creerà il DOM dal quale otterremo lo schema.

Il metodo *getPostLine* implementa la funzionalità per ottenere i riferimenti ai numeri di riga del documento originale utilizzati dal preprocessore per elaborare gli errori restituiti dal validatore schema.

6.3.2.2 DppLineItem

Gestisce il riferimento tra i numeri di riga attraverso i vari stadi di trasformazione del documento.

6.3.2.3 XmlDocManager

Realizza la gestione del file XML che si vuole validare. Per mezzo del metodo *setSchemaRef* genera una versione modificata dell'XML andando ad inserire gli attributi xmlns:xsi e xsi:noNamespaceSchemaLocation all'interno del primo tag XML.

6.3.3 preValidator.types

Il package preValidator.types contiene le classi che gestiscono i tipi del documento DTD++.

6.3.3.1 TypeList

Gestisce la lista dei tipi e degli AttList. Ad ogni tag estratto dal dpp corrisponde un tipo che viene aggiunto alla lista tramite il metodo *addType*. Se il tag corrisponde ad un AttList si utilizza invece il metodo *addAttGroup*. I tipi e gli AttList vengono archiviati all'interno di due hashTable. Una volta riempita la struttura dati tramite il metodo *linkGroup* si eseguono le associazioni tra gli AttList e i relativi tipi.
Per effettuare le ricerche all'interno della struttura sono stati implementati i metodi: *getType*, permette di estrarre l'oggetto Type relativo; *getStringType*, restituisce il nome da utilizzare all'interno dello schema; *getBaseType*, restituisce il nome del tipo da cui è derivato; *getOriginalType*, restituisce il nome del tipo base, fra quelli predefiniti da schema, da cui è derivato.
Il costruttore della classe chiama il metodo *initPredefined* che inizializza i tipi predefiniti da schema.

6.3.3.2 Type e classi derivate

Definiscono i tipi dei tag. Contengono il nome del tag, la stringa da inserire nella generazione dello schema.

6.3.3.3 ParametricList

Classe che gestisce le entità parametriche. Attraverso il metodo *put* le entità parametriche vengono inserite nella lista specificandone la chiave, i valori, il tipo e la posizione all'interno del file di origine.

Il metodo *adjustList* effettua le sostituzioni all'interno della lista stessa. Al termine di questa operazione la lista, salvo errori nel dtd++, dovrebbe essere priva di entità parametriche al suo interno.
Tramite il metodo *getValue*, data una chiave e una posizione di fine ricerca, è possibile ottenere il valore dell'entità parametrica ricercata, restituisce *null* altrimenti.

6.3.4 preValidator.errors

Il package preValidator.errors contiene le classi che gestiscono i messaggi di errore.

6.3.4.1 ErrrorValue

Classe che definisce l'errore. Contiene i metodi per la stampa dei messaggi d'errore. Tramite i costruttori è possibile costruire l'errore definendone il tipo, la posizione di inizio e di fine dell'errore, una serie di valori da visualizzare al momento della stampa e un livello di criticità dell'errore.
Tramite il metodo *toString*, chiamato con un riferimento al documentManager, si ricava la stringa d'errore con un preciso riferimento alla riga nel quale questo errore è avvenuto.

6.3.4.2 ErrorType

Definisce le varie tipologie d'errore attraverso una serie di valori statici.

6.3.4.3 ErrorList

Gestisce la lista degli errori. È possibile aggiungere un ErrorValue tramite il metodo *addElement*. Il metodo *size* ritorna il numero di errori mentre *clear* azzera lo stato della lista. Il metodo toString crea una string contenente lo stato d'errore degli ErrorValue contenuti nella lista.

6.3.4.4 ElaborateError

Classe astratta per l'elaborazione dei messaggi d'errore restituiti dal validatore.

Le classi che ne derivano devono implementare un metodo *ElaborateError* , metodo che esegue l'operazione di trasformazione. Nella versione corrente sono presenti due classi derivate da ErrorConverter: ErrorConverterMSXML e ErrorConverterXerces. La prima effettua la trasformazione richiesta mentre la seconda, non ancora implementata correttamente, restituisce l'errore così come l'ha ricevuto in input.

6.3.5 preValidator.parsers

Il package preValidator.parsers contiene le classi che effettuano lo scanning e il parsing. Queste classi sono generate automaticamente grazie a Jflex e JavaCup. I file contenenti le regole per la generazione delle classi sono:

- PerLex.flex, genera le classi per la gestione delle entità parametriche;

- parser.flex, genera le classi che generano i token per il parser;

- parser.par, genera le classi che costituiscono il parser.

6.3.6 preValidator.parserStructs

Il package preValidator.parserStruct contiene le classi che gestiscono la strutturazione e rappresentazione del documento da generare.

6.3.6.1 Tags

Classe contenitore di tutti i tag. Nel campo *items* sono contenuti i tag generati dal parser nell'ordine in cui sono stati trovati nel file originale.

Il campo *declaration* contiene ciò che va posto all'inizio del documento schema e che non ha una relazione.

Il campo *typeList* contiene la lista dei tipi e degli attributi, viene inizializzata tramite il metodo *initiTypeList*.

I metodi *getDocType* e *getNameSpace* restituiscono rispettativamente il valore da inserire all'interno del tag DOCTYPE dello schema e il valore della proprietà targetNamespace del tag <schema>.

6.3.6.2 ContentModel

Classe astratta che definisce il content model. Genera, in pratica, la maggior parte dello schema. Le classi Element, Attribute, e le classi che definiscono le entità contengono tutte un content model.

Una serie di campi definiscono i facet del content model:

- cardinality, le occorrenze minime e massime;
- length, la lunghezza minima, massima o esatta dell’elemento, in caso di numeri decimali definisce il numero di cifre intere e decimali;
- range, il valore massimo e minimo, inclusivo e non inclusivo, che il valore può avere;
- pattern, un’espressione regolare;
- enumeration, una lista di valori;
- whitespace, la gestione degli spazi bianchi.

È possibile creare una struttura ad albero attraverso il campo childs. Childs è un vettore che permette di associare al content model altri content model. Se il vettore childs è vuoto è necessario specificare un valore nel campo value. Questo identifica il content model corrente come foglia all'interno dell'albero.
Per aggiungere facet o altri child si utlizza il metodo add. Il metodo sistema automaticamente l'oggetto in base alla sua istanza.
Il metodo setCard permette di ereditare i facet da un altro content model, il metodo non sovrascrive facet già esistenti ma si limita a impostare quelli ancora non definiti.
SetItem permette invece di impostare direttamente il vettore dei childs, viene utilizzato in fase di generazione del DOM dal parser.
Il campo isAnonymous definisce se il CM è di tipo anonimo, se ha quindi un nome. In caso positivo indica che al momento della generazione dello schema deve incapsulare tutta la generazione all'interno di un tag <SimpleItem> se di tipo semplice e <ComplexType> se di tipo complesso.
Se al content model corrisponde un'AttlistGroup viene definito il campo attGroup

6.3.6.3 SimpleContentModel

Definisce il content model di un tipo semplice. Il campo simpleBaseType definisce il tipo da cui deriva, deve essere sempre definito.
Se il campo isList è impostato a true il content model si comporta come una lista, nel caso semplice il generatore dello schema si limita ad inserire l'intero content model all'interno di un tag <list>. Nel caso più complesso in cui il content model appartenga ad un Element viene generato un <SimpleType> di transizione.

IsEmpty permette di realizzare un SimpleType senza contenuti generando un tag di tipo Empty

6.3.6.4 ComplexContentModel

Definisce il content model di tipo complesso. In base al valore del campo type vengono gestiti i content model contenuti all'interno del vettore childs. CM_TYPE_CHOICE realizza uno schema di tipo choice, CM_TYPE_SEQ di tipo sequenze, CM_TYPE_ALL di tipo all. Se di tipo CM_TYPE_VALUE viene restituito un tag schema di tipo element che riferisce ad un tipo base definito nel campo value. CM_TYPE_ANY realizza uno schema di tipo any, il tipo di any viene registrato nel campo anyType e la lista dei namespace dal vettore anyNameSpaceList.

6.3.6.5 ComplexReferenceContentModel

Definisce un content model che ha come unico elemento un tipo complesso.

Viene utilizzato nelle dichiarazioni di elementi che contengono un'entità complessa.

<!ELEMENT nomeElemento (@tipocomplesso;)>

6.3.6.6 Tag

Classe astratta da cui derivano tutte le classi relative ai tag del DTD++.

I campi *left* e *right* contentgono la posizione iniziale e finale del tag all'interno del documento DTD++.

Il metodo astratto *getSchema* permette, per ciascuna delle classi derivate, di restituire una rappresentazione in XML Schema della propria struttura.

6.3.6.7 AttList

Classe che definisce una lista di attributi. Genera un elemento XML Schema di tipo attributeGroup.

Il campo *attributes* è un vettore di classi Attribute, il campo *name* è una stringa con il nome dell'attributeGroup.

6.3.6.8 Attribute

Classe che definisce un attribute.

Il campo *name* contiene il nome dell'attibuto, *contentModel* il Content model e *defaultValue* il valote di default dell'attributo.

6.3.6.9 AttributeDefault

Classe che definisce il valore di default di un attribute.

Il campo type contiene il tipo di attributo e può avere valori REQUIRED, IMPLIED, FIXED e DEFAULT.

Il valore di default dell'attributo viene invece definito nel campo value.

6.3.6.10 EntityCharRef

Classe che definisce l'entità interna del DTD.

6.3.6.11 EntitySimple

Classe che definisce un'entity simple.

Viene creata indicando il nome e il content model. Se deriva da un'altra entity il metodo *getParent* restituisce una stringa contenente il nome dell'entity da cui deriva. Il content model è definito all'interno del campo *simpleContentModel.*

Il metodo *isUnion* restituisce *true* se il content model è di tipo union.

6.3.6.12 EntityComplex

Classe che definisce un'entity complex.

Il nome dell'entità è contenuto nel campo *name* mentre il vettore *attlist* contiene una lista di oggetti Attribute.

La gestione del content model è delegata interamente al campo *property* di tipo EntityComplexProperty.

6.3.6.13 EntityComplexProperty

Classe che definisce le proprietà di un tipo complesso.

Nel caso in cui il tipo derivi per restrizione o estensione vengono impostati rispettivamente a true i campi *isRestricted* e *isExtended* e viene assegnato il nome del tipo da cui si ereditano le proprietà nel campo *derivedType*. In ultimo il campo *contentModel* contiene il riferimento al content model del tipo.

I costruttori permettono di creare un tipo non derivato o derivato per estensione. Per effettuare una derivazione per restrizione è necessario chiamare successivamente alla creazione dell'oggetto il metodo *setRestriction.*

La classe espone anche un metodo *isMixed* che ritorna true nel caso in cui il content model sia di tipo mixed.

6.3.6.14 FacetCardinality

Classe che definisce il numero di occorrenze del content model.

Genera la stringa "maxOccurs=x minOccurs=y" all'interno dei tag schema.

6.3.6.15 FacetEnumeration

Classe che definisce un facet di tipo Enumeration.

Il campo items contiene la lista dei valori di enumerazione.

6.3.6.16 FacetLength

Classe che definisce un facet per la lunghezza delle stringhe e le caratteristiche dei numeri decimali.

La classe stampa in maniera esclusiva il facet "length" o i facet "minLength" e "maxLenght" o i facet "totalDigits" e "fractionDigits".

6.3.6.17 FacetPattern

Classe che definisce un facet per la definizione dei pattern.

6.3.6.18 FacetRange

Classe che definisce un facet per il range dei numeri.

I valori possono essere di tipo inclusivo o esclusivo.

6.3.6.19 FacetWhiteSpace

Classe che definisce un facet di tipo WhiteSpace.

6.3.6.20 Pair

Classe che rappresenta una coppia di valori separati da un punto o da una virgola. Viene utilizzata dalle classi FacetRange, FacetCardinality e FacetLength per assegnarne i rispettivi valori.

6.3.6.21 Element

Classe che definisce un element.

Il campo *name* contiene il nome dell'elemento, il campo *contentModel* il content model. Se il campo *attGroup* è impostato a true allora l'element fa riferimento ad un'attlistgroup.

Le occorrenze sono definite nel campo *occurs* di tipo FacetCardinality.

6.3.6.22 Comment

Classe che definisce il Tag commento.

Il valore viene inserito comprensivo dei tag di inizio e fine commmento dei DTD <!-- e --> che vengono poi filtrati direttamente dal costruttore.

6.3.6.23 TargetNS

Classe che definisce il Tag TARGETNS.

6.3.7 preValidator.utils

Il package preValidator.utils contiene classi di vario tipo utilizzate dal programma per la stampa e l'accesso al disco.

6.3.7.1 Debug

Classe per la stampa dei messaggi di debug del parser e del lexer

6.3.7.2 Disk

Classe che gestisce gli accessi al disco.

Il metodo *loadFromFile* carica un file di testo dal disco e restituisce in una stringa il suo contenuto.

Il metodo *saveToFile* salva il contenuto di una stringa in un file sul disco.

6.3.7.3 IndentSchema

Classe che gestisce l'indentazione nella generazione dello schema.

Il campo *numTab* tiene memoria del numero di indentazioni in un dato momento. Questo valore viene aumentato o diminuito tramite i metodi *addTab* e *delTab*. Il metodo *getTab* restituisce la stringa da anteporre alla corrente riga dello schema.

6.3.7.4 PrintSchema

Classe di utility per la stampa dello schema. Fornisce un metodo statico *countLine* che conta le righe di una stringa.

6.3.7.5 Verbose

Classe utilizzata per la stampa della modalità verbose.

6.4 Utilizzo e installazione

6.4.1 Utilizzo

L'utilizzo del preprocessore avviene su linea di comando.
L'input dei file avviene dal file system, non è possibile utilizzare lo standard input. L'output viene invece emesso sia su file che sullo standard output, seguendo le indicazioni impostate dagli switch. Gli errori vengono visualizzati sullo standard error.
La sintassi per utilizzarlo è la seguente:

```
java –jar PreValidator.jar  [-v] [-nv] [-schema fileName]
[-out] [-no-validator-error] [-validator fileName] [-
stdout-error] [-xerces] [-msxml] [-no-force-schema] [-no-
delete-temp] [file.xml] <file.dpp>
```

ovvero, più nel dettaglio:

-v — **Verbose mode**. Durante l'esecuzione il preprocessore segnala sullo standard output lo stato in cui si trova. Possono essere specificati anche altri 2 livelli di verbose-mode tramite le opzioni –v5 e –v6. La prima stampa sullo standard output le informazioni di debug del

parser, la seconda sia quelle del parser che quelle dello scanner.

-nv	**Do not validate**. Disabilita l'operazione di validazione, il preprocessore si limita a trasformare il documento DTD++ in XML Schema.
-schema ***fileName***	**Save Schema to file**. Salva la trasformazione del documento DTD++ in XML Schema nel file specificato.
-out	**Print Schema to stdout**. Stampa sullo standard output la trasformazione del DTD++ in XML Schema, questa opzione disabilita l'operazione di validazione.
-no-validator-error	**Do not process validator error**. Disabilita il processing degli errori restituiti dal validatore. Utile nel caso in cui il messaggio d'errore restituito dal validatore non sia compatibile con il preprocessore. L'errore viene stampato sullo standard error così come ricevuto dal validatore.
-validator ***fileName***	**Validator file name**. Determina il validatore da utilizzare nel processo. Se non specificato, il validatore di default si chiama "validator.exe" e si deve trovare nella stessa directory del preprocessore.
-stdout-error	**Error message on stdout**. Imposta il preprocessore in modo che riceva i messagi di

errore del validatore sullo standard output invece che sullo standard error.

-xerces	**Validate with Xerces**. Abilita Xerces come validatore. Equivale ad eseguire il processo con i parametri '-validator "java –jar validateXML-1.1-bin.jar –in" –in –no-validator-error'.
-msxml	**Validate with MSXML**. Abilita MSXML come validatore. Equivale ad eseguire il processo con i parametri '-validator validator.exe'. Questa è anche la configurazione di default.
-no-force-schema	**Do not specify schema**. Non passa il nome dello schema come parametro al validatore ma lo inserisce direttamente nel file XML. In questo caso viene creato un file XML temporaneo.
-no-delete-temp	**Do not delete temp file**. Se abilitato non cancella i file temporanei creati durante l'esecuzione.

file.xml

Il file XML che si vuole validare. L'omissione di questo parametro disabilita l'operazione di validazione. Nel caso non sia stato specificato l'output sul file dello schema generato viene abilitata automaticamente la stampa sullo standard output dello schema.

file.dpp

Il file DTD++ che si vuole processare.

6.4.2 Installazione

Il preprocessore funziona senza particolari requisiti hardware o software. È necessario aver installato una JVM 1.4.1[7] o successiva.

All'interno del pacchetto ci sono tre file:

PreValidator.jar	Il preprocessore
validateXML-1.1-bin.jar	GTRI XML Validation Tool[8]. Il validatore Xerces.
validator.exe	Il validatore MSXML. Per funzionare necessita del runtime di DotNet[9] versione 1.1 e del MSXML 4.0[10]

Per utilizzare il preprocessore è necessario avere tutti i file del pacchetto e i documenti, sia XML che DTD++, all'interno della stessa directory. Inoltre tale directory deve essere la directory di esecuzione dell'applicazione.

[7] http://java.sun.com/
[8] http://justicexml.gtri.gatech.edu/extern/v2/validateXML-1.1/
[9] http://msdn.microsoft.com/netframework/downloads/
[10] http://msdn.microsoft.com/library/en-us/dnanchor/html/anch_xmlprod.asp

7 CONCLUSIONI

La semplicità di DTD++ derivata da DTD e la potenza espressiva del linguaggio ne agevolano l'utilizzo rispetto agli altri linguaggi proposti nel corso di questi anni, come si può notare nella figura 8.1.

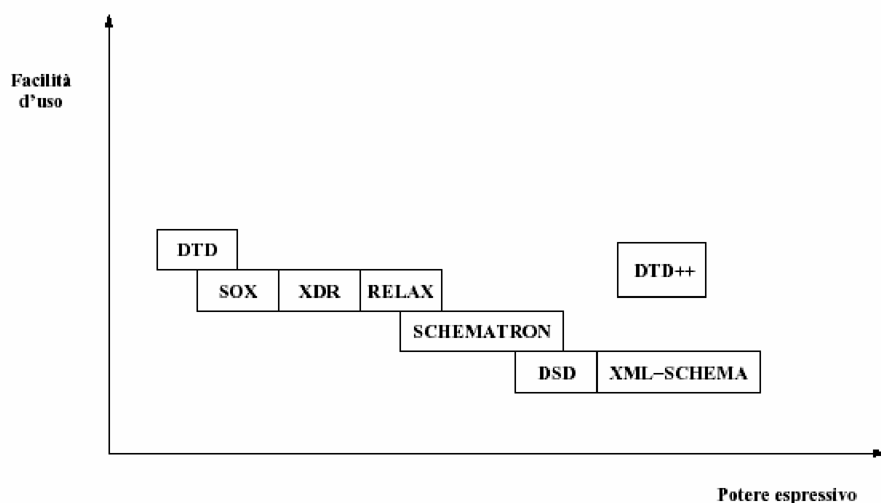

Figura 8.2 DTD++ e la sua collocazione tra i linguaggi di definizione dei tipi [Amo02]

La realizzazione di un preprocessore per la validazione dei documenti con DTD++ completa l'opera di Nicola Amorosi e rende DTD++ un linguaggio pronto per essere utilizzato anche in un ambiente di produzione.

La funzione di PreValidator for DTD++, che è per l'appunto quella di preprocessare il documento DTD++ trasformandolo in un documento XML Schema, lo rende una soluzione a parte dei problemi tipici che si hanno nell'abbracciare nuove tecnologie. Ai vantaggi già offerti da DTD++, quali la semplicità e la potenzialità espressiva che ne permettono un facile apprendimento, il preprocessore aggiunge, oltre alla funzionalità primaria per cui è stato realizzato, soluzioni pratiche alle problematiche di condivisione dei documenti con terzi e ai rischi di

insuccesso della nuova tecnologia. La possibilità di trasformare in XML Schema qualsiasi DTD++ prodotto lascia infatti sempre una porta aperta verso lo standard.
Inoltre l'utilizzo di validatori esterni permette di utilizzare sempre le tecnologie più avanzate.
La possibilità di esportare il documento nel formato XML Schema va guardata considerando le possibilità che XML Schema offre. È infatti possibile, grazie a questo, utilizzare tutti i software che fanno un uso di Schema, per esempio dei generatori automatici di calssi java o c++.
L'accoppiata DTD++ e PreValidator for DTD++ non deve considerarsi un punto d'arrivo bensì una soluzione stabile e ben funzionante da cui poter creare un sistema equivalente a XML Schema.
DTD++ potrebbe implementare altri costrutti senza perdere la caratteristica di base che lo contraddistingue, la semplicità. Un elemento equivalente a *group* di XML Schema si potrebbe aggiungere senza apportare modifiche significative al linguaggio così come la possibilità di associare gruppi di attributi ai tipi complessi, e non solo agli elementi come avviene ora.
Il preprocessore, invece, presenta alcune lacune nell'interfacciamento con i validatori. Manca, infatti, un sistema flessibile per la gestione dei software esterni. Al momento, per utilizzare validatori diversi da quelli predefiniti, Xerces e MSXML, è necessario intervenire direttamente sul codice.
Benché si sia cercato di ottimizzare al meglio il software si potrebbe ottenere un ulteriore miglioramento delle prestazioni con la realizzazione di un sistema di cache.
Per quanto riguarda gli sviluppi futuri, il software si presta ad essere integrato in qualsiasi ambiente.
Effettuando piccole modifiche al codice è possibile trasformare il preprocessore in una libreria per la gestione dei documenti DTD++. Ciò permetterebbe integrare DTD++ direttamente all'interno di vari tipi di programmi, per esempio un editor per documenti XML.

APPENDICE A - MATHML

In questa appendice si trova la versione completa della versione DTD++ del documento MathML 2.0

```
<!ENTITY % Content-elementary-functions.class
"exp|ln|log|logbase|sin|cos|tan|sec|csc|cot|arcsin|arccos|arctan|arcsec|arccsc|arccot|s
inh|cosh|tanh|sech|csch|coth|arccosh|arccoth|arccsch|arcsech|arcsinh|arctanh">

<!ENTITY % Presentation-token.class "mi|mo|mn|mtext|ms">
<!ENTITY % Presentation-table.class "mtable|maligngroup|malignmark">
<!ENTITY % Presentation-script.class
"msub|msup|msubsup|munder|mover|munderover|mmultiscripts">
<!ENTITY % Presentation-layout.class
"mrow|mfrac|msqrt|mroot|mpadded|mphantom|mfenced|menclose">


<!ENTITY % Content-constructs.class
"apply|interval|inverse|condition|declare|lambda|piecewise|bvar|degree">
<!ENTITY % Content-statistics.class
"mean|sdev|variance|median|mode|moment|momentabout">
<!ENTITY % Content-vector-calculus.class "divergence|grad|curl|laplacian">
<!ENTITY % Content-linear-algebra.class
"vector|matrix|determinant|transpose|selector|vectorproduct|scalarproduct|outerproduct"
>
<!ENTITY % Content-calculus.class
"int|diff|partialdiff|limit|lowlimit|uplimit|tendsto">
<!ENTITY % Content-relations.class "eq|neq|leq|lt|geq|gt|equivalent|approx|factorof">
<!ENTITY % Content-sets.class
"set|list|union|intersect|in|notin|subset|prsubset|notsubset|notprsubset|setdiff|card|c
artesianproduct">
<!ENTITY % Content-constants.class
"naturalnumbers|primes|integers|rationals|reals|complexes|emptyset|exponentiale|imagina
ryi|pi|eulergamma|true|false|infinity|notanumber">
<!ENTITY % Content-logic.class "and|or|xor|not|exists|forall|implies">
<!ENTITY % Content-functions.class
"compose|domain|codomain|image|domainofapplication|ident">
<!ENTITY % Content-arith.class
"abs|conjugate|factorial|arg|real|imaginary|floor|ceiling|quotient|divide|rem|minus|plu
s|times|power|root|max|min|gcd|lcm|sum|product">
<!ENTITY % Content-tokens.class "cn|ci|csymbol">

<!ENTITY % Definition.attrib "encoding (#STRING) definitionURL (#ANYURI)">


<!TARGETNS "http://www.w3.org/1998/Math/MathML">


<!-- common/attribs.xsd -->
<!-- The type of "class" is from the XHTML modularization with Schema
     document -->

<!ENTITY % Common.attrib "
  class (#NMTOKENS)
  style (#STRING)
  xref (#IDREF)
  id (#ID)
  href (#ANYURI)
  ">
<!-- common/math.xsd -->
```

```
<!-- The four groups that govern a lot of things -->

<!-- currently very lax. Should be tightened from Chapter 5 -->



<!ENTITY % PresExpr.class "%Presentation-token.class;|%Presentation-
layout.class;|%Presentation-script.class;|%Presentation-
table.class;|mspace|maction|merror|mstyle">

<!ENTITY % ContExpr.class "%Content-tokens.class;|%Content-arith.class;|%Content-
functions.class;|%Content-logic.class;|%Content-constants.class;|%Content-
sets.class;|%Content-relations.class;|%Content-elementary-functions.class;|%Content-
calculus.class;|%Content-linear-algebra.class;|%Content-vector-
calculus.class;|%Content-statistics.class;|%Content-constructs.class;|semantics">

<!ENTITY % Presentation-expr.class "%PresExpr.class;|%ContExpr.class;"> <!-- OK -->
<!ENTITY % Content-expr.class "%ContExpr.class;|%PresExpr.class;">       <!-- OK -->



<!-- presentation/common-attribs.xsd -->
<!-- The mathematics style attributes. These attributes are valid on all
     presentation token elements except "mspace" and "mglyph", and on no
     other elements except "mstyle". -->

<!ENTITY % Token-style.attrib "
  mathvariant (#STRING(normal|bold|italic|bold-italic|double-struck|bold-
fraktur|script|bold-script|fraktur|sans-serif|bold-sans-serif|sans-serif-italic|sans-
serif-bold-italic|monospace))
  mathsize (#simple-size;)|(#length-with-unit;)
  mathcolor (#STRING)
  mathbackground(#STRING)
  ">

<!-- These operators are all related to operators. They are valid on "mo"
     and "mstyle". -->

<!ENTITY % Operator.attrib "
  form (#STRING(prefix|infix|postfix))
  lspace (#length-with-unit;)|(#named-space;)
  rspace (#length-with-unit;)|(#named-space;)
  fence (#BOOLEAN)
  separator (#BOOLEAN)
  stretchy (#BOOLEAN)
  symmetric (#BOOLEAN)
  movablelimits (#BOOLEAN)
  accent (#BOOLEAN)
  largeop (#BOOLEAN)
  minsize (#length-with-unit;)|(#named-space;)
  maxsize (#length-with-unit;)|(#named-space;)|(#infinity;)|(#FLOAT)
  ">


<!-- presentation/token.xsd -->

<!-- The content of presentation token elements is either normal
  characters, "mglyph" ones or alignment marks -->

<!ENTITY % Glyph-alignmark.class "malignmark|mglyph">

<!-- "mi" -->
<!-- "mi" is supposed to have a default value of its "mathvariant" attribute
     set to "italic" -->
<!ENTITY % mi.attlist "%Token-style.attrib; %Common.attrib;">
<!ENTITY @ mi.type "(#PCDATA(%Glyph-alignmark.class;)*)" "%mi.attlist;">
<!ELEMENT mi (@mi.type;)>

<!-- "mo" -->
```

```
<!ENTITY % mo.attlist "%Operator.attrib; %Token-style.attrib; %Common.attrib;">
<!ENTITY @ mo.type "(#PCDATA(%Glyph-alignmark.class;)*)" "%mo.attlist;">
<!ELEMENT mo (@mo.type;)>


<!-- "mn" -->

<!ENTITY % mn.attlist "%Token-style.attrib; %Common.attrib;">
<!ENTITY @ mn.type "(#PCDATA(%Glyph-alignmark.class;)*)" "%mn.attlist;">
<!ELEMENT mn (@mn.type;)>

<!-- "mtext" -->

<!ENTITY % mtext.attlist "%Token-style.attrib; %Common.attrib;">
<!ENTITY @ mtext.type "(#PCDATA(%Glyph-alignmark.class;)*)" "%mtext.attlist;">
<!ELEMENT mtext (@mtext.type;)>

<!-- "ms" -->

   <!-- the values of "lquote" or "rquote" are not restricted to be
        one character strings... -->
<!ENTITY % ms.attlist "lquote (#STRING) '&quot;' rquote (#STRING) '&quot;'  %Token-
style.attrib; %Common.attrib;">
<!ENTITY @ ms.type "(#PCDATA(%Glyph-alignmark.class;)*)" "%ms.attlist;">
<!ELEMENT ms (@ms.type;)>

<!-- And the group of any token -->


<!-- common/math.xsd -->
<!-- The four groups that govern a lot of things -->

<!-- currently very lax. Should be tightened from Chapter 5 -->

<!--
<!ENTITY % Presentation-expr.class "%PresExpr.class;|%ContExpr.class;">
<!ENTITY % Content-expr.class "%ContExpr.class;|%PresExpr.class;">

<!ENTITY % PresExpr.class "%Presentation-token.class;|%Presentation-
layout.class;|%Presentation-script.class;|%Presentation-
table.class;|mspace|maction|merror|mstyle">

<!ENTITY % ContExpr.class "%Content-tokens.class;|%Content-arith.class;|%Content-
functions.class;|%Content-logic.class;|%Content-constants.class;|%Content-
sets.class;|%Content-relations.class;|%Content-elementary-functions.class;|%Content-
calculus.class;|%Content-linear-algebra.class;|%Content-vector-
calculus.class;|%Content-statistics.class;|%Content-constructs.class;|semantics">
-->

<!-- "math" -->

<!ENTITY % Browser-interface.attrib "
  baseline (#STRING)
  overflow (#STRING(scroll|elide|truncate|scale)) 'scroll'
  altimg (#ANYURI)
  alttext (#STRING)
  type (#STRING)
  name (#STRING)
  height (#STRING)
  width (#STRING)
  ">

<!ENTITY % math.attlist "
  %Browser-interface.attrib;
  macros  (#STRING)
  display (#STRING(block|inline)) 'inline'
  %Common.attrib;
  ">
```

```
<!ENTITY % math.content "%PresExpr.class;|%ContExpr.class;">
<!ENTITY @ math.type "(%math.content;)*" "%math.attlist;">
<!ELEMENT math (@math.type;)>

<!-- content/constant.xsd -->

<!-- a common type for all this -->

<!ENTITY @ Constant.type "" "%Definition.attrib; %Common.attrib;">

<!-- Basic sets -->
<!ELEMENT naturalnumbers (@Constant.type;)>
<!ELEMENT primes (@Constant.type;)>
<!ELEMENT integers (@Constant.type;)>
<!ELEMENT rationals (@Constant.type;)>
<!ELEMENT reals (@Constant.type;)>
<!ELEMENT complexes (@Constant.type;)>

 <!-- Empty set -->
<!ELEMENT emptyset (@Constant.type;)>

<!-- Basic constants -->
<!ELEMENT exponentiale (@Constant.type;)>
<!ELEMENT imaginaryi (@Constant.type;)>
<!ELEMENT pi (@Constant.type;)>
<!ELEMENT eulergamma (@Constant.type;)>

<!-- Boolean constants -->
<!ELEMENT true (@Constant.type;)>
<!ELEMENT false (@Constant.type;)>

<!-- Infinty -->
<!ELEMENT infinity (@Constant.type;)>

<!-- NotANumber -->
<!ELEMENT notanumber (@Constant.type;)>



<!-- elementary-functions.xsd -->

<!-- a common type for all this -->
<!ENTITY @ Elementary-functions.type "" "%Definition.attrib; %Common.attrib;">

<!-- Exp and logs -->
<!ELEMENT exp (@Elementary-functions.type;)>
<!ELEMENT ln (@Elementary-functions.type;)>
<!ELEMENT log (@Elementary-functions.type;)>

<!-- special element of the base of logarithms -->
<!ENTITY % logbase.content "%Content-expr.class;">
<!ENTITY @ logbase.type "(%logbase.content;)" "%Common.attrib;">
<!ELEMENT logbase (@logbase.type;)>

<!-- Trigonometric functions -->
<!ELEMENT sin (@Elementary-functions.type;)>
<!ELEMENT cos (@Elementary-functions.type;)>
<!ELEMENT tan (@Elementary-functions.type;)>
<!ELEMENT sec (@Elementary-functions.type;)>
<!ELEMENT csc (@Elementary-functions.type;)>
<!ELEMENT cot (@Elementary-functions.type;)>
<!ELEMENT arcsin (@Elementary-functions.type;)>
<!ELEMENT arccos (@Elementary-functions.type;)>
<!ELEMENT arctan (@Elementary-functions.type;)>
<!ELEMENT arccot (@Elementary-functions.type;)>
<!ELEMENT arccsc (@Elementary-functions.type;)>
<!ELEMENT arcsec (@Elementary-functions.type;)>
```

```
<!-- Hyperbolic trigonometric functions -->
<!ELEMENT sinh (@Elementary-functions.type;)>
<!ELEMENT cosh (@Elementary-functions.type;)>
<!ELEMENT tanh (@Elementary-functions.type;)>
<!ELEMENT sech (@Elementary-functions.type;)>
<!ELEMENT csch (@Elementary-functions.type;)>
<!ELEMENT coth (@Elementary-functions.type;)>
<!ELEMENT arccosh (@Elementary-functions.type;)>
<!ELEMENT arccoth (@Elementary-functions.type;)>
<!ELEMENT arccsch (@Elementary-functions.type;)>
<!ELEMENT arcsech (@Elementary-functions.type;)>
<!ELEMENT arcsinh (@Elementary-functions.type;)>
<!ELEMENT arctanh (@Elementary-functions.type;)>

<!-- And the group of everything -->


<!-- presentation/table.xsd -->

<!-- Common attributes - presentation/table.xsd -->

<!ENTITY % Table-alignment.attrib "
  rowalign (#STRING/(top|bottom|center|baseline|axis)( top| bottom| center| baseline|
axis)*/) 'baseline'
  columnalign (#STRING/(left|center|right)( left| center| right)*/) 'center'
  groupalign (#STRING)
  ">

<!-- "mtr" -->

<!ENTITY % mtr.attlist "
  %Table-alignment.attrib;
  %Common.attrib;
  ">

<!ENTITY % mtr.content "mtd">
<!ENTITY @ mtr.type "(%mtr.content;)+" "%mtr.attlist;">
<!ELEMENT mtr (@mtr.type;)>

<!-- "labeledtr" -->

<!ENTITY % mlabeledtr.attlist "
  %Table-alignment.attrib;
  %Common.attrib;
  ">

<!ENTITY % mlabeledtr.content "mtd">
<!ENTITY @ mlabeledtr.type "(%mlabeledtr.content;)+" "%mlabeledtr.attlist;">
<!ELEMENT mlabeledtr (@mlabeledtr.type;) >

<!-- "mtd" -->

<!ENTITY % mtd.attlist "
  %Table-alignment.attrib;
  columnspan (#POSINTEGER) '1'
  rowspan (#POSINTEGER) '1'
  %Common.attrib;
  ">

<!ENTITY % mtd.content "(%Presentation-expr.class;)?" >
<!ENTITY @ mtd.type "(%mtd.content;)*" "%mtd.attlist;">
<!ELEMENT mtd (@mtd.type;)>

<!-- "mtable" -->

<!ENTITY % mtable.attlist "
  %Table-alignment.attrib;
  align (#STRING) 'axis'
  alignmentscope (#STRING/(true|false)( true| false)*/) 'true'
```

```
  columnwidth (#STRING) 'auto'
  width (#STRING) 'auto'
  rowspacing (#STRING) '1.0ex'
  columnspacing (#STRING) '0.8em'
  rowlines (#STRING) 'none'
  columnlines (#STRING) 'none'
  frame (#STRING(none|solid|dashed)) 'none'
  framespacing (#STRING) '0.4em 0.5ex'
  equalrows (#BOOLEAN) 'false'
  equalcolumns (#BOOLEAN) 'false'
  displaystyle (#BOOLEAN) 'false'
  side   (#STRING(left|right|leftoverlap|rightoverlap)) 'right'
  minlabelspacing (#length-with-unit;) '0.8em'
  %Common.attrib;
  ">

<!ENTITY % mtable.content "mtr|mlabeledtr">
<!ENTITY @ mtable.type "(%mtable.content;)+" "%mtable.attlist;">
<!ELEMENT mtable (@mtable.type;)>

<!-- "maligngroup" -->

<!ENTITY % maligngroup.attlist "
  groupalign (#STRING(left|center|right|decimalpoint))
  %Common.attrib;
  ">

<!ENTITY @ maligngroup.type "" "%maligngroup.attlist;">
<!ELEMENT maligngroup (@maligngroup.type;)>

<!-- "malignmark" -->

<!ENTITY % malignmark.attlist "
  edge (#STRING(left|right)) 'left'
  %Common.attrib;
  ">

<!ENTITY @ malignmark.type "" "%malignmark.attlist;">
<!ELEMENT malignmark (@malignmark.type;)>



<!-- presentation/script.xsd -->

<!-- "msub" -->

<!ENTITY % msub.attlist "
  subscriptshift (#length-with-unit;)
  %Common.attrib;
  ">

<!ENTITY @ msub.type "(%Presentation-expr.class;)[2,2]" "%msub.attlist;">
<!ELEMENT msub (@msub.type;)>

<!-- "msup" -->

<!ENTITY % msup.attlist "
  superscriptshift (#length-with-unit;)
  %Common.attrib;
  ">

<!ENTITY @ msup.type "(%Presentation-expr.class;)[2,2]" "%msup.attlist;">
<!ELEMENT msup  (@msup.type;)>

<!-- "msubsup" -->

<!ENTITY % msubsup.attlist "
  subscriptshift (#length-with-unit;)
  superscriptshift (#length-with-unit;)
```

```
  %Common.attrib;
  ">

<!ENTITY @ msubsup.type "(%Presentation-expr.class;)[3,3]" "%msubsup.attlist;">
<!ELEMENT msubsup  (@msubsup.type;)>

<!-- "munder" -->

<!ENTITY % munder.attlist "
  accentunder (#BOOLEAN)
  %Common.attrib;
  ">

<!ENTITY @ munder.type "(%Presentation-expr.class;)[2,2]" "%munder.attlist;">
<!ELEMENT munder  (@munder.type;)>

<!-- "mover" -->

<!ENTITY % mover.attlist "
  accent (#BOOLEAN)
  %Common.attrib;
  ">

<!ENTITY @ mover.type "(%Presentation-expr.class;)[2,2]" "%mover.attlist;">
<!ELEMENT mover (@mover.type;)>

<!-- "munderover" -->

<!ENTITY % munderover.attlist "
  accent (#BOOLEAN)
  accentunder (#BOOLEAN)
  %Common.attrib;
  ">


<!ENTITY @ munderover.type "(%Presentation-expr.class;)[3,3]" "%munderover.attlist;">
<!ELEMENT munderover  (@munderover.type;)>

<!-- "mmultiscripts", "mprescripts" and "none" -->

<!ENTITY % mmultiscripts.attlist "
  %Common.attrib;
  ">

<!ENTITY % Presentation-expr-or-none.class "%Presentation-expr.class;|none">
<!ENTITY % mmultiscripts.content "
                                   (%Presentation-expr.class;),
                                   (
                                         (%Presentation-expr-or-none.class;),
                                         (%Presentation-expr-or-none.class;)
                                   )*,
                                   (
                                         mprescripts,
                                         (
                                                 (%Presentation-expr-or-none.class;),
                                                 (%Presentation-expr-or-none.class;)*
                                         )
                                   )*
                                   " >

<!ENTITY @ mmultiscripts.type "(%mmultiscripts.content;)" "%mmultiscripts.attlist;">
<!ELEMENT mmultiscripts (@mmultiscripts.type;)>

<!-- Nothing... -->
<!ENTITY @ none.type "">
<!ELEMENT none (@none.type;)>

<!-- also void -->
<!ENTITY @ mprescripts.type "">
```

```
<!ELEMENT mprescripts (@mprescripts.type;)>


<!-- presentation/layout.xsd -->
<!-- "mrow" -->

<!ENTITY % mrow.attlist "
  %Common.attrib;
  ">

<!ENTITY @ mrow.type "(%Presentation-expr.class;)*" "%mrow.attlist;">
<!ELEMENT mrow (@mrow.type;)>

<!-- "mfrac" -->

<!ENTITY % mfrac.attlist "
  bevelled (#BOOLEAN)
  denomalign (#centering;) 'center'
  numalign (#centering;) 'center'
  linethickness (#length-with-optional-unit;)|(#thickness;) '1'
  %Common.attrib;
  ">

<!ENTITY @ mfrac.type "(%Presentation-expr.class;)[2,2]" "%mfrac.attlist;">
<!ELEMENT mfrac (@mfrac.type;)>

<!-- "msqrt" -->

<!ENTITY % msqrt.attlist "
  %Common.attrib;
  ">

<!-- "msqrt" has an "inferred mrow" if more than one argument -->
<!ENTITY @ msqrt.type "(%Presentation-expr.class;)*" "%msqrt.attlist;">
<!ELEMENT msqrt (@msqrt.type;)>

<!-- "mroot" -->

<!ENTITY % mroot.attlist "
  %Common.attrib;
  ">

<!ENTITY @ mroot.type "(%Presentation-expr.class;)[2,2]" "%mroot.attlist;">
<!ELEMENT mroot (@mroot.type;)>

<!-- "mpadded" -->
<!ENTITY # mpadded-space "(#STRING/(\+|-)?([0-9]+|[0-9]*\.[0-9]+)(((%?)
*(width|lspace|height|depth))|(em|ex|px|in|cm|mm|pt|pc))/)">

<!-- MaxF: definition from spec seems wrong, fixing to ([+|-] unsigned-number
(%[pseudo-unit]|pseudo-unit|h-unit)) | namedspace | 0 -->
<!ENTITY # mpadded-width-space "(#STRING/((\+|-)?([0-9]+|[0-9]*\.[0-9]+)(((%?)
*(width|lspace|height|depth)?)|(width|lspace|height|depth)|(em|ex|px|in|cm|mm|pt|pc)))|
((veryverythin|verythin|thin|medium|thick|verythick|veryverythick)mathspace)|0/)">

<!ENTITY % mpadded.attlist "
  width (#mpadded-width-space;)
  lspace (#mpadded-width-space;)
  height (#mpadded-width-space;)
  depth (#mpadded-width-space;)
  %Common.attrib;
  ">

<!ENTITY @ mpadded.type "(%Presentation-expr.class;)*" "%mpadded.attlist;">
<!ELEMENT mpadded (@mpadded.type;)>

<!-- "mphantom" -->

<!ENTITY % mphantom.attlist "
```

```
  %Common.attrib;
  ">

<!ENTITY @ mphantom.type "(%Presentation-expr.class;)*" "%mphantom.attlist;">

<!ELEMENT mphantom (@mphantom.type;)>

<!-- "mfenced" -->

<!ENTITY % mfenced.attlist "
  open (#STRING) '('
  close (#STRING) ')'
  separators (#STRING) ','
  %Common.attrib;
  ">

<!ENTITY @ mfenced.type "(%Presentation-expr.class;)*" "%mfenced.attlist;">
<!ELEMENT mfenced (@mfenced.type;)>

<!-- "menclose" -->

<!ENTITY % menclose.attlist "
  notation (#STRING(actuarial|longdiv|radical)) 'longdiv'
  %Common.attrib;
  ">

<!ENTITY @ menclose.type "(%Presentation-expr.class;)*" "%menclose.attlist;">
<!ELEMENT menclose (@menclose.type;)>

<!-- Content/construct.xsd -->

<!-- "apply" -->

<!ENTITY % apply.attlist "
  %Common.attrib;
  ">

<!ENTITY % apply.content "%Content-expr.class;">
<!ENTITY @ apply.type "(%apply.content;)*" "%apply.attlist;">
<!ELEMENT apply  (@apply.type;)>

<!-- "interval" -->

<!ENTITY % interval.attlist "
  closure  (#STRING(closed|open|open-closed|closed-open)) 'closed'
  %Common.attrib;
  ">

<!ENTITY @ interval.type "(%Content-expr.class;)[0,2]" "%interval.attlist;">
<!ELEMENT interval (@interval.type;)>

<!-- "inverse" -->
<!ENTITY % inverse.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ inverse.type "" "%inverse.attlist;">
<!ELEMENT inverse (@inverse.type;)>

<!-- "condition" -->

<!ENTITY % condition.attlist "
  %Definition.attrib;
  ">

<!ENTITY % condition.content "%Content-expr.class;">
<!ENTITY @ condition.type "(%condition.content;)+" "%condition.attlist;">
<!ELEMENT condition (@condition.type;)>
```

```
<!-- "declare" -->

<!ENTITY % declare.attlist "
  type (#STRING)
  scope (#STRING)
  nargs (#NONNEGINTEGER)
  occurrence (#STRING(prefix|infix|function-model))
  %Definition.attrib;
  ">

<!ENTITY % declare.content "%Content-expr.class;">
<!ENTITY @ declare.type "(%declare.content;)+" "%declare.attlist;">
<!ELEMENT declare (@declare.type;)>

<!-- "lambda" -->
<!ENTITY % lambda.attlist "
  %Common.attrib;
  ">

<!ENTITY % lambda.content "%Content-expr.class;">
<!ENTITY @ lambda.type "(%lambda.content;)+" "%lambda.attlist;">
<!ELEMENT lambda (@lambda.type;)>


<!-- "piecewise" and its inner elements -->
<!ENTITY % otherwise.content "%Content-expr.class;">

<!ENTITY @ otherwise.type "(%otherwise.content;)" "%Common.attrib;">
<!ELEMENT otherwise (@otherwise.type;)>
<!ENTITY % piece.content "%Content-expr.class;">

<!ENTITY @ piece.type "(%piece.content;)+">
<!ELEMENT piece (@piece.type;)>

<!ENTITY % piecewise.attlist "
  %Common.attrib;
  ">

<!ENTITY % piecewise.content "piece*,((otherwise,piece*)*)">
<!ENTITY @ piecewise.type "(%piecewise.content;)" "%piecewise.attlist;">
<!ELEMENT piecewise (@piecewise.type;)>

<!-- "bvar" -->
<!ENTITY % bvar.attlist "
  %Common.attrib;
  ">

<!ENTITY % bvar.content "%Content-expr.class;">

<!ENTITY @ bvar.type "(%bvar.content;)+" "%bvar.attlist;">
<!ELEMENT bvar (@bvar.type;)>

<!-- "degree" -->
<!ENTITY % degree.attlist "
  %Common.attrib;
  ">

<!ENTITY % degree.content "%Content-expr.class;">
<!ENTITY @ degree.type "(%degree.content;)+" "%degree.attlist;">
<!ELEMENT degree (@degree.type;)>


<!-- content/statistic.xsd -->

<!-- "mean" -->
<!ENTITY % mean.attlist "
  %Definition.attrib;
  %Common.attrib;
```

```
  ">

<!ENTITY @ mean.type "" "%mean.attlist;">
<!ELEMENT mean (@mean.type;)>

<!-- "sdev" -->
<!ENTITY % sdev.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ sdev.type "" "%sdev.attlist;">
<!ELEMENT sdev (@sdev.type;)>

<!-- "variance" -->
<!ENTITY % variance.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ variance.type "" "%variance.attlist;">
<!ELEMENT variance (@variance.type;)>

<!-- "median" -->
<!ENTITY % median.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ median.type "" "%median.attlist;">
<!ELEMENT median (@median.type;)>

<!-- "mode" -->
<!ENTITY % mode.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ mode.type "" "%mode.attlist;">
<!ELEMENT mode (@mode.type;)>

<!-- "moment" -->

<!ENTITY % moment.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ moment.type "" "%moment.attlist;">
<!ELEMENT moment (@moment.type;)>

<!-- "momentabout" -->

<!ENTITY % momentabout.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY % momentabout.content "%Content-expr.class;">
<!ENTITY @ momentabout.type "(%momentabout.content;)+" "%momentabout.attlist;">
<!ELEMENT momentabout (@momentabout.type;)>


<!-- content/vector-calculus.xsd -->
<!-- "divergence" -->

<!ENTITY % divergence.attlist "
  %Definition.attrib;
  %Common.attrib;
```

```
  ">
<!ENTITY @ divergence.type "" "%divergence.attlist;">
<!ELEMENT divergence (@divergence.type;)>

<!-- "grad" -->

<!ENTITY % grad.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">
<!ENTITY @ grad.type "" "%grad.attlist;">
<!ELEMENT grad (@grad.type;)>

<!-- "curl" -->

<!ENTITY % curl.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">
<!ENTITY @ curl.type "" "%curl.attlist;">
<!ELEMENT curl (@curl.type;)>

<!-- "laplacian" -->

<!ENTITY % laplacian.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">
<!ENTITY @ laplacian.type "" "%laplacian.attlist;">
<!ELEMENT laplacian (@laplacian.type;)>


<!-- content/linear-algebra.xsd -->

<!-- "vector" -->

<!ENTITY % vector.attlist "
  %Common.attrib;
  ">

<!ENTITY % vector.content "%Content-expr.class;">
<!ENTITY @ vector.type "(%vector.content;)+" "%vector.attlist;">
<!ELEMENT vector (@vector.type;)>

<!-- "matrix" -->
<!ENTITY % matrix.attlist "
  %Common.attrib;
  ">

<!ENTITY % matrix.content "matrixrow">
<!ENTITY @ matrix.type "(%matrix.content;)+" "%matrix.attlist;">
<!ELEMENT matrix (@matrix.type;)>

<!-- "matrixrow" -->
<!ENTITY % matrixrow.attlist "
  %Common.attrib;
  ">

<!ENTITY % matrixrow.content "%Content-expr.class;">
<!ENTITY @ matrixrow.type "(%matrixrow.content;)+" "%matrixrow.attlist;">
<!ELEMENT matrixrow (@matrixrow.type;)>

<!-- "determinant" -->

<!ENTITY % determinant.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">
<!ENTITY @ determinant.type "" "%determinant.attlist;">
```

```
<!ELEMENT determinant (@determinant.type;)>

<!-- "transpose" -->
<!ENTITY % transpose.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">
<!ENTITY @ transpose.type "" "%transpose.attlist;">
<!ELEMENT transpose (@transpose.type;)>

<!-- "selector" -->
<!ENTITY % selector.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ selector.type "" "%selector.attlist;">
<!ELEMENT selector (@selector.type;)>

<!-- "vectorproduct" -->
<!ENTITY % vectorproduct.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ vectorproduct.type "" "%vectorproduct.attlist;">
<!ELEMENT vectorproduct (@vectorproduct.type;)>

<!-- "scalarproduct" -->

<!ENTITY % scalarproduct.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ scalarproduct.type "" "%scalarproduct.attlist;">
<!ELEMENT scalarproduct (@scalarproduct.type;)>

<!-- "outerproduct" -->
<!ENTITY % outerproduct.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ outerproduct.type "" "%outerproduct.attlist;">
<!ELEMENT outerproduct (@outerproduct.type;)>



<!-- content/calculus.xsd -->

<!-- "int" -->
<!ENTITY % int.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ int.type "" "%int.attlist;">
<!ELEMENT int (@int.type;)>

<!-- "diff" -->

<!ENTITY % diff.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">
<!ENTITY @ diff.type "" "%diff.attlist;">
<!ELEMENT diff (@diff.type;)>
```

```
<!-- "partialdiff" -->

<!ENTITY % partialdiff.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ partialdiff.type "" "%partialdiff.attlist;">
<!ELEMENT partialdiff (@partialdiff.type;)>

<!-- "limit" -->
<!ENTITY % limit.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">
<!ENTITY @ limit.type "" "%limit.attlist;">
<!ELEMENT limit (@limit.type;)>

<!-- "lowlimit" -->

<!ENTITY % lowlimit.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY % lowlimit.content "%Content-expr.class;">
<!ENTITY @ lowlimit.type "(%lowlimit.content;)+" "%lowlimit.attlist;">
<!ELEMENT lowlimit (@lowlimit.type;)>

<!-- "uplimit" -->

<!ENTITY % uplimit.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY % uplimit.content "%Content-expr.class;">
<!ENTITY @ uplimit.type "(%uplimit.content;)+" "%uplimit.attlist;">
<!ELEMENT uplimit (@uplimit.type;)>

<!-- "tendsto" -->

<!ENTITY % tendsto.attlist "
  type (#STRING)
  %Definition.attrib;
  %Common.attrib;
  ">
<!ENTITY @ tendsto.type "" "%tendsto.attlist;">
<!ELEMENT tendsto (@tendsto.type;)>

<!-- content/relations.xsd -->

<!-- a common type for all this -->
<!ENTITY @ Relations.type "" "%Definition.attrib; %Common.attrib;">

<!ELEMENT eq (@Relations.type;)>
<!ELEMENT neq (@Relations.type;)>
<!ELEMENT leq (@Relations.type;)>
<!ELEMENT lt (@Relations.type;)>
<!ELEMENT geq (@Relations.type;)>
<!ELEMENT gt (@Relations.type;)>
<!ELEMENT equivalent (@Relations.type;)>
<!ELEMENT approx (@Relations.type;)>
<!ELEMENT factorof (@Relations.type;)>

<!-- content/sets.xsd -->

<!-- "set" -->
```

```
<!-- "type" could be "multiset" or "normal" or anything else -->
<!ENTITY % set.attlist "
  type (#STRING)
  %Common.attrib;
  ">

<!ENTITY % set.content "%Content-expr.class;">
<!ENTITY @ set.type "(%set.content;)*" "%set.attlist;">
<!ELEMENT set (@set.type;)>

<!-- "list" -->
<!ENTITY % list.attlist"
  order (#STRING(lexicographic|numeric))
  %Common.attrib;
  ">

<!ENTITY % list.content "%Content-expr.class;">
<!ENTITY @ list.type "(%list.content;)*" "%list.attlist;">
<!ELEMENT list (@list.type;)>

<!-- "union" -->
<!ENTITY % union.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ union.type "" "%union.attlist;">
<!ELEMENT union (@union.type;)>

<!-- "intersect" -->

<!ENTITY % intersect.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ intersect.type "" "%intersect.attlist;">
<!ELEMENT intersect (@intersect.type;)>

<!-- "in" -->

<!ENTITY % in.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ in.type "" "%in.attlist;">
<!ELEMENT in (@in.type;)>

<!-- "notin" -->

<!ENTITY % notin.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ notin.type "" "%notin.attlist;">
<!ELEMENT notin (@notin.type;)>

<!-- "subset" -->
<!ENTITY % subset.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ subset.type "" "%subset.attlist;">
<!ELEMENT subset (@subset.type;)>

<!-- "prsubset" -->
```

```
<!ENTITY % prsubset.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ prsubset.type "" "%prsubset.attlist;">
<!ELEMENT prsubset (@prsubset.type;)>

<!-- "notsubset" -->

<!ENTITY % notsubset.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ notsubset.type "" "%notsubset.attlist;">
<!ELEMENT notsubset (@notsubset.type;)>

<!-- "notprsubset" -->

<!ENTITY % notprsubset.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ notprsubset.type "" "%notprsubset.attlist;">
<!ELEMENT notprsubset (@notprsubset.type;)>

<!-- "setdiff" -->

<!ENTITY % setdiff.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ setdiff.type "" "%setdiff.attlist;">
<!ELEMENT setdiff (@setdiff.type;)>

<!-- "card" -->

<!ENTITY % card.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ card.type "" "%card.attlist;">
<!ELEMENT card (@card.type;)>

<!-- "cartesianproduct" -->

<!ENTITY % cartesianproduct.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ cartesianproduct.type "" "%cartesianproduct.attlist;">
<!ELEMENT cartesianproduct (@cartesianproduct.type;)>

<!-- content/logic.xsd -->

<!-- a common type for all this -->

<!ENTITY @ Logic.type "" "%Definition.attrib; %Common.attrib;">
<!ELEMENT and (@Elementary-functions.type;)>
<!ELEMENT or (@Logic.type;)>
<!ELEMENT xor (@Logic.type;)>
<!ELEMENT not (@Logic.type;)>
<!ELEMENT exists (@Logic.type;)>
```

```
<!ELEMENT forall (@Logic.type;)>
<!ELEMENT implies (@Logic.type;)>

<!-- content/functions.xsd -->

<!ENTITY @ Functions.type "" "%Definition.attrib; %Common.attrib;">

<!-- "compose" -->
<!ELEMENT compose (@Functions.type;)>

<!-- Domain, codomain and image -->

<!ELEMENT domain (@Functions.type;)>
<!ELEMENT codomain (@Functions.type;)>
<!ELEMENT image (@Functions.type;)>

<!-- "domainofapplication" -->

<!ENTITY % domainofapplication.content "%Content-expr.class;">
<!ENTITY @ domainofapplication.type "(%domainofapplication.content;)"
"%Definition.attrib; %Common.attrib;">
<!ELEMENT domainofapplication (@domainofapplication.type;)>

<!-- identity -->

<!ELEMENT ident (@Functions.type;)>


<!-- content/arith.xsd -->

<!ENTITY @ Arith.type "" "%Definition.attrib; %Common.attrib;">

<!-- The elements -->

<!ELEMENT abs (@Arith.type;)>
<!ELEMENT conjugate (@Arith.type;)>
<!ELEMENT arg (@Arith.type;)>
<!ELEMENT real (@Arith.type;)>
<!ELEMENT imaginary (@Arith.type;)>

<!ELEMENT floor (@Arith.type;)>
<!ELEMENT ceiling (@Arith.type;)>

<!ELEMENT power (@Arith.type;)>
<!ELEMENT root (@Arith.type;)>

<!ELEMENT minus (@Arith.type;)>
<!ELEMENT plus (@Arith.type;)>
<!ELEMENT sum (@Arith.type;)>
<!ELEMENT times (@Arith.type;)>
<!ELEMENT product (@Arith.type;)>

<!ELEMENT max (@Arith.type;)>
<!ELEMENT min (@Arith.type;)>

<!ELEMENT factorial (@Arith.type;)>
<!ELEMENT quotient (@Arith.type;)>
<!ELEMENT divide (@Arith.type;)>
<!ELEMENT rem (@Arith.type;)>
<!ELEMENT gcd (@Arith.type;)>
<!ELEMENT lcm (@Arith.type;)>

<!-- content/token.xsd -->

<!ENTITY % Content-token.content "%Presentation-expr.class;">

<!-- "cn" -->
<!ENTITY % cn.attlist "
  base (#POSINTEGER[2,36])
```

```
  type (#NMTOKEN(integer|rational|real|complex-cartesian|complex-polar|constant))
  %Definition.attrib;
  %Common.attrib;
  ">
<!-- the content of "cn" may have <sep> elements in it -->

<!ENTITY @ sep.type "">
<!ELEMENT sep (@sep.type;)>

<!ENTITY % cn.content "%Presentation-expr.class;|sep">
<!ENTITY @ cn.type "(#PCDATA(%cn.content;)*)" "%cn.attlist;">
<!ELEMENT cn (@cn.type;)>

<!-- "ci" -->
<!ENTITY % ci.attlist "type (#STRING)   %Definition.attrib; %Common.attrib;">

<!ENTITY @ ci.type "(#PCDATA(%Content-token.content;)*)" "%ci.attlist;">
<!ELEMENT ci (@ci.type;)>

<!-- "csymbol" -->

<!ENTITY % csymbol.attlist "
  %Definition.attrib;
  %Common.attrib;
  ">

<!ENTITY @ csymbol.type "(#PCDATA(%Content-token.content;)*)" "%csymbol.attlist;">
<!ELEMENT csymbol (@csymbol.type;)>

<!-- presentations/common-types.xsd -->

<!-- Simple sizes -->
<!ENTITY # simple-size "(#STRING(small|normal|big))">

<!-- Centering values -->
<!ENTITY # centering "(#STRING(left|center|right))">

<!-- The named spaces -->

<!-- this is also used in the value of the "width" attribute on the
     "mpadded" element -->
<!ENTITY # named-space
"(#STRING(veryverythinmathspace|verythinmathspace|thinmathspace|mediummathspace|thickma
thspace|verythickmathspace|veryverythickmathspace))">

<!-- Thickness -->
<!ENTITY # thickness "(#STRING(thin|medium|thick))">

<!-- number with units used to specified lengths -->
<!ENTITY # length-with-unit "(#STRING/(-?([0-9]+|[0-9]*\.[0-
9]+)*(em|ex|px|in|cm|mm|pt|pc|%))|0/)" >

<!ENTITY # length-with-optional-unit "(#STRING/-?([0-9]+|[0-9]*\.[0-
9]+)*(em|ex|px|in|cm|mm|pt|pc|%)?/)">

<!-- This is just "infinity" that can be used as a length -->
<!ENTITY # infinity "(#STRING(infinity))">

<!-- colors defined as RGB -->
<!ENTITY # RGB-color "(#STRING/#(([0-9]|[a-f]){3}|([0-9]|[a-f]){6})/)">

<!-- content/semantic.xsd -->

<!-- "annotation" -->

<!ENTITY % annotation.attlist "
  encoding (#STRING)
 %Common.attrib;
 ">
```

```
<!ENTITY  @ annotation.type "(#PCDATA)" "%annotation.attlist;">
<!ELEMENT annotation (@annotation.type;)>

<!-- "annotation-xml" -->

<!ENTITY % annotation-xml.attlist "
  encoding (#STRING)
 %Common.attrib;
 ">

<!ENTITY % annotation-xml.content "ANY{##any}">
<!ENTITY @ annotation-xml.type "%annotation-xml.content;" "%annotation-xml.attlist;">
<!ELEMENT annotation-xml (@annotation-xml.type;)>

<!-- "semantics" -->
<!ENTITY % semantics.attlist "
  definitionURL (#ANYURI)
  encoding (#STRING)
 %Common.attrib;
 ">

<!ENTITY % Annotation.class "annotation|annotation-xml">

<!ENTITY % semantics.content "(%Content-expr.class;),(%Annotation.class;)*">

<!ENTITY @ semantics.type "(%semantics.content;)*" "%semantics.attlist;">
<!ELEMENT semantics (@semantics.type;)>

<!-- presentation/characters.xsd -->

<!ENTITY % mglyph.attlist "
  alt (#STRING)
  fontfamily (#STRING)
  index (#POSINTEGER)
  ">

<!ENTITY @ mglyph.type "" "%mglyph.attlist;">
<!ELEMENT mglyph (@mglyph.type;)>

<!-- presentation/error.xsd -->

<!ENTITY % merror.attlist "
  %Common.attrib;
  ">

<!ENTITY % merror.content "%Presentation-expr.class;">
<!ENTITY @ merror.type "(%merror.content;)*" "%merror.attlist;">
<!ELEMENT merror (@merror.type;)>

<!-- presentation/space.xsd -->


<!ENTITY % mspace.attlist "
  width  (#length-with-unit;)|(#named-space;) '0em'
  height (#length-with-unit;) '0ex'
  depth (#length-with-unit;) '0ex'
  linebreak  (#STRING(auto|newline|indentingnewline|nobreak|goodbreak|badbreak)) 'auto'
  %Common.attrib;
  ">
<!ENTITY @ mspace.type "" "%mspace.attlist;">
<!ELEMENT mspace (@mspace.type;)>

<!-- presentation/style.xsd -->

<!-- "mstyle" -->

<!ENTITY % mstyle.attlist "
  scriptlevel (#INTEGER)
```

```
  displaystyle (#BOOLEAN)
  scriptsizemultiplier (#DECIMAL) '0.71'
  scriptminsize (#length-with-unit;) '8pt'
  color (#STRING)
  background (#STRING) 'transparent'
  veryverythinmathspace (#length-with-unit;) '0.0555556em'
  verythinmathspace (#length-with-unit;) '0.111111em'
  thinmathspace (#length-with-unit;) '0.166667em'
  mediummathspace (#length-with-unit;) '0.222222em'
  thickmathspace (#length-with-unit;) '0.277778em'
  verythickmathspace (#length-with-unit;) '0.333333em'
  veryverythickmathspace (#length-with-unit;) '0.388889em'
  linethickness (#length-with-optional-unit;)|(#thickness;) '1'
  %Operator.attrib;
  %Token-style.attrib;
  %Common.attrib;
  ">
<!ENTITY % mstyle.content "%Presentation-expr.class;">
<!ENTITY @ mstyle.type "(%mstyle.content;)+" "%mstyle.attlist;">
<!ELEMENT mstyle (@mstyle.type;)>

<!-- presentation/actions.xsd -->

<!ENTITY % maction.attlist "
  actiontype (#STRING) #REQUIRED
  selection (#POSINTEGER) '1'
  %Common.attrib;
  ">

<!ENTITY % maction.content "%Presentation-expr.class;">

<!ENTITY @ maction.type "(%maction.content;+)" "%maction.attlist;">
<!ELEMENT maction (@maction.type;)>
```

BIBLIOGRAFIA

[ABCD03] Ron Ausbrooks, Stephen Buswell, David Carlisle, Stéphane Dalmas and others. Mathematical Markup Language (MathML) Version 2.0 (Second Edition), october 2003. http://www.w3.org/TR/MathML2/

[Amo02] Nicola Amorosi, Un'estensione sintattica dei DTD per la validazione dei documenti XML, 2001

[BM01] Paul V.Biron and Ashok Malhotra. XML Schema Part 2: Datatypes, may 2001. http://www.w3.org/TR/xmlschema-2/

[BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0, february 1998. http://www.w3.org/TR/REC-xml

[BPSM00] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Second Edition), october 2000. http://www.w3.org/TR/REC-xml

[DFM+99] Andrew Davinson, Matthew Fuchs, Hedin Mette, Jain Mudita, et al. Schema for Object-Oriented XML 2.0, july 1999 . http://www.w3.org/TR/NOTE-SOX

[Fal01] David C. Fallside. XML Schema Part 0: Primer, may 2001. http://www.w3.org/TR/xmlschema-0/

[FrTh98] C. Frankston, H.S. Thompson. XML Data Reduced, July 1998. http://www.ltg.ed.ac.uk/~ht/XMLDataReduced.htm

[Har99] Harold E. R., XML Bible, Foster City, IDG Books Worldwide, 1999.

[Jel] Rick Jelliffe. Schematron. http://www.ascc.net/xml/resource/schematron/schematron.html

[JI] JSA end ISO. Relax. http://www.xml.gr.jp/relax/

[LJM+98] Andrew Layman, Edward Jung, Eve Maler, et al. XML-Data, W3C Note, January 1998. http://www.w3.org/TR/1998/NOTE-XML-Data

[RaB] AT&T Labs Research and University of Aarhus at BRICS. DSD. http://www.brics.dk/DSD/

[TBMM01] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures, may 2001. http://www.w3.org/TR/xmlschema-1/

[VAG03] Fabio Vitali, Nicola Amorosi, Nicola Gessa. Datatype -and namespace-aware DTDs, in *Extreme Markup Languages 2003: Proceedings*. 2003