Università degli Studi
Roma "La Sapienza"

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

anno accademico 2004-05

Relazione finale su progetto interno

Analizzatore di similarità tra DTD

Candidato
Giovanni Saponaro
795441

Corso
Progetto di linguaggi e traduttori

Introduzione

Obiettivo di questo progetto è calcolare quantitativamente quanto due file siano simili tra loro. Vengono infatti presi in input due file di un tipo preciso (Document Type Definition, d'ora in poi DTD) e viene restituito in output un numero percentuale che ne rappresenta il grado di similarità.

In seguito, si precisa il concetto di *similarità strutturale* ed esso viene messo in pratica tramite l'implementazione in Java di una metrica *ad hoc*, in modo da poter individuare e valutare elementi in comune e differenze tra i due file, o meglio tra i due alberi corrispondenti che vengono costruiti univocamente dal *parser*, il quale costituisce una prima parte dell'applicazione.

Il programma esegue poi un algoritmo ricorsivo sulla radice dei due alberi ottenuti nella prima fase; ciò costituisce la computazione della similarità vera e propria.

Le due fasi di computazione – rispettivamente *parsing* ed esecuzione dell'algoritmo di similarità – sono ben distinte dal punto di vista della programmazione, ma non lo sono per l'utente: visto dall'esterno, tutto avviene in modo trasparente. Dopo l'esecuzione del programma, vengono visualizzati su schermo i due alberi DTD seguiti dal punteggio di similarità calcolato.

Indice

Capitolo 1

Manuale d'uso

Se si dispone di un sistema operativo Microsoft Windows, la prima volta che si usa l'applicazione occorre lanciare i seguenti file *batch* presenti sul CDROM del progetto:

- `setenv.bat`, per impostare correttamente le variabili di ambiente `CLASSPATH` e `PATH`;
- `build.bat`, per compilare sia la grammatica `DTDParser.jj` (supponendo che JavaCC sia installato in `c:\programmi\javacc-3.2`) che i sorgenti Java del progetto.

Poi, si può eseguire il programma digitando da riga di comando `run` seguito dai nomi dei due file di input. Esempio:

```
run file1.dtd file2.dtd
```

Alternativamente, si può lanciare uno dei seguenti file *batch* con input predefiniti:

- `empty.bat`;
- `hotel.bat`;
- `normal.bat`;
- `uni.bat`.

Se il sistema operativo è UNIX, GNU/Linux o Mac OSX, fare riferimento al file `README.txt` di istruzioni sul CDROM.

Capitolo 2

Nozioni preliminari

Prima di entrare nel merito del progetto, vale a dire nella trattazione di file DTD, è bene spendere qualche parola sulla teoria dei linguaggi formali in informatica e su uno strumento ben più famoso delle DTD stesse, in funzione del quale esse sono state inventate: il linguaggio a marcatura estensibile, meglio noto come XML (dal suo nome in inglese, *Extensible Markup Language*).

2.1 Linguaggi formali

Per approfondire le nozioni che vengono in questa sede soltanto accennate, si vedano [LSF] e [LMC] in bibliografia.

Definizione 2.1 (alfabeto)**.** Chiamiamo *alfabeto* un insieme finito di simboli (o *caratteri*) e lo denotiamo con la lettera $\mathcal{A}$.

Esempio 2.2. I seguenti sono alfabeti:

- L'alfabeto latino, $\mathcal{A} = \{a \dots z, A \dots Z\} \cup \{\textit{simboli di interpunzione e spazio}\}$.
- L'insieme dei 128 simboli ASCII.
- L'insieme delle cifre arabe: $\mathcal{A} = \{0, 1, \dots, 9\}$.
- L'alfabeto binario: $\mathcal{A} = \{0, 1\}$.

Definizione 2.3 (stringa)**.** Una *stringa* o *parola* s su $\mathcal{A}$ è una sequenza finita di simboli di $\mathcal{A}$; inoltre, ϵ denota la stringa vuota.

Una stringa si ottiene concatenando simboli di un alfabeto.

Esempio 2.4. Stringhe o parole su un alfabeto $\mathcal{A}$:

- $\mathcal{A}$ è l'alfabeto latino e $s = Buongiorno$.
- $\mathcal{A}$ è l'insieme dei simboli ASCII e $s =$ `aRw #@\$PO>`.
- $\mathcal{A} = \{0, 1\}$ e $s = 101010111001$.
- $\mathcal{A} = \{\}$ e $s = \epsilon$, cioè la stringa vuota.

Definizione 2.5 (linguaggio)**.** Un *linguaggio* su $\mathcal{A}$ è un insieme, finito o infinito, di stringhe su $\mathcal{A}$.

Equivalentemente si può dire che, dato $\mathcal{A}$, un *linguaggio* è un qualsivoglia sottoinsieme di $\mathcal{A}^*$, dove l'asterisco indica la chiusura transitiva e riflessiva.

Quando si parla di *definizione di un linguaggio* si intende in effetti la definizione di un insieme di stringhe; la definizione stabilisce quali stringhe appartengono al linguaggio e quali no.

La teoria dei linguaggi formali si occupa pertanto della definizione della *sintassi* di un linguaggio ma non della sua semantica; in seguito preciseremo l'importanza di questo fatto.

Definizione 2.6 (grammatica)**.** Definiamo *grammatica* $\mathcal{G}$ la quadrupla $\langle \mathcal{A}, V_N, S, P \rangle$, dove $\mathcal{A}$ e V_N sono insiemi disgiunti finiti, non vuoti e:

1. $\mathcal{A}$ è detto l'alfabeto *terminale* di $\mathcal{G}$.
2. V_N è detto l'alfabeto *non terminale* o alfabeto delle variabili.
3. S è il simbolo iniziale o *assioma*, $S \in V_N$.
4. P è l'insieme delle *produzioni* di $\mathcal{G}$, dove una produzione è una coppia $\langle v, w \rangle$ con $v \in (\mathcal{A} \cup V_N)^+$ e contiene almeno un simbolo non terminale, mentre w è un elemento arbitrario di $(\mathcal{A} \cup V_N)^*$.

Per specificare la definizione di un linguaggio esistono diversi formalismi; i più noti sono:

- espressioni regolari;
- automi a stati finiti;
- grammatiche non contestuali in Backus-Naur Form o Extended Backus-Naur Form;
- XML Schema;
- DTD.

Tra quelli appena citati, gli strumenti formali usati concretamente in questo progetto sono le DTD e le grammatiche EBNF, rispettivamente per i file di input e per la definizione di una grammatica che riconosca tutte e sole le DTD valide.

2.2 XML

XML è un *metalinguaggio*, vale a dire che è più astratto, espressivo, potente dei comuni linguaggi formali dell'informatica, anzi serve proprio a definirne di nuovi: XML crea linguaggi a marcatura personalizzati. È dunque un metodo per mettere dei dati strutturati in file di testo.

Per comprendere cosa si intende per *dati strutturati*, conviene pensare innanzitutto a comuni applicazioni informatiche dagli usi più disparati: fogli elettronici, agende, parametri di configurazione, transazioni finanziarie, disegni tecnici e quant'altro. Ebbene, di norma i programmi che producono e gestiscono tali dati li salvano anche su disco, per cui possono usare il formato binario oppure quello testuale; quest'ultimo ha l'immediato vantaggio di essere leggibile da un comune editor di testi. XML è proprio l'insieme di linee guida per progettare file di testo atti a contenere i dati, in un modo che produca file che siano:

- facili da generare e leggere da parte di un computer;
- non ambigui;
- tali da scongiurare pericoli tipici dei formati per dati né strutturati né standard, quali la mancanza di estensibilità, la carenza di supporto per l'internazionalizzazione e la dipendenza da una particolare piattaforma hardware o da un particolare sistema operativo.

L'ascesa di XML prosegue senza sosta dal 1998, anno in cui è diventato uno standard approvato dal W3C (World Wide Web Consortium). È oggi accettato come standard *de facto* per l'archiviazione di testi su Web, in particolare per applicazioni professionali che consentono un'indicizzazione dei documenti assai articolata.

Tra i possibili impieghi di XML citiamo la gestione avanzata dei link, distribuiti in varie categorie funzionali, lo sviluppo di protocolli *platform-independent* per lo scambio dei dati, e non ultima la possibilità di far comunicare dispositivi elettronici (ad esempio PDA, agende elettroniche, navigatori satellitari etc.) tra di loro o con un computer, comunicazione questa che in passato non era affatto scontata da ottenere a causa della mancanza di standard nell'interfacciamento.

Altri progressi sono rappresentati dalla possibilità di processare automaticamente i documenti ricevuti e dalla *personalizzazione*, in quanto i dati strutturati ricevuti possono essere visualizzati grazie ai fogli di stile a cascata (*Cascading Style Sheet*, CSS), secondo modalità scelte dall'utente. Qualcosa di confrontabile con le query di SQL, benché in quel caso si abbia a che fare con basi di dati ovverosia tabelle, mentre XML tratta strutture ad albero.

2.3 DTD

Passiamo ora alle DTD, il mezzo più semplice per specificare proprietà strutturali di documenti XML.

Per inciso, un documento XML *può* contenere una DTD ma questo non è obbligatorio. Quando essa è presente, però, la DTD stessa *deve* precedere il primo elemento del documento XML, cioè l'elemento radice.

Per associare una DTD a un'istanza di documento XML, si deve inserire in testa a esso una marca che viene chiamata Document Type Declaration – e non "Definition" –, la quale a sua volta è concettualmente divisa in due: interna ed esterna. Quest'ultima è tutta scritta in un file – come suggerisce il nome – esterno e con estensione `.dtd`; ecco il tipo di file d'interesse per il nostro progetto.

Precisiamo quanto appena detto secondo la definizione delle specifiche [W3C], facendo attenzione a non confondere DTD (Document Type Definition) con Document Type Declaration.

Una XML Document Type Declaration contiene o punta a dichiarazioni di marcatura che costituiscono una grammatica per una classe di documenti. La grammatica è la DTD (Document Type Definition).

La Document Type Declaration, per quel che la riguarda, può puntare a un sottoinsieme esterno – il quale è propriamente un tipo particolare di `ENTITY` esterna – contenente dichiarazioni di marcatura, oppure può contenere tali dichiarazioni direttamente in un sottoinsieme esterno, oppure può fare entrambe le cose.

La DTD di un documento XML consiste nell'unione dei due sottoinsiemi, interno ed esterno.

Un esempio di Document Type Declaration è la seguente:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Per riassumere quanto detto finora, una DTD, che di per sé *non* è un file XML in quanto la sua sintassi interna conserva delle vestigia del vecchio formato SGML (decisamente non XML: risale agli anni '60!), serve a definire

le strutture ammesse per l'istanza di documenti XML; essa elenca quindi degli insiemi di dichiarazioni:

- di tipi di elemento;
- di attributi;
- di entità;
- di notazione.

L'insieme delle dichiarazioni di tipi di elemento sarà l'unico preso in considerazione dall'analizzatore di similarità del progetto. Tutti gli altri verranno solo considerati dal punto di vista della correttezza sintattica, ma ai fini della nostra applicazione non hanno alcun contenuto semantico e, in parole povere, vengono "dimenticati" dall'applicazione non appena passa dalla prima fase di *parsing* a quella di calcolo di similarità.

Si ricorda che una dichiarazione di tipo di elemento ha la seguente sintassi:
`<!ELEMENT nome-elemento content-model>`
Il *content model* è un'espressione regolare (particolarmente adatta a essere rappresentata come un albero, così come i documenti XML) la quale specifica la struttura degli elementi di tipo `nome-elemento`. Inoltre, il *content model* è costruito su:

- ulteriori tipi di elemento;
- `#PCDATA`, che rappresenta del testo libero senza etichette.

Gli operatori usati nel *content model*, che sono poi quelli tipici delle espressioni regolari, sono:

- sequenza ovvero `AND`, indicata con `(a, b, ..., h)`;
- alternativa ovvero `OR`, indicata con `(a | b | ... | k)`;
- opzionalità, indicata con `a?`;
- ripetizione zero o più volte, indicata con `a*`;
- ripetizione una o più volte, indicata con `a+`.

Con `a`, `b`, ... abbiamo indicato un generico *content model*.

Capitolo 3

Parsing e alberi DTD

Illustriamo ora la prima fase del programma, che consiste nel *parsing validante* di ciascuno dei due file DTD che vengono forniti dall'utente.

Ragioniamo su una singola DTD. L'applicazione ha a disposizione una classe Java `DTDParser`, che è stata generata automaticamente a partire dalla grammatica `DTDParser.jj` che definisce tutte e sole le DTD valide. Ciò significa che, qualora vi siano errori sintattici nel file, l'applicazione deve terminare immediatamente, stampando su schermo un messaggio esplicativo che segnali quale dichiarazione – ovvero quale riga – della DTD ha causato l'errore violando la grammatica.

Contestualmente a tale parsing, avviene una *traduzione guidata dalla sintassi*; è qui che, oltre agli aspetti prettamente sintattici, si aggiunge un ulteriore strato stavolta semantico, che in generale è diverso da applicazione ad applicazione.

Il fatto che la grammatica sia univoca e precisa mentre l'aspetto semantico da associare a ciascuna regola di derivazione non si possa generalizzare, ma debba essere specificato di volta in volta al seconda del programma che si deve implementare, è fondamentalmente la conferma del fatto che gli aspetti sintattici di qualsiasi programma che prenda in input delle DTD siano generalizzabili, mentre quando si tratta di *significato* si spalanca una porta che costringe a ragionare su ogni particolare problema e formalizzare algoritmi *ad hoc*, non esistendo un comun denominatore.

Questo ci porta a descrivere uno strumento usato per l'implementazione del parser di DTD: JavaCC.

JavaCC è un compiler-compiler, vale a dire un programma che riceve in input una grammatica formale con una sintassi di tipo EBNF, e genera automaticamente un analizzatore sintattico e lessicale (cioè un *parser*) top-down.

Il fatto che l'analisi sintattica sia di tipo top-down significa che l'albero

sintattico di derivazione della stringa di input (da non confondersi con gli alberi DTD *ad hoc* che andremo a costruire in fase di traduzione guidata dalla sintassi) viene generato a partire dalla radice o assioma scendendo fino alle foglie, che sono i simboli terminali della medesima stringa di input. Nel nostro caso, la stringa di input al parser sarà un intero file DTD comprensivo dei suoi caratteri di a capo.

Poiché riconoscere una stringa rispetto a una grammatica è un'operazione di per sé non deterministica, in generale questo porta a tempi esponenziali – dunque inaccettabili – per risolvere il problema del riconoscimento e di conseguenza anche quello dell'analisi sintattica. Di contro, un analizzatore sintattico deve essere molto efficiente:

1. tipicamente si esige che l'analizzatore sintattico giri in tempi *lineari* rispetto alle dimensioni della stringa di input;

2. inoltre, si richiede allo stesso analizzatore di costruire l'albero sintattico leggendo solo pochi simboli della stringa per volta; anzi, di norma solo 1 simbolo della stringa per volta, che viene detto *simbolo di lookahead*. Questo vincolo è dettato dal fatto che memoria e disco rigido dei computer non sono infiniti: sarebbe bello per un analizzatore poter leggere un grande file atomicamente ovvero "in un colpo solo", ma questo non è possibile per motivi pratici e a causa delle dimensioni notevoli (ordini di grandezza del gigabyte, 10^9 caratteri) che un file può raggiungere nelle applicazioni moderne.

È pertanto necessario che le regole di produzione della grammatica rispettino dei vincoli che la rendano adatta all'analisi sintattica. In particolare, essa è adatta allo scopo se per tale grammatica è possibile costruire un algoritmo di analisi sintattica *deterministico*.

La grammatica non può essere ambigua, altrimenti il problema diventerebbe inerentemente non deterministico perché esisterebbero tanti alberi sintattici per la stessa sequenza di input, inoltre la forma delle regole deve essere tale da evitare il non determinismo. Per JavaCC, la grammatica deve essere di tipo LL(1), e vedremo ora che questa è una condizione molto forte ma necessaria.

3.1 Grammatiche LL(k) e LL(1)

Definizione 3.1 (grammatica LL(k))**.** Una grammatica è LL(k) se ammette un riconoscitore deterministico che è in grado di costruire la derivazione canonica sinistra della stringa di input usando k simboli di lookahead e leggendo la stringa da sinistra verso destra.

La prima L di "LL(k)" indica la direzione in cui si legge la stringa: da sinistra a destra (*Left-to-right*). La seconda L indica che la stringa di input viene ricostruita per derivazione sinistra (*Leftmost derivation*), dunque il *parser* è top-down. Il numero tra parentesi indica il numero di simboli di *lookahead*.

Meccanicamente, possiamo ora dare la seguente definizione sulla falsariga della precedente.

Definizione 3.2 (grammatica LL(1))**.** Una grammatica è LL(1) se ammette un riconoscitore deterministico che è in grado di costruire la derivazione canonica sinistra della stringa di input usando 1 simbolo di lookahead e leggendo la stringa da sinistra verso destra.

Occorrono un altro paio di nozioni per far capire come si è lavorato con JavaCC.

Definizione 3.3 (ricorsione sinistra)**.** Una grammmatica presenta *ricorsione sinistra* (diretta) se esiste almeno una sua regola di produzione che ha forma

$$X \to X\alpha$$

Definizione 3.4 (prefisso comune)**.** Una grammatica presenta un *prefisso comune* se esistono almeno due sue regole nella forma

$$A \to \alpha\beta_1$$

$$A \to \alpha\beta_2$$

con $\alpha \in V_N^*$ indicante una qualsiasi forma di frase.

Ciò detto, condizioni necessarie ma non sufficienti affinché una grammatica sia LL(1) sono:

- la grammatica deve essere priva di ricorsioni sinistre;
- la grammatica deve essere priva di prefissi comuni.

Per fortuna, quando una grammatica presenta una di queste due caratteristiche che la rendono non LL(1), è sempre possibile operare trasformazioni equivalenti che portino a rispettare le due condizioni necessarie lasciando invariato il linguaggio corrispondente.

Come già sottolineato, le due condizioni di cui sopra non sono sufficienti per poter affermare che una grammatica è LL(1). Quello che si fa quando si deve scrivere un parser è:

1. presentare una grammatica non contestuale al compiler-compiler e provare a compilarla;

2. se il compiler-compiler rileva delle ambiguità perché la grammatica non è LL(1), eliminare le ricorsioni sinistre e i prefissi comuni;

3. se non ci sono alternative, aumentare il numero di *lookahead* solo per la regola di produzione che causa i problemi;

4. se ora la grammatica è LL(1) fermarsi, altrimenti tornare al primo passo.

Illustriamo a questo punto alcune parti salienti della nostra grammatica `DTDParser.jj` .

3.2 Grammatica fornita a JavaCC: `DTDParser.jj`

In sintassi EBNF, la grammatica di tutte e sole le DTD è la seguente:

```
ctypedecl   ::=  '<!DOCTYPE' S Name (S ExternalID)? S? ('['
                 intSubset ']' S?)? '>'
DeclSep     ::=  PEReference | S
intSubset   ::=  (markupdecl | DeclSep)*
markupdecl  ::=  elementdecl | AttlistDecl | EntityDecl |
                 NotationDecl | PI | Comment
```

I non terminali usati nella parte destra delle regole di derivazione hanno questo significato:

- il nonterminale `S` indica spazi bianchi (tabulazioni, invio, etc.);
- `NameChar ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender`
- `Name ::= (Letter | '_' | ':') (NameChar)*`
- `Names ::= Name (#x20 Name)*`
- `Nmtoken ::= (NameChar)+`
- `Nmtokens ::= Nmtoken (#x20 Nmtoken)*`
- `EntityValue ::= '"' ([^%&"] | PEReference | Reference)* '"' | "'" ([^%&'] | PEReference | Reference)* "'"`

- AttValue ::= '"' ([^<&"] | Reference)* '"' |
  "'" ([^<&'] | Reference)* "'"
- SystemLiteral ::= ('"' [^"]* '"') | ("'" [^']* "'")
- PubidLiteral ::= '"' PubidChar* '"' | "'" (PubidChar -
  "'")* "'"
- PubidChar ::= #x20 | #xD | #xA | [a-zA-Z0-9] |
  [-'()+,./:=?;!*#@$_%]

La grammatica `DTDParser.jj` si basa proprio sulla grammatica ufficiale delle DTD appena illustrata. Un esempio significativo è la regola qui di seguito, corredata di azioni semantiche.

```
ElementDeclaration elem_Decl()
  throws ElementDeclarationException, NodeException :
{ /* attributi: */
  Token t;
  Node cm;
}
{
  "<!ELEMENT" t=<ALPHA_STRING> cm=content_Model() <CTAG>
{
  if(cm.getName().equals("IGNORE") ||
     cm.getName().equals("") ||
     (t.image.length() >= 3 &&
      t.image.substring(0,3).toLowerCase().equals("xml"))
     )
  return null;
  return new ElementDeclaration(t.image, cm);
}
}
```

In sostanza, cosa fa la regola `elem_Decl()`? Sintatticamente, espande il non terminale `elem_Decl()` nella sequenza:

```
"<!ELEMENT" <ALPHA_STRING> content_Model() <CTAG>
```

dove `content_model()` è a sua volta un non terminale da espandere, mentre le altre tre parole sono *token*, ovvero identificatori letterali predefiniti.

Poi – aspetto pregnante della regola di produzione – c'è un contenuto semantico associato, con un'istruzione `if`.

Se il *content model* della dichiarazione di elemento corrisponde alla stringa "IGNORE" (una stringa di comodo definita da noi per fare la potatura degli alberi, come verrà spiegato in seguito) oppure corrisponde alla stringa vuota, oppure inizia per i tre caratteri "xml" a meno delle maiuscole (cosa vietata nelle specifiche XML), allora la regola deve restituire un valore `null` al non terminale padre, che saprà come gestirlo opportunamente. In tutti gli altri casi, ovvero se la dichiarazione è valida, deve restituire un'istanza della struttura dati `ElementDeclaration` contenente il nome dell'elemento e la struttura ad albero del *content model*.

Dal momento che già abbiamo dato cenno di alberi e di strutture dati personalizzate per il progetto, sarà bene ora illustrarle. Ricordiamo che il fine ultimo di questa prima fase (fase di *parsing*) dell'applicazione è quello di rappresentare una DTD come un albero in Java, sufficientemente semplice per essere visitato da un algoritmo e per subire operazioni, ma al tempo stesso che preservi il significato del file DTD in questione.

3.3 Alberi DTD

Definizione 3.5 (albero)**.** Dato un insieme $\mathcal{N}$ di nodi, un *albero* è definito in modo induttivo in questo modo:

- $v \in \mathcal{N}$ è un albero;
- se $T_1, \ldots, T_n$ sono alberi, allora $(v, [T_1, \ldots, T_n])$ è un albero [1].

Definizione 3.6 (albero etichettato)**.** Dato un insieme $\mathcal{A}$ di etichette e indicando con $N(T) \subseteq \mathcal{N}$ l'insieme dei nodi di un albero T , un *albero etichettato* è una coppia (T, φ), dove φ è una funzione etichettatrice tale che per ogni $v \in N(T), \varphi(v) \in \mathcal{A}$.

Le etichette usate per etichettare un albero appartengono a un insieme di marche di elemento, insieme che indichiamo con $\mathcal{EN}$; chiamiamo inoltre $\mathcal{V}$ l'insieme delle foglie del generico albero.

Per rappresentare in memoria l'albero di una DTD ai fini di poter applicare alla sua radice vari algoritmi, si è scelto di scrivere da zero una struttura dati molto semplice: la struttura `Node` (nodo), avente le seguenti variabili:

```
private String name;
public Node[] children;
```

[1] Per alleggerire la notazione, d'ora in poi denoteremo con C i sottoalberi di T, nel senso che $C = [T_1, \ldots, T_n]$, qualora sia rilevante sapere che T è un sottoalbero interno di una DTD.

```
<!ELEMENT biblioteca (libro+,registro?,impiegato)>
<!ELEMENT libro (autore,titolo)>
<!ELEMENT registro (#PCDATA)>
<!ELEMENT impiegato (#PCDATA)>
<!ELEMENT autore (#PCDATA)>
<!ELEMENT titolo (#PCDATA)>
```

Figura 3.1: Esempio di DTD

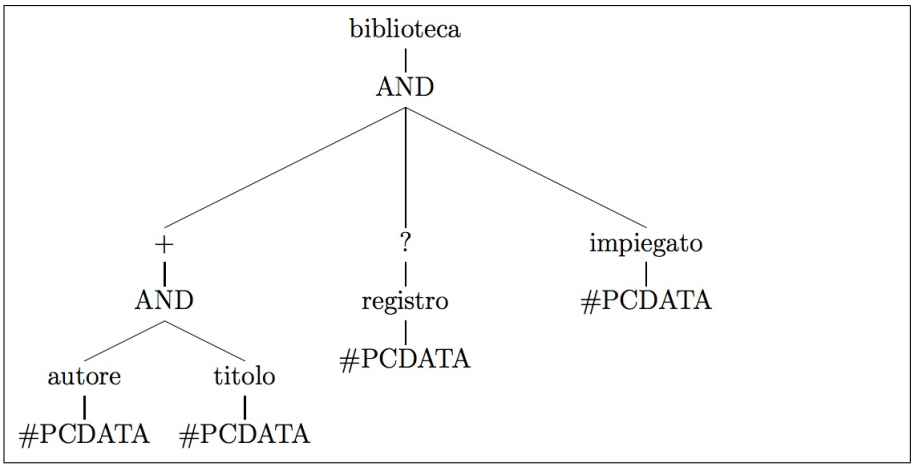

Figura 3.2: Esempio di albero DTD

Tali variabili rappresentano rispettivamente il nome del nodo e l'elenco dei suoi figli come *array* di `Node`, eventualmente vuoto.

Aggiungendo nodi in modo gerarchico tra loro grazie a un opportuno metodo

```
void addChild(Node n, int numero_del_figlio)
```

si ottiene proprio una struttura dati ad albero. In particolare, si tratta di un albero n-ario ordinato.

Per una spiegazione degli altri metodi presenti nella classe `Node` si veda l'ultimo capitolo.

3.4 Semplificazioni

Una delle caratteristiche di XML consiste nelle tante opzioni possibili quando si modellano i sottoelementi di un documento nelle DTD. Nella stesura del *content model* di una DTD, c'è a disposizione una certa gamma di operatori, che sono quelli tipici delle espressioni regolari.

Per ciascun sottoelemento è possibile specificare se esso è opzionale ("?"), se esso può comparire molteplici volte ("$*$" se zero o più volte, e "+" se una o più volte), se alcuni sottoelementi sono alternativi tra di loro ("|" cioè OR logico) oppure sono raggruppati in sequenze in cui tutti devono comparire ("," cioè AND logico).

È nostra intenzione concentrarci su un sottoinsieme di DTD, per semplicità. Facciamo pertanto le seguenti ipotesi:

- consideriamo solo le dichiarazioni di elemento, eventualmente annidate;
- scartiamo gli attributi, i quali del resto possono essere pensati come un caso particolare degli elementi;
- ne consegue che di fatto consideriamo solo elementi non vuoti; ciò non è particolarmente limitativo né deve sconvolgere in quanto gli elementi vuoti sono sempre trattabili al pari di elementi `#PCDATA` col vincolo di avere contenuto nullo;
- non prendiamo in considerazione sequenze di operatori unari ("`?`", "`*`", "`+`") dal momento che esiste sempre una rappresentazione equivalente più concisa che usi un singolo operatore;
- ogni nodo corrisponde a un elemento, o a un tipo di elemento, o a un operatore.

L'albero che il *parser* costruisce dalla DTD che riceve in ingresso è un albero etichettato sui nodi.

Introduciamo l'insieme di operatori $\mathcal{OP} = \{$`?`, `*`, `+`, `AND`, `OR`$\}$ per poter rappresentare vincoli su opzionalità, ripetizioni, sequenze, alternative tra elementi. L'operatore `AND` causa una sequenza di elementi tutti obbligatori, l'operatore `OR` un'alternativa di elementi (una tra le alternative deve essere presente), l'operatore "`?`" l'opzionalità, mentre gli operatori "`*`" e "`+`" rappresentano ripetizione di elementi (zero o più volte, una o più volte, rispettivamente).

In tutti gli alberi DTD, l'etichetta del nodo radice appartiene a $\mathcal{EN}$ e tale nodo possiede un solo arco uscente.

Osservazione 3.7. Sotto le ipotesi fatte, gli unici nodi con più di un arco uscente possono essere solamente quelli etichettati `AND` o `OR`, e mai nodi con nome di elemento.

Infine, si ha che tutti i nodi etichettati da *tipi* di elemento sono foglie dell'albero. Sia $\mathcal{ET}$ l'insieme dei possibili tipi di elemento base; per la nostra applicazione, $\mathcal{ET} = \{$`#PCDATA`, `ANY`$\}$.

Con l'armamentario formale che abbiamo definito, precisiamo ora la nozione di DTD vista come albero.

Definizione 3.8 (albero DTD)**.** Una DTD è un albero etichettato (T, φ_T) definito dall'insieme di etichette $(\mathcal{EN} \cup \mathcal{ET} \cup \mathcal{OP})$, e gode delle seguenti proprietà:

1. T è nella forma $(v, [T'])$, con $\varphi_T(v) \in \mathcal{EN}$;
2. per ogni sottoalbero (v, C) di T, $\varphi_T(v) \in (\mathcal{EN} \cup \mathcal{OP})$;
3. per ogni sottoalbero foglia v di T, $\varphi_T(v) \in \mathcal{ET}$;
4. per ogni sottoalbero (v, C) di T, se $\varphi_T(v) \in \{$`OR`, `AND`$\}$ allora $C = [T_1, \ldots, T_n]$, con $n > 1$;
5. per ogni sottoalbero (v, C) di T, se $\varphi_T(v) \in (\{$`?`, `*`, `+`$\} \cup \mathcal{EN})$ allora $C = [T']$.

L'introduzione degli operatori $\mathcal{OP} = \{$`?`, `*`, `+`, `AND`, `OR`$\}$ permette di rappresentare univocamente e formalmente la struttura di *tutte* le DTD possibili. Introdurre l'operatore `AND` è stato necessario per poter distinguere tra il caso di un elemento contenente un'alternativa tra sequenze (per esempio `<!ELEMENT a(b|c1,c2))>`) e il caso di un elemento contenente invece l'alternativa tra tutti gli elementi di una sequenza (`<!ELEMENT a(b|c1|c2)>`).

Capitolo 4

Metriche di similarità

La similarità, che gioca un ruolo di notevole importanza in diversi campi di ricerca, funge da principio di organizzazione e catalogazione in molti aspetti pratici: è usata dagli uomini per classificare concetti ed enti anche astratti, oltre che per *formare* nuovi concetti e generalizzazioni.

Da un punto di vista informatico si distinguono astrattamente diversi possibili tipi di similarità misurabile:

1. a livello di dati;
2. a livello di tipi di dato, cioè di *schemi* o modelli o strutture, a seconda del dominio di applicazione in cui si opera;
3. a entrambi i livelli, confrontando tanto i dati quanto i loro tipi.

Il primo modo è utile se si vogliono creare *cluster* di informazioni relativi allo stesso argomento e classificare tali informazioni; ad esempio, raggruppare immagini grafiche che riguardano lo stesso soggetto.

Il secondo approccio di similarità possibile, relativo ai tipi, diventa rilevante per lo *schema clustering* oppure nell'ambito di integrazioni di tanti schemi che descrivono sì lo stesso tipo di informazione ma usano strutture dati diverse. È questo il concetto su cui opera il progetto in questione.

Infine, la similarità sia di dato che di tipo può essere usata per identificare un generatore di dati comune e per applicare a questi ultimi le proprietà specificate nel tipo.

Ortogonalmente a questa prima classificazione, la similarità si può concentrare sui contenuti oppure sulle strutture di dati coinvolti.

Poiché una quantità enorme di informazioni scambiate sul Web sta man mano aderendo al formato XML e le applicazioni hanno sempre più bisogno di "parlare XML" (ricevere, accedere, manipolare documenti XML con vincoli

e condizioni più o meno rilassati e risultati più o meno approssimati), è logico comprendere come la possibilità di valutare la similarità sia utile e attuale.

4.1 Similarità tra DTD: princìpi di base

Confrontando due file DTD ovvero i due alberi corrispondenti, può capitare che alcuni elementi presenti nella prima DTD manchino nella seconda oppure che la prima ne abbia in eccesso rispetto all'altra, o viceversa. Non solo: gli elementi, pur essendo comuni, potrebbero essere disposti in un ordine (gerarchia) differente tra i due alberi. E ancora: un sottoinsieme delle marche di una delle DTD potrebbero essere non proprio identiche ma concettualmente vicine a marche dell'altro file, secondo relazioni definite in un dizionario. Questo implicherebbe la necessità di gestire, oltre all'uguaglianza tra nomi di marche, anche la similarità semantica tra essi.

Il lettore avrà già intuito che il problema di definire una metrica per stabilire la similarità tra due DTD è arduo, e il grado di raffinatezza dell'algoritmo che si vuole progettare è inversamente proporzionale alla facilità di concepirlo, pianificarlo, gestirlo informaticamente a livello algoritmico e di codice. Non c'è limite alla granularità che potremmo implementare, a scapito però di complicazioni concettuali.

Per fini pratici scegliamo una raffinatezza limitata operando alcune ipotesi semplificative. Tanto per cominciare, consideriamo solo l'uguaglianza tra marche e mai una vicinanza concettuale tra esse: o una certa marca è presente in entrambe le DTD in input (e in questo caso il grado di similarità restituita sarà maggiore di 0%), oppure due marche in posizioni corrispondenti hanno nomi diversi. Non ci sono vie di mezzo.

Un'altra semplificazione che operiamo è quella di confrontare solo le marche di tipo `ELEMENT` e di scartare a priori i vari `ATTLIST` ed `ENTITY`. Del resto, questa scelta era stata messa in atto già a livello di *parsing*.

4.2 Come valutare le differenze

L'algoritmo di similarità, in buona sostanza, consiste nell'identificazione e successiva valutazione numerica di:

- elementi comuni, cioè che appaiono in entrambe le DTD;
- elementi differenti: quelli che compaiono solo in una delle due DTD.

Per quanto riguarda la valutazione degli elementi, questa avviene prendendo in considerazione principalmente due fattori.

1. L'algoritmo assegna un *peso* a seconda del *livello* a cui gli elementi comuni sono situati nei due alberi da confrontare. Elementi a livelli più alti, cioè vicini alla radice del documento, devono possedere un coefficiente di peso maggiore dei sottoelementi più periferici, quelli profondamente annidati nell'albero, vicino alla sua fronda.

2. Successivamente, la valutazione prende in considerazione la *struttura* degli elementi in comune. Trovare in comune due elementi "complessi" – i quali sono a loro volta grandi sottoalberi – deve premiare di più che trovare in comune due elementi foglia.

Introdurremo ora alcune funzioni matematiche ricorsive di ausilio per questa prima parte dell'algoritmo, quella di *parsing*.

4.3 La funzione livello

Come già accennato, è auspicabile che la misura di similarità sia coerente col fatto che elementi "in alto" nell'albero, dunque vicini alla radice, abbiano un peso maggiore di quelli più periferici.

Il concetto di *livello* di un elemento è strettamente collegato a quello della sua altezza nell'albero, oppure –specularmente– della sua profondità.

Dato un albero T che rappresenta una DTD, definiamo il livello di T come il numero dei nodi, *non contando gli operatori*, toccati lungo il massimo cammino possibile in T, dalla radice verso una foglia.

Il motivo per cui i nodi in $\mathcal{OP}$ non contano ai fini della funzione livello è il seguente: i nodi operatore di una DTD influenzano soltanto la *larghezza* degli alberi corrispondenti alla classe di XML associati, non la loro profondità; siccome a noi in questo momento interessa la funzione livello, che associa a ogni nodo un valore dipendente proprio dalla sua profondità nell'albero, ecco spiegato perché questi operatori non incrementano il conteggio lungo i cammini radice-foglia.

Formalizziamo quanto appena stabilito.

Definizione 4.1 (funzione livello)**.** Sia $T = (v, [T_1, \ldots, T_n])$ un sottoalbero di una DTD. Allora la funzione *livello* è così definita:

$$level(T) = \begin{cases} 1 + \max_{i=1}^{n} level(T_i) & \text{se } T \in \mathcal{EN} \\ \max_{i=1}^{n} level(T_i) & \text{se } T \in \mathcal{OP} \\ 0 & \text{altrimenti.} \end{cases}$$

Ricordiamo che $\mathcal{EN}$ è l'insieme delle marche elemento, mentre $\mathcal{OP}$ denota gli operatori {`?`, `*`, `+`, `AND`, `OR`}. Nella definizione precedente, inoltre, T_i denota il generico eventuale sottoalbero figlio di T.

4.4 Rilevanza di un nodo

Chiamiamo $l = level(D)$ il livello di una DTD D (del suo nodo radice) e $\gamma = 2$ un *fattore di rilevanza* di un livello dell'albero di D rispetto al successivo.

Il nodo radice ha dunque rilevanza $r_l = \gamma^l$; quelli del livello inferiore sono divisi per un fattore γ e hanno dunque rilevanza γ^{l-1}, e così via.

Definizione 4.2 (rilevanza)**.** Per un generico livello i dell'albero corrispondente a una DTD D, la rilevanza vale γ^{l-i}.

Dopo aver parlato di *level*, andiamo ora a definire una seconda, fondamentale funzione che associa un valore a ciascun nodo.

4.5 La funzione peso

Abbiamo scritto di voler considerare la più semplice struttura possibile di una DTD. Questa scelta è particolarmente sensata per applicazioni che lavorino, invece che con sole DTD in input come la nostra, con un mix di file DTD e file XML in ingresso; si veda ad esempio [BGM].

Per motivi di semplicità, non prendiamo in considerazione gli elementi opzionali o ripetuti. Inoltre, per quanto riguarda elementi alternativi (in `OR` logico), l'algoritmo dovrà considerare *solo uno* di tali elementi: presumibilmente quello con la struttura (sottoalbero) più semplice.

Si capisce già che la seguente funzione ricorsiva assocerà un valore diverso a ogni nodo a seconda che esso sia una marca di elemento oppure un vincolo operatore.

Definizione 4.3 (funzione peso)**.** Sia D una DTD con r_l rilevanza corrispondente. Allora si definisce la seguente funzione *peso*[1]:

$$weight(D, r_l) = \begin{cases} r_l & \text{se } D \in \{\texttt{\#PCDATA}, \texttt{ANY}\} \\ 0 & \text{se } D \in \{\texttt{*}, \texttt{?}\} \\ weight(D', r_l) & \text{se } D = \text{“}\texttt{+}\text{”};\ D' \text{ è il figlio di } D \\ \sum_{i=1}^{n} weight(D_i, r_l) & \text{se } D = \text{“}\texttt{AND}\text{”};\ D_i \text{ sono i figli di } D \\ \min_{i=1}^{n} weight(D_i, r_l) & \text{se } D = \text{“}\texttt{OR}\text{”};\ D_i \text{ sono i figli di } D \\ \sum_{i=1}^{n} weight(D_i, \frac{r_l}{\gamma}) + r_l & \text{altrimenti.} \end{cases}$$

Cominciamo a illustrare l'ultimo caso della funzione, il più importante e delicato. Dice questo: “se il nodo di cui occorre calcolare il peso è una marca

[1] Con un leggero abuso di notazione, indichiamo con D non la DTD nel suo insieme ma il nodo che inizialmente è il nodo radice della DTD stessa, mentre nelle chiamate ricorsive D sarà meramente la radice del sottoalbero attuale.

– un nome di elemento –, allora aggiungi r_l al subtotale della similarità poi applica l'algoritmo ai vari figli, *dividendo per un fattore* $\gamma = 2$ il secondo parametro (r_l), che dalla prossima chiamata, inclusa, è quindi aggiornato".

Il significato del quarto e del quinto caso si intuisce facilmente. Se il nodo di cui vogliamo calcolare il peso è un vincolo `AND`, allora il suo peso sarà uguale alla somma dei pesi di tutti i figli, mantenendo il parametro r_l di cui sopra immutato. Se il nodo è invece un `OR`, il suo peso è il minimo dei pesi di tutti i figli del nodo `OR` stesso, anche qui lasciando il parametro di rilevanza immutato; esplorare solo il sottoalbero figlio con peso minimo corrisponde a selezionare il sottoalbero con la struttura DTD più semplice possibile.

Continuiamo ad analizzare le varie possibilità della funzione dal basso verso l'alto. Il terzo caso dice che il peso di un nodo di vincolo "`+`" è il peso del suo figlio, che tra l'altro è per forza di cose un figlio unico, per come abbiamo costruito l'albero a partire dalla DTD in fase di *parsing*.

La seconda opzione della funzione peso è quella che essa incontri un nodo "`*`" o "`?`"; in questi casi, coerentemente con le ipotesi semplificative sugli alberi DTD, appiattiamo il vincolo di cardinalità al valore 0, potando di fatto il nodo in questione con tutti i suoi eventuali figli.

Infine, il primo caso elencato della funzione considera i nodi foglia, che per ipotesi possono essere solo `#PCDATA` o `ANY`. A essi la funzione associa direttamente il valore attuale del parametro di rilevanza r_l, che sarà tanto più piccolo quanto la foglia è profonda, in quanto a ogni livello si ha un aumento esponenziale del denominatore dell'espressione. La rilevanza del primo livello, in altre parole della radice, vale $r_l = \gamma^l$; la rilevanza del secondo livello $r_l/\gamma = r_l/2$; quella del terzo livello $r_l/\gamma^2 = r_l/4$ e così via. Pertanto, trovare una marca di elemento in comune tra i due alberi a un livello i-esimo aggiungerà al subtotale di similarità una quantità r_l/γ^{i-1}, il che rispetta uno degli obiettivi iniziali: più un nodo è vicino alla radice, più esso pesa nel computo in positivo o in negativo (a seconda che sia in comune tra le due DTD o meno, rispettivamente).

La figura 4.1 a pagina 21 rappresenta un esempio dei valori risultanti dalle due funzioni *level* e *weight* su ogni nodo.

4.6 Ruolo del parametro γ

Al crescere del valore assegnato al numero naturale γ, l'algoritmo assegna maggiore rilevanza a elementi in comune che si trovano a un livello alto – vicino alla radice – piuttosto che a elementi periferici, prossimi alla fronda dell'albero, in definitiva strutturalmente poco rilevanti.

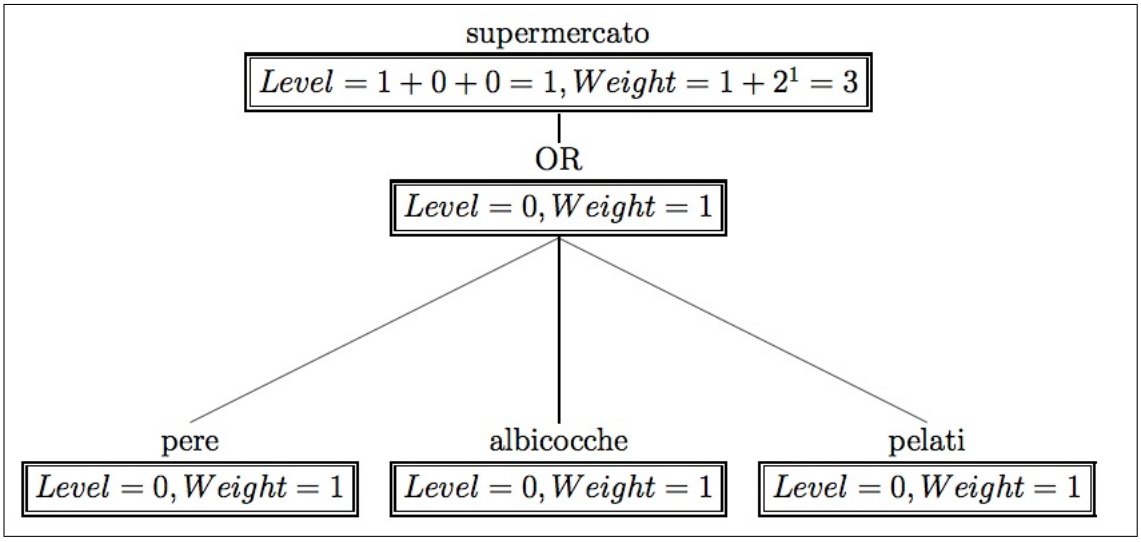

Figura 4.1: Esempio di livelli e pesi

Se poniamo $\gamma = 1$, tutte le informazioni sono valutate rilevanti alla stessa maniera, perciò il fatto che un elemento appaia al terzo o al quarto livello della struttura dati non conta.

Viceversa, la scelta di porre $\gamma = 2$ fa sì che i nodi di un certo livello abbiano rilevanza doppia rispetto ai loro figli. Con $\gamma \geq 3$ il divario sarebbe ancora maggiore.

In questa sede noi supporremo sempre che γ valga 2. L'utente che volesse fare esperimenti con altri valori del parametro può modificare la riga

```
private int GAMMA = 2
```

all'inizio del file `DTDSimilarity.java`.

4.7 Casi particolari di input

Prima di illustrare il caso generale dell'algoritmo, adatto a computare la similarità di qualsiasi coppia di DTD, citiamo due situazioni limite.

Il primo caso particolare su cui vale la pena spendere qualche parola è quello in cui i due alberi DTD ottenuti dal *parser* siano vuoti, ovvero contengano zero nodi. Questo può capitare se i due file di testo forniti in input sono proprio file vuoti di 0 byte, ma non solo: tenendo a mente che stiamo prendendo in considerazione le dichiarazioni di elemento, un albero vuoto può risultare anche da un file DTD che contiene sì zero dichiarazioni di elemento, ma magari ne contiene di altre: di esse viene controllata la

correttezza sintattica rispetto alla grammatica, ma poi vengono scartate in quanto prive di contenuto semantico nel nostro caso. Un esempio di tale file è dato dalla seguente DTD:

```
<!-- This DTD generates a totally empty tree! -->
<!ATTLIST Blah id ID #REQUIRED
          num NMTOKEN #IMPLIED>
<!ENTITY STUFF "Some more junk follows">
<!-- over, and over, and beyond... -->
```

Ebbene, tale input – che *è* valido secondo le specifiche W3C – genera in base alla nostra grammatica `DTDParser.jj` un albero DTD vuoto poiché non ha dichiarazioni di tipo di elemento. Come si deve comportare l'applicazione se le vengono forniti in input due DTD "vuote", nell'accezione del termine appena spiegata? Deve rispondere che il punteggio di similarità calcolato è pari al massimo: 100%. Dovremo tener conto di ciò con un `if` apposito nell'implementazione Java dell'algoritmo.

Una seconda situazione particolare di fronte a cui il programma si può trovare è la seguente: in input vengono passati due file DTD identici a meno dell'ordine alfabetico di nodi fratelli. L'applicazione deve anche in questo caso restituire il punteggio 100%. Il motivo è abbastanza sottile: i due file, in effetti, *parlano delle stesse cose* ovvero hanno il medesimo contenuto semantico.

Un modo furbo per far sì che l'algoritmo tratti due alberi diversi ma semanticamente uguali come oggetti identici è riordinare alfabeticamente tutte le generazioni di fratelli, in modo ricorsivo a partire dalla radice, per entrambi gli alberi. Facendo questo riordinamento a priori – prima dell'esecuzione dell'algoritmo di similarità, ma dopo di aver stampato su schermo un disegno dei due alberi: qui sta il trucco –, siamo certi di ottenere il punteggio 100% come è giusto che sia.

4.8 Caso generale dell'algoritmo di similarità

La metrica di similarità assegna dei punti ogni volta che, confrontando i due alberi, trova elementi in comune ovvero nodi con lo stesso nome. In estrema sintesi funziona così:

1. Confronta i nomi delle due radici. Se esse si chiamano in modo diverso, si può dedurre che le DTD trattano sicuramente concetti differenti perché non hanno strutture comuni; pertanto, assegna 0% come punteggio

e termina. Se invece il confronto tra le stringhe dà esito positivo, vai al passo successivo.

2. Assegna un certo punteggio iniziale e continua a visitare i rispettivi figli del nodo corrente, confrontandoli sulla base dello stesso principio del passo precedente.

Abbiamo parlato informalmente di "certo punteggio iniziale", e bisogna quantificarlo. Qui entrano in gioco le funzioni *level* e *weight* precedentemente introdotte, oltre al valore di rilevanza r_l inizialmente pari a γ^l. Finalmente, enunciamo l'algoritmo di similarità nella sua versione definitiva.

1. Scegli tra le due radici quella con la rilevanza massima. Chiamiamo il suo albero "albero pesante" e N la corrispondente radice.

2. Somma i pesi di tutti i nodi dell'albero pesante e chiama questo valore "massima similarità possibile".

3. Confronta N con la radice dell'altro albero (quello leggero) e in caso di esito negativo fermati restituendo 0%. Nel caso positivo, invece, vai al prossimo passo.

4. Aggiungi[2] $weight(N)$ al subtotale e confronta ricorsivamente i nomi dei figli. Chiama il risultato totale di questo passo "similarità relativa".

5. Restituisci il risultato della divisione $\frac{\text{similarità relativa}}{\text{massima similarità possibile}}$ con al più due cifre decimali.

L'unica novità introdotta nella definizione appena data è in realtà quella dell'ultimo passo, la divisione. La sua ragion d'essere è puramente estetica, di comodo: vogliamo un risultato normalizzato in percentuale per poterlo più facilmente quantificare nella nostra comune percezione umana, tutto qua. In altre parole, abbiamo scelto arbitrariamente di ottenere valori tipo "82.03%" piuttosto che un equivalente "1584/1931". Si noti che tanto il numeratore quanto il denominatore della frazione $\frac{\text{similarità relativa}}{\text{massima similarità possibile}}$ hanno valori interi, mentre il risultato normalizzato in generale non gode di questa proprietà.

[2]Ecco quello che avevamo chiamato "un certo punteggio".

Capitolo 5

Implementazione Java

Mostriamo ora alcune scelte progettuali compiute in fase di programmazione. I sorgenti del progetto sono divisi nei seguenti quattro *package*, accanto ai quali elenchiamo qui di seguito le classi Java appartenenti.

```
similarity:  DTDSimilarity
|
|\
| \
|  \___ similarity.data:  ElementDeclaration
|                         Node
|\
| \
|  \___ similarity.dtdparser:  DTDParser
|                              DTDParserConstants
|                              DTDParserTokenManager
|                              ParseException
|                              SimpleCharStream
|                              Token
|                              TokenMgrError
 \
  \
   \___ similarity.error:  ElementDeclarationException
                           NodeException
```

5.1 La classe `DTDSimilarity`

La directory corrispondente a questo package contiene solo un file, che però è il più importante e delicato di tutta l'applicazione, anche perché è quello che

corrisponde alla classe Java `DTDSimilarity` che poi andrà eseguita da riga di comando. Illustriamo dunque le parti salienti di `DTDSimilarity.java`.

```
import similarity.dtdparser.*;

public class DTDSimilarity {
```

La seguente variabile è una sola ed è dichiarata statica perché noi trattiamo sì due file contemporaneamente, però la classe `DTDParser` che abbiamo a disposizione opera in input e in output con un file solo.

```
        private static DTDSimilarity sim;
```

Questi oggetti, ciascuno dei quali è un albero di zero o più `Node` annidati, conterranno il risultato del *parsing* dei file di input, dopo aver gestito le possibili eccezioni delle corrispondenti coppie di istanze di `File` e `FileInputStream`, rispettivamente.

```
        private static Node root1, root2;
```

Base del parametro esponenziale *rilevanza* $r_l = \gamma^l$ già ampiamente discussa in precedenza. Il parametro viene sfruttato durante il calcolo della similarità verso la fine di questo stesso file.

```
        private int GAMMA = 2;

        private double similarity(Node tree1, Node tree2) throws
         NodeException
        {
```

Questa variabile in virgola mobile a doppia precisione conterrà il risultato finale dell'algoritmo.

```
                result = 0;
```

Caso particolare: i due alberi consistono in due nodi singoli, cioè senza figli. Il corrispondente risultato dev'essere quello massimo: 100%.

```
                if(tree1.getNumChildren() == tree2.getNumChildren() &&
                   tree1.getNumChildren() == 0 &&
                   tree1.getName().equals(tree2.getName())) {
                        result = 100;
                        return result;
                }
```

Dopo aver visualizzato i due alberi su schermo, riordina alfabeticamente tutte le generazioni di fratelli, così da rispettare il caso particolare secondo cui due alberi semanticamente uguali ma sintatticamente diversi devono restituire punteggio massimo.

```
/* bubble sort siblings */
tree1.sortSiblings();
tree2.sortSiblings();
```

Se il primo albero è quello "pesante", allora calcola il valore "massima similarità possibile" col seguente *escamotage*: invoca l'algoritmo di similarità di caso generale passando `tree1` sia come primo che come secondo argomento. Se invece l'albero "pesante" è il secondo, fai la cosa speculare.

```
if(tree1.weight(relevance1) > tree2.weight(relevance2))
        max_sim_possible =
         general_similarity(tree1, tree1);
else
        max_sim_possible =
         general_similarity(tree2, tree2);
```

Il valore "similarità relativa" è dato dall'invocazione dell'algoritmo generico con entrambi gli alberi per argomento. Si noti che l'ordine tra i due è in questo caso ininfluente perché abbiamo asserito che un nodo in più conta esattamente quanto l'opposto di un nodo in meno.

```
double rel_result = general_similarity(tree1, tree2);
```

Normalizza il risultato in percentuale.

```
return (rel_result/max_sim_possible) * 100;
}
```

Ecco ora il metodo di similarità generale, fulcro non solo di questa classe Java ma di tutta l'applicazione.

```
private double general_similarity(Node tree1, Node tree2) {
```

Variabile interna al metodo.

```
double res = 0;
```

Coerentemente con quanto scritto più volte nel capitolo precedente: se le due radici si chiamano in modo differente, termina subito restituendo 0%.

```
            if(! tree1.getName().equals(tree2.getName()) )
                    return res;
```

Poni il risultato temporaneo uguale a $weight(T_{pesante}, r_l)$.

```
            res = Math.max(tree1.weight(relevance1),
             tree2.weight(relevance2));
```

Se entrambi gli alberi sono dei padri – cioè se è lecito usare la ricorsione –, chiama ricorsivamente l'algoritmo per tante volte quanti sono i figli dell'albero con meno figli tra i due (invocare l'algoritmo per un numero superiore di volte non avrebbe senso).

```
            if(tree1.children != null && tree2.children != null) {
                    for(int i = 0;
                     i < Math.min(tree1.children.length,
                      tree2.children.length);
                      i++) {
                            res +=
                             general_similarity(tree1.children[i],
                              tree2.children[i]);
                    } /* end for */
            }
            return res;
    }
```

Omettiamo il metodo `main` che non fa altro che controllare la correttezza dei parametri passati a riga di comando.

```
}
```

5.2 Strutture dati

Il *package* `similarity.data` contiene le due strutture dati su cui si basa l'applicazione, soprattutto per quanto riguarda la prima fase di *parsing* che si occupa di costruirle e annidarle in fase di traduzione guidata dalla sintassi. La fase di similarità, invece, opera visite nei due alberi precedentemente costruiti, alberi che altro non sono che insiemi di nodi collegati.

Cominciamo proprio a illustrare la classe `Node`, che è necessariamente ricca di metodi ausiliari.

```
package similarity.data;

import similarity.error.NodeException;

public class Node {
```

Ogni nodo ha una stringa `name` e un *array* di altri nodi figli che può essere nullo nel caso in cui il nodo sia una foglia oppure se i figli non gli sono stati ancora collegati.

```
        public Node[] children;
        private String name;
```

Non riportiamo in questa sede i seguenti metodi, piuttosto banali:

- `public Node(String n) throws NodeException`
- `public String getName()`
- `public void setName()`
- `public void addChild(Node n, int i) throws NodeException`

La seguente funzione verifica ricorsivamente se nell'albero sottostante al nodo `this` (compreso) è presente o meno un nodo che si chiama come l'argomento passato.

```
        public boolean isPresent(String s) {
                if(this.getName().equals(s))
                        return true;
                if (children != null)
                        for (int c = 0; c < children.length; c++)
                                if(children[c].isPresent(s))
                                        return true;
                return false;
        }
```

Omettiamo le funzioni `sortSiblings` e `bubbleSort` le quali, dato il nodo corrente `this` che deve essere un padre, mettono in ordine alfabetico i figli, i nipoti, etc. Il metodo di ordinamento *bubble sort* ha il pregio della semplicità ma come difetto ha un alto costo temporale: $\mathcal{O}(n^2)$ nella dimensione n dell'input (si veda [ItA]); ciononostante, con *array* di pochissimi elementi come nel nostro caso la scelta di usare questo algoritmo è accettabile.

Anche la funzione `height`, usata solo da altri metodi di questa classe e non da classi esterne, è piuttosto semplice pur essendo ricorsiva.

Segue un metodo molto delicato per attaccare due alberi tra loro, metodo che a sua volta chiama altre funzioni ausiliarie.

```
public void attach(Node tree) throws NodeException {

        if(children == null) { /* "this", aka "res"
         for the client, is just 1 node */
                if(this.getName().equals( tree.getName() )) {
                        this.copyChildren(tree);
                }
        }

        else {
                Node n;
                for(int c = 0; c < children.length; c++) {
                        n = children[c];
                        n.replaceLeaf(tree);
                        n.attach(tree);
                }
        }
}

public void replaceLeaf(Node tree) throws NodeException {
        if(this.height() == 0) { /* n is a leaf */
                if(this.getName().equals( tree.getName() ))
                        this.copyChildren(tree);
        }
}
```

Riportiamo ora le due funzioni chiave cui abbiamo dedicato molte spiegazioni nel capitolo precedente: *level* e *weight*.

Per cominciare, ricordiamo che $level(T) = \begin{cases} 1 + \max_{i=1}^{n} level(T_i) & \text{se } T \in \mathcal{EN} \\ \max_{i=1}^{n} level(T_i) & \text{se } T \in \mathcal{OP} \\ 0 & \text{altrimenti.} \end{cases}$

```
public int level() {
        if (this.height() == 0) /* this is a leaf */
                return 0;
        else if (this.isOP()) {
```

```
                Node n = this.children[0];
                int lev_max = n.level();
                int i = this.children.length;
                for (int c = 1; c < i; c++) {
                        n = this.children[c];
                        int lev = n.level();
                        if (lev > lev_max)
                                lev_max = lev;
                }
```

Poiché siamo nel caso $T \in \mathcal{OP}$, non incrementiamo il valore del livello e calcoliamo quest'ultimo sulla base del massimo livello tra i nodi figli, valore che è stato calcolato poche righe più sopra.

```
                return lev_max;
        }
```

Se invece $T \in \mathcal{EN}$...

```
        else {
                Node n = this.children[0];
                int lev_max = n.level();
                int i = this.children.length;
                for (int c = 1; c < i; c++) {
                        n = this.children[c];
                        int lev = n.level();
                        if (lev > lev_max)
                                lev_max = lev;
                }
```

...incrementiamo il valore e chiamiamo ricorsivamente il metodo stesso.

```
                return 1 + lev_max;
        }
}

public boolean isOP() {
        return  this.getName().equals("?") ||
                this.getName().equals("*") ||
                this.getName().equals("+") ||
                this.getName().equals("AND") ||
                this.getName().equals("OR");
}
```

Segue la funzione peso, con una differenza rispetto alla definizione teorica

$$weight(D, r_l) = \begin{cases} r_l & \text{se } D \in \{\texttt{\#PCDATA}, \texttt{ANY}\} \\ 0 & \text{se } D \in \{\texttt{*}, \texttt{?}\} \\ weight(D', r_l) & \text{se } D = \text{“}\texttt{+}\text{”};\ D' \text{ è il figlio di } D \\ \sum_{i=1}^{n} weight(D_i, r_l) & \text{se } D = \text{“}\texttt{AND}\text{”};\ D_i \text{ sono i figli di } D \\ \min_{i=1}^{n} weight(D_i, r_l) & \text{se } D = \text{“}\texttt{OR}\text{”};\ D_i \text{ sono i figli di } D \\ \sum_{i=1}^{n} weight(D_i, \frac{r_l}{\gamma}) + r_l & \text{altrimenti.} \end{cases}$$

La differenza è questa: nell'implementazione Java, il nodo corrente che in precedenza avevamo chiamato D non è un parametro della funzione bensì l'oggetto chiamante `this`; la rilevanza r_l, invece, rimane un parametro. Si noterà come il seguente codice ricalchi pedissequamente la definizione, con l'aggiunta di un controllo sui nodi senza nome o con nome “IGNORE”, che sono quelli potati.

```
public double weight(double relevance) {

        double result = 0;

        if( this.getName().equals("") ||
            this.getName().equals("IGNORE") ||
            this.getName().equals("*") ||
            this.getName().equals("?") )
                return result;

        else if (
             this.getName().equals("#PCDATA") ||
             this.getName().equals("ANY"))
                return relevance;

        else if (this.getName().equals("+")) {
             if(children != null)
                     return this.children[0].weight(relevance);
             else
                     return result;
        }

        else if (this.getName().equals("AND")) {
                double sum = 0;
                for (int c = 0; c < children.length; c++)
                        sum += children[c].weight(relevance);
```

```
                return sum;
        }
        else if (this.getName().equals("OR")) {
                double min = 0;
                if(! children[0].getName().equals("IGNORE"))
                        min = children[0].weight(relevance);
                for (int c = 1; c < children.length; c++) {
                        if(! children[c].getName().equals("IGNORE")) {
                                double cur =
                                 children[c].weight(relevance);
                                if (cur < min)
                                        min = cur;
                        }
                        else
                                continue;
                }
                return min;
        }
```

Ed ecco l'ultimo caso della funzione *weight*, quello in cui la chiamata ricorsiva passata a tutti i figli vede il parametro r_l diviso per $\gamma = 2$.

```
        else {
                double temp = 0;
                if(children != null)
                        for (int c = 0; c < children.length; c++) {
                                temp +=
                                 children[c].weight(relevance/GAMMA);
                        }
```

Si ricordi che dobbiamo restituire il parametro di rilevanza *non ancora dimezzato*, più il risultato di *weight* applicato ai figli con il parametro – viceversa – dimezzato.

```
                return temp + relevance;
        }
} /* end weight */

public boolean isTAG() {
        return  !(this.isOP() ||
```

```
                                    this.getName().equals("#PCDATA") ||
                                    this.getName().equals("ANY") ||
                                    this.getName().equals("EMPTY"));
    }
```

La seconda e ultima struttura dati *ad hoc* del progetto, molto più snella di `Node`, è `ElementDeclaration`. Non ne riportiamo il codice Java completo ma diciamo solo che un oggetto istanza di `ElementDeclaration` ha due campi, rispettivamente la stringa di nome proprio e un `Node` (il quale a sua volta contiene un *array* di nodi figli).

- `private String elemName;`
- `private Node contentModel;`

Il nome "`contentModel`" della variabile nodo non è casuale: essa corrisponde proprio al *content model* di una dichiarazione di elemento del file DTD, dunque ha senso che sia una struttura ad albero. Un esempio di *content model* è `(a*,(b|(c,d,e?)))`.

5.3 Il *package* `similarity.dtdparser`

Questo *package* contiene tutti i file generati automaticamente da JavaCC sulla base della grammatica `DTDParser.jj` .

Un primo insieme contiene classi generiche, uguali da *parser* a *parser*:

- `SimpleCharStream.java`
- `Token.java`
- `ParseException.java`
- `TokenMgrError.java`

I rimanenti tre file sono invece specifici per la nostra applicazione:

- `DTDParser.java`
- `DTDParserConstants.java`
- `DTDParserTokenManager.java`

L'analisi sintattica vera e propria è competenza di:

1. `DTDParserTokenManager.java`: questa classe analizza il flusso di caratteri in ingresso suddividendolo in porzioni (*token*) in base alla specifica contenuta in `DTDParser.jj` ; ogni porzione si vede un oggetto della classe `Token` associato, in particolare con informazioni quali `kind` (tipo di *token*) e `image` (porzione di testo corrispondente) a cui accediamo per associare azioni semantiche a ogni regola di produzione.

2. `DTDParser.java`: anche questa classe ha un ruolo di primo piano in quanto, in corrispondenza 1:1 con `DTDParser.jj` , contiene un metodo Java per ogni simbolo nonterminale. Nel nostro caso, la funzione associata all'assioma è

   ```
   final public Node start() throws ParseException,
    ElementDeclarationException, NodeException
   ```

5.4 Gestione delle eccezioni

Le eccezioni sono competenza dalle due classi del *package* `similarity.error`, di cui non è interessante riportare il codice. Banalmente, esse sfruttano l'ereditarietà caratteristica dei linguaggi orientati a oggetti come Java invocando con una chiamata `super` le eccezioni della classe padre, ovvero `Exception.java`.

Bibliografia

[BGM] Elisa Bertino, Giovanna Guerrini e Marco Mesiti, *A Matching Algorithm for Measuring the Structural Similarity between an XML Document and a DTD and its Applications*, Information Systems, 29(1): 23-46, 2004.

[ItA] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein, *Introduction to Algorithms, Second Edition*, The MIT Press, 2001.

[JavaCC] `http://javacc.dev.java.net`

[LMC] Giorgio Ausiello, Fabrizio d'Amore e Giorgio Gambosi, *Linguaggi, modelli, complessità*, FrancoAngeli, 2003.

[LSF] Luigia Carlucci Aiello e Fiora Pirri, *Dispense del corso di* Linguaggi e sistemi formali, Università di Roma "La Sapienza", anno accademico 2003-2004.

[PLT] Riccardo Rosati, Luigi Dragone e Marco Ruzzi, *Lucidi ed esercitazioni del corso di* Progetto di linguaggi e traduttori, Università di Roma "La Sapienza", anno accademico 2004-2005.

[W3C] VV. AA., *Extensible Markup Language (XML) 1.0 (Third Edition)*, W3C Recommendation 04 February 2004, `http://www.w3.org/TR/REC-xml/`

Indice analitico