Un po' di Informatica [1]

Luca Roversi

24 aprile 2012

Indice

Premessa

Questa premessa è un amalgama di pensieri, espressi da vari specialisti dell'informatica, con lo scopo di descrivere la natura dell'informatica come scienza che può valere la pena di studiare per raggiungere fini analoghi a quelli che si pongono la matematica o la fisica: avere un punto di vista in più per guardare il mondo.

L'effetto finale potrà assumere toni ideologici, ma, del resto, si vuole parlare di informatica, tentando di "propagandarne" una certa utilità.

Le tecnologie digitali: opportunità inedite ma anche criticità inedite [1]

Nuovi orizzonti ed opportunità

Le tecnologie dell'informazione e delle trasmissioni digitali, ovvero le tecnologie informatiche, hanno aperto *orizzonti* assolutamente inediti in quasi tutti gli ambiti dell'attività umana organizzata. In special modo, stanno creando eccezionali *opportunità* per sviluppare nuove tipologie di servizi e strumenti, sempre più interattivi e personalizzabili, e per migliorare in modo radicale quelli esistenti. Alcuni esempi sono:

- La possibilità di accumulare ed elaborare moli di dati, ordini di grandezza maggiori di quanto mai ipotizzato prima, e di organizzarli e mantenerli in banche dati accessibili tempestivamente.
- La possibilità di creare e offrire nuove esperienze virtuali, anche in modo remoto, costituiscono una combinazione di potenzialità straordinarie, per molti versi ancora inesplorate.

Insidie e loro sorgente

Non tutto però, procede sempre con la naturalezza ed efficacia auspicate. La rivoluzione digitale dettata dalla diffusione, riuscita o mancata, di tecnologie dell'informazione e comunicazione sembra invece essere accompagnata da insidie

[1] Liberamente tratto da [**?**]

e difficoltà molto più di qualunque altra rivoluzione tecnologica avvenuta in passato.

Per analizzare, seppur schematicamente, tali rischi e problematiche, è opportuno partire con l'indicare l'origine delle criticità, e una via ragionevole per superarle.

Abilità informatiche *vs.* Cultura informatica

Non è ancora chiara la differenza tra "abilità informatiche" e "cultura informatica".

Informatica come uso della tecnologia. In generale, si guarda all'informatica come un soggetto che "genera" tecnologie e strumenti per l'uso quotidiano. Tale visione può spingere le persone all'acquisto di strumenti tecnologici trasformandole in *consumatrici di informatica* piuttosto che in *utilizzatrici consapevoli di prodotti informatici*.

Questo è, essenzialmente, consumismo di tecnologia, probabilmente molto apprezzato dalle imprese informatiche, soprattutto quando colpisce vertici delle pubbliche amministrazioni, perché allora queste lo riverberano su tutta la comunità, provocando un'onda di acquisti, in generale, non strettamente necessari.

Visione strumentale e abilità informatiche. Tuttavia le implicazioni della visione tecnologica, osservate nel paragrafo precedente, sono in linea con lo stile di vita "occidentale" e questo non dovrebbe, certamente, creare alcuno scandalo.

Il vero problema.

> *Il problema nasce quando si pone la visione tecnologica come base della cosiddetta alfabetizzazione informatica, legando le abilità informatiche agli standard tecnologici correnti.*

Il motivo è che si illudono le persone di possedere conoscenze autentiche.

Al contrario, esse si troveranno in possesso di competenze che svaniscono al prossimo mutamento del paradigma *hardware* o *software*.

Una visione più scientifica dell'informatica

Obiettivo.

> *Puntare a smettere di vedere l'informatica come la familiarità di accesso e utilizzo al calcolatore per mezzo di programmi standard preconfigurati che ai più paiono, calcolatore e programmi, intoccabili.*

Tesi. Tale obiettivo è supportato dalla "dimostrabilità" della seguente tesi:

> *Vedendo l'informatica come scienza, non solo come tecnologia, e diffondendone l'aspetto culturale, si diventa migliori utilizzatori e produttori di strumenti digitali, proprio perché utenti più critici e consapevoli.*
>
> *Soprattutto, vedendo l'informatica come scienza, si può guadagnare un'ulteriore prospettiva da cui osservare e descrivere il mondo.*

La validità della tesi dovrebbe essere "dimostrata" nelle sezioni seguenti.

Apprendere ed usare un po' di cultura informatica

> *La cultura informatica, può diventare sia una prospettiva inedita sul mondo, sia una lingua franca per il dialogo interdisciplinare e intersettoriale.*

La cultura informatica come prospettiva inedita. La prospettiva inedita si basa sul proporre di vedere l'informatica come scienza che offre una lettura della realtà non riconducibile a quella di altre discipline.

Per realizzare un tale fine occorrerebbe:

- afferrarne i modelli di ragionamento,
- conoscerne le tecniche per affrontare i problemi,
- riflettere sulle modalità proprie dell'informatica per delinearne potenzialità e limiti.

In particolare, è l'individuare limiti la chiave per capire l'utilità di un qualsiasi strumento al fine di usarlo quando veramente serve, cercando, invece, alternative nel momento in cui lo strumento perde efficacia.

L'informatica, ovviamente, non sfugge a questa idea.

Domanda spontanea.

> *L'informatica è la scienza di cosa?*

L'informatica come scienza delle metodologie per risolvere problemi per mezzo dei calcolatori. In primo luogo, l'informatica è la scienza delle metodologie generali per "automatizzare" la soluzione a problemi.

Divide et impera. Saper decomporre un problema in sottoproblemi fino a quando questi si possono risolvere con tecniche specifiche *ad hoc* proprie della disciplina in questione e poi ricomporre via via le soluzioni intermedie fino a raggiungere la soluzione completa è il tipico principio informatico *divide et impera*. In particolare:

> *Divide et impera* è un principio di *semplificazione cognitiva*, ma spesso conduce anche a una maggiore efficienza nelle procedure.

Esempi di *divide et impera.* Essi sono *ricercare* una parola in un vocabolario, o definire una procedura per realizzare un qualsiasi compito pratico, quali: cucinare una pietanza, stirare una camicia, mettere in ordine un mazzo di carte, pesare un oggetto su una bilancia, pagare in contanti per un acquisto.

Peculiarità dell'informatica. Si può certamente osservare che l'analisi di un problema per sottocomponenti, col fine di ricomporre le soluzioni parziali, in accordo con una descrizione astratta, ovvero per mezzo di un modello, è sicuramente tipico di molte discipline.

La peculiarità dell'informatica, nei confronti di questa metodologia, è che i suoi strumenti, generalmente di natura linguistica, permettono l'analisi di un contesto e la sua descrizione per mezzo di modelli a granularità diverse, non monoliticamente, come per tante discipline, anche esatte.

La descrizione "informatica" può avere una natura "telescopica" che permette di mettere a fuoco e circoscrivere i problemi specifici di uno specifico livello di astrazione nel descrivere la soluzione di un problema.

Complessità ed interdisciplinarità dell'informatica. Saper analizzare un contesto conduce ad una comprensione sistemica, sempre più necessaria di ambiti ormai dominati dalla complessità.

È stata proprio l'informatica, forse, la scoperta più inter e multidisciplinare di questo ultimo mezzo secolo, che è poi la ragione della sua diffusione, del carattere pervasivo delle tecnologie informatiche in tante discipline e attività umane.

"Natura informatica" diffusa. Si è scoperto che ogni ambito di attività o ramo del sapere possiede una componente *software*, un "residuo" informatico, che è possibile mettere a fuoco attraverso una prospettiva metodologica computazionale alla disciplina, sia in termini algoritmico procedurali che in termini linguistico-logico-semantici.

Visione informatica e visione estesa. Il mettere a fuoco l'*aspetto informatico* in una qualsiasi disciplina o attività, permette di analizzare la disciplina stessa sotto una nuova luce che la rende più strutturale e strutturata nel senso che:

> *Quel che conta della disciplina non sono più i concetti in sé che la compongono, ma le differenze o i rapporti tra di essi, il modo con cui le correlazioi si creano, ovvero le procedure.*
>
> *Ad esempio, il "contenuto informatico degli scacchi" è quel che rimane quando non ci sono più la scacchiera e le pedine. Quel che rimane è la rappresentazione delle regole di trasformazione tra configurazioni con cui descrivere ogni passo di una qualsiasi partita.*

È possibile far emergere l'aspetto informatico, ovvero gli aspetti procedurale e linguistico-semantico, all'interno di numerose attività e discipline: dalla metrica

e struttura delle filastrocche, al rito della danza, dalle manovre ferroviarie, al gioco. L'informatica si è rivelata così, prima ancora di essere calata nelle sue applicazioni, una lingua franca per il dialogo interdisciplinare, ovvero un antidoto all'incomunicabiità degli specialismi.

Informatica *vs.* altre scienze: andata e ritorno. Attraverso la vera alfabetizzazione informatica, specialisti di altre discipline possono beneficiare del corredo concettuale che le è proprio.

Infatti, se davvero esposti alla cultura informatica, da un lato è possibile accrescere la propria produttività individuale attraverso l'uso sapiente e più consapevole di ausili tecnologici.

Dall'altro, sarà anche possibile intrecciare scambi concettuali con l'informatica sul piano dei contenuti perché nel momento in cui si diventa utilizzatori consapevoli di strumenti reali e concettuali nuovi la distanza tra due discipline, o tra attività con radici e metodologie diverse, tende ad annullarsi.

Informatica: Elogio di Babele[2]

Dopo aver propagandato la possibile utilità di diventare utilizzatori più coscienti dell'armamentario informatico, approfondiamo una delle possibili prospettive, con cui guardare a tale armamentario.

L'unità dei linguaggi

Le origini. Abbiamo già provato a descrivere l'informatica molto succintamente come:

> *Scienza delle metodologie generali per "automatizzare" la soluzione a problemi.*

È utile ricodare che la nascita dell'informatica precede di almeno dieci anni la disponibilità di quei manufatti che chiamiamo calcolatori (*computer*) e l'utilizzo dei quali oggi tendiamo a confondere con quella scienza[3].

L'"automatizzabile", o meglio il "calcolabile". Lo studio delle attività che siano atomatizzabili, o, più tecnicamente, *calcolabili* matura nei primi anni trenta del novecento, nel cuore della logica matematica, per opera di K. Gödel, A. Church, S. Kleene, E. Post e, soprattutto, Alan M. Turing.

Il "calcolo" è manipolazione simbolica. Mediante un'analisi dei processi di calcolo umani (in particolare delle limitazioni della memoria e della percezione), Turing identifica il "calcolo" con la possibilità di manipolazione combinatoria di insiemi finiti e discreti di simboli:

[2] Liberamente tratto da [**?**]

[3] "*Computer science is no more about computers than astronomy is about telescopes*", E. W. Dijkstra, premio Turing 1972. Il premio Turing, il più importante riconoscimento internazionale per un informatico, è attribuito ogni anno dall'associazione professionale degli informatici americani (ACM).

> *Calcolare consiste nel copiare e rimpiazzare simboli, tratti da un alfabeto finito, secondo regole, anch'esse finite, e fissate in anticipo.*

Il "calcolo" è espletabile da automi. La descrizione dell'attività di calcolo, ovvero il modello, è così semplice da poter essere descritto come l'attività di un automa: la *macchina di Turing*.

Essa, in accordo con regole memorizzate in una lista finita, sposta o (ri)scrive un simbolo alla volta su un nastro.

Il "calcolo" è generale. L'aspetto sorprendente è che la macchina di Turing, dal sapore estremamente rudimentale, basato sull'analisi dei processi di calcolo umani[4], è un modello così generale, da includere una macchina che è in grado di emulare tutte le altre.

Cosa significa "modello generale"? Consideriamo l'esecuzione di una macchina su un suo dato di ingresso. La macchina funziona in modo deterministico, applicando le regole ai dati. Se abbiamo una descrizione della macchina (ovvero un elenco delle regole in base alle quali opera) possiamo seguire la sua esecuzione, passo passo, con carta e matita.

Questo procedimento, che si riassume in:

1. fissiamo una descrizione della macchina,
2. fissiamo una descrizione di un dato di ingresso,
3. "seguiamo" l'esecuzione passo passo della macchina,

è il procedimento di calcolo. Turing mostra che anche questo calcolo[5] è esprimibile, a sua volta, come una macchina di Turing.

La macchina di Turing universale. Riassumendo:

> *Tra tutte le macchine di Turing c'è quella* ***universale****.*
>
> *La macchina di Turing universale è in grado di simulare tutte le altre.*

Non importa "costruire" una specifica macchina per ogni calcolo di nostro interesse. Basta costruire la macchina universale e fornirle la descrizione, ovvero il programma, del calcolo che vogliamo eseguire. La macchina universale eseguirà per noi il programma sui dati di nostra scelta.

Dalla descrizine astratta alla realizzazione concreta. Sarà John Von Neumann a rendersi conto presto delle potenzialità tecnologiche insite in quest'analisi, progettando un manufatto che avrebbe realizzato, con valvole e fili, una macchina di Turing universale, cioè un computer, non più *human*, ma *hardware which calculates*.

[4] Turing's "machines": These machines are humans who calculate, L. Wittgenstein, Remarks on the Philosophy of Psychology, Vol. 1, Blackwell, Oxford, 1980.

[5] Quello, cioè, che, data una descrizione della macchina e un particolare dato, esegue, o simula, la macchina su quel dato.

I calcolatori che stanno sulle nostre scrivanie sono, ancor oggi, variazioni tecnologicamente avanzate del progetto ingegneristico di John Von Neumann, basato sul progetto puramente speculativo, di natura astratta, di Alan Turing.

Lettura linguistica dell'Informatica

Saremo preponderantemente interessati alla lettura linguistica dell'informatica. Come conseguenza, vedremo alcuni aspetti tecnologici avanzati ad essa collegati.

Tanti linguaggi, un solo concetto di "calcolabile". Non vi è calcolo senza linguaggio, ma la varietà dei linguaggi *non produce concetti diversi di calcolo*, perché tutti sono codificabili nella semplice macchina di Turing universale. Un solo linguaggio, che può essere rappresentato da sequenze finite di una sola coppia di simboli, esaurisce, mediante codifica, tutto quanto è esprimibile per la manipolazione automatica.

In particolare, ci occuperemo di provare a capire di cosa si occupi l'informatica, e di come la si possa usare, proprio concetrandoci su uno specifico linguaggio, l'XML, che ci permetterà di descriverne infiniti altri, ma con caratteristiche ben precise, e "verificabili" automaticamente da un calcolatore.

Tutti i linguaggi del "calcolabile" sono equivalenti. L'XML è uno dei linguaggi della babele cui si accenna nel titolo.

La babele nasce sin dall'inizio dell'informatica.

Esistono più speculatori del "calcolabile". Intendiamo che non c'è stato solo Turing, a speculare sui e a definire i processi di calcolo effettivo per mezzo di rappresentazioni linguistiche, indipendenti da qualsiasi realizzazione tecnologica del processo di calcolo descritto.

Negli stessi anni, altri descrivono in cosa consista "calcolare", partendo da presupposti ed analisi diverse:

Alonzo Church. Introduce il λ-calcolo, un formalismo basato sulla riscrittura successiva di sequenze di caratteri che corrispondono ad espressioni simboliche.

Kurt Gödel. Sviluppa un modo di definire funzioni per "ricorrenza", o ricorsione.

Emil Post. Definisce un sistema di celle contigue che si influenzano tra loro e che evolvono per passi discreti.

FORTRAN, LISP, Java, . . . , XML, DTD, XSLT, . . . In seguito, quando la disponibilità di calcolatori elettronici richiederà di poterli programmare in modo sempre più semplice ed efficace, saranno introdotte altre centinaia di linguaggi di programmazione per descrivere il calcolo.

Tuttavia, questa moltiplicazione di linguaggi non intacca la generalità del concetto di "calcolabile". Tutti questi linguaggi, infatti, sono solo materializzazioni diverse dello stesso concetto, sviluppatesi come conseguenza di sensibilità descrittive e obiettivi diversi.

Ad esempio, se una certa corrispondenza tra numeri naturali è calcolabile da una macchina di Turing, allora esiste una frase nel λ-calcolo di Church, o nelle funzioni ricorsive di Gödel, o di un qualsiasi altro linguaggio di programmazione, che costruisce la stessa corrispondenza. Il punto è che il modo in cui si descrive come costruire la corrispondenza in quesione cambia, ma il concetto di "cosa è calcolabile" è invariante rispetto alla descrizione.

L'informatica come strumento linguistico per descrivere il mondo

Una delle peculiarità dell'informatica è quindi sollecitare lo sviluppo di linguaggi per descrivere fenomeni di particolare interesse.

Tali fenomeni sono le procedure che ci conducono a risolvere dei problemi che possiamo ritenere interessanti.

Gli interpreti naturali di tali descrizioni sono i calcolatori.

È come dire che:

> *L'informatica è la scienza una delle cui caratteristiche peculiari è descrivere il mondo ai calcolatori.*

In accordo con tale definizione, ci occuperemo proprio di descrivere il mondo ai calcolatori, sotto forma di documenti opportunamente strutturati.

Un calcolatore potrà operare su tali documenti, interpretandoli con diverse finalità.

Vedremo come il produrre tali descrizioni ci forzerà a riflettere su vari aspetti quali, ad esempio:

- quanto non sia necessariamente banale scegliere il giusto livello descrittivo nel parlare di un qualche soggetto d'interesse,
- come sia possibile caratterizzare l'insieme dei soggetti di nostro interesse,
- cosa possa significare dare un'interpretazione a concetti, attraverso un calcolatore.

A tal fine, come già annunciato, sfrutteremo la tecnologia linguistica fornitaci dal linguaggio per descrivere linguaggi XML.

Capitolo 1

Introduzione

Molti metodi possono esistere per scoprire il perché è sostenibile la posizione secondo la quale l'informatica è la scienza con la quale si descrive il mondo ai calcolatori.

Il nostro metodo mescolerà teoria e pratica, la quale, evidenzia quanto l'essenza della teoria è stata assimilata.

Un sito per *cv*. Visitiamo http://europass.cedefop.europa.eu/it/home. Tra le varie finalità, il sito è progettato per raccogliere e diffondere *curriculum vitae* di persone che vogliano partecipare ad iniziative di studio o lavoro in Europa.

In quanto persone che desiderino "giocare all'informatico" cominciamo con l'osservare come i *cv* vengono raccolti.

I *cv* sono raccolti per mezzo di un'interfaccia web. Ovvero, all'interessato *non è* permesso inviare un *cv* scritto con un programma di elaborazione di sua scelta.

È dato per scontato che, se la scelta fosse lasciata all'interessato all'invio, nella stragrande maggioranza dei casi gli organizzatori del sito riceverebbero un *file* scritto con uno degli elaboratori di testo più diffusi — non è difficile immaginare quale —, in accordo con una scelta tipografica essenzialmente arbitraria.

Le prima domanda (fondamentale).

> Perché si impedisce l'invio di *cv* composti con elaboratori testi ampiamente diffusi e normalmente utilizzati dalle persone?

Possibile risposta alla prima domanda Senza troppi preamboli:

> Siccome gli elaboratori di testo diffusi memorizzano i *file* usando *formati proprietari* né gli organizzatori del servizio di raccolta, né coloro che inviano il *cv* si possono ritenere pienamente proprietari delle informazioni rilasciate.

Per "formato proprietario" intendiamo che i dati sono memorizzati in accordo con convenzioni che non sono generalmente note, a meno di *reverse engineering*.

Esempio 1 (Cosa può nascondere un formato proprietario.) Per renderci conto di cosa possa significare "formato proprietario" facciamo un esperimento.

Prima passo Supponiamo che sul nostro calcolatore sia utilizzabile il più noto, ad oggi, anno 2011, elaboratore di testi. Supponiamo di scrivere su di esso un abbozzo di *cv* che memorizziamo come *file* di nome `cv`. Anche se questo può non essere evidente da quanto appare sullo schermo, il vero nome del *file* che abbiamo memorizzato è `cv.doc`. Il suffisso `.doc` è l'estensione del *file* ed indica quale sia il programma che lo ha generato e che potrà, in generale, accedervi per eventuali modifiche.

Per ipotesi, supponiamo di aver memorizzato `cv.doc` in cui i nostri dati anagrafici includevano il nostro nome Luca.

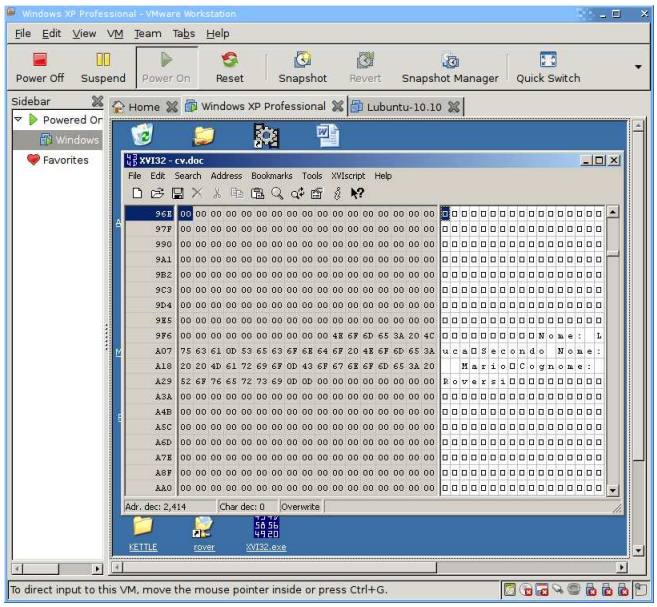


Figura 1.1: Il *file* `cv.doc` letto con l'elaboratore di *file* esadecimale `XVI32.exe`

Secondo passo Scarichiamo da www.chmaas.handshake.de l'archivio `hex-editor-xvi32.zip`. L'archivio scompattato, contiene (anche) il programma `XVI32.exe`. Esso è un elaboratore testi esadecimale. Con buona approssimazione, questo significa che `XVI32.exe` legge il contenuto dei *file* con la massima granularità possibile.

Terzo passo Usiamo `XVI32.exe` per leggere il contenuto di `cv.doc`. Una volta caricato `cv.doc` il programma `XVI32.exe` ci rivela l'esistenza di un enorme quantità di caratteri non intellegibili tra i quali esistono anche quelli effettivamente digitati sulla tastiera. Se, infatti, cerchiamo il testo "Luca" `XVI32.exe` scova tale sequenza di caratteri, mostrando la finestra in Fig. 1.1.

Il motivo per cui memorizzare un *file* `doc` "vuoto", nel quale non abbiamo introdotto alcun testo, occupa una quantità di spazio molto superiore a quel che ci potremmo aspettare è stato confermato tramite `XVI32.exe`. Esso ha rivelato l'effettiva presenza di informazione sotto forma di sequenze non intelleggibili di caratteri. Tali sequenze sono sicuramente legate almeno alla necessità di memorizzare le informazioni per la corretta impaginazione del testo. In realtà non è dato sapere quel che essa codifica a tal punto che la configurazione dell'elaboratore testi che produce *file* `.doc` può continuare a memorizzare parti di testo cancellate, semplicemente oscurando la loro presenza in fase di stampa, su carta, o sullo schermo, senza eliminarle del tutto. `XVI32.exe` potrebbe rivelare la pre-

senza di tale testo oscurato, proprio perché il suo funzionamento è indipendente da come le informazioni sono memorizzate nei *file* `.doc`.

Seconda domanda (fondamentale). In base a quanto osservato sinora:

> È necessario inviare *cv* già impaginati?

Risposta certa alla seconda domanda. Essa è: "No, non è affatto necessario." e i motivi sono essenzialmente due.

Da un lato l'impaginazione è superflua se immaginiamo che in una banca dati di *cv* cercheremo quelli di persone che soddisfino specifici requisiti. Oppure, lo useremo per condurre indagini di natura qualitativa e quantitativa sulle caratteristiche di coloro i quali inviano il *cv*. Queste sono operazioni che non vorremo condurre a mano, ma per le quali vorremo sfruttare calcolatori elettronici opportunamente programmati. I calcolatori sono totalmente indifferenti all'impaginazione. Anzi, la loro programmazione, finalizzata alle operazioni menzionate, è molto meno prona agli errori, se l'informazione da cercare è espressa in modo essenziale, ovvero senza ridondanze relative al modo in cui i *cv* debbano essere stampati nel caso un addetto alle risorse umane voglia leggerlo.

Separare *contenuto* e *forma*. Semmai, è vero il contrario. Solo una volta individuati nel modo più efficiente, ovvero per via automatizzata, i *cv* che soddisfano i criteri di ricerca, è opportuno stamparli. Tuttavia l'ordine di presentazione delle notizie di un *cv* può cambiare a seconda delle finalità per le quali si cercano persone con *cv* di un certo tipo. Quindi anche lo stile di presentazione deve poter essere il più flessibile possibile in accordo con l'idea seguente che separa il *contenuto* dalla *forma* con cui il contenuto deve essere eventualmente presentato:

- Esiste una sola rappresentazione dei *cv* adatta alla manipolazione per mezzo di calcolatori elettronici. La manipolazione è generelamente mirata ad individuare insiemi di *cv* interessanti per un certo scopo.
- Esiste un insieme, in linea di principio illimitato, di possibili strutturazioni tipografiche dei *cv* individuati al passo precedente.

Cosa fa `europass.cedefop.europa.eu`? Il sito in questione è un esempio di realizzazione della filosifia appena illustrata.

Per mezzo di una sequenza di *form*, pagine web in cui poter inserire dati, raccoglie quelli relativi ad un *cv*.

Una volta raccolti i dati, alcune scelte sono disponibili. Una è memorizzare il *cv* sul proprio calcolatore in formato XML, di cui riportiamo qui sotto un estratto e del quale — il linguaggio XML — diverremo esperti:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet
     href="http://europass.cedefop.europa.eu/xml/cv_en_GB.xsl"
```

```
     type="text/xsl"?>
<europass:learnerinfo
    locale="en_GB"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:europass="http://europass.cedefop.europa.eu/Europass/V2.0"
    xsi:schemaLocation
      ="http://europass.cedefop.europa.eu/Europass/V2.0
        http://europass.cedefop.europa.eu/xml/EuropassSchema_V2.0.xsd">
<docinfo>
<issuedate>2011-02-09T17:24:15+01:00</issuedate>
<xsdversion>V2.0</xsdversion>
<comment>Automatically generated Europass CV</comment>
</docinfo>
<prefs>
<field before="step1.lastName" name="step1.firstName"></field>
<field keep="true" name="step1.addressInfo"></field>
<field keep="false" name="step1.telephone"></field>
<field keep="false" name="step1.mobile"></field>
<field keep="false" name="step1.fax"></field>
<field keep="false" name="step1.email"></field>
<field keep="false" name="step1.nationality"></field>
<field format="" keep="false" name="step1.birthDate"></field>
<field keep="true" name="step1.gender"></field>
<field keep="false" name="step1.photo"></field>
<field keep="true" name="step1.application.label"></field>
<field format="/text/short" name="step3List[0].period"></field>
<field before="step4List" keep="false" name="step3List"></field>
<field keep="false" name="step4List"></field>
<field keep="false" name="step5.motherLanguages"></field>
<field keep="false" name="step5.foreignLanguageList"></field>
<field keep="false" name="step6.socialSkills"></field>
<field keep="false" name="step6.organisationalSkills"></field>
<field keep="false" name="step6.technicalSkills"></field>
<field keep="false" name="step6.computerSkills"></field>
<field keep="false" name="step6.artisticSkills"></field>
<field keep="false" name="step6.otherSkills"></field>
<field keep="false" name="step6.drivingLicences"></field>
<field keep="false" name="step7.additionalInfo"></field>
<field keep="false" name="step7.annexes"></field>
<field keep="false" name="grid"></field>
</prefs>
<identification>
<firstname>Luca</firstname>
<lastname>Roversi</lastname>
<contactinfo>
<address>
```

```
<addressLine>Bla</addressLine>
<municipality>Bla</municipality>
<postalCode>Bla</postalCode>
<country>
<label>Bla</label>
</country>
</address>
<telephone></telephone>
<fax></fax>
<mobile></mobile>
<email></email>
</contactinfo>
<demographics>
<gender>M</gender>
<nationality>
    <label></label>
</nationality>
</demographics>
        </identification>
<application>
<label>Astronauta</label>
        </application>
<languagelist>
<language xsi:type="europass:mother">
<label></label>
</language>
        </languagelist>
<skilllist>
<skill type="social"></skill>
<skill type="organisational"></skill>
<skill type="technical"></skill>
<skill type="computer"></skill>
<skill type="artistic"></skill>
<skill type="other"></skill>
<structured-skill
    xsi:type="europass:driving"></structured-skill>
  </skilllist>
  <misclist>
<misc type="additional"></misc>
<misc type="annexes"></misc>
  </misclist>
</europass:learnerinfo>
```

La struttura sintattica del *file* XML qui sopra riportato non è molto intelleggibile da un umano, per esserlo, invece, agli "occhi" di un calcolatore opportunamente programmato. Noi arriveremo a capire come programmare un calcolatore

affinché abbia occhi per, in linea di principio, un qualsiasi *file* XML.

La struttura di uno qualsiasi dei *cv* raccoglibili dal sito in questione è quindi documentata in maniera liberamente intelleggibile, almeno da un qualsiasi calcolatore.

Dall'intelleggibilità segue la possibilità di scrivere programmi che permettono la manipolazione automatizzata di insiemi di *cv* con le finalità più disparate, tra le quali, ad esmpio, la buona presentazione tipografica nel così detto formato europeo dei *cv*.

Relativamente alla presentazione tipografica, il sito stesso permette la traduzione — letteralmente — di ogni *cv* dal formato XML ad almeno uno tra i formati XHTML e PDF, tenendo conto della lingua in cui occorre compilare il *cv*, se specificato.

Ad esempio, `cv_it_IT.xsl` è il programma per la trasformazione del *cv* da XML a XHTML, usando la lingua italiana, mentre `cv_en_GB.xsl` è quello per la generazione in lingua inglese.

Una parte cospicua dell'insieme di risorse è disponibile *on-line* alla pagina: *Risore tecniche europass*.

Memorizzando alcuni di tali documenti sul proprio computer ed apportando alcune semplici modifiche, possiamo avere uno strumento completo per la gestione di insiemi di *cv*, con le seguenti caratteristiche:

- sono in formato non proprietario (XML),
- la strutturazione del *cv* è pubblica (DTD equivalente),
- possiamo produrre (XSLT) una versione XHTML aggiornata da publicare immediatamente *on-line*, ad ogni aggiornamento del *cv*,
- possiamo usare gli srumenti di conversione disponibili sul sito per produrre le versioni nei formati `.doc`, `.odt`, `.xhtml` e `.pdf`, partendo dal medesimo documento sorgente XML col nostro *cv*.

1.0.1 Potenzialità

Il meccanismo di raccolta ed utilizzo di *cv* descritto sinora è trasferibile a qualsiasi contesto in cui sia necessario gestire in maniera economica e flessibile grosse quantità di dati, da presentare in formato essenzialmente testuale. Un'amministrazione pubblica è certamente un esempio canonico.

Inoltre, la completa trasparenza della struttura e dei contenuti dei *cv* gestiti da `europass.cedefop.europa.eu` permette il trasferimento di tali dati ad altri contesti semplicemente per via telematica, permettendo la loro ristrutturazione, sempre automatizzata, a piacimento.

Nel nuovo contesto, grazie alla conoscenza della struttura dei *cv* è possibile trasformare ciscuno di essi in nuovi documenti XML con gli stessi dati, magari integrati con nuovi, ma con diversa struttura.

Ad esempio, per un qualche motivo, potrebbe essere utile trasformare ogni *cv* ricevuto da `europass.cedefop.europa.eu` in uno con la seguente "forma":

```
<cv>
 <hobbies>
  <!-- elenco degli hobbies -->
 </hobbies>
 <dati-anagrafici>
  <nome>D</nome>
  <cognome>R</cognome>
  <data-nascita mese="01"
                giorno="18"
                anno="2000"/>
 </dati-anagrafici>
 <impieghi-precedenti>
  <impiego dal="2002" al="ora">
    Bla, Bla, Bla, Bla, ...
  </impiego>
  <!-- elenco degli impieghi precedenti -->
 </impieghi-precedenti>
</cv>
```

1.0.2 Conclusioni

Da quanto detto sinora si evince l'importanza di impostare correttamente la gestione di dati in formato elettronico, passando necessariamente per una loro efficace ed economica rappresentazione. In molti casi, questo può essere fatto sfruttando gli strumenti offerti dalla tecnologia XML.

Per questo motivo, descriveremo i "retroscena" di semplici, ma significativi, processi di progettazione — tipici del lavoro da informatico —, finalizzati alla gestione automatizzata di dati. L'attenzione sarà posta su cosa significhi progettare e manipolare per mezzo di un calcolatore documenti XML.

Questo ci metterà di fronte a problemi di rappresentazione di concetti, più o meno complessi, per mezzo di documenti XML, così come sarà necessario intuire cosa significhi programmare un calcolatore per produrre un testo tipograficamente accettabile, a partire da uno XML, zeppo di etichette più, o meno, intelleggibili.

Questa sperimentazione illustrerà una delle tipiche attività dell'informatico che, come ogni matematico, deve "imbrigliare" in linguaggi poco espressivi i fenomeni del mondo.

Sarà così possibile intuire cosa facciano gli informatici, i quali, per automatizzare le attività più disparate, devono trovarne rappresentazioni efficaci per mezzo di linguaggi così semplici da poter essere interpretati da un calcolatore elettronico.

La rappresentazione di concetti tramite testi XML sarà la chiave sperimentale per toccare con mano cosa significhi tale "esercizio" di rappresentazione.

1.1 Obiettivi

- **Impareremo a formalizzare concetti.** Ovvero occorrerà saper descrivere, senza ambiguità, al giusto livello di astrazione, un qualsiasi soggetto di interesse, attraverso **linguaggi** opportuni.
- **Impareremo le regole grammaticali di linguaggi manipolabili da un calcolatore elettronico.** Essi saranno caratterizzati da una estrema prolissità e rigidità strutturale, entrambe necessarie per essere **interpretate** da un calcolatore.
- **Impareremo a programmare un calcolatore.** Significherà scrivere con sintassi opportune le "sequenze" di operazioni che **interpretate** da un calcolatore, permetteranno di **trasformare frasi di un linguaggio in frasi di un altro linguaggio**.

Riassumendo ...

Impareremo a costruire delle ***rappresentazioni di porzioni del mondo****, attraverso concetti, alcune metodologie e strumenti adatti allo scopo, che possano essere "gestite" da un calcolatore.*

Ovvero, come un matematico, si imparerà a descrivere il mondo con strumenti formali, ma esisterà un vincolo in più: la descrizione deve avvenire attraverso strumenti linguistici interpretabili da interpreti precisi, ma senza intuizione.

Capitolo 2

Base della tipografia web: XHTML

- **Impareremo la sintassi base del linguaggio XHTML.** XHTML è il nome di un linguaggio i cui documenti rispettano sia la sintassi **HTML**, acronimo per *hyper-text meta-language*, sia quella XML.

  Esattamente come l'**HTML**, l'XHTML serve per programmare documenti interpretabili dai *browser* per la navigazione in Internet.

NOTE

- Il capitolo è di livello introduttivo e mira a fornire il necessario per comprendere quanto illustrato nei capitoli successivi.

  Per approfondimenti, esistono innumerevoli pubblicazioni, sia cartacee, sia *on-line*, facilmente individuabili per mezzo dei motori di ricerca. Ad esempio, un sito *on-line* di riferimento, in italiano, sui linguaggi **HTML** e XHTML è http://www.html.it/.

- Il capitolo contiene diversi esempi di testi XHTML. Lo scopo è sollecitare lo spirito di sperimentazione del lettore, nel senso seguente.

  È possibile copiare le linee di codice XHTML presentate in queste note in un qualche *file*, usando un qualche elaboratore di testi.

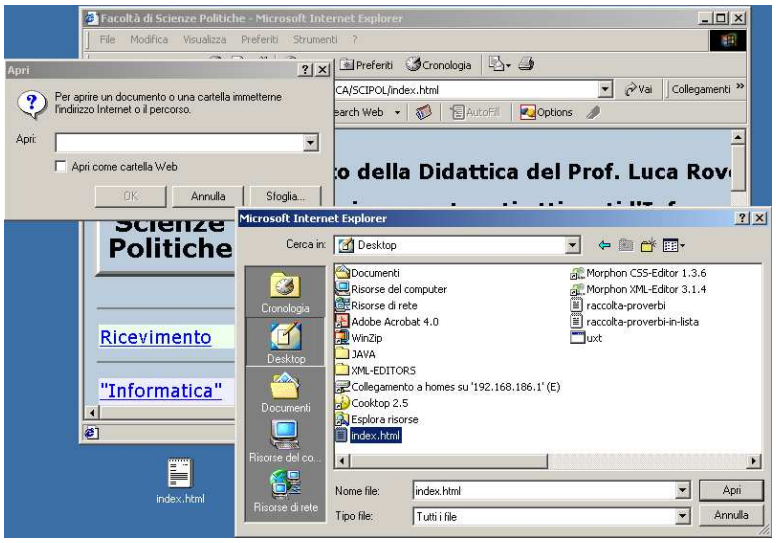
  È buona norma che i *file* così prodotti soddisfino le seguenti proprietà:

  - il loro nome deve avere estensione `.html`, ovvero deve terminare con la sequenza di caratteri `.html`;
  - è opportuno che le prime due righe di ogni documento siano:

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

  Il lettore è invitato ad interpretare ogni documento con estensione `.html` per mezzo di un *browser* per la navigazione internet.

  Ad esempio, supponendo di aver memorizzato il documento `index.html` sul proprio *desktop*, per interpretarlo con il *browser* Internet explorer, occorre selezionare la voce `Apri...` del *menù* `File`, e, quindi, "aprire" il documento `index.html`:



Questa attività permetterà di verificare, sperimentalmente, quali siano le regole di interpretazione tipografica dei documenti, il cui nome termina con l'estensione `.html`, realizzate dai *browser* per la navigazione internet.

2.1 Documenti XHTML

2.1.1 Struttura della pagina

Un documento XHTML è un documento XML con radice `html` che ha due discendenti diretti `head` e `body`, ciascuno col seguente significato:

- `head` (testa): contiene informazioni che riguardano il modo in cui il documento deve essere letto ed interpretato.

  Nella *head* si possono inserire:

  – *tag* sfruttati dai motori di ricerca per classificare la pagina,
  – linee di codice che programmano azioni da far eseguire ai *browser* che interpretano la pagina,
  – fogli di stile.

  Due *tag* inseribili come figli di `head` che vale la pena di menzionare sono `meta` e `title`:

  ```
  <meta http-equiv="Content-Type"
        content="text/html; charset=iso-8859-1"/>
  ```

  indica al *browser* per mezzo dell'attributo `charset` che deve usare l'insieme di caratteri occidentale e non, ad esempio, quello giapponese, per interpretare la pagina.

  ```
  <title>Titolo della pagina</title>
  ```

  determina il titolo della pagina che compare sulla barra in alto del *browser*. È buona norma compilare il contenuto di questo *tag* per evitare di avere pagine senza titolo.

- `body` (corpo): racchiude il contenuto vero e proprio del documento.

Esempio 2 (Scheletro di pagina XHTML.) Da quanto detto, una pagina XHTML ha la seguente struttura:

```
<html>
 <head>
   <meta http-equiv="Content-Type"
         content="text/html; charset=iso-8859-1"/>
   <title>Titolo</title>
 </head>
 <body>
   <!-- Contenuto della pagina  -->
 </body>
</html>
```

Esercizio 1 (Interpretare una prima pagina XHTML.) Modificare a piacere il contenuto del *tag* body del testo XHTML dell'Esempio 2 ed interpretarlo, ovvero visualizzarlo, con un *browser*.

2.1.2 Impostare il colore di sfondo

È possibile scegliere il colore di sfondo con cui verrà interpretata una pagina XHTML da parte di un *browser*.

Il colore di sfondo si determina specificando un valore per l'attributo `bgcolor` del *tag* `body`. `bgcolor` sta per *background color*, ovvero colore di sfondo. La seguente tabella contiene un elenco di possibili valori per `bgcolor`:

Colore	Parola chiave	Valore esadecimale
arancione	orange	#FFA500
blu	blue	#0000FF
bianco	white	#FFFFFF
giallo	yellow	#FFFF00
grigio	gray	#808080
marrone	brown	#A52A2A
nero	black	#000000
rosso	red	#FF0000
verde	green	#008000
viola	violet	#EE82EE

Esercizio 2 ("Ciao" in blu.) Utilizzando un qualche *browser*, verificare che:

```
<html>
 <head>
  <title>Ciao in blu</title>
 </head>
 <body bgcolor="blue">
 CIAO!
 </body>
</html>
```

produce una pagina con sfondo blu, con la scritta "CIAO!" e con titolo "Ciao in blu".

Molti dei colori elencati sono disponibili anche nelle varianti `dark` e `light`.

Esercizio 3 Descrivere i cambiamenti, prodotti dai seguenti testi XHTML, in un *browser* che li interpreti, rispetto al testo dell'Esercizio 2:

```
<!-- Primo testo XHTML -->
<html>
 <head>
  <title>Ciao in blu scuro</title>
```

```
 </head>
 <body bgcolor="darkblue">
 CIAO!
 </body>
</html>

<!-- Secondo testo XHTML -->
<html>
 <head>
  <title>Ciao in blu chiaro</title>
 </head>
 <body bgcolor="lightblue">
 CIAO!
 </body>
</html>

<!-- Terzo testo XHTML con
     valore dell'attributo
     bgcolor in notazione
     esadecimale            -->
<html>
 <head>
  <title>Ciao in blu</title>
 </head>
 <body bgcolor="#0000FF">
 CIAO!
 </body>
</html>
```

Commento 1
Vale la pena sottolineare che il colore visualizzato da un *browser*, oltre che dal valore, esadecimale, o meno, specificato per `bgcolor`, dipende dalla risoluzione della scheda video, relativamente al colore. In genere, la risoluzione del colore può variare da un minimo di 256 colori ad un massimo di svariati milioni di colori.

Esercizio 4 (Colori diversi con risoluzioni diverse.) Supponiamo di usare una qualche versione di Windows per visualizzare il seguente testo XHTML:

```
<html>
 <head>
  <title>Ciao in un qualche colore</title>
 </head>
 <body bgcolor="#EC99CE">
 CIAO!
 </body>
</html>
```

Cambiare la risoluzione dei colori visualizzati sul monitor, seguendo il "cammino": `Pannello di controllo > Schermo > Impostazioni`, e rivisualizzare la pagina; la tonalità di colore è sensibilmente diversa passando da 256 a 65.536 colori.

Commento 2
È importante assegnare sempre un colore di sfondo ad una pagina, sia esso il bianco, corrispondente a `bgcolor="#FFFFFF"`. Altrimenti, si può incappare nel comportamento predefinito del *browser* che, per definizione, assegna alla pagina, come colore di sfondo, quello che l'utente ha impostato nella finestra del sistema operativo: se questo è nero e la pagina non contiene alcuna definizione di colore di sfondo, la pagina sarà anch'essa nera.

2.1.3 Inserire un'immagine di sfondo

È possibile inserire un'immagine sullo sfondo di una pagina XHTML. A tal fine si specifica il nome del file che contiene l'immagine, ad esempio `immagine.gif`, come valore per l'attributo `background` del *tag* `body`.

Per definizione, il contenuto di `immagine.gif` viene ripetuto orizzontalmente e verticalmente fino a riempire l'intero sfondo della pagina XHTML.

Esercizio 5 ("Ciao" su immagine.) Si procuri una semplice immagine, ovvero un file che abbia estensione `.gif` (*graphical interchange format*) o `.png` (*portable network graphics*). Ad esempio, la si può cercare su *Internet*, usando parole chiave del tipo "*good background tiles*" o "*nice background patterns*". Con le prime si può "finire" su designshard e, indirettamente, su dromoscopio.

Una volta memorizzata la figura col `immagine.gif`, nella stessa cartella che contiene il seguente testo XHTML:

```
<html>
 <head>
  <title>Ciao su immagine</title>
 </head>
 <body background="immagine.gif">
  CIAO!
 </body>
</html>
```

verificare che essa sia interpretata ripetendo l'immagine come previsto, e sovrapponedo allo sfondo la scritta "CIAO!".

Esercizio 6 ("Ciao" su immagine e sfondo.) Utilizzare simultaneamente i due attributi `bgcolor` e `background`, come nel testo XHTML seguente:

```
<html>
 <head>
  <title>Ciao su immagine e sfondo colorato</title>
 </head>
```

```
 <body bgcolor="#0000FF" background="immagine.gif">
  CIAO!
 </body>
</html>
```

e verificare che l'interpretazione ne combina i due effetti: l'immagine di sfondo viene visualizzata, e lo sfondo della pagina colorato.

2.2 Testo

2.2.1 Impostare il colore del testo in un'intera pagina

Per definizione un *browser* usa il colore nero per rappresentare il testo contenuto nel *body* di un documento XHTML. Il colore del testo in tutto il documento si determina assegnando un valore all'attrbuto `text` del *tag* `body`. I valori utilizzabili per `text` sono identici a quelli elencati per impostare il colore di sfondo.

Esercizio 7 ("Ciao" bianco su blu.) Una volta verificato che il seguente documento XHTML:

```
<html>
 <head>
  <title>Ciao bianco su blu</title>
 </head>
 <body bgcolor="#0000ff" text="#ffffff">
 CIAO!
 </body>
</html>
```

è interpretato, scrivendo "CIAO!" in bianco su sfondo blu, cambiare a piacere i valori degli attributi `bgcolor` e `text`, per variare gli accostamenti di colore.

2.2.2 Titoli, paragrafi e blocchi di testo

Il testo contenuto nel *tag* `body` può essere etichettato affinché la sua presentazione tipografica sia gradevole e significativa per il lettore. Ad esempio, le parti di testo che costituiscono il titolo di una eventule sezione dovranno risultare più evidenti, rispetto al testo dei paragrafi, esattamente come succede con un usuale testo stampato.

Illustriamo i principali *tag* di strutturazione tipografica del testo, per mezzo di un esempio.

Esempio 3 (Intestazioni e paragrafi.) Segue un testo XHTML in cui compaiono alcuni tipi diversi di intestazione e di paragrafi:

```
<html>
 <head>
  <title>Intestazioni e paragrafi</title>
 </head>
 <body bgcolor="#ffffff">

  <h1>Testo di un titolo</h1>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>
```

```
  <h2>Testo di un titolo meno vistoso</h2>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>

  <h3>Testo di un titolo ancora meno vistoso</h3>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>

 </body>
</html>
```

Commento 3

- Esiste una gerarchia di *tag* per specificare intestazioni, o, equivalentemente, titoli. La gerarchia è composta da sei livelli: da `h1`, che produce il titolo più evidente fino a `h6`, per il titolo più discreto.

  La lettera "`h`" in ciascuno dei *tag* `h1`, ..., `h6` sta per *heading*, ovvero intestazione.

- **Ogni intestazione è un blocco.** Ogni tag `h1`, ..., `h6` è un *elemento blocco* nel senso che l'interpretazione di un *browser* qualsiasi interrompe il flusso del testo, lasciando una riga vuota prima e dopo il testo racchiuso da un *tag* di tipo *heading*.

- **Ogni paragrafo è un blocco.** Il paragrafo è l'unità di base di suddivisione di un testo. Ogni *tag* `p` è un elemento di tipo blocco nel senso che l'interpretazione di un *browser* qualsiasi interrompe il flusso continuo del testo, lasciando una riga vuota prima dell'etichetta di apertura `<p>` e dopo l'etichetta di chiusura `</p>`.

2.2.3 Allineare il testo

Tutti i *tag* introdotti finora permettono di allineare il testo, assegnando valori opportuni all'attributo `align`.

Esercizio 8 (Allineare il testo.) Verificare che, su uno sfondo rosa, il seguente documento XHTML:

```
<html>
 <head>
  <title>Allineare il testo</title>
 </head>
 <body bgcolor="pink">

  <h1 align="center">Testo di un titolo</h1>
   <p align="justify">
      Testo di un paragrafo. Testo di un paragrafo.
```

```
      Testo di un paragrafo. Testo di un paragrafo.
      Testo di un paragrafo. Testo di un paragrafo.
   </p>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>

  <h2 align="right">Testo di un titolo meno vistoso</h2>
   <p>Testo di un paragrafo. Testo di un paragrafo.
      Testo di un paragrafo. Testo di un paragrafo.
      Testo di un paragrafo. Testo di un paragrafo.
   </p>
   <p>Testo di un paragrafo. Testo di un paragrafo.
      Testo di un paragrafo. Testo di un paragrafo.
      Testo di un paragrafo. Testo di un paragrafo.
   </p>

 </body>
</html>
```

è interpretato in modo che: il primo titolo sia centrato, grazie al valore `center` dell'attributo `align`, il primo paragrafo sia giustificato, grazie all'uso del valore `justify`, il secondo titolo sia allineato a destra, grazie all'uso del valore `right`.

2.2.4 Andare a capo

Il flusso di un testo può essere spezzato senza l'utilizzo di un elemento blocco come il paragrafo `p`, usando il *tag break* (interruzione) `br`.

Esercizio 9 (Andare a capo.) Verificare che il seguente documento XHTML:

```
<html>
 <head>
  <title>Andare a capo</title>
 </head>
 <body>
  <h1>Testo di un titolo</h1>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>
   <br/>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>
   <br/>
   <br/>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>
   <br/>
   <br/>
   <br/>
   <p>Testo di un paragrafo. Testo di un paragrafo.</p>
 </body>
</html>
```

è interpretato:

1. andando a capo tra il primo ed il secondo paragrafo,

2. andando a capo, e lasciando una riga vuota, tra il secondo ed il terzo paragrafo,

3. andando a capo, e lasciando due righe vuote, tra il terzo ed il quarto paragrafo.

2.2.5 Scegliere lo stile

Con "stile di testo" si intende la variante di un carattere tipografico.

L'XHTML mette a disposizione *tag* che definiscono lo stile tipografico dei caratteri. Seguono quelli principali:

- Il *tag* `b` usato come in: `<b>testo in grassetto</b>` genera testo in grassetto.

- Il *tag* `i` usato come in: `<i>testo in corsivo</i>` genera testo in corsivo.

  Siccome, in generale, la leggibilità del corsivo in un *browser* non è di buona qualità è consigliabile evitare di evidenziare in corsivo blocchi di lunghezza considerevole. Meglio limitarsi a poche parole.

- Il *tag* `u` usato come in: `<u>testo sottolineato</u>` genera testo sottolineato, in genere da evitare, perché si confonde con i *link*, vale a dire, con i collegamenti ipertestuali.

- Il *tag* `sup` usato come in: `E=mc<sup>2</sup>` genera testo spostato al di sopra della linea di scrittura ed è utile per scrivere, ad esempio, gli esponenti delle potenze in formule matematiche.

  Il *tag* `sub`, al contrario, genera testo spostato al di sotto della linea di scrittura, come in: `H<sub>2</sub>O`.

2.2.6 Scegliere il font del testo.

Ogni *browser* è predefinito per presentare un testo attraverso il tipo di carattere, identificato col nome `Times`.

Caratteri con grazie. Il tipo di carattere `Times` è ottimo per la carta stampata, ma affatica la lettura del testo, visualizzato sullo schermo di un computer. Il "problema" nasce dal fatto che `Times` è un tipo di carattere della famiglia *con grazie*, ovvero con quegli abbellimenti delle lettere, che, sulla carta, rendono più leggibile il testo.

Caratteri senza grazie. Per la presentazione di un testo sullo schermo di un computer sono più adatti i tipi di carattere della famiglia *senza grazie*. Quelli più diffusi, che essenzialmente ogni calcolatore è in grado di visualizzare, sono: `Verdana`, `Arial` e `Helvetica`.

L'attributo `face` del *tag* `font` permette di determinare il tipo di carattere usato per la visualizzazione del testo.

Esempio 4 (Usare il tipo di carattere `Helvetica`.) La riga di testo XHTML:

```
<font face="Helvetica, sans-serif">Senza grazie</font>
```

indica al *browser* del computer in uso quanto segue:

- se il tipo di carattere `Helvetica` è disponibile, allora il testo "`Senza grazie`" è rappresentato col tipo di carattere `Helvetica`;
- Altrimenti, viene usato il tipo di carattere senza gazie (`sans-serif`) disponibile.
- Nel caso fosse indisponibile un qualsiasi tipo di carattere senza grazie, viene utilizzato il tipo di carattere di *default* con grazie `Times`.

Esercizio 10 (Tipi di carattere diversi.) Interpretare con un *browser* il seguente documento XHTML:

```
<html>
 <head>
  <title>Tipi di testo diversi</title>
 </head>
 <body>
  <h3>Titolo di un paragrafo con testo senza grazie</h3>
   <p>
    <font face="Verdana
               ,Arial
               ,Helvetica
               ,sans-serif">
               Testo di un paragrafo senza grazie.
               Testo di un paragrafo senza grazie .
    </font>
   </p>
  <h3>Titolo di un paragrafo con testo con grazie</h3>
   <p>
    <font face="Georgia
               ,Times New Roman
               ,Times
               ,serif">
               Testo di un paragrafo con grazie.
               Testo di un paragrafo con grazie.
```

```
    </font>
   </p>
 </body>
</html>
```

e verificare che il primo paragrafo sia riprodotto, usando un qualche tipo di carattere senza grazie, mentre il secondo paragrafo è interpretato con tipi di caratteri dotati di grazie.

2.2.7 Scegliere il colore del testo

Per stabilire il colore del testo, anche carattere per carattere, occorre assegnare un valore all'attributo `color` del *tag* `font`. Il valore è uno di quelli legali per la definizione del colore di sfondo di un documento XHTML.

Esempio 5 (Testo (carattere per carattere) colorato.) Ciascuna delle due righe seguenti:

```
<font color="blue">testo blu</font>
<font color="#0000FF">testo blu</font>
```

inserita in un testo XHTML, colora di blu le due parole "testo blu".

Esempio 6 (Colore e tipo, scelti simultaneamente.) Colore e tipo di carattere possono essere scelti simultaneamente, assegnando gli opportuni valori agli attributi `face` e `color` del *tag* `font`, come nelle seguenti righe di codice XHTML:

```
<font face="Verdana, Arial, Helvetica, sans-serif"
      color="blue"> testo blu in Verdana</font>
<font face="Verdana, Arial, Helvetica, sans-serif"
      color="red">testo rosso</font>
```

Esercizio 11 (Codice diverso, stesso colore.) Esiste una spiegazione intuitiva sul perché, inserendo, uno alla volta, i due seguenti frammenti di codice XHTML, l'uno diverso dall'altro, in uno stesso documento XHTML, si ottiene lo stesso risultato?

```
<!-- primo blocco di codice -->
<font face="Verdana, Arial, Helvetica, sans-serif"
      color="blue">
  testo blu in Verdana
</font>
<br/>
<font face="Verdana, Arial, Helvetica, sans-serif"
      color="red">
  testo rosso
</font>
```

```
<!-- secondo blocco di codice -->
<font face="Verdana, Arial, Helvetica, sans-serif"
      color="blue">
  testo blu in Verdana
  <br/>
  <font color="red">
    testo rosso
  </font>
</font>
```

2.2.8 Digressione

Sebbene non sia molta, l'esperienza accumulata sin qui nel produrre semplici documenti XHTML permette di poter confrontare la qualità del codice XHTML prodotto da elaboratori di testo che permettano di memorizzare un testo in formato `.html`.

Supponiamo che il testo XHTML dell'Esercizio 10 sia stato memorizzato nel *file* `uno.html`, ad esempio, usando il programma XRay 2.0, e sia letto da un *browser*.

Supponiamo di "copiare ed incollare" il testo che compare nella finestra del *browser* nel programma `Microsoft Word`.

La presentazione tipografica dei testi nel *browser* e in `Microsoft Word` dovrebbero essere identiche.

Il passo successivo è memorizzare due *file* distinti `due.mht` e `tre.html`, usando `Microsoft Word`. Domande:

- Il *file* `due.mht` viene interpretato dal *browser* `Mozilla Firefox`?
- Il *file* `tre.html` viene interpretato dal *browser* `Mozilla Firefox`?
- I *file* `due.mht` e `tre.html` superano il controllo sintattico effettuato da XRay 2.0, ad esempio?

In definitiva, il formato `html` prodotto da `Microsoft Word` sembra allineato con quello ritenuto standard?

2.3 Elenchi

L'XHTML permette di definire elenchi di tre tipi:

- ordinati;
- non ordinati;
- di definizioni.

Il meccanismo definitorio ed interpretativo è analogo per tutti e tre i tipi di elenco:

- esiste un *tag* contenitore che delimita inizio, fine e tipo dell'elenco;
- all'interno di questo *tag* si elencano gli elementi della lista, ciascuno individuato da un opportuno, ulteriore *tag*, in accordo con la seguente struttura generale:

```
<elenco-di-elementi>
    <elemento>primo elemento</elemento>
    <elemento>secondo elemento</elemento>
    <!-- .... e cosi' via -->
</elenco-di-elementi>
```

2.3.1 Elenchi ordinati

Il *tag* che definisce un elenco ordinato è `ol` che significa *ordered list*, mentre gli elementi sono definiti dal *tag* `li` che significa *list item*.

Gli elenchi ordinati sono contraddistinti dall'enumerazione progressiva ed ordinata degli elementi che compongono la lista.

Esercizio 12 (Un elenco ordinato.) Verificare che il seguente testo:

```
<p>
Testo che precede la lista.
  <ol>
    <li>primo elemento</li>
    <li>secondo elemento</li>
    <li>terzo elemento</li>
  </ol>
Testo che segue la lista.
</p>
```

inserito in un documento XHTML, produce un elenco ordinato composto da tre punti.

Ciascun punto è numerato con cifre arabe in ordine crescente, con le seguenti caratteristiche:

- l'elenco lascia una riga di spazio prima e dopo il testo che, eventualmente lo circonda, come avviene per il *tag* p;
- gli elementi dell'elenco sono rientrati di uno spazio verso destra.

Esercizio 13 (Elenchi ordinati annidati.) Scrivere un testo XHTML che generi un elenco ordinato di tre punti in cui, ogni punto, contenga un elenco ordinato di due punti. Notare che l'inclusione di un nuovo elenco all'interno di un elenco preesistente non lascia spazio, né prima, né dopo l'elenco.

L'attributo `type` del *tag* `ol` permette di modificare la numerazione degli elementi, non necessariamente individuata da numeri arabi.

Esercizio 14 (Numerazioni alternative a quella araba.) Inserendo uno alla volta le seguenti linee di codice:

```
<!-- Numeri arabi -->
 <ol type="1">

<!-- Lettere minuscole -->
 <ol type="a">

<!-- Lettere maiuscole -->
 <ol type="A">

<!-- Numeri romani minuscoli -->
 <ol type="i">

<!-- Numeri romani maiuscoli -->
 <ol type="I">
```

in un testo XHTML, sperimentare numerazioni alternative.

2.3.2 Elenchi non ordinati

Il *tag* che definisce un elenco non ordinato è `ul` che significa *unordered list*, mentre gli elementi sono definiti dal *tag* `li` che significa *list item*, esattamente come per il tag `ol`.

Gli elenchi non ordinati sono contraddistinti dall'elencazione per punti degli elementi che compongono la lista.

Esercizio 15 (Un elenco non ordinato.) Verificare che il seguente testo:

```
<p>
Testo che precede la lista.
  <ul>
    <li>primo elemento
    <li>secondo elemento
```

```
    <li>terzo elemento
  </ul>
Testo che segue la lista.
```

inserito in un documento XHTML, produce un elenco non ordinato composto da tre punti, individuati da un pallino pieno.

Esercizio 16 (Elenchi non ordinati annidati.) Scrivere un testo XHTML che generi un elenco non ordinato di due punti in cui, ogni punto, contenga un elenco non ordinato di due punti.

Esercizio 17 (Elenchi ordinati e non ordinati, annidati.) Scrivere un testo XHTML che mescoli arbitrariamente elenchi ordinati e non ordinati.

L'attributo `type` del *tag* `ul` permette di modificare il simbolo con cui vengono evidenziati gli elementi dell'elenco.

Esercizio 18 (Punti non ordinati, alternativi.) Inserendo uno alla volta le seguenti linee di codice:

```
<!-- Pallino pieno -->
 <ul type="disc">

<!-- Pallino vuoto -->
 <ul type="circle">

<!-- Quadratino -->
 <ul type="square">
```

in un testo XHTML, determinare il tipo di carattere, usato dal *browser* per elencare i punti della lista non ordinata.

2.3.3 Elenchi di definizioni

Il *tag* che definisce un elenco di definizioni è `dl` che significa *definition list*.

Gli elementi dell'elenco sono definiti per mezzo di due *tag*, raffinando lo schema iniziale della definizione di elenchi.

I due *tag* sono `dt` che significa *definition term*, e `dd` che significa *definition description*.

Gli elenchi di definizioni elencano i *list item*, usando il contenuto del figlio dell'elemento `dt` per contraddistinguere il punto all'interno dell'intero elenco.

Esercizio 19 (Un elenco di definizioni.) Verificare che il seguente testo:

```
<p>
Testo che precede l'elenco.
  <dl>
    <dt>termine 1</dt><dd>descrizione 1</dd>
```

```
    <dt>termine 2</dt><dd>descrizione 2</dd>
    <dt>termine 3</dt><dd>descrizione 3</dd>
  </dl>
Testo che segue l'elenco.
```

inserito in un documento XHTML, produce un elenco di definizioni in cui:

- il *tag* `dt` non produce alcun rientro, a differenza del *tag* `li`,
- il rientro è prodotto dal *tag* `dd`.

Esercizio 20 (Annidiamo arbitrario di elenchi.) In un testo XHTML, mescolare arbitrariamente elenchi ordinati, non ordinati e liste di definizione.

2.4 Le tabelle

2.4.1 Tabella: struttura di base

L'XHTML permette di definire tabelle che organizzano il testo in griglie con un numero prefissato di colonne ed un numero arbitrario di righe.

Il *tag* di creazione delle tabelle è `table`.

Ogni riga di tabella è individuata dal *tag* `tr`, che significa *table row*. Il *tag* `table` può contenere un numero arbitrario di righe.

Ogni riga può contenere celle, create con il *tag* `td`, che significa *table data*.

Il tag `tabella` ha l'attributo `border` che stabilisce lo spessore delle linee che, idealmente, costituiscono la griglia della tabella.

Esercizio 21 (Tabella 2×2.) Verificare che il seguente codice:

```
<table border="1">
    <tr>
       <td>prima</td>
       <td>seconda</td>
   </tr>
   <tr>
      <td>terza</td>
      <td>quarta</td>
   </tr>
</table>
```

inserito in un documento XHTML, produce una tabella di due righe per due colonne, con un sottile bordo tra una cella e l'altra.

Sperimentare l'effetto sullo spessore delle linee della griglia, variando il valore dell'attributo `border`, od eliminandolo completamente.

Esercizio 22 (Tabella 3×2.) Verificare che il seguente codice:

```
<table border="1">
    <tr>
       <td>prima</td>
       <td>seconda</td>
   </tr>
   <tr>
      <td>terza</td>
      <td>quarta</td>
      <td>quinta</td>
   </tr>
</table>
```

inserito in un documento XHTML, produce una tabella di due righe per tre colonne, in cui l'ultima cella della seconda riga non ha corrispondente nella prima riga.

Tutti i *tag* `table`, `tr` e `td` ammettono gli attributi `width` (ampiezza) e `height` (altezza). Tali parametri accettano valori di ampiezza ed altezza in *pixel*, i punti che costituiscono le immagini ed i testi sullo schermo, ed in termini percentuali.

Esercizio 23 (Ampiezze assolute e relative.) Verificare (a occhio) che il seguente codice:

```
<table width="300" border="1">
    <tr>
       <td width="25%">prima</td>
       <td width="75%">seconda</td>
   </tr>
   <tr>
      <td width="25%">terza</td>
      <td width="75%">quarta</td>
   </tr>
</table>
```

inserito in un documento XHTML, produca una tabella larga 300 *pixel*, e tale che la prima colonna occupi il 25% di tale ampiezza, mentre la seconda ne occupi il 75%. L'altezza delle celle è determinata dal loro contenuto.

Esercizio 24 (Ampiezze relative.) Verificare (a occhio) che il seguente codice:

```
<table width="90%" border="1">
    <tr>
       <td width="50%">prima</td>
       <td width="50%">seconda</td>
   </tr>
   <tr>
      <td width="25%">terza</td>
      <td width="75%">quarta</td>
   </tr>
</table>
```

inserito in un documento XHTML, produca una tabella larga il 90% di quella della finestra del *browser*, e tale che prima e seconda colonna occupino, ciascuna, il 50% dell'ampiezza della tabella.

Le ampiezze relative della seconda riga, quindi, sono ignorate e l'altezza delle celle è determinata dal loro contenuto.

Un ulteriore attributo, valido per `table`, `tr` e `td` è `align`, che allinea l'intera tabella, od il contenuto delle celle, come desiderato. I valori ammessi per `align`, per tutti i tag `table`, `tr` `td`, sono: `left`, `center` e `right`. Il valore `justify` è invece interpretato correttamente come arogmento di `tr` e `td`.

Esercizio 25 (Allineamento del testo in tabella.) Verificare (a occhio) che il seguente codice:

```
<table align="right" width="90%" border="1">
 <tr>
  <td align="justify"
      width="30%"> justify justify justify justify
                   justify justify justify justify
                   justify justify justify justify</td>
  <td align="right"
      width="70%"> right right right right right right right
                   right right right right right right right
                   right right</td>
  </tr>
  <tr>
    <td align="center"
        width="30%"> center center center center center
                     center center center center center
                     center center center center</td>
    <td align="left"
        width="70%"> left left left left left left left
                     left left left left left left left
                     left left left left left left</td>
   </tr>
</table>
```

inserito in un documento XHTML, produca una tabella larga il 90% di quella della finestra del *browser*, in cui il testo delle varie celle è allineato come indicato dai valori degli attributi `align`.

Ulteriore strutturazione di tabelle. Esistono *tag* per un'ulteriore strutturazione delle tabelle:

- `caption`, da usarsi come primo figlio del *tag* `table`. Serve per associare o una descrizione o un titolo alla tabella;
- `thead`, da usarsi come secondo figlio del *tag* `table`. Individua un insieme di righe che costituiscono l'intestazione delle varie colonne, presenti in una tabella.

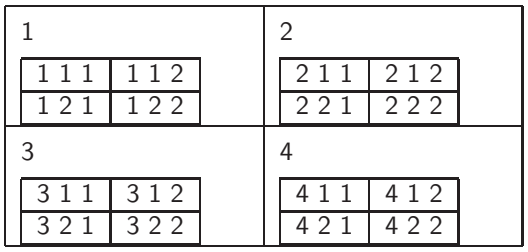
  Le celle figlie del *tag* `thead` possono essere individuate dal *tag* `th`, invece che `td`. `th` sta per *table head* ed il testo in esso contenuto è rappresentato in grassetto;
- `tfoot`, da usarsi come terzo figlio del *tag* `table`. Individua un insieme di righe che costituiscono il piede delle varie colonne, presenti in una tabella;
- `tbody`, da usarsi come quarto figlio del *tag* `table`. Individua l'insieme di righe che costituiscono il corpo vero e proprio della tabella.

Esercizio 26 (Tabella con titolo, intestazione e piede.) Verificare che il seguente codice:

```
<table align="center" width="90%" border="1">
<caption>Descrizione</caption>
<thead>
    <tr>
       <th> header1 header1 header1 header1 header1 header1
            header1 header1 header1 header1 header1 header1</th>
       <th> header2 header2 header2 header2 header2 header2
            header2 header2 header2 header2 header2 header2</th>
   </tr>
</thead>
<tfoot>
    <tr>
       <td> footer1 footer1 footer1 footer1 footer1 footer1
            footer1 footer1 footer1 footer1 footer1 footer1</td>
       <td> footer2 footer2 footer2 footer2 footer2 footer2
            footer2 footer2 footer2 footer2 footer2 footer2</td>
   </tr>
</tfoot>
<tbody>
    <tr>
       <td align="justify"
           width="30%"> justify justify justify justify justify
                        justify justify justify justify justify
                        justify justify</td>
       <td align="right"
           width="70%"> right right right right right right right
                        right right right right right right right
                        right right</td>
   </tr>
   <tr>
       <td align="center"
           width="30%"> center center center center center center
                        center center center center center center
                        center center</td>
       <td align="left"
           width="70%"> left left left left left left left left
                        left left left left left left left left
                        left left left left</td>
   </tr>
</tbody>
</table>
```

inserito in un documento XHTML, produce una tabella con un titolo “Descrizione”, una riga di intestazione in grassetto, due righe di corpo, ed una riga di piede.

Esercizio 27 (Tabelle annidate.) Scrivere il codice XHTML che generi una tabella come la seguente:



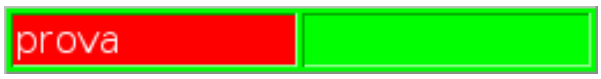
2.4.2 Ulteriori attributi di `table`, `tr`, `td`

Con i *tag* `table`, `tr`, `td`, o `th`, in analogia al tag `body`, è possibile determinare lo sfondo di una tabella, imponendo che esso sia di un certo colore, tramite l'attributo `bgcolor`.

Esercizio 28 (Colori di sfondo di una tabella e delle celle.) Verificare che il seguente codice:

```
<table width="75%"
       border="1"
       align="center"
       bgcolor="#00FF00">
    <tr>
      <td width="50%"
          bgcolor="#FF0000">
              <font color="#FFFFFF">prova</font>
          </td>
         <td width="50%"></td>
     </tr>
</table>
```

inserito in un documento XHTML, produca una tabella come la seguente:



e spiegare il perché.

Infine, con l'attributo `nowrap` si può impedire che il testo una cella vada a capo, se non impartendo esplicitamentea un comando *break* `br`.

Esercizio 29 ("A capo" esplicito in una tabella.) Verificare che il seguente codice:

```
<table width="100"
       border="1">
   <tr>
      <td nowrap>
           Se non lo vogliamo non si va a capo.
           <br/>
           Qui si va a capo.
       </td>
    </tr>
</table>
```

inserito in un documento XHTML, produca una tabella larga 100 *pixel*, che non sono sufficenti a contenere il testo:

`Se non lo vogliamo non si va a capo.`

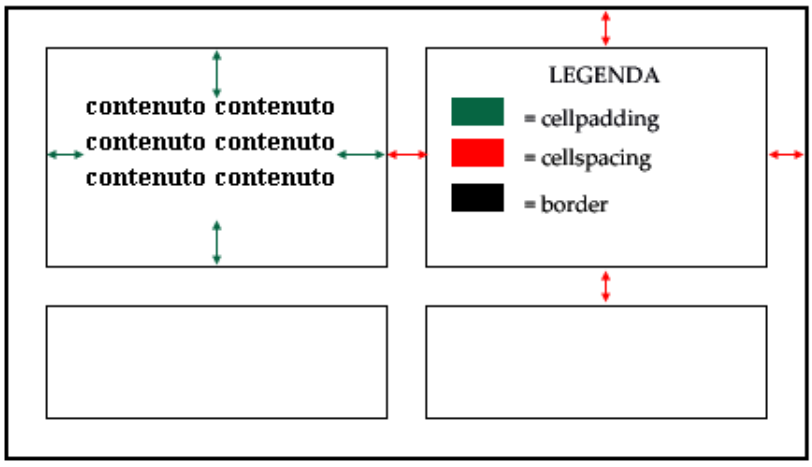
Nonostante questo, l'effetto dell'attributo `nowrap` evita che tale testo sia spezzato. L'unico modo per andare a capo diventa il *tag* `br`.

2.4.3 Ulteriori attributi del solo *tag* `table`

Due attributi del *tag* `table` permettono di regolare le distanze tra i margini della tabella, o delle celle, ed il contenuto di queste ultime:

- `cellspacing` specifica la distanza in *pixel* tra una cella e l'altra, oppure tra una cella e il bordo. Il valore predefinito è un *pixel*.
- `cellpadding` indica la distanza tra il contenuto di una cella ed il bordo della tabella: un numero esprime la distanza in *pixel*, ma il valore assegnato a `cellpadding` può essere una quantità percentuale. Il valore predefinito assegnato a `cellpadding` è zero.

Segue una rappresentazione grafica del rapporto tra i valori degli attributi `cellspacing` e `cellpadding`:



2.5 *Link*

I *link* sono lo strumento con cui "saltare" da un testo all'altro, costruendo iper-testi, costituiti da documenti XHTML. Ogni *link* si compone di due parti:

- **Contenuto**. Esso è la parte del *link* resa visibile nel testo dal *browser*, generalmente in colore blu, con stile sottolineato.

- **Indirizzo.** Al contrario del contenuto, esso non è visualizzato dal *browser*.

  L'indirizzo può indicare una pagina diversa da quella che contiene il *link* in oggetto, oppure può indicare un punto, interno della pagina che contiene il *link* stesso.

"Click". Con un "click" sul testo visibile del *link*, il *browser* interpreta l'indirizzo associato e si sposta, se possibile, nel punto descritto dall'indirizzo.

2.5.1 Sintassi generale per i *link*

La sintassi generale del *tag* per la definizione di un *link* è:

```
<a href="Indirizzo">Contenuto</a>
```

Il nome del *tag* `a` è un'abbreviazione per *anchor*, ovvero àncora.

La scelta di questo nome è legato all'interpretazione del meccanismo di funzionamento delle componenti il *link*.

2.5.2 Metafora di interpretazione dei *link*

Testa di un *link*. Ogni *link* `<a href="Indirizzo">Contenuto</a>` in un documento `X.html` è un'àncora con la testa nel documento `X.html` stesso.

Coda di un *link*. La coda dell'àncora è (la radice) del documento che si trova ad `Indirizzo`. Ovvero, i *link* creano àncore virtuali tra due testi XHTML.

"Luogo" della coda. La coda dell'àncora può essere sia un punto del documento `X.html`, sia la radice di una altro documento XHTML `Y.html`.

- ***Link* interni.** Nel caso la coda sia un punto del documento `X.html` si parla di *link* interno.

- ***Link* esterni.** Nel caso la coda sia la radice di un altro documento `Y.html` si parla di *link* esterno.

2.5.3 *Link* esterni

Un *link*:

```
<a href="Indirizzo">Contenuto</a>
```

è esterno se `Indirizzo` assume la forma di un indirizzo assoluto.

Esempio 7 (Un *link* esterno con indirizzo assoluto.) Nel *tag anchor*:

```
<a href="http://www.di.unito.it/index.html">
    Dipartimento di Informatica -- Torino
</a>
```

il contenuto è: "`Dipartimento di Informatica -- Torino`".

La **testa** dell'àncora è il documento che lo contiene.

La **coda** dell'àncora è il documento il cui indirizzo assoluto è indicato dal valore:

```
http://www.di.unito.it/index.html
```

dell'attributo `href`.

Componenti l'indirizzo assoluto. L'indirizzo assoluto è composto di alcune parti:

- `index.html` indica il nome del documento XHTML da interpretare, trasferendolo da un calcolatore a quello in uso.
- `www.di.unito.it` individua il calcolatore da cui prelevare il documento `index.html`.

  `www.di.unito.it` è l'indirizzo mnemonico di un calcolatore all'interno di tutta la rete Internet.
- `http://` indica il protocollo con cui negoziare lo scambio dei documenti tra il calcolatore in uso e quello con indirizzo `www.di.unito.it`. `http` è un acronimo per *hyper-text transfer protocol*, ovvero protocollo per il trasferimento di iper-testi.

2.5.4 Percorsi relativi

Nell'Esempio 7, precedente, supponiamo che il testo che contiene

```
<a href="http://www.di.unito.it/index.html">
    Dipartimento di Informatica -- Torino
</a>
```

sia `X.html` e sia memorizzato proprio sul calcolatore `www.di.unito.it` così come lo è `index.html`.

Le componenti `http://` e `www.di.unito.it/` del valore dell'attributo `href` diventano, allora, inutili.

Ovvero, per "saltare" da `X.html` a `index.html` basta il *tag*:

```
<a href="index.html">
    Dipartimento di Informatica -- Torino
</a>
```

Quando l'indirizzo assoluto è inutile. Se la testa e la coda dell'àncora sono due documenti XHTML distinti, ma ***memorizzati nella stessa cartella dello stesso calcolatore***, è sufficiente specificare solo il nome del documento "coda" come valore di `href`.

Esercizio 30 (Un iper-testo con due documenti.) Sia `X.html` il seguente documento XHTML:

```
<html>
 <head>
  <title>Documento testa</title>
 </head>
 <body>
  <a align="center"
     href="Y.html">Salta alla coda</a>
 </body>
</html>
```

Sia `Y.html` il seguente documento XHTML:

```
<html>
 <head>
  <title>Documento coda</title>
 </head>
 <body>
  Sei arrivato alla coda!
  <br/>
  <a align="center"
     href="X.html">Salta alla testa</a>
 </body>
</html>
```

Memorizziamo entrambi i documenti in una stessa cartella di uno stesso calcolatore. Interpretare `X.html` con un *browser* e verificare come sia possibile "saltare" alternativamente tra `X.html` ed `Y.html`, usando i *link* in essi contenuti.

2.5.5 Code "arbitrarie" di *link*

La coda dell'àncora non deve essere necessariamente un testo XHTML.

Essa può essere un documento immagine, un archivio, etc..

Interpretazioni di code non in formato XHTML

A seconda della natura del documento àncora cambia il meccanismo d'interpretazione del *browser*.

- **Immagini.** Documenti immagine, con estensione, ad esempio, `.gif`, `.jpg` o `.png`, vengono direttamente visualizzati nella pagina del *browser*.

  Un esempio di *link* con un indirizzo assoluto ad un file immagine di tipo `.jpg` è il seguente:

  ```
  <a href="http://www.mdxc.org/antarctica/tramonto.jpg">
  Tramonto lunare</a>.
  ```

  Osserviamo che l'estensione del file non è `.html`, ma `.jpg`.

- **HTML standard.** Documenti XHTML, con estensione `.html`, oppure *Portable Document Format*, con estensione `.pdf`, vengono visualizzati direttamente nel *browser*, ammesso che, per i documenti `.pdf`, sia installato almeno il programma *reader* associato. In caso contrario, il *browser* chiederà all'utente se memorizzare il documento sul computer in uso.

  Un esempio di *link* con un indirizzo relativo ad un file archivio di tipo `.zip` è il seguente:

  ```
  <a href="archivio.zip">Archivio</a>.
  ```

  Osserviamo che l'estensione del file non è `.html`, ma `.zip`.

- **Archivi.** Documenti archivio, con estensione `.zip`, vengono salvati sul calcolatore in uso, o "aperti" da un apposito programma.

- **Indirizzi di posta elettronica.** Il *tag*:

  ```
  <a href="mailto:NomeAccount@IndirizzoServerDiPosta">
  Mandami un'e-mail</a>.
  ```

  tramite un *click* sul contenuto `Mandami un'e-mail`, richiama, automaticamente, il programma per l'invio della posta elettronica.

Commento 4 (Nomi dei documenti.)
È utile usare un paio di accorgimenti per la definizione dei nomi di documenti:

- è consigliabile non lasciare spazi vuoti nei nomi dei documenti perché non sempre vengono interpretati uniformemente dai sistemi operativi, installati sui calcolatori con cui ci si connette attraverso Internet. In generale, lo spazio deve essere rimpiazzato per mezzo del trattino di sottolineatura: `nome_documento.html`

- maiuscole e minuscole, usate nel nome di un documento, possono fare la differenza. È, quindi, necessario essere coerenti con i criteri di assegnazione dei nomi ai documenti.

2.5.6 *Link* interni

Un *link* è interno se il valore dell'attributo `href`, che specifica l'indirizzo della coda dell'àncora, è preceduto dal carattere #, detto cancelletto o *sharp*:

```
<a href="#Indirizzo">Contenuto</a>
```

In tal caso la coda dell'àncora *corrispondente* deve essere specificata con la seguente sintassi:

```
<a name="Indirizzo">Contenuto della coda</a>
```

L'attributo `name` del *tag* `a` costituisce la coda di un'àncora. Un *browser* "salta" ad essa se il suo valore coincide con quello di un attributo `href`, di un *tag* `a`, contenuto nello stesso documento, ed il cui primo carattere sia #.

Esercizio 31 (Singolo iper-testo con *link* interni.) Sia `X.html` il seguente documento XHTML:

```
<html>
  <head>
    <title>Documento con link interno</title>
  </head>
  <body>
   <h2>
      <a name="I-paragrafo">Primo paragrafo</a>
    </h2>
    <p>
      <a href="#II-paragrafo">Al secondo paragrafo</a>
    </p>
    <p>
  Primo Primo Primo Primo Primo Primo Primo Primo Primo Primo
  Primo Primo Primo Primo Primo Primo Primo Primo Primo Primo
  Primo Primo Primo Primo Primo Primo Primo Primo Primo Primo
  Primo Primo Primo Primo Primo Primo Primo Primo Primo Primo
  </p>
    <h2>
      <a name="II-paragrafo">Secondo paragrafo</a>
    </h2>
    <p>
      <a href="#I-paragrafo">Al primo paragrafo</a>
    </p>
    <p>
  Secondo Secondo Secondo Secondo Secondo Secondo Secondo Secondo
  Secondo Secondo Secondo Secondo Secondo Secondo Secondo Secondo
  Secondo Secondo Secondo Secondo Secondo Secondo Secondo Secondo
  Secondo Secondo Secondo Secondo Secondo Secondo Secondo Secondo
  </p>
```

```
 </body>
 </html>
```

Interpretare `X.html` con un *browser* e verificare come sia possibile "saltare" alternativamente tra i due paragrafi ivi contenuti. La verifica avrà successo solo restringendo, a sufficienza, l'ampiezza verticale della finestra del *browser*.

2.5.7 Alcuni ulteriori attributi di *link*

Per definizione, un *browser* interpreta la coda dell'àncora, visualizzando il documento XHTML corrispondente al posto di quello che contiene la testa dell'àncora.

Attributo `target`

L'attributo `target` del *tag* `a` permette di modificare tale comportamento. In particolare, se il suo valore è `_blank` la coda dell'àncora viene visualizzata in una nuova finestra del *browser*.

Esercizio 32 (Apertura in una nuova finestra.) Sia `X.html` il seguente documento XHTML:

```
<html>
  <head>
    <title>Documento con link interno</title>
  </head>
  <body>
   <h2>
      <a name="I-paragrafo">Primo paragrafo</a>
    </h2>
    <p>
      <a href="#II-paragrafo"
         target="_blank">Al secondo paragrafo</a>
    </p>
    <p>
  Primo Primo Primo Primo Primo Primo Primo Primo Primo Primo
  Primo Primo Primo Primo Primo Primo Primo Primo Primo Primo
  Primo Primo Primo Primo Primo Primo Primo Primo Primo Primo
  Primo Primo Primo Primo Primo Primo Primo Primo Primo Primo
  </p>
    <h2>
      <a name="II-paragrafo">Secondo paragrafo</a>
    </h2>
    <p>
      <a href="#I-paragrafo">Al primo paragrafo</a>
    </p>
    <p>
  Secondo Secondo Secondo Secondo Secondo Secondo Secondo Secondo
```

```
 Secondo Secondo Secondo Secondo Secondo Secondo Secondo Secondo
 Secondo Secondo Secondo Secondo Secondo Secondo Secondo Secondo
 Secondo Secondo Secondo Secondo Secondo Secondo Secondo Secondo
 </p>
</body>
</html>
```

Interpretare `X.html` con un *browser* e verificare che ogni "salto" al Secondo paragrafo apre una nuova finestra del *browser*.

Attributo `title`

L'attributo `title` del *tag* `a` permette di descrivere, attraverso un breve testo, il significato del testo XHTML che costituisce la coda dell'àncora. Il testo esplicativo compare nel momento in cui la "mosca" del *mouse* si sofferma per una qualche frazione di secondo sul contenuto di un *link*.

Esercizio 33 (Attributo `title` del *tag* `a`.) Salvato il seguente codice:

```
<a title="Home page del Dipartimento di Informatica
          dell'Universita' di Torino"
   href="http://www.di.unito.it/index.html" >
   Dipartimento di Informatica -- Torino
</a>
```

in un testo XHTML, interpretarlo con un *browser*, e far comparire sullo schermo il valore dell'attributo `title`.

Commento 5
I motori di ricerca usano anche il contenuto dell'attributo `title` del *tag* `a` per catalogare i documenti XHTML.

2.6 Esercizi riassuntivi

Esercizio 34 (Insieme di documenti XHTML riassuntivi.) 1. Considerare un concetto ragionevolmente strutturato. Esempi possibili sono un *curriculum vitae*, un documento che presenti una nostra attività, la pubblicità per un viaggio.

2. Impostare un documento XHTML `radice.html` "corrispondente", orientativamente con la seguente struttura:

figura	Intestazione
Link a `Y.html`	Testo Testo Testo Testo Testo Testo Testo Testo
Link a CODA	Testo Testo Testo Testo Testo Testo Testo Testo
CODA	

adatto per una rappresentazione gradevole del concetto in un *browser*.

3. Il documento `Y.html` di cui si parla in `radice.html` è da immaginare come appendice, da aprire in una nuova finestra del *browser*, con dettagli, elencati per punti, sul documento principale.

4. Per inserire la figura di cui si parla nel documento `radice.html`, sperimentare l'uso del *tag*:

```
<img src="figura.jpg"/>
```

sul cui significato è possibile informarsi, usando la documentazione su *Internet*.

2.7 Dove siamo

In prima istanza, abbiamo visto cosa facciano i *browser* per la navigazione in internet: interpretano documenti XHTML in modo che essi possano essere letti ed usati opportunamente da qualche "navigante".

Inoltre, supponendo di utilizzare eleboratori di testo che esportano quel che vi scriviamo in pagine XHTML, possiamo giudicarne la qualità. Infatti, per gioco, potremmo:

- scrivere un semplice testo per mezzo dei due eleboratori testo il Microsoft Word e Open Office Writer.
- salvare il testo, esportandono formato XHTML.
- confrontare il codice XHTML (eventualmente) generato.

L'idea è che l'elaboratore è tanto migliore quanto più raffinatamente permette di gestire le informazioni memorizzabili nella sezione `head`, dato che esse sono usate dai motori di ricerca, e quanto più essenziale è il codice XHTML generato.

Capitolo 3

Generalizziamo XHTML

3.1 Sintassi e Formalizzazione

La sintassi di un linguaggio è lo strumento da cui partire per poter *formalizzare* concetti. In questo contesto, *sinonimi* ragionevoli per "*formalizzare*" sono: strutturare, dare forma, per quanto possibile, *univoca*, evidenziare il significato di un concetto attraverso la sua sintassi, ovvero, attraverso la sua "forma".

Esistono formalizzazioni sia attraverso il linguaggio naturale, sia attraverso linguaggi "artificiali":

- Il linguaggio naturale è uno degli strumenti più diffusi per formalizzare concetti.
- I linguaggi artificiali hanno lo stesso scopo, ma possono anche soddisfare l'esigenza di poter essere interpretati da interpreti "stupidi" come un calcolatore. Per tale motivo rispettano regole non usuali, generalmente più restrittive, rispetto ai linguaggi naturali.

XML. Impareremo ad usare (la sintassi) XML come meta-sintassi adatta a formalizzare linguaggi che descrivano concetti, o entità di interesse, "arbitrariamente" complessi. Ovvero, la tecnologia XML fornisce un linguaggio che per certi versi è "universale": questo linguaggio "universale" permette di definirne altri.

Più tecnicamente, XML è un linguaggio, che ubbidisce ad una precisa grammatica e che serve per definire altre grammatiche.

3.2 Formalizzazione e linguaggio naturale

Il linguaggio naturale costituisce uno strumento *formale* per la descrizione delle entità più varie, con diversi gradi di astrazione.

Esempio 8 (Mela.) Per riferirci ad una mela, non mostriamo l'oggetto, ma formuliamo, a voce o scrivendola, la parola "mela", in grado di evocare il giusto concetto.

In questo caso, *formalizzare* significa usare le lettere dell'alfabeto, ovvero simboli, che, combinati in un'opportuna sequenza, assumono un significato (essenzialmente) univoco, invalso dall'uso.

Il linguaggio naturale è tanto sofisticato ed espressivo che permette di "parlare di se stesso".

Esempio 9 (Analisi logica.) L'analisi logica di un testo *formalizza* la struttura di un testo, esplicitando la funzione che ciascuna frase, o suo componente, gioca nel testo stesso:

- **"La maestra insegna".** "La maestra" è il soggetto e "insegna" il predicato verbale.

- **"La maestra insegna le tabelline".** "La maestra" è soggetto, "insegna" il predicato verbale e "le tabelline" il complemento oggetto.

  Il complemento oggetto è riconoscibile perché esso risponde implicitamente a: "che cosa?".

- **"Il cane di Franca abbaia".** "Il cane" è il soggetto, "abbaia" il predicato, e "di Franca" il complemento di specificazione.

  Il complemento di specificazione è riconoscibile perché esso risponde implicitamente a: "di chi?", "di cosa?".

3.3 Formalizzazione "artificiale"

Gli stessi (meta-)significati evidenziati con l'analisi logica dell'Esempio 9 si possono ritrovare nell'Esempio 10 che utilizza un linguaggio alternativo, meno naturale del precedente, ma con lo stesso scopo:

Esempio 10 (Analisi logica alternativa.)

```
<AnalisiLogica>
 <Frase>
   <FraseSemplice>
    <Soggetto valore="La maestra"/>
    <Predicato valore="insegna"/>
   </FraseSemplice>
 </Frase>

 <Frase>
   <FraseSemplice>
     <Soggetto valore="La maestra"/>
     <Predicato valore="insegna"/>
   </FraseSemplice>
   <Complemento tipo="oggetto" valore="le tabelline">
     <Domanda valore="CheCosa"/>
   </Complemento>
 </Frase>

 <Frase>
   <FraseSemplice>
     <Soggetto valore="Il cane"/>
     <Predicato valore="abbaia"/>
   </FraseSemplice>
   <Complemento tipo="specificazione" valore="di Franca">
     <Domanda valore="DiChi"/>
     <Domanda valore="DiCheCosa"/>
   </Complemento>
 </Frase>
```

```
</AnalisiLogica>
```

Commento 6
Il testo dell'Esempio 10 è scritto con la sintassi ad etichette del linguaggio XML. Alcune osservazioni immediate:

- il testo è estremamente prolisso, al limite della ridondanza di informazione;
- la struttura del testo sembra essere molto rigida;
- il testo non ha una strutturazione "naturale".

3.4 Obiettivi didattici

Uno degli obiettivi didattici è imparare ad usare strutturazioni sintattiche di concetti, attraverso l'etichettatura delle loro descrizioni, in analogia a quanto appena mostrato.

Il motivo è sperimentare i gradi di libertà nella scelta del livello di dettaglio cui si può scendere nell'analisi dei concetti da formalizzare. Più i concetti sono complessi, più la loro analisi ed etichettatura richiedono tempo ed esperienza, aspetti non irrilevanti rispetto ai costi di un progetto di informatizzazione.

3.5 Semantica ed Invarianti, ovvero proprietà comuni

- *Semantica* è sinonimo di *significato*. Impareremo a riflettere sulla nozione di semantica, col fine di definire il significato dei linguaggi descritti per mezzo dell'XML, la possibile "madre di molti (non tutti) linguaggi non naturali".

- Impareremo a scrivere una grammatica usando la grammatica del linguaggio **Data Type Definition** (DTD) adatto a specificare le proprietà strutturali invarianti, relative ad un dato concetto.

3.5.1 Semantica e linguaggio naturale

Parlando di una mela, identifichiamo un *insieme* costituito da qualsiasi esempio di frutto che soddisfi certe caratteristiche. Ovvero:

> la parola *mela* riassume un concetto che descrive una classe di esempi di frutti simili, secondo un certo insieme di criteri.

3.5.2 Semantica per linguaggi "artificiali"

La semantica di un *linguaggio non naturale*, od *artificiale*, si definisce in analogia con quanto detto per il linguaggio naturale:

> la semantica è data da regole che accomunano le strutture delle componenti una frase del linguaggio scelto.

Esempio 11 (Struttura delle frasi XML per l'analisi logica.) Segue un'idea di quali siano le regole che accomunano le strutture di una frase del testo XML che raccoglie l'analisi logica di frasi nell'Esempio 10:

- L'etichettatura `AnalisiLogica` dell'Esempio 10 può contenere un *insieme arbitrario* di frasi la cui struttura ad etichette sia quella di `Frase`.

  È come dire che ogni elemento che appartenga all'insieme delle analisi logiche di frasi in italiano ha una struttura riconducibile a quella della porzione di testo compresa tra le due etichette di nome `AnalisiLogica`. Quell'insieme definisce la semantica del concetto "analisi logica". Equivalentemente, la semantica dell'etichetta `AnalisiLogica` è costituita da tutti i risultati di una attività di analisi logica.

- Inoltre, ogni frase ha una struttura descrivibile in accordo con l'etichettatura `Frase`, ovvero, ciascuna di esse può essere costituita o da una frase con struttura `FraseSemplice`, o da una frase con struttura `FraseSemplice`, seguita da una con struttura `Complemento`.

  Analogamente al punto precedente, tutte le frasi la cui struttura sia riconducibile a quella del testo compreso tra le etichette di nome `FraseSemplice`

appartengono alla semantica del concetto "frase semplice". Equivalentemente, la semantica dell'etichetta `FraseSemplice` è costituita da tutte le "frasi semplici".

- etc. ...

La sintassi del linguaggio **Data Type Definition** (DTD), che fa parte della tecnologia XML, permette la descrizione della semantica di linguaggi XML, intesa come insieme di proprietà strutturali che accomunano le frasi di documenti XML che interessano.

Esempio 12 (Semantica DTD per tutti gli esempi di Analisi logica) Un po' di impegno ed intuizione permettono di ripercorrere nel testo DTD seguente le regole che definiscono la semantica cui appartengono tutti i testi XML che costituiscono il risultato di una qualche analisi logica dell'Esempio 10:

```
<!ELEMENT AnalisiLogica (Frase*)>
<!ELEMENT Frase (FraseSemplice | (FraseSemplice,Complemento))>
<!ELEMENT FraseSemplice (Soggetto, Predicato)>
<!ELEMENT Soggetto EMPTY>
  <!ATTLIST Soggetto valore CDATA #REQUIRED>
<!ELEMENT Predicato EMPTY>
  <!ATTLIST Predicato valore CDATA #REQUIRED>
<!ELEMENT Complemento (Domanda+)>
  <!ATTLIST Complemento tipo (oggetto | specificazione) #REQUIRED
                        valore CDATA #REQUIRED>
<!ELEMENT Domanda EMPTY>
  <!ATTLIST Domanda valore (CheCosa | DiChi | DiCheCosa) #REQUIRED>
```

La prima riga:

```
<!ELEMENT AnalisiLogica (Frase*)>
```

si legge: "Un elemento `AnalisiLogica` (di *un* testo XML in cui raccogliere i risultati di analisi logiche) è costituito da una sequenza di elementi `Frase`".

La seconda riga

```
<!ELEMENT Frase (FraseSemplice | (FraseSemplice,Complemento))>
```

si legge: "Un elemento `Frase` (di un testo XML in cui raccogliere i risultati di analisi logiche) è costituito o da un elemento `FraseSemplice`, o da un elemento `FraseSemplice`, immediatamente seguito da un elemento `Complemento`".

La descrizione delle regole di quali siano i documenti XML legali dell'insieme continua in maniera analoga fino alla determinazione della struttura degli elementi atomici, non più scomponibili.

Il linguaggio DTD descrive la struttura XML. Si osserva che il documento DTD non specifica il contenuto degli elementi, ma solo la loro struttura: ogni frase con tale struttura, indipendentemente dalle parole che contiene, appartiene alla semantica descritta dal documento DTD stesso.

Esercizio 35 (Proprietà invarianti di espressioni binarie.) Descrivere, con frasi succinte, la struttura che accomuna tutte le espressioni numeriche composte da somme o sottrazioni di numeri binari, ovvero di numeri costituiti da sequenze delle sole cifre 0 e 1.

3.5.3 Obiettivi didattici

Uno degli obiettivi didattici è imparare a determinare la semantica di un linguaggio per mezzo del linguaggio DTD.

3.6 Interpretazione e Problemi (Computazionali)

- *Interpretare* può significare il trasformare, o il tradurre, un concetto di partenza in uno equivalente, ma in un linguaggio più "chiaro".
- L'*interpretazione per traduzione* in un linguaggio alternativo è proprio il modo con cui i calcolatori risolvono problemi, computazionalmente.

Impareremo almeno a leggere algoritmi in linguaggio XSLT, che interpretano, traducendoli, documenti XML.

3.6.1 Interpretazione e linguaggio naturale

Capire cosa si sta ascoltando o leggendo significa saper interpretare correttamente quanto recepito.

Relativamente al linguaggio naturale, il processo di interpretazione può richiedere estrema sofisticatezza, dato che occorre individuare un significato del testo ascoltato, o letto, tra gli innumerevoli possibili.

In generale, infatti, il linguaggio naturale è ambiguo. Una stessa frase, a seconda del contesto in cui si trova, ha significati diversi. In casi estremi, una frase può essere capita solo contestualizzandola.

Esempio 13 (Frase inerentemente ambigua.) Si consideri la frase:

> *Quella persona mescola l'impasto col biscotto.*

Due possibili interpretazioni sono:

- l'impasto contiene il biscotto e non è dato sapere quale sia lo strumento con cui la persona mescola l'impasto;
- lo stumento, usato per mescolare, è il biscotto e l'impasto può contenere, o meno, biscotti.

In generale, solo interpreti sofisticati, quali le persone, possono risolvere, agevolmente, problemi ambigui.

Esercizio 36 (Frase inerentemente ambigua.)

1. Formulare una definizione di frase *inerentemente*, o *intrinsecamente*, ambigua, deducendola da quanto detto finora.
2. Formulare un esempio di frase inerentemente ambigua, che soddisfi la definizione, data al punto precedente.
3. Ideare un criterio per stabilire quando una frase sia inerentemente ambigua, in accordo con la definizione data.

Interpretazione di Problemi computabili

Uno degli aspetti caratterizzanti l'Informatica è la delineazione formale dell'insieme di problemi risolvibili attraverso procedure automatizzabili, ovvero interpretabili da interpreti poco sofisticati, quali i calcolatori elettronici, se rapportati alle potenzialità di una persona.

I problemi risolvibili da un calcolatore, per quanto complessi, debbono essere espressi in linguaggi strutturalmente molto semplici, se paragonati al linguaggio naturale.

I linguaggi per calcolatori devono poter esprimere *algoritmi*:

> Un *algoritmo* è una sequenza finita di passi, interpretabili ciascuno in un tempo finito.

I documenti XML possono definire algoritmi il cui scopo sia descrivere quale interpretazione dare a documenti XML che appartengono alla stessa semantica, definita tramite documenti DTD. L'interpretazione prende corpo mediante la trasformazione dei documenti XML iniziali in documenti che rispettano le regole di un qualche linguaggio alternativo, non necessariamente XML.

Esempio 14 (Valutazione di espressioni.) Sia dato il seguente testo, che rappresenta una semplice espressione tra numeri interi:

```
<!-- Testo: espressioni numeriche -->
<espressione>
  <somma>
     <espressione>
       <moltiplicazione>
          <espressione>
            <valore v="2"/>
          </espressione>
          <espressione>
            <valore v="3"/>
          </espressione>
       </moltiplicazione>
     </espressione>
     <espressione>
       <somma>
          <espressione>
            <valore v="2"/>
          </espressione>
          <espressione>
            <valore v="3"/>
          </espressione>
       </somma>
     </espressione>
  </somma>
</espressione>
```

la cui semantica sia data dal testo DTD:

```
<!ELEMENT espressione (somma | moltiplicazione | valore)>
<!ELEMENT somma (espressione, espressione)>
<!ELEMENT moltiplicazione (espressione, espressione)>
<!ELEMENT valore EMPTY >
 <!ATTLIST valore v CDATA #REQUIRED>
```

Il testo seguente descrive un algoritmo che, preso un *qualsiasi testo che soddisfi la semantica data*, lo traduce, interpretandolo, nel numero intero che costituisce il valore della espressione:

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

  <xsl:template match="espressione">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="somma">
   <xsl:variable name="sx">
     <xsl:apply-templates select="espressione[position()=1]"/>
   </xsl:variable>
   <xsl:variable name="dx">
     <xsl:apply-templates select="espressione[position()=2]"/>
   </xsl:variable>
   <xsl:value-of select="$sx + $dx"/>
  </xsl:template>

  <xsl:template match="moltiplicazione">
   <xsl:variable name="sx">
     <xsl:apply-templates select="espressione[position()=1]"/>
   </xsl:variable>
   <xsl:variable name="dx">
     <xsl:apply-templates select="espressione[position()=2]"/>
   </xsl:variable>
   <xsl:value-of select="$sx * $dx"/>
  </xsl:template>

  <xsl:template match="valore">
   <xsl:value-of select="@v"/>
  </xsl:template>

</xsl:stylesheet>
```

Esempio 15 (Estrazione di frasi.) Sia dato il seguente testo, con l'analisi logica di alcune frasi in italiano:

```
<!-- Testo: Analisi logica -->
 <AnalisiLogica>
     <Frase>
       <FraseSemplice>
    <Soggetto valore="La maestra"/>
    <Predicato valore="insegna"/>
       </FraseSemplice>
  </Frase>

     <Frase>
       <FraseSemplice>
    <Soggetto valore="La maestra"/>
    <Predicato valore="insegna"/>
       </FraseSemplice>
   <Complemento tipo="oggetto" valore="le tabelline">
    <Domanda valore="CheCosa"/>
   </Complemento>
  </Frase>

     <Frase>
       <FraseSemplice>
    <Soggetto valore="Il cane"/>
    <Predicato valore="abbaia"/>
       </FraseSemplice>
   <Complemento tipo="specificazione" valore="di Franca">
    <Domanda valore="DiChi"/>
    <Domanda valore="DiCheCosa"/>
   </Complemento>
  </Frase>
 </AnalisiLogica>
```

la cui semantica sia data dal testo DTD:

```
<!ELEMENT AnalisiLogica (Frase*)>
<!ELEMENT Frase (FraseSemplice | (FraseSemplice,Complemento))>
<!ELEMENT FraseSemplice (Soggetto, Predicato)>
<!ELEMENT Soggetto EMPTY>
  <!ATTLIST Soggetto valore CDATA #REQUIRED>
<!ELEMENT Predicato EMPTY>
  <!ATTLIST Predicato valore CDATA #REQUIRED>
<!ELEMENT Complemento (Domanda+)>
  <!ATTLIST Complemento tipo (oggetto | specificazione) #REQUIRED
                        valore CDATA #REQUIRED>
```

```
<!ELEMENT Domanda EMPTY>
  <!ATTLIST Domanda valore (CheCosa | DiChi | DiCheCosa) #REQUIRED>
```

Si ipotizzi che l'informazione relativa all'analisi logica non interessi, e, per un qualche motivo, sia necessario scrivere "in chiaro" le sole frasi in italiano che il testo contiene, come segue:

```
La maestra insegna
La maestra insegna le tabelline
Il cane di Franca abbaia
```

Il testo seguente descrive un algoritmo che, preso un *qualsisi testo che soddisfi la semantica data*, lo traduce, interpretandolo, nella lista di frasi volute:

```
<!-- Frasi in chiaro, estratte dal testo "Analisi logica" -->
<?xml version="1.0"?>
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="AnalisiLogica">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="Frase">
   <xsl:variable name="S">
     <xsl:apply-templates select="FraseSemplice/Soggetto/@valore"/>
   </xsl:variable>
   <xsl:variable name="P">
     <xsl:apply-templates select="FraseSemplice/Predicato/@valore"/>
   </xsl:variable>
   <xsl:variable name="C">
     <xsl:apply-templates select="Complemento/@valore"/>
   </xsl:variable>
   <xsl:variable name="Ct">
     <xsl:apply-templates select="Complemento/@tipo"/>
   </xsl:variable>

   <xsl:choose>
     <xsl:when test="$Ct ='oggetto'">
      <xsl:value-of select="$S"/>
      <xsl:value-of select="$P"/>
     <xsl:value-of select="$C"/><br/>
     </xsl:when>
     <xsl:when test="$Ct ='specificazione'">
      <xsl:value-of select="$S"/>
      <xsl:value-of select="$C"/>
```

```
      <xsl:value-of select="$P"/><br/>
     </xsl:when>
     <xsl:otherwise>
      <xsl:value-of select="$S"/>
      <xsl:value-of select="$P"/><br/>
     </xsl:otherwise>
   </xsl:choose>

  </xsl:template>
</xsl:stylesheet>
```

Commento 7
È necessario riflettere sulla *generalità* degli algoritmi definiti negli Esempi 14, 15.

Essi sono in grado di interpretare correttamente un qualsiasi documento che soddisfi la semantica data. Equivalentemente, ognuno degli algoritmi dati trasforma nel risultato voluto un qualsiasi documento che soddisfi le proprietà strutturali definite.

Obiettivi didattici

Uno degli obiettivi didattici è imparare a leggere e scrivere algoritmi d'interpretazione di concetti espressi tramite una grammatica XML.

Capitolo 4

Sintassi e Formalizzazione

- **Impareremo a *marcare* testi.** "Marcare un testo" significa evidenziare il significato delle componenti che costituiscono un concetto espresso tramite il testo stesso.

  Esempio 16 Il seguente testo evidenzia la classificazione delle parti componenti una semplice frase, in accordo con l'analisi logica:

  ```
  <articolo>La</articolo>
  <soggetto>mamma</soggetto>
  <predicato>mangia</predicato>
  ```

- **Impareremo a verificare la correttezza sintattica della marcatura.** Indipendentemente dall'efficacia e dalla coerenza dell'etichettatura, rispetto agli obiettivi assunti, occorre che la sintassi con cui si esegue l'etichettatura stessa sia strutturalmente corretta.

  Esempio 17 La marcatura seguente non è sintatticamente corretta e ne consegue l'impossibilità di individuare le parti etichettate, perdendo il senso dell'etichettatura stessa.

  ```
  <articolo>La
  <soggetto><predicato>mamma</soggetto>
  </articolo>mangia</predicato>
  ```

4.1 Etichettare per formalizzare

Formalizzare, o, equivalentemente, *strutturare sintatticamente* un concetto significa:

- **individuare** regolarità nella descrizione del concetto;
- **fissare tali regolarità** in una struttura linguistica opportuna.

Esempio 18 (Numero intero.) L'affermazione: "Un **numero** (intero) è composto da una **sequenza** di **cifre**", suggerisce una strutturazione sintattica del concetto "numero".

"Cifra" è un concetto primitivo associabile ad un valore. "Sequenza", invece, è un concetto il cui significato è noto dall'uso che se ne fa.

Quindi, una possibile strutturazione sintattica ad etichette può essere:

```
<numero>
 <cifra>1</cifra>
 <cifra>2</cifra>
 <cifra>3</cifra>
</numero>
```

Esercizio 37 (Tagging di Numeri telefonici.) Etichettare/formalizzare un numero telefonico, evidenziando la natura delle sue componenti.

Suggerimento: (i) Osservare che un numero telefonico è almeno un numero in cui insiemi di cifre hanno finalità diverse; (ii) Individuare tutte le parti componenti un numero; (iii) Tentare di rispettare le regole, per ora intuitive, di una buona etichettatura.

4.1.1 L'etichettatura ha una struttura

Le *etichette*, o *tag*, devono essere usate secondo regole grammaticali ben precise, i cui *aspetti salienti* sono:

- I *tag* "viaggiano" in coppia: ad un *tag* di apertura ne deve corrispondere uno di chiusura. L'analogia più calzante suggerisce di vedere i *tag* come delle parentesi;
- Proprio come le parentesi, i *tag* possono essere annidati e composti in sequenza:

$$([\quad][\{\quad\}]).$$

Esercizio 38 (*Tag* e parentesi.) Riformulare in un linguaggio ad etichette annidamento e sequenzalizzazione di parentesi in modo da descrivere almeno l'esempio ([ ][{ }]) dato. Esiste un qualche potenziale vantaggio nell'usare etichette al posto delle parentesi usuali?

Esempio 19 ("Beef Parmesan with Garlic Angel Hair Pasta".) Una ricetta per cucinare capelli d'angelo conditi con carne, parmigiano ed aglio, è la seguente:

```
Beef Parmesan with Garlic Angel Hair Pasta

Ingredients

1 1/2 pounds beef cube steak
1 onion, sliced into thin rings
1 green bell pepper, sliced in rings
1 cup Italian seasoned bread crumbs
1/2 cup grated Parmesan cheese
2 tablespoons olive oil
1 (16 ounce) jar spaghetti sauce
1/2 cup shredded mozzarella cheese
12 ounces angel hair pasta
2 teaspoons minced garlic
1/4 cup butter

Directions

Preheat oven to 350 degrees F (175 degrees C).

Cut cube steak into serving size pieces. Coat meat with
the bread crumbs and parmesan cheese. Heat olive oil in
a large frying pan, and saute 1 teaspoon of the garlic
for 3 minutes. Quick fry (brown quickly on both sides)
meat. Place meat in a casserole baking dish, slightly
overlapping edges. Place onion rings and peppers on top
of meat, and pour marinara sauce over all.

Bake at 350 degrees F (175 degrees C) for 30 to 45
minutes, depending on the thickness of the meat. Sprinkle
mozzarella over meat and leave in the oven till bubbly.

Boil pasta al dente. Drain, and toss in butter and 1
teaspoon garlic. For a stronger garlic taste, season with
garlic powder. Top with grated parmesan and parsley for
color. Serve meat and sauce atop a mound of pasta!

HINT:
make the meat ahead of time, and refrigerate over night,
the acid in the tomato sauce will tenderize the meat even
more. If you do this, save the mozzarella till the last
```

```
minute.

Calories 1167
Protein 71g
Fat 52g
Carbohydrates 101g
```

Commento 8
Come ogni **ricetta**, quella appena proposta è, essenzialmente, un **algoritmo** per un **interprete** molto sofisticato: un **cuoco**. L'algoritmo elenca un certo insieme di informazioni:

- ingredienti, passi di preparazione, qualche commento, e magari qualche notizia sulle caratteristiche nutrizionali;
- ogni ingrediente semplice ha un nome, ed è possibile che sia necessario specificarne una quantità, in una certa unità di misura, a meno che la quantità ne sia priva.

Tagging di "Beef Parmesan with Garlic Angel Hair Pasta".

Una volta individuati i blocchi concettuali che la compongono, la struttura essenziale della ricetta è evidenziabile per mezzo di una opportuna etichettatura, o *tagging*:

```
<ricetta>
   <titolo>Beef Parmesan with Garlic AngelHair Pasta</titolo>
   <ingrediente nome="beef cube steak"
                quantita="1.5"
                unita="pound"/>
   <!-- La lista di ingredienti continua. -->

   <preparazione>
     <passo>
       Preheat oven to 350 degrees F (175 degrees C).
     </passo>

     <!--
       Lista di tre elementi che ripercorre i passi
       della esposizione informale della ricetta.
     -->

   </preparazione>

   <commento>
     Make the meat ahead of time, and refrigerate over night,
     the acid in the tomato sauce will tenderize the meat
```

```
    even more. If you do this, save the mozzarella till the
    last minute.
  </commento>

  <nutrizione calorie="1167"
              grassi="23"
              carboidrati="45"
              proteine="32"/>
</ricetta>
```

Commento 9
L'etichettatura precedente introduce alcune **novità**, rispetto a quanto detto finora sulla natura delle etichettature:

- I *tag* `ingrediente` e `nutrizione` sono corredati di *attributi*.

  Per esempio, gli attributi del primo sono `nome`, `quantita` e `unita`.

  Nel caso considerato, ogni attributo specifica valori di grandezze ad esso correlate.

- Usa *tag* con sintassi "anomala" che permettono di inserire commenti.

  In "`<!-- testo del commento -->`", l'etichetta di inizio commento è `<!--`, mentre `-->` ne indica la fine.

  L'etichettatura per l'inserimento dei commenti è da considerarsi *meta*. Ovvero la sua sintassi è imposta e non ha nulla a che vedere con l'etichettatura `<commento> ... </commento>` che individua l'uso di una porzione del testo da

- Sfrutta una *forma abbreviata di etichetta*, quando essa non deve contenerne altre. Il *tag*:

  ```
  <nutrizione calorie="1167"
              grassi="23"
              carboidrati="45"
              proteine="32"/>
  ```

  è equivalente a:

  ```
  <nutrizione calorie="1167"
              grassi="23"
              carboidrati="45"
              proteine="32">
  </nutrizione>
  ```

Esercizio 39 (Possibili ristrutturazioni della ricetta.) Si consideri l'etichettatura di "Beef Parmesan with Garlic Angel Hair Pasta", appena proposta.

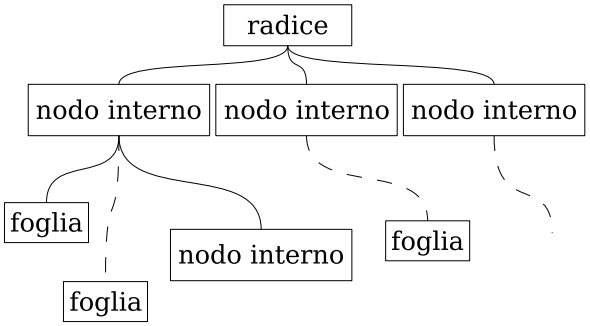
1. Suggerire una strutturazione alternativa del *tag* `nutrizione` ed evidenziarne vantaggi o svantaggi.

2. Suggerire una ristrutturazione del contenuto di un `passo`, con lo scopo di rendere accessibile la preparazione della ricetta ad una persona con qualche esperienza di cucina;

3. Suggerire una ristrutturazione del contenuto di un `passo` con lo scopo di rendere accessibile la preparazione della ricetta ad un interprete che richieda di essere guidato passo, dopo passo. Un interprete siffatto, ad esempio, potrebbe essere proprio un calcolatore elettronico.

4.1.2 Tagging ed alberi

Le proprietà di **annidamento** e **composizione** in sequenza dei *tag* inducono una strutturazione del risultato che può essere rappresentato come un *albero* (rovesciato):



- Ogni *nodo interno*, o *nodo elemento*, dell'albero:
  - assume il nome corrispondente ad una coppia di *tag* di apertura e chiusura;
  - possiede necessariamente dei discendenti.

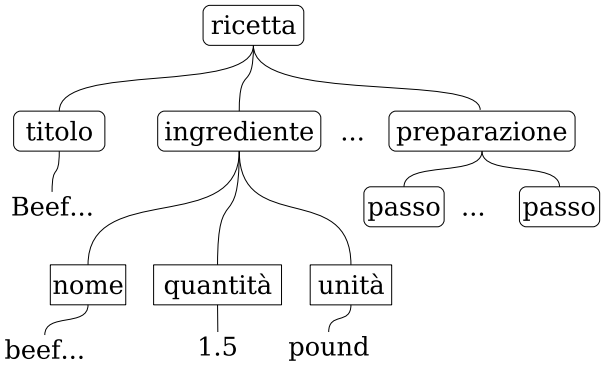
    Alcuni possono essere degli altri nodi interni.

    Altri possono essere nodi attributi.

- I nodi "foglia" terminano i "rami" dell'albero.

  Contengono i dati veri e propri sotto forma di sequenze di caratteri alfanumerici (*text strings*).

Esempio 20 Albero sintattico dell'etichettatura di "Beef Parmesan with Garlic Angel Hair Pasta":



L'albero proposto introduce una classificazione ulteriore tra nodi:

- I nodi smussati riassumono in un unico nome i *tag* di apertura e chiusura di un elemento, individuato nel testo;
- I nodi rettangolari, individuano il nome di attributo, associato ad un *tag* di apertura elemento, presente nel testo;
- I nodi "penzolanti", ovvero le foglie, sono i concetti primitivi, non più strutturabili e possono essere immaginati come il valore del nodo attributo o elemento cui sono collegati.

Esercizio 40 (Albero sintattico di un numero telefonico.) Preso un esempio di numero telefonico, fornitane l'etichettatura che evidenzi il significato delle sue parti, scrivere l'albero sintattico corrispondente.

Il seguente esercizio, per quanto, oggettivamente, possa risultare non banale, serve a delineare l'importanza dell'utilizzare notazioni diverse, per un medesimo concetto, col fine di evidenziarne aspetti diversi. Ovvero, l'esercizio serve per giustificare, nella maniera più evidente possibile, il motivo per cui può essere utile rappresentare i testi etichettati anche sotto forma di alberi sintattici.

Seguendo il suggerimento del testo, la soluzione all'Esercizio 41 deve passare per la rappresentazione ad albero di una frase in italiano, inizialmente scritta, come usuale, su di una riga di testo.

Proprio la rappresentazione alternativa ad albero permette la definizione di cosa sia una frase ambigua.

Esercizio 41 (Albero di frasi ambigue.) Sia data la frase:

La persona mescola l'impasto col biscotto.

"Dimostrarne" l'ambiguità intrinseca, sfruttando la nozione di albero sintattico.

Suggerimenti: (i) eseguire l'analisi logica della frase evidenziando la doppia valenza delle sue componenti; (ii) disegnare l'albero sintattico di tutte le "versioni" dell'etichettatura; (iii) confrontare gli alberi e rispondere alla domanda: "Esiste un criterio universale per dire che solo uno, tra quelli proposti è corretto?"

È importante notare che il cambio di rappresentazione è un'attività molto comune. Ad esempio, tutti hanno un'idea di cosa sia il concetto "amore", ma dell'amore si può parlare con le canzoni, per mezzo di un film o tramite una poesia. Pur trattando lo stesso concetto astratto, ogni mezzo espressivo menzionato può evidenziare aspetti differenti dell'amore, arricchendone la descrizione.

Con i dovuti distinguo, il cambio di rappresentazione della frase:

La persona mescola l'impasto col biscotto.

in albero sintattico ci permette di capire come si classificano le frasi ambigue da quelle non ambigue, permettendoci di saperne di più sulle proprietà delle frasi in italiano.

4.1.3 Metodologia essenziale di etichettatura

Dato un testo e fissato l'obiettivo da raggiungere per mezzo della formalizzazione ad etichette, *il processo di tagging* ***non porta necessariamente ad un risultato unico***.

In generale, tuttavia, vale la pena seguire l'idea che:

- i **tag** corrispodono a **sostantivi** del testo;
- i **tag** possono avere degli **attributi**, corrispondenti a **proprietà** del sostantivo cui ciascun *tag* si riferisce.

4.2 Formalizzazione, XML e parsificazione

"XML = etichettatura corretta + alcune istruzioni per il calcolatore"

è uno slogan ragionevolmente riassuntivo per definire cosa sia un documento XML completo.

L'etichettatura è quanto risulta dalla decorazione di un testo tramite *tag*.

Le **istruzioni** per il calcolatore sono ***tag* predefiniti**, ovvero fissati a priori, con cui corredare il testo etichettato affinché, un calcolatore "incaricato" di elaborarlo, possa individuarlo come testo XML, producendo i risultati attesi.

4.2.1 Sintassi di documenti XML

Ogni documento XML *essenziale*:

- inizia col *tag* `<?xml version="1.0"?>`;

  Esso indica la versione XML usata. Lo scopo è mantenere una corretta intepretazione del documento man mano che le versioni di XML verranno estese e modificate nel tempo.

- per ogni *tag* `<elemento>` di inizio descrizione il documento contiene il corrispondente *tag* `</elemento>` di fine descrizione, *ammesso che il valore di* `elemento` *possa essere non vuoto*.

  Questa richiesta implica che i nomi dei *tag* siano *case sensitive*, ovvero è importante fare attenzione all'uso delle lettere minuscole e maiuscole nella definizione dei *tag*: `<greeting>` ***non è*** il *tag* di apertura per `</Greeting>`, mentre lo è per `</greeting>`.

- per ogni *tag* di inizio descrizione di un `elemento` *necessariamente vuoto*, vale la pena, ma non è obbligatorio, usare la sintassi contratta `<elemento/>`, che ingloba *tag* di apertura e chiusura.

- è riformulabile come albero (grafo con un *singolo elemento radice* connesso ed aciclico), "fondendo" i *tag* di apertura `<elemento>` e chiusura `</elemento>` in un unico nodo `elemento` dell'albero stesso;

- può contenere meta-elementi commento `<!-- testo del commento -->`.

Esercizio 42 Trasformare in documenti XML il risultato di quelli prodotti fino a questo punto come soluzione degli esercizi assegnati.

4.2.2 Parsificazione

La correttezza sintattica di documenti XML può essere verificata automaticamente per mezzo di programmi opportuni. Tale verifica automatica passa sotto il nome di *parsificazione*.

Supponiamo di avere un documento `documento.xml`, scritto con un programma di elaborazione di testi che permetta di memorizzare un documento in formato testo puro. Ad esempio, per chi usi una qualche versione del sistema operativo `Windows`, `Blocco note` è l'elaboratore di testi da usare.

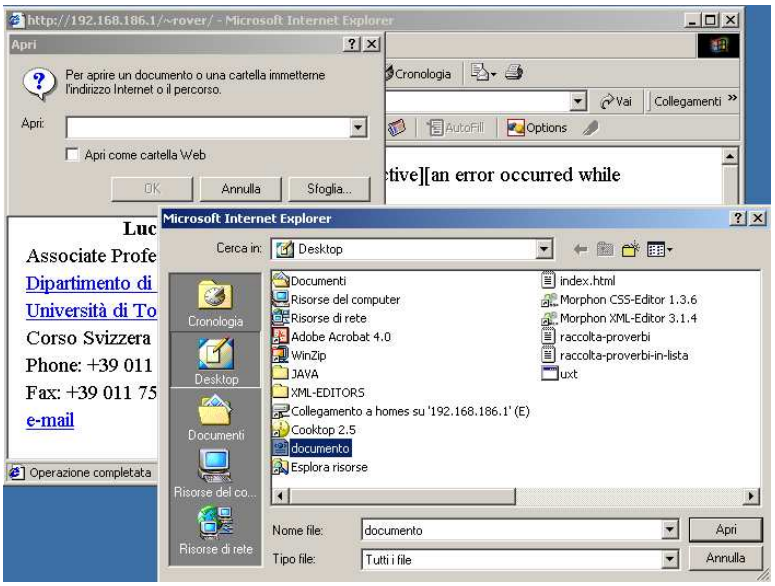
Il modo più semplice per parsificare `documento.xml` è leggere `documento.xml` in un *browser* internet sufficientemente recente come `Mozilla`, `Opera` o `Internet Explorer`.

Supponendo di usare una qualche versione del sistema operativo `Windows`, e che `documento.xml` sia memorizzato sul *Desktop*, leggere `documento.xml` in con `Internet Explorer` significa:

- eseguire un "click" sulla voce `Apri...` del *menù* `File`,

- far apparire tutti i file memorizzati sul *Desktop*,

- selezionare `documento.xml`:



I *browser* menzionati "capiscono" che `documento.xml` debba essere parsificato grazie al fatto che il suo nome ha estensione `.xml`.

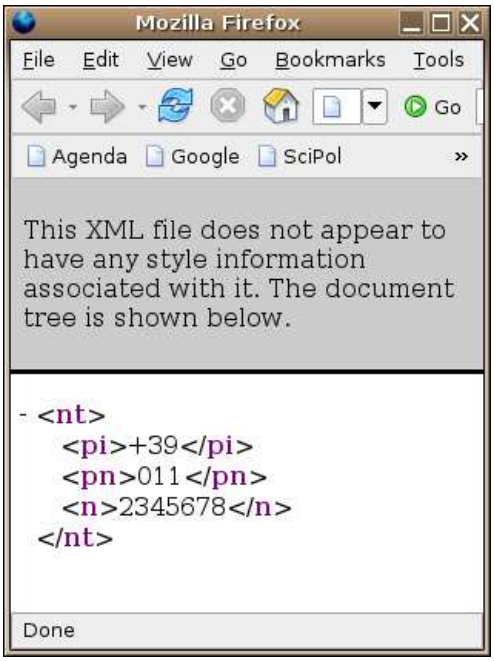
Se `documento.xml` non contiene alcun errore sintattico il *browser* parsifica correttamente il documento, che viene riprodotto sullo schermo, in accordo con regole gradevoli di rappresentazione delle sue varie parti.

Altrimenti, in caso di documento sintatticamente scorretto, il *browser* segnala l'errore, spesso con messaggi abbastanza criptici, che dovrebbero suggerire come correggere l'errore.

Esempio 21 (Risultati di parsificazione di testi XML.) Sia dato il seguente testo XML:

```
<?xml version="1.0"?>
<nt>
 <pi>+39</pi>
 <pn>011</pn>
 <n>2345678</n>
</nt>
```

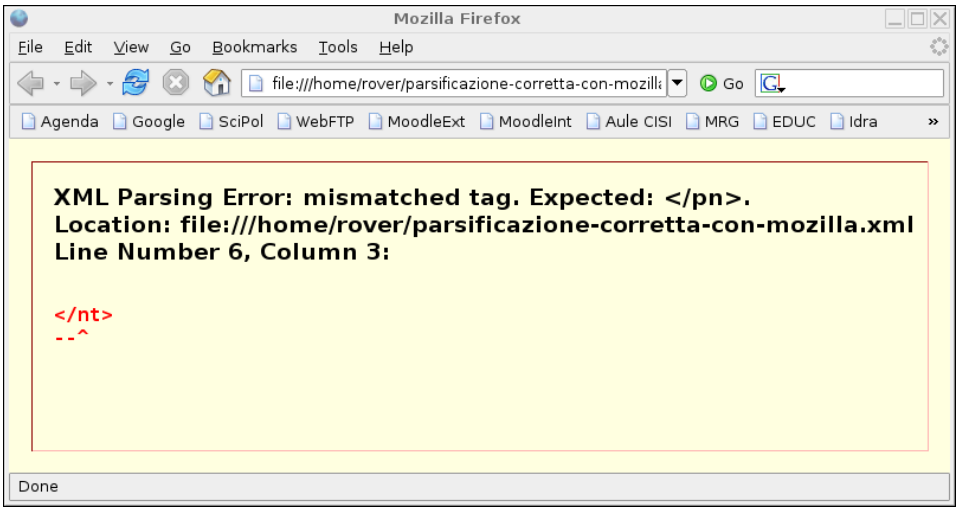
La sua parsificazione per mezzo del *browser* `Mozilla` produce il seguente risultato:



Introducendo un errore sintattico nel testo, come segue:

```
<?xml version="1.0"?>
<nt>
 <pi>+39</pi>
 <pn>011<pn>
 <n>2345678</n>
</nt>
```

il *browser* "protesta", indicando che manca un *tag* di chiusura:



Esercizio 43 (Parsificazione di documenti XML.) Usando uno dei *browser* menzionati, parsificare i documenti XML scritti sinora. Introdurre appositamente errori sintattici con lo scopo di imparare ad interpretare i messaggi d'errore segnalati dal *browser*.

Possibili "strategie" di generazione degli errori sintattici relativi sono:

- inserire *tag* non annidati correttamente;
- inserire *tag* con nome coincidente ma caratteri maiuscoli o minuscoli in posizioni diverse;
- inserire *tag* con caratteri speciali come "due punti", "cifre numeriche", "spazi", . . . ;
- inserire attributi nei *tag* ed omettere le "virgolette" che racchiudono i valori, oppure l'"uguale" tra nome dell'attributo e valore;
- sperimentare l'uso dello "spazio" nel nome di un attributo.

Infine, inserire qualche meta-commento per verificarne l'interpretazione da parte del *browser*.

4.3 Esercizi riassuntivi

Esercizio 44 (Etichettatura XML di concetti "reali".) Una volta considerati esempi significativi di documento per ciascuno dei seguenti concetti:

- curriculum vitae,
- ricetta medica,
- assegno bancario,

- agenda giornaliera.

compiere i seguenti passi:

1. Dichiarare uno scopo di utilizzo per ciascun documento considerato, se ritenuto necessario.
2. Etichettare ciascun documento, evidenziandone le parti ritenute essenziali per lo scopo prefissato.
3. Trovare un'etichettatura alternativa a quella ottenuta al punto precendente, ma sempre coerente con lo scopo prefissato.
4. Scrivere l'albero sintattico delle etichettature ottenute ai due punti precedenti.
5. Trasformare ogni documento prodotto in documento XML, aggiungendo, eventualmente dei (meta-)commenti.

4.4 Dove siamo

Quanto imparato in questo capitolo permette, idealmente, di cominciare a realizzare parte di un possibile progetto di gestione di *cv* generici, analogo a quello sviluppato in europass.cedefop.europa.eu. In particolare, grazie all'etichettatura XML, siamo in grado di definire *esempi* di documenti XML che rappresentino *cv*. Siamo anche consci del fatto che tale struttura non è necessariamente unica, nel senso che:

- non è detto esista un'unica via per strutturare, tramite etichette XML, dei testi, affinché essi rappresentino cv. Banalmente, anche solo il nome delle etichette non è unico e può dipendere dal progettista della formalizzazione di un *cv* per mezzo di un documento XML;
- i *cv* di diverse persone, con esperienze diverse, non hanno, necessariamente le stesse informazioni: c'è chi ha esperienze di lavoro all'estero, e chi no, oppure chi conosce lingue straniere, e chi conosce solo quella nativa. In generale, chi non possiede un certo tipo di esperienza non include nel proprio *cv* la voce corrispondente, per poi lasciarla priva di dati; semplicemente omette tale voce.

Inoltre, siamo in grado di far verificare ad un parsificatore che l'etichettatura di un documento rispetti i vincoli strutturali che lo rendono un testo XML: il primo passo necessario all'elaborazione automatica di testi XML.

Capitolo 5

DTD e semantica di documenti XML

- **Impareremo a definire semantiche**. La definizione di una semantica passa attraverso la definizione di insiemi che contengono *casi*, o *istanze*, o *esempi* differenti di un medesimo concetto.

  Le nostre istanze saranno documenti XML.

  > Gli insiemi di istanze sono definiti specificando proprietà strutturali, ovvero etichette legali, e la loro posizione relativa.

  Solo i testi XML che soddisfino proprietà date apparterranno all'insieme.

  Esempio 22 (Alcuni numeri naturali.) 23, 234, 2 ... sono esempi di numeri naturali la cui struttura è evidenziabile per mezzo di una etichettatura:

  ```
  <numero>              <numero>              <numero>
  <cifra>2</cifra>      <cifra>2</cifra>      <cifra>2</cifra>
  <cifra>3</cifra>      <cifra>3</cifra>      </numero>
  </numero>             <cifra>4</cifra>
                        </numero>
  ```

  Per definire la semantica di numeri naturali occorre osservare che l'*insieme* dei numeri naturali è formato da tutti i casi di numero accomunati dalle seguenti caratteristiche:

  1. un `numero` (intero) è una sequenza di almeno una `cifra`,
  2. ogni `cifra` è un concetto primitivo, denotata per mezzo di uno dei simboli 0, 1, 2, ...

- **Impareremo ad usare il linguaggio DTD per definire la semantica di documenti XML.**

  Semantica per mezzo del linguaggio naturale. In prima approssimazione, una semantica, ovvero un insieme di istanze del medesimo concetto, può essere espressa tramite frasi del linguaggio naturale, esattamente come abbiamo fatto ai punti 1 e 2 dell'Esempio 22 qui sopra.

  Semantica con un linguaggio non naturale per calcolatori. Dovendo definire semantiche che siano manipolabili da un calcolatore, occorre introdurre linguaggi adatti.

  Modello concettuale: alberi sintattici. Gli alberi sintattici, e, in particolare, ***i gradi di libertà*** che esistono per definirli, costituiranno il nostro primo passo verso una descrizione formale di semantica, interpretabile da un calcolatore.

DTD descrive alberi sintattici. Gli alberi sintattici saranno molto utili per descrivere una semantica, ma, alla fine, useremo effettivamente il linguaggio DTD (*Data Type Definition*).

Esso è uno strumento formale con cui descrivere come costruire alberi sintattici, ed è interpretabile da un calcolatore elettronico.

Interpretazione dei documenti DTD. Ogni documento DTD rappresenta un insieme di documenti XML. Ogni elemento dell'insieme è un'istanza, o esempio del concetto descritto dal documento DTD dato. Per questo diciamo che ogni documento DTD definisce la semantica di un concetto.

Esempio 23 (DTD per numeri interi.) Le seguenti clausole esemplificano l'uso del linguaggio DTD per definire la semantica dei numeri interi, visti come sequenze di cifre (Confrontare con l'Esempio 22):

```
<!ELEMENT numero (cifra+)>
<!ELEMENT cifra (#PCDATA)>
```

La prima clausola "afferma" che l'elemento `numero` contiene almeno un elemento `cifra`. La seconda, che l'elemento `cifra` è una foglia di un albero sintattico il cui valore è costituito da dati "qualsiasi".

DTD ed attributi dei *tag*. Il linguaggio DTD permette di formalizzare attributi e relative proprietà.

Esempio 24 (Cifre legali per numeri interi.) Nella sintassi DTD è possibile descrivere che i valori assumibili dal concetto `cifra` debbano essere limitati all'insieme $\{0, 1, \ldots, 9\}$ nel modo seguente:

```
<!ELEMENT cifra EMPTY>
<!ATTLIST cifra
          v (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
          #REQUIRED>
```

La prima clausola dice che l'elemento `cifra` ha struttura vuota; la seconda che `cifra` ha la proprietà `v` (valore) che può assumere uno dei valori `0 ... 9`. Quindi, `<cifra v="1"/>` e `<cifra v="7"/>` sono esempi *validi* di `cifra`, mentre `<cifra v="10"/>`, `<cifra v="a"/>` o `<cifra z="a"/>` sono controesempi.

5.1 Gradi di libertà nel costruire alberi sintattici

Ci avviciniamo alla definizione di una semantica, utilizzando gli alberi sintattici.

Ipotesi. Si supponga di avere un insieme X di testi XML che esprimano diversi casi di un medesimo concetto, come nell'Esempio 22 che descrive una struttura possibile dei numeri naturali.

Il modo più astratto, con l'informazione strutturale necessaria, per pensare a ciascuno dei testi in X è il relativo albero sintattico.

Caratteristiche strutturali. Per formalizzare la semantica che l'insieme X definisce, occorre:

Descrivere le caratteristiche strutturali comuni agli elementi di X.

La descrizione di tali caratteristiche strutturali comuni avviene ponendo dei limiti sui gradi di libertà nel poter definire la struttura di alberi sintattici.

5.1.1 I gradi di libertà

I gradi di libertà nel costruire un albero riguardano (almeno):

- *i nomi dei nodi*,
- *l'ampiezza di ogni livello* dell'albero,
- *la profondità massima* dell'intero albero,
- *la natura delle foglie*.

5.1.2 Analisi dei gradi di libertà

Supponiamo di voler definire la struttura comune agli elementi che appartengono ad un certo insieme X che esprime una semantica. Per realizzare questo obiettivo occorre definire quanto segue:

- **Nomi dei nodi degli alberi dei documenti XML in X.** I nomi devono appartenere all'insieme di etichette degli elementi che devono stare in X.

  Esempio 25 (Nomi dei nodi di alberi per numeri naturali.) L'insieme dei nomi dei nodi degli alberi per i numeri naturali nell'Esempio 22 è {`numero`, `cifra`}.

- **Lunghezza dei cammini di un albero di un elemento di X.**

  Un *albero descrive una gerarchia tra concetti* per mezzo della gerarchia tra i propri nodi.

  I discendenti di un nodo rappresentano i suoi *concetti costituenti*.

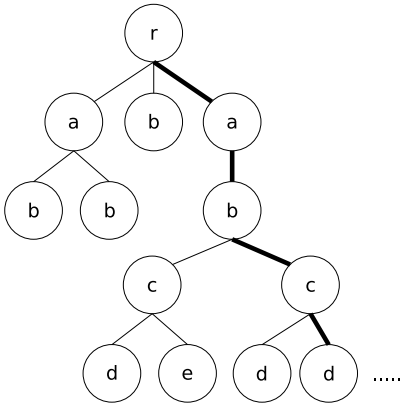
  Al crescere del dettaglio descrittivo corrisponde l'aumento della *lunghezza dei cammini*.

  La lunghezza di un cammino tra due nodi equivale alla loro distanza.

Definizione 1 (Distanza.) La distanza tra due nodi di un albero è il numero di archi, o rami, attraversati per andare da uno all'altro.

Esempio 26 (Distanza.) Nell'albero qui sotto due nodi di etichetta c sono i discendenti di un nodo b e, quindi, ne sono i concetti costituenti.

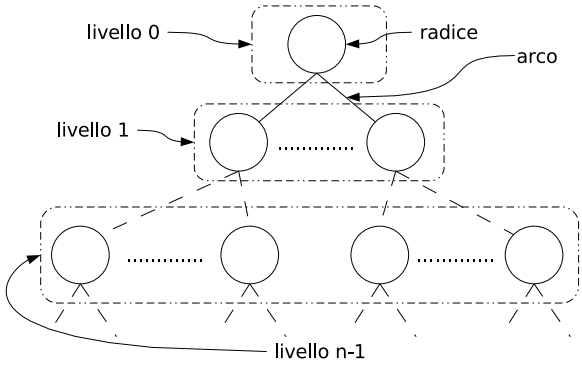
Lo stesso dicasi del nodo d, raggiunto dal cammino evidenziato in grassetto, a partire da r, rispetto al nodo c più a destra:



In particolare, i nodi r e d, connessi dal cammino evidenziato, distano 4 (archi).

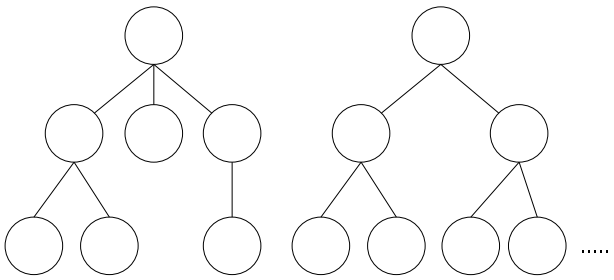
- **Ampiezza di ogni livello in un albero di un elemento di X.**

Definizione 2 (Livello.) Un livello è costituito da tutti i nodi con la stessa distanza dalla radice.



Definizione 3 (Ampiezza globale di un livello.) L'*ampiezza globale* di un livello è il numero globale di nodi a quel livello.

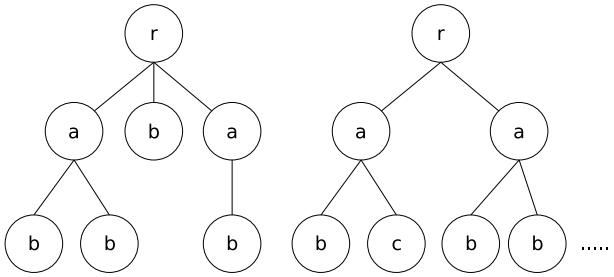
Esempio 27 (Ampiezza globale di un livello.) L'albero più a sinistra qui sotto ha ampiezza 1 a livello 0, 3 a livello 1 e 2, e 0 ad ogni livello maggiore di 2. L'altro ha ampiezza 1 a livello 0, 2 a livello 1, illimitata, ma finita, a livello 3 e 0 ad ogni livello maggiore di 2:



Definizione 4 (Ampiezza parziale.) L'*ampiezza parziale* è il numero di nodi con la stessa etichetta, ad uno stesso livello.

L'ampiezza globale può essere vista come la somma delle *ampiezze parziali*.

Esempio 28 (Ampiezze parziali di un livello.) L'albero più a sinistra qui sotto ha ampiezza parziale: (i) 1 a livello 0; (ii) 2 a livello 1, relativamente al nodo (con etichetta) a, e 1, relativamente al nodo b; (iii) 3 a livello 2, relativamente al nodo b; (iv) 0 ad ogni livello maggiore di 2.



L'altro ha ampiezza parziale: (i) 1 a livello 0; (ii) 2 a livello 1; (iii) 1 a livello 2, relativamente al nodo c ed illimitata, ma finita, relativamente al nodo b; (iv) 0 ad ogni livello maggiore di 2.

- **Natura delle foglie di un albero di un elemento di X.** Le foglie rappresentano i concetti primitivi, ovvero i nodi che non hanno ulteriori discendenti.

Esercizio 45 (Costruzione di alberi.) Definire almeno un albero sintattico per ciascuno dei seguenti casi:

1. La lunghezza massima di ogni cammino è 0.

2. L'ampiezza globale del livello 1 è 1, mentre quella del livello 2 è 5.

3. L'ampiezza globale del livello 2 è 5, somma di ampiezze parziali 3 e 2.

Una volta risolto, è utile riflettere sul significato di questo esercizio. Esso elenca tre *specifiche informali*, ovvero contiene tre frasi in italiano, che impongono quali siano le proprietà di alberi che riteniamo interessanti, per un qualche motivo. Tranne la prima specifica, le altre due individuano, ciascuna, un insieme infinito di alberi; tanti, infatti, sono gli alberi che soddisfano le richieste espresse. Dovrebbe, quindi, risultare naturale immaginare che il significato di un qualsiasi concetto noi esprimiamo per mezzo del linguaggio naturale, in generale, è rappresentato proprio da innumerevoli esempi del concetto espresso, e non da una singola istanza.

5.1.3 Alberi come documenti DTD

Un linguaggio astratto, non grafico, analogo all'XML, ma diverso da esso, introdotto specificamente per definire alberi sintattici passa sotto il nome di DTD.

DTD è acronimo di *Data Type Definition*.

Il linguaggio DTD permette di descrivere i gradi di libertà di costruzione degli alberi sintattici.

Dalla semantica "naturale" a quella formale. Il nostro obiettivo è formalizzare una semantica.

Inizialmente utilizzeremo il linguaggio naturale per ragioni evidenti di praticità nel descrivere le caratteristiche dell'insieme di documenti XML di cui definire la semantica.

Il passo successivo sarà la traduzione della descrizione in linguaggio naturale in linguaggio DTD.

5.2 Sintassi DTD per definire alberi sintattici

Vediamo come usare concretamente la sintassi DTD per esprimere i gradi di libertà possibili nel definire alberi sintattici di documenti XML.

Obiettivo. Quello che impareremo può essere utilizzzato per comporre testi effettivi, che, ad un certo punto, useremo per *verificare automaticamente*, attraverso il calcolatore, se un documento XML appartiene all'insieme che ci interessa definire, ovvero se appartiene ad una semantica intesa.

5.2.1 DTD per alberi di profondità "zero"

L'albero con la struttura più semplice contiene un unico nodo radice che coincide con l'unica foglia dell'albero stesso.

Pur fissando la profondità ad un unico nodo, rimangono alcuni gradi di libertà per definirne la struttura.

DTD per definire una foglia vuota

Per motivi dipendenti dalle necessità di progettazione, sull'unico nodo di un albero sintattico può essere necessario imporre il vincolo strutturale più limitante:

> *Il nodo non può contenere alcun valore.*

La clausola DTD corrispondente, che andrà inserita nel documento DTD che definisce la semantica che ci interessa, è la seguente:

```
<!ELEMENT nome-elemento EMPTY>
```

Commento 10

- `nome-elemento` sarà scelto nell'insieme di etichette che compongono i documenti XML che dovranno appartenere alla semantica in via di definizione.

Esempio 29 (Approssimazione minima di numero naturale.) Parlare dell'esistenza del concetto `numero`, senza poter dire alcunché sulla sua struttura, significa dover usare una delle etichettature XML seguenti:

```
<numero/>  <!-- oppure -->   <numero></numero>
```

Esse dicono dell'esistenza di un concetto `numero`, ma nulla di più.

La sintassi DTD corrispondente diventa: `<!ELEMENT numero EMPTY>`.

DTD per definire una foglia arbitraria

Sempre per motivi dipendenti dalle necessità di progettazione, supponendo che l'albero di cui occorre definire le proprietà debba avere un singolo nodo, rimane il caso in cui la struttura dell'unico nodo dell'albero in oggetto non può sottostare ad alcun vincolo strutturale, avendo già escluso il caso "foglia vuota", trattato al punto precedente.

Due varianti. Per questa situazione, riassumibile con lo *slogan*: "unico nodo, no vincoli", esistono due varianti, formalizzate dalle seguenti clausole DTD:

- `<!ELEMENT nome-elemento ANY> <!-- Assenza di struttura -->`.

  `nome-elemento` sarà scelto nell'etichettatura di cui si vuole definire la semantica.

  `ANY` indica che non si assume a priori alcuna strutturazione del contenuto di `nome-elemento`: il valore del nodo `nome-elemento` è una *qualsiasi sequenza di caratteri*.

- `<!ELEMENT nome-elemento (#PCDATA)> <!-- Struttura ovvia -->`.

  `#PCDATA` è una abbreviazione di *Parsed Character Data*. `#PCDATA` indica che, come nel caso precedente, il contenuto di `nome-elemento` possa essere una *qualsiasi sequenza di caratteri*.

La differenza tra `ANY` e `#PCDATA` è metodologica.

- La struttura `ANY` si sceglie in fase di progettazione nel momento un cui si demandi ad un successivo momento la descrizione della struttura di `nome-elemento`.
- La struttura `#PCDATA`, invece, costituisce un momento finale di progettazione, in cui si decide che il contenuto di `nome-elemento` debba essere proprio una sequenza arbitraria di caratteri.

Esempio 30 (Approssimazioni di numeri naturali.) Etichettature con un singolo nodo che costituiscono una approssimazione, molto lasca, della definizione dell'insieme di numeri naturali, possono essere:

```
<numero/>

<numero></numero>

<numero>34</numero>

<numero>2dkj345</numero>

<numero>asqdeqf   dfwqe</numero>
```

Le sintassi DTD corrispondenti sono:

```
<!ELEMENT numero ANY> <!-- e --> <!ELEMENT numero (#PCDATA)>
```

5.2.2 DTD per alberi con livelli ad ampiezza arbitraria

Strutturazione gerarchica. La descrizione di concetti per mezzo di documenti XML avviene tramite la *strutturazione gerarchica* delle etichette di ciascun documento: dalle etichette strutturalmente più complesse, ovvero da quelle che contengono altre etichette, a quelle più semplici.

Gerarchie come annidamenti. Le gerarchie tra etichette si costruiscono per *annidamento reciproco* delle etichette stesse, determinando così *ampiezza* e *profondità* livello per livello.

Obiettivo. Vedremo come definire quanto ampio possa essere un dato albero sintattico ad un dato livello.

La sintassi DTD, infatti, permette di definire gerarchie la cui ampiezza, livello per livello, può essere fissa o variabile.

DTD per alberi con livelli ad ampiezza fissa

Per ogni livello di un albero sintattico è possibile stabilire un numero fisso di discendenti tramite la seguente **clausola DTD**:

```
<!ELEMENT LivI (LivIpiu1-1, LivIpiu1-2, ... , LivIpiu1-n)>
```

definisce l'elemento (etichetta, nodo, *tag*, concetto) `LivI` al livello `I` di un albero sintattico.

Il nodo `LivI` ha `n` discendenti diretti

```
LivIpiu1-1, LivIpiu1-2, ..., LivIpiu1-n
```

al livello `I + 1`.

Il nome degli elementi discendenti contiene la particella `Ipiu1` del livello in cui si trova l'elemento padre.

La particella non è scritta esplicitamente nella forma `I + 1` perché la sintassi XML non accetta la presenza del carattere `+` come parte della definizione di un'etichetta.

Esempio 31 (Approssimazioni di numeri naturali.) Le seguenti clausole:

```
<!ELEMENT numero (cifra,cifra)>
<!ELEMENT cifra (#PCDATA)>
```

descrivono numeri naturali composti da, *esattamente*, due *etichette* `cifra`; nulla di più.

Due istanze di etichettature per `numero` possono essere:

```
<numero>                <numero>
  <cifra>1</cifra>         <cifra>13</cifra>
  <cifra>2</cifra>         <cifra>2aa b</cifra>
</numero>               </numero>
```

Commento 11
La formalizzazione di `numero` ha due difetti:

- non è sufficientemente generale. Esistono numeri naturali con un numero arbitrario di cifre ed essa non permette di descriverne la struttura al livello di dettaglio scelto;
- non è sufficientemente vincolante. Le cifre, in realtà, possono contenere sequenze arbitarie di caratteri.

Esercizio 46 (Numeri con sole tre cifre.) Partendo dall'Esempio 31, definire una semantica per il concetto di `numero` tale che:

- `numero` contenga *solo* tre cifre.

Non è necessario conoscere quali siano le cifre che compongono `numero`.

Tuttavia, un qualsiasi esempio di `numero` non deve poter contenere altro al di fuori delle tre cifre.

Suggerimento. L'esempio di `numero` di cui definire la semantica è:

```
<numero>
   <cifra/>
   <cifra/>
   <cifra/>
</numero>
```

DTD per alberi con livelli ad ampiezza variabile

La definizione di una semantica, in generale, richiede flessibilità nel definire l'ampiezza di un certo livello di un albero sintattico.

Seguono i diversi casi di clausole DTD per definire livelli di ampiezza variabile in un albero sintattico.

- `<!ELEMENT LivI (LivIpiu1*)>` definisce l'elemento `LivI` ad un livello dato, ad esempio `I`, di un albero sintattico.

  Esso può avere un *numero arbitrario*, ma finito, di discendenti diretti `LivIpiu1` al livello `I + 1`.

  Esercizio 47 (Numeri con sole cifre, in numero arbitrario.) Definire una semantica per il concetto di `numero naturale` tale che:

  – `numero` contenga una quantità arbitraria di cifre.

Non è necessario conoscere quali siano le cifre che compongono `numero`.

Tuttavia, un qualsiasi esempio di `numero` non deve poter contenere altro al di fuori di elementi `cifra`.

- `<!ELEMENT LivI (LivIpiu1?)>` definisce l'elemento `LivI` ad un livello dato, ad esempio `I`, di un albero sintattico.

  Esso può avere *al più* un discendente diretto `LivIpiu1` al livello `I + 1`.

  Esercizio 48 (Numeri arbitrari con sole cifre e segno.) Definire una semantica per il concetto di `numero` tale che:

  - `numero` contenga una quantità arbitraria di cifre,
  - `numero` contenga *al più* un segno.

  Non è necessario conoscere né quali siano le cifre, né quale sia il segno che compongono `numero`.

  Tuttavia, un qualsiasi esempio di `numero` non deve poter contenere altro al di fuori di elementi `cifra` e `segno`.

- `<!ELEMENT LivI (LivIpiu1+)>` definisce l'elemento `LivI` ad un livello dato, ad esempio `I`, di un albero sintattico.

  Esso può avere un *numero arbitrario*, ma finito e *non vuoto*, di discendenti diretti `LivIpiu1` al livello `I + 1`.

  Esercizio 49 (Numeri con cifre e segno.) Definire una semantica per il concetto di `numero` tale che:

  - `numero` contenga una quantità arbitraria, *non nulla*, di cifre,
  - `numero` contenga al più un segno.

  Non è necessario conoscere né quali siano le cifre, né quale sia il segno che compongono `numero`.

  Tuttavia, un qualsiasi esempio di `numero` non deve poter contenere altro al di fuori di elementi `cifra` e `segno`.

DTD per alberi i cui livelli abbiano elementi in alternativa

Per ragioni progettuali, è possibile che i discendenti di un nodo di un albero sintattico debbano poter essere scelti tra varie *alternative*, esprimibili con la seguente **clausola DTD**:

```
<!ELEMENT LivI (LivIpiu1-1 | LivIpiu1-2 | ... | LivIpiu1-n)>
```

Essa descrive l'esistenza dell'etichetta `LivI` ad un livello dato, ad esempio `I`, di un albero sintattico. `LivI` può avere *un discendente diretto*, scelto nell'insieme:

$$\{\texttt{LivIpiu1-1}, \ldots, \texttt{LivIpiu1-n}\} \ .$$

Esercizio 50 (Numeri con cifre e segno specificati.) Definire una semantica per il concetto di `numero` tale che:

- `numero` contenga una quantità arbitraria, *non nulla*, di cifre,
- `numero` contenga al più un segno.

Si vogliono conoscere le cifre ed il segno componenti ogni esempio di `numero`.

Tuttavia, un qualsiasi esempio di `numero` non deve poter contenere altro al di fuori di cifre e segno.

Suggerimento. Sfruttare la possibilità di definire elementi vuoti, uno in alternativa all'altro.

5.2.3 DTD per alberi a profondità arbitraria

La profondità di un albero cresce "annidando" le clausole DTD che definiscono l'ampiezza dei vari livelli.

Esempio 32 (Annidamento di clausole DTD.) Un documento DTD può contenere le clausole:

```
<!ELEMENT r (a|(a,b)*)|(c*|d?)>
<!ELEMENT b (a)>
<!ELEMENT c (d)>
<!ELEMENT a #PCDATA>
<!ELEMENT d #PCDATA>
```

`r` ha discendenti diretti con strutture diverse. Una prima classe di tali discendenti diretti, a sua volta, non ha stuttura unica, ma è una tra elementi semplici `a` o una sequenza, anche vuota, di coppie di nodi il cui primo elemento è `a` ed il secondo `b`. Anche la seconda classe di discendenti diretti di `r` offre un'alternativa: o una sequenza, anche vuota, di elementi `c`, o al più un elemento `d`.

Per annidare regole DTD in modo da comporre alberi sintattici di ampiezza e profondità arbitraria occorre usare la **clausola DTD generale**:

```
<!ELEMENT element-name content-model>
```

in cui `element-name` ha il significato ovvio, mentre `content-model` deve essere composto seguendo una opportuna sintassi, delineata dai seguenti punti:

- `content-model` (come già visto) può essere pari a `EMPTY`. In tal caso `element-name` è una foglia senza alcun contenuto.
- `content-model` (come già visto) può essere pari a `ANY`. In questo caso `element-name` è una foglia e può contenere sequenze arbitrarie di caratteri.
- `content-model` (come già visto) può essere pari a `#PCDATA`. Anche in questo caso `element-name` è una foglia e può contenere sequenze arbitrarie di caratteri.

- `content-model` può essere pari ad una *espressione regolare deterministica* (ERD). Una ERD è generata utilizzando le seguenti regole:

`E --> element-name`	caso base
`E --> (E,...,E)`	sequenza di lunghezza fissa
`E --> E*`	sequenza di lunghezza arbitraria
`E --> E+`	sequenza di lunghezza arbitraria, non vuota
`E --> E?`	sequenza di lunghezza, al più unitaria
`E --> (E|...|E)`	alternativa

L'idea è che ogni regola indica come generare la struttura di un livello di albero sintattico.

Esempio 33 (Generazione, uso e "lettura" di ERD.) – La regola `E --> LivIp1` genera l'espressione `LivIp1` che può essere usata nella clausola:

```
<!ELEMENT LivI LivIp1>
```

Il significato è che `LivI` ha un unico discendente `LivIp1`.

– La sequenza di regole:

```
E --> (E,E)
  --> (LivIp11,E)
  --> (LivIp11,LivIp12)
```

genera l'espressione `(LivIp11,LivIp12)` che può essere usata nella clausola:

```
<!ELEMENT LivI (LivIp11,LivIp12)>
```

Il significato è che `LivI` ha due discendenti `LivIp11` e `LivIp12`, da usarsi nell'ordine indicato.

– La sequenza di regole:

```
E --> (E,E)
  --> (LivIp11,E)
  --> (LivIp11,E*)
  --> (LivIp11,LivIp12*)
```

genera l'espressione `(LivIp11,LivIp12*)`.

Essa può essere usata nella clausola:

```
<!ELEMENT LivI (LivIp11,LivIp12*)>
```

Il significato è che `LivI` ha un numero illimitato, ma finito, di discendenti che vanno elencati nell'ordine indicato: il discendente più a sinistra nell'albero sintattico con radice `LivI` è `LivIp11`, seguito, "verso destra", da un numero arbitrario di occorrenze di `LivIp12`.

– La sequenza di regole:

```
E --> E*
  --> (E,E)*
  --> (E,LivIp12)*
  --> (LivIp11,LivIp12)*
```

genera l'espressione `(LivIp11,LivIp12)*`.

Essa può essere usata nella clausola:

```
<!ELEMENT LivI (LivIp11,LivIp12)*>
```

Il significato è che `LivI` ha un numero illimitato, ma finito, di discendenti, costituiti da coppie di elementi `LivIp11` e `LivIp12`.

– La sequenza di regole:

```
E --> E?
  --> (E|E)?
  --> (E|LivIp12)?
  --> (LivIp11|LivIp12)?
```

genera l'espressione `(LivIp11|LivIp12)?`.

Essa può essere usata nella clausola:

```
<!ELEMENT LivI (LivIp11|LivIp12)?>
```

Il significato è che `LivI` ha al più un discendente scelto tra gli elementi `LivIp11` e `LivIp12`.

Esercizio 51 (Generazione di espressioni regolari elementari.) Scrivere almeno una sequenza di generazione delle seguenti ERD:

1. `(b, (c | d+), (e | f))`
2. `(b?, c*, (( d | e)+ | f ))`
3. `(b+, (c* | d?), e*, (f | g)?, (h+ | i*))`

Esercizio 52 (Sintesi di espressioni regolari elementari.) 1. Scrivere una ERD che formalizzi una sequenza composta da almeno una terna.

Ogni terna deve essere formata dagli elementi `a`, `b`, `c`.

In ciascuna terna, `c` deve necessariamente comparire, mentre `a` e `b` possono anche non comparire.

2. Scrivere una ERD che formalizzi una struttura ad albero costituita da un'alternativa tra due coppie.

La prima coppia può essere usata un numero arbitrario di volte all'interno dell'albero.

Al contrario, la seconda coppia deve essere usata almeno una volta.

Gli elementi componenti le due coppie sono distinti l'uno dall'altro.

- `content-model` può essere pari a

```
(#PCDATA|element-name-1|...|element-name-n)*
```

che rapresenta il caso *contenuto misto*, ovvero un'alternanza arbitraria di `#PCDATA` ed elementi:

```
element-name-1 ...  element-name-n
```

Esercizio 53 (Elemento a contenuto misto.) Formalizzare la semantica del seguente esempio di etichettatura:

```
<contenuto-misto>
  Prima parte del testo sequito da un elemento <el1/>
  a sua volta seguito da un altro elemento <el2/>
  a sua volta seguito dallo stesso elemento <el2/>
  a sua volta seguito da un'occorrenza dell'elemento
  iniziale <el1/>
  seguito da due occorrenze consecutive degli
  elementi sinora inframezzati: <el1/> <el2/>
</contenuto-misto>
```

Commento 12
Gli elementi base, costituenti le ERD, sono solo `element-name`.

Ovvero `#PCDATA` non può occorrere come componente di una ERD.

Per poter descrivere strutture in cui etichette e testo non strutturato, denotato da `#PCDATA`, siano arbitrariamente inframezzati è necessaria la clausola "contenuto misto", la quale ***deve cominciare con il tipo di elemento*** `#PCDATA` e ***deve "terminare" col simbolo*** "`*`".

Esercizio 54 ("Semplificazione" di semantica.) Data la seguente grammatica che definisce numeri interi, composti da sole cifre binarie:

```
<!ELEMENT piu   EMPTY>
<!ELEMENT meno  EMPTY>
<!ELEMENT segno (piu | meno)>
<!ELEMENT cifra0 EMPTY>
<!ELEMENT cifra1 EMPTY>
<!ELEMENT cifra (cifra0|cifra1)>
<!ELEMENT numero-intero (segno?, cifra+)>
```

ristrutturarla, eliminando l'uso degli elementi `cifra` e `segno`, pur continuando a descrivere lo stesso concetto.

Esercizio 55 (Sintesi di documenti XML.) Scrivere almeno un esempio ed un controesempio di etichettature XML relative a DTD che contengano una sola clausola tra quelle elencate nei tre punti seguenti:

1. `<!ELEMENT a (b, (c | d+), (e | f))>`

2. `<!ELEMENT a (b?, c*, (( d | e)+ | f ))>`

3. `<!ELEMENT a (b+, (c* | d?), e*, (f | g)?, (h+ | i*))>`

assieme clausole che definiscano ogni elemento `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i` di tipo `#PCDATA`.

Esercizio 56 (DTD per nomi di documenti elettronici.) Definire un documento DTD che specifichi un formato ragionevole da usarsi per assegnare nomi significativi a documenti elettronici i quali contengano esercizi svolti in date diverse, su argomenti diversi, da studenti di una classe.

Esercizio 57 (DTD per etichette XML.) Scrivere una grammatica DTD di nome `tag-foglia` che specifichi la struttura di *tag* che siano foglia di un documento XML, ovvero che siano definiti da una clausola DTD `<!ELEMENT tag-foglia EMPTY>`. Esempi di nodi che vogliamo rappresentare come documenti XML sono: `<tag-foglia/>`, `<tag-foglia a="v1"/>`, `<tag-foglia a="v1" b="v2"/>`, *etc.* Una possibile rappresentazione XML di `<tag-foglia a="v1"/>` è la seguente:

```
<tag-foglia>
 <minore/>
 <contenuto>
   <nome/>
   <attributo>
    <spazio/><nome/><uguale/><apici/><valore/><apici/>
   </attributo>
 </contenuto>
 <chiusura>
   <slash/><maggiore/>
 </chiusura>
</tag-foglia>
```

Esercizio 58 (Grammatica DTD per ERD.) Scrivere una grammatica DTD, di nome `erd`, che specifichi la struttura delle *espressioni regolari deterministiche* ERD, nel senso seguente:

- Per ogni ERD `E`, che possa contenere sia ERD generiche, sia nomi di elementi, deve esistere un documento XML E che appartenga alla semantica `erd`, e che rappresenti `E`.
- Per ogni documento XML E che appartenga alla semantica `erd` deve esistere una ERD che lo rappresenti.

Un esempio di documento XML E che appartenga alla semanrica cercata potrebbe essere:

```
<erd>
  <sequenza-limitata>
```

```
        <erd>
           <sequenza-arbitraria>
              <erd>
                <foglia>a</foglia>
              </erd>
           </sequenza-arbitraria>
        </erd>
        <erd>
           <sequenza-limitata>
              <erd>
                 <sequenza-arbitraria>
                       <erd>
                           <foglia>b</foglia>
                       </erd>
                 </sequenza-arbitraria>
              </erd>
              <erd>
                 <foglia>c</foglia>
              </erd>
           </sequenza-limitata>
        </erd>
     </sequenza-limitata>
  </erd>
```

5.3 Sintassi per attributi in un albero sintattico

Un documento DTD oltre alla definizione della *struttura degli elementi*, tramite le clausole:

```
<!ELEMENT e-nome e-modello-contenuto>
```

può contenere anche clausole:

```
<!ATTLIST e-nome a-nome a-tipo a-modello>
```

per la descrizione degli *attributi* di un dato elemento.

Ogni `<!ATTLIST e-nome a-nome a-tipo a-modello-di-uso>`:

- associa all'elemento `e-nome` un attributo `a-nome`;
- descrive il *tipo dell'attributo*, ovvero l'insieme di valori legali, assumibili dall'attributo stesso, per mezzo di quanto specificato tramite `a-tipo`;
- specifica la *regola di utilizzo* dei valori dell'attributo, per mezzo di quanto descritto da `a-modello-di-uso`.

5.3.1 Sintassi per `a-tipo`

La sintassi per specificare `a-tipo` prevede alcuni casi alternativi:

- essa permette di usare una tra le tre parole chiave `CDATA`, `ID`, `IDREF`;
- essa permette di ***enumerare*** *esplicitamente un insieme di valori legali alternativi.*

Commento 13
Il primo carattere del valore di un attributo può solo essere una lettera, una sottolineatura, o il simbolo "`:`".

5.3.2 Sintassi per `a-modello-di-uso`

La sintassi per specificare `a-modello-di-uso` prevede i quattro seguenti casi alternativi:

1. essa permette di usare una delle parole chiave tra `#REQUIRED` e `#IMPLIED`;
2. essa permette di usare una coppia in cui il primo elemento sia la parola chiave `#FIXED`, mentre il secondo è un valore arbitrario;
3. essa permette di usare un valore arbitrario senza farlo precedere da alcuna delle parole chiave precedenti.

5.3.3 Interpretazione delle definizioni di attributi

La sintassi per `a-tipo` e per `a-modello-di-uso` permette di ottenere diverse combinazioni il cui significato esamineremo qui di seguito, evidenziando che i valori utilizzabili per `a-modello-di-uso` in una data clausola:

```
<!ATTLIST e-nome a-nome a-tipo a-modello-di-uso>
```

sono funzione di quelli specificati per `a-tipo` della stessa clausola.

I casi possibili.

- `<!ATTLIST e-nome a-nome CDATA a-modello-di-uso>`. Il valore `CDATA` sta per *character data*.

  Esso indica che (essenzialmente) ogni sequenza di caratteri è un valore legale per l'attributo `a-nome`.

  `a-modello-di-uso` può assumere tutti i valori possibili, con la seguente interpretazione:

  – `<!ATTLIST e-nome a-nome CDATA #REQUIRED>` impone che `a-nome` debba necessariamente comparire nella definizione di `e-nome`.

    Esempio 34 `<r/>` non appartiene alla seguente semantica:

    ```
    <!ELEMENT r EMPTY>
    <!ATTLIST r a CDATA #REQUIRED>
    ```

    mentre `<r a="valore a piacere"/>` vi appartiene.

  – `<!ATTLIST e-nome a-nome CDATA #IMPLIED>` definisce `a-nome` come facoltativo.

    Equivalentemente, `a-nome` non deve necessariamente comparire nella definizione di `e-nome`.

    Se vi compare, ad `a-nome` deve essere assegnato un valore.

    Se non vi compare, non è dato sapere quale sia il valore assegnato a `a-nome`.

    Esempio 35 Sia `<r/>` che `<r a="valore a piacere"/>` appartengono alla seguente semantica:

    ```
    <!ELEMENT r EMPTY>
    <!ATTLIST r a CDATA #IMPLIED>
    ```

  – `<!ATTLIST e-nome a-nome CDATA #FIXED "valore arbitrario">` impone che il valore utilizzabile per `a-nome` sia solo, ed esattamente, `valore arbitrario`.

    Ovvero `a-nome` è una costante.

Esempio 36 `<r a="altro valore"/>` non appartiene alla seguente semantica:

```
<!ELEMENT r EMPTY>
<!ATTLIST r a CDATA #FIXED "valore arbitrario">
```

mentre sia `<r a="valore arbitrario"/>` che `<r/>` vi appartengono.

Commento 14
`<r/>` è nella semantica specificata perché l'omissione dell'attributo è automaticamente interpretata come se esso fosse esplicitamente indicato, dato che `a` non può avere altri valori, se non `valore arbitrario`.

– `<!ATTLIST e-nome a-nome CDATA "valore arbitrario">` *sembra equivalente a* `<!ATTLIST e-nome a-nome CDATA #IMPLIED>` .
In realtà, `"valore arbitrario"` è un *valore di default*. Ovvero, se `a-nome` non compare in una delle istanze di `e-nome`, allora `a-nome` è come se comparisse associato a `"valore arbitrario"`

Esempio 37 Sia `<r/>`, sia `<r a="altro valore"/>`,
sia `<r a="valore arbitrario"/>` appartengono alla seguente semantica:

```
<!ELEMENT r EMPTY>
<!ATTLIST r a CDATA "valore arbitrario">
```

Ovviamente `<r b="qualsiasi valore"/>` non vi appartiene.

- `<!ATTLIST e-nome a-nome (v-1|...|v-n) a-modello-di-uso>`. L'*enumerazione esplicita* di valori `(v-1|...|v-n)` elenca *tutti e soli* i valori assumibili dall'attributo `a-nome`.

  `a-nome` potrà assumere un solo valore alla volta tra quelli enumerati.

  `a-modello-di-uso` può assumere tutti i valori possibili con la seguente interpretazione:

  – `<!ATTLIST e-nome a-nome (v-1|...|v-n) #REQUIRED>` impone che `a-nome` debba necessariamente comparire nella definizione di `e-nome`.

  Esercizio 59 (Enumerazione esplicita di valori `REQUIRED`.) Elencare esempi e controesempi relativi alla seguente semantica:

  ```
  <!ELEMENT r EMPTY>
  <!ATTLIST r a (v1|v2) #REQUIRED>
  ```

  – `<!ATTLIST e-nome a-nome (v-1|...|v-n) #IMPLIED>` dice che `a-nome` non deve necessariamente comparire nella definizione di `e-nome`.

Se vi compare, `a-nome` deve assumere uno dei valori enumerati.

Se non compare, non è dato sapere quale sia il valore ad esso assegnato.

Esercizio 60 (Enumerazione esplicita di valori `IMPLIED`.) Elencare esempi e controesempi relativi alla seguente semantica:

```
<!ELEMENT r EMPTY>
<!ATTLIST r a (v1|v2) #IMPLIED>
```

– `<!ATTLIST e-nome a-nome (v-1|...|v-n) #FIXED "v">` *ha senso* solo se `v` coincide con uno dei valori enumerati esplicitamente.

In tal caso l'unico valore utilizzabile per `a-nome` è solo, ed esattamente, `v`.

Se ne conclude che varrebbe la pena usare la clauosla:

```
<!ATTLIST e-nome a-nome CDATA #FIXED "v">.
```

Esercizio 61 (Enumerazione esplicita di valori `FIXED`.) Elencare esempi e controesempi relativi alla seguente semantica:

```
<!ELEMENT r EMPTY>
<!ATTLIST r a (v1|v2) #FIXED "v1">
```

– `<!ATTLIST e-nome a-nome (v-1|...|v-n) "v">` ha senso solo se `v` coincide con uno tra quelli elencati.

`v` assume il significato di *valore di default*.

Ovvero, `v` è utilizzato come valore di `a-nome`, nel caso `a-nome` non sia inserito esplicitamente nella definizione di `e-nome`.

Esempio 38 `<r/>`, `<r a="v1"/>` e `<r a="v2"/>` appartengono alla seguente semantica:

```
<!ELEMENT r EMPTY>
<!-- v e' uno tra v1 e v2 -->
<!ATTLIST r a (v1|v2) "v">
```

se `v` coincide con `v1` o con `v2`. Inoltre, se, ad esempio, `v` coincide con `v1`, allora `<r/>` è equivalente a `<r a="v1"/>`.
Invece, `<r b="qualsiasi valore"/>` non vi appartiene.

- `<!ATTLIST e-nome a-nome ID a-modello-di-uso>`. La clausola definisce attributi identificatori, da cui l'abbreviazione `ID`. La clausola impone l'*unicità* di ciascun valore assunto da `a-nome` all'interno dell'intero documento XML in esame.

  `a-modello-di-uso` può assumere tutti i valori possibili, con la seguente interpretazione:

– `<!ATTLIST e-nome a-nome ID #REQUIRED>` impone che `a-nome` debba necessariamente comparire nella definizione di `e-nome`.

Esercizio 62 (Identificatore `REQUIRED`.) Elencare esempi e controesempi relativi alla seguente semantica:

```
<!ELEMENT r (e*)>
<!ELEMENT e EMPTY>
<!ATTLIST e a ID #REQUIRED>
```

– `<!ATTLIST e-nome a-nome ID #IMPLIED>` definisce `a-nome` come facoltativo.

Se nello stesso testo XML compare più di un elemento `e-nome` senza che `a-nome` sia stato specificato, i valori intesi sono tutti distinti tra loro, ma non è dato sapere quali siano.

Esempio 39 (Identificatore IMPLIED.) La seguente semantica:

```
<!ELEMENT r (e*)>
<!ELEMENT e EMPTY>
<!ATTLIST e a ID #IMPLIED>
```

ammette `<r><e/><e/></r>` e `<r><e a="id2"/><e/></r>` come esempi, mentre `<r><e a="id2"/><e a="id2"/></r>` è un controesempio.

– `<!ATTLIST e-nome a-nome ID #FIXED "v">` imporrebbe `v` come unico valore utilizzabile per `a-nome`.

Questo vincolo, unito alla richiesta di unicità di `v`, valido nell'intero testo XML, implicherebbe l'unicità anche di `e-nome`.

Considerata l'eccessiva ristrettezza della conseguenza, *non è ammesso usare identificatori con valori costante.*

– `<!ATTLIST e-nome a-nome ID "v">` come il precedente, non è un caso ammesso, considerata la ristrettezza del vincolo imposto.

- `<!ATTLIST e-nome a-nome IDREF a-modello-di-uso>`. La clausola definisce degli attributi usati per riferirsi a valori di identificatori. "`IDREF`" sta per *reference to identifier*. La clausola impone che il valore assunto da `a-nome` *coincida* con uno di quelli presenti nel documento XML ed associato ad un attributo di tipo `ID`.

  `a-modello-di-uso` può assumere tutti i valori possibili, con la seguente interpretazione:

  – `<!ATTLIST e-nome a-nome IDREF #REQUIRED>` impone che `a-nome` debba necessariamente comparire nella definizione di ogni esempio di `e-nome`.

Esercizio 63 (Riferimenti `REQUIRED` ad `ID`.) Elencare esempi e controesempi relativi alla seguente semantica:

```
<!ELEMENT r (e*,f*)>
<!ELEMENT e EMPTY>
<!ATTLIST e a ID #REQUIRED>
<!ELEMENT f EMPTY>
<!ATTLIST f l IDREF #REQUIRED>
```

– `<!ATTLIST e-nome a-nome IDREF #IMPLIED>` definisce `a-nome` come facoltativo.
Se nello stesso XML compare più di un elemento `e-nome` di tipo `ID` senza che per esso `a-nome` sia stato specificato, non è dato sapere quali siano i valori assunti dagli elemeneti `IDREF`.

Esercizio 64 (Riferimenti `IMPLIED` ad `ID`.) Elencare esempi e controesempi relativi alla seguente semantica:

```
<!ELEMENT r (e*,f*)>
<!ELEMENT e EMPTY>
<!ATTLIST e a ID #IMPLIED>
<!ELEMENT f EMPTY>
<!ATTLIST f l IDREF #IMPLIED>
```

– `<!ATTLIST e-nome a-nome IDREF #FIXED "v">` impone `v` come unico valore utilizzabile per `a-nome`.
Questo vincolo, impone l'esistenza di un'occorrenza di *tag* nel testo XML preso in esame il cui attributo sia di tipo `ID` ed abbia valore identico a `v`.

Esercizio 65 (Riferimenti `FIXED` ad `ID`.) Elencare esempi e controesempi relativi alla seguente semantica:

```
<!ELEMENT r (e*,f*)>
<!ELEMENT e EMPTY>
<!ATTLIST e a ID #IMPLIED>
<!ELEMENT f EMPTY>
<!ATTLIST f l IDREF #FIXED "d">
```

– `<!ATTLIST e-nome a-nome IDREF "v">` indica `v` come *valore di default*.
Ovvero, `v` è utilizzato come valore di `a-nome` nel caso `a-nome` non sia inserito esplicitamente nella definizione di un esempio di `e-nome`.
In tal caso, però, occorre che `v` sia il valore di un attributo di tipo `ID`.

Esercizio 66 (Riferimenti di *default* ad `ID`.) Elencare esempi e controesempi relativi alla seguente semantica:

```
<!ELEMENT r (e*,f*)>
<!ELEMENT e EMPTY>
<!ATTLIST e a ID #IMPLIED>
<!ELEMENT f EMPTY>
<!ATTLIST f l IDREF "d">
```

- `a-tipo` potrebbe assumere anche i valori `ENTITY`, `ENTITIES`, `NMTOKEN`, `NMTOKENS`, `NOTATION`. Gli scopi didattici non ne richiedono l'uso.

5.4 Esercizi Riassuntivi

Esercizio 67 (Relazione tra etichettaura e semantica.) Per ciascuno dei casi seguenti, correggere o il documento XML, o la grammatica DTD associatagli, nel caso il testo XML non appartenga alla semantica definita dal secondo.

Etichettatura	Semantica

`<es-id>`
` <a i="s1"/>`
` <a i="s2"/>`
` <a i="s1"/>`
`</es-id>`
	`<!ELEMENT es-id (a*)>`
	`<!ELEMENT a EMPTY>`
	`<!ATTLIST a i ID #REQUIRED>`

Etichettatura	Semantica

`<es-id>`
` <a i="s1"/>`
` <a i="s2"/>`
` <b i="s3"/>`
`</es-id>`
	`<!ELEMENT es-id (a*, b)>`
	`<!ELEMENT a EMPTY>`
	`<!ATTLIST a i ID #REQUIRED>`
	`<!ELEMENT b EMPTY>`
	`<!ATTLIST b i IDREF #REQUIRED>`

Etichettatura	Semantica

`<es-id>`
` <a i="s1"/>`
` <a i="s2"/>`
` <b i="s2"/>`
`</es-id>`
	`<!ELEMENT es-id (a* | b)>`
	`<!ELEMENT a EMPTY>`
	`<!ATTLIST a i ID #REQUIRED>`
	`<!ELEMENT b EMPTY>`
	`<!ATTLIST b i IDREF #REQUIRED>`

Esercizio 68 (Semantica del concetto "numero telefonico".) Definire una semantica di numero telefonico in cui inglobare numeri composti da un prefisso internazionale, da uno nazionale, e dal numero di utenza.

Il prefisso internazionale deve poter essere espresso secondo due formati. Il primo prevede che esso cominci con "00". Il secondo richiede che il prefisso internazionale cominci col simbolo "+". In entrambi i casi, si prosegue con almeno una cifra.

Il vincolo sul prefisso nazionale è che esso contenga almeno due cifre.

È necessario escludere dalla semantica qualsiasi altro concetto.

Ovvero, numeri telefonici validi sono: `001 04 1234567`, `+1 04 1234567`. Al contrario `01 04 1234567`, `+ 04 1234567` e `+ 04 123sf67` non devono appartenere alla semantica.

Esercizio 69 (Semantica del concetto "ricetta che evolve".) La formalizzazione di una semantica per il concetto "ricetta culinaria" può avere una strutturazione intuitiva standard, nel senso che chiunque si aspetterebbe almeno un titolo, un elenco di ingredienti, ed una lista di passi di preparazione.

Tuttavia, in generale, una buona ricetta migliora con l'aggiunta di specifici e personali accorgimenti come, ad esempio, l'aggiunta di un ingrediente accompagnata dall'aggiunta di un qualche passo di preparazione, inizialmente non esplicito.

Quindi, non è inusuale osservare delle note a margine del testo originale della ricetta.

È come dire che una ricetta può evolvere.

Scopo di questo esercizio è cercare una semantica per ricette adatte ad evolversi, come risultato di piccoli aggiustamenti, riguardanti nuovi passi intermedi, nuovi ingredienti e l'uso di strumenti non indicati nel testo originale.

La semantica dovrebbe essere orientata ad includere ricette in cui:

- ogni passo debba essere numerato univocamente;
- tale numerazione deve essere utile ad individuare utensili ed ingredienti, aggiunti nel tempo.

Esercizio 70 (Albero genealogico.) Pensando ad un albero genealogico, è usuale riferirsi implicitamente a quello che indica il susseguirsi delle generazioni di una data famiglia.

Tuttavia, è usuale tracciare l'albero genealogico anche riguardo a cani di razza.

Scrivere la semantica DTD del concetto di albero genealogico che possa essere usato indifferentemente in entrambi i casi.

Esercizio 71 (DTD per le tabelle XHTML.) In base alla descrizione fatta finora, definire la DTD che costituisca almeno una parte della semantica delle tabelle XHTML.

Esercizio 72 (DTD di porzioni XHTML.) L'obiettivo di questo esercizio è arrivare a scrivere documenti DTD che specifichino istanze di documenti XML ragionevolmente sofisticati.

Procedere per passi successivi, definendo DTD che descrivano insiemi di istanze XHTML di complessità crescente.

Eventualmente, prendere spunto dal seguente elenco di richieste:

1. Caratterizzare istanze XHTML che contengano solo paragrafi.
2. Caratterizzare istanze XHTML con sole liste ordinate i cui elementi `li` contengano elementi descritti al punto 1.

3. Caratterizzare istanze XHTML con sole tabelle i cui elementi `td` contengano elementi descritti al punto 1.

4. Caratterizzare istanze XHTML con sole tabelle i cui elementi `td` possano contenere elementi descritti ai punti 2 o 1.

5. Caratterizzare istanze XHTML in cui gli elementi descritti ai punti 1, 2 e 3 possano essere arbitrariamente inframezzati.

6. Caratterizzare istanze XHTML in cui paragrafi, liste e tabelle possano essere arbitrariamente inframezzati e annidati.

5.5 Dove siamo

Per un momento torniamo al servizio in europass.cedefop.europa.eu per la gestione di *cv*. Il messaggio del capitolo in via di conclusione a proposito di tale servizio è quanto sia necessario individuare precisamente l'insieme dei documenti XML che esso gestisce. Sappiamo distingure tra *cv* "legali", che potranno essere manipolati elettronicamente, e *cv* mal strutturati. La descrizione dei *cv* ben strutturati deve necessariamente essere formale, assumendo la forma, ad esempio, di documenti DTD. Senza DTD non si saprebbe di che insieme di documenti XML si stia parlando.

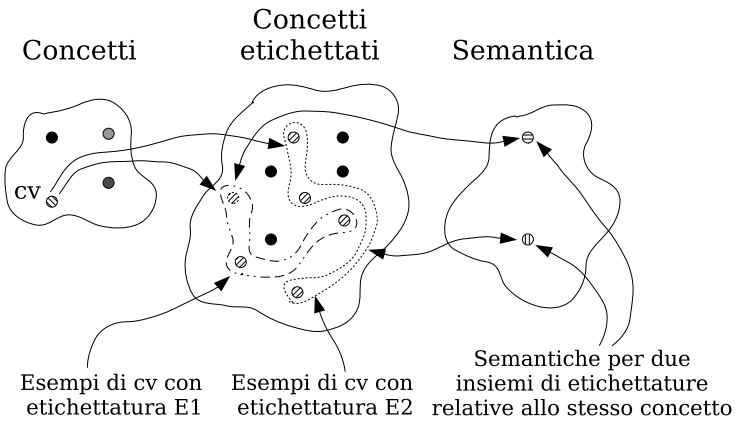


Figura 5.1: Concetti, etichettature, semantica.

La Figura 5.1 generalizza il contesto, che possiamo leggere come segue. In "natura" esiste un'infinità di concetti di cui parliamo per mezzo del linguaggio naturale.

Abbiamo visto che voler elaborare automaticamente aspetti relativi ad un dato concetto può richiedere una rappresentazione opportuna degli esempi testuali del concetto stesso: le etichettature XML.

Si osserva, tuttavia, che, pur a parità di obiettivi, volendo elaborare esempi dello stesso concetto, l'etichettatura può non essere univoca. Ad esempio, ciascuno è libero di immaginare etichettature differenti `E1` ed `E2` per lo stesso concetto *cv*, che originano insiemi di esempi disgiunti di *curricula*, come evidenziato dalla Figura 5.1.

Una semantica, espressa in termini di testo DTD, riassume le caratteristiche strutturali comuni di un insieme di documenti XML, che fanno capo alla medesima etichettatura. In Figura 5.1 questo fatto è rappresentato indicando l'esistenza di una semantica per ciascuna delle etichettature, ipotizzate per la descrizione di *cv*.

Esercizio 73 (riassuntivo sulla formalizzazione di semantiche.) Scrivere un un documento DTD che definisca una semantica per almeno uno dei seguenti concetti: (i) *curriculum vitae*, (ii) ricetta medica, (iii) assegno bancario, (iv) agenda impegni giornalieri, (v) questionario con domande che richiedano risposte in alternativa, (vi) pagina *Facebook* o *Twitter*, (vii) pagina di *Blog*, (viii) pagina di catalogo turistico, (ix) *brochure* esplicativa.

Infine possiamo anche sostenere di aver acquisito maggiore coscienza di quel che inconsciamente facciamo mentre parliamo o scriviamo. Scrivendo, usiamo regole analoghe a quelle fissate nei documenti DTD, ma, in realtà, molto più complesse in termini di utilizzo, per costruire frasi sintatticamente corrette, in accordo con la grammatica italiana. Quindi, compiamo azioni molto sofisticate. Lo studio e la scrittura di DTD materializza questo processo in un ambito in cui le frasi corrette che dobbiamo scrivere sono rivolte all'uso di un calcolatore.

Capitolo 6

Validazione di documenti XML

- **Impareremo a *validare* documenti XML, automaticamente, per mezzo di almeno un programma *validatore*.**

  Ovvero, vedremo come verificare se un esempio di documento XML, appartenga, o meno, all'insieme di documenti descritti da un documento DTD.

  A tal fine occorrono:

  – un documento DTD che descriva la struttura di tutti gli esempi di documenti XML che appartengono ad una stessa semantica, ovvero, all'insieme di documenti XML che interessa validare;

  – almeno un documento XML `sorgente.xml`. Esso potrà essere un esempio di documento che appartenga, o meno, all'insieme descritto dal documento DTD precedente. Se `sorgente.xml` appartiene effettivamente alla semantica data, si dice che `sorgente.xml` è *valido* per tale semantica. Altrimenti, `sorgente.xml` non è valido.

  È possibile verificare automaticamente la validità di documenti, rispetto ad una semantica, per mezzo di programmi validatori.

  Utilizzeremo l'elaboratore di testi XML XMLCopy. Esso semplifica la scrittura di documenti XML ed incorpora un validatore.

6.1 Preparare un documento XML per la validazione

Per validare un documento `sorgente.xml`, ovvero per verificare che appartenga ad una semantica descritta da un documento DTD, occorre specificare in `sorgente.xml` quale sia il documento DTD.

A tal fine occorre includere clausole opportune nel preambolo di `sorgente.xml`. Il preambolo di ogni documento XML è la zona di testo che segue immediatamente l'intestazione `<?xml version="1.0" ?>`.

DTD *inline*. È data come segue:

```
<?xml version="1.0" encoding="UTF-8"?>

  <!-- Preambolo XML con un testo DTD esplicito -->

  <!DOCTYPE elemento-radice [
     <!ELEMENT elemento-radice ....>
     <!-- resto della grammatica DTD -->
  ]>

<!-- resto del documento XML -->
```

Esempio 40 (DTD *inline*.) Il seguente documento XML contiene una DTD *inline*:

```
<?xml version="1.0"?>
<!DOCTYPE r [
 <!ELEMENT r (a,b,b)>
 <!ELEMENT b (a)>
 <!ELEMENT a (#PCDATA)>
]>
<r>
  <a></a>
  <b><a>Prima istanza</a></b>
  <b><a> Seconda istanza</a></b>
</r>
```

DTD esterna. È data come segue:

```
   <?xml version="1.0" encoding="UTF-8"?>
     <!-- Preambolo XML con un ruferimento ad un file
          esterno, che contiene il testo DTD          -->

    <!DOCTYPE numero SYSTEM "testo-DTD-esterno.dtd">

    <!-- resto del documento XML -->
```

Esempio 41 (DTD esterna.) Il seguente documento XML indica una DTD esterna:

```
<?xml version="1.0"?>
<!DOCTYPE r SYSTEM "external.dtd">
<r>
  <a></a>
  <b><a>Prima istanza</a></b>
  <b><a> Seconda istanza</a></b>
</r>
```

Nel *file* `external.dtd` possono comparire le clausole:

```
 <!ELEMENT r (a,b,b)>
 <!ELEMENT b (a)>
 <!ELEMENT a (#PCDATA)>
```

6.2 Documenti di riferimento

Come sorgente usiamo `raccolta-proverbi.xml` con una DTD *inline*, come segue:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE raccolta [
 <!ELEMENT raccolta (titolo,proverbio*)>
 <!ELEMENT titolo (#PCDATA)>
 <!ELEMENT proverbio (#PCDATA|evidenzia|agrave|egrave)*>
  <!ATTLIST proverbio origine CDATA #IMPLIED>
 <!ELEMENT evidenzia (#PCDATA)>
 <!ELEMENT agrave EMPTY>
 <!ELEMENT egrave EMPTY>
]>

<raccolta>
  <titolo>Proverbi</titolo>
  <proverbio origine="Cina">
  Chi sente <evidenzia>dimentica</evidenzia>,
  chi vede <evidenzia>ricorda</evidenzia>,
  chi fa <evidenzia>impara</evidenzia>.
 </proverbio>
  <proverbio origine="Norvegia">
  Se ognuno pulisce davanti a casa,
  tutta la citt<agrave/> <egrave/> pulita.
 </proverbio>
</raccolta>
```

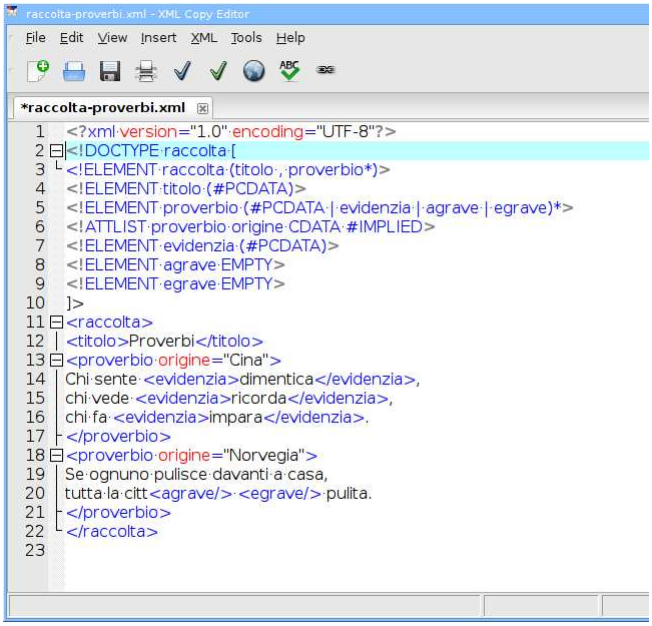
Il documento `raccolta-proverbi.xml`, oltre alla DTD, contiene una lista di proverbi, ciascuno associato alla nazione di origine.

Per riassumere in pochi passi l'uso di XMLCopy come validatore, supponiamo `raccolta-proverbi.xml` sia memorizzato sul *desktop*.

6.3 Validazione con XMLCopy

XMLCopy è un elaboratore testi che facilita la composizione di testi XML e li valida, a patto che essi indichino una DTD *inline* o esterna.

Dopo aver installato XMLCopy ed aver scritto il documento XML di riferimento nel *file* `raccolta-proverbi.xml`, leggiamo `raccolta-proverbi.xml` in XMLCopy:

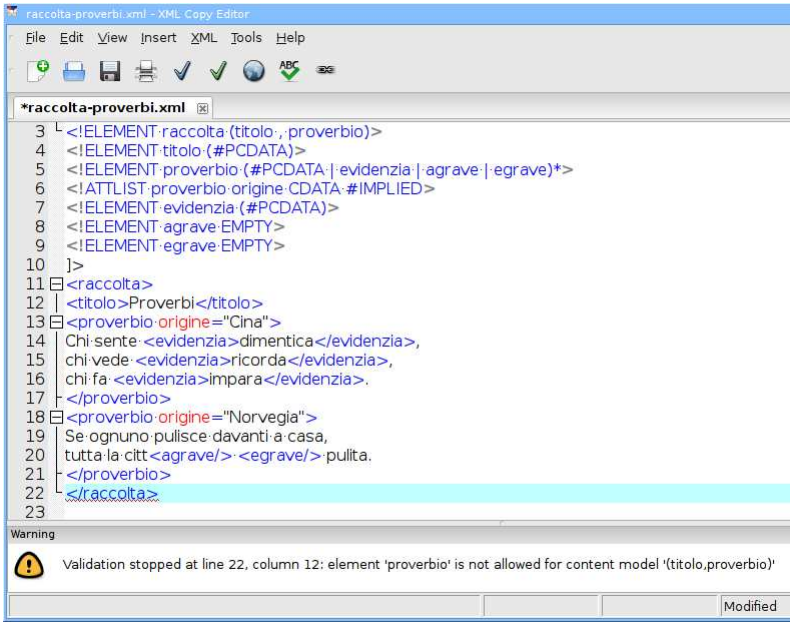


Parsificazione. Un *click* sul simbolo di spunta *blu*, verifica se il testo rispetta la sintassi XML. Una finestra con un messaggio opportuno compare sia che il testo XML sia sintatticamente corretto, sia in caso di errori.

Validazione. Un *click* sul simbolo di spunta *verde*, *valida*, o meno, il documento, rispetto alla DTD specificata.

Nel caso di `raccolta-proverbi.xml` non dovrebbe essere segnalato alcun errore, siccome esso è valido rispetto alla DTD *inline*.

Ad esempio, cancellando dalla DTD l'asterisco che segue l'elemento `proverbio`, XMLCopy reagisce indicando che il testo XML non è più valido:



Esercizio 74 (Validare con XMLCopy.) Generare appositamente degli errori di validazione, modificando `raccolta-proverbi.xml` nella sua parte XML o DTD.

Esercizio 75 (Semantiche di "confine".) Alcuni degli esercizi seguenti hanno lo scopo di far riflettere sul significato di semantiche non usuali per delineare la differenza tra alcune clausole DTD.

1. Scrivere almeno un esempio di etichettatura XML che stia nella semantica DTD seguente:

   ```
   <!ELEMENT s ANY>
   <!ELEMENT a (#PCDATA)>
   ```

2. Scrivere almeno un esempio di etichettatura XML che stia nella semantica DTD seguente:

   ```
   <!ELEMENT s ANY>
   <!ELEMENT a EMPTY>
   ```

3. Scrivere almeno un esempio ed un controesempio di etichettature XML, relative alla semantica DTD seguente:

   ```
   <!ELEMENT s (#PCDATA)>
   <!ELEMENT a (#PCDATA)>
   ```

4. Scrivere almeno un esempio ed un controesempio di etichettature XML, relative alla semantica DTD seguente:

```
<!ELEMENT s (#PCDATA)>
<!ELEMENT a EMPTY>
```

5. Scrivere almeno un esempio ed un controesempio di etichettature XML, relative alla semantica DTD seguente:

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

6. Scrivere almeno un esempio di etichettatura XML che stia nella semantica DTD seguente:

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

Esercizio 76 (Equivalenza tra grammatiche.) 1. Dire se, e perché, le due seguenti grammatiche DTD:

```
<!-- DTD 1 -->              <!-- DTD 2 -->
<!ELEMENT s ANY>            <!ELEMENT s (#PCDATA | a)*>
<!ELEMENT a (#PCDATA)>      <!ELEMENT a (#PCDATA)>
<!ELEMENT b (#PCDATA)>
```

sono equivalenti, ovvero se esse definiscono la stessa semantica, ovvero se gli insiemi di etichettature che esse descrivono sono identici.

2. Dire se, e perché, le due seguenti grammatiche DTD:

```
<!-- DTD 1 -->              <!-- DTD 2 -->
<!ELEMENT s ANY>            <!ELEMENT s (#PCDATA | a)*>
<!ELEMENT a (#PCDATA)>      <!ELEMENT a (#PCDATA)>
```

sono equivalenti.

3. Sia dato `sorgente.xml`, contenente il solo elemento `<a> </a>`.

   Siano date le due seguenti grammatiche DTD:

```
<!-- DTD 1 -->              <!-- DTD 2 -->
<!ELEMENT a (b*)>           <!ELEMENT a (b)*>
<!ELEMENT b EMPTY>          <!ELEMENT b EMPTY>
```

   A quale semantica appartiene `sorgente.xml`?

6.4 Ulteriori validatori

Esistono numerosi strumenti per la progettazione e la gestione di documenti XML e DTD. Alcuni possono essere disponibili *on-line* come servizi liberamente dispobili. I seguenti validatori: `DTD Validator` della `w3schools`, e *Scholarly Technology Group Validator* alla Brown University sono due possibili esempi. Anche l'elaboratore testi XML XRay 2.0, oltre a parsificare documenti XML, ne permette la validazione.

Esercizio 77 (Validatori dal comportamento omogeneo?) Non è scontato che i validatori si comportino nello stesso modo. Può valere la pena sperimentare l'uso di più validatori per osservarne il comportamento sui due seguenti testi XML.

Il primo sorgente è:

```
<?xml version="1.0" ?>
<!DOCTYPE es-id [
    <!ELEMENT es-id (a* | b)>
    <!ELEMENT a EMPTY>
    <!ATTLIST a i ID #REQUIRED>
    <!ELEMENT b EMPTY>
    <!ATTLIST b i IDREF #REQUIRED>
] >
<es-id>
  <a i="s1"/>
  <a i="s2"/>
  <b i="s2"/>
</es-id>
```

Esercizio 78 (Identificatori FIXED.) Sperimentare il comportamento di XML-Copy, o di altri validatori, sul seguente testo XML:

```
<?xml version="1.0" ?>
<!DOCTYPE r [
        <!ELEMENT r (e*)>
        <!ELEMENT e EMPTY>
        <!ATTLIST e a ID #FIXED "v">
]>
<r>
        <e a="v"/>
</r>
```

Capitolo 7

Interpretazione tramite trasformazione

- **Rifletteremo su cosa significhi interpretare un esempio di concetto, rappresentato da un documento XML, appartenente ad una data semantica, rappresentata da un documento DTD, ovvero, appartenente ad un insieme di esempi del medesimo concetto.**

  Formalmente, realizzeremo l'interpretazione come trasformazione di alberi sintattici in alberi sintattici.

- **Come caso particolare delle riflessioni al punto precedente, vedremo che fornire una adeguata veste tipografica ad un esempio di concetto, equivale a darne un'interpretazione.**

7.1 Interpretare equivale a trasformare

Idea chiave di quel che segue è:

> *Interpretare un concetto equivale a darne una presentazione alternativa.*
>
> Equivalentemente, senza *trasformazione* del concetto iniziale in
>
> "*qualcosa di diverso, ma equivalente*"
>
> non ha senso parlare di interpretazione.

Esempio 42 (Interpretazioni di varia natura.)

- Un cuoco è l'interprete di una ricetta. La rappresentazione alternativa che egli produce consiste nel piatto preparato.
- Un interprete (mediatore?) linguistico converte frasi di un linguggio in frasi di un altro linguaggio.
- Uno studente, durante un esame scritto, interpreta domande, producendo risposte corrispondenti.

7.2 Interpretazione di concetti (etichettati)

Il nostro scopo è intepretare, per mezzo di un calcolatore, concetti astratti, rappresentati per mezzo di testi etichettati. Ricordiamo che:

- *Etichettare esempi* di rappresentazioni testuali di un concetto ne evidenzia le *componenti rilevanti* per uno scopo prefissato.
- Una *semantica riassume caratteristiche comuni alla struttura* di un insieme di esempi, relativi allo stesso concetto in esame.

Interpretare concetti etichettati, appartenenti ad una data semantica, equivale a trasformare alberi sintattici in alberi sintattici. Partire da un insieme di concetti in una stesssa semantica, quindi con struttura sintattica simile, permette una descrizione uniforme del processo di interpretazione.

7.2.1 Interpretazione tra concetti arbitrari

Discuteremo brevemente il significato di interpretare un concetto arbitrario in un altro concetto arbitrario.

L'ipotesi è disporre di esempi testutali del primo concetto, etichettati secondo la sintassi XML.

Lo scopo è produrre un esempio del secondo concetto in relazione "ragionevole" con l'esempio di partenza.

Più formalmente, supponiamo C_1 e C_2 siano concetti astratti qualsiasi. Quindi, possiamo immaginare che E_{C_1} sia un testo XML che costitusce un esempio di C_1 e che E_{C_2} sia un altro testo XML, esempio per C_2. Per quel che abbiamo detto sinora, E_{C_1} apparterrà ad una semantica formalizzabile con un documento DTD S_1, mentre E_{C_2} apparterrà ad S_2.

Scopo. Per ogni esempio (di etichettatura XML) E_{C_1} di C_1, occorre *interpretare* E_{C_1} in qualche E_{C_2}, esempio di C_2.

Metodo. Occorre descrivere come le strutture riassunte da S_1 possano essere trasformate in strutture di S_2, preservandone il significato inteso. Questo equivarrà a poter effettivamente trasformare E_{C_1} in E_{C_2}.

Perché correlare semantiche e non solo esempi?

- Le semantiche descrivono le proprietà comuni a tutti gli esempi di un concetto.
- Progettare una trasformazione pensando ad un singolo esempio, può indurre a trascurare aspetti strutturali di altri oggetti, individuati dalla medesima semantica, rendendo impossibile l'interpretazione di un qualsiasi esempio di C_1 nel corrispondente di C_2.

Esempio 43 (Un'interpretazione tra numeri.) Siano date le due seguenti semantiche `numero-a` e `numero-b`, i cui commenti illustrano le proprietà strutturali.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- numero-a descrive numeri binari,
     ovvero composti da due sole cifre.
     Ogni esempio di numero-a puo'
     essere senza cifre.
     In caso contrario la sequenza di
     cifre che lo compongono non puo'
     avere zeri non significativi -->
<!DOCTYPE numero-a  [
<!ELEMENT vuoto EMPTY>
```

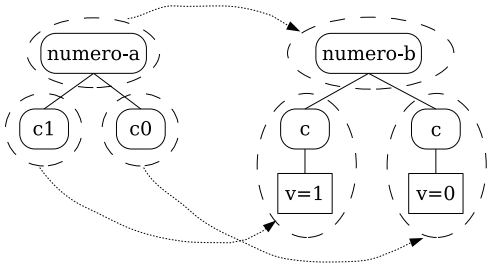
```
<!ELEMENT c1 EMPTY>
<!ELEMENT c0 EMPTY>
<!ELEMENT numero-a (vuoto|c0|(c1,(c0|c1)*))>
]>

<?xml version="1.0" encoding="UTF-8"?>
<!-- Anche numero-b descrive numeri binari.
     Ogni esempio di numero-b e' una
     sequenza di cifre in cui possano
     comparire piu' zeri non significativi -->
<!DOCTYPE numero-b  [
 <!ELEMENT c EMPTY>
  <!ATTLIST c v (0|1) #REQUIRED>
 <!ELEMENT numero-b (c)*>
]>
```

In particolare:

- `<numero-a><c1><c0></numero-a>` è un esempio per `numero-a`.
- `<numero-a><c0><c0></numero-a>` è un controesempio per `numero-a`.
- `<numero-b><c v="0"/><c v="0"/></numero-b>` è un esempio per `numero-b`.

Significati e corrispondenze.

- *Interpretare* `numero-a` *in* `numero-b` *significa trasformare ogni esmpio valido per* `numero-a` *in uno valido per* `numero-b`.
- È ragionevole immaginare che `<numero-a><c1><c0></numero-a>` corrisponda a `<numero-b><c v="0"><c v="1"></numero-b>`.
- Il modo "naturale" per definire la trasformazione è basarsi sulla "*visita*" dell'albero sintattico dell'esempio di `numero-a`, creando le seguenti corrispondenze:



I nodi "cifra" della prima semantica diventano nodi con attributi della seconda, in cui ogni attributo ha un valore numerico corrispondente a quello dell'elemento di `numero-a` da cui "proviene".

Commento 15
In linea di principio, la definizione astratta di interpretazione, intesa come trasformazione da alberi sintattici in alberi sintattici, ammette che ogni semantica sia interpretabile (trasformabile) in una *qualsiasi altra* semantica.

Infatti, nonostante possa risultare privo di senso, è possibile immaginare una interpretazione (trasformazione) di (semantica di) *cv* in (semantica di) assegno bancario.

Esercizio 79 (Dal *cv* all'assegno bancario.) Supponiamo di disporre di una semantica DTD dei concetti `curriculum-vitae` e `assegno-bancario`.

Ipotizzare anche che `<assegno-bancario/>` sia un esempio valido per la seconda semantica, ovvero che tra gli esempi validi, esista un assegno bancario minimale.

Descrivere, con parole proprie, quale tipo di corrispondenza possa essere definita per poter interpretare un qualsiasi esempio della prima semantica in un esempio della seconda.

Commento 16 ("Risposte istintive" a 79.)
Probabilmente ne esistono due:

- **Prima risposta.** La semantica `assegno-bancario.dtd` è definita così da inglobare, in qualche modo, le proprietà di `curriculum-vitae.dtd`.

  Una soluzione di questo tipo è criticabile per due motivi:

  - Ignora, ovvero non sfrutta, l'ipotesi secondo cui `<assegno-bancario/>` è un esempio di `assegno-bancario.dtd`.
  - Se, per creare una corrispondenza tra un `curriculum-vitae` e un `assegno-bancario` la semantica di `assegno-bancario` deve essere stravolta per inglobare quella di `curriculum-vitae`, allora si perde il senso dell'etichettatura.

- **Seconda risposta.** Le semantiche `assegno-bancario.dtd` e `curriculum-vitae.dtd` sono del tutto indipendenti.

  La strada è corretta e richiede di sfruttare opportunamente l'esistenza dell'elemento `<assegno-bancario/>` in `assegno-bancario.dtd`, definendo un'interpretazione che sia la meno informativa di tutte.

La riflessione sull'esistenza di interpretazioni assolutamente non informative, ovvero di interpretazioni che non permettono l'associazione di parti strutturali di un concetto di partenza (sorgente) con parti strutturali di un concetto di arrivo (oggetto), costituisce proprio lo scopo dell'esercizio.

7.2.2 Sorgente ed oggetto

Per parlare dell'interpretazione di una semantica in un'altra è utile definire i concetti di *sorgente* e *oggetto*.

- **Sorgente** indica una semantica S le cui istanze XML debbano essere interpretate. Di riflesso, ogni documento XML s che appartenga a S è un (documento) *sorgente*.
- **Oggetto** indica una semantica O le cui istanze XML sono usate come interpretazione. Di riflesso, ogni documento XML o che appartenga ad O è un (documento) *oggetto*.

7.3 Interpretazione tipografica di concetti

Quanto detto sinora sul meccanismo di interpretazione tra due concetti che appartengano a due semantiche generiche è stato estremamente generale ed aveva scopo introduttivo.

> *Noi siamo interessati soprattutto all classe di interpretazioni tipografiche.*

Supponendo che C_1 sia un concetto con semantica S_1:

- L'*interpretazione tipografica* di ogni esempio E_{C_1} di C_1, che appartenga a S_1, trasforma E_{C_1} in un testo (oggetto) equivalente.
- La caratteristica fondamente dei testi oggetto ottenuti per trasformazione è che ciascuno di essi assegna una connotazione visuale "gradevole ed appropriata" alle componenti del testo sorgente.

Esempio 44 (Proverbi in "bella copia".) La strutturazione ad etichette della seguente raccolta di proverbi:

```
<raccolta>
<titolo>Proverbi</titolo>
 <proverbio origine="Cina">
  Chi sente <evidenzia>dimentica</evidenzia>,
  chi vede <evidenzia>ricorda</evidenzia>,
  chi fa <evidenzia>impara</evidenzia>.
 </proverbio>

 <proverbio origine="Norvegia">
  Se ognuno pulisce davanti a casa,
  tutta la citt<agrave/> <egrave/> pulita.
 </proverbio>
</raccolta>
```

suggerisce una possibile sua interpretazione visuale per mezzo di un testo, costituito da:

- un titolo, seguito da
- una lista dei proverbi, in cui

- ogni punto espliciti il paese d'origine.

Una possibile interpretazione tipografica del testo potrebbe, quindi, essere:

Esercizio 80 (Interpretazione tipografica di Proverbi.) 1. Scrivere il testo XHTML che un *browser* interpreterebbe tipograficamente come nell'immagine precedente.

2. Scrivere il testo XHTML che un *browser* interpreterebbe tipograficamente come segue:

7.4 Qualche intuizione su come funziona

Ci accingiamo a costruire l'intuizione su cosa significhi interpretare un documento sorgente in uno oggetto.

Partiremo da due istanze di documenti XML. Una sarà il documento sorgente. L'altra conterrà il programma, ovvero l'insieme di regole che, se opportunamente lette, descrivono come trasformare il sorgente in un documento XHTML. Vedremo che le regole sono scritte nel "dialetto" XML di nome XSLT.

7.4.1 Documenti di riferimento

Per raggiungere il nostro scopo, fissiamo sorgente XML e programma XSLT.

- **Il sorgente.** Esso è `raccolta-proverbi.xml` e contiene una lista di proverbi, ciascuno associato alla nazione d'origine:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- <!DOCTYPE raccolta SYSTEM "raccolta-proverbi.dtd"> -->
<raccolta>
  <titolo>Proverbi</titolo>
  <proverbio origine="Cina">
  Chi sente <evidenzia>dimentica</evidenzia>,
  chi vede <evidenzia>ricorda</evidenzia>,
  chi fa <evidenzia>impara</evidenzia>.
 </proverbio>
  <proverbio origine="Norvegia">
  Se ognuno pulisce davanti a casa,
  tutta la citt<agrave/> <egrave/> pulita.
 </proverbio>
</raccolta>
```

  Per comodità, ricordiamo che la DTD esterna `raccolta-proverbi.dtd` contiene:

```
 <!ELEMENT raccolta (titolo,proverbio*)>
 <!ELEMENT titolo (#PCDATA)>
 <!ELEMENT proverbio (#PCDATA|evidenzia|agrave|egrave)*>
  <!ATTLIST proverbio origine CDATA #IMPLIED>
 <!ELEMENT evidenzia (#PCDATA)>
 <!ELEMENT agrave EMPTY>
 <!ELEMENT egrave EMPTY>
```

- **Il programma.** `raccolta-proverbi-in-lista.xsl` è il programma. È un documento XSLT che guida la trasformazione del documento sorgente nel corrispondente oggetto.

  `raccolta-proverbi-in-lista.xsl` è un insieme di regole di trasformazione.

  Scopo delle regole. Lo scopo delle regole è guidare la "visita" dell'albero sintattico di `raccolta-proverbi.xml` in modo che, in corrispondenza di ogni nodo in esso contenuto, venga effettuata una qualche operazione che, composta con quelle che la precedono e con quelle che la seguono, generi un documento XHTML `raccolta-proverbi-in-lista.html`.

  `raccolta-proverbi-in-lista.html` è l'interpretazione tipografica di `raccolta-proverbi-in-lista.xml`.

Il significato delle componenti `raccolta-proverbi-in-lista.xsl` è evidenziato dai commenti che sottolineano come i testi XSLT siano regole di trasformazione che debbano essere pensate relativamente alle parti di testo sorgente cui devono essere applicate:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <!-- Regola che usa la radice del documento sorgente
      per produrre quella del documento oggetto -->
 <xsl:template match="/">
  <html>
   <head/>
   <xsl:apply-templates select="raccolta"/>
  </html>
 </xsl:template>

 <!-- Regola che predispone la presentazione della raccolta di
      proverbi come titolo, seguito da un lista di elementi -->
 <xsl:template match="raccolta">
  <body>
   <xsl:apply-templates select="titolo"/>
   <ul>
    <xsl:apply-templates select="proverbio"/>
   </ul>
  </body>
 </xsl:template>

 <!-- Regola che sceglie una rappresentazione
      tipografica per il titolo -->
 <xsl:template match="titolo">
  <h1>
   <xsl:value-of select="."/>
  </h1>
 </xsl:template>

 <!-- Regola che trasforma ogni proverbio in un elemento di
      lista nell'elenco. L'elemento comincia col paese d'origine
      e segue col testo del proverbio -->
 <xsl:template match="proverbio">
  <li>
   <xsl:apply-templates select="@origine"/>
    <xsl:apply-templates/>
  </li>
 </xsl:template>
```

```
<!-- Regola che evidenzia in grassetto l'origine del
     proverbio  -->
<xsl:template match="@origine">
 <b>
  <xsl:value-of select="."/>:
 </b>
</xsl:template>

<!-- Regola che trasforma le parti evidenziate in corsivo -->
<xsl:template match="evidenzia">
  <i><xsl:value-of select="."/></i>
</xsl:template>

<!-- Regola che traduce correttamente gli accenti -->
<xsl:template match="agrave">&#224; <!-- a accentata grave -->
</xsl:template>

<xsl:template match="egrave">&#232; <!-- e accentata grave -->
</xsl:template>

</xsl:stylesheet>
```

Da sorgente ad oggetto attraverso l'XSLT

La Figura 7.1 riassume il collegamento logico tra il documento *sorgente* `raccolta-proverbi-in-lista.xml` ed il documento *oggetto* `raccolta-proverbi-in-lista.html`, attraverso il documento *programma* `raccolta-proverbi--in-lista.xsl`.

Struttura della figura. La figura è, idealmente, una tabella di due righe per due colonne.

Prima riga. Da sinistra a destra, essa contiene programma e sorgente.

Seconda riga. Da sinistra a destra, contiene programma ed oggetto.

7.4.2 Significato della prima riga della tabella

Essa collega elementi ed attributi del sorgente a porzioni del programma.

Ogni porzione di testo del programma è una *regola di trasformazione*.

- *Sintassi delle regole.* È evidente che le regole sono scritte in accordo con la sintassi dei *tag* XML.
- *Attivazione.* Ogni regola è associata all'occorrenza di elemento che la attiva.
- *Regole attivate da un tag.* La maggior parte delle regole contiene l'attributo `match="nome-tag"`.

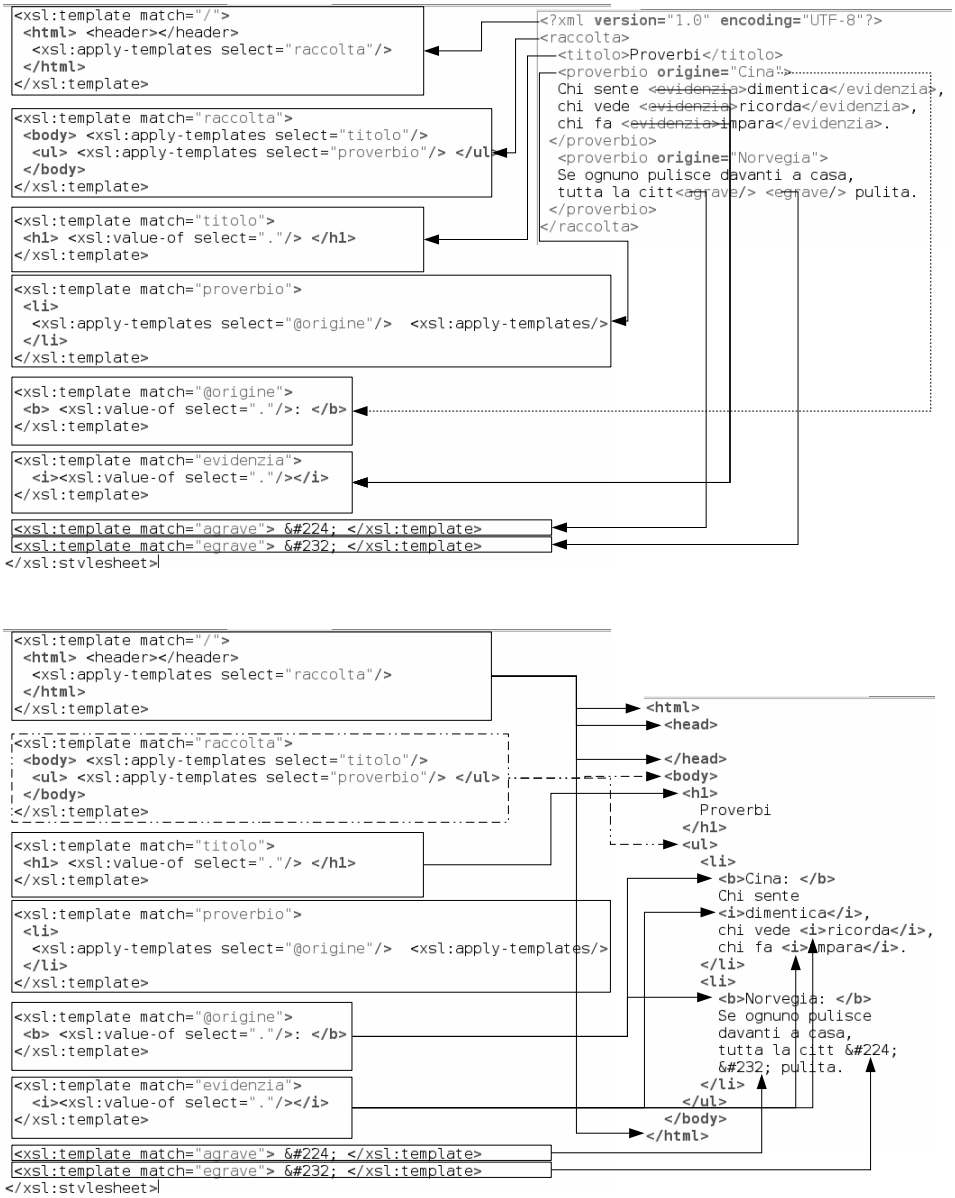


Figura 7.1: Da XML ad XHTML, attraverso XSLT

Ogni regola con tale attributo è attivata da un *tag* `nome-tag` contenuto nel sorgente, ed incontrato durante la visita del suo albero sintattico.

- *Regole attivate da un attributo.* Qualche regola contiene l'attributo `match="@nome-attributo-sorgente"`.

  Ogni regola con tale attributo è attivata da un *tag*, contenuto nel sorgente, dotato di un attributo `nome-attributo-sorgente`, ed incontrato durante la visita del suo albero sintattico.

7.4.3 Significato della seconda riga della tabella

Essa descrive l'effetto dell'attivazione delle regole del programma.

Situazione di partenza. Occorre immaginare che il documento oggetto sia inizialmente vuoto, ovvero che possa essere pensato come un foglio bianco su cui poter scrivere dei caratteri.

Iterazione delle regole. La trasformazione avviene tentando l'attivazione delle regole sugli elementi dell'albero sintattico che descrive il sorgente.

L'albero sintattico è visitato in *preordine*, ovvero, cominciando dai nodi più a sinistra e più profondi.

Quando nessuna regola è più applicabile l'iterazione termina ed il documento oggetto è disponibile per l'utilizzo.

Cosa produce l'attivazione di una regola?

"Attivare" una regola può significare "scrivere caratteri".

Primo esempio: regola per la radice sorgente. È la prima regola che incontriamo nel programma XSLT.

Essa contiene l'attributo `match="/"`.

Quindi, essa è attivata dalla radice del documento sorgente.

Dopo l'attivazione, la regola scrive nel documento oggetto il testo che troviamo tra il *tag* di inizio regola `<xsl:template match="/">` e quello di fine regola `</xsl:template>`.

Tale testo è costituito dai *tag* XHTML `<html><head/></html>`.

Una volta scritto il testo, il processo di interpretazione delle regole riprende in accordo con quanto specificato dalla clausola
`<xsl:apply-templates select="raccolta"/>`.

Tale clausola indica che nel documento sorgente occorre cercare occorrenze del *tag* `raccolta`.

Secondo esempio: regola attivata dal *tag* con attributo `raccolta`. È la seconda regola che incontriamo nel testo del programma XSLT.

Essa contiene l'attributo `match="raccolta"`.

Quindi, essa è attivata da ogni occorrenza del *tag* `raccolta` nel documento sorgente.

Dopo l'attivazione, la regola scrive nel documento oggetto il testo che troviamo tra il *tag* di inizio regola `<xsl:template match="raccolta">` e quello di fine regola `</xsl:template>`.

Tale testo è costituito dai *tag* XHTML `<body><ul></ul></body>`.

Una volta scritto il testo, il processo di interpretazione delle regole riprende in accordo con quanto specificato dalle clausole
`<xsl:apply-templates select="titolo"/>` e
`<xsl:apply-templates select="proverbio"/>`.

La prima indica che nel documento sorgente occorre cercare occorrenze del *tag* `titolo` cui seguirà l'aggiunta di testo XHTML, nel documento oggetto, tra i *tag* `<body>` ed `<ul>`.

La seconda indica che nel documento sorgente occorre cercare occorrenze del *tag* `proverbio` cui seguirà l'aggiunta di testo XHTML, nel documento oggetto, tra i *tag* `<ul>` ed `<\ul>`.

Funzionamento analogo delle altre regole. Con un meccanismo simile a quello descritto per le due prime regole che incontriamo nel testo del programma XSLT possiamo effettivamente produrre il testo XHTML che troviamo in seconda riga e seconda colonna della Figura 7.1.

7.5 Dove siamo

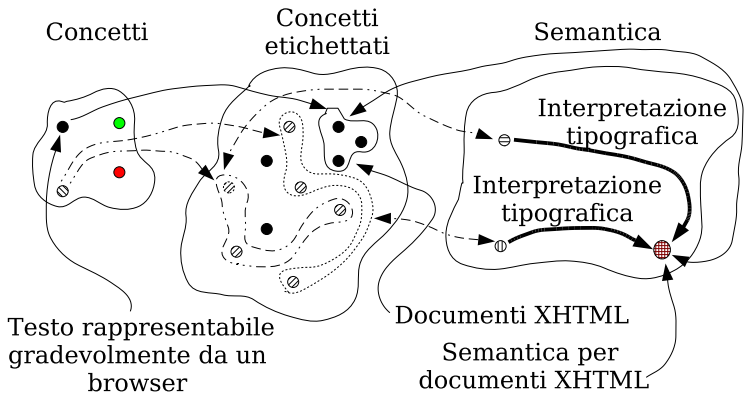


Figura 7.2: Concetti, etichettature, semantica ed interpretazione

Commentiamo la Figura 7.2.

In "natura" esiste il concetto dei *testi rappresentati gradevolmente per mezzo di un browser* per la navigazione internet.

Analogamente a qualsiasi altro concetto, possono esistere più etichettature, ovvero più semantiche, per esprimere le caretteristiche strutturali di testi con rappresentazioni tipografiche gradevoli in un *browser*: XHTML è uno di questi.

L'interpretazione tipografica descrive le regole per trasformare una semantica che indentifica un insieme di documenti XML, usati come rappresentazione di un qualche concetto astratto, nella semantica per documenti XHTML. L'interpretazione è detta "tipografica" proprio perché i *browser* interpretano gradevolmente i documenti XHTML.

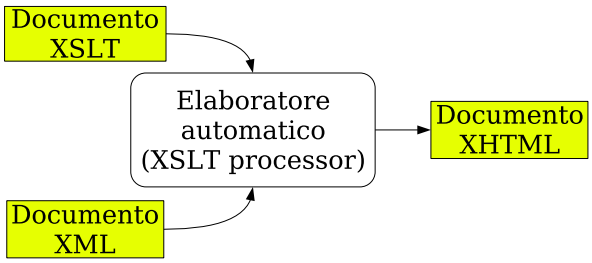
Capitolo 8

Interpretazione ed Elaborazione Automatica

Interpreteremo documenti XML per mezzo XSLT *processors*.

8.1 XSLT *processor*

Ogni XSLT *processor* è un programma che usa due documenti XML come dati di ingresso, in accordo col seguente schema:



Uno dei documenti, che per comodità chiamiamo `sorgente.xml`, è il sorgente da interpretare.

Il secondo documento, che per comodità chiamiamo `trasformazione.xsl`, è un XML *StyLesheeT*, abbreviato in XSLT. Esso guida il *processor* nell'interpretazione di `sorgente.xml`. Il *file* `trasformazione.xsl`, usando la sintassi XML, descrive regole di trasformazione.

Lo schema precedente suggerisce il punto di vista di questo capitolo, illustrando che il risultato di un *processor* debba essere un documento XHTML.

In realtà, un XSLT *processor* può produrre un qualsiasi tipo di documento anche completamente scorrelato da una qualsiasi semantica DTD.

I concetti *pattern matching* e *template* , che vedremo in seguito nel loro dettaglio, sono i due strumenti concettuali che guidano la scrittura delle regole in `trasformazione.xsl`. In linea di massima:

pattern matching sta per "identificazione di modello".

> Il *pattern matching* è un processo realizzabile per mezzo di un calcolatore elettronico che individua parti del testo sorgente ed il cui fine, nel caso in esame, è guidare una trasformazione.

template sta per "modello strutturale".

> Un *template* descrive una porzione di un documento oggetto da produrre, che può dipendere da una porzione di documento sorgente.

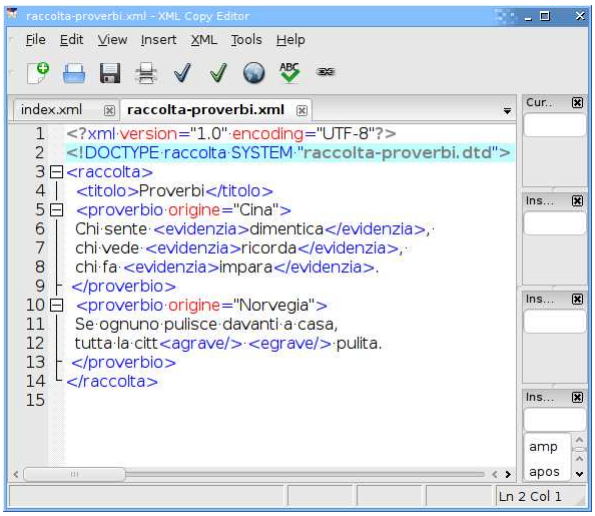
8.2 L'XSLT *processor* di XMLCopy

L'ipotesi per usare l'XSLT *processor* in XMLCopy è di aver già memorizzato:

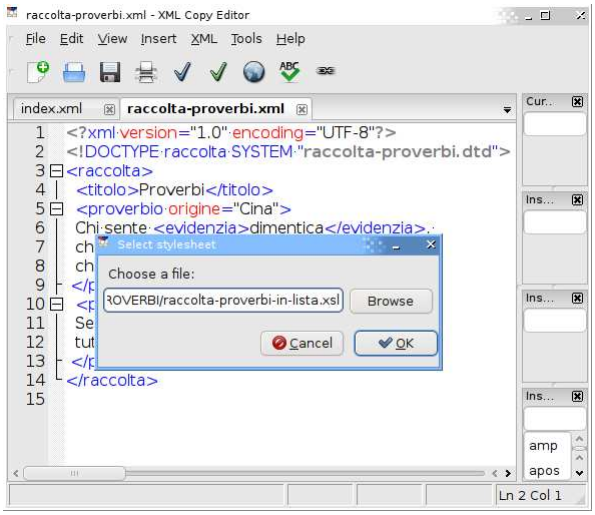
`raccolta-proverbi.xml` che usiamo come sorgente e

`raccolta-proverbi-in-lista.xsl` , il nostro documento XSLT, entrambi in Sezione 7.4.1, in una qualche cartella.

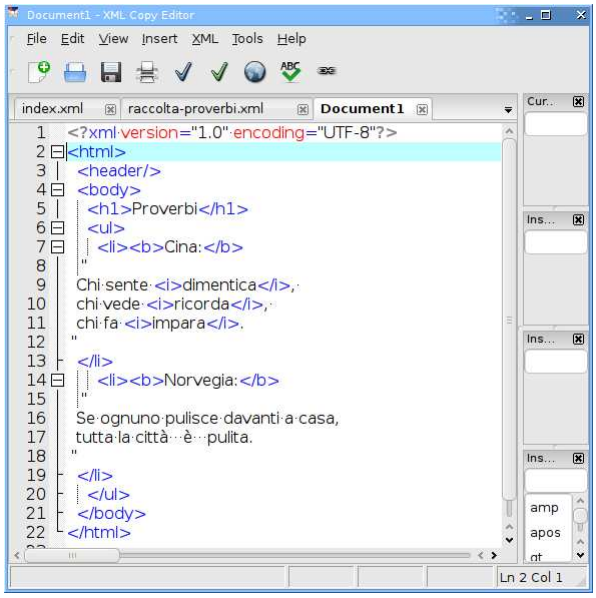
1. Leggiamo `raccolta-proverbi.xml` in XMLCopy:



2. Per eseguire la trasformazione pigiamo il tasto `F8`, o apriamo il menù a tendina `XML` per sceglie la voce `XSL Transform...`. Si apre una classica finestra per mezzo della quale cercare un *file* che, nel nostro caso, per ipotesi, dovrà essere `raccolta-proverbi-in-lista.xsl`:

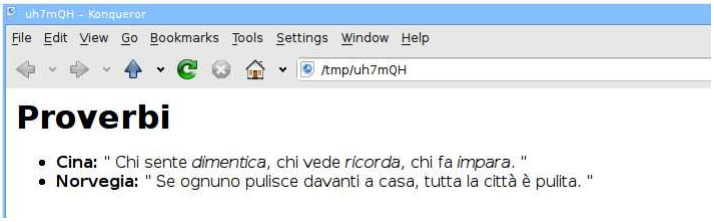


3. Viene creato un nuovo *tab* di nome `Document1` che contiene un documento XHTML:



4. Nel caso interessi, per memorizzare `Document1` sul disco basta cliccare sull'icona a forma di *floppy disk*, oppure premere contemporaneamente i tasti `Control` e `s`. Nome ragionevole da assegnare al *file* da memorizzare è `raccolta-proverbi-in-lista.html`.

   Quest'ultimo è visualizzabile in un *browser*. Tuttavia, indipendentemente dall'aver memorizzato `raccolta-proverbi-in-lista.html` sul disco, esso è visualizzabile attraverso il *browser* di default con un *click* sull'icona di XMLCopy che ritrae il mondo:



   Possiamo osservare che `raccolta-proverbi.xml` è stato interpretato in un documento XHTML gradevolmente riproducibile da un *browser*.

Esercizio 81 (*Browser* come XSLT *processor*.) I *browser* attuali incorporano un XSLT *processor*. Per verificarlo, modifichiamo `raccolta-proverbi.xml` modificandone il preambolo. Immediatamente dopo la riga d'intestazione:

```
<?xml version="1.0" encoding="UTF-8"?>
```

aggiungiamo il testo:

```
<?xml-stylesheet type="text/xsl"
                 href="raccolta-proverbi-in-lista.xsl"?>
```

e salviamo la nuova versione.

Il testo appena aggiunto indica il foglio XSLT per interpretare `raccolta-proverbi.xml`. Per semplicità, assumiamo che il foglio XSLT sia nella stessa cartella del sorgente.

Cosa succede leggendo la nuova versione di `raccolta-proverbi.xml` in un *browser*?

Capitolo 9

Modello di Calcolo per l'Interpretazione

Impareremo ad usare il linguaggio XSLT. Lo scopo è interpretare insiemi di documenti XML che appartengano alla stessa semantica, ovvero che siano descritti da uno stesso documento DTD.

In particolare, capiremo:

- i criteri per comporre un documento XSLT che, alla fine, conterrà regole di trasformazione di documenti XML generici in altri documenti;
- il meccanismo di attivazione delle regole di trasformazione, contenute in documenti XSLT;
- il meccanismo di costruzione di "porzioni" di documenti XML oggetto, per mezzo delle regole che costituiscono un documento XSLT;
- il meccanismo **ricorsivo** di "visita" dell'albero sorgente di un documento XML per produrre il documento oggetto, tramite la scelta del **nodo attivo** e delle regole di trasformazione opportune.

9.1 Documenti XSLT

Ogni documento XSLT è un documento XML che **formalizza un algoritmo**.

Un *algoritmo* è costituito da una sequenza finita di passi, ciascuno interpretabile da un interprete prefissato.

Nel caso in esame, l'interprete è un *processor* XSLT.

Scopo dell'algoritmo, descritto dal documento XSLT, è trasformare ogni documento XML che appartenga ad una semantica sorgente in un documento oggetto *non* necessariamente XML.

Ogni documento XSLT è composto da due parti distinte **Preambolo** e **Insieme di regole d'interpretazione**.

9.1.1 Preambolo di un documento XSLT

La sintassi essenziale del preambolo di un documento XSLT è la seguente:

```
<?xml version="1.0"?>
 <!-- Clausola che descrive la natura di questo documento -->
 <xsl:stylesheet
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      version="1.0">

 <!-- Clausola che descrive la natura del documento oggetto -->

 <!-- Segue l'insieme di regole
      costituenti questo documento XSLT -->
</xsl:stylesheet>
```

Senza questa intestazione gli interpreti XSLT non classificano il documento come parte dell'insieme dei documenti XSLT, e, quindi, non "vedono" la parte seguente l'intestazione come elenco di regole di trasformazione.

Inoltre, il commento

```
<!-- Clausola che descrive la natura del documento oggetto -->
```

va sostituito con un *tag* che specifichi la natura del documento oggetto, come segue:

- Nel caso l'oggetto sia un documento XML, il *tag* deve essere:

  ```
  <xsl:output method="xml"/>
  ```

  In questo modo la sintassi XML dell'oggetto non viene "confusa" con quella del programma XSLT.

- Nel caso l'oggetto sia un documento XHTML, il *tag* deve essere

  ```
  <xsl:output method="html"/>
  ```

  Anche in questo caso, la sintassi XHTML dell'oggetto non viene "confusa" con quella del programma XSLT.

- Nel caso l'oggetto non sia un documento XML, il *tag* deve essere

  ```
  <xsl:output method="text"/>
  ```

Commento 17
L'attributo `xmlns:xsl` con valore `http://www.w3.org/1999/XSL/Transform`, è nella categoria *namespaces*, della semantica XSLT.

9.1.2 Regole d'interpretazione di un documento XSLT

L'insieme di regole, contenute in documento XSLT, è espresso sotto forma di *sequenza*:

- Ogni **regola** descrive un **modello di trasformazione** (*template*) di una porzione del documento XML sorgente in una porzione del documento oggetto;

- Ogni **regola** contiene un **predicato di applicazione** del *template*.

  Il predicato è una "frase" del linguaggio XPath di cui si introdurranno solo gli elementi essenziali.

Esempio 45 (*Template* con predicato di applicazione universale.) La regola seguente:

```
<xsl:template match="*">
 <html><html/>
</xsl:template>
```

descrive un *template* (di parte) di un albero oggetto.

- Il predicato di attivazione è il valore `*` dell'attributo `match`.
  - Esso è *soddisfatto* da un qualsiasi nodo dell'abero sintattico del documento XML sorgente in esame da parte di un *processor* XSLT.
  - Il soddisfacimento del predicato di attivazione implica l'applicazione del *template*, da parte del *processor* XSLT, al nodo attivo del documento XML sorgente.
- Il *template* è la porzione racchiusa tra i *tag* di inizio e fine elemento `xsl:template`.

  Nel caso in esame il *template* produce un albero sintattico oggetto, costituito dal solo nodo `html`.

9.2 Interpretazione di una regola

Stabilito che un documento XSLT è formato da regole che devono essere usate per trasformarne uno XML in un altro, non necessariamente XML, occorre dettagliare:

- il meccanismo di **attivazione di una regola**, ovvero i criteri che ne decretano la "**necessità**" di utilizzo da parte del *processor* XSLT;
- il meccanismo di **costruzione di** "**porzioni**" di albero sintattico, da parte del *template*: una volta agglomerate le porzioni, in accordo con l'ordine di attivazione delle regole, si ottiene il risultato globale della trasformazione.

9.2.1 Sintassi del predicato di attivazione

Prima di introdurre il meccanismo di attivazione di una regola, ovvero il meccanismo di applicazione dei *template*, presentiamo la sintassi dei predicati base di attivazione delle regole.

Il *predicato di attivazione* di una regola è definito dalla espressione (del linguaggio XPath) situata tra le "virgolette" che individuano il valore dell'attributo `match` del *tag* di apertura `xsl:template`.

Esempio 46 Struttura generica di regola, con predicato generico.

```
<xsl:template match="espressione che definisce il predicato">
 <!-- struttura del template -->
</xsl:template>
```

Gli scopi didattici, finalizzati alla comprensione del modello comportamentale di ogni *processor* XSLT, limitano l'interesse ai soli *predicati di attivazione* seguenti:

- "`/`", che identifica la radice dell'albero sintattico sorgente,
- "`*`", che identifica un qualsiasi nodo dell'albero sintattico sorgente,
- "`nome-elemento-sorgente`", che identifica esattamente un esempio, ovvero *occorrenza*, del nodo `nome-elemento-sorgente` dell'albero sintattico sorgente,
- "`text()`", che identifica solo foglie contenenti sequenze di caratteri di tipo `#PCDATA` dell'albero sintattico sorgente.

9.2.2 Uso del predicato di attivazione

Vediamo, ora, il meccanismo di attivazione di una regola, tramite il soddisfacimento del suo predicato di attivazione. L'attivazione segue alcuni criteri generali:

- Occorre immaginare che, in qualsiasi istante dell'utilizzo di un documento XSLT da parte di un *processor* XSLT, esiste un *nodo attivo nell'albero sorgente* che chiamiamo N.
- Il *processor* **interpreta una regola alla volta** scelta dall'insieme di regole del documento XSLT in funzione della struttura di N.

  Infatti, N può soddisfare il *predicato di attivazione* di una qualche regola, la quale viene usata effettivamente per produrre una porzione del documento oggetto.

  L'individuazione di N dipende dal comportamento globale del *processor* XSLT, ma la comprensione del funzionamento globale non è rilevante per capire come una singola regola venga attivata.

Ipotesi di lavoro. Siano:

- S l'albero sorgente,
- N il **nodo attivo** in S,
- R una regola con predicato di attivazione P_R in un documento XSLT da interpretare.

Caso con P_R uguale a "/". Se P_R coincide con "/", allora P_R è soddisfatto dal solo elemento radice `<?xml version="1.0"encoding="UTF-8"?>` di un qualsiasi documento XML sorgente.

L'applicazione della regola può produrre una porzione di albero oggetto.

Esempio 47 La regola:

```
<xsl:template match="/">
  <html>
   <head/>
   <body>
   </body>
  </html>
</xsl:template>
```

è attivata dal nodo radice `<?xml version="1.0" encoding="UTF-8"?>`, presente in ogni documento XML, e produce l'albero oggetto:

```
  <html>
   <head/>
   <body>
   </body>
  </html>
```

Esercizio 82 (Attivazione del predicato "/".) Elencare i nodi del seguente sorgente XML:

```
<?xml version="1.0" encoding="UTF-8"?>
 <radice/>
```

che possano attivare il predicato della regola:

```
<xsl:template match="/">
  <!-- Template -->
</xsl:template>
```

Caso con P_R uguale a "∗". Se P_R coincide con "∗", allora P_R è soddisfatto, indipendentemente dal nome del nodo attivo N e l'applicazione della regola può produrre una porzione di albero oggetto.

Esempio 48 La regola:

```
<xsl:template match="*">
  <h2>Prova</h2>
</xsl:template>
```

è attivata da uno qualsiasi fra `<?xml version=1.0 encoding=UTF-8?>`, `nota`, `msg`, `sorgente`, `da` e `testo` del seguente documento d'esempio:

```
<?xml version="1.0" encoding="UTF-8"?>
<nota>
 <msg>
  <sorgente>telefono</sorgente>
  <da>Tizio</da>
  <testo>Arrivero' al piu' presto</testo>
 </msg>
```

```
 <msg>
  <sorgente>e-mail</sorgente>
  <da>Caio</da>
  <testo>Non so se arrivero'</testo>
 </msg>
</nota>
```

Se, per ipotesi, N coincidesse con la prima occorrenza di `sorgente`, la regola produrrebbe `<h2>Prova</h2>`. Lo stesso risultato si avrebbe se N coincidesse con la seconda occorrenza di `sorgente`, o con la terza di `testo`, ovvero, con l'occorrenza di un qualsiasi nodo del sorgente.

Esercizio 83 (Attivazione del predicato "*".) Dato il sorgente:

```
<?xml version="1.0" encoding="UTF-8"?>
 <radice>
   <nodo>
     <valore>A</valore>
     <nodo-sx><foglia/></nodo-sx>
     <nodo-dx><foglia/></nodo-dx>
   </nodo>
 </radice>
```

e la regola:

```
<xsl:template match="*">
  <!-- template -->
</xsl:template>
```

il predicato `match="*"`: i) sarebbe attivato se N fosse `nodo`, oppure ii) sarebbe attivato se N fosse `nodo-sx`?

Caso con P_R uguale a "nome-elemento". Se P_R ed il nodo corrente N coincidono con `nome-elemento`, allora P_R è soddisfatto e la regola può produrre una porzione di albero oggetto.

Esempio 49 La regola:

```
<xsl:template match="testo">
  <td>Prova</td>
</xsl:template>
```

è attivata *solo* nel caso in cui il nodo attivo N coincida con una delle due occorrenze del nodo `testo` nel documento d'esempio seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<nota>
 <msg>
  <sorgente>telefono</sorgente>
```

```
  <da>Tizio</da>
  <testo>Arrivero' al piu' presto</testo>
 </msg>
 <msg>
  <sorgente>e-mail</sorgente>
  <da>Caio</da>
  <testo>Non so se arrivero'</testo>
 </msg>
</nota>
```

In entrambi i casi la regola produce `<td>Prova</td>`. Altrimenti, se il testo oggetto non contenesse alcun nodo `testo`, la regola non sarebbe mai attivata, quindi mai usata.

Esercizio 84 (Attivazione di un predicato "`elemento`".) Dato il sorgente:

```
<?xml version="1.0" encoding="UTF-8"?>
 <radice>
   <nodo>
     <valore>A</valore>
     <nodo><foglia/></nodo>
     <nodo><foglia/></nodo>
   </nodo>
 </radice>
```

e la regola:

```
<xsl:template match="nodo">
  <!-- template -->
</xsl:template>
```

i) Il predicato `match="nodo"` sarebbe attivato se N fosse l'occorrenza più esterna di `nodo`? ii) Il predicato `match="nodo"` sarebbe attivato se N fosse `valore`? iii) Il predicato `match="valore"` sarebbe attivato se N fosse l'occorrenza più esterna di `nodo`? E se N fosse un'occorrenza di `nodo` tra quelle più interne? iv) Il predicato `match="valore"` sarebbe attivato se N fosse pari a `valore`?

Caso con P_R uguale a "`text()`". Se P_R coincide con "`text()`" e il nodo attivo N è una foglia dell'albero sorgente, contenente un valore `PCDATA`, allora P_R è soddisfatto e la regola può produrre una porzione di albero oggetto.

Esempio 50 La regola:

```
<xsl:template match="text()">
  <b>Foglia</b>
</xsl:template>
```

è attivata solo nel caso in cui N coincida con una delle occorrenze delle foglie `telefono`, `Tizio`, *etc.*, del seguente documento d'esempio:

```
<?xml version="1.0" encoding="UTF-8"?>
<nota>
 <msg>
  <sorgente>telefono</sorgente>
  <da>Tizio</da>
  <testo>Arrivero' al piu' presto</testo>
 </msg>
 <msg>
  <sorgente>e-mail</sorgente>
  <da>Caio</da>
  <testo>Non so se arrivero'</testo>
 </msg>
</nota>
```

In tal caso la regola produce sempre `<b>Foglia</b>`; altrimenti la regola non può essere usata.

Esercizio 85 (Attivazione di un predicato "`text()`".) Dato il sorgente:

```
<?xml version="1.0" encoding="UTF-8"?>
 <radice>
   <nodo>
     <valore>A</valore>
     <nodo><foglia/></nodo>
     <nodo><foglia/></nodo>
   </nodo>
 </radice>
```

e la regola:

```
<xsl:template match="text()">
  <!-- template -->
</xsl:template>
```

i) Il predicato `match="text()"` sarebbe attivato se N fosse l'occorrenza più esterna di `nodo`? ii) Il predicato `match="text()"` sarebbe attivato se N fosse la foglia `A`? iii) Il predicato `match="text()"` sarebbe attivato se N fosse `foglia`?

9.2.3 *Template*, ovvero Modello di trasformazione

Vediamo, ora, il meccanismo di uso dei *template* per produrre (porzioni di) documenti oggetto.

Il *template* di una regola è individuato dai *tag* di apertura e chiusura dell'elemento `xsl:template`.

Esempio 51 Segue una generica struttura di regola, con un generico predicato.

```
<xsl:template match="espressione che definisce il predicato">
 <!-- struttura del template -->
</xsl:template>
```

Per capire il modello comportamentale di ogni *processor* XSLT, illustriamo casi di *template* che possiamo ritenere di base.

***Template* come sequenza di caratteri.** Il *template* può essere una *sequenza arbitraria di caratteri legali*. "Legale" significa che i caratteri debbano poter esere utilizzabili in un testo XML, ovvero nel testo XSLT che contiene il *template* stesso.

Esempio 52 Supponiamo che la seconda occorrenza di "da", nel seguente documento sia il *nodo attivo*:

```
<?xml version="1.0" encoding="UTF-8"?>
<nota>
 <msg>
  <sorgente>telefono</sorgente>
  <da>Tizio</da>
  <testo>Arrivero' al piu' presto</testo>
 </msg>
 <msg>
  <sorgente>e-mail</sorgente>
  <da>Caio</da>  <!-- attivo, per ipotesi -->
  <testo>Non so se arrivero'</testo>
 </msg>
</nota>
```

La regola seguente sarebbe attivata:

```
<xsl:template match="da">
 Sequenza arbitraria
</xsl:template>
```

e, dal comportamento atteso per un *processor* XSLT, la sequenza di caratteri "Sequenza arbitraria" sarebbe il valore risultato.

Esercizio 86 (Risultato non strutturato.) Qual'è il risultato di applicare la regola:

```
<xsl:template match="*">
  Uguale per tutti!
</xsl:template>
```

al sorgente:

```
<?xml version="1.0" encoding="UTF-8"?>
 <radice>
   <nodo>
     <valore>A</valore>
     <nodo-sx><foglia/></nodo-sx>
     <nodo-dx><foglia/></nodo-dx>
   </nodo>
 </radice>
```

sotto una delle seguenti ipotesi: i) il nodo attivo sia `valore`? ii) il nodo attivo sia `nodo-sx`? iii) il nodo attivo sia `nodo-dx`?

***Template* pari a `<xsl:value-of select="."/>`** Se il *template* è "`<xsl:value-of select="."/>`", esso restituisce il valore del nodo attivo, individuato, *in ogni contesto*, dall'espressione (XPath) "`.`".

Esempio 53 Si supponga che la seconda occorrenza del nodo "`da`", nel seguente documento, sia il *nodo attivo*:

```
<?xml version="1.0" encoding="UTF-8"?>
<nota>
 <msg>
  <sorgente>telefono</sorgente>
  <da>Tizio</da>
  <testo>Arrivero' al piu' presto</testo>
 </msg>
 <msg>
  <sorgente>e-mail</sorgente>
  <da>Caio</da>  <!-- Attivo, per ipotesi -->
  <testo>Non so se arrivero'</testo>
 </msg>
</nota>
```

La regola seguente sarebbe attivata:

```
  <xsl:template match="da">
   <xsl:value-of select=".">
  </xsl:template>
```

e, come effetto dell'interpretazione di `<xsl:value-of select=".">`, produrrebbe il valore `Caio`.

Esercizio 87 (Risultato (eventualmente) strutturato.) Qual'è il risultato di applicare la regola:

```
<xsl:template match="*">
  <xsl:value-of select="."/>
</xsl:template>
```

al sorgente:

```
<?xml version="1.0" encoding="UTF-8"?>
 <radice>
   <nodo>
     <valore>A</valore>
     <nodo-sx><foglia/></nodo-sx>
     <nodo-dx><foglia/></nodo-dx>
   </nodo>
 </radice>
```

sotto una delle seguenti ipotesi:

1. il nodo attivo sia `nodo`?
2. il nodo attivo sia `valore`?
3. il nodo attivo sia `nodo-dx`?

Template `<xsl:apply-templates/>` Se "`<xsl:apply-templates/>`" è il *template* l'intero processo di interpretazione è **riattivato da capo**. Quindi, esso va inquadrato nella sua completezza, dato che conosciamo il suo comportamento, regola per regola.

9.3 Interpretazione di un documento XSLT

Sin qui, abbiamo descritto il meccanismo di attivazione, ed il relativo meccanismo di uso, del *template* di una singola regola in un documento XSLT.

In generale, ogni documento XSLT contiene più regole. Quindi, il passo successivo, è stabilire come si usano insiemi di regole.

A tal fine fissiamo alcuni elementi del discorso. Siano:

- P un *processor* XSLT,
- X un sorgente XML,
- F un documento XSLT,
- O il risultato parziale, dell'applicazione di F ad X, per mezzo di P,

da usarsi in accordo con lo schema all'inizio del Capitolo 8.

In particolare, l'applicazione di F ad X, per mezzo di P, si configura come una "**visita**" dell'**albero** sintattico di X, durante la quale occorre:

- Individuare il *nodo attivo* all'interno dell'insieme N_c dei nodi di X **candidati a diventare attivi**,
- **Attivare**, se esiste, una **singola regola** di trasformazione, tra tutte quelle il cui predicato di attivazione sia soddisfatto dal nodo attivo,

- Applicare, lungo i rami dell'albero, livello per livello, il **meccanismo ricorsivo** di trasformazione, finché non sia possibile costruire un risultato, rappresentato da un albero.

9.3.1 Funzionamento di massima del *processor* P

Supponiamo di aver fissato un insieme N_c di occorrenze di nodi del documento X da trasformare, oltre a supporre di conoscere X, F ed O. Il *processor* P, applicato ad X, F, O ed N_c si comporta come segue:

1. *Estrae* da N_c un nodo a che, per definizione, diventa quello *attivo*.
2. Cercando in F, determina la regola R il cui predicato di attivazione sia meglio soddisfatto da a.
3. Crea una porzione o di risultato,
   - eventualmente, inserendo in o parti della struttura di a,
   - eventualmente, estendendo O con l'inclusione di o,
   - eventualmente, attivando **ricorsivamente** l'intepretazione,
     (a) dopo aver riformulato l'insieme N_c di nodi correnti, affinché esso contenga (parte) dei discendenti diretti di a,
     (b) "saltando" al passo 1.

Commento 18
Un ciclo di funzionamento ricorsivo del *processor* P, appena descritto, è rappresentato in **Figura 9.1**.

- Per ipotesi, si parte con un insieme iniziale N_c arbitrario di nodi candidati a diventare attivi, ovvero, con un insieme arbitrario da cui P comincia la trasformazione.

  Allo stesso modo, supponiamo che l'albero oggetto O sia anch'esso arrivato ad un punto di costruzione arbitrario;
- Il secondo passo in Figura 9.1 estrae da N_c il nodo attivo a;
- Per ipotesi, nel terzo passo, R_1 non è attivata da a, ma R_2 lo è;
- L'attivazione di R_2 produce un risultato parziale o, con cui estendere O. L'estensione è indicato col simbolo "+" in "O+o". Il significato è che o viene "aggiunto in modo opportuno" ad O; non è da intendersi come somma aritmetica;
- Nell'ultimo passo, evidenziato in Figura 9.1, l'insieme N_c ingloba sia i successori diretti dell'ex nodo attivo a, sia i nodi candidati precedenti tra i quali a è stato scelto.

La descrizione di massima del procedimento di interpretazione, appena completata, lascia alcuni dettagli irrisolti: ad esempio, la determinazione del nodo attivo. Tali dettagli possono essere studiati attraverso un'indagine sperimentale sul funzionamento di un *processor* XSLT, che sviluppiamo qui di seguito.

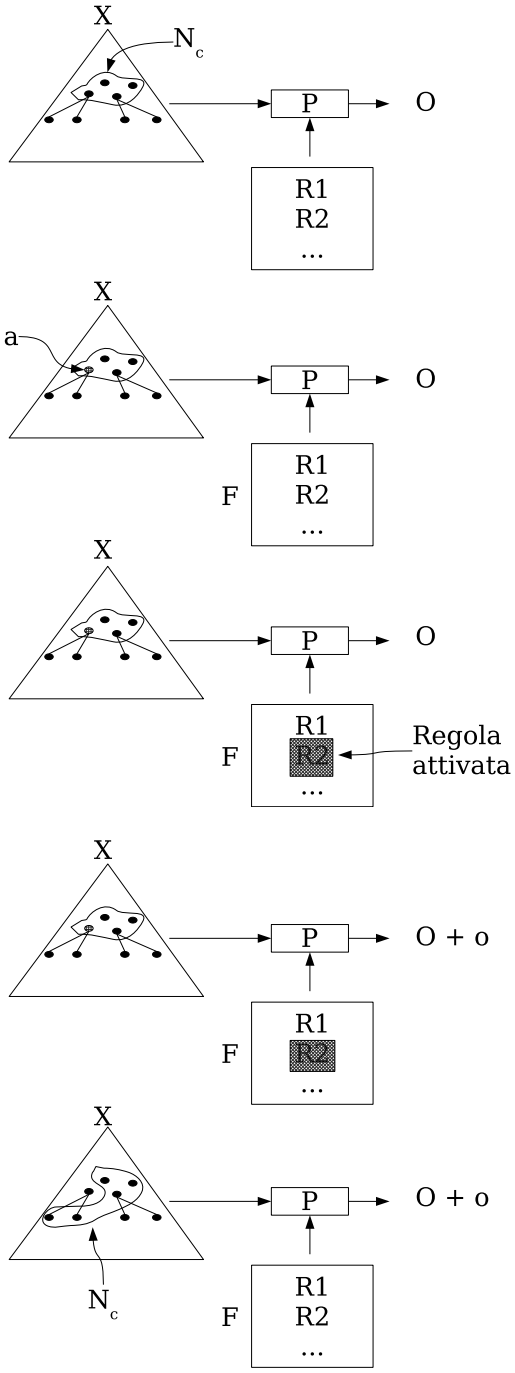


Figura 9.1: Funzionamento di un ciclo di un *processor* XSLT

9.4 Indagine sperimentale sull'interpretazione

È possibile un'indagine sperimentale del comportamento di un *processor* XSLT, applicandolo ad alcuni documenti di riferimento.

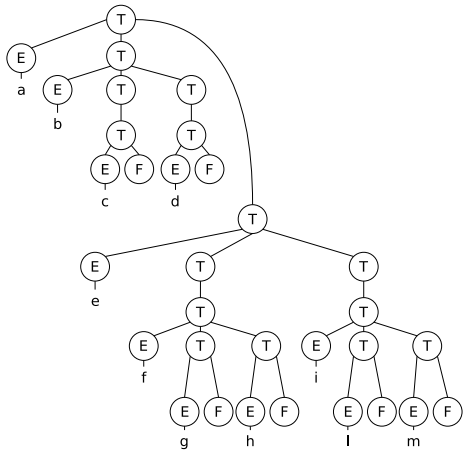
Il primo documento di riferimento è `albero-binario.xml`. Esso gioca il ruolo di un generico documento XML X:

```
<?xml version="1.0"?>
<T>
 <E>a</E>
 <T>
  <E>b</E>
  <T> <E>c</E> <F/></T>
  <T> <E>d</E> <F/></T>
 </T>
 <T>
  <E>e</E>
  <T>
   <E>f</E>
   <T> <E>g</E> <F/></T>
   <T> <E>h</E> <F/></T>
  </T>
  <T>
   <E>i</E>
   <T> <E>l</E> <F/></T>
   <T> <E>m</E> <F/></T>
  </T>
 </T>
</T>
```

`albero-binario.xml` appartiene alla semantica `albero-binario.dtd`:

```
<!ELEMENT T ((E , T , T) | (E , F))>
<!ELEMENT F EMPTY>
<!ELEMENT E (#PCDATA)>
```

Il corrispondente **albero binario** è:



Per definizione, un **albero binario** è un albero nel quale *ogni nodo* ha *al più* due discendenti diretti.

Il secondo documento di riferimento è `visita-canonica.xsl`. Esso gioca il ruolo del generico documento XSLT F, che contiene le **regole base** che devono essere **interpretate da un qualsiasi** *processor* XSLT P:

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

     <xsl:template match="/"> <!-- R1 -->
       <xsl:apply-templates/>
     </xsl:template>

     <xsl:template match="*"> <!-- R2 -->
       <xsl:apply-templates/>
     </xsl:template>

     <xsl:template match="text()"> <!-- R3 -->
       <xsl:value-of select="."/>
     </xsl:template>

    </xsl:stylesheet>
```

9.4.1 Visita in preordine

Illustriamo il *comportamento standard* di interpreti XSLT, applicando il documento `visita-canonica.xsl` al documento `albero-binario.xml`.

Ciascun *processor* XSLT, *in assenza di altre indicazioni*, realizza una visita in *preordine* di qualsiasi albero sintattico sorgente cui sia applicato.

Nel caso in esame, applicare il documento `visita-canonica.xsl` al sorgente `albero-binario.xml`, per mezzo di un *processor* XSLT, produce la sequenza `abcdefghilm`, che appunto, corrisponde alla visita in preordine di `albero-binario.xml`.

Dato un qualsiasi albero T, per definizione, la **visita in preordine** di T compie i seguenti passi:

1. Se T è una foglia, allora la visita ne "legge" il contenuto ed, eventualmente, lo usa;

2. Se T è composto da un elemento informazione I e da un certo numero di sottoalberi $T_1, \ldots, T_n$, elencati a partire da quello più a sinistra, la visita, "legge" I, quindi continua prima su T_1, poi su T_2, fino a T_n.

Esercizio 88 (Verifica sperimentale della visita in preordine.) Applicare `visita-canonica.xsl` a `albero-binario.xml` e verificare il comportamento del *processor* XSLT di XMLCopy.

9.4.2 Simulazione della visita

Il comportamento del *processor* XSLT, che, in mancanza di altri vincoli, sviluppa una visita in preordine degli alberi sintattici sorgente cui è applicato, può essere simulato, evidenziando, passo per passo, le seguenti informazioni:

- N_c, l'insieme dei **nodi candidati** a diventare **nodo attivo**,

- a, il **nodo attivo**,

- R, la regola attivata dal nodo attivo che ne soddisfa il predicato di attivazione,

- o la porzione di struttura del risultato da integrare nel risultato finale O.

Una tabella con un numero di colonne sufficiente a contenere le informazioni elencate per la simulazione ne permette una rappresentazione conveniente:

Ordine	**Azioni**	N_c	a	**Regola**	O
1	def. nodi candidati	`?xml...`			$\emptyset$
2	def. nodo attivo	$\emptyset$	`?xml...`		$\emptyset$
3	att. regola	$\emptyset$	`?xml...`	`R1`	$\emptyset$
4	output	$\emptyset$	`?xml...`	`R1`	`<?xml...>`
5	ricorsione	$\emptyset$	`?xml...`	`R1`	`<?xml...>`
6	def. nodi candidati	`T`			`<?xml...>`
7	def. nodo attivo	$\emptyset$	`T`		`<?xml...>`
8	att. regola	$\emptyset$	`T`	`R2`	`<?xml...>`
9	output	$\emptyset$	`T`	`R2`	`<?xml...>`
10	ricorsione	$\emptyset$	`T`	`R2`	`<?xml...>`
11	def. nodi candidati	`E, T, T`			`<?xml...>`
12	def. nodo attivo	`T, T`	`E`		`<?xml...>`
13	att. regola	`T, T`	`E`	`R2`	`<?xml...>`
14	output	`T, T`	`E`	`R2`	`<?xml...>`
15	ricorsione	`T, T`	`E`	`R2`	`<?xml...>`
16	def. nodi candidati	`a, T, T`			`<?xml...>`
17	def. nodo attivo	`T, T`	`a`		`<?xml...>`
18	att. regola	`T, T`	`a`	`R3`	`<?xml...>`
19	output	`T, T`	`a`	`R3`	`<?xml...> a`
20	ricorsione	`T, T`	`a`	`R3`	`<?xml...> a`
21	def. nodi candidati	`T, T`			`<?xml...> a`
22	def. nodo attivo	`T`	`T`		`<?xml...> a`
23	att. regola	...	...		...

Commento 19

- Le transizioni tra le coppie di passi (6,7), (10,11) e (15,16) aggiornano l'insieme N_c.
- Nella transizione dal passo 15 al 16 la foglia `a` è inserita nell'insieme di **nodi candidati** già presenti in N_c.
- Nella transizione dal passo 16 al 17 la foglia `a` diventa il nodo attivo evidenziando il punto in cui la visita diventa in preordine.
- Nella transizione dal passo 18 al 19 la foglia `a` viene aggiunta al documento di uscita.

Esercizio 89 (Previsione risultati.) Sia dato il documento XSLT seguente:

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

```
<xsl:template match="/"> <!-- R1 -->
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="*"> <!-- R2 -->
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="text()"> <!-- R3 -->
  <xsl:value-of select="."/>
</xsl:template>

</xsl:stylesheet>
```

1. Prevedere il risultato dell'interpretazione del seguente documento XML:

```
<?xml version="1.0"?>
<!-- Il nodo <E> segue i due sottoalberi
     di ogni nodo interno -->
<T>
 <T>
  <T> <E>c</E> <F/></T>
  <T> <E>d</E> <F/></T>
  <E>b</E>
 </T>
 <T>
  <T>
   <T> <E>g</E> <F/></T>
   <T> <E>h</E> <F/></T>
   <E>f</E>
  </T>
  <T>
   <T> <E>l</E> <F/></T>
   <T> <E>m</E> <F/></T>
   <E>i</E>
  </T>
  <E>e</E>
 </T>
 <E>a</E>
</T>
```

2. Prevedere il risultato dell'interpretazione del seguente documento XML:

```
<?xml version="1.0"?>
<!-- Il nodo <E> e' tra i due sottoalberi
     di ogni nodo interno -->
```

```
<T>
 <T>
  <T> <E>c</E> <F/></T>
  <E>b</E>
  <T> <E>d</E> <F/></T>
 </T>
 <E>a</E>
 <T>
  <T>
   <T> <E>g</E> <F/></T>
   <E>f</E>
   <T> <E>h</E> <F/></T>
  </T>
  <E>e</E>
  <T>
   <T> <E>l</E> <F/></T>
   <E>i</E>
   <T> <E>m</E> <F/></T>
  </T>
 </T>
</T>
```

Commento 20
Ovviamente, il risultato è ricavabile per mezzo del *processor* XSLT in XMLCopy.

L'interesse dell'esercizio sta proprio nell'anticiparne il comportamento, simulandolo.

9.5 Dove siamo

Abbiamo imparato cosa significhi scrivere un documento XSLT. Esso contiene un elenco di regole il cui scopo è trasformare un documento sorgente XML in un documento oggetto; quest'ultimo, nel caso si voglia abbia in una veste tipografica gradevole, potrà essere un documento XHTML.

Capitolo 10

Algoritmi

Impareremo a scrivere algoritmi per mezzo di documenti XSLT con lo scopo di interpretare tipograficamente documenti XML.
In particolare:

- Un algoritmo sarà ottenuto **limitando** opportunamente la **strategia** di visita di *default*, quella in preordine, che trasforma l'**intero** albero sintattico di un documento XML sorgente in un documento oggetto.
- Le limitazioni sulla strategia di visita consisteranno nel controllo della definizione dell'insieme di nodi candidati a diventare attivi, permettendo anche visita e valutazione dei nodi attributo.

10.1 La visita di *default*

- La strategia di *default* di visita un albero sintattico, da parte di un interprete XSLT, avviene in preordine ed è guidata dall'interpretazione del seguente documento XSLT:

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

     <!-- Attivata dalla radice del documento XML sorgente -->
     <xsl:template match="/">
       <xsl:apply-templates/>
     </xsl:template>

     <!-- Attivata da un qualsiasi nodo
          del documento XML sorgente
          che non sia un nodo testo -->
     <xsl:template match="*">
       <xsl:apply-templates/>
     </xsl:template>

     <!-- Attivata dai nodi foglia -->
     <xsl:template match="text()">
       <xsl:value-of select="."/>
     </xsl:template>

    </xsl:stylesheet>
```

  in cui:

  – **ciascun nodo** del sorgente è **coinvolto** dalla visita,
  – `<xsl:apply-templates/>` attiva **ricorsivamente** la visita,

– l'attivazione ricorsiva della visita **aggiorna l'insieme dei nodi candidati** a diventare nodo attivo,
– l'aggiornamento dei nodi candidati avviene tramite l'inclusione di **tutti *i discendenti immediati*** dell'ultimo nodo attivo.

10.2 Modifica della visita di *default*

La **modifica** della strategia di **visita di *default*** si ottiene:

- **scegliendo quali discendenti** dell'ultimo nodo attivo includere nell'insieme di nodi candidati,
- **potendo includere nodi attributo** nell'insieme di nodi candidati a diventare attivi.

Analizzeremo qui di seguito gli strumenti sintattici per scegliere i discendenti e decidere la, eventuale, inclusione dei nodi attributo nel processo d'interpretazione.

10.2.1 Scelta dei discendenti

Per ipotesi, in un documento XML sorgente X, siano:

- a il nodo attivo,
- N_c l'insieme dei nodi candidati a diventare attivi, a escluso.

L'interpretazione di `<xsl:apply-templates/>` attiva la visita ricorsiva, **includendo in N_c l'insieme di tutti i discendenti diretti di a.**

Invece:

`<xsl:apply-templates select="nome-nodo"/>`
estende N_c con le sole occorrenze di `nome-nodo`, prese tra i *discendenti diretti* di a, ammesso che ne esistano.

Commento 21
In generale, all'attributo `select` può essere associata un'espressione arbitraria di un linguaggio XPath, di cui introdurremo gli aspettti principali, che individua, essenzialmente, un qualsiasi insieme di nodi del documento X.

10.2.2 Esempio dettagliato di scelta dei discendenti diretti

L'esempio si basa sul documento di riferimento `alberi-arbitrari.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE A [
 <!ELEMENT A (D,(U,V)*)>
```

```
 <!ELEMENT D (#PCDATA)>
 <!ELEMENT U (A)>
 <!ELEMENT V (A)>
]>

<A>
 <!-- Dato -->
 <D>0</D>
 <!-- I coppia di sotto alberi -->
 <U> <A><D>1U</D></A> </U>
 <V> <A><D>1V</D></A> </V>
 <!-- II coppia di sotto alberi -->
 <U> <A><D>2U</D></A> </U>
 <V> <A><D>2V</D></A> </V>
</A>
```

la cui **semantica formalizza** la seguente idea sulla struttura degli alberi:

1. un albero `A` contiene un dato `D` ed una sequenza arbitraria di coppie di componenti `U` e `V`,
2. ciascuna delle componenti `U` e `V` è anch'essa un albero,
3. ciascun dato è un elemento non strutturato.

Un algoritmo di visita parziale

Il seguente algoritmo `alberi-arbitrari.xsl` visita i nodi dati ed i soli **sotto-alberi con radice elemento** `U` del documento `alberi-arbitrari.xml`:

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

  <xsl:template match="/"> <!-- R1 -->
    <xsl:apply-templates/>  <!-- Ric1 -->
  </xsl:template>

  <xsl:template match="A">           <!-- R2 -->
    <xsl:apply-templates select="D"/> <!-- Ric2.D -->
    <xsl:apply-templates select="U"/> <!-- Ric2.U -->
  </xsl:template>

  <xsl:template match="D"> <!-- R3 -->
    <xsl:apply-templates/>  <!-- Ric3 -->
  </xsl:template>
```

```
<xsl:template match="U"> <!-- R4 -->
  <xsl:apply-templates/>  <!-- Ric4 -->
</xsl:template>

<xsl:template match="text()">  <!-- R5 -->
  <xsl:value-of select="."/>    <!-- V5 -->
</xsl:template>

</xsl:stylesheet>
```

Commento 22

- La regola `R1` è soddisfatta dal solo nodo radice del documento sorgente.

  Non produce alcunché da inserire nel documento oggetto.

  Si limita ad attivare la chiamata ricorsiva sull'insieme dei discendenti diretti della radice del documento oggetto, ovvero sull'unico nodo `A`.

- La regola `R2`, quando il nodo attivo è `A`, attiva la ricorsione in "due tempi", inserendo nell'insieme dei nodi candidati solo due dei **sotto**-insiemi possibili di **tutti** i discendenti diretti del nodo `A` stesso.

  - Il "primo tempo" della ricorsione è attivata **solo** sui nodi `D`, discendenti diretti di `A`.
    Questo comportamento si ottiene assegnando il valore `D` all'attributo di selezione `select` del nodo `xsl:apply-templates`.
  - Il "secondo tempo" della ricorsione è attivata **solo** sui nodi `U`, discendenti diretti di `A`.
    Questo comportamento si ottiene al termine della prima ricorsione, assegnando il valore `U`, in `xsl:apply-templates`, all'attributo `select`.

- I nodi `V`, discendenti diretti di `A`, non saranno mai inseriti nell'insieme dei nodi candidati, proprio perché esplicitamente esclusi dalle scelte precedenti.

- Le regole `R3` ed `R4` sono soddisfatte dalle sole occorrenze dei nodi `D` e `U`, rispettivamente.

  Non producono alcunché da inserire nel documento oggetto.

  Si limitano ad attivare la chiamata ricorsiva sull'insieme dei discendenti diretti dei nodi `D` e `U`.

- Nel caso `D`, l'unico discendente è una foglia che attiva la regola `R5` che ne restituisce il valore.

- Nel caso `U`, l'unico discendente è un nodo `A`, e la regola attivata sarà `R2`, **ricominciando il processo** descritto sin qui, su uno dei sotto-alberi sorgente.

- Il documento oggetto prodotto conterrà la sequenza di caratteri `0 1U 2U`.

Simulazione dell'algoritmo di visita parziale

La Figura 10.1 raccoglie la simulazione, passo dopo passo, dell'algoritmo di visita parziale, introdotto nella sezione precedente.

- La simulazione evidenzia il comportamento ciclico di un interprete XSLT:
  1. determinazione del nodo attivo,
  2. determinazione della regola attivata,
  3. applicazione del *template* per produrre la porzione di *output*,
  4. aggiornamento dell'insieme di nodi candidati, in caso di attivazione ricorsiva dell'interpretazione.
- Il passo 10 "**ricorda**" la **sequenza di azioni** `Ric2.D` **e** `Ric2.U`, corrispondenti alle due attivazioni della ricorsione che costituiscono il contenuto del template della regola `R2`.
- L'attivazione della **ricorsione** `Ric2.U` **è lasciata in sospeso** finché la valutazione dell'intero sotto albero `<A><D>1U</D></A>` non si conclude con la produzione del risultato (parziale) `1U`.
- L'**attivazione della ricorsione** `Ric2.U` **riprende al passo 20** in seguito al quale l'insieme di nodi candidati N_c è ripristinato a contenere le radici dei sotto alberi la cui valutazione è stata lasciata in sospeso al passo 10.

Esercizio 90 (Modifiche all'algoritmo `alberi-arbitrari.xsl`.) Modificare l'algoritmo `alberi-arbitrari.xsl` affinché il documento oggetto, prodotto dalla sua applicazione a `alberi-arbitrari.xml`, soddisfi uno dei seguenti requisiti:

1. Il documento oggetto contenga la sequenza `0 1V 2V`.
2. Il documento oggetto contenga una occorrenza della sequenza "`elementoV`", per ogni occorrenza dell'elemento `V` nel sorgente.
3. Il documento oggetto appartenga alla semantica XHTML ed abbia una interpretazione tipografica ragionevole. Ad esempio, l'interpretazione della "versione" XHTML di `alberi-arbitrari.xml` potrebbe produrre il testo:

```
<html>
  <head>
  </head>
  <body>
    <b>a b c d e f g h i l m </b>
  </body>
</html>
```

Commento 23
Le soluzioni proposte per ciascuno dei punti dell'esercizio precedente devono produrre il risultato corretto per qualsiasi documento sorgente nella semantica cui `alberi-arbitrari.xsl` appartiene, e non solo se applicati ad `alberi-arbitrari.xsl`.

Ordine	**Azioni**	N_c	a	**Regola**	O
1	def. nodi candidati	`?xml...`			$\emptyset$
2	def. nodo attivo	$\emptyset$	`?xml...`		$\emptyset$
3	att. regola	$\emptyset$	`?xml...`	`R1`	$\emptyset$
4	output	$\emptyset$	`?xml...`	`R1`	$\emptyset$
5	`Ric1`	$\emptyset$	`?xml...`	`R1`	$\emptyset$
6	def. nodi candidati	`A`			$\emptyset$
7	def. nodo attivo	$\emptyset$	`A`		$\emptyset$
8	att. regola	$\emptyset$	`A`	`R2`	$\emptyset$
9	output	$\emptyset$	`A`	`R2`	$\emptyset$
10	`Ric2.D`, `Ric2.U`	$\emptyset$	`A`	`R2`	$\emptyset$
11	def. nodi candidati, `Ric2.U`	`D`			$\emptyset$
12	def. nodo attivo, `Ric2.U`	$\emptyset$	`D`		$\emptyset$
13	att. regola, `Ric2.U`	$\emptyset$	`D`	`R3`	$\emptyset$
14	output, `Ric2.U`	$\emptyset$	`D`	`R3`	$\emptyset$
15	`Ric3`, `Ric2.U`	$\emptyset$	`D`	`R3`	$\emptyset$
16	def. nodi candidati, `Ric2.U`	`O`			$\emptyset$
17	def. nodo attivo, `Ric2.U`	$\emptyset$	`O`		$\emptyset$
18	att. regola, `Ric2.U`	$\emptyset$	`O`	`R5`	$\emptyset$
19	output, `Ric2.U`	$\emptyset$	`O`	`R5`	`O`
20	`Ric2.U`	$\emptyset$	`O`	`R5`	`O`
21	def. nodi candidati	`U`, `U`			`O`
22	def. nodo attivo	`U`	`U`		`O`
23	att. regola	`U`	`U`	`R4`	`O`
24	output	`U`	`U`	`R4`	`O`
25	`Ric4`	`U`	`U`	`R4`	`O`
26	def. nodi candidati	`A`, `U`			`O`
27	def. nodo attivo	`U`	`A`		`O`
28	att. regola	`U`	`A`	`R2`	`O`
29	output	`U`	`A`	`R2`	`O`
30	`Ric2.D`, `Ric2.U`	`U`	`A`	`R2`	`O`
31	def. nodi candidati, `Ric2.U`	`D`, `U`			`O`
32	def. nodo attivo, `Ric2.U`	`U`	`D`		`O`
33	att. regola, `Ric2.U`	`U`	`D`	`R3`	`O`
34	output, `Ric2.U`	`U`	`D`	`R3`	`O`
35	`Ric3`, `Ric2.U`	`U`	`D`	`R3`	`O`
36	def. nodi candidati, `Ric2.U`	`1U`, `U`			`O`
37	def. nodo attivo, `Ric2.U`	`U`	`1U`		`O`
38	att. regola, `Ric2.U`	`U`	`1U`	`R5`	`O`
39	output, `Ric2.U`	`U`	`1U`	`R5`	`O 1U`
40	`Ric2.U`	`U`	`1U`	`R5`	`O 1U`
41	def. nodi candidati	`U`			`O 1U`
42	def. nodo attivo	$\emptyset$	`U`		`O 1U`
43	att. regola	$\emptyset$	`U`	`R4`	`O 1U`
44	output	...	...	...	...

Figura 10.1: Simulazione dell'algoritmo di visita parziale

10.2.3 Visita e valutazione dei nodi attributo

Ci concentreremo ora sul problema di visitare anche gli attributi dei nodi di un documento XML sorgente X.

Per ipotesi, supponiamo di aver individuato in X i seguenti elementi:

- a il nodo attivo,
- N_c l'insieme dei nodi candidati, senza a.

L'interpretazione di `<xsl:apply-templates/>` attiva la visita ricorsiva, **includendo in N_c l'insieme di tutti i discendenti diretti di a.**

Invece:

`<xsl:apply-templates select="@nome-attributo"/>`
estende N_c con il nodo attributo `nome-attributo` del nodo attivo a, ammesso che esista.

Commento 24
In generale, all'espressione valore di `select` può essere associata un'espressione arbitraria di un linguaggio XPath che individua insiemi (essenzialmente) qualsiasi di attributi di nodi del documento X.

10.2.4 Esempio dettagliato di visita di nodi attributo

Usiamo il documento `alberi-arbitrari-con-attributo.xml` di riferimento:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE A [
 <!ELEMENT A (D,(U,V)*)>
 <!ELEMENT D EMPTY>
  <!ATTLIST D w CDATA #REQUIRED>
 <!ELEMENT U (A)>
 <!ELEMENT V (A)>
]>

<A>
 <!-- Dato -->
 <D w="0"/>
 <!-- I coppia di sotto alberi -->
 <U> <A><D w="1U"/></A> </U>
 <V> <A><D w="1V"/></A> </V>
 <!-- II coppia di sotto alberi -->
 <U> <A><D w="2U"/></A> </U>
 <V> <A><D w="2V"/></A> </V>
</A>
```

la cui semantica formalizza la seguente idea sulla struttura degli alberi:

1. un albero `A` contiene un dato `D` ed una sequenza arbitraria di coppie di componenti `U` e `V`,
2. ciascuna delle componenti `U` e `V` è anch'essa un albero,
3. ciascun dato è caratterizzato dall'esistenza di un attributo che contiene il valore effettivo del dato stesso.

Un algoritmo di visita parziale che legge un attributo

Il seguente algoritmo `alberi-arbitrari-con-attributo.xsl` **visita l'attributo di ciascuna occorrenza dei nodi** `D` ed i soli sotto-alberi con radice elemento `U` del documento `alberi-arbitrari-con-attributo.xml`:

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

  <xsl:template match="/"> <!-- R1 -->
   <xsl:apply-templates/>   <!-- Ric1 -->
  </xsl:template>

  <xsl:template match="A">          <!-- R2 -->
   <xsl:apply-templates select="D"/> <!-- Ric2.D -->
   <xsl:apply-templates select="U"/> <!-- Ric2.U -->
  </xsl:template>

  <xsl:template match="D">           <!-- R3 -->
   <xsl:apply-templates select="@w"/> <!-- Ric3 -->
  </xsl:template>

  <xsl:template match="U"> <!-- R4 -->
   <xsl:apply-templates/>   <!-- Ric4 -->
  </xsl:template>

  <xsl:template match="@w">  <!-- R5 -->
   <xsl:value-of select="."/>
  </xsl:template>

 </xsl:stylesheet>
```

Commento 25

- La regola `R1` è soddisfatta dal solo nodo radice del documento sorgente.

  Non produce alcunché da inserire nel documento oggetto.

  Si limita ad attivare la chiamata ricorsiva sull'insieme dei discendenti diretti della radice del documento oggetto, ovvero sull'unico nodo `A`.

- La regola `R2`, quando il nodo attivo è `A`, attiva la ricorsione in "due tempi", inserendo nell'insieme dei nodi candidati solo due sotto-insiemi dei discendenti diretti del nodo `A` stesso.
  - Il "primo tempo" della ricorsione è attivata **solo** sui nodi `D`, discendenti diretti di `A`.

    Questo comportamento si ottiene assegnando il valore `D` all'attributo di selezione `select` del nodo `xsl:apply-templates`.
  - Il "secondo tempo" della ricorsione è attivata **solo** sui nodi `U`, discendenti diretti di `A`.

    Questo comportamento si ottiene al termine della prima ricorsione, assegnando il valore `U`, in `xsl:apply-templates`, all'attributo `select`.
- I nodi `V`, discendenti diretti di `A`, non saranno mai inseriti nell'insieme dei nodi candidati, proprio perché esplicitamente esclusi dalle scelte precedenti.
- Le regole `R3` ed `R4` sono soddisfatte dalle sole occorrenze dei nodi `D` e `U`, rispettivamente.

  Non producono alcunché da inserire nel documento oggetto.

  Ciascuna di esse si limita ad attivare una ricorsione.
- Nel caso `D`, la ricorsione **non** viene attivata sull'insieme dei discendeti diretti di `D` [1].

  Invece, l'**attivazione** della ricorsione avviene **sull'attributo** del nodo `D`.

  La sintassi dell'espressione di selezione usa il nome dell'attributo da visitare, preceduto dal simbolo "`@`".

  Ne consegue l'attivazione della regola `R5`, che valuta il contenuto dell'attributo `w`, per mezzo dell'elemento `<xsl:value-of select="."/>`.
- Nel caso `U`, l'unico discendente è un nodo `A`, e la regola attivata sarà `R2`.
- Il documento oggetto prodotto conterrà la sequenza di caratteri `0 1U 2U`.

Simulazione dell'algoritmo di visita parziale anche su attributi

La Figura 10.2 raccoglie la simulazione, passo dopo passo, dell'algoritmo di visita parziale, che coinvolge anche nodi attributo, introdotto alla sezione precedente.

- La simulazione evidenzia il comportamento ciclico di un interprete XSLT:
  1. determinazione del nodo attivo,

[1] Per inciso, una tale attivazione non avrebbe alcun effetto sul documento oggetto, considerato che ogni occorrenza di `D` non ha discendenti diretti.

Ordine	**Azioni**	N_c	a	**Regola**	O
1	def. nodi candidati	?xml...			∅
2	def. nodo attivo	∅	?xml...		∅
3	att. regola	∅	?xml...	R1	∅
4	output	∅	?xml...	R1	∅
5	Ric1	∅	?xml...	R1	∅
6	def. nodi candidati	A			∅
7	def. nodo attivo	∅	A		∅
8	att. regola	∅	A	R2	∅
9	output	∅	A	R2	∅
10	Ric2.D, Ric2.U	∅	A	R2	∅
11	def. nodi candidati, Ric2.U	D			∅
12	def. nodo attivo, Ric2.U	∅	D		∅
13	att. regola, Ric2.U	∅	D	R3	∅
14	output, Ric2.U	∅	D	R3	∅
15	Ric3, Ric2.U	∅	D	R3	∅
16	def. nodi candidati, Ric2.U	w			∅
17	def. nodo attivo, Ric2.U	∅	w		∅
18	att. regola, Ric2.U	∅	w	R5	∅
19	output, Ric2.U	∅	w	R5	0
20	Ric2.U	∅	w	R5	0
21	def. nodi candidati	U, U			0
22	def. nodo attivo	U	U		0
23	att. regola	U	U	R4	0
24	output	U	U	R4	0
25	Ric4	U	U	R4	0
26	def. nodi candidati	A, U			0
27	def. nodo attivo	U	A		0
28	att. regola	U	A	R2	0
29	output	U	A	R2	0
30	Ric2.D, Ric2.U	U	A	R2	0
31	def. nodi candidati, Ric2.U	D, U			0
32	def. nodo attivo, Ric2.U	U	D		0
33	att. regola, Ric2.U	U	D	R3	0
34	output, Ric2.U	U	D	R3	0
35	Ric3, Ric2.U	U	D	R3	0
36	def. nodi candidati, Ric2.U	w, U			0
37	def. nodo attivo, Ric2.U	U	w		0
38	att. regola, Ric2.U	U	w	R5	0
39	output, Ric2.U	U	w	R5	0 1U
40	Ric2.U	U	w	R5	0 1U
41	def. nodi candidati	U			0 1U
42	def. nodo attivo	∅	U		0 1U
43	att. regola	∅	U	R4	0 1U
44	output	...	...	...	...

Figura 10.2: Simulazione dell'algoritmo di visita parziale anche su attributi

2. determinazione della regola attivata,
3. applicazione del *template* per produrre la porzione di *output*,
4. aggiornamento dell'insieme di nodi candidati, in caso di attivazione ricorsiva dell'interpretazione.

- Il passo 10 "**ricorda**" **la sequenza di azioni** `Ric2.D` **e** `Ric2.U`, corrispondenti alle due attivazioni della ricorsione che costituiscono il contenuto del template della regola `R2`.
- Il passo 16 inserisce il nodo attributo `w` nell'insieme N_c dei nodi candidati all'attivazione.
- L'attivazione della **ricorsione** `Ric2.U` **è lasciata in sospeso** finché la valutazione dell'intero sotto albero `<A><D>1U</D></A>` non è conclusa, inserendo il risultato (parziale) `1U`.
- L'attivazione della **ricorsione** `Ric2.U` **riprende al passo 20** in seguito al quale l'insieme di nodi candidati N_c è ripristinato a contenere le radici dei sotto alberi la cui valutazione è stata lasciata in sospeso al passo 10.

Esercizio 91 (Modifiche a `alberi-arbitrari-con-attributo.xsl`.) Modificare l'algoritmo `alberi-arbitrari-con-attributo.xsl` affinché la sua applicazione a `alberi-arbitrari-con-attributo.xml` produca il seguente documento XHTML:

```
<html>
   <header/>
   <body>
      <table>
         <tr>
            <td>0</td>
            <td>
               <table>
                  <tr>
                     <td>1U</td>
                  </tr>
               </table>
            </td>
            <td>
               <table>
                  <tr>
                     <td>2U</td>
                  </tr>
               </table>
            </td>
            <td>
               <table>
```

```
                  <tr>
                     <td>1V</td>
                  </tr>
               </table>
            </td>
            <td>
               <table>
                  <tr>
                     <td>2V</td>
                  </tr>
               </table>
            </td>
         </tr>
      </table>
   </body>
</html>
```

Commento 26

- La soluzione proposta deve produrre il risultato corretto per qualsiasi documento sorgente che appartenga alla semantica per cui l'algoritmo `alberi-arbitrari-con-attributi.xsl` è stato scritto;
- Notare che l'annidamento delle tabelle ricomincia in corrispondenza della radice di ogni sotto-albero del documento sorgente.

10.3 Definizione di valore di elementi

Scopo della sezione è definire ad un livello più formale il concetto di **valore di un nodo**, o **valore di un sotto-albero** di un documento XML, finora utilizzato informalmente.

10.3.1 Legenda per calcolo del valore di un nodo

Il valore di un nodo dipende dalla sua natura:

- Il valore di un nodo **testo**, che soddisfa il predicato `text()`, coincide con la sequenza di caratteri che compongono il testo stesso.
- Il valore di un nodo **elemento** è la sequenza di caratteri, ottenuta concatenando, ricorsivamente, i valori degli elementi discendenti.

  La concatenazione non contiene:

  – il nome dei *tag* attraversati,
  – il contenuto dei nodi commento.
- Il valore di un nodo **attributo** `nome-attributo="espressione-valore"` associa a `nome-attributo` il valore delle componenti `espressione-valore`, private dei caratteri vuoti (spazi) iniziali e finali.

- Il valore di un nodo **commento** è la sequenza di caratteri tra `<!--` e `-->`, escludendo sia `<!--` che `-->`.

10.3.2 Valutazione del valore di elementi

- Il *tag* XSLT che restituisce il valore del nodo attivo è:

```
<xsl:value-of select="."/>
```

- L'espressione ".", valore dell'attributo `select`, è deputata ad individuare il nodo attivo.

- Il risultato fornito dalla valutazione del nodo `xsl:value-of` di una porzione dell'albero sintattico del sorgente viene copiato nel documento oggetto.

10.3.3 Esempi di valutazione

Il testo XML di riferimento per gli esempi è il sorgente `tavola-periodica.xml` seguente:

```
<?xml version="1.0"?>
<tavola-periodica> <!-- Frammento di tavola periodica -->
  <atomo stato=" GAS ">
    <nome>Idrogeno</nome>
    <simbolo>H</simbolo>
    <numero-atomico>1</numero-atomico>
    <peso-atomico>1.00794</peso-atomico>
    <punto-ebollizione unita="Kelvin">20.28</punto-ebollizione>
    <punto-fusione unita="Kelvin">13.81</punto-fusione>
    <densita unita="gr/cm cubo">
      0.0000899
    </densita>
  </atomo>
  <atomo stato=" GAS ">
    <nome>Elio</nome>
    <simbolo>He</simbolo>
    <numero-atomico>2</numero-atomico>
    <peso-atomico>4.0026</peso-atomico>
    <punto-ebollizione unita="Kelvin">4.216</punto-ebollizione>
    <punto-fusione unita="Kelvin">0.95</punto-fusione>
    <densita unita="grams/cubic centimeter">
      0.0001785
    </densita>
  </atomo>
</tavola-periodica>
```

Esempio 54 (Valutazione di un sotto-albero `atomo`.) Il valore della prima occorrenza di `atomo` in `tavola-periodica.xml`, che include "spazi" e "ritorni a capo", è:

```
Idrogeno
H
1
1.00794
20.28
13.81


  0.0000899
```

Esercizio 92 (Lista dei valori di `nome`.) Scrivere un documento XSLT che, applicato a `tavola-periodica.xml`, elenchi la lista dei valori dei nodi `nome` in un documento XHTML.

Esempio 55 (Valutazione dell'unico commento.) Il valore dell'unico nodo commento nel sorgente è: `Frammento di tavola periodica`.

Commento 27
`comment()` è il predicato di attivazione soddisfatto da un nodo commento, da usare analogamente al predicato `text()`:

```
<xsl:template match="comment()">
  <!-- template -->
</xsl:template>
```

Esercizio 93 (Valutazione di un commento.) Scrivere un documento XSLT che, applicato a `tavola-periodica.xml` produca un oggetto XHTML la cui interpretazione sia:

Nota: Il titolo deve essere il risultato della valutazione del nodo commento del sorgente.

Esempio 56 Nel caso il nodo attivo sia l'attributo di una delle due occorrenze di `atomo`, la regola:

```
<xsl:template match="@stato">
 <xsl:value-of select=".">
</xsl:template>
```

produce il valore `GAS`, ovvero una sequenza di tre caratteri, e non di cinque, che includerebbero i due "spazi" antecedente e conseguente `GAS`.

Esercizio 94 (Verifica sulla valutazione di attributi.) Scrivere un documento XSLT che, applicato a `tavola-periodica.xml`, produca un oggetto XHTML la cui interpretazione sia:

Nota: I tratti orizzontali che precedono e seguono il contentuto delle celle `Stato` hanno lo scopo di evidenziare l'eliminazione degli spazi.

10.4 Dove siamo

Siamo al punto in cui possiamo cominciare ad automatizzare le attività di gestione di cv, da parte di `Trova l'impiegato`, una volta fissata una opportuna semantica per i cv stessi. In particolare, con una piccola estensione di quanto detto finora, a proposito delle espressioni che costituiscono il valore dei predicati di attivazione delle regole XSLT, siamo in grado di estrarre porzioni di cv che soddisfino requisiti ben precisi.

In particolare, come esempio guida, svilupperemo nel dettaglio un algoritmo in grado di riprodurre in "bella copia", porzioni di cv di persone con almeno 41 anni, automunite.

La soluzione sarà generale, permettendo di risolvere molti problemi analoghi.

10.4.1 Ipotesi di partenza

Supponiamo che, dopo una attenta valutazione di quali dati sia utile disporre, all'interno di un cv, `Trova l'impiegato` abbia definito un documento `cv.dtd` che formalizzi la seguente semantica per i cv che essa tratterà automaticamente:

```
<!DOCTYPE curriculae [
 <!ELEMENT curriculae (cv)*>
 <!ELEMENT cv (dati-anagrafici,esperienze-lavorative*,
               hobbies*,altro*)>
 <!ELEMENT dati-anagrafici (nome,cognome,nascita)>
 <!ELEMENT nome (#PCDATA)>
 <!ELEMENT cognome (#PCDATA)>
 <!ELEMENT nascita EMPTY>
  <!ATTLIST nascita anno CDATA #REQUIRED>
 <!ELEMENT esperienze-lavorative ANY>
 <!ELEMENT hobbies ANY>
 <!ELEMENT altro   EMPTY>
  <!ATTLIST altro valore (Automunito|Brevetto) #REQUIRED>
]>
```

Alcune osservazioni sono immediate:

- `Trova l'impiegato` mantiene un unico documento in cui compaiono, in sequenza, i cv di diverse persone, ciascuno individuato dall'elemento `cv`;
- la struttura di ciascun cv è estremamente stringata. In un caso reale, questo sarebbe inaccettabile, ma, senza perdere generalità e significatività dell'esempio, è più che sufficiente per i nostri scopi didattici.

Una istanza valida per `cv.dtd` è il seguente documento XML di riferimento, che chiameremo `cv.xml`:

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

<curriculae>
 <cv>
  <dati-anagrafici>
   <nome>Giulio</nome><cognome>Cesare</cognome>
   <nascita anno="1960"/>
  </dati-anagrafici>
  <esperienze-lavorative> Bla </esperienze-lavorative>
  <esperienze-lavorative> Bla, Bla </esperienze-lavorative>
  <altro valore="Automunito"/>
 </cv>
 <cv>
  <dati-anagrafici>
   <nome>Scipione</nome><cognome>L'Africano</cognome>
```

```
   <nascita anno="1970"/>
  </dati-anagrafici>
  <esperienze-lavorative> Bla </esperienze-lavorative>
 </cv>
 <cv>
  <dati-anagrafici>
   <nome>Alessandro</nome><cognome>Magno</cognome>
   <nascita anno="1960"/>
  </dati-anagrafici>
  <esperienze-lavorative> Bla </esperienze-lavorative>
  <esperienze-lavorative> Bla, Bla </esperienze-lavorative>
  <altro valore="Brevetto"/>
 </cv>
 <cv>
  <dati-anagrafici>
   <nome>Napoleone</nome><cognome>Bonaparte</cognome>
   <nascita anno="1965"/>
  </dati-anagrafici>
  <esperienze-lavorative> Bla </esperienze-lavorative>
  <altro valore="Automunito"/>
 </cv>
 <cv>
  <dati-anagrafici>
   <nome>Cesare</nome><cognome>Augusto</cognome>
   <nascita anno="1970"/>
  </dati-anagrafici>
  <esperienze-lavorative> Bla, Bla </esperienze-lavorative>
  <altro valore="Automunito"/>
 </cv>
</curriculae>
```

Si nota che, al momento della scrittura di queste righe, febbraio 2006, sia ha che:

- `Scipione L'Africano` ha meno di 41 anni e non ha specificato se sia automunito o meno;
- `Alessandro Magno` ha più di 41 anni, ma non è automunito;
- `Cesare Augusto` ha meno di 41 anni, pur essendo automunito;
- i rimanenti `Giulio Cesare` e `Napoleone Bonaparte` hanno più di 41 anni e sono automuniti.

Il problema

Dopo le osservazioni precedenti sull'età e sulla disponibilità di un'automobile, da parte delle persone cui i cv si riferiscono, supponiamo di porci il seguente problema:

> Scrivere un algoritmo, ovvero un documento XSLT, che, preso un elenco di *curriculae*, validi per la semantica data, restituisca le parti salienti dei soli *curriculae* di persone con almeno 41 anni, automunite.

Una volta scritto l'algoritmo, applicandolo all'esempio di documento XML, dato, otterremmo i cv di `Giulio Cesare` e `Napoleone Bonaparte`.

In più se dovessimo aggiungere *curriculae* di persone che soddisfino i requisiti, l'algoritmo li aggiungerebbe automaticamente nell'elenco estratto.

10.4.2 L'algoritmo

Un documento XSLT, che chiameremo `cv.xslt`, che descrive l'algoritmo cercato per risolvere il problema enunciato è il seguente:

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

  <xsl:template match="/"> <!-- Regola R1 -->
      <html>
        <head>
          <title>CV di persone automunite,
         con almeno 41 anni</title>
        </head>
        <body>
          <xsl:apply-templates/>
        </body>
      </html>
  </xsl:template>

  <xsl:template match="curriculae"> <!-- Regola R2 -->
      <h1>CV di persone automunite, con almeno 41 anni</h1>
      <xsl:apply-templates/>
  </xsl:template>

 <!-- Regola R3:
      * l'espressione valore del predicato di attivazione
        della regola impone un doppio vincolo:
        a) cv deve avere un elemento nascita, figlio di
        dati-anagrafici che contenga un attributo anno
        con valore inferiore a 1965, e
        b) cv deve avere un figlio altro il cui attributo
        abbia valore 'Automunito'
      * se il nodo cv corrente soddisfa il vincolo, tutti
        i suoi componenti, tranne hobbies, sono inclusi
        nel risultato
```

```
 -->
  <xsl:template match="cv[1965>=dati-anagrafici/nascita/@anno
                       and 'Automunito'=altro/@valore]">

    <hr/>
    <xsl:apply-templates select="dati-anagrafici"/>
    <h3>Esperienze lavorative:</h3>
     <ul>
      <xsl:apply-templates select="esperienze-lavorative"/>
     </ul>
    <h3>Altro:</h3>
     <ul>
      <xsl:apply-templates select="altro"/>
     </ul>
 </xsl:template>

 <!-- Regola R4: se il vincolo di applicazione della regola
                 precedente non e' soddisfatto, per il nodo cv
                 corrente e' prevista l'applicazione della regola
                 seguente, che fornisce un risultato vuoto
 -->
 <xsl:template match="cv"/>

<!-- Regole per una gradevole pubblicazione dei curriculae -->

 <xsl:template match="dati-anagrafici"> <!-- Regola R5 -->
  <h2>Cv di <xsl:value-of select="nome"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="cognome"/></h2>
  <bf>Anno di nascita:</bf> <xsl:value-of select="nascita/@anno"/>
 </xsl:template>

 <xsl:template match="esperienze-lavorative"> <!-- Regola R6 -->
  <li><xsl:value-of select="."/></li>
 </xsl:template>

 <xsl:template match="altro"> <!-- Regola R7 -->
  <li><xsl:value-of select="@valore"/></li>
 </xsl:template>

</xsl:stylesheet>
```

`cv.xslt` specifica un algoritmo di visita di un documento valido per `cv.dtd`, quale è, ad esempio, `cv.xml`. In particolare:

- La regola `R1` si attiva sulla radice assoluta del documento `cv.xml` e costruisce l'impalcatura di un corretto documento XHTML che, come vedremo,

conterrà una lista di paragrafi, ciascuno corrispondente ad uno dei cv che soddisfano i criteri del problema dato;

- La regola `R2` si attiva sull'elemento radice della DTD, rispetto a cui `cv.xml` è valido, ed introduce l'intestazione dell'intero documento XHTML da produrre;

- La regola `R3` è il cuore di tutto l'algoritmo. Essa produce i paragrafi, corrispondenti ai cv che soddisfano i criteri di selezione.

  Il suo predicato di attivazione "dice" che la regola va applicata quando l'elemento attivo `cv` è tale che le due seguenti condizioni sono soddisfatte:

  1. esiste un elemento `nascita`, figlio dell'elemento `dati-anagrafici` il cui attributo `anno` ha valore minore o uguale a 1965. L'espressione che esprime questo vincolo è la prima, contenuta tra le due parentesi quadre che seguono l'elemento `cv`:

     ```
     1965>=dati-anagrafici/nascita/@anno
     ```

     La sua sintassi descrive le dipendenze tra i vari elementi ed attributi nell'albero sintattico di `cv.xml`:

     - `nascita/@anno` individua l'attributo `anno` come figlio dell'elemento `nascita`,
     - `dati-anagrafici/nascita` individua l'elemento `nascita` come figlio dell'elemento `dati-anagrafici`;

  2. esiste un elemento `altro` il cui attributo `valore` ha valore pari a `Automunito`. L'espressione che esprime questo vincolo è la seconda, contenuta tra le due parentesi quadre che seguono l'elemento `cv`:

     ```
     'Automunito'=altro/@valore
     ```

     La sua sintassi descrive le dipendenze tra i vari elementi ed attributi nell'albero sintattico di `cv.xml`:

     - `altro/@valore` individua l'attributo `valore` come figlio dell'elemento `altro`;

  3. entrambi gli elementi `dati-anagrafici` e `altro` devono essere figli dell'elemento attivo `cv`. Questo è indicato dal fatto che, nella espressione del predicato di attivazione, l'elemento `cv` è seguito da due parentesi quadre che contengono le espressioni con cui si esprimono i vincoli sui valori degli attributi `anno` e `valore`:

     ```
     cv[espressione-con-vincolo-su-nome and
       espressione-con-vincolo-su-valore]
     ```

     In particolare, `espressione-con-vincolo-su-nome` è l'espressione:

     ```
     1965>=dati-anagrafici/nascita/@anno
     ```

     mentre `espressione-con-vincolo-su-valore` è l'espressione:

```
'Automunito'=altro/@valore
```

Una volta attivata, R3 richiama la ricorsione in modo da visitare gli elementi figli del nodo cv attivo ed introdurre nel documento XHTML oggetto le informazioni relative, per mezzo delle regole R5, R6 ed R7.

- La regola R4 si applica a qualsiasi elemento cv che non soddisfi il predicato di attivazione della regola R3, ovvero ai cv di persone che o non hanno almeno 41 anni, o che non sono automunite. Siccome R4 non produce alcunché, in corrispondenza di tali cv, le loro informazioni sono completamente omesse dal documento XHTML finale;

- Nella regola R5 vale la pena notare l'elemento:

  ```
  <xsl:text> </xsl:text>
  ```

  che contiene uno "spazio". Esso è necessario per separare opportunamente il nome dal cognome nel documento XHTML oggetto,c he si sta producendo.

Esercizio 95 (cv di 41-enni automuniti.) Verificare il risultato dell'interpretazione di cv.xml, pre mezzo di cv.xslt, usando Morphon XML Editor o XRay 2.0.

Modificare il predicato di attivazione di R3, affinché, ad esempio, l'elenco dei cv stampati contenga solo quelli delle persone con una, o entrambe, le seguenti caratteristiche:

- possedere un brevetto;

- essere nate nel 1970.

Appendice A

Soluzioni agli esercizi

Soluzione all'Esercizio 35, pagina 61.

Una possibile descrizione delle proprietà invarianti:

- Un'espressione può essere un numero, oppure,
- un'espressione può essere la somma o la sottrazione tra due espressioni.
- Un numero è una sequenza di cifre.
- Una cifra è uno dei simboli seguenti $\{0, 1\}$.

Soluzione all'Esercizio 36, pagina 62.

1. Un frase è inerentemente ambigua se il suo significato è chiaro solo una volta fissato il contesto in cui essa si colloca.
2. Una banale modifica dell'esempio nel testo: "Il bambino gusta il brodo con il pane". Solo vedendo il bambino, capiamo se nel brodo c'è il pane inzuppato, o se il bambino intinge il pane di volta in volta.
3. Il criterio è, in realtà, sofisticato, nel seguente senso: è possibile fornire una definizione formale di *ambiguità intrinseca*, basato sulla struttura sintattica di una frase. La definizione sarà evidente quando si osserverà come la sintassi XML possa definire alberi sintattici.

Soluzione all'Esercizio 37, pagina 71.

- Sfruttando una possibile etichettatura di *numero* si ha:

```
<numero_telefonico>
  <prefisso_internazionale>
    <numero>
      <cifra>2</cifra>
      <cifra>3</cifra>
    </numero>
  </prefisso_internazionale>
  <prefisso_nazionale>
    <numero>
      <cifra>0</cifra>
      <cifra>1</cifra>
      <cifra>1</cifra>
    </numero>
  </prefisso_nazionale>
   <numero>
     <cifra>1</cifra>
     <cifra>2</cifra>
     <cifra>3</cifra>
   </numero>
</numero_telefonico>
```

- Sfruttando la struttura di numero, ma specializzandone gli usi:

```
<numero_telefonico>
  <prefisso_internazionale>
    <cifra>2</cifra>
    <cifra>3</cifra>
  </prefisso_internazionale>
  <prefisso_nazionale>
    <cifra>0</cifra>
    <cifra>1</cifra>
    <cifra>1</cifra>
  </prefisso_nazionale>
    <numero>
     <cifra>1</cifra>
     <cifra>2</cifra>
     <cifra>3</cifra>
   </numero>
</numero_telefonico>
```

Soluzione all'Esercizio 38, pagina 71.

```
<tonda>
```

```
  <quadra>
  </quadra>
  <quadra>
    <graffa>
    </graffa>
  </quadra>
</tonda>
```

Il vantaggio potenziale nell'usare etichette sta nel poter inventare nomi arbitrari per parentesi, oltre a quelli usuali e naturali usati nella soluzione.

Soluzione all'Esercizio 39, pagina 74.

1. Segue una soluzione ragionevolmente coerente con la metodologia essenziale di etichettatura (Sezione 4.1.3):

   ```
   <nutrizione>
        <quantita    valore="1" unita-di-misura="porzione"/>
        <calorie     kjoul="1167"/>
        <grassi      grammi="23"/>
        <carboidrati grammi="45"/>
        <proteine    grammi="32"/>
   </nutrizione>
   ```

   L'etichettatura scelta descrive `nutrizione` come un concetto complesso, costituito da concetti più primitivi. I concetti componenti sono, però, così primitivi che non è il caso di scomporli ulteriormente. La loro descrizione si riduce a grandezze numeriche, per la cui descrizione sono adatti gli attributi. Notiamo che, rispetto alla formalizzazione originale di `nutrizione`, abbiamo aggiunto l'unità di misura.

2. Si consideri la porzione di testo "Beef Parmesan with Garlic Angel Hair Pasta":

   ```
   Quick fry (brown quickly on both sides) meat. Place meat
   in a casserole baking dish, slightly overlapping edges.
   Place onion rings and peppers on top of meat, and pour
   marinara sauce over all.
   ```

   che costituisce un (macro)`passo`.

   Una proposta possibile di ristrutturazione potrebbe mirare a rendere più granulare la descrizione delle operazioni, magari numerandole, in modo da poter aggiungere passi ulteriori di spiegazione, se il livello di dettaglio scelto fosse insufficiente.

   L'etichettatura potrebbe essere la seguente:

```
  <passo ordine="1">quick fry on both sides</passo>
  <passo ordine="2">
    Place meat in a casserole baking dish,
    slightly overlapping edges
  </passo>
  <passo ordine="3">
    Place onion rings and peppers on top of meat,
    and pour marinara sauce over all.
</passo>
```

3. Si consideri ancora la medesima porzione di testo "Beef Parmesan with Garlic Angel Hair Pasta":

   ```
   Quick fry (brown quickly on both sides) meat. Place meat
   in a casserole baking dish, slightly overlapping edges.
   Place onion rings and peppers on top of meat, and pour
   marinara sauce over all.
   ```

   Segue una proposta ad altissima "granularità" di dettaglio:

   ```
   <passo tipo="fry"
          tempo="20 secs"></passo>
   <passo tipo="flip"></passo>
   <passo tipo="fry"
          tempo="20 secs"></passo>
   <passo tipo="place"
          posto="casserole baking dish"></passo>
   <!-- cosi' "via" -->
   ```

   Commento 28
   Il testo è praticamente illeggibile ad un cuoco, ma è molto più vicino al modo di vedere di un puro esecutore, quale potrebbe essere un calcolatore, magari dotato di utensili adatti a cucinare, o che simula una seduta virtuale di cucina.

   Un tale livello di dettaglio richiede almeno una classificazione di tutti i tipi di passo eseguibili. Quelli evidenziati sono: `fry`, `flip`, `place`.

   Occorre inoltre prevedere la necessità di individuare senza ambiguità qual è il soggetto delle azioni.

   Infatti, i passi etichettati agiscono implicitamente su pezzi di carne.

   Quindi, per usare l'aglio (*garlic*) occorrerà prevedere un'operazione esplicita di *cambio soggetto*, usata al momento opportuno:

   ```
   <passo tipo="change subject">garlic</passo>
   ```

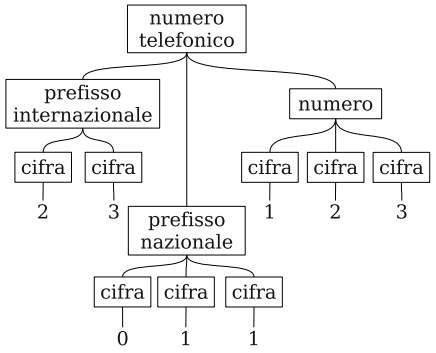
   L'osservazione conclusiva è che la difficoltà nello specificare il dettaglio delle operazioni è sintomatica della complessità delle informazioni contenute nei passi di una ricetta, scritta in linguaggio naturale per una persona.

Soluzione all'Esercizio 40, pagina 76.

L'etichettatura di numero telefonico scelto è:

```
<numero_telefonico>
  <prefisso_internazionale>
    <cifra>2</cifra>
    <cifra>3</cifra>
  </prefisso_internazionale>
  <prefisso_nazionale>
    <cifra>0</cifra>
    <cifra>1</cifra>
    <cifra>1</cifra>
  </prefisso_nazionale>
   <numero>
     <cifra>1</cifra>
     <cifra>2</cifra>
     <cifra>3</cifra>
   </numero>
</numero_telefonico>
```

il cui albero risulta essere:



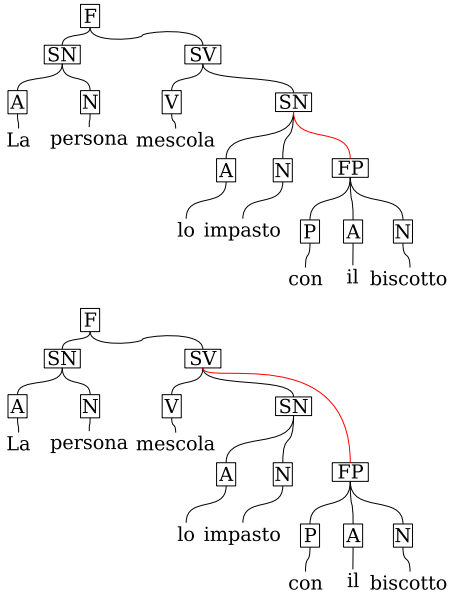
Soluzione all'Esercizio 41, pagina 76.

Questa soluzione evidenzia come l'ambiguità di una frase, normalmente spiegata in termini semantici, dicendo che la frase si capisce solo contestualizzandola, si riduca ad un esercizio sintattico, risultato di un'oculata analisi della struttura sintattica della frase.

La seguente legenda fissa una strutturazione possibile di semplici frasi in italiano, un cui prototipo è: "La persona mescola lo impasto con il biscotto":

Simbolo	Significato
F	Frase
SN	Sintagma nominale
N	Nome
SV	Sintagma verbale
V	Verbo
FP	Frase Preposizionale
A	Articolo
P	Preposizione

La strutturazione della frase prototipo, in accordo con la legenda origina due possibili alberi sintattici non sovrapponibili, ciascuno dei quali descrive un possibile significato dell'unica frase prototipo:



Secondo il primo albero il biscotto è parte dell'impasto; nel secondo il biscotto è lo strumento con cui si mescola.

L'esistenza di due alberi sintattici differenti per la stessa frase è un esempio di criterio formale per stabilire quando una frase sia ambigua.

Soluzione all'Esercizio 45

1. Un singolo nodo, senza discendenti e ascendenti, etichettato con un nome abitrario, non ha cammini.

2. L'albero ha un nodo $n_{0,0}$ a livello 0, la radice, un nodo $n_{1,1}$ a livello 1, e cinque nodi $n_{2,1}, \ldots, n_{2,5}$ a livello 2.

   L'unico discendente diretto di $n_{0,0}$ è $n_{1,1}$, mentre $n_{1,1}$ ha come diescendenti diretti tutti e soli i nodi $n_{2,1}, \ldots, n_{2,5}$.

3. L'albero ha un singolo nodo radice, un numero non prefissato di nodi al livello 1 ed esattamente cinque nodi $n_{2,1}, \ldots, n_{2,5}$ a livello 2.

   Ciascun nodo $n_{2,i}$, con $i \in \{1, \ldots, 5\}$, può essere discendente diretto di un qualsiasi nodo a livello 1, non necessariamente lo stesso di cui $n_{2,j}$, con $i \neq j$, è discendente diretto.

Soluzione all'Esercizio 46, pagina 93.

Siccome non è necessario specificare quali siano le cifre componenti `numero`, `cifra` può essere vuota:

```
<!ELEMENT cifra EMPTY>
<!ELEMENT numero (cifra, cifra, cifra)>
```

Soluzione all'Esercizio 47, pagina 93.

Siccome non è necessario specificare quali siano le cifre componenti `numero`, `cifra` può essere vuota:

```
<!ELEMENT cifra EMPTY>
<!ELEMENT numero (cifra*)>
```

Soluzione all'Esercizio 48, pagina 94.

Siccome non è necessario specificare né quali siano le cifre componenti `numero`, né quale sia il segno, `cifra` e `segno` possono essere vuoti:

```
<!ELEMENT segno EMPTY>
<!ELEMENT cifra EMPTY>
<!ELEMENT numero (segno?, cifra*)>
```

Soluzione all'Esercizio 49, pagina 94.

Una possibile soluzione:

```
<!ELEMENT segno EMPTY>
<!ELEMENT cifra EMPTY>
<!ELEMENT numero (segno?, cifra+)>
```

Soluzione all'Esercizio 50, pagina 95.

Una possibile soluzione che prevede l'utilizzo di numeri con sole tre cifre diverse è:

```
<!ELEMENT piu   EMPTY>
<!ELEMENT meno  EMPTY>
<!ELEMENT segno (piu | meno)>
<!ELEMENT cifra1 EMPTY>
<!ELEMENT cifra2 EMPTY>
<!ELEMENT cifra3 EMPTY>
<!ELEMENT cifra (cifra1|cifra2|cifra3)>
<!ELEMENT numero (segno?, cifra+)>
```

Soluzione all'Esercizio 51, pagina 97.

Le sequenze di generazione seguenti non sono le uniche possibili:

1. `E --> (E,E,E) --> (E,(E|E),E) --> (E,(E|E),(E|E))`
   `--> (E,(E|E+),(E|E)) --> (b,(E|E+),(E|E)) --> ...`
   `--> (b,(c|d+),(e|f))`

2. `E --> (E,E,E) --> (E,E,(E|E)) --> (E,E,(E+|E))`
   `--> (E,E,((E|E)+|E)) --> (E?,E,((E|E)+|E))`
   `--> (E?,E*,((E|E)+|E)) --> ... --> (b?,c*,((d|e)+|f))`

3. `E --> (E,E,E,E,E) --> (E+,E,E,E,E) --> (E+,E,E*,E,E)`
   `--> (E+,E,E*,E?,E) --> (E+,(E|E),E*,E?,E)`
   `--> (E+,(E|E),E*,(E|E)?,E)`
   `--> (E+,(E|E),E*,(E|E)?,(E|E)) --> ...`
   `-->  (E+,(E*|E?),E*,(E|E)?,(E+|E*)) --> ...`
   `--> (b+,(c*|d?),e*,(f|g)?,(h+|i*))`

Soluzione all'Esercizio 52, pagina 97.

1. Possibili soluzioni sono `(a*,b*,c)+` e `(a*,b*,c+)+`. Perché?

2. `((a,b)*|(c,d)+)`

Soluzione all'Esercizio 53, pagina 98.

La formalizzazione può essere:

```
<!ELEMENT contenuto-misto (#PCDATA | el1 | el2 )*>
<!ELEMENT el1 EMPTY>
<!ELEMENT el2 EMPTY>
```

Soluzione all'Esercizio 54, pagina 98.

```
<!ELEMENT piu    EMPTY>
<!ELEMENT meno   EMPTY>
<!ELEMENT cifra0 EMPTY>
<!ELEMENT cifra2 EMPTY>
<!ELEMENT numero-intero ((piu | meno)?, (cifra0 | cifra1)+)>
```

Soluzione all'Esercizio 55, pagina 98.

1. Esempio:

   `<a> <b>b</b> <d>d0</d> <d>d1</d> <f>f</f> </a> .`

   Controesempio: `<a> <b>b</b> <e>e</e> </a>`.

2. Esempio: `<a> <f>f</f> </a>`.

   Il seguente controesempio contiene due errori:

```
<a><b>b0</b><b>b1</b><e>e</e><f>f</f></a>
```

3. Esempi:

```
<a><b></b><d></d><f></f><h></h></a>
<!-- oppure -->
<a><b>b</b><d>d</d><f>f</f></a>
```

   Il seguente controesempio contiene tre errori:

```
<a>
  <b>b</b>
  <d>d0</d>
  <d>d1</d>
  <f>f1</f>
  <g>g1</g>
  <g>g2</g>
</a>
```

Soluzione all'Esercizio 56, pagina 99.

Una possibile soluzione è la DTD:

```
<!ELEMENT nomi-documenti
(data,separatore,esercizio,separatore,studente,suffisso)>
<!ELEMENT data (aa,separatore-data,mm,separatore-data,gg)>
<!ELEMENT separatore EMPTY>
<!ELEMENT esercizio EMPTY>
<!ELEMENT studente (#PCDATA)>
<!ELEMENT suffisso EMPTY>
<!ELEMENT aa EMPTY>
<!ELEMENT separatore-data EMPTY>
<!ELEMENT mm EMPTY>
<!ELEMENT gg EMPTY>
```

che valida una struttura come:

```
<?xml version="1.0" encoding="UTF-8"?>
<nomi-documenti>
  <data>
    <aa/>
    <separatore-data/>
    <mm/>
    <separatore-data/>
```

```
    <gg/>
  </data>
  <separatore/>
  <esercizio/>
  <separatore/>
  <studente>NomeCognome</studente>
  <suffisso/>
</nomi-documenti>
```

Soluzione all'Esercizio 57, pagina 99.

Segue una possibile semantica con cui validare una generalizzazione ragionevole dell'esempio di testo XML, inserito, come esempio, nell'esercizio in oggetto.

```
<!ELEMENT tag-foglia (minore,contenuto,chiusura)>
<!ELEMENT minore EMPTY>
<!ELEMENT slash EMPTY>
<!ELEMENT maggiore EMPTY>
<!ELEMENT chiusura (slash,maggiore)>
<!ELEMENT contenuto (nome,attributo*)>
<!ELEMENT attributo (spazio,nome,uguale,apice,valore,apice)>
<!ELEMENT nome EMPTY>
<!ELEMENT spazio EMPTY>
<!ELEMENT apici EMPTY>
<!ELEMENT valore EMPTY>
<!ELEMENT uguale EMPTY>
```

Soluzione all'Esercizio 58, pagina 99.

Segue una possibile semantica in cui sia le sequenze di ERD di lunghezza prefissata, sia le alternative tra un numero finito di ERD, sono "simulate" con sequenze ed alternative tra due elementi:

```
<!ELEMENT foglia (#PCDATA)>
<!-- sequenze limitate di lunghezza n sono
     "simulate" con sequenze binarie, annidate
     per n-1 volte -->
<!ELEMENT sequenza-limitata (erd,erd)>
<!ELEMENT sequenza-arbitraria (erd*)>
<!ELEMENT sequenza-non-vuota (erd+)>
<!ELEMENT sequenza-singolo-elemento (erd?)>
<!-- alternative tra n elementi sono
     "simulate" con alternative binarie,
     annidate per n-1 volte -->
<!ELEMENT alternativa (erd | erd)>
<!ELEMENT erd ( foglia
              | sequenza-limitata
```

```
| sequenza-arbitraria
| sequenza-non-vuota
| sequenza-singolo-elemento
| alternativa  )>
```

Soluzione all'Esercizio 59, pagina 103.

- Esempi: `<r a="v1"/>` e `<r a="v2"/>`.
- Controesempi: `<r/>` e `<r b="valore a piacere"/>`.

Soluzione all'Esercizio 60, pagina 104.

- Esempi: `<r/>`, `<r a="v1"/>` e `<r a="v2"/>`.
- Controesempi: `<r b="valore a piacere"/>`.

Soluzione all'Esercizio 61, pagina 104.

- Esempi: `<r/>` `<r a="v1"/>`.
- Controesempi: `<r a="v2"/>` e `<r b="valore a piacere"/>`.

Soluzione all'Esercizio 62, pagina 105.

- Esempi:

```
<r/>
<!-- oppure -->
<r><e a=":id1"/></r>
<!-- oppure -->
<r><e a=":id1"/><e a=":id2"/></r>
```

- Controesempi:

```
<r><e a=""/></r>
<!-- oppure -->
<r><e/></r>
<!-- oppure -->
<r><e a=":id1"/><e a=":id1"/></r>
```

Soluzione all'Esercizio 63, pagina 106.

- Eesempi:

```
<r>
  <e a="u"/> <e a="d"/> <e a="t"/>
  <f l="u"/> <f l="d"/> <f l="u"/>
</r>
```

- Controesempi:

```
<r>
  <e a="u"/> <e a="d"/> <e a="t"/>
  <f l="u"/> <f l="u"/> <f l="q"/>
</r>
<!-- oppure -->
<r>
  <e a="u"/> <e a="u"/> <e a="t"/>
  <f l="u"/> <f l="u"/> <f l="q"/>
</r>
<!-- oppure -->
<r>
  <e a="u"/> <e a="u"/> <e a="d"/>
  <f l="u"/> <f l="d"/> <f/>
</r>
```

Soluzione all'Esercizio 64, pagina 106.

- Eesempi:

```
 <r>
   <e/>       <e a="d"/> <e a="t"/>
   <f l="d"/> <f/>       <f/>
</r>
```

- Controesempi:

```
<r>
  <e/> <e a="d"/> <e a="t"/>
  <f l="u"/> <f/> <f/>
</r>
```

Soluzione all'Esercizio 65, pagina 106.

- Esempi:

```
<r>
  <e/> <e a="d"/> <e a="t"/>
  <f l="d"/> <f l="d"/>
</r>
```

- Controesempi:

```
<r>
  <e/> <e a="d"/> <e a="t"/>
  <f l="d"/> <f l="u"/> <f l="t"/>
</r>
```

Soluzione all'Esercizio 66, pagina 106.

- Esempi:

```
<r>
  <e/> <e a="d"/> <e a="t"/>
  <f l="d"/> <f/>
</r>
```

- Controesempi:

```
<r>
  <e/> <e a="d"/> <e a="t"/>
  <f l="d"/> <f l="u"/> <f l="t"/>
</r>
```

Soluzione all'Esercizio 67, pagina 107.

- Uno dei due valori `s1` deve essere modificato in uno distinto da `s1` e da `s2`. Oppure, occorre trasformare il tipo di `i` in qualcosa di diverso da `ID`.
- Il valore `s3` deve essere modificato in uno a scelta tra `s1` o `s2`. Oppure, occorre trasformare il tipo dell'attrobuto `i` dell'elemento `b` in qualcosa di diverso da `IDREF`.
- L'intero elemento `<b i="s2"/>` deve essere eliminato. Oppure, occorre trasformare la definizione di `es-id` da `(a*|b)` in, ad esempio, `(a*|b)*`.

Soluzione all'Esercizio 68, pagina 107.

Una prima DTD, eventualmente memorizzata in un file `semantica.dtd`, che soddisfi i criteri stabiliti è la seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
 <!-- cifra -->
  <!ELEMENT c EMPTY>
   <!ATTLIST c v (0|1|2|3|4|5|6|7|8|9) #REQUIRED>
 <!-- "coda" del prefisso internazionale -->
  <!ELEMENT cpi (c+)>
 <!-- "testa" del prefisso internazionale -->
  <!ELEMENT tpi EMPTY>
   <!ATTLIST tpi v (00 | piu) #REQUIRED> <!-- "piu" necessario -->
 <!-- prefisso internazionale -->
  <!ELEMENT pi (tpi,cpi)>
 <!-- prefisso nazionale -->
  <!ELEMENT pn (c,c+)>
 <!-- utenza -->
  <!ELEMENT u (c+)>
```

```
 <!-- numero telefonico -->
  <!ELEMENT nt (pi,pn,u)>
```

Essa contiene numeri telefonici del tipo:

```
<?xml version="1.0"?>
<!DOCTYPE nt
  SYSTEM "semantica.dtd">
<nt>
  <pi>
    <tpi v="00"/>
    <cpi>
        <c v="1"/>
    </cpi>
  </pi>
  <pn>
    <c v="0"/>
    <c v="1"/>
  </pn>
  <u>
    <c v="1"/>
    <c v="2"/>
    <c v="3"/>
  </u>
</nt>
```

oppure

```
<?xml version="1.0"?>
<!DOCTYPE nt
  SYSTEM "semantica.dtd">
<nt>
    <pi>
        <tpi v="piu"/>
        <cpi>
            <c v="1"/>
        </cpi>
    </pi>
    <pn>
        <c v="0"/>
        <c v="1"/>
        <c v="1"/>
    </pn>
    <u>
        <c v="1"/>
        <c v="2"/>
        <c v="3"/>
```

```
    </u>
</nt>
```

Al contrario, un controesempio è:

```
<?xml version="1.0"?>
<!DOCTYPE nt
  SYSTEM "semantica.dtd">
<nt>
    <pi>
        <tpi v="0"/>
        <cpi>
            <c v="1"/>
        </cpi>
    </pi>
    <pn>
        <c v="0"/>
        <c v="1"/>
        <c v="1"/>
    </pn>
    <u>
        <c v="1"/>
        <c v="2"/>
        <c v="33"/>
    </u>
</nt>
```

Segue una seconda DTD, con relativo esempio validato:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE nt  [
<!ELEMENT nt (pi,pn,u)>
<!ELEMENT pi ((piu|(cz,cz)),(cz|c)+)>
<!ELEMENT piu EMPTY>
<!ELEMENT cz EMPTY>
  <!ATTLIST  cz v CDATA #FIXED "0">
<!ELEMENT c EMPTY>
  <!ATTLIST  c   v (1|2|3|4|5|6|7|8|9) #REQUIRED>
<!ELEMENT pn ((cz|c),(cz|c)+)>
<!ELEMENT u (cz|c)+>
]>

<nt>
 <pi>
   <piu/><c v="3"/><c v="9"/>
 </pi>
```

```
 <pn>
  <cz/><c v="1"/>
 </pn>
 <u>
   <c v="1"/><c v="2"/><c v="3"/>
 </u>
</nt>
```

Questa DTD è più raffinata della precedente. Essa permette di imporre in maniera naturale vincoli che esistono nella definizione di numeri telefonici italiani, ma che non sono espressi dal testo dell'esercizio.

Soluzione all'Esercizio 69, pagina 108.

Una semantica DTD eventualmente memorizzata in un file `semantica.dtd`, che potrebbe soddisfare i criteri stabiliti è la seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
 <!-- Utensile -->
<!ELEMENT u EMPTY>
  <!ATTLIST u pr IDREF #REQUIRED
              n   (casseruola
                  |cucchiaio-legno
                  |setaccio
                  |etc             ) #REQUIRED>

<!-- Ingrediente -->
<!ELEMENT i (#PCDATA)>
  <!ATTLIST i pr IDREF #REQUIRED>

<!-- Passo ricetta -->
<!ELEMENT pr (#PCDATA)>        <!-- Il passo contiene una
                                    descrizione-->
  <!ATTLIST pr o ID #REQUIRED> <!-- Ordine del passo
                                    nella sequenza -->

<!-- Ricetta -->
<!ELEMENT ricetta (pr|i|u)*>
```

Essa permette di descrivere una ricetta che evolve, senza troppo riguardo nei confronti di un ordine esplicito. Al contrario l'ordine è implicito e la coerenza tra passi, ingredienti ed utensili si avvantaggia dell'uso di identificatori:

```
<?xml version="1.0"?>
<!DOCTYPE ricetta SYSTEM "semantica.dtd">

<ricetta>
    <pr o="p1">
```

```
        <!-- Descrizione -->
    </pr>
    <pr o="p2">
        <!-- Descrizione -->
    </pr>
    <i pr="p2">
        <!-- Descrizione ingrediente -->
    </i>
    <i pr="p2">
        <!-- Descrizione ingrediente -->
    </i>
    <u n="casseruola" pr="p1"/>
    <u n="etc" pr="p2"/>
    <pr o="p3">
        <!-- Descrizione -->
    </pr>
</ricetta>
```

Una soluzione alternativa potrebbe essere:

```
<?xml version="1.0"?>
<!DOCTYPE ricetta [
<!ELEMENT ricetta (titolo,passi,ingredienti,utensili)>
<!ELEMENT titolo (#PCDATA)>
<!ELEMENT passi (passo+)>
<!ELEMENT passo ANY>
 <!ATTLIST passo ordine ID #REQUIRED>
<!ELEMENT ingredienti (ingrediente*)>
<!ELEMENT ingrediente EMPTY>
 <!ATTLIST ingrediente nome ID #REQUIRED>
 <!ATTLIST ingrediente quantita ID #REQUIRED>
 <!ATTLIST ingrediente ordine IDREF #REQUIRED>
<!ELEMENT utensili (utensile*)>
<!ELEMENT utensile EMPTY>
 <!ATTLIST utensile nome ID #REQUIRED>
 <!ATTLIST utensile ordine IDREF #REQUIRED>
 ] >
```

Soluzione all'Esercizio 70, pagina 108.

Forniamo due abbozzi di soluzione. Entrambi dovrebbero assicurare di poter esprimere situazioni usuali come:

- L'esistenza di più discendenti da una stessa coppia di avi.
- L'esistenza di una terna di avi `A`, `B` e `C` tali che le coppie `A`,`B` e `A`,`C`, ad esempio, abbiano ciascuna un insieme distinto (ovviamente) di discendenti.

Segue una prima bozza di DTD che struttura le dipendenze discendente-avi sfruttando la naturale gerarchia degli alberi descritti da una semantica DTD.

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

 <!DOCTYPE AlberoGenealogico [
 <!ELEMENT AlberoGenealogico (Componente*)>
 <!ELEMENT Componente (Padre?, Madre?, ( Persona | Cane )) >
 <!ELEMENT Padre  (Componente)>
 <!ELEMENT Madre  (Componente)>

 <!ELEMENT Persona  ( Nome
                    , Cognome
                    , Carica*
                    , NascitaData?
                    , NascitaLuogo?
                    , MorteData?)>

 <!ELEMENT Cane      ( Razza
                     , Nome
                     , NascitaData
                     , NascitaLuogo
                     , ColoreMantello
                     , NumeroTatuaggio
                     , Proprietari+  )>

 <!ELEMENT NascitaData (Data)>
 <!ELEMENT MorteData   (Data)>
 <!ELEMENT Data        EMPTY >
 <!ATTLIST Data data #REQUIRED>

 <!ELEMENT NascitaLuogo EMPTY>
 <!ATTLIST NascitaLuogo luogo #REQUIRED>

 <--! Da completare -->
 ] >
```

Il difetto della prima bozza è che un avo che abbia più discendenti debba essere ripetuto più volte nel documento XML che raccoglie l'albero genealogico. È come dire che il documento di radice `AlberoGenealogico` è un "foresta": contiene tanti alberi distinti quanti sono i discendenti ultimi inseriti. Ovvero, supponiamo che `A` e `B` abbiano due figli `C` e `D`. Questa situazione verrebbe riassunta come segue:

```
<AlberoGenealogico>
        <!-- Albero relativo a C -->
        <Componente>
```

```
        <Padre>
                <Componente>
                        <Persona>
                                <Nome n="A"/>
                                <!-- Completare -->
                        </Persona>
                </Componente>
        </Padre>
        <Madre>
                <Componente>
                        <Persona>
                                <Nome n="B"/>
                                <!-- Completare -->
                        </Persona>
                </Componente>
        </Madre>
        <Persona>
                <Nome n="C"/>
                <!-- Completare -->
        </Persona>
        </Componente>

        <!-- Albero relativo a D -->
        <Componente>
        <Padre>
                <Componente>
                        <Persona>
                                <Nome n="A"/>
                                <!-- Completare -->
                        </Persona>
                </Componente>
        </Padre>
        <Madre>
                <Componente>
                        <Persona>
                                <Nome n="B"/>
                                <!-- Completare -->
                        </Persona>
                </Componente>
        </Madre>
        <Persona>
                <Nome n="C"/>
                <!-- Completare -->
        </Persona>
        </Componente>
</AlberoGenealogico>
```

Siccome la necessità di "duplicare un intero albero per inserire un novo discendente sembra sconveniente, segue una seconda bozza in cui suggeriamo di usare gli attributi di tipo `ID` e `IDREF` per descrivere le dipendenze discendente-avi nel modo seguente:

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

.<!DOCTYPE AlberoGenealogico [
 <!ELEMENT AlberoGenealogico (Componente*)>

 <!ELEMENT Componente ( Persona | Cane ) >

 <!ATTLIST Componente identita ID    #REQUIRED
                      padre    IDREF #IMPLIED
                      madre    IDREF #IMPLIED>

 <!ELEMENT Persona  ( Nome
                    , Cognome
                    , Carica*
                    , NascitaData?
                    , NascitaLuogo?
                    , MorteData?)>

 <!ELEMENT Cane      ( Padre
                     , Madre
                     , Razza
                     , Nome
                     , NascitaData
                     , NascitaLuogo
                     , ColoreMantello
                     , NumeroTatuaggio
                     , Proprietari+  )>

 <!ELEMENT NascitaData (Data)>
 <!ELEMENT MorteData   (Data)>
 <!ELEMENT Data        EMPTY >
 <!ATTLIST Data data #REQUIRED>

 <!ELEMENT NascitaLuogo EMPTY>
 <!ATTLIST NascitaLuogo luogo #REQUIRED>

 <--! Da completare -->
 ] >
```

Allora, la situazione precedente diventa la seguente:

```
<AlberoGenealogico>
```

```
      <!-- Componente A -->
        <Componente identita="1">
                <Persona>
                        <Nome n="A"/>
                        <!-- Completare -->
                </Persona>
        </Componente>

      <!-- Componente B -->
        <Componente identita="2">
                <Persona>
                        <Nome n="B"/>
                        <!-- Completare -->
                </Persona>
        </Componente>

      <!-- Componente C -->
        <Componente identita="3" padre="1" madre="2">
                <Persona>
                        <Nome n="C"/>
                        <!-- Completare -->
                </Persona>
        </Componente>

      <!-- Componente D -->
        <Componente identita="4" padre="1" madre="2">
                <Persona>
                        <Nome n="D"/>
                        <!-- Completare -->
                </Persona>
        </Componente>
</AlberoGenealogico>
```

Soluzione all'Esercizio 75, pagina 118.

1. `<s>testo1</s>` senza occorrenze di `a`, oppure

   ```
   <s>testo1 <a>testo2</a> testo3 </s>
   ```

   con occorrenze di `a`.

2. `<s>testo1</s>` senza occorrenze di `a`, oppure `<s>testo1 <a></a> </s>` con sole occorrenze vuote (una ...) di `a`.

3. Esempio: `<s>testo1 testo3 testo5 </s>`.

   Controesempio: `<s>testo1 testo3 testo5 <a>testo6</a></s>`.

4. Esempio: `<s>testo1 testo3 testo5 </s>`.

   Controesempio: `<s>testo1 testo3 testo5 <a/></s>`.

5. Esempio: `<a>testo1<b>testo2</b>testo3<b>testo4</b><b></b></a>`.

   Controesempio:

   ```
   <a>testo1<b>testo2</b>testo3<b>testo4</b><b></b><c/></a>
   ```

6. `<a>testo1<b>testo2</b>testo3<b>testo4</b><b></b><c/></a>`.

Soluzione all'Esercizio 76, pagina 119.

1. Le semantiche differiscono. Per esempio `<s>t1<a>t2</a><b>t3</b></s>` appartiene alla semantica di `DTD 1`, ma non a quella di `DTD 2`. Il motivo è che non definisce `b`.

2. Le semantiche definite coincidono. `DTD 1` definisce un insieme di alberi la cui radice è `s` con un contenuto qualsiasi, elemento `a` incluso, dato che le sue caratteristiche sono conosciute. Il contenuto arbitrario di `s`, quindi, può essere visto come un'alternanza arbitraria tra testo qualsiasi ed esempi dell'elemento `a`. Ma questa è la definizione dell'insieme caratterizzato da `DTD 2`.

3. Ad entrambe.

Soluzione all'Esercizio 78, pagina 120.

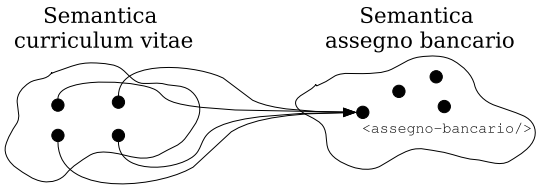
Un qualsiasi validatore, conforme all'interpretazione stabilita del linguaggio DTD, segnalerebbe che solo le regole d'uso `#IMPLIED` e `#REQUIRED` sono ammesse per `ID`.

Nulla cambierebbe, rimpiazzando `<!ELEMENT r (e*)>` con `<!ELEMENT r (e?)>`, per limitare strutturalmente l'uso di `e`.

Soluzione all'Esercizio 79

L'interpretazione ovvia, priva di senso dal punto di vista intuitivo, ma coerente con lo schema di interpretazione illustrato, trasformerà ogni esempio di `curriculum-vitae` sempre nello stesso esempio `<assegno-bancario/>` della semantica `assegno-bancario`.

Graficamente, la soluzione si configura come segue:



Soluzione all'Esercizio 80

1. Il testo XHTML cercato è:

```
<html>
   <header></header>
   <body>
      <h1><i>Proverbi</i></h1>
      <ul>
         <li>
            <b>Cina:</b>
               " Chi sente <i>dimentica</i>,
                 chi vede <i>ricorda</i>,
                 chi fa <i>impara</i>. "
         </li>
         <li>
            <b>Norvegia:</b>
            " Se ognuno pulisce davanti a casa,
              tutta la citt&agrave; &egrave; pulita. "
         </li>
      </ul>
   </body>
</html>
```

2. Il testo XHTML cercato è:

```
<html>
   <header></header>
   <body>
      <h1><i>Proverbi</i></h1>
      <table>
         <tr>
```

```
            <td><b>Origine</b></td>
            <td><b>Proverbio</b></td>
         </tr>
         <tr>
            <td><i>Cina</i></td>
            <td>
                 Chi sente <i>dimentica</i>,
                 chi vede <i>ricorda</i>,
                 chi fa <i>impara</i>.
            </td>
         </tr>
         <tr>
            <td><i>Norvegia</i></td>
            <td>
                 Se ognuno pulisce davanti a casa,
                 tutta la citt&agrave; &egrave; pulita.
            </td>
         </tr>
      </table>
   </body>
</html>
```

Soluzione all'Esercizio 82

Solo la radice "assoluta" del documento XML, ovvero:

`<?xml version=1.0 encoding=UTF-8?>`.

Soluzione all'Esercizio 83

i) Sì. ii) Sì.

Soluzione all'Esercizio 84

i) Sì. ii) No. iii) No in entrambi i casi. iv) Sì.

Soluzione all'Esercizio 85

i) Sì. ii) No. iii) No.

Soluzione all'Esercizio 86

La foglia "`Uguale per tutti!`" in ognuno dei casi prospettati.

Soluzione all'Esercizio 87

1. La sequenza di caratteri che descrive l'intero sotto albero la cui radice è il nodo attivo, ovvero:

```
<valore>A</valore>
<nodo-sx><foglia/></nodo-sx>
<nodo-dx><foglia/></nodo-dx>
```

2. La "sequenza" di caratteri `A`.

3. La seqenza di caratteri che descrive il sotto albero la cui radice è il nodo attivo, ovvero `<foglia/>`.

Soluzione all'Esercizio 88

Nessuna nota: se usati correttamente, ciascuno degli interpreti indicati produce il risultato previsto.

Soluzione all'Esercizio 89

1. Il risultato è:

```
<?xml version="1.0" encoding="utf-8"?>
c d b g h f l m i e a.
```

2. Il risultato è:

```
<?xml version="1.0" encoding="utf-8"?>
c b d a g f h e l i m+.
```

Soluzione all'Esercizio 90

1. Trasformare la regola `R2` in:

```
<xsl:template match="A"> <!-- R2 -->
  <xsl:apply-templates select="D"/> <!-- Ric2.D -->
  <xsl:apply-templates select="V"/> <!-- Ric2.V -->
</xsl:template>
```

   e la regola `R4` in:

```
<xsl:template match="V"> <!-- R4 -->
  <xsl:apply-templates/> <!-- Ric4 -->
</xsl:template>
```

2.
```
<?xml version="1.0"?>
<xsl:stylesheet
     version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/"> <!-- R1 -->
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="A"> <!-- R2 -->
  <xsl:apply-templates select="D"/> <!-- Ric2.D -->
  <xsl:apply-templates select="V"/> <!-- Ric2.U -->
</xsl:template>

<xsl:template match="D"> <!-- R3 -->
</xsl:template>

<xsl:template match="V"> <!-- R4 -->
  elementoV              <!-- Ric4 -->
</xsl:template>

</xsl:stylesheet>
```

3. Segue l'algoritmo che produce una rappresentazione di `alberi-arbitrari.xml` analoga a quella suggerita:

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

<xsl:template match="/"> <!-- R1 -->
```

```
  <html>
  <header/>
   <body>
    <xsl:apply-templates/>
   </body>
  </html>
</xsl:template>

<xsl:template match="A"> <!-- R2 -->
  <ul>
   <xsl:apply-templates select="D"/> <!-- Ric2.D -->
   <xsl:apply-templates select="U"/> <!-- Ric2.U -->
   <xsl:apply-templates select="V"/> <!-- Ric2.U -->
  </ul>
</xsl:template>

<xsl:template match="D">          <!-- R3 -->
  <li><xsl:apply-templates/></li> <!-- Ric3 -->
</xsl:template>

<xsl:template match="U"> <!-- R4' -->
  <xsl:apply-templates/> <!-- Ric4' -->
</xsl:template>

<xsl:template match="V"> <!-- R4'' -->
  <xsl:apply-templates/> <!-- Ric4'' -->
</xsl:template>

<xsl:template match="text()">  <!-- R5 -->
  <b><xsl:value-of select="."/></b>
</xsl:template>

</xsl:stylesheet>
```

Il risultato è

```
<html>
  <head>
  </head>
  <body>
    <b>a b c d e f g h i l m </b>
  </body>
</html>
```

Soluzione all'Esercizio 91

Segue la proposta:

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

  <xsl:template match="/"> <!-- R1 -->
   <html>
   <header/>
    <body>
     <xsl:apply-templates/>
    </body>
   </html>
  </xsl:template>

  <xsl:template match="A"> <!-- R2 -->
   <table>
    <tr>
     <xsl:apply-templates select="D"/> <!-- Ric2.D -->
     <xsl:apply-templates select="U"/> <!-- Ric2.U -->
     <xsl:apply-templates select="V"/> <!-- Ric2.V -->
    </tr>
   </table>
  </xsl:template>

  <xsl:template match="D"> <!-- R3 -->
   <td><xsl:apply-templates select="@w"/></td> <!-- Ric3 -->
  </xsl:template>

  <xsl:template match="@w"> <!-- R4 -->
   <xsl:value-of select="."/> <!-- Ric4 -->
  </xsl:template>

  <xsl:template match="U"> <!-- R5 -->
   <td><xsl:apply-templates/></td>  <!-- Ric5 -->
  </xsl:template>

  <xsl:template match="V"> <!-- R6 -->
   <td><xsl:apply-templates/></td>  <!-- Ric6 -->
  </xsl:template>

 </xsl:stylesheet>
```

Soluzione all'Esercizio 92

Una possibile soluzione è il documento XSLT seguente:

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

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

  <xsl:template match="/">
   <html>
    <xsl:apply-templates/>
   </html>
  </xsl:template>

  <xsl:template match="tavola-periodica">
   <body>
    <xsl:apply-templates/>
   </body>
  </xsl:template>

  <xsl:template match="atomo">
   <xsl:apply-templates select="nome"/>
  </xsl:template>

  <xsl:template match="nome">
   <xsl:value-of select="."/><br/>
  </xsl:template>
</xsl:stylesheet>
```

Applicandolo al sorgente `tavola-periodica.xml` il documento oggetto è:

```
<html><body>
  Idrogeno<br>
  Elio<br>
</body></html>
```

Soluzione all'Esercizio 93

Una possibile soluzione è il documento XSLT seguente:

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

 <xsl:template match="/">
  <html>
   <head/>
   <body>
    <xsl:apply-templates select="tavola-periodica"/>
   </body>
```

```
 </html>
</xsl:template>

<xsl:template match="tavola-periodica">
  <xsl:apply-templates select="comment()"/>
  <table>
   <tr>
     <td><b>Elemento</b></td>
     <td><b>Numero atomico</b></td>
     <td><b>Peso atomico</b></td>
   </tr>
    <xsl:apply-templates select="atomo"/>
  </table>
</xsl:template>

<xsl:template match="comment()">
 <h1><xsl:value-of select="."/></h1>
</xsl:template>

<xsl:template match="atomo">
 <tr>
  <td><xsl:apply-templates select="nome"/></td>
  <td><xsl:apply-templates select="numero-atomico"/></td>
  <td><xsl:apply-templates select="peso-atomico"/></td>
 </tr>
</xsl:template>

<xsl:template match="nome">
 <i><xsl:value-of select="."/></i>
</xsl:template>

<xsl:template match="numero-atomico">
 <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="peso-atomico">
 <xsl:value-of select="."/>
</xsl:template>

</xsl:stylesheet>
```

Soluzione all'Esercizio 94

Una possibile soluzione è il documento XSLT seguente:

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

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

 <xsl:template match="/">
  <html>
   <head/>
   <body>
    <xsl:apply-templates select="tavola-periodica"/>
   </body>
  </html>
 </xsl:template>

 <xsl:template match="tavola-periodica">
   <xsl:apply-templates select="comment()"/>
   <table>
    <tr>
      <td><b>Elemento</b></td>
      <td><b>Stato</b></td>
    </tr>
     <xsl:apply-templates select="atomo"/>
   </table>
 </xsl:template>

 <xsl:template match="comment()">
  <h1><xsl:value-of select="."/></h1>
 </xsl:template>

 <xsl:template match="atomo">
  <tr>
   <td><xsl:apply-templates select="nome"/></td>
   <td><xsl:apply-templates select="@stato"/></td>
  </tr>
 </xsl:template>

 <xsl:template match="nome">
  <i><xsl:value-of select="."/></i>
 </xsl:template>

 <xsl:template match="@stato">
  -<xsl:value-of select="."/>-
 </xsl:template>

</xsl:stylesheet>
```

Appendice B

Risorse

B.1 *Software*

XRay 2.0 e XMLCopy sono gli elaboratori XML, liberamente utilizzabili, che adotteremo per la parte sperimentale di questo testo.

B.1.1 Applicazioni "portabili"

portableapps.com fornisce programmi di larga utilità utilizzabili liberamente.

L'aspetto peculiare è che i programmi su portableapps.com sono utilizzabili senza procedura di installazione. Ricordiamo che l'instllazione non è sempre una procedura disponibile, soprattutto quando si lavora su calcolatori elettronici non nostri.

Tra quelli disponibili, Kompozer è un elaboratore di testi XHTML utile per la parte sperimentale del corso.

B.2 Siti

Segue un breve elenco di siti potenzialmente interessanti dai quali è possibile ricavare un'idea di progetti ed applicazioni degli strumenti XML, DTD e XSLT.

- Un manifesto sull'informatica come disciplina umanistica.
- Nuovo Codice dell'Amministrazione Digitale.
- Dati Aperti su Wikipedia ne introducintroduce il concetto.
- Formato Aperto su Wikipedia. ne introducintroduce il concetto
- europass.cedefop.europa.eu è il sito la raccolta e la diffusione di *cv* cui ci siamo riferiti come progetto di riferimento per fornire giustificazioni ai vari argomenti contenuti in questo testo.
- Unicode su www.unicode.org illustra le caratteristiche fondamentali dello standard `Unicode`. www.alanwood.net è un utile e curioso compendio al sito Unicode su www.unicode.org, in cui è possibile trovare molto materiale sulle capacità dei *browser* di rappresentare correttamente i caratteri alfabetici più disparati.
- xml.org raccoglie progetti per la strutturazione di informazioni relative ad ambiti rilevanti di utilizzo: *e-government*, *Human Resources–XML* (HR-XML), *etc.*.
- w3schools.com fornisce molte risorse per approfondire dal punto di vista più tecnico quel che questo testo sviluppa.

Appendice C

Testi sull'argomento

Questo non è un testo solo sulle tecnologie XML. Chi sia interessato ad approfondire tale argomento, non avrà che l'imbarazzo della scelta, potendo consultare sia testi *on-line*, liberamente accessibili, sia testi stampati. In generale, tuttavia, tali testi sono indirizzati a persone avvezze al lavoro da informatico.

Appendice D

Esempi di possibili testi d'Esame

NOTA. Ciascun testo d'esame presenterà esercizi ispirati, per argomento e difficoltà, a quelli illustrati qui di seguito.

D.1 Dati Aperti

- Definizione?
- Quali sono i diritti fondamentali associati al concetto?
- Qual è qualche esempio di dato aperto?

D.2 Formati Aperti

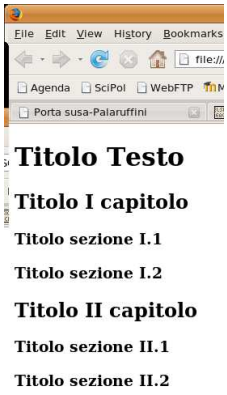
- Definizione?
- Qual è qualche esempio di formato aperto?
- Qual è qualche esempio di formato non aperto?
- Esistono esempi di formati aperti utilizzabili da programmi per i quali è necessario aver acquistato la licenza di utilizzo?

D.3 XHTML

1. Modificare il seguente sorgente XHTML rimpiazzando il testo "in corsivo" con testo in "grassetto".

```
<html>
<header/>
<body>
 testo <i>testo</i>
 <ul>
  <li>testo <i>testo</i></li>
  <li>testo <i>testo</i></li>
 </ul>
</body>
</html>
```

2. Ricostruire il sorgente XHTML la cui interpretazione tipografica, per mezzo di un *browser* internet, sia la seguente:



3. Scrivere la grammatica DTD del sottoinsieme di documenti XHTML con le seguenti caratteristiche:

   - ogni documento ha radice `html`, seguita da esattamente due figli: `head` e `body`;
   - il `body` può contenere solo una sequenza di lunghezza arbitraria, quindi, anche vuota, di paragrafi;
   - ogni paragrafo `p` è definito con la clausola:

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

   Assumiamo che il paragrafo abbia l'attributo, opzionale, `align` che possa assumere solo i due valori `center` e `left`.

D.4 Sintassi e Formalizzazione

1. Sia dato il seguente documento:

```
<?xml version="1.0"?>
 <cml>
  <molecola titolo="acqua>
   <atomi>
    <matrice BUILTIN="ELSYM">H O H</matrice>
   </atomi>
   <legami>
    </matrice BUILTIN="ATID1">1 2<matrice>
    <matrice BUILTIN="ATID2">2  3<matrice>
    <Matrice BUILTIN="ORDER">1  1</matrice>
   </legami>
 </cml>
```

   Fornirne due versioni distinte, entrambre sintatticamente corrette, rispetto alla sintassi XML.

2. Dato il documento XML:

```
<?xml version="1.0"?>
 <c>
  <d>
   Descr.
  </d>
  <u v="X"/>
  <p i="A" f="B"/>
  <e>
   <l f="F" g="G"/>
   <t>S</t>
  </e>
 </c>
```

   scriverne l'albero di sintassi astratta.

3. Data la formulazione del seguente concetto, scrivere una istanza di documento XML, sintatticamente corretto, che lo formalizzi:

   - Un libro ha un titolo ed è costituito da una sequenza non vuota di capitoli;
   - Un capitolo ha un numero d'ordine, un titolo ed è composto da una sequenza di sezioni;
   - Una sezione ha un numero d'ordine, un titolo ed un corpo, costituito da testo.

D.5 Semantica e Proprietà invarianti

1. Scrivere, passo per passo, una sequenza di generazione della seguente espressione regolare deterministica: `(a,(b|c)*)`.

2. Sintetizzare il documento DTD più restrittivo alla cui semantica appartenga il seguente documento XML:

```
<radice>
 <e1/>
 <e1/>
 <e2>A<e2/>
</radice>
```

   Variante. Il numero di istanze XML da usarsi per ricavare la grammatica DTD, in generale, può essere arbitrario.

3. Dato il testo seguente con la descrizione delle proprietà invarianti di un concetto, formalizzarlo un documento DTD "equivalente":

- Un libro ha un titolo ed è costituito da una sequenza non vuota di capitoli;
- Un capitolo ha un numero d'ordine, un titolo ed è composto da una sequenza di sezioni;
- Una sezione ha un numero d'ordine, un titolo ed un corpo, costituito da testo.

D.6 XHTML: soluzioni

1. ```
   <html>
   <header/>
   <body>
    testo testo
    <ul>
     <li>testo testo</li>
     <li>testo testo</li>
    </ul>
   </body>
   </html>
   ```

2. ```
   <html>
   <header/>
   <body>
    <h1>Titolo Testo</h1>
    <h2>Titolo I capitolo</h2>
      <h3>Titolo sezione I.1</h3>
      <h3>Titolo sezione I.2</h3>
    <h2>Titolo II capitolo</h2>
      <h3>Titolo sezione II.1</h3>
      <h3>Titolo sezione II.2</h3>
   </body>
   </html>
   ```

3. La semantica del sottoinsieme di documenti XHTML è la seguente:

   ```
   <!DOCTYPE html [
    <!ELEMENT html (head,body)>
    <!ELEMENT head ANY>
    <!ELEMENT body (p*)>
    <!ELEMENT p (#PCDATA)>
     <!ATTLIST p align (center | left) #IMPLIED>
   ]>
   ```

D.7 Sintassi e Formalizzazione: soluzioni

1. Seguono le due proposte:

   (a) La prima, oltre alle altre quattro correzioni necessarie, chiude l'elemento `molecola` in modo da includervi la descrizione degli atomi che la compongono ed i legami relativi:

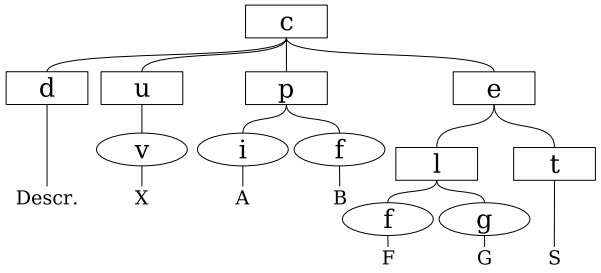
   ```
   <?xml version="1.0"?>
    <cml>
     <molecola titolo="acqua">
   ```

```
 <atomi>
  <matrice BUILTIN="ELSYM">H O H</matrice>
 </atomi>
 <legami>
  <matrice BUILTIN="ATID1">1 2</matrice>
  <matrice BUILTIN="ATID2">2  3</matrice>
  <matrice BUILTIN="ORDER">1  1</matrice>
 </legami>
</molecola>
</cml>
```

(b) La seconda, oltre alle altre quattro correzioni necessarie, chiude l'elemento `molecola` appena possibile:

```
<?xml version="1.0"?>
 <cml>
  <molecola titolo="acqua"/>
   <atomi>
    <matrice BUILTIN="ELSYM">H O H</matrice>
   </atomi>
   <legami>
    <matrice BUILTIN="ATID1">1 2</matrice>
    <matrice BUILTIN="ATID2">2  3</matrice>
    <matrice BUILTIN="ORDER">1  1</matrice>
   </legami>
 </cml>
```

2. L'albero di sintassi astratta cercato è:



3. Nella soluzione che segue il nome di elementi ed attributi, ed il valore di questi ultimi, non sono rilevanti. Al contrario, al fine della correttezza è rilevante la stutturazione degli elementi e la loro relazione reciproca:

```
<libro>

 <capitolo ordine="2">
  <titolo valore="X"/>
```

```
 <sezione ordine="1">
  <titolo valore="X1"/>
  <corpo>Testo Testo Testo ... </corpo>
 </sezione>

 <sezione ordine="2">
  <titolo valore="X2"/>
  <corpo>Testo Testo Testo ... </corpo>
 </sezione>
</capitolo>

<capitolo ordine="1">
 <titolo valore="Y"/>
</capitolo>

</libro>
```

D.8 Semantica e Proprietà invarianti: soluzioni

1. `E --> (E) --> (E,E) --> (E,(E)*) --> (E,(E|E)*)`
   `--> (a,(E|E)*) --> (a,(b|E)*) --> (a,(b|c)*)`

2. La seguente semantica è la più restrittiva perché lascia arbitrarietà solo rispetto al contenuto di `e2`, che, tuttavia, non è arbitrario in senso assoluto, dato che ha tipo `PCDATA`, e non `ANY`:

```
<!DOCTYPE radice [
 <!ELEMENT radice (e1,e1,e2)>
 <!ELEMENT e1 EMPTY>
 <!ELEMENT e2 (#PCDATA)>
]>
```

3.
```
<!DOCTYPE libro [
 <!ELEMENT libro (titolo,capitolo+)>
 <!ELEMENT capitolo (titolo,sezione*)>
  <!ATTLIST capitolo ordine ID #REQUIRED>
 <!ELEMENT sezione (titolo,corpo)>
  <!ATTLIST sezione ordine ID #REQUIRED>
 <!ELEMENT titolo (#PCDATA)>
 <!ELEMENT corpo (#PCDATA)>
]>
```