Esperanto con XSL

Giampaolo Bottoni
CILEA, Segrate

Abstract
L'XSL è un linguaggio basato sull'XML e pensato per ristrutturare documenti XML. Questo articolo intende presentarlo in uno stile familiare per gli utenti dei supercalcolatori del CILEA utilizzando un esempio didattico insolito ed apparentemente tra i più impegnativi: un traduttore pilotato da italiano in esperanto. Ovviamente il traduttore ha funzionalità molto ridotte ma non totalmente banali e la sua implementazione fornisce spunti per evidenziare caratteristiche dell'XSL di tutto rilievo come linguaggio di programmazione "sui generis". Lo scopo è dunque duplice: smentire il pregiudizio di chi considera l'XSL un linguaggio di applicabilità limitata all'editoria sul WWW e far conoscere i pregi di un linguaggio "informatico ante litteram" ormai centenario ma rinverdito dalla nascita di nuovi contesti applicativi (internet, integrazione europea) impensabili all'epoca di Zamenhof, il suo inventore.

Keywords: Supercalcolo, Didattica on-line, Portale, Linguistica computazionale, Programmazione XML, XSL.

L'esperanto è una lingua dalla grammatica molto semplice e regolare mentre l'XSL è un potente linguaggio dedicato alla rielaborazione di documenti XML. Sembra dunque logico sperimentare la potenza dell' XSL nella rielaborazione di un documento che dovrà *apparire* scritto direttamente in esperanto ma che, in realtà, è frutto di una manipolazione del documento scritto in *italiano a tag* e trasformato, via XSL, in modo da apparire tradotto in esperanto: una traduzione pilotata da italiano in esperanto, fatta tramite l'XSL.

Per pilotare il file XSL che, nello specifico, si chiama **esperantx-zv001.xsl** (il numero 001 potrebbe indicare il numero della versione del traduttore XSL-esperantesco) bisogna scrivere un documento che faccia uso di una serie di marche progettate per il fine desiderato. Bisognerà inoltre seguire una serie di regole grammaticali che derivano in parte dalle regole grammaticali dell'esperanto stesso ed in parte dalla struttura dell'XSL che rende facili le manipolazioni del testo se i nodi sono strutturati in un dato modo piuttosto che in un altro.

Consideriamo innanzi tutto la struttura che deve avere il tipico documento XML da gestire con il traduttore XSL-esperantesco.

```
<?xml version='1.0'?>
<?xml-stylesheet type="text/xsl"
                 href ="esperantx-zv001.xsl"  ?>
<!DOCTYPE doc [ <!ENTITY biblioteca
     SYSTEM "esperantx-itlib-zv001.xml" >] >
<doc>
&biblioteca;
...mettere qui le marche esperantesche...
</doc>
```

In sostanza, mentre **esperantx-zv001.xsl** rappresenta il file XSL che deve attuare la traduzione, il file **esperantx-itlib-zv001.xml** rappresenta la libreria di vocaboli del vocabolario italiano-esperanto usato per fare la traduzione. Anche in questo caso il numero 001 indica quale è la versione del vocabolario (in esperanto vocabolario si dice *vortaro*) che si vuole utilizzare.

In altre parole, ogni documento contenente le marche dell'italiano a tag, deve essere un file XML dotato di un DTD in cui si definisce come ENTITY il file XML contenente il vocabolario italiano - esperanto e il file XML deve avere la "Processing Instruction" che fa riferimento al file XSL che attua materialmente la traduzione.

Osservazione: normalmente un DTD (Document Type Definition) serve per rendere un file XML non solo "ben formato" ossia scritto in modo formalmente corretto, ma anche "valido" ossia conforme alle specifiche contenute nel DTD che può essere sia esterno al file XML sia interno, come suggerito qui. Nel nostro caso il DTD manca di

ogni indicazione su quali debbano essere le marche usate nel file XML e dunque il file XML non può essere valido! Qui però il DTD viene usato solo per includere nel file XML un file esterno ossia il file della libreria contenente il vocabolario italiano - esperanto. L'operazione di inclusione si attua definendo una ENTITY che indica dove sta il file che funge da vocabolario e specifica il nome da usarsi per includere il file. Qui il nome scelto è `biblioteca` e dunque scrivendo come prima frase attiva del file XML la citazione dell'entità esterna ossia: `&biblioteca;` si attua la voluta azione di fusione tra il vocabolario e tutto il resto del documento che fa uso del vocabolario. Il file XSL vede dunque un unico gigantesco file in cui una grossa parte invisibile serve come fonte di dati per la traduzione del testo da tradurre.
Come si è visto, la traduzione si basa su tre file di cui solo uno diverso di volta in volta:

1. un file XML contenente il testo da tradurre scritto in italiano a tag e diverso di volta in volta;
2. un file XML contenente il vocabolario italiano esperanto che viene modificato solo per essere arricchito di nuovi vocaboli;
3. un file XSL contenente le regole seguite per attuare il processo di traduzione che resta immutato ma che potrebbe anche venire modificato per migliorare il processo logico di traduzione.

Bisogna dunque spiegare quali sono le regole grammaticali accettate nell'italiano a tag (versione 001) e come deve essere realizzata la libreria italiano-esperanto che può essere ampliata con l'inclusione di nuovi lemmi e sottolemmi.

Italiano a tag

Con questo termine si indicano le regole che vanno seguite per rendere comprensibile la frase italiana all'azione del traduttore italiano - esperanto.

Una traduzione grossolana potrebbe consistere nella sostituzione, uno ad uno, dei vocaboli italiani con quelli esperanteschi. Già questo tipo di rozzo traduttore avrebbe da gestire un grosso problema: l'identificazione delle varie parole italiane. L'italiano è la mia lingua preferita [;-)], ha una corrispondenza tra testo e pronuncia *quasi* perfetta (ci sono sì alcuni trabocchetti: glicine o glucosio ma aglio e figlio, oppure gli accenti, àncora e ancòra) ma privilegia la musicalità rispetto all'osservanza delle regole: i verbi irregolari italiani sono un esercizio mnemonico sado-maso a cui si deve sottomettere chi, magari solo per amore del fascino turistico e storico italiano, voglia imparare la lingua di Dante. Dunque un traduttore o gestisce l'estrosità dei verbi italiani con un cumulo di regole a cui c'è sempre qualche eccezione o... affronta il problema con la forza bruta informatica ovvero gestisce un grosso vocabolario in cui sono raccolte tutte ma proprio tutte le parole italiane: singolare senza alcun legame col plurale, coniugazione dei verbi strampalata. L'idea del supervocabolario di tutte le varianti delle parole italiane sembra peregrina ma... non lo è. Ho fatto una indagine sui "Promessi Sposi" manzoniani. Se si considerano distinte tutte le sequenze di caratteri delimitate da spazi bianchi o dalla punteggiatura si hanno... 20000 vocaboli distinti. Una cifra ridicolmente piccola che smentisce il pessimismo di chi, facendo distinzione tra "ho", "hai", "avessi", "avessero", "ebbi", "avremmo", "hanno", tende a credere che i vocaboli diversi siano milioni o decine di milioni.
Del resto se anche fosse così, cosa sono 10 milioni di parole, 100 milioni di bytes a confronto della capienza di un comune CD ?
Ma il bello è che siamo ben lungi da questa situazione. Con 20000 parole si cataloga l'intero libro dei Promessi Sposi e poi, analizzando altri testi il numero certo cresce ma... sempre più lentamente ed ho motivo di credere che un catalogo di 200 mila parole sarebbe ampiamente adeguato non solo al riconoscimento di frasi giornalistiche ma anche di frasi da romanziere o da relazione scientifica non specialistica.

Dunque consideriamo superato il problema dell'irregolarità linguistica dell'italiano immaginando di avere un file XML, di adeguate dimensioni, che fornisca, per ogni persona della coniugazione irregolare, l'infinito del verbo a cui fare riferimento.

Vediamo ora quali informazioni gestire perché richieste dall'esperanto.
Il fatto che sia possibile spiegare qui, in poche righe, le informazioni grammaticali necessarie all'esperanto chiarisce il perché sia sensato tentare di realizzare in XSL un traduttore italiano-esperanto e non... esperanto-latino o esperanto inglese. In esperanto non ci sono eccezioni grammaticali e dunque la pagina del traduttore XSL potrà essere di 50 KByte non di 500 o di 5 MegaByte!
Vediamo alcuni punti qualificanti della grammatica esperantesca:

- L'articolo determinativo (è *la*) è indeclinabile. Non esiste un articolo indefinito (un, uno, dei). Se manca l'articolo determinativo si può immaginare che sia presente un articolo indefinito virtuale.
- I sostantivi si accordano con gli aggettivi solo per il numero e la declinazione. Dunque non esistono forme femminili dell'aggettivo. Gli aggettivi e i sostantivi possono essere solo singolari o plurali (non esiste il duale come in greco...)
- A parte il nominativo che è retto anche dalla stragrande maggioranza delle preposizioni, l'unica altra declinazione è l'accusativo che individua l'oggetto che subisce l'azione del verbo transitivo.
- I verbi possono essere transitivi (quando l'azione attuata dal soggetto opera sull'oggetto), intransitivi (quando l'azione non richiede un oggetto) e riflessivi quando un verbo transitivo fa coincidere il soggetto con l'oggetto.
- I pronomi sono i tipici sei (noi, voi, essi anche in variante singolare)
- Gli aggettivi possessivi non ammettono l'articolo determinativo (mio, non il mio; tuo, non il tuo...).
- La frase interrogativa deve essere esplicitamente riconoscibile in base ad una parola di valore interrogativo (chi?, che_cosa?, di_che_tipo?, di_chi?, dove?, come?, quanto?, perché?, quando?) o da una parola che ha come unico scopo quello di segnalare la domanda (in italiano non esiste, potrebbe essere indicata da forse_che?).
- I verbi hanno coniugazioni che non dipendono dalla persona che, viceversa, deve essere sempre specificata esplicitamente (eccezione l'imperativo per il quale il tu può essere sottinteso). Notare che non esiste il congiuntivo. A seconda dei casi il congiuntivo va sostituito da un'altra coniugazione.
- I verbi (le azioni) hanno sei coniugazioni (presente, passato, futuro, condizionale, imperativo e infinito) due participi (attivo e passivo) che si ripartiscono in tre varianti temporali (passato, presente e futuro) e in tre modi (sostantivo, aggettivo e gerundio altrimenti detto aggettivo di una azione).
- Gli stati (ovvero ciò che non rappresenta una azione) sono tre ovvero lo stato di sostantivo, di aggettivo e di avverbio (altrimenti detto aggettivo di una azione).
- Le parole hanno una struttura modulare per cui molte parole possono essere considerate delle perifrasi pronunciate con un unico vocabolo (simile all'italiano maldicenza == dire male, oppure anteposto == posto davanti: in italiano sono casi rari mentre in esperanto molte parole sono l'unione di varie radici che concorrono a chiarire il significato della parola nel suo complesso).
- Ogni parola è costituita da una o più radici e può iniziare con un prefisso e terminare con un suffisso.
- Esistono parole indeclinabili. I numeri cardinali e le preposizioni sono esempi tipici di parole indeclinabili.
- Le parole declinabili sono costituite da almeno una radice eventualmente preceduta da un prefisso e necessariamente conclusa da un suffisso.
- Compito dei suffissi è quello di specificare il numero (singolare o plurale), la declinazione (nominativo o accusativo), la coniugazione del verbo, la natura dell'oggetto, ossia se verbo, avverbio, sostantivo aggettivo.
- I prefissi sono alcune parole speciali che servano a caratterizzare meglio la parola.
- Le preposizioni e gli avverbi nativi sono parole indeclinabili che declinano gli stati costituiti da insiemi di aggettivi e sostantivi o servono a strutturare la frase in frase reggente e frasi secondarie e perifrasi.

Se questo elenco delle caratteristiche dell'esperanto può sembrare lungo, anche se necessario per capire cosa fa il traduttore XSL, va osservato che... questi punti sono straordinariamente pochi se si pensa che rappresentano praticamente **tutte le regole grammaticali dell'esperanto** e che queste regole **sono sempre rispettate** senza eccezioni!
La marca più importante che il traduttore riconosce si chiama, non per caso, `<e>` ossia come l'iniziale di esperanto. La sua definizione formale è la seguente:

```
<e [n="[-]#[.#]"]>
    [chiave1[(#)|]...chiave9[[(#)|[comando][=?]]
</e>
```

dove le parti opzionali sono racchiuse tra parentesi quadra; con il simbolo # si indica una o più cifre decimali, con *chiave1* si indica una parola chiave presente nel vocabolario. In realtà qui si è indicata una sola chiave prefissa ma in realtà ce ne possono essere varie, *chiave2*, *chiave3* etc. fino a quella che va considerata l'ultima della lista di chiavi ossia *chiave9*. Ogni chiave viene separata dalla successiva dal carattere "|". Con *comando* si intende una indicazione grammaticale che consente di selezionare il suffisso

opportuno esperantesco che concretizza l'indicazione grammaticale; infine *=?* rappresenta un segnale per fare in modo che venga stampata non solo la radice selezionata da *chiave9* ma tutto quanto è contenuto nel *vortaro* a chiarimento del significato della *chiave9* e per facilitare meglio la scelta del numero del lemma. Il numero del lemma va eventualmente scritto prima del carattere "|" e racchiuso tra una parentesi tonda aperta e la barra; in altre parole è rappresentato da *(#|* dove al solito il carattere # indica una o più cifre decimali ossia la posizione della radice nel lemma indicato dalla chiave. Ovviamente ogni chiave della lista, *chiave1*, *chiave2*,... *chiave9* può richiedere la precisazione del numero del lemma a cui fare riferimento.
Esempi di nodi <e> :

```
<e>quando|-</e>
<e>ramo(2|nome=?</e>
<e>uomo|femmina|nomi</e>
<e n="5">hanno|imperativo</e>
```

Nel primo esempio il comando usato indica che la radice è indeclinabile.
Nel secondo esempio viene usato il secondo lemma della chiave ramo, il comando è nome cioè la radice viene usata come sostantivo singolare e viene richiesta la visualizzazione di tutto quello che il vocabolario contiene a riguardo della chiave ramo.
Nel terzo caso viene costruita la parola corrispondente a "donne"; si tratta di una parola composta dalla radice di "essere umano" a cui si aggiunge il concetto di "sesso femminile" e il tutto si completa con il comando che indica che si vuole un sostantivo plurale.

Nel quarto esempio si attribuisce un numero d'ordine al nodo (n="5") in modo che la sequenza delle parole possa essere cambiata; inoltre si usa la radice corrispondente al verbo "hanno" ma con coniugazione imperativa. Si noti che aver scritto "hanno" come chiave e non "avere" o "avessi" è del tutto indifferente poiché la chiave serve solo a individuare la radice appropriata ma la coniugazione è controllata dal comando.
Consideriamo ora la serie dei comandi riconosciuti attualmente dalla pagina XSL a proposito della marca <e>.

Si noti che tra i comandi mancano quelli che servono per costruire i participi attivi e passivi nelle tre forme di sostantivo, aggettivo e avverbio (che rappresenta il gerundio).

Dato che i participi ammettono il tempo passato, presente e futuro e che possono essere al nominativo o all'accusativo come pure al singolare o al plurale questa varietà di situazioni avrebbe causato una notevole crescita di comandi. Si è preferito dunque trattare i participi come radici ed includerli nel vocabolario.
Ovviamente questo comporta una certa prolissità di marche per specificare un dato participio ma... questa è solo una versione dimostrativa di traduttore realizzato con l' XSL.

Dopo aver illustrato la sintassi della marca più importante, la marca <e>, è ora opportuno elencare tutte le marche riconosciute dal traduttore.

Alcune di esse hanno solo funzione di formattazione del testo dato che le andate a capo, la punteggiatura etc. devono essere materialmente tradotte in comandi HTML per poter generare l'effetto grafico che ci si aspetterebbe da esse.

Marca	Spiegazione
<artcob>	Si tratta di una marca che non ha contenuto. Serve solo come esempio di come si possono fare marche che producono un effetto specifico e limitato: in questo caso viene generato l'articolo determinativo (la). Essendo però generato da una marca e non scritto direttamente, esso può venire spostato in base al valore dell'eventuale attributo n.
<e>	La sintassi di questa marca è già stata illustrata. Si tratta della marca che ha la funzione fondamentale: la costruzione della parola (vorto) esperantesca.
<p>	Serve per cambiare riga. Scrive un punto e inserisce un tag dell'HTML.
<pe>	Serve per ordinare le marche in senso crescente. Tutte le marche contenute vengono ordinate in base al valore dell'attributo n. I valori ammessi per n possono essere anche negativi e frazionari.
<s>	Serve per inserire uno spazio bianco.
<t>	Serve per inserire semplice testo nel punto voluto. Accetta l'attributo n="#" e dunque il testo può essere spostato nella posizione voluta con la marca <pe>.
<v>	Serve per inserire una virgola seguita da uno spazio bianco

Comando	Spiegazione
nome	Sostantivo singolare declinato al nominativo o retto da preposizione
nomi	Sostantivo plurale declinato all'indicativo o retto da preposizione
onome	Sostantivo singolare declinato all'accusativo
onomi	Sostantivo plurale declinato all'accusativo
aggettivo	Aggettivo singolare declinato al nominativo o retto da preposizione
aggettivi	Aggettivo plurale declinato al nominativo o retto da preposizione
oaggettivo	Aggettivo singolare declinato all'accusativo
oaggettivi	Aggettivo plurale declinato all'accusativo
avverbio	Avverbio normalmente attribuito ad un verbo
oavverbio	Avverbio con concetto di moto a luogo (avanti con idea di andare avanti e non di stare avanti)
presente	Verbo indicativo presente
passato	Verbo indicativo passato
futuro	Verbo indicativo futuro
condizionale	Verbo al condizionale
imperativo	Verbo all'imperativo
infinito	Verbo all'infinito
-	Nessun suffisso dopo la radice. La parola è indeclinabile. Dopo la radice viene però messo uno spazio bianco di separazione.
+	Nessun suffisso dopo la radice. La radice deve servire come elemento compositivo di una parola costituita da una perifrasi. Dopo la radice non viene messo neppure uno spazio bianco per consentire di concatenare la radice con la parola successiva.

Va osservato che l'attributo n è ammesso per tutte le marche qui illustrate. Dunque se un insieme di nodi è contenuto entro un nodo <pe> tale insieme verrà automaticamente ordinato in senso crescente dell'attributo n.

In conclusione dunque si è visto che la pagina XSL deve gestire alcuni nodi di formattazione ed il nodo <e> che, oltre ad essere caratterizzato come tutti gli altri nodi dall'attributo n, è dotato di un valore costituito da una lista di chiavi, eventualmente accompagnate ciascuna da un numero di lemma e la lista può concludersi con una parola che funge da comando per la selezione del suffisso e da un indicatore di assistenza per ottenere la stampa di tutto quanto riguarda l'ultima chiave della lista di chiavi.

Vocabolario italiano-esperanto

Come si è visto ogni documento contiene un certo numero di marche <e> caratterizzate da una lista di chiavi che vanno esplicitate con l'aiuto di un opportuno vocabolario. Il vocabolario deve essere contenuto in un file XML e la sua radice, che può essere diversa dalla radice del file XML, si deve chiamare <vortaro>.

Per evitare spiegazioni magari prolisse cerchiamo di capire, innanzi tutto, questo esempio pratico di file XML contenente un vocabolario valido.

```
<?xml version='1.0'?>
<vortaro>
<grammatica>
<a t="uomo"><r>viro</r> Dal latino
        vir, italiano:   virile |
            <r>homo</r> persona |
            <r>E.T.</r> marziano
</a>
<a t="femmina"><r>ino</r> Per fare il
      femminile  dei sostantivi maschili</a>
<a t="cucciolo"><r>ido</r></a>
<a t="essere"><r>esti</r> Il verbo essere,
      cardine anche in esperanto</a>
</grammatica>
<a t="ebbi" a="avere" />
<a t="avessero" a="avere" />
<a t="ragazzo"><r>knabo</r></a>
<a t="avere"><r>havi</r> Qui puntano le
          varianti italiane: hanno, ebbi..
</a>
<a t="fossero" a="essere" />
...etc...
</vortaro>
```

In questo esempio il nodo <vortaro> rappresenta direttamente la radice del file XML ossia non è, come avrebbe potuto essere, un qualche nodo figlio o discendente della radice.
Quello che dovrebbe saltare maggiormente agli occhi dovrebbe essere il ruolo importante dei nodi di nome <a>. Sono nodi che hanno sempre un attributo t e qualche volta anche un attributo a. I nodi <a> contengono dei nodi <r> che a loro volta contengono testo in esperanto. Oltre a contenere questi nodi <r>, i nodi <a> possono contenere testo vario in italiano che, in qualche modo, aiuta a capire il significato del testo in esperanto contenuto dai nodi <r>.
Si noti che i nodi <a> possono essere figli diretti del nodo <vortaro> o possono anche essere figli di un qualche discendente del nodo <vortaro> come qui, per esempio: alcuni sono figli del nodo <grammatica> che, a sua volta, è figlio del nodo <vortaro>.

Nell'esempio ora analizzato, il file XML risulta ben formato ma non valido poiché manca un qualsiasi DTD con cui validarlo ossia da usare come specifica delle marche ed attributi il cui uso è consentito. Non è necessario corredare il file XML con un DTD per poterlo usare da vocabolario italiano-esperanto ma anche un semplice DTD può servire a chiarire meglio le regole che vanno rispettate nella realizzazione del vocabolario.
Ecco dunque uno dei DTD che potrebbe essere associato al file XML:

```
<!DOCTYPE vortaro [
<!ELEMENT vortaro ANY>
<!ELEMENT grammatica ( a+ )>
<!ELEMENT a (#PCDATA | r )* >
<!ATTLIST a t CDATA #REQUIRED
                    a CDATA #IMPLIED >
<!ELEMENT r (#PCDATA) >
]>
```

Naturalmente sarebbe possibile realizzare molte altre versioni di DTD che possano servire allo scopo. Questa è stata scritta in modo da essere piuttosto breve ma abbastanza completa per consentire a chiunque di capire come funziona un qualsiasi DTD e cosa permette di specificare.

Innanzi tutto si noti che il DTD inizia con la direttiva !DOCTYPE che, considerata come una specie di commento, ingloba tutte le altre parti del DTD. La direttiva specifica che il DTD si riferisce ad un documento che abbia come radice un nodo di nome <vortaro>. Subito dopo <vortaro> c'è una parentesi quadra aperta che si chiude solo alla fine del DTD e che circoscrive tutte le definizioni di elementi (ossia le marche ammesse) e di attributi degli elementi.
Ovviamente, poichè il DTD è stato fatto per un file XML che abbia come radice un nodo di nome <vortaro> occore specificare quali sono gli

attributi del nodo <vortaro> e quali sottonodi possono essere figli o discendenti del nodo <vortaro>. La strategia che si è seguita è stata quella del massimo *permissivismo* possibile ossia di imporre solo quello che è indispensabile rendere vincolante. Dunque, come si legge nella prima dichiarazione di !ELEMENT il nodo vortaro può avere come nodi figli qualsiasi tipo di nodo e in qualsiasi ordine: infatti abbiamo scritto la parola magica ANY. Inoltre, poiché manca una dichiarazione !ATTLIST che riguardi il nodo <vortaro> esso non può avere nessun attributo. Naturalmente si sarebbe potuto consentire un certo numero di attributi, mettiamo un attributo per specificare la data o la versione del vocabolario, ma questi sono perfezionamenti non indispensabili e quindi sono stati tralasciati.

Subito dopo la dichiarazione !ELEMENT per la marca <vortaro> c'è quella riguardante la marca <grammatica>. In realtà non ci sarebbe stata nessuna necessità di usare questa marca ma si voleva sottolineare il fatto che i nodi <a> possono essere non solo figli diretti del nodo <vortaro> ma anche suoi discendenti.
Per la marca <grammatica> invece di essere permissivi e dunque scrivere ANY si è imposto che il solo tipo di nodo figlio debba essere un nodo di tipo <a>. Si tratta di un vincolo che serve qui solo per far vedere come si fa ad imporre vincoli di questo tipo. Il carattere "+" che accompagna il nome del nodo figlio ammesso serve ad imporre che il nodo <grammatica> debba contenere "parecchi ma ALMENO un nodo <a>" ossia non può essere "vuoto" di nodi <a>.

A questo punto ecco la parte dedicata a come deve essere fatto un nodo <a>. Da quanto scritto si deduce che esso può contenere del testo sciolto inframezzato da nodi di tipo <r>. Però va anche notato che c'è un asterisco dopo la parentesi tonda e questo vuol dire "parecchi ma anche ZERO tipi di quanto precede", ossia il nodo <a> potrebbe essere anche un nodo vuoto di testo sparso e privo di nodi <r>. Questo perché il nodo <a> serve sia come fonte diretta di radici di esperanto sia come nodo di reindirizzamento quando si vuole associare una voce italiana ad una altra, mettiame reindirizzare "hanno" all'infinito del verbo ossia ad "avere".
La direttiva !ATTLIST che segue immediatamente serve a specificare che il nodo <a> deve obbligatoriamente avere un attributo di nome t (lo si dichiara #REQUIRED) e può facoltativamente contenere un attributo di nome a (lo si dichiara #IMPLIED).
L'attributo di tipo t serve sempre perché il suo valore rappresenta la chiave di selezione del nodo <a> mentre l'attributo a serve solo per attuare il reindirizzamento ad un altro nodo <a>.

Per concludere, la direttiva !ELEMENT per definire cosa può contenere un nodo di tipo <r>: con #PCDATA si dichiara che esso può contenere stringhe di caratteri e questo è quanto serve dato che le radici esperantesche sono appunto stringhe di caratteri. In sostanza #PCDATA a proposito del contenuto dei nodi corrisponde a CDATA a proposito del valore degli attributi.

Dato che, arrivati qui, abbiamo tutte le informazioni su cosa il traduttore deve fare e su come sono memorizzati i dati del vocabolario siamo in grado di capire finalità e disponibilità e ci resta da capire solo un piccolo dettaglio [;-)]: il come farlo.

Trasformazioni XSL (*hic sunt leones*)

Naturalmente non è possibile chiarire qui ogni aspetto di XSL ma almeno... si cercherà di chiarire come funziona e di vedere come va utilizzato in questo caso pratico: si tratta di applicare cose utili in moltissimi altri contesti applicativi e dunque questi esempi dovrebbero essere di stimolo ad uno studio approfondito di XSL, fatto con metodo e non, come qui, a *spot*.
XSL è un linguaggio di programmazione costituito da istruzioni scritte in XML e fatto per generare un file XML utilizzando dati posti, prevalentemente, in un altro file XML.
Dato che il ruolo dell'XML è quello di fornire uno standard per la rappresentazione e strutturazione di dati alfanumerici il ruolo dell'XSL è molteplice: si potrebbe dire in parte *dominante* ed in parte *subalterno*.

Il ruolo è dominante quando il risultato finale è ottenuto come prodotto diretto di una trasformazione diretta di un file XML in... qualcosa d'altro ossia un file HTML destinato ad essere visualizzato da un browser (qui indispensabile Internet Explorer 6).

Dunque XSL gioca un ruolo dominante qui, nel trasformare l'unione di italiano a tag e vocabolario in una pagina HTML che il lettore potrebbe credere scritta direttamente in esperanto.
Tuttavia il ruolo di XSL che maggiormente dovrebbe interessare i programmatori è, secondo me, quella di linguaggio *subordinato* ossia nel ruolo di un algoritmo (molto complesso e articolato) utilizzato per trasformare un oggetto "documento XML" in un altro oggetto "documento XML" ma in un ambiente di programmazione tradizionale, C++, Java, VisualBasic, Pascal, Fortran, C#...
In effetti il ruolo stesso di XML sarebbe marginale se XML non rappresentasse una soluzione *trasversale* applicabile non in uno ma in tutti i linguaggi di programmazione usati dai programmatori. Ed in questo contesto, XSL, visto come algoritmo di manipolazione di dati XML contribuisce a rendere sempre più appetibile la realizzazione di programmi che dialogano col mondo esterno tramite file XML con grande flessibilità di struttura. Con XSL il file XML prodotto da una qualche altra applicazione potrà essere velocemente ristrutturato in modo da essere conforme alle regole della nostra applicazione.

Per dare concretezza al discorso e mostrare come il tutto si riduca a poche linee di programmazione riporto qui un frammento di documento HTML con istruzioni JavaScript.

```
<xml id="fonte">
<radice>
<nodo>Questo testo sta dentro un nodo </nodo>
</radice>
</xml>

<xml id="agente">
<?xml version='1.0'?>
<a:transform  xmlns:a=
      "http://www.w3.org/1999/XSL/Transform"
       version="1.0">
<a:template match="/">
<contenitore>  Contenitore di dati
<a:apply-templates/>
</contenitore>
</a:template>
<a:template match="nodo">
<blocco>
<a:apply-templates />
</blocco>
</a:template>
</a:transform>
</xml>

<xml id="prodotto">
<scatola>Scatola semivuota</scatola>
</xml>

<script>
function trasforma(){
   var ok=
      prodotto.loadXML(
          fonte.transformNode(agente));
   alert("Ottengo: "+ok);
       return 0;
    }
</script>
```

In questa tabella che dobbiamo pensare estratta da una pagina HTML abbiamo di fronte ben tre file XML incapsulati dentro la pagina HTML ed uno script con una funzione JavaScript. La riga *cruciale* è quella scritta in grassetto ma per apprezzare cosa fa è utile poter dare una occhiata ai tre file XML. Quello contenuto nell' *isola* di nome `fonte` contiene un nodo, di nome `<nodo>` il cui contenuto verrà inserito nel file XML dell' isola `prodotto` che al momento contiene un file che andrà distrutto. La trasformazione viene attuata usando il file XML contenuto nell'isola `agente`; questo file è un file XSL perché la sua radice ha un nome opportuno (ossia `transform` ma poteva anche essere `stylesheet`) e il suo spazio dei nomi (indicato da `xmlns:a="..."`) è proprio quello che il browser riconosce essere dei documenti XSL (ossia: http://www.w3.org/1999/XSL/Transform).

(Due note: i nodi XSL iniziano tutti per `a:` appunto per segnalare che sono loro quelli che operano sugli altri nodi che, viceversa, non fanno ma subiscono le trasformazioni. Nei testi si trova quasi ovunque `xsl:template` piuttosto che `a:template`; è solo una questione di gusti e qui si trova `a:template` invece che `pippo:template` solo per il fatto che `a:` è più breve di `xsl:` e a maggior ragione di `pippo:`.)

Essendo stato chiarito che l'XSL è un linguaggio di programmazione come tanti altri, come il Fortran o come il Lisp, occorre capire dove sta la sua testa e la sua coda ossia quali sono le sue function o le sue subroutine e quale function funge da main_program.

La citazione del Lisp è intenzionale: nel Lisp (propagandato a suo tempo come il *linguaggio dell'intelligenza artificiale*) dati e funzioni han-

no tutti la stessa struttura, una lista di nomi che possono avere valore di per sé od essere riferimenti ad altre liste. Nell' XSL che è un linguaggio fatto per processare file XML ed è a sua volta un file XML, le funzioni sono dei normali oggetti che popolano un normale file XML ossia nodi dotati di argomenti e in gradi di avere dei contenuti ossia dei sottonodi con specifici compiti.

Per comprendere più agevolmente quanto detto da qui in avanti sarebbe opportuno avere sotto gli occhi il listato del traduttore esperantesco ossia del file **esperantx-zv001.xsl**, anche se i punti qualificanti verranno riportati qui anche se a grosse linee.

Il processo esecutivo del processore XSL si capisce facilmente se lo si assimila al viaggio turistico di un *turista* che si sposta principalmente in auto, ma talvolta anche in aereo, sulle strade del paese che sta visitando. Il paese visitato dal *turista* è il file XML che fornisce la rete viaria del viaggio mentre il file XSL è il taccuino delle cose che il *turista* vuole fare se si imbatterà in certe cose. Principalmente il *turista* scriverà un suo resoconto di quanto ha visto e questo resoconto sarà, a modo suo, un altro paese ossia la versione, l'idea del mondo che il *turista* ha visitato e si è creato. Altra azione importante che il *turista* può decidere di fare è prendere l'aereo ossia decidere che quello che ha visto in un certo sito, ovvero nodo del paese, gli basta e vuole visitare un altro sito o una altra lista di di siti. Risulta quindi molto importante il modo in cui il *turista* può specificare dove andare, la o le destinazioni finali e le condizioni che devono sussistere perché il *turista* sbarchi ed inizi una nuova tappa del viaggio.
In altre parole, facendo riferimento alla situazione di un normale linguaggio di programmazione, per capire come evolveranno i calcoli bisognerà tener presente non solo del punto in cui è arrivata l'esecuzione ossia quale istruzione è attiva in un dato momento ma anche la *posizione* in cui si trova un ideale segnalibro che si sta muovendo sull'albero costituito dai nodi del file XML usato come fonte dei dati. Questo segnalibro (ovvero il *turista*) può attivare o, rigidamente, una data procedura (function o subroutine) o quelle function che richiedono, per operare, di usare un dato tipo di nodo e che vengono eseguite per tutti i nodi di quel tipo che sono figli del nodo in cui sta il segnalibro (ovvero *turista*).

Per figurarsi la cosa si immagini che un essere umano decida di fare una data operazione su tutte le immagini che stanno in una data directory del proprio disco fisso; mettiamo che voglia rimpicciolirle tutte per risparmiare spazio. Si porta nella directory voluta e qui guarda tutti i file che hanno estensione BMP e li trasforma uno per uno riducendone le dimensioni. Ha agito non in base a un programma definito ma in base a quello che ha trovato in quella directory. Vediamo ora come fare a parte le metafore...

Per dare un po' di sistematicità al discorso consideriamo questa tabellina che raccoglie gli elementi del linguaggio e li descrive in base alle normali istruzioni disponibile in un normale linguaggio di programmazione.

```
<a:variable name="xxx" select="yyy" />
```

xxx=yyy *ossia la definizione di una variabile e l'assegnazione di un dato valore alla variabile. Notare però che questa assegnazione può essere fatta SOLTANTO UNA VOLTA e dunque, per cambiare il valore della variabile non si può far altro che dichiarare una nuova variabile e dare ad essa il valore cambiato. Si capisce facilmente che XSL può fare anche calcolo numerico ma non è molto predisposto per farlo !!!*

```
<a:param name="..."  select="..." />
```

Si tratta dell'argomento di una function o subroutine che può essere, se si desidera, inizializzato a un qualche valore per difetto. Il valore per difetto viene però cancellato dal valore assegnato all'argomento al momento della chiamata (call). Dunque, è una variante della variabile XSL e quindi anche il param può possedere un unico valore che non può più venir modificato. Per cambiarlo bisogna dichiarare una nuova variabile e dare ad essa il nuovo valore!!!

```
<a:template match="..."  name="xxx" >
...
</a:template>
```

function xxx(yyy){ ... } *ossia rappresenta la definizione della procedura (function o subroutine di un normale linguaggio di programmazione). Mentre una function o subroutine deve avere necessariamente un nome per poter essere chiamata (call) il nome in XSL è opzionale perché la function (ovvero template) può essere attivato dal verificarsi di eventi come la presenza di un dato tipo di nodo figlio del nodo in cui sta il segnalibro o turista che dir si voglia*

```
<a:call-template name="fff" >
   <a:with-param name="xxx" select="$vvv" />
   <a:with-param name="yyy" select="$www" />
   ...
</a:call-template>
```

call fff(xxx,yyy,...) *ovvero chiamata di una subroutine o function con impostazione degli argomenti a determinati valori. Ovviamente se chiamo il template di nome fff passandogli il param xxx inizializzato al valore posseduto dalla variabile vvv occorre che il template abbia una definizione del param che gli viene passato; se tralascio di passare un dato parametro bisogna fare in modo che il param nel template sia dichiarato con un valore di difetto che serve in questi casi.*

Notare che si deve mettere un "$" davanti al simbolo delle variabili.
Notare anche che l'espressione assegnata ad ogni select può essere molto complessa ossia non essere una semplice espressione algebrica ma un vero e proprio criterio di selezione di certi nodi con certi tipi di attributi e contenuti.

```
<a:apply-templates select="...">
   <a:with-param name="xxx" select="$vvv" />
   ....
</a:apply-templates>
```

call?quelb_che_trovo?(xxx,...) *ovvero... è questa la vera originalità dell'XSL ossia la possibilità di chiamare templates alla spero_in_dio purchè ci siano i nodi che si adattano al template attivabile. Molto spesso la apply-templates viene usata senza alcuna select e senza nessun argomento with-param. In sostanza, si punta al fatto che nel nodo del file XML in cui è arrivato il turista ovvero il segnalibro, ci saranno solo certi tipi di nodi figli e se per qualcuno di essi c'è il template appropriato... ecco che verrà attivato il o i template appropriati.*
Con la select della apply-templates posso però non solo scendere lungo l'albero dei figli e discendenti del nodo in cui il segnalibro (o il turista) sta ma ... saltare a nodi completamente diversi. Per questo ho accennato alla metafora del turista che prende l'aereo e va a vedere quali template sono attivabili in nodi anche diversissimi da quelli in cui ha fatto la apply-templates.

```
<a:choose>
  <a:when test="...">
  ....
  </a:when>
  ....
  <a:otherwise>
  ....
  </a:otherwise>
</a:choose>
```

Si tratta del tipico costrutto per l'esecuzione condizionata, del genere dell' IF...THEN...ELSE...ENDIF del Fortran. Anche se qui non ho potuto descrivere tutti i costrutti dell'XSL mi sembra utile citare questo costrutto anche per ribadire lo stile XML assunto dalle istruzioni stesse dell' XSL. Noto anche questo aspetto importante per chi programma in XSL.
Le variabili XSL sono molto poco flessibili dato che accettano una sola assegnazione di valore. Tuttavia l'assegnazione puo' essere il risultato finale di una serie di attivazione di template controllati da test specificati con questo costrutto. In altre parole, al posto dei puntini può essere scritta una qualche call-template o apply template con la conseguenza che il risultato dei template attivati non costituirà l'output della trasformazione dell'XSL ma diventerà il valore (stabile) della variabile (si fa per dire).

<a:value-of select="..." />

In Fortran sarebbe ***write(unit=*,*)*** *...ossia il comando per trascrivere sull'unità attiva in quel momento il risultato della valutazione dell'espressione indicato dall'attributo* ***select***

In un normale linguaggio di programmazione le istruzioni sono separate le une dalle altre con particolari caratteri o con la semplice andata a capo, come nel caso del Fortran ma in XSL esiste un nuovo genere di dato ossia il testo *fuori* dai nodi ossia tra la marca di chiusura di un nodo e quella di apertura del nodo successivo. Tale testo, se non costituito da soli caratteri bianchi o da caratteri di cambio riga viene semplicemente trascritto nel file di uscita.
Se a questa nozione aggiungiamo quella di come riconoscere la `function main` ovvero quale template ha le stesse funzioni del `program` del Fortran possiamo cominciare ad individuare il punto da cui parte l'esecuzione e da lì analizzare le varie strade possibili.

```
<a:transform
   xmlns:a=
   http://www.w3.org/1999/XSL/Transform
   version="1.0">
                ...
<a:template match="/">
<a:apply-templates/>
<br/>Amen.
</a:template>
                ...
</a:transform>
```

Ecco il punto da cui parte tutto: il template che dichiara di individuare la radice del file XML che il *turista* deve esplorare ovvero quello in cui c'è scritto: `match="/"` dove la barra è il sinonimo di radice del file XML comunque si chiami la marca che funge da radice.
Notare che il nome di ogni marca, in XML si divide in due parti: il *cognome* ed il *nome*, separati dal carattere ":". Come nella vita comune, in XML il cognome può essere anche omesso ma nel caso dell'XSL no. L'XSL impone che il nome della radice non sia arbitrario ma solo, equivalentemente, `transform` oppure `stylesheet` ed occorre obbligatoriamente indicare anche quali marche vanno considerate marche di comando ossia quelle a cui il processore XSL deve obbedire e quali invece marche *passive*. Questo è necessario perché potrebbero esserci file XSL che generano come risultato file XSL e dunque usano marche dello stesso nome, alcune come marche di comando ed altre come marche di dati (che verranno solo in un secondo tempo trattate come marche di comando). La distinzione si ottiene indicando nel nodo radice, che si chiama `a:transform` (o, se avessi preferito, anche `tizio:transform`) che il cognome `a` rappresenta l'abbreviazione di "http://www.w3.org/1999/XSL/Transform" che è, in realtà, il vero nome di *casa XSL*.
Ecco dunque che il processore XSL non può aver più dubbi: se troverà una marca di nome `a:pippo` cercherà di fare quanto è previsto che

faccia col comando XSL `pippo` mentre se troverà una marca di nome `pluto` la tratterà come un semplice dato e controllerà solo che alla marca di apertura corrisponda l'adeguata marca di chiusura. Diventa dunque chiaro il perché il processore lascerà inerte il nodo `<br/>` mentre si darà un sacco da fare quando troverà il nodo `a:apply-templates` che dice al nostro "*turista XSL*" di attivare tutte le function (cioè i template) i cui requisiti di utilizzo sono soddisfatti dai nodi figli del nodo radice del file XML su cui il *turista* sta gironzolando.
Immaginando di avere inventato una variante di Fortran (dunque un linguaggio non fatto da istruzioni in stile XML ma di normali righe di testo, ci troveremmo di fronte a un sorgente del genere:

```
program transform
  do
    call procedura_adatta_qui()
    if (mancano_procedure_adatte_qui()) then
       exit
    end if
  end do
  write(unit=*,fmt=*) "<br/>Amen"
  stop
end program transform
```

Ovviamente sarebbe un programma *un po' strano* che richiederebbe il concetto di posizione sull'albero XSL che il *turista* sta visitando... Comunque, ... per dare l'idea...
Il programma principale del traduttore in esperanto è stato definito così *di bocca buona* sul tipo di procedure da attivare perché, come si è visto, esistono molte specie di marche che possono esse utilizzate e ciascuna parecchie volte oltre che, cert'une in modo annidato ovvero come nodi figli o nipoti di nodi figli del nodo radice. Per ciascun tipo di marca: `<e>`, `<p>`, `<pe>`, `<v>` etc. bisogna dunque specificare una appropriata procedura (cioè un template) che sappia manipolare gli attributi e il testo contenuto nel nodo col nome della marca. Ovviamente la marca che richiede il template più complicato è quella di nome `<e>`, mentre altre marche sono molto facili da trattare e aiutano a capire come funziona XSL.
Guardiamo innanzi tutto come è fatto un template che *fortunatamente* non fa nulla. Abbiamo visto che ogni file XML trattato dal traduttore XSL deve essere l'unione del testo da tradurre e del vocabolario. Guai però se il vocabolario generasse un qualsiasi effetto diretto sul risultato! Come si è visto, si è deciso di mettere tutti i dati del vocabolario dentro un nodo di nome `<vortaro>` e quindi dobbiamo fare un template che si attivi in corrispondenza di marche `vortaro`:

```
<a:template match="vortaro">
    <!-- ora mi riposo -->
</a:template>
```

Ovviamente nel template potrebbe non esserci neppure la frase che, viceversa è presente solo per ricordare come vanno fatti i commenti in XSL: esattamente come in XML dato che un file XSL è una sottospecie di file XML.
Procedendo per gradi vediamo come trattare marche un po' più impegnative come quella destinata al nodo `<pe>`.La caratteristica di questo nodo è quella di consentire un riordino dei nodi figli in base al valore dell'attributo `n` specificato in ciascuno dei nodi figli. Il valore di `n` è trattato come un numero reale in virgola mobile e l'ordinamento deve essere, per scegliere uno dei tanti modi possibili, in senso crescente. Ecco dunque il template che soddisfa questi requisiti:

```
<a:template match="pe">
<!-- serve per ordinare i tag
       in base al valore dell'attributo n -->
<a:apply-templates>
     <a:sort data-type="number"
             select="@n"
             order="ascending"/>
</a:apply-templates>
</a:template>
```

La principale differenza rispetto al template che è servito da main-program è che in questo caso il contenuto del nodo `a:apply-templates` non è vuoto ma è costituito dal nodo figlio `a:sort` con cui è possibile ordinare i risultati in base al valore dell'attributo `n`. Notare il carattere `"@"` anteposto al nome dell'attributo e che consente di precisare che si vuole prendere in considerazione appunto un attributo. Omettendo il carattere chiocciola sarebbe stato selezionato non un attributo ma un figlio del nodo che sia caratterizzato dal nome `n`.

Si noti che l'istruzione `a:apply-templates` contenuta nel template si applica indistintamente a tutti i nodi figli del nodo `<pe>` che ha attivato il template. In altre parole questo vuol dire che un nodo `<pe>` può contenere come figlio un nodo `<pe>` che dunque opererà un riordino dei dati dei nodi propri figli e questo blocco di dati verrà a sua volta riordinato in base al valore del suo argomento `n` confrontato col valore di altri nodi fratelli caratterizzati da altri valori dei propri argomenti `n`.
Queste caratteristiche ricorsive potrebbero venire opportunamente trattate nell'ambito del

traduttore per traslocare di posto intere frasi secondarie in modo che il testo originario *suoni bene* nella lingua nativa, in questo caso l'italiano, mentre il risultato finale sia *gradevole* in esperanto.
I restanti nodi, ad esclusione del nodo <e>, sono ugualmente di natura elementare.
Per avere un esempio di nodo che non ha natura ricorsiva si consideri il seguente:

```
<a:template match="articolo">
     <i>la </i>
</a:template>
```

In sostanza questo template non vieta che il nodo articolo abbia un contenuto ma... ignora completamente tale contenuto e produce unicamente la parola la contenuta in un nodo <i> che, in HTML, serve per generare l'effetto del corsivo. Dato che il traduttore è fatto in modo che l'esperanto appaia in corsivo è logico che l'articolo determinativo sia appunto in corsivo.

Passiamo ora ad esaminare i template maggiormente impegnativi dal punto di vista concettuale, quelli che meglio evidenziano la potenza dell' XSL. Guardiamo innanzi tutto il template destinato a lavorare sui nodi di tipo <a> che, come si ricorderà contengono dati del vocabolario dimenticandoci di chiederci come diavolo sia possibile fare in modo che l'esecuzione riesca ad arrivare ad attivare i nodi del vocabolario se abbiamo appena visto che c'è uno sbarramento che impedisce al nostro *turista* di arrivare ai nodi del vocabolario. Si è visto infatti che il template destinato al nodo <vortaro> ha il compito di ... NON FAR NULLA e dunque il *turista*, il punto attivo dell'esecuzione, non potrà mai scendere ai nodi del vortaro passando attraverso l'antenato di tutti, il nodo <vortaro> !

```
<a:template match="a">
  <a:param name="rango" select="1" />
  <a:variable name="z"
    select="string(.//r[number($rango)])" />
  <a:value-of select="substring($z,1,
                    string-length($z)-1)" />
</a:template>
```

In termini di un linguaggio tradizionale tipo Fortran si può parlare di una subroutine senza nome ma che diventa attiva alla presenza di un nodo di nome <a>. Tale subroutine ammette un argomento di nome rango che però è opzionale e che, se non viene specificato nella apply-templates, assume il valore di 1 (notare: vale una stringa contenente il carattere che rappresenta la cifra 1).

La terza riga esemplifica cose molto importanti: si tratta di un esempio di definizione di una variabile, la variabile z. Si è già detto che, usando la terminologia tradizionale dei normali linguaggi di programmazione sarebbe stato meglio chiamarla *costante* poiché una variabile XSL, una volta dichiarata, non può più cambiare valore. Gli inventori dell'XSL l'hanno voluta chiamare variable e così sia. Quello che però è molto istruttivo è il modo con cui viene calcolato il valore da attribuire alla z: viene fatto uso di un linguaggio specifico, chiamato XPath concepito allo scopo di poter individuare il punto esatto del file XML da cui si vogliono estrarre valori numerici, di stringa o addirittura sottorami del file XML stesso.
Il fatto che z possa avere il valore di una intera struttura di nodi spiega il perché l'espressione "string(.//r[number($rango)])" sia fatta così: applicazione della function intrinseca di XPath string(...). Vogliamo cioè che z sia la rappresentazione sotto forma di stringa, del suo argomento. Altro esempio di applicazione di una funzione intrinseca è number($rango) dove si mostra come usare il valore posseduto dal parameter rango definito la riga precedente. Si usa la funzione number(...) per trasformare ovviamente in dato numerico il valore di $rango che altrimenti sarebbe una stringa fatta da una sequenza di cifre.
A questo punto, cosa fondamentale da capire, chiediamoci che cosa rappresenta, per esempio questa espressione:

```
.//r[3]
```

avendo magari immaginato che $rango valga "3" e che dunque number($rango) valga il numero 3.
Si tratta del nocciolo stesso del linguaggio XPath ossia il modo di indicare un particolare nodo della struttura del file XML su cui il *turista* sta gironzolando. L'espressione dice questo: partendo dal punto in cui sei (ovvero ".") esplora tutti i discendenti, figli, nipoti, bisnipoti del nodo in cui stai (ovvero "//") e conta i discendenti che hanno marca di nome "r". Trovato il terzo discendente con queste caratteristiche prendi quello che contiene.
Dovrebbe contenere solo testo ma se contenesse ulteriori sottorami non ci sarebbero problemi perché l'oggetto comunque ottenuto verrà trasformato in testo dalla funzione string(...)

messa lì appunto per dare la certezza che la variabile z valga sempre una stringa.
In sostanza il linguaggio XPath gestisce espressioni molto simili a quelle che si scrivono per specificare un qualsiasi file memorizzato sul disco rigido ma .... XPath usa operatori molto più potenti di quelli usati sulla riga comando di Unix o della finestra DOS in Windows.
Non è il caso di illustrare qui tutti i comandi e gli operatori ammessi da Xpath ma uno di questi mi sembra particolarmente semplice, potente ed interessante.
Quando usiamo il path di un file particolare possiamo scrivere, per esempio in DOS:

"C:/windows/web/bullet.gif"

per indicare il file bullet.gif posto in un particolare sottodirettorio:

"C:/windows/web/".

Immaginiamoci però come sarebbe bello poter scrivere, come si può fare con Xpath:

"C://windows//web//bullet.gif".

In questo caso avremmo dato un compito molto più laborioso al sistema operativo ma avremmo molta più flessibilità e probabilità di scovare il file bullet.gif. Infatti avremmo chiesto di guardare non soltanto i figli della radice del disco "C:/" ma tutti i discendenti, scendendo anche di mille livelli fino a trovare, eventualmente, una directory di nome windows/ e poi, trovatala, esplorare da lì in profondità fino a trovare una directory di nome web/ e da lì cercare non solo i file presenti nella directory ma in tutte le sottodirectory. Forse sono stato prolisso ma questo dovrebbe dare una idea nel notevolissimo carico di lavoro che il comando "//" produce rispetto al comando "/". La possibilità di dover fare molti tentativi prima di riuscire ad individuare un file che soddisfa il criterio di selezione è molto superiore. Ad esempio, potrebbero esserci molte directory di nome windows/ ovviamente non allo stesso livello ma figlie di genitori diversi, e così anche potrebbero esistere molte directory web/ discendenti da una stessa directory windows/e così via...

In conclusione, se si vuole dare un criterio di selezione molto flessibile ossia applicabile a file XML di struttura anche molto variata, è molto consigliabile utilizzare "//" in luogo di "/".
Per verifica: se guardiamo il micro esempio di <vortaro> riportato qui in precedenza, il comando ".//r[3]" darebbe come risultato "E.T." anche se il *turista* si trovasse non nel primo nodo <a> ma anche nel nodo <grammatica> o addirittura alla radice ossia in <vortaro>.
Anche l'ultima istruzione del template ha caratteristiche interessanti. In esperanto le parole sono costituite da una concatenazione di stringhe, le radici, e terminate da un suffisso che, nella maggior parte dei casi è un singolo carattere, "o" per i sostantivi, "a" per gli aggettivi, "i" per i verbi all'infinito ed "e" per gli avverbi. Nel file XML che funge da vocabolario, le traduzioni dei vocaboli italiani, sostantivi e aggettivi al singolare, verbi all'infinito, viene fatta riportando la parola completa mentre in pratica serve solo la radice ossia la parola privata dell'ultimo carattere. Questo spiega il perché l'attributo select ha quel valore: "substring($z, 1, string-length($z) -1)". Detto in parole, viene calcolata la lunghezza della stringa che rappresenta il valore di $z e viene usata la sottostringa costituita dai caratteri di $z dal primo al penultimo.
Il punto cruciale è la selezione dell'appropriato nodo di tipo <a>. Si risolve utilizzando la funzione key(...,...) che opera su stringhe che rappresentano nomi identificativi di chiavi. Chiameremo chiave la chiave di cui abbiamo bisogno e definiamo le sue caratteristiche tramite il nodo <a:key> che viene messo alla radice del file XSL, fuori da tutti i template:

```
<a:transform
      xmlns:a=
        "http://www.w3.org/1999/XSL/Transform"
      version="1.0">
<a:key name="chiave"
      match="//vortaro//a" use="@t"/>
                    ...
</a:transform>
```

L'attributo match consente di definire quali sono i nodi da prendere in considerazione e di conseguenza il tipo di nodo che verrebbe puntato dalla function key(...,...) se la chiamata avesse successo ovvero quandi si riesce a trovare almeno un nodo che soddisfi il criterio di selezione. Si noti il valore che viene dato a match. Nel nostro caso si richiede di scovare tutti i nodi <a> che siano non solo figli ma eventuamente discendenti di qualsiasi grado di nodi di nome <vortaro> i quali possono essere discendenti di qualsiasi grado del nodo che funge da radice del documento da visualizzare in esperanto.
Stiamo di nuovo sfruttando la potenza dell' operatore "//" del linguaggio XPath. I nodi <a> potranno dunque appartenere a vocabolari diversi; dunque non un solo, pur gigantesco voca-

bolario ma, eventualmente diversi vocabolari utilizzati in base alle esigenze del testo.
Se, per esempio il documento fosse un articolo scientifico di medicina che descrive la strategia di archiviazione di cartelle cliniche e analisi ecografiche si potrebbe rafforzare il vocabolario italiano esperanto con quello specifico di termini medici e con quello di termini informatici e questo senza dover modificare il nostro traduttore XSL.
Va osservato che è evidentemente semplice definire una chiave ma che l'attuazione del compito è tutt'altro che banale. I nodi di nome `<a>` del vocabolario italiano-esperanto saranno circa 30-50 mila e bisognerà tener conto anche del vocabolario di tutte le possibili parole italiane ossia il vocabolario sussidiario che potrebbe contenere oltre 100 mila nodi `<a>`. E questa ricerca andrà fatta per ogni radice che compone ciascuna parola esperantesca.
L'attributo `use` consente di specificare quale deve essere la stringa posseduta da un dato nodo `a`. Il valore `"@t"` significa che si deve usare l'attributo `t` del nodo. Il carattere chiocciola anteposto al nome indica che ci si riferisce al valore di un attributo e non a quello di un nodo.
Definita la chiave ecco come viene usata:

```
<a:template name="spezzoni" >
  <a:param name="chiaven" select="'?'" />
  ....
   <a:variable name="chiave" select=
 "substring-before(concat($chiaven,'('),'(')"
  />
    ....
  <a:variable name="yt"
        select="key('chiave',$chiave)" />
    ....
  <a:choose>
  <a:when test="$yt/@a">
   <a:call-template name="esploro">
     ....
  <a:with-param
       name="y"
       select= "key('chiave',string($yt/@a))"
       />
   </a:call-template>
  </a:when>
  <a:when test="$yt">
  <a:call-template name="esploro">
      ....
  <a:with-param name="y" select="$yt" />
  </a:call-template>
  </a:when>
  ....
  </a:choose>
</a:template>
```

Nell'ambito del template di nome `spezzoni`, privo dell'attributo `match` e dunque attivabile soltanto tramite una `<a:call-template>` la funzione `key(...,...)` andrebbe usata più di una volta e dunque è preferibile memorizzare l'eventuale nodo `<a>` individuato in una variabile di servizio di nome `yt`.
La funzione `key()` fa uso della chiave di cui si parla ed usa come stringa di confronto `$chiave` ossia la sottostringa che si trova prima del primo carattere `"("` del parametro `$chiaven` ricevuto in argomento dal template.
Poiché `$chiaven` potrebbe non contenere nessun carattere `"("`, viene fatta la concatenazione di `$chiaven` con tale carattere in modo da non rischiare mai di ottenere una stringa vuota.
Si osservi la doppia modalità di chiamata del templateesploro messa in opera nel nodo `a:choose`.
Se si verifica la condizione `"$yt/@a"` ossia se il nodo `<a>` possiede un attributo `a` viene passato all'esploro non il nodo stesso ma il frutto della nuova ricerca con chiave. Si ipotizza che questa seconda ricerca abbia sempre successo ossia che `"key( 'chiave', string($yt/@a))"` dia sempre un nodo valido. Ovviamente il vocabolario non deve contenere errori ossia se abbiamo incluso il nodo `<a t="fossero" a="essere" />` non bisogna dimenticarsi di definire il nodo `<a t="essere">`.

Essendo ormai stato chiarito come funziona l'attivazione dei nodi `<a>` non ci resta che analizzare il funzionamento del nodo di nome `<e>` che è reso complesso dal fatto che può contenere una stringa suddivisibile in tante parti (come carattere di separazione si usa `"|"`) e che, per ogni sottostringa, va fatta una ricerca della appropriata radice esperantesca.
Viene fatto uso del meccanismo della ricorsione nel seguente modo:

```
<a:template match="e">
  <a:variable name="ss"
      select="concat(string(.),'|')" />
  <a:variable name="xchiave"
      select="substring-before($ss,'|')"
      />
  <a:variable name="xdopo"
      select="substring-after(string(.),'|')"
      />
  <a:call-template name="spezzoni" >
     <a:with-param name="chiaven"
          select="$xchiave" />
     <a:with-param name="dopo"
          select="$xdopo" />
  </a:call-template>
</a:template>
                  ....
<a:template name="spezzoni" >
   <a:param name="dopo" select="'?'" />
   <a:variable name="poi"
      select="contains($dopo, '|')" />
      ....
  <a:if test="$poi" >
```

```
    <a:variable name="xchiave"
       select="substring-before($dopo,'|')"
      />
    <a:variable name="xdopo"
      select="substring-after($dopo,'|')"
      />
    <a:call-template name="spezzoni" >
      <a:with-param name="chiaven"
         select="$xchiave" />
      <a:with-param name="dopo"
         select="$xdopo" />
    </a:call-template>
  </a:if>
</a:template>
```

In sostanza il template `e` serve solo per inizializzare il processo ricorsivo che viene attuato dal template `spezzoni` che non possiede un attributo `match` e che può quindi essere attivato o da se stesso o, la prima volta, dal template `e` (chiamato impropriamente con questo nome: sarebbe stato più corretto ma più lungo parlare di template senza nome fatto per attivarsi in corrispondenza di nodi di nome `<e>`).
Anche qui si hanno esempi vari di come vanno usati parametri e variabili e di come si fruttano le funzioni per le operazioni su stringhe di XPath. L'unica considerazione specifica al processo ricorsivo in sé è il fatto che il test per proseguire o meno nel processo ricorsivo viene fatto solo in coda a tutto e consiste nella verifica della presenza o meno di un carattere "`|`" nel resto della stringa da analizzare. Dunque si può parlare di *ricorsione di coda* ossia di una modalità ottimale di eseguire il processo ricorsivo. In pratica, visto che una parola in esperanto è costituita al più da tre o quattro radici non ci sarebbe la stretta necessità di fare economie sul numero di dati da conservare sullo stack. Le caratteristiche ottimali servono qui solo dal punto di vista didattico.
A conclusione di questa analisi sul traduttore XSL vediamo alcuni aspetti del template `esploro` che ha il compito di generare materialmente il risultato ossia la parola in esperanto che va normalmente completata da un appropriato suffisso che determina la natura grammaticale della parola stessa.

```
<a:template name="esploro">
   <a:param name="oltre" />
   <a:param name="nv" />
   <a:param name="y" />
   <a:variable name="valido"  select=
"substring-before(concat($oltre,'=?'),'=?')"
     />
   <a:choose>
        ...
      <a:when test="$valido = 'nome'">
        <i><a:apply-templates select="$y">
             <a:with-param
                name="rango" select="$nv" />
          </a:apply-templates>o
      </a:when>
         ...
   </a:choose>
   <!-- Se c'e' =? allora
          deve scrivere le spiegazioni -->
   <a:if test="contains($oltre,'=?' )">
       [<a:value-of select="string($y)" />]
   </a:if>
</a:template>
```

Si può dunque vedere che l'attributo `$y` vale il nodo di nome `<a>` selezionato tramite le chiavi usate nel template `spezzoni`. Viene qui riportato il modo con cui si fa un sostantivo. Il sostantivo singolare è rappresentato dal suffisso `o`. Il parametro `rango` serve per specificare il lemma giusto dato che una data parola in italiano può avere varie traduzioni dato che, in italiano, è ambigua. Si pensi alle preposizioni italiane *a*, *di*, *da* che permettono di esprimere tutta una varietà di complementi che, in una lingua razionale come l'esperanto è bene esprimere con caratteri diversi (per esempio: a ==> *al, en, laŭ, pre ...*; da==> *de ,el, ĉe, tra, pro, per, kun, je, por* ... e così via..).
In italiano il significato di queste preposizioni va desunto dal contesto e questo rende molto più difficile il lavoro di traduzione completamente automatica che sarebbe viceversa facilitato se fatto tra lingue progettate per non avere se non rare ambiguità risolvibili col contesto.
L'esperanto dunque non rappresenta l'unica lingua progettabile né la migliore sotto ogni punto di vista ma è, certamente, una lingua molto più adatta di ogni lingua creatasi per azione storica ad essere processata in modo automatico e convertita in una qualche lingua adatta per scopi settoriali. Cercando un paragone espressivo, in termini informatici, l'esperanto avrebbe il ruolo svolto dai linguaggi di alto livello nei confronti dell'assembler e dei vari linguaggi macchina, molto efficienti ed efficaci ma anche molto specifici e finalizzati.
Questi ultimi sono dunque fatti per essere vicini alla macchina mentre l'esperanto è fatto per non creare barriere ed essere, in senso pratico ma anche in senso culturale e simbolico, *vicino all'uomo*.

Appendice: micro-stele di Rosetta

Riporto qui la traduzione dell'abstract (==*resumo*) in due *lingue ponte*: ovviamente l'inglese (mi spiace per il francese che studiai al liceo) e l'esperanto:

XSL is a language based on XML and designed for XML documents reorganization. This paper aims at proposing XSL to CILEA

supercomputers users in a "familiar style" using an unusual didactic and apparently very demanding example: a piloted translator from Italian to Esperanto. Obviously, translator's capabilities are very reduced but not completely trivial and its implementation allows highlighting relevant XSL features, unforeseen in a "sui generis" programming language. The purpose of this paper becomes so twofold: demonstrate the possibility to profitably using XSL not only in the WWW editing and explain the value of an "ante litteram" computer oriented language, hundred year-old but now again "green" thanks to new scenarios (Internet, European countries unification) unthinkable at Zamenhof, Esperanto inventor, age.

XSL estas lingvaĵo bazita sur XML kaj pensita por restrukturi dokumentojn de tipo XML. Ĉi artikolo celas ĝin prezenti laŭ familiara stilo por la uzantoj de la superkomputiloj de CILEA, utiligante nekutiman didaktikan ekzemplon kaj laŭaspekte inter la plej engaĝaj: direktita traduksistemo de la itala al esperanto. Kompreneble la traduksistemo havas kapablojn tre modestajn, sed ne tute banalajn, kaj ĝia realigado donas elementojn por evidentigi karakterizojn de XSL tute gravajn kiel programa lingvaĵo siatipa. La celo estas do duflanka: dementi la antaŭjuĵon de tiuj kiuj konsideras XSL-on lingvaĵo kun apliko limigita je la eldonado sur la TTT, kaj konigi la bonajn kvalitojn de lingvaĵo informadika jam antaŭtempa, nun jam centjara, sed reverdigita per la naskiĝo de novaj aplikaj medioj (Interreto, Eŭropa integriĝo) neimageblaj je la tempo de Zamenhof, ĝia kreinto.

La parola "reverdigita" fornisce un esempio delle caratteristiche *agglutinanti* dell'esperanto (che anche l'italiano possiede, seppure in minore misura). La parola viene costruita tramite l'uso di varie radici: re-verd-ig-it-a. Questo consente di ricostruirne il senso nel caso sia sconosciuta o sia un neologismo. Per inciso alludo al verde in quanto colore dell'esperanto (oltre che della religione islamica).

Bibliografia

[1] Il World Wide Web Consortium (W3C) con la pagina dedicata all' eXtensible Stylesheet Language (http://www.w3.org/Style/XSL/)

[2] Convertitore XSL: il programma SAXON (in Java, gratuitamente scaricabile) di Michael H. Kay (http://saxon.sourceforge.net/)

[3] Sito con corsi dedicati a XML ed XSLT etc. (http://www.devguru.com/home.asp)

[4] La pag. dell'"Universàla Esperànto Asocìo" (http://www.uea.org/index_5.html) non ha pagine in italiano

[5] La Federazione Esperantista Italiana (FEI http://www.esperanto.it/indice2.html)

[6] Internacia Katolika Unuiĝo Esperantista (http://www.ikue.org/): gli angeli)

[7] Il vocabolario di riferimento dell'esperanto (PIV: Plena Ilustrita Vortaro) di 1265 pag. pubblicato dalla SAT (Sennacieca Asocio Tutmonda)… a volte detta dei **sat**anici) (http://satesperanto.free.fr/eldonajhoj/novapiv.html)

[8] Note di grammatica eo in italiano: (http://www.cilea.it/~bottoni/doc/eo-note/)

[9] La rivista belga, in esperanto, Monato (http://www.esperanto.be/fel/mon/)

[10] Programma multilingue per Windows (http://www.cursodeesperanto.com.br/en/)

[11] L'UNL (Universal Networking Language http://www.undl.org/)

[12] L'istituto di linguistica computazionale del CNR di Pisa (http://www.ilc.pi.cnr.it/)

[13] Corso di linguistica computazionale dell'Università di Pavia per il trattamento automatico del linguaggio naturale a vari livelli (http://www.unipv.it/wwwling/lingcomp.html).