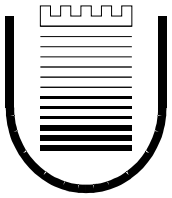
Università degli Studi di Roma "Tor Vergata"



Facoltà di Ingegneria

Laurea in Ingegneria Informatica
Relazione finale

REALIZZAZIONE DI UN PARSER PER LOGICHE DESCRITTIVE ALL'INTERNO DI UN ONTOLOGY EDITOR

Relatore	Candidato
Prof.ssa Maria Teresa Pazienza	Corrado Campisano

Anno accademico 2008/2009

Due cose riempiono l'animo di ammirazione
e venerazione sempre nuova e crescente,
quanto più spesso e più a lungo la riflessione si occupa di esse:
il cielo stellato sopra di me, e la legge morale in me.

Queste due cose io non ho bisogno di cercarle
e semplicemente supporle come se fossero avvolte nell'oscurità,
o fossero nel trascendente fuori del mio orizzonte;
io le vedo davanti a me e le connetto immediatamente
con la coscienza della mia esistenza.

(Immanuel Kant)

Alla mia famiglia
e a chi ha sempre creduto in me.

Indice

Elenco delle figure

Elenco delle tabelle

1 – Introduzione

Un **ontology editor** è un'applicazione progettata per assistere nella creazione e nella gestione di ontologie; una “ontologia”, di cui daremo definizione più esaustiva nel seguito, consiste nella formale specificazione di una concettualizzazione di un dominio di interesse [103], [107].

Una delle funzionalità più significative per un ontology editor consiste nella possibilità di inserire nuove asserzioni a partire da quelle esistenti, in particolare assiomi di equivalenza, sussunzione e disgiunzione tra classi.

Tali assiomi sono generalmente forniti dall'utente sotto forma di una stringa in un linguaggio formale, che abbia comunque connotazione di “human readability”, quale la sintassi adottata per la formulazione di espressioni nelle logiche descrittive (Description Logics, di seguito “DL”), ovvero quello formalizzato recentemente come “Manchester Syntax” (di seguito “MS”) per la serializzazione del linguaggio per ontologie OWL.

L'obiettivo di questo lavoro è proprio l'**implementazione** di un **parser** capace di interpretare tali stringhe, limitatamente alla sintassi delle DL, e la sua **integrazione** nell'ontology editor “Semantic Turkey”.

L'integrazione non è limitata all'invocazione del parser all'interno dell'editor, ma prevede anche l'elaborazione dell'output prodotto dal parser verso l'ontologia in lavorazione, in cui viene inserito l'assioma fornito dall'utente.

2 – Stato dell’arte

In questo capitolo si descrive lo stato attuale delle tematiche del semantic web, degli standard relativi (RDF, RDFS, OWL) o riconducibili (DL, MS) e degli strumenti per l'editing di ontologie.

2.1 – Ontologie

Il termine **ontologia** *“ha una storia complessa, all'interno e all'esterno dell'informatica”* [507]; originariamente introdotto in ambito filosofico, deriva dai termini greci *òntos* e *lògos,* letteralmente: “discorso sull'essere”, ed indica una branca fondamentale della metafisica, che si occupa dello studio dell'essere o dell'esistenza e delle sue categorie.

Il termine è poi stato adottato in ambito informatico, nel campo dell'intelligenza artificiale e della ingegneria della conoscenza ed in particolare nello sviluppo del Semantic Web, per indicare una rappresentazione formale, “machine-understandable”, condivisa ed esplicita, della concettualizzazione di un dominio di interesse [103], [107].

Tale adozione, un po' forzata quando con ontologia si intende l'artefatto informatico contenente la descrizione del dominio di interesse, non è comunque fuori luogo. Infatti, per la macchina, che dovrà processare le informazioni in quello contenute, quanto indicato nell'ontologia è proprio “il mondo che esiste”.

2.2 – Semantic Web

La “nascita” del semantic web, o meglio la sua prima “uscita pubblica”, può essere ricondotta all'omonimo articolo dello stesso “padre del web” Tim Berners-Lee [102], in cui si auspicava un'evoluzione del web, capace di produrre documenti “machine-understandable” e non solo “machine-representable”.

Il suo “concepimento” risale invece all'articolo “Semantic Web road map” [101], in cui tale auspicio era già compiutamente espresso:

> *The Web was designed as an information space, with the goal that it should be useful not only for human-human communication, but also that machines would be able to participate and help.*
> *One of the major obstacles to this has been the fact that most information on the Web is designed for human consumption, and even if it was derived from a database with well defined meanings (in at least some terms) for its columns, that the structure of the data is not evident to a robot browsing the web.*
> *Leaving aside the artificial intelligence problem of training machines to behave like people, the Semantic Web approach instead develops languages for expressing information in a machine processable form.*

E' il caso di sottolineare come si auspichi una evoluzione e non una sostituzione: l'obiettivo è di arrivare a produrre documenti che possano al tempo stesso essere letti ed apprezzati dagli esseri umani, ma anche acceduti ed interpretati da agenti automatici.

Gli agenti in questione sono quelli introdotti, nel contesto dell'intelligenza artificiale, come “agenti razionali” [100], in grado di “*rappresentare la conoscenza ed applicarvi un ragionamento*”.

Una discussione approfondita sul Semantic Web esula evidentemente dagli scopi del presente lavoro, che si limiterà a trattarne gli aspetti salienti per l'obiettivo dello stesso.

2.2.1 – Il Web attuale: al più la semantica è implicita

Il web, nella sua attuale formulazione, rende possibile l'accesso ad una enorme (secondo alcuni eccessiva [108]) quantità di dati.

Questi dati possono essere processati, indicizzati e recuperati dalle macchine, come fanno ad esempio i motori di ricerca, ma non possono essere "compresi" dalle stesse, che non possono quindi capire quali dati contengano le informazioni di interesse.

Anche nel caso in cui tali dati siano "strutturati" (ad esempio pagine web realizzate con XML-XSLT o nel caso dei Web Services), si rimane sempre a livello sintattico: la semantica deve essere "hard coded" nelle applicazioni che ne fanno uso.

Ciò è particolarmente evidente nel caso dei Web Services:

- la struttura basata su XML-Schema non ha una semantica, se non in forma "implicita" (ad esempio nei nomi assegnati agli elementi): è compito di chi implementa il client stabilire, caso per caso, come interpretare i dati ricevuti e come valorizzare quelli inviati;
- le potenzialità introdotte dal meccanismo publish/discover/interact (UDDI+WSDL) non possono essere sfruttate in automatico da una applicazione client, in quanto:
    - sebbene il client possa recuperare (tramite UDDI) un elenco di Web Services potenzialmente utili ai suoi scopi, non ha modo di "decidere" se lo siano effettivamente o quale sia il più appropriato;
    - sebbene il client possa recuperare (tramite WSDL) quale sia la sintassi (in termini di struttura XML) con cui costruire le richieste e processare le risposte, non ha la possibilità di "comprendere" tale struttura.

2.2.2 – La necessità di una semantica esplicita

Risulta quindi necessario, per l'evoluzione auspicata, **esplicitare la semantica** dei dati che vengono processati (motori di ricerca) o scambiati (Web Services).

Per arrivare a questo risultato, però, occorre un meccanismo che consenta alle macchine di costruire una propria "rappresentazione" o "concettualizzazione" della realtà, per mezzo della quale possano assegnare un significato ai dati con cui operano, che altrimenti rimarrebbero solo dei "significanti".

In breve, le macchine devono avere una "**base di conoscenza**" con cui completare il triangolo semiotico [105], di cui, allo stato attuale, possono accedere solo al significante:

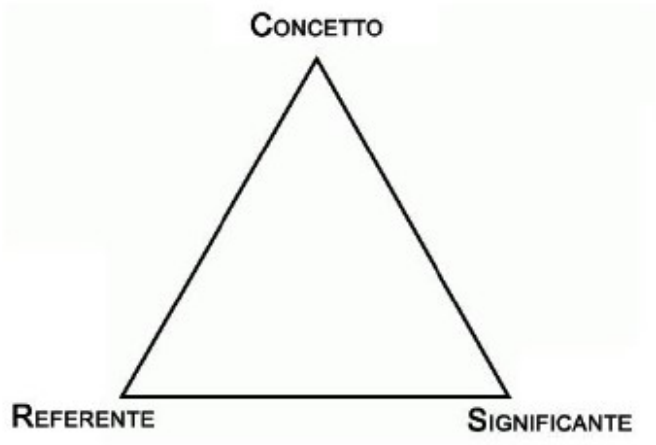


Figura 1: Il triangolo semiotico

Per realizzare tale base di conoscenza, sono stati definiti una serie di modelli e di standard, come illustrato nella sezione seguente.

2.3 – Linguaggi per la descrizione di ontologie (RDF/RDFS/OWL)

Lo stato attuale delle specifiche per i linguaggi per la descrizione di ontologie è il risultato del lavoro del ***Web Ontology Working Group***, appartenente al *W3C (World Wide Web Consortium)*.

L'obiettivo che il WOWG si è posto è quello di poter stabilire delle relazioni tra termini che rappresentino univocamente i concetti del dominio di interesse, con le quali realizzare delle "reti semantiche", navigabili (analogamente a quanto avviene sul web attuale"con gli *hyperlinks*) sia dagli utenti umani che da processi automatici.

I principi che hanno guidato questo gruppo di lavoro sono i seguenti:

- riferimenti non ambigui: i termini devono poter essere interpretati univocamente;
- decentralizzazione: la mancanza di restrizioni sui contenuti, imposta da una autorità centralizzata, è stata uno dei principali fattori di successo per il web e deve essere mantenuta, nonostante possa portare a problemi di consistenza;
- design minimalista: è necessario progettare un modello comune generico, utilizzabile nei contesti e per gli scopi più diversi;
- interoperabilità ed estensibilità: per consentire alle diverse comunità di lavorare indipendentemente, aggiungendo nuova informazione senza dover modificare quella preesistente;
- aggregazione "su fiducia": stante la decentralizzazione, è possibile avere sia contraddizioni (non essendoci restrizioni sulle affermazioni) che interpretazioni diverse (operando in contesti semantici differenti), per cui ogni applicazione, o agente, deve poter gestire delle "relazioni di fiducia" con altri agenti, assumendo come valide le loro asserzioni e rigettando quelle di altri.

2.3.1 – Architettura del Semantic Web

Coerentemente coi principi sopra esposti, il Semantic Web è composto da una pila di modelli, di tecnologie e di standard, come illustrato nella immagine seguente (si riporta la formulazione originale [104]):

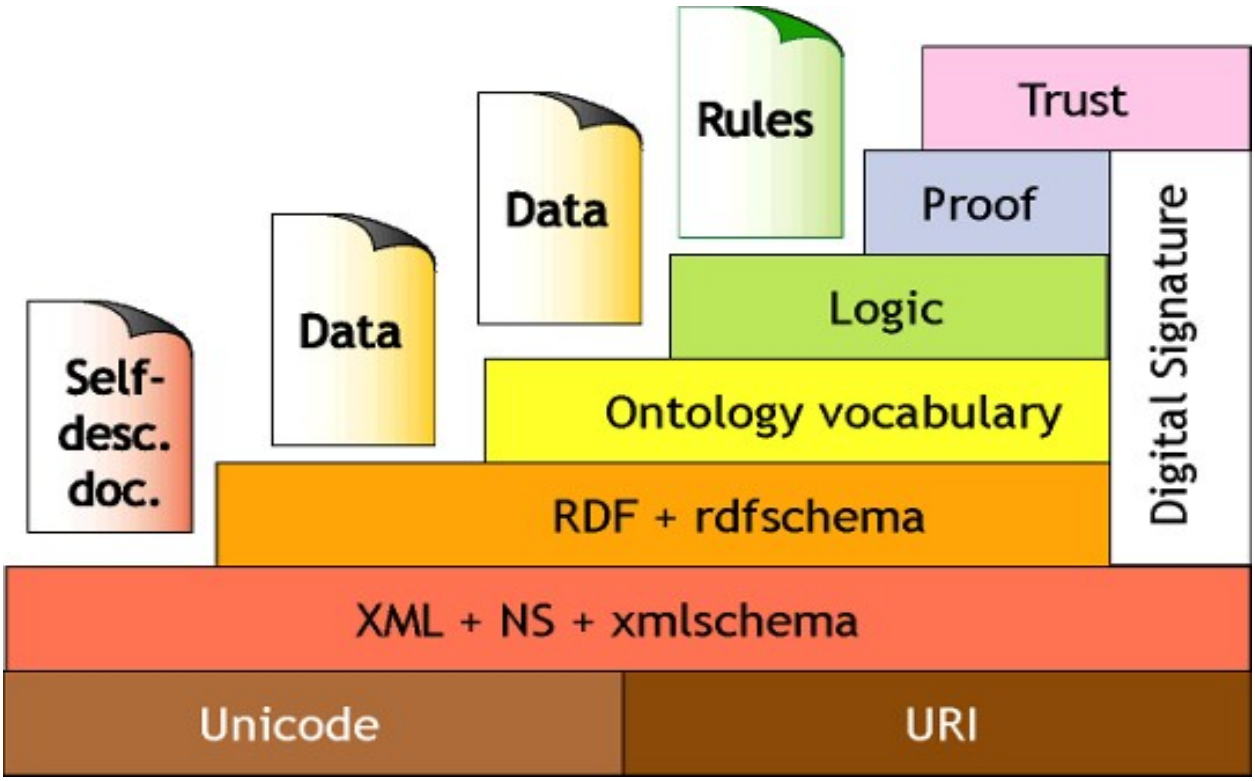


Figura 2: lo stack del Semantic Web

In particolare si evidenziano:

- lo standard **URI** (Unified Resource Locator) [300] utilizzato per identificare univocamente le "risorse" che compongono l'ontologia;
- gli standard **XML** (eXtensible Markup Language) [301], **XML-NS** (Name Spaces in XML) [302] e XML-Schema/**XSD** (XML Schema Definition) [303] utilizzati soprattutto per definire una sintassi e consentire una serializzazione "human-readable" e al contempo "machine-accessible";
- gli standard **RDF** (Resource Description Framework) [500] e **RDFS** (RDF-Schema o

RDF-Vocabulary) [502] che costituiscono il linguaggio per descrivere le risorse e i loro tipi;

- lo standard **OWL** (Ontology Web Language) [505], che costituisce il linguaggio per definire una ontologia, descrivendo le risorse e le loro relazioni, oltre quanto già possibile con RDF(S).

Nelle prossime sezioni si descrivono questi ultimi due, in quanto fondamentali per la realizzazione del presente lavoro.

2.3.2 – RDF e RDFS

Costituiscono il modello di base per poter fare "asserzioni" (*statements*) sugli elementi del dominio di interesse. Di seguito se ne illustrano le caratteristiche principali, mentre si rinvia ai riferimenti, sia per i dettagli tecnici ([500] e seguenti) che per le motivazioni storiche [106].

2.3.2.1 – RDF (Resource Description Framework)

La struttura sottostante di ogni espressione in RDF può essere vista come un grafo orientato, composto di due nodi (soggetto e oggetto) e di un arco orientato (proprietà) che li collega, nel complesso detto "**tripla**", come illustrato di seguito:

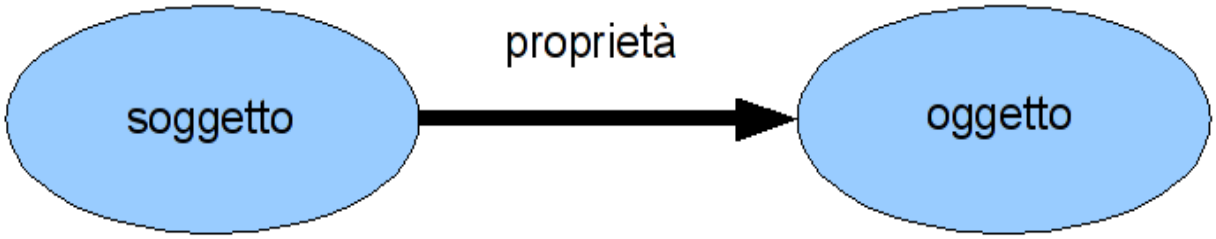


Figura 3: modello di una "tripla" RDF

Un **grafo RDF** è un insieme di triple come quella illustrata sopra, in cui:

- ogni entità del web può essere rappresentata da un nodo del grafo;
- la direzione dell'arco (che rappresenta una proprietà o "predicato") è volta ad indicare quale nodo sia il soggetto e quale l'oggetto dello "statement";
- sia i nodi che gli archi sono in generale delle "risorse" e sono identificate tramite un URI, eccetto:
  - "blank nodes": nodi "ad uso interno" e dotati solo di un identificatore "interno";

  - “literals”: valori letterali (“plain” o “typed”) che possono essere solo “nodi-oggetto”;
- il significato complessivo del grafo è la congiunzione (AND logico) di tutte le triple che lo compongono;
- relazioni di arità superiore a due possono essere realizzate “reificando” una proprietà in un nodo RDF.

E' il caso di sottolineare come lo standard RDF sia un modello astratto (rappresentato dal grafo), per il quale sono stati definiti anche una serie di “sintassi per la serializzazione”, tra cui RDF/XML, Turtle, N-Triples, N3.

2.3.2.2 – RDFS (RDF-Schema o RDF-Vocabulary)

Con RDF, come illustrato nel paragrafo precedente, è possibile realizzare dei semplici modelli di dati che hanno la forma propria delle reti semantiche (vedere [100]), ma, rispetto ad esse, mancano di due capacità fondamentali:

- non è possibile, con il solo RDF, definire diversi livelli di astrazione, specificando le classi (tipi) delle risorse e organizzando queste in una relazione tassonomica;
- di conseguenza non è possibile imporre delle restrizioni sulle proprietà e sulla loro applicazione alle diverse risorse; in particolare non è possibile imporre che una proprietà debba avere un determinato dominio ("domain") o un determinato codominio ("range").

Al fine di estendere RDF per ottenere un linguaggio di rappresentazione della conoscenza dalle caratteristiche simili a quelle delle reti semantiche, il W3C ha definito [502] una serie di primitive specifiche per la definizione di un vocabolario di meta-dati, tra le quali:

- *rdfs:Resource* : classe a cui appartengono tutte le entità descritte in RDF ("risorse");
- *rdfs:Class* : consente di definire una categoria ("classe") di risorse;
- *rdf:type* : consente di asserire l'appartenenza di una risorsa ad una determinata classe;
- *rdf:Property* : classe a cui appartengono tutte le risorse che sono proprietà in RDF;
- *rdfs:subClassOf* : permette di specificare una gerarchia semantica tra le classi;
- *rdfs:subPropertyOf* : analoga alla precedente, consente di creare una tassonomia delle proprietà;
- *rdfs:domain* : consente di imporre ad una proprietà che il suo dominio di applicazione sia una determinata classe, ovvero che tutti i "soggetti" a cui la proprietà è applicabile debbano appartenere a quella classe;
- *rdfs:range* : analoga alla precedente, ma in riferimento al codominio, ovvero consente di imporre che tutti gli "oggetti" di una tripla in cui è coinvolta una proprietà debbano appartenere ad una determinata classe.

2.3.3 – OWL (Ontology Web Language)

Analogamente a quanto visto per RDFS rispetto a RDF, a sua volta OWL consente di estendere le capacità di rappresentazione della conoscenza fornite da RDFS, al fine di raggiungere l'obiettivo del Semantic Web [102]:

For the semantic web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning.

A livello di RDF/RDFS, si evidenziano limitazioni e problematiche:

- con RDFS è possibile applicare solo inferenze di tipo tassonomico, in quanto una base di conoscenza costruita solo con esso non consente di fare asserzioni di altro tipo e quindi di derivare, con il ragionamento, nuove asserzioni che non siano di tipo tassonomico
- il potere espressivo di RDF corrisponde al sottoinsieme esistenziale-congiuntivo (EC) della logica del primo ordine: non è possibile esprimere la negazione, né la congiunzione;
- pur essendo un sottoinsieme della logica del primo ordine, consente in linea di principio di fare asserzioni in merito ad altre asserzioni, capacità propria di logiche di ordine superiore;
- una problematica simile è relativa al fatto che, in RDF, è consentito trattare ad un tempo una classe come tale (collezione di individui) e come istanza di un'altra classe (individuo), ancora una capacità propria di logiche di ordine superiore.

I primi tentativi di superare le limitazioni e di evitare le problematiche sopra riportate sono stati i seguenti:

- DAML (DARPA Agent Markup Language), sviluppato in ambito USA (DARPA) a partire da RDF;
- OIL (Ontlogy Inference Layer), sviluppato in ambito UE (IST) a partire da DL e RDFS.

Con la creazione di un “joint commitee” USA-UE, i due tentativi precedenti sono stati unificati nel linguaggio DAML+OIL, che consentiva di estendere il paradigma RDF(S), fornendo un vocabolario più ricco per descrivere le relazioni tra classi, la possibilità di asserire circa la cardinalità di una proprietà, una collezione più ricca di tipi e di caratteristiche delle proprietà.

A partire da DAML+OIL, il Web Ontology Working Group è arrivato a creare OWL, rilasciato pubblicamente come *W3C Recommendation* nel 2004 [503].

2.3.3.1 – Sottolinguaggi OWL

OWL fornisce tre sottolinguaggi di espressività crescente che sono stati progettati per essere utilizzati da determinate comunità di sviluppatori e utenti.

- ***OWL Lite*** aiuta quegli utenti che hanno soprattutto bisogno di una gerarchia di classificazione (tassonomia) e semplici restrizioni. Per esempio, benché consenta limitazioni sulla cardinalità di una proprietà, permette soltanto valori di cardinalità di 0 o 1. Lo sviluppo di uno strumento di supporto per OWL Lite è più semplice di quello per i suoi parenti più espressivi e OWL Lite fornisce un percorso di migrazione più rapido per i thesaurus e le altre tassonomie.
- ***OWL DL*** supporta quegli utenti che vogliono il massimo dell'espressività senza perdere la completezza computazionale (tutte le conclusioni hanno la garanzia di essere calcolabili) e la decidibilità (tutte le computazioni finiscono in un tempo definito) dei sistemi di ragionamento. OWL DL comprende tutti i costrutti del linguaggio OWL con delle restrizioni come quelle sulla separazione del tipo (una classe non può essere un individuo o una proprietà, così come una proprietà non può essere un individuo o una classe). OWL DL si chiama così a causa della sua corrispondenza con le *logiche descrittive* [400], un campo di ricerca che ha studiato un particolare frammento decidibile della logica del primo ordine. OWL DL è stato progettato per supportare la parte relativa alle logiche descrittive ed ha auspicabili proprietà computazionali per i sistemi di ragionamento.
- ***OWL Full*** è destinato agli utenti che vogliono la massima espressività e libertà sintattica di RDF senza le garanzie computazionali. Per esempio, in OWL Full una classe può essere trattata contemporaneamente come una collezione di individui e come un individuo a pieno titolo. E' improbabile che qualsiasi software di ragionamento possa sostenere un ragionamento completo per ciascuna caratteristica di OWL Full.

Ciascuno di questi sottolinguaggi è un'estensione del suo modello più semplice, sia in ciò che può essere legalmente espresso sia in ciò che può essere validamente concluso.

2.3.3.2 – Classi (class descriptions)

Come descritto in [505] e in [506], le “class descriptions” sono definite come gli elementi di base per produrre “class axioms”, laddove in generale le classi (come assiomi o come descrizioni) sono tra gli elementi base per costruire una ontologia (insieme a proprietà ed individui).

In OWL esistono sei tipi di “class descriptions”, o meglio sei modi per affermare l'esistenza di una classe:

1. tramite un class identifier (URI);
2. mediante l'enumerazione esaustiva di tutti gli individui appartenenti alla classe;
3. mediante una “property restriction”
4. mediante l'intersezione di più classi;
5. mediante l'unione di più classi;
6. mediante il complemento di una classe.

Nel seguito, però, si adotterà una nomenclatura e una tassonomia differenti, la cui bontà risulterà evidente nel momento in cui si analizzerà la realizzazione del parser:

A – classByName (tipo 1);

B – classByEnum (tipo 2);

C – classByPropRestr (tipo 3), con i sottotipi:

C.1 – classByValueRestr ([505], Value Restriction):

C.1.A – owl:allValuesFrom; *

C.1.B – owl:someValuesFrom; *

C.1.C – owl:hasValue;

C.2 – classByCardRestr ([505], Cardinality Restriction):

C.2.A - owl:maxCardinality;

C.2.B – owl:minCardinality;

C.2.C – owl:cardinality;

D - classByBoole (tipi 4, 5 e 6), con i sottotipi:

D.4 – owl:intersectionOf *

D.5 – owl:unionOf *

D.6 – owl:complementOf *

Le tipologie marcate con asterisco permettono una composizione “ricorsiva”, nel senso che sono a loro volta composte da altre “class descriptions”.

2.3.3.3 – Assiomi (class axioms)

Come descritto in [505] e in [506], i “class axioms” utilizzano una “class description” (di uno qualsiasi dei tipi precedenti, eventualmente composita) come “oggetto” per generare asserzioni relative alla “class description” (esclusivamente del tipo 1) che è il “soggetto” dell'assioma.

In OWL esistono 3 tipi di “class axioms”:

1. **rdfs:subClassOf** : ereditato da RDFS come meccanismo per l'organizzazione in tassonomie;
2. **owl:equivalentClass** : consente di stabilire l'equivalenza tra classi; data la natura orientata agli insiemi di OWL, due classi si intendono equivalenti se hanno la stessa “estensione”, ovvero lo stesso insieme di individui;
3. **owl:disjointWith** : all'opposto del caso precedente, permette di affermare che due classi non hanno alcun individuo in comune.

2.4 – Description Logics

Le logiche descrittive (in inglese, "*description logics*") sono una famiglia di logiche utilizzate per rappresentare la conoscenza in un dominio di applicazione.

In primo luogo sono definiti i concetti rilevanti per quel dominio e, di seguito, utilizzando questi concetti si specificano le proprietà degli oggetti e degli individui appartenenti al dominio.

Gli studi riguardanti le logiche descrittive si focalizzano sulla ricerca di metodi che descrivano in modo sempre più specifico il dominio di interesse tale da essere utilizzato per costruire applicazioni intelligenti.

In questo contesto il termine intelligente è riferito all'abilità di un sistema di trovare delle conseguenze implicite rispetto a quelle fornite esplicitamente.

Sistemi così caratterizzati sono definiti sistemi basati su conoscenza (*knowledge based system*).

Una delle caratteristiche dei linguaggi utilizzati per la rappresentazione della conoscenza è quella di essere forniti di semantiche basate sulla logica, con una enfasi particolare al cosiddetto ragionatore ("reasoner").

Il reasoner permette di derivare la conoscenza implicita, ricavandola da quella esplicita presente in una ontologia.

Dal punto di vista della conoscenza umana una logica descrittiva permette di strutturare e capire il mondo che ci circonda attraverso la classificazione degli individui e dei concetti.

La classificazione dei concetti permette di determinare le relazioni tra i sotto-concetti e/o i super-concetti che a loro volta permettono di strutturare la tassonomia in modo gerarchico.

La tassonomia è utile per ricavare informazioni circa le connessioni logiche tra concetti differenti: la classificazione degli individui e/o degli oggetti determina se uno specifico individuo è sempre una istanza di un determinato concetto.

2.4.1 – Breve storia

I primi studi sulla rappresentazione della conoscenza iniziarono negli anni 70 e possono essere divisi in maniera sommaria in due categorie:

- formalismi basati sulla logica, che si sono sviluppati partendo dalla convinzione che il calcolo dei predicati poteva essere usato in modo non ambiguo per descrivere l'ambiente circostante;
- rappresentazioni non basate su una logica, ma sviluppate tenendo conto di nozioni più cognitive, come le strutture di reti e rappresentazioni basate su compiti specifici, derivate da esperimenti riguardanti la capacità della mente umana di ricordare concetti o di eseguire compiti, come la risoluzione di puzzle matematici.

Negli approcci basati sulla logica, il linguaggio utilizzato per la rappresentazione della conoscenza è una variante del calcolo dei predicati del primo ordine, dove il compito del ragionatore è quello di verificare la consequenzialità logica dei costrutti.

Al contrario, in un approccio non logico, spesso basato sull'utilizzo di interfacce grafiche, la conoscenza è rappresentata per mezzo di una struttura dati ad hoc ed il ragionatore ha il compito di eseguire delle procedure ad hoc utili a manipolare queste strutture. Tra queste rappresentazioni ricordiamo le reti semantiche [402] e i frame [401].

Le reti semantiche sono state sviluppate con l'apporto del lavoro di Quillian [402] con l'obiettivo di caratterizzare, per mezzo di strutture di reti cognitive, sia la conoscenza che il ragionatore in determinati sistemi. Sebbene esistano delle differenze significative tra le reti semantiche e i frame entrambi hanno delle basi comuni. Infatti sono considerate delle strutture di rete dove la struttura ha lo scopo di rappresentare un insieme di individui e le loro relazioni.

La ricerca nell'ambito delle logiche descrittive iniziò con lo studio dei sistemi terminologici per enfatizzare che il linguaggio utilizzato per la rappresentazione della conoscenza doveva caratterizzare la terminologia di base adottata nella modellazione del dominio di interesse.

Più tardi l’enfasi fu spostata sui costrutti ammessi dal linguaggio che descrivevano i concetti, da cui il nome di linguaggi di concetto. In anni più recenti, dopo che l’attenzione degli esperti è stata rivolta verso le proprietà dei sistemi logici, ha preso sempre più piede nella terminologia comune il concetto di logica descrittiva.

Più recentemente le logiche descrittive sono state utilizzate come formalismo di base per lo sviluppo degli standard promossi dalla "Semantic Web Initiative" del W3C, quali OWL.

2.4.2 – Famiglie di logiche descrittive

A seconda delle possibilità e delle restrizioni imposte nell'utilizzo dei diversi costrutti ammessi dal linguaggio, ogni famiglia logica viene caratterizzata da una particolare capacità espressiva, che può essere valutata utilizzando una notazione letterale standard:

1. ***AL*** : indica la logica degli attributi e introduce gli operatori di congiunzione ed i quantificatori universale ed esistenziale;
2. ***C*** : descrive la possibilità di usare l'operatore di negazione;
3. ***S*** : estende la ***ALC*** con l'ulteriore possibilità di definire la chiusura transitiva di un ruolo;
4. ***H*** : fornisce la possibilità di definire gerarchie tra ruoli;
5. ***O*** : consente l'operatore di enumerazione;
6. ***I*** : permette di riferirsi al ruolo inverso;
7. ***F***, ***N*** e ***Q*** : caratterizzano le possibilità di definire cardinalità rispettivamente funzionale, semplice e qualificata (in ordine di espressività crescente);
8. ***D*** : descrive la possibilità di riferirsi a domini concreti, ovvero introduce le istanze dei concetti, permettendo di asserire l'appartenenza di un'istanza (oggetto concreto) ad un concetto (ente astratto).

In quest'ottica, ad esempio, il sottolinguaggio OWL DL di può essere considerato equivalente ad una logica descrittiva di tipo ***SHOIN*(*D*)**.

2.5 – Strumenti per Ontology Editing

Di seguito si procede ad una breve rassegna dei principali ontology editor, evidenziandone le caratteristiche relative alle funzionalità analoghe a quella oggetto di questo lavoro.

Per un elenco più esaustivo dei vari ontology editor attualmente disponibili, si rimanda ai riferimenti ([200]).

2.5.1 – Top Braid Composer

Si tratta di un prodotto di natura commerciale, fornito dalla Top Quadrant [201], ma alla cui realizzazione hanno contribuito diversi sviluppatori provenienti dall'ambiente accademico e che fa ampio uso di software “open” (Eclipse, Jena, librerie della Apache Foundation) e di cui è disponibile anche una versione “free”.

Il prodotto, in merito alla funzionalità in esame, ovvero la possibilità di specificare dei “class axiom” su una determinata classe dell'ontologia (evidenziata con rettangolo rosso nell'immagine sottostante), fornisce sia uno “wizard” grafico (rettangolo blu), che la possibilità di specificare tali assiomi imputando una stringa in “Manchester Syntax” [508] (rettangolo verde).

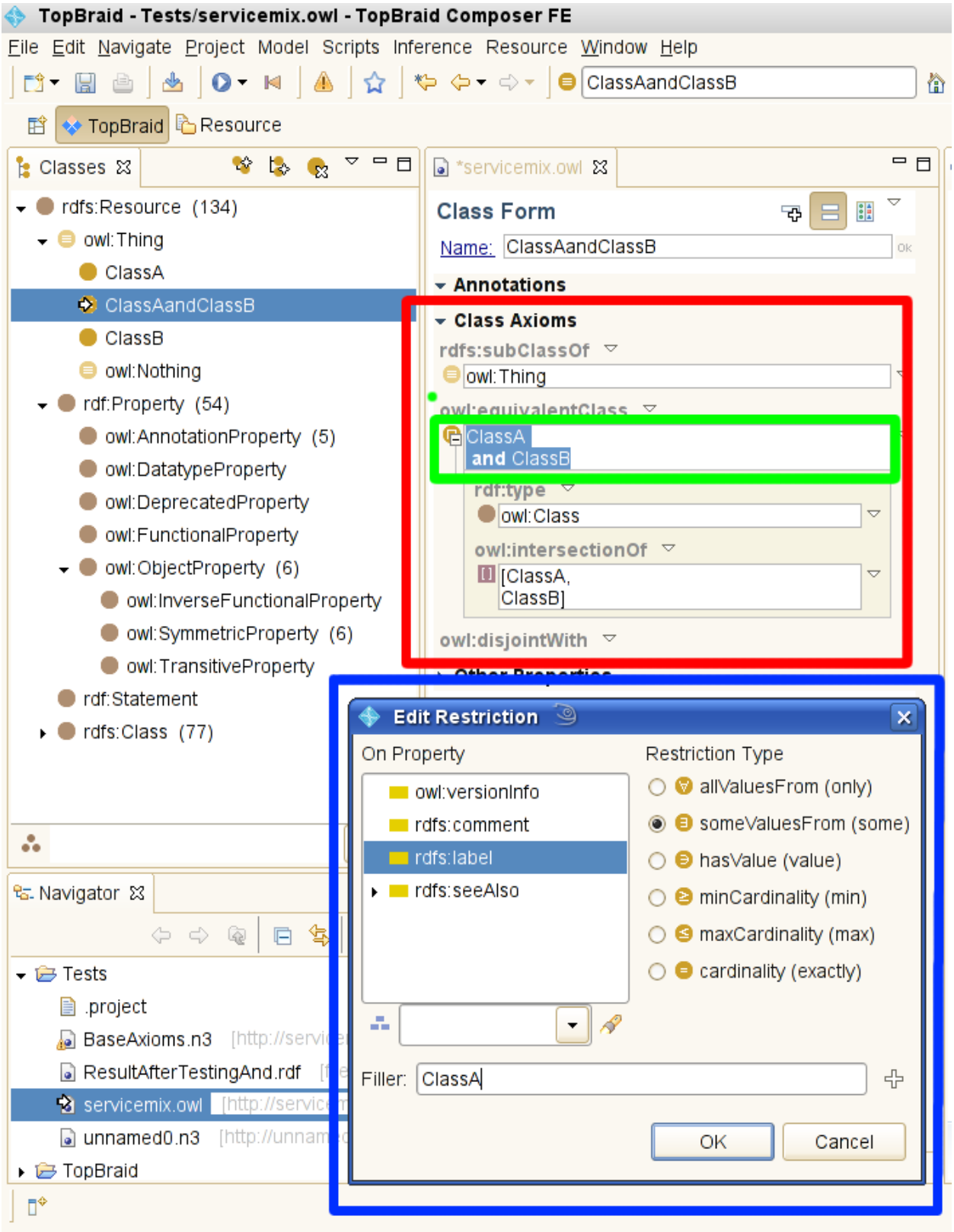


Figura 4: Top Braid Composer

2.5.2 – Protégé

E' stato il primo ontology editor e, per un lungo periodo di tempo, il più utilizzato; sviluppato in ambito accademico, in particolare dall'università di Stanford (Stanford Center for Biomedical Informatics Research), è disponibile con licenza “open source” [202].

Lo strumento, analogamente a Top Braid Composer, permette di specificare dei “class axiom” su una determinata classe dell'ontologia (evidenziata con rettangolo rosso nell'immagine sottostante), sia attraverso uno “wizard” grafico (rettangolo blu), che inserendo una stringa in “Manchester Syntax” [508] (rettangolo verde).

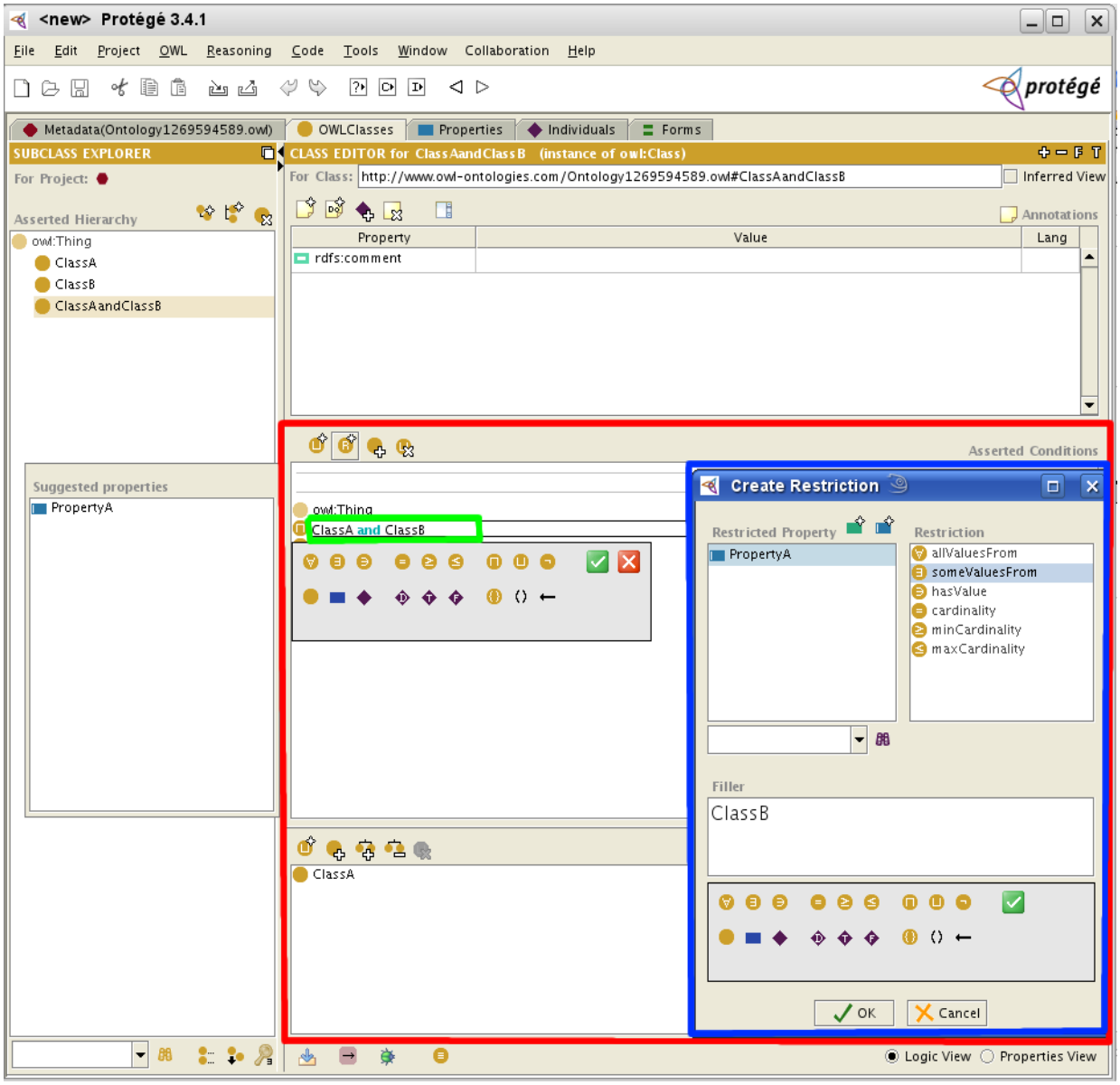


Figura 5: Protégé 3.4.1

2.5.3 – Semantic Turkey

Si tratta di uno strumento originariamente pensato per il “semantic bookmarking”, ma utilizzabile anche per lo sviluppo di ontologie, realizzato dall'università di Tor Vergata (gruppo ART) e disponibile con licenza “open source” [203].

Allo stato attuale, non è provvisto di uno strumento per l'immissione di “class axioms”, la realizzazione del quale, limitatamente alle stringhe in “description logics”, è proprio l'obiettivo di questo lavoro.

A differenza di Top Braid Composer e di Protégé, Semantic Turkey ha due peculiarità:

- non è un'applicazione standalone, ma una Mozilla Extension per il browser Firefox;
- è composto di un componente server e di un componente client.

Nel seguito si illustrerà in maggiore dettaglio l'architettura di Semantic Turkey; quanto qui anticipato risulta necessario per esigenze di trattazione, in quanto si farà riferimento a tali componenti prima di dettagliare l'architettura dello strumento.

Nella immagine sottostante si mostra il componente client che implementa tale funzionalità.

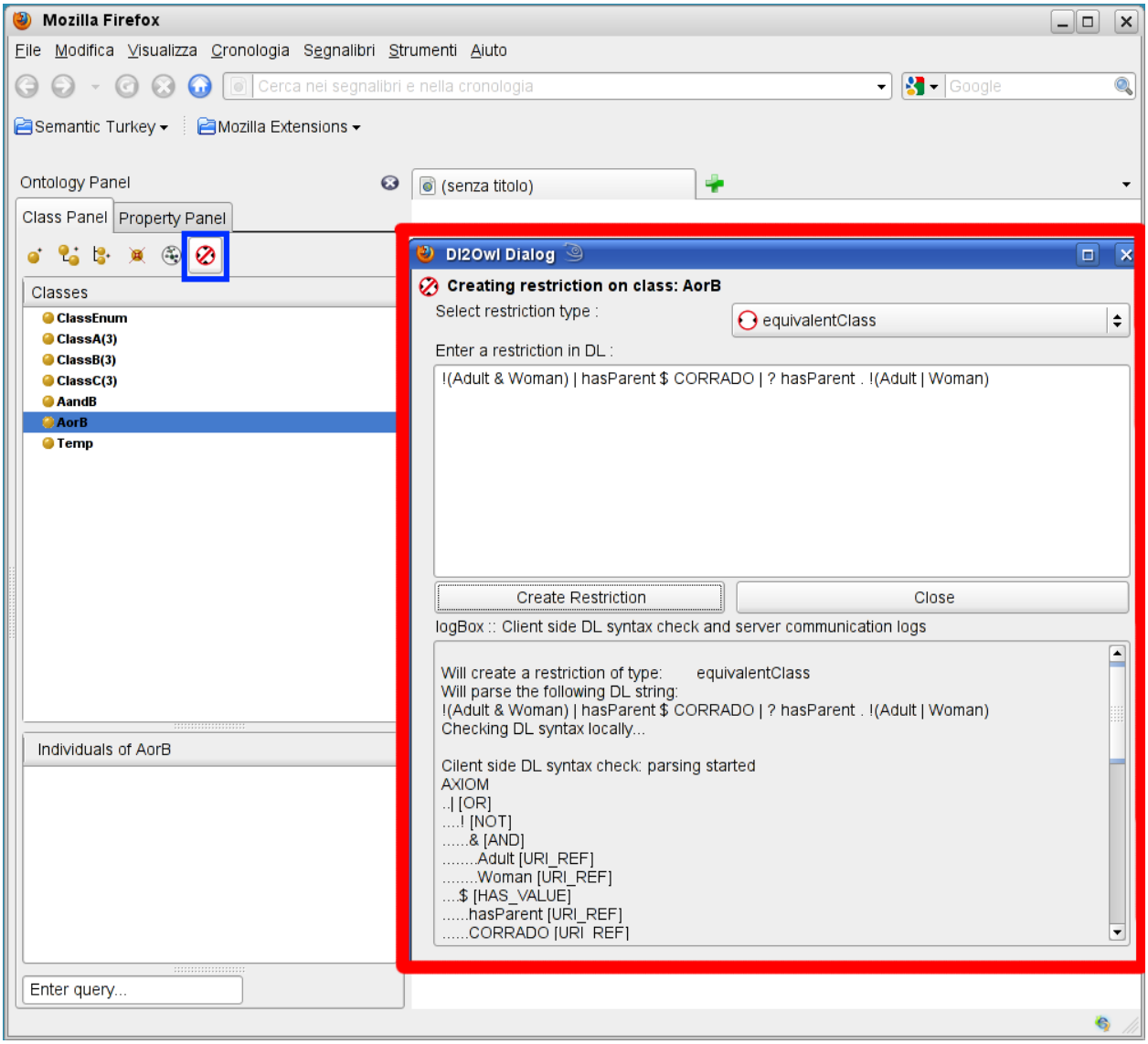


Figura 6: Semantic Turkey

2.6 – Parsing e parser generators

In questa sezione si fornisce una breve panoramica sul parsing e sui generatori di parser, rimandando ai riferimenti per eventuali approfondimenti.

2.6.1 – Parsing o “analisi sintattica”

Una operazione di “parsing” o meglio, di “analisi sintattica” [600], [601], è definita come segue:

> *In computer science and linguistics, parsing, or, more formally, syntactic analysis, is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar.*

Tale operazione è svolta in due fasi:

- dalla stringa in input, mediante un'analisi lessicale (“lexing”), viene prodotta una sequenza di “token” (blocchi di testo che sono stati riconosciuti come “elementi atomici”);
- dalla sequenza di token, mediante un'analisi sintattica (“parsing” vero e proprio), viene prodotta una struttura dati, implicita nella stringa fornita.

E' utile sottolineare, in riferimento all'illustrazione seguente, come le due fasi sono svolte da due componenti distinti, il “`Lexer`” (blocco verde) ed il “`Parser`” vero e proprio (blocco giallo).

Le due fasi sono concettualmente distinte, come riportato in precedenza, e spesso anche a livello di implementazione si hanno due componenti software distinti (ad esempio nel nostro caso, come vedremo).

Tuttavia, nel linguaggio comune, si chiama “parser” l'insieme dei due (blocco grigio).

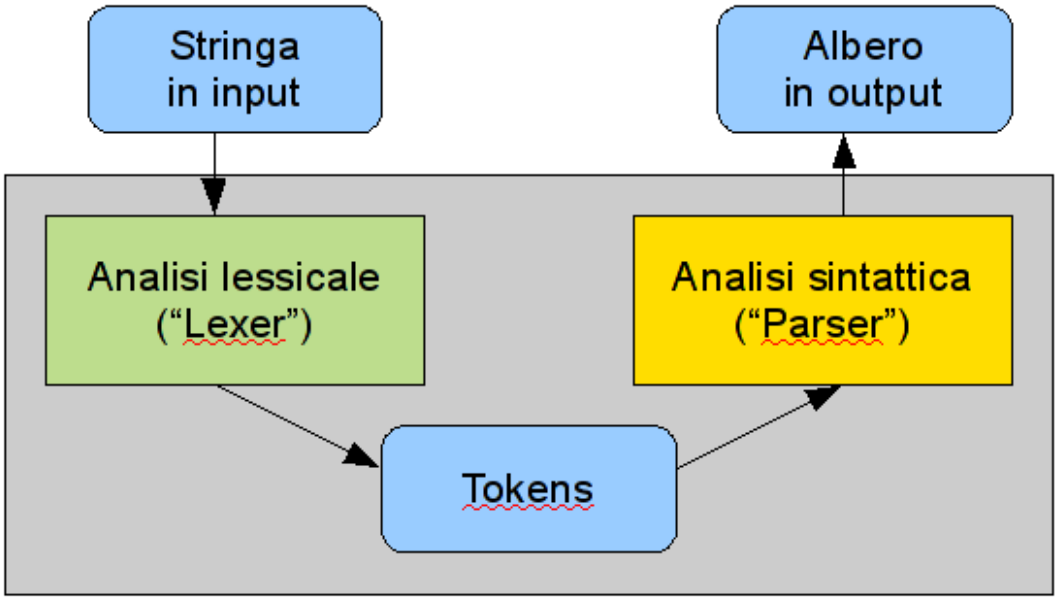


Figura 7: flusso di un'operazione di parsing

E' opportuno notare come l'operazione di parsing in sé si esaurisce nel momento in cui viene riconosciuta e prodotta in output la struttura dati.

Le eventuali operazioni successive, compiute su quanto prodotto in output, non fanno parte, strettamente parlando, del “parser”, ma sono piuttosto relative all'uso che si vuole fare dell'informazione (ora strutturata) che si è ottenuta mediante il parsing.

Questa argomentazione non è fine a se stessa: viene riportata in quanto ha determinato alcune scelte nella progettazione della soluzione, come illustrato nelle successive sezioni, in particolare in “3.3 – Progetto DL2OWL”.

2.6.2 – Grammatiche generative

Come descritto nella citazione al paragrafo precedente, affinché sia possibile effettuare il parsing di una stringa, in particolare per quanto riguarda l'analisi sintattica, quella deve possedere una certa struttura sintattica, ovvero deve essere conforme ad una "grammatica", intesa come insieme di regole in osservazione delle quali la stringa è stata generata.

Per ovvie esigenze di sintesi si evita una discussione approfondita della tematica delle grammatiche generative e/o della "gerarchia di Chomsky" ([602], [603]) e ci si limita a riportare la suddetta gerarchia, per individuare in essa dove si colloca la sintassi delle DL.

La gerarchia di Chomsky è composta dai seguenti livelli:

- **Grammatiche di tipo-0** (grammatiche illimitate) include tutte le grammatiche formali. Queste grammatiche generano esattamente tutti i linguaggi che possono essere riconosciuti da una Macchina di Turing.
- **Grammatiche di tipo-1** (grammatiche dipendenti dal contesto) generano linguaggi dipendenti dal contesto. I linguaggi descritti da queste grammatiche sono esattamente tutti i linguaggi che possono essere riconosciuti da una macchina di Turing non deterministica nella quale il nastro è limitato da un numero costante di volte la lunghezza dell'input.
- **Grammatiche di tipo-2** (grammatiche libere dal contesto) generano linguaggi liberi dal contesto. Questi linguaggi sono esattamente tutti i linguaggi che possono essere riconosciuti da un automa a pila non deterministico. I linguaggi "context free" sono teoricamente le basi per la sintassi di molti linguaggi di programmazione.
- **Grammatiche di tipo-3** (grammatiche regolari) generano linguaggi regolari. Questi linguaggi sono esattamente tutti i linguaggi riconosciuti da un automa a stati finiti. Questa famiglia di linguaggi formali può essere ottenuta con espressioni regolari. I linguaggi regolari sono comunemente usati per definire modelli di ricerca (search patterns) e la struttura lessicale dei linguaggi di programmazione.

Nella figura seguente si fornisce una rappresentazione grafica di tale gerarchia:

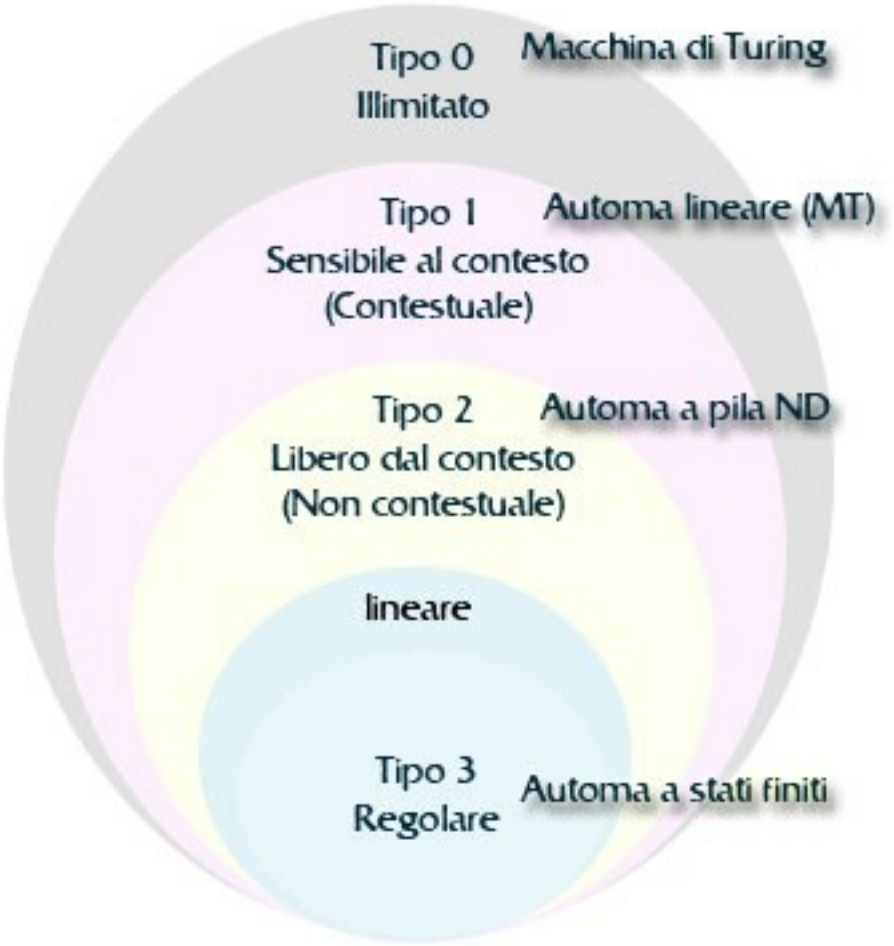


Figura 8: gerarchia di Chomsky ed automi associati

Nel nostro caso, in particolare, la sintassi delle stringhe in Description Logics è tale da:

- rientrare nella categoria delle “grammatiche libere dal contesto” [605];
- per quanto al punto precedente, l'automa che ne può effettuare il riconoscimento è un “automa a pila” (push down automation);
- la struttura dati che tale automa produce in uscita, per rappresentare la struttura sintattica, è un albero (in generale n-ario, vista la definizione di “and”, “or” e “enum”).

2.6.3 – Generatori di parser

Per ovvie esigenze di sintesi, ci si limiterà ad osservare come la maggior parte dei generatori di parser [604] è effettivamente basato sulla definizione di una grammatica che, ad un tempo:

- definisce le regole di composizione per le stringhe da considerarsi "valide", ai fini del parser;
- definisce le regole alla base del processo di analisi sintattica che il parser andrà ad effettuare.

Nel nostro caso specifico, avendo a che fare con grammatiche "context free" e con automi "push down", le tipologie di parser idonei sono quelli che possono effettuare una leftmost-derivation (top-down: "parser LL") o una rightmost-derivation (bottom-up: "parser LR") [605].

La scelta tra una delle due tipologie può essere motivata da considerazioni relative alle prestazioni, che in questa sede non considereremo: ai fini del presente lavoro si è scelto di optare per un parser LL in funzione di altre valutazioni, in primo luogo l'adeguatezza rispetto al problema (look-ahead arbitrario), la disponibilità (in termini di licenza d'uso) e la maturità dello stesso.

2.6.4 – ANTLR, generatore di parser LL(*)

Il generatore di parser che stato adottato è ANTLR [609], così descritto dai propri autori:

> *ANTLR v3 is the most powerful, easy-to-use parser generator built to date, and represents the culmination of more than 15 years of research by Terence Parr, professor of computer science and graduate program director at the University of San Francisco, CA, USA.*

Si tratta di un generatore di parser LL, capace di look-ahead arbitrario, indicato con LL(*), ovvero della capacità di analizzare un numero qualsiasi di token successivi a quello di volta in volta in esame.

Questo consente di non porre limiti alla complessità delle stringhe DL in input: qualora si fosse adottato un parser con look-ahead limitato a *k* token, indicato con LL(*k*), si sarebbero dovute porre delle limitazioni alle stringhe in input: il parser generato non sarebbe stato in grado di valutare più di *k* token successivi a quello di volta in volta in esame.

Nel complesso, la scelta è stata motivata dai seguenti fattori:

- adeguatezza rispetto al problema;
- disponibilità come software “open source” (BSD license);
- maturità del software;
- disponibilità di documentazione ([610], [611]);
- esistenza di una comunità di utenti e sviluppatori molto attiva ([612]);
- disponibilità di strumenti aggiuntivi, quali:
    - ANTLR-Works [614], per lo sviluppo ed il debugging della grammatica;
    - ANTLR V3 plugin for Maven [617].

3 – Approccio

Per implementare la funzionalità di immissione di “class axioms” in Semantic Turkey, si è scelto di procedere come segue:

- l'operazione di parsing è stata delegata ad un parser generato da grammatica:
  - il codice del parser viene generato con ANTLRv3, nel linguaggio Java per il componente server e nel linguaggio JavaScript per quello client;
  - il software risultante, organizzato come progetto Maven [616], viene rilasciato come libreria esterna a Semantic Turkey (di seguito indicata come “DL2OWL”), nel quale viene successivamente integrato;
- l'operazione di parsing è svolta in due tempi:
  - nel componente client, ai soli fini della verifica della sintassi della stringa DL fornita dall'utente (in caso di errore l'utente viene avvertito del problema e la stringa non viene inviata al componente server);
  - nel componente server, in cui, dopo la verifica sintattica, viene prodotta una struttura dati ad albero, che viene quindi processata per inserire l'assioma nell'ontologia.

Nelle sezioni seguenti vengono illustrati in dettaglio:

- i requisiti del parser, in termini di corrispondenza tra una stringa in DL e l'equivalente in OWL;
- l'implementazione di tali requisiti mediante una grammatica context-free;
- il progetto Maven-based “DL2OWL”.

L'integrazione del software in Semantic Turkey viene invece trattata nel capitolo successivo, dopo aver introdotto l'architettura di questo, al fine di rendere più comprensibile la trattazione.

3.1 – Requisiti del parser

Come descritto in “2.4.2 – Famiglie di logiche descrittive”, il sottolinguaggio OWL DL può essere considerato equivalente ad una logica descrittiva di tipo ***SHOIN(D)***.

Una trattazione esaustiva di tale equivalenza esula dagli scopi del presente lavoro, in cui ci si limita a considerare la “Mappatura DL → OWL” [400], riportata nella tabella sottostante come requisito per l'implementazione del parser.

Elemento OWL	**Simbolo UNICODE**	**Simbolo ASCII**	**Esempio in DL**	**Semantica**
allValuesFrom	∀ (0x2200)	*	∀ figlio . Medico	Tutte le risorse del codominio della proprietà “figlio” appartengono alla classe “Medico”
someValuesFrom	∃ (0x2203)	?	∃ figlio . Medico	Esiste almeno una risorsa del codominio della proprietà “figlio” che appartiene alla classe “Medico”
hasValue	∋ (0x220B)	$	figlio ∋ GINO	Il valore della proprietà “figlio” corrisponde alla risorsa “GINO”
cardinality	= (0x003D)	=	= 2 figli	La proprietà “figli” ha esattamente 2 valori
minCardinality	≥ (0x2264)	>	≥ 2 figli	La proprietà “figli” ha almeno 2 valori
maxCardinality	≤ (0x2265)	<	≤ 2 figli	La proprietà “figli” ha al più 2 valori
complementOf	¬ (0x2310)	!	¬ Medico	Tutto ciò che non appartiene alla classe “Medico”
intersectionOf	⊓ (0x2293)	&	Medico ⊓ Dentista	Tutto ciò che appartiene alle classi “Medico” e “Dentista”
unionOf	⊔ (0x2294)	|	Medico ⊔ Dentista	Tutto ciò che appartiene alle classi “Medico” o “Dentista”
enumeration	{...}	{...}	{GINO PINO}	La classe composta dagli individui “GINO” e “PINO”

Tabella 1: mappatura DL → OWL

NOTA 1: la grammatica dovrà essere in grado di supportare sia i simboli UNICODE che quelli ASCII, analogamente a quanto permettono Top Braid Composer e Protègè.

NOTA 2: nella tabella si utilizza la convenzione DL per cui:

- i concetti atomici DL (classi in OWL) sono indicati con la prima lettera maiuscola ed il resto minuscolo (es. “Medico”);
- i ruoli atomici DL (proprietà in OWL) sono indicati con tutte lettere minuscole (es. “figlio”);
- le istanze DL (individui in OWL) sono indicati con tutte lettere maiuscole (es. “GINO”)

Tale convenzione è tuttavia solo una “best practice” ai fini della leggibilità, nell'implementazione della grammatica non viene considerata, in quanto non ha corrispondenza in OWL.

NOTA 3: in fase di analisi si è stabilito che la stringa DL debba rappresentare, analogamente a quanto avviene sugli analoghi editor, solo la “class description-oggetto” e non uno statement completo di equivalenza/sussunzione/disgiunzione.

3.2 – Sviluppo della grammatica del parser

La grammatica da cui generare il parser è stata sviluppata principalmente utilizzando lo strumento ANTLR-Works [614], di grande utilità sia per la stesura che per il debugging delle grammatiche.

Tale strumento consente infatti:

- in fase di editing, di visualizzare una rappresentazione grafica di ogni regola della grammatica;
- in fase di debugging, di controllare sul "parsing tree" (albero delle alternative) il comportamento del parser generato a fronte di un input, il che è utile sia a fronte di input sintatticamente corretti che no;
- nel caso l'output sia una struttura ad albero AST ("Abstract Syntax Tree"), di visualizzare immediatamente la struttura ed il contenuto dello stesso.

Di seguito, a titolo di esempio, si riporta la rappresentazione dell'AST prodotto dal parser, a fronte del seguente input:

```
Human & ({eurasia Africa medio-oriente} | ? Has_Child_1 .
Philosophiae-Doctor) & (* Has_Child-2 . Physician &
Has_Child_New $ Bill_Clinton)
```

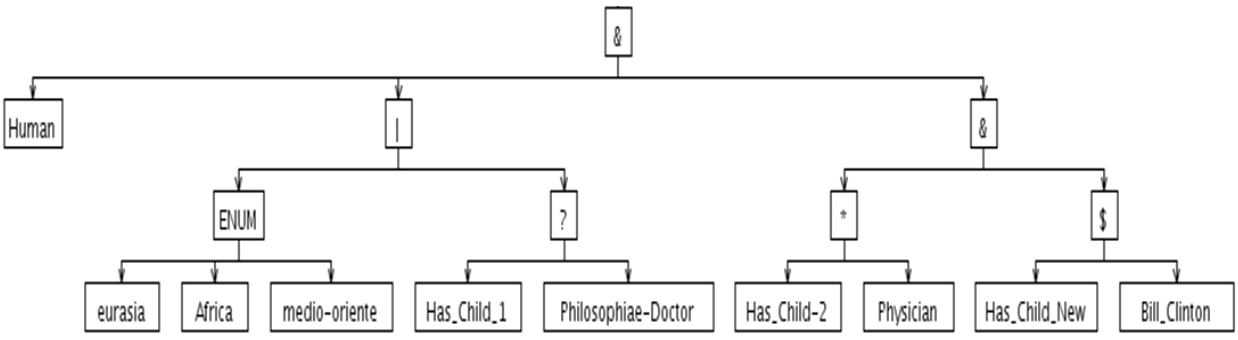


Figura 9: rappresentazione dell'AST prodotto dal parser

3.2.1 – Dettaglio sulla grammatica

Riportare ed illustrare in questa sede tutto il codice che compone la grammatica renderebbe la trattazione troppo pesante, per cui ci si limiterà a mostrarne gli aspetti salienti, considerando anche che tutto il codice è abbondantemente provvisto di commenti, in particolare in riferimento allo standard OWL.

3.2.1.1 – Grammatica "monolitica"

Stante la non eccessiva complessità della grammatica, si è scelto di non separare la grammatica nelle sue componenti "lexer" e "parser", ma di usare una grammatica monolitica.

Tale scelta, sebbene contraddica le "best practices" indicate in [611], ha consentito di lavorare più velocemente con lo strumento ANTRL-Works, che, nel caso di grammatiche "separate" ha evidenziato qualche pecca in fatto di stabilità, il che si traduceva spesso nella necessità di riavviare lo strumento stesso nel caso di modifiche su entrambe le componenti.

3.2.1.2 – Grammatiche Java e JavaScript

Al fine di generare codice nei due linguaggi, sono state create due grammatiche, identiche in tutto nella parte propriamente relativa alla grammatica e che si differenziano solo per i dettagli relativi ai diversi "target", che si riportano di seguito.

Nella grammatica Java, non è stato necessario specificare il linguaggio-target, essendo questo per default proprio Java.

Nella grammatica JavaScript è invece necessaria tale direttiva, come segue:

```
options {
     language=JavaScript;
     ...
}
```

All'opposto, mentre in JavaScript non avrebbe senso definire un "package", per la versione Java si è scelto, per esigenze di "pulizia" della struttura dei sorgenti, di specificare un package, come riportato di seguito:

```
@lexer::header{
package eu.servicemix.dl2owl.antlr.generated;
}
@parser::header{
package eu.servicemix.dl2owl.antlr.generated;
}
```

3.2.1.3 – Output AST "diretto"

La struttura dati che meglio si adatta a rappresentare l'output del parser, come evidente sia dalla natura della stringa DL in input, che dalla struttura OWL che ci si aspetta in output, è un albero.

Si è scelto quindi di sviluppare la grammatica in modo che non esegua nulla (*), a fronte del "match" di ogni regola, se non di popolare l'albero AST (Abstract Syntax Tree) che viene restituito come output.

Per ottenere tale risultato, è stato innanzitutto impostato tale tipo di output nelle opzioni:

```
options {
     output=AST;
     ASTLabelType=CommonTree;
}
```

E' stato poi sufficiente utilizzare la sintassi ANTLR per il popolamento dei nodi, in ogni regola che lo richiedesse, come si vede di seguito per la regola "intersection", che codifica un "and" logico:

```
intersection
     :    c1=classDescr AND c2=classDescr (AND c3=classDescr)*
          -> ^(AND $c1 $c2 $c3*);
```

(*) : a seguito di ogni alternativa di ogni regola, la sintassi di ANTLR permette di inserire del codice, nel linguaggio-target, ad esempio per memorizzare dati, eseguire azioni, ritornare valori.

Visto che una struttura ad albero può essere costruita e restituita in maniera automatica, utilizzando l'istruzione "`-> ^(...)`", si è preferito scegliere questa strada, che comporta i seguenti vantaggi:

- il codice della grammatica risulta più "pulito" e leggibile;
- si evita di doversi preoccupare di quegli errori di sintassi che si possono commettere inserendo il codice nel linguaggio-target;
- si è evitato di dover scrivere due volte tale codice, per le due grammatiche (Java e JavaScript), cosa necessaria nel caso di un approccio diverso.

A titolo di esempio si illustra la regola precedente, riportata di seguito per comodità; per i dettagli sulla sintassi e sulle possibilità offerte da ANTLR si rimanda a [611].

```
intersection
     :    c1=classDescr AND c2=classDescr (AND c3=classDescr)*
          -> ^(AND $c1 $c2 $c3*);
```

Nell'esempio, "`intersection`" è il nome della regola, che viene "attivata" ("match" della regola) qualora nella stringa in input si incontri una sequenza composta da:

- una sequenza che a sua volta "attivi" la regola "`classDescr`" (la possibilità di combinare regole consente la composizione per ricorsione);
- il token "`AND`";
- una ulteriore sequenza per la regola "`classDescr`";
- la ripetizione, per un numero arbitrario di volte dei due punti precedenti, codificati mediante "`(AND c3=classDescr)*`".

Nel caso la regola "`intersection`" venga attivata, l'istruzione "`-> ^(AND $c1 $c2 $c3*)`" impone al parser di creare e ritornare un sotto-albero composto da:

- il token "`AND`" come root node;
- i figli "`$c1 $c2 $c3*`" (dove "`$c1`" e "`$c1`" sono variabili interne al parser in cui sono state memorizzate le prime due "`classDescr`", mentre "`$c3*`" è un array in cui sono state memorizzate le ulteriori, eventuali, "`classDescr`").

Nella figura 7, riportata precedentemente, è possibile individuare due esempi della regola "`intersection`" in azione.

3.2.1.3 – Struttura della grammatica

Coerentemente con quanto indicato in "2.3.3.2 – Classi (class descriptions)", si è adottato la seguente tassonomia per le "class descriptions", in luogo di quella proposta in [505]:

A – classByName (tipo 1);

B – classByEnum (tipo 2);

C – classByPropRestr (tipo 3), con i sottotipi:

C.1 – classByValueRestr ([505], Value Restriction):

C.1.A – owl:allValuesFrom; *

C.1.B – owl:someValuesFrom; *

C.1.C – owl:hasValue;

C.2 – classByCardRestr ([505], Cardinality Restriction):

C.2.A - owl:maxCardinality;

C.2.B – owl:minCardinality;

C.2.C – owl:cardinality;

D - classByBoole (tipi 4, 5 e 6), con i sottotipi:

D.4 – owl:intersectionOf *

D.5 – owl:unionOf *

D.6 – owl:complementOf *

Tutte le "class descriptions" precedenti sono catturate dalla regola omonima, che raggruppa il primo livello:

```
classDescr
     :    classByBoole
     |    classByPropRestr
     |    classByName
     |    classByEnum;
```

Tale regola non produce alcun albero, in quanto tale operazione è delegata alle sotto-regole (o alle loro ulteriori sotto-regole), come riportato di seguito:

```
classByName
     :    URI_REF;
```

```
classByEnum
     :    '{' (URI_REF)+ '}' -> ^(ENUM URI_REF+);
```

```
classByPropRestr
     : classByValueRestr
     | classByCardRestr;
```

```
classByBoole
     :    '(' intersection ')' -> ^(intersection)
     |    '(' union ')' -> ^(union)
     |    complement;
```

La “regola principale”, che ritorna l'albero complessivo è invece la seguente:

```
axiom
     :    intersection   EOF       -> ^(AXIOM intersection)
     |    union          EOF       -> ^(AXIOM union)
     |    classDescr     EOF       -> ^(AXIOM classDescr);
```

Come si vede, tale regola utilizza anche le sotto-sotto regole “`intersection`” e “`union`”: tale scelta è stata fatta per evitare il problema della “left recursion”, che viene illustrata nella sezione seguente.

3.2.1.4 – Il problema della Left Recursion

Vale la pena di soffermarsi sulle due ultime regole "`classByBoole`" e "`axiom`", in quanto la loro struttura, apparentemente "innaturale" è stata scelta per evitare il problema di "left recursion" [607] che può affliggere i parser LL [606], come ANTLR.

Il problema si verificherebbe nel caso si adottassero le seguenti regole, più "naturali" ad una prima analisi:

```
axiom
     :    classDescr     EOF       -> ^(AXIOM classDescr);
```

```
classDescr
     :    classByBoole
     |    classByPropRestr
     |    classByName
     |    classByEnum;
```

```
classByBoole
     :    intersection -> ^(intersection)
     |    union -> ^(union)
     |    complement;
```

```
intersection
     :    c1=classDescr AND c2=classDescr (AND c3=classDescr)*
          -> ^(AND $c1 $c2 $c3*);
```

```
union
     :    c1=classDescr OR c2=classDescr (OR c3=classDescr)*
          -> ^(OR $c1 $c2 $c3*);
```

Le ricorsione a sinistra si verificherebbe per le due regole "`intersection`" e "`union`", che per loro natura producono delle "`classDescr`" (attraverso "`classByBoole`") e al contempo le utilizzano.

L'alternativa proposta in [608] renderebbe meno leggibile la grammatica, oltre a non essere di immediata comprensione; la scelta fatta qui, comunque, si pone in tale ottica.

3.2.2 - Override delle eccezioni per robustezza

Per default, i parser generati da ANTLR implementano un meccanismo di “recovery” dagli errori sintattici (nel lexer, a livello di token, e nel parser, a livello di regole), limitandosi a segnalare l'errore con una stampa testuale su “sys.err”.

Tale comportamento, auspicabile forse nella maggior parte delle applicazioni, potrebbe comportare grossi problemi nel nostro caso, in quanto potrebbe comportare asserzioni “errate” sull'ontologia, che ne risulterebbe fortemente danneggiata: il grafo RDF sottostante si troverebbe ad avere un insieme di triple “errate”, la cui rimozione potrebbe essere estremamente difficile.

Inoltre, la mancata segnalazione dell'errore, da parte del parser nel componente client, renderebbe completamente superflua l'integrazione del parser nel client; come indicato in “3 – Approccio”, infatti, il compito del parser nel componente client è proprio la verifica della correttezza sintattica della stringa DL fornita dall'utente.

Ad aggravare la situazione, c'è il fatto che la documentazione ufficiale [611], in cui ci sono indicazioni per effettuare l'override di tale gestione “con recovery” delle “eccezioni”, fa riferimento ad una versione precedente di ANTLR: i metodi da sovrascrivere non sono quelli indicati, in quanto nella nuova versione sono cambiate diverse cose (alcuni metodi sono stati sostituiti *tout-court*, altri hanno cambiato “signature”).

In mancanza di informazioni ufficiali, è stato necessario ricorrere alla mailing list di ANTLR [612], oltre che a una serie di ricerche su internet e sulla wiki di ANTLR [610], per arrivare a capire quali fossero i metodi su cui fare override e come.

Ulteriore complicazione è relativa al fatto che tale override prevede un diverso approccio per ogni target-language: nelle prossime sezioni si riportano le soluzioni specifiche per Java e JavaScript, di seguito invece si riporta l'elemento comune, in quanto intrinseco alla grammatica.

A livello di grammatica, è stato imposto, sulla regola principale, che una volta fatto il match di "`intersection`", "`union`" o "`classDescr`", non deve esserci null'altro nella stringa di input, il che si realizza mediante il token "interno" (definito per default) "**EOF**":

```
axiom
     :    intersection   EOF       -> ^(AXIOM intersection)
     |    union          EOF       -> ^(AXIOM union)
     |    classDescr     EOF       -> ^(AXIOM classDescr);
```

3.2.2.1 – Implementazione dell'override in Java

Questa implementazione è stata la più problematica, in quanto uno dei metodi su cui fare override non prevedeva il lancio di una eccezione (la sua "signature" risulta diversa da quella indicata nella documentazione [611]).

Per aggirare tale ostacolo, si sono dovute creare due eccezioni "customizzate" (una per il lexer e una per il parser), che estendono la "RuntimeException", che consente di non dover dichiarare il suo "lancio" ("throw") nella "signature" del metodo [613]:

> *RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.*
> *A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.*

Le eccezioni "custom" ("`LexerException`" e "`ParserException`") sono state definite:

- inizialmente internamente alla grammatica (come inner-class, una nel lexer e una nel parser), in quanto l'uso di ANTLR-Works ne sarebbe stato impattato (si sarebbe dovuto ricorrere a qualche "trucco" per cambiare il class-path);
- una volta verificata la bontà della soluzione, sono state portate fuori dalla grammatica (che si limita ad utilizzarle).

Di seguito si riporta il codice della seconda implementazione.

3.2.2.1.1 – Override in Java : grammatica

```
@lexer::header{
package eu.servicemix.dl2owl.antlr.generated;
// adding import statement
import eu.servicemix.dl2owl.antlr.LexerException;
}


@parser::header{
package eu.servicemix.dl2owl.antlr.generated;
// adding import statement
import eu.servicemix.dl2owl.antlr.ParserException;
}


//  overriding methods in Parser
@parser::members
{
      @Override
      protected Object recoverFromMismatchedToken(IntStream input, int ttype,
BitSet follow) throws RecognitionException {
            throw new ParserException(input);
      }

      @Override
      public Object recoverFromMismatchedSet(IntStream input,
RecognitionException e, BitSet follow) throws RecognitionException {
          throw e;
      }
}


//  overriding methods in Lexer
@lexer::members
{
      @Override
      public void reportError(RecognitionException recExc) {
            throw new LexerException(recExc, getCharErrorDisplay(recExc.c),
getErrorMessage(recExc, this.getTokenNames()));
      }
}
// overriding catch behavior
@rulecatch {
    catch (RecognitionException e) {
        throw e;
    }
}
```

3.2.2.1.2 – Override in Java : eccezione del lexer

```
package eu.servicemix.dl2owl.antlr;


import org.antlr.runtime.RecognitionException;


/**
 * @author Corrado Campisano
 *
 * Inner class to represent a LexerException, i.e. an exception risen by the
lexer.
 * It stores the token text, error message and token position (row, col) in
the input text.
 */
public class LexerException extends RuntimeException {

      int iRow = -1;
      int iCol = -1;
      String sToken;
      String sError;


      public LexerException(RecognitionException recExc, String sCurrToken,
String sErrMsg) {
            iRow = recExc.line;
            iCol = recExc.charPositionInLine;
            sToken = sCurrToken;
            sError = sErrMsg;
      }


      public String toString(){
            String sTmp = "LEXER ERROR: " + sError;
            sTmp += ",\t token: " + sToken;
            sTmp += ",\t row: " + iRow + ", col: " + iCol;
            return sTmp;
      }
}
```

3.2.2.1.3 – Override in Java : eccezione del parser

```
package eu.servicemix.dl2owl.antlr;


import org.antlr.runtime.IntStream;
import org.antlr.runtime.Token;
import org.antlr.runtime.TokenStream;


/**
 * @author Corrado Campisano
 *
 * Inner class to represent a ParserException, i.e. an exception risen by the
parser.
 * It stores the token text, error message and token position (row, col) in
the input text.
 */
public class ParserException extends RuntimeException {

	IntStream input = null;

	int iRow=-1;
	int iCol=-1;
	String sToken="";
	String sError="Mismatched Token";


	public ParserException(IntStream input) {
		Token currToken = ((TokenStream)input).LT(1);
		iRow = currToken.getLine();
		iCol = currToken.getCharPositionInLine();
		sToken = currToken.getText();
	}


	public String toString(){
		String sTmp = "PARSER ERROR: " + sError;
		sTmp += ",\t token: " + sToken;
		sTmp += ",\t row: " + iRow + ", col: " + iCol;
		return sTmp;
	}
}
```

3.2.2.2 – Implementazione dell'override in JavaScript

L'implementazione dell'override in JavaScript è risultata molto più semplice, a livello "pratico" (mancando il problema della "signature" e della eccezione) anche se leggermente più "oscuro" (per la minore esperienza con questo linguaggio, specialmente nella sua versione "vagamente object oriented").

E' stato sufficiente inserire quanto segue nella grammatica:

```
//  overriding methods in Parser
@parser::members
{
      Dl2OwlJsSafeParser.prototype.emitErrorMessage = Calculator.logError;
}
//  overriding methods in Lexer
@lexer::members
{
      Dl2OwlJsSafeLexer.prototype.emitErrorMessage = Calculator.logError;
}
```

Quanto sopra semplicemente ridefinisce la funzione JavaScript che viene invocata per la stampa del messaggio di errore, in particolare assegnandola ad un metodo della classe "`Calculator`", che viene definita nel client (nel senso del codice JavaScript che invoca il parser sul componente client di Semantic Turkey) come segue:

```
var parsedOK = true;


var Calculator = {

  logError: function(msg) {
      parsedOK = false;
      appendLogs("ERROR:\n" + msg);
  },

  ...

};
```

Nel client, dopo l'invocazione del parser, viene valutata la variabile di controllo "`parsedOK`" per verificare la validità sintattica della stringa inserita.

3.2.3 – Correzione di un bug nelle librerie JavaScript

Come accade con tutti i software, gli sviluppatori possono commettere degli errori e quanto da loro prodotto può presentare dei "bug".

Nel caso di software opensource, però, gli utilizzatori hanno due vantaggi:

- potendo esaminare i sorgenti, possono identificare direttamente le cause del problema, anziché limitarsi a segnalarlo;
- potendo modificare i sorgenti, hanno la possibilità di provare a correggerlo, proporre la "patch" al mantainer e contribuire direttamente alla soluzione del problema.

Un caso del genere si è verificato durante la realizzazione di questo lavoro: si rimanda a [615] per i dettagli sulla segnalazione, sulla correzione proposta e la sua successiva accettazione.

Qui ci si limita a sottolineare come, curiosamente, l'origine dell'errore ha una qualche attinenza con la materia del presente lavoro: la causa del problema consiste in un errore di natura tassonomica relativamente alla gerarchia delle eccezioni.

Il problema scaturisce dal fatto che l'eccezione "`NoViableAltException`" è una sottoclasse dell'eccezione "`RecognitionException`", per cui nella clausola "`catch`" riportata sotto (codice originale), il blocco del secondo "`if`" non viene mai eseguito, in quanto il primo "`if`" intercetta anche l'occorrenza delle eccezioni "`NoViableAltException`":

```
catch (re) {
      if ( re instanceof org.antlr.runtime.RecognitionException ) {
            this.reportError(re);
      } else if (re instanceof org.antlr.runtime.NoViableAltException) {
            this.reportError(re);
            this.recover(re);
      } else {
            throw re;
      }
}
```

E' stato quindi sufficiente invertire l'ordine delle due clausole “catch” per correggere l'errore:

```
catch (re) {
      if (re instanceof org.antlr.runtime.NoViableAltException) {
            this.reportError(re);
            this.recover(re);
      } else if (re instanceof org.antlr.runtime.RecognitionException) {
            this.reportError(re);
      } else {
            throw re;
      }
}
```

3.3 – Progetto DL2OWL

Come già anticipato, si è scelto di sviluppare i due parser come progetti “standalone”, i cui “deliverables” risultanti consistono:

- in una libreria Java per il lato server;
- un insieme di script JavaScript per il lato client.

Tale scelta di “modularità” è giustificabile con una serie di motivazioni:

- allo stato attuale, Semantic Turkey utilizza già tutta una serie di librerie e di software di terze parti;
- l'implementazione dei parser, che si limitano a ricevere in input una stringa ed a restituire in output una struttura dati, è indipendente dal resto del codice di Semantic Turkey;
- l'integrazione della libreria nell'ontology editor viene ridotta a:
    - due semplici chiamate (sia nel client che nel server), per effettuare la sola operazione di parsing;
    - nell'elaborazione, lato server, della struttura ad albero ritornata (operazione che, a differenza di quella precedente, è molto più “coupled” con il resto del codice di Semantic Turkey);

In particolare, si sottolinea come la modularità e il minore livello di “coupling” offrono diversi vantaggi, per quanto riguarda la manutenzione correttiva ed evolutiva del software risultante dal presente lavoro:

- mantenendo la struttura ad albero prodotta in output dal parser (la quale, a meno delle etichette utilizzate, è propria della sintassi delle DL), è possibile manutenere la grammatica in modo del tutto indipendente dall'integrazione della stessa in Semantic Turkey;
- qualora si volesse implementare un parser addizionale per la “Manchester Syntax”, sarebbe sufficiente sviluppare in modo analogo la nuova grammatica e generare il parser, limitando l'integrazione in Semantic Turkey alle semplici chiamate per l'operazione di

parsing, in quanto per l'elaborazione dell'output si potrebbe riutilizzare, con minime modifiche, il codice già sviluppato (questo avendo l'accortezza di sviluppare la nuova grammatica in modo che produca in output una struttura ad albero identica all'attuale, ipotesi assolutamente lecita visti i requisiti).

Di seguito si illustrano i dettagli dei progetti.

3.3.1 – Gestione del progetto con Maven

Il codice di Semantic Turkey è organizzato come un insieme di progetti Maven [616], un potente strumento di supporto allo sviluppo di applicazioni Java, che automatizza tutta una serie di attività:

- gestione delle dipendenze (recupera tutte le librerie necessarie al codice del progetto, definite dichiarativamente in un file di configurazione);
- compilazione dei sorgenti e build dei package (jar, war, ecc);
- esecuzione dei test (JUnit);
- generazione della documentazione (JavaDoc, report dei test eseguiti, elenco dipendenze, informazioni generali sul progetto), eventualmente pubblicabile automaticamente su internet.

Nel caso di un software sviluppato da un gruppo di più persone, l'utilizzo di uno strumento come Maven, stante la sua possibilità di integrarsi con i più diffusi Concurrent Versioning Systems, oltre a consentire ai diversi sviluppatori di lavorare in parallelo su diversi “trunk”, consente al “mantainer” di effettuare agevolmente il “merge” dei vari contributi nel “trunk” principale, una volta che, grazie agli automatismi visti sopra, ha potuto verificare velocemente la validità dei vari contributi.

3.3.2 – Organizzazione dei progetti Maven

Dovendo realizzare due distinti parser, uno per il linguaggio Java e l'altro per JavaScript, si è scelto di definire due progetti Maven distinti e di raggrupparli in un ulteriore progetto "contenitore":

- dl2owl : progetto contenitore;
- dl2owl-java : progetto per il target Java;
- dl2owl-javascript : progetto per il target JavaScript.

Mentre il progetto Java risulta "full mavenized" (test JUnit, packaging jar), quello JavaScript purtroppo non lo è, a causa della minore maturità di Maven (nato per progetti Java) e del plugin ANTLRv3-Maven [617] su tale linguaggio:

- un bug di antlr-maven-plugin (nella fase di generazione della grammatica) richiede degli interventi manuali sul file di configurazione del progetto (pom.xml);
- non è stato possibile implementare i test automatici;
- non è stato possibile produrre un package, ma occorre recuperare manualmente l'insieme di script prodotti (quelli generati e quelli sviluppati come "client").

Nella sezione seguente, dopo aver introdotto l'architettura di Semantic Turkey, si illustra l'integrazione di tali progetti nel codice dello strumento.

4 – Architettura

Nei paragrafi che seguono si descrive l'architettura di Semantic Turkey e l'integrazione del parser nei sui componenti client e server.

4.1 – Architettura di Semantic Turkey

Semantic Turkey [203] è realizzato come una Mozilla Extension [205]: a questa categoria appartengono le componenti software che consentono di aggiungere nuove funzionalità alle applicazioni della famiglia Mozilla (Firefox, SeaMonkey e Thunderbird).

Tale scelta implementativa è giustificata dall'obiettivo per cui tale software è stato sviluppato: come anticipato in precedenza, si tratta di uno strumento principalmente pensato il “semantic bookmarking”, ma utilizzabile anche per lo sviluppo di ontologie.

L'integrazione in Mozilla Firefox (il più diffuso browser open source) consente quindi di disporre dell'elemento di base per la navigazione nel Web, a cui vengono aggiunte, tramite l'estensione, le capacità “semantiche” desiderate.

Al fine di consentire lo sviluppo di ontologie, però, oltre al componente client con cui interagisce l'utente, è stato necessario dotare tale estensione del browser di un componente capace di memorizzare l'ontologia (meccanismo di persistenza), oltre che di un componente server intermedio, che consentisse all'interfaccia utente di interagire con la base dei dati.

Il risultato consiste quindi in un'**architettura a tre livelli** (*three tiers architecture*):

- un'interfaccia utente (*presentation layer / view*): realizzato come estensione Mozilla (in senso stretto), basata sulle tecnologie XUL, XBL, XPCOM, Javascript;
- un middleware (*business logic layer / controller*): realizzato come insieme di servlet Java, esposte tramite un server HTTP (Jetty) che viene caricato dinamicamente all'apertura del browser, attraverso un bundle OSGi (Felix framework);
- un meccanismo di persistenza (*data layer / model*): realizzato utilizzando le OWL ART API [206], un wrapper di differenti tecnologie per la memorizzazione triple RDF (attualmente viene utilizzata Sesame [207])

Il termini “componente client” e “componente server”, introdotti in precedenza per esigenze di trattazione, coincidono rispettivamente con il *presentation layer / view* e con il *business logic layer / controller*.

Nella figura seguente si riporta l'architettura completa di Semantic Turkey.

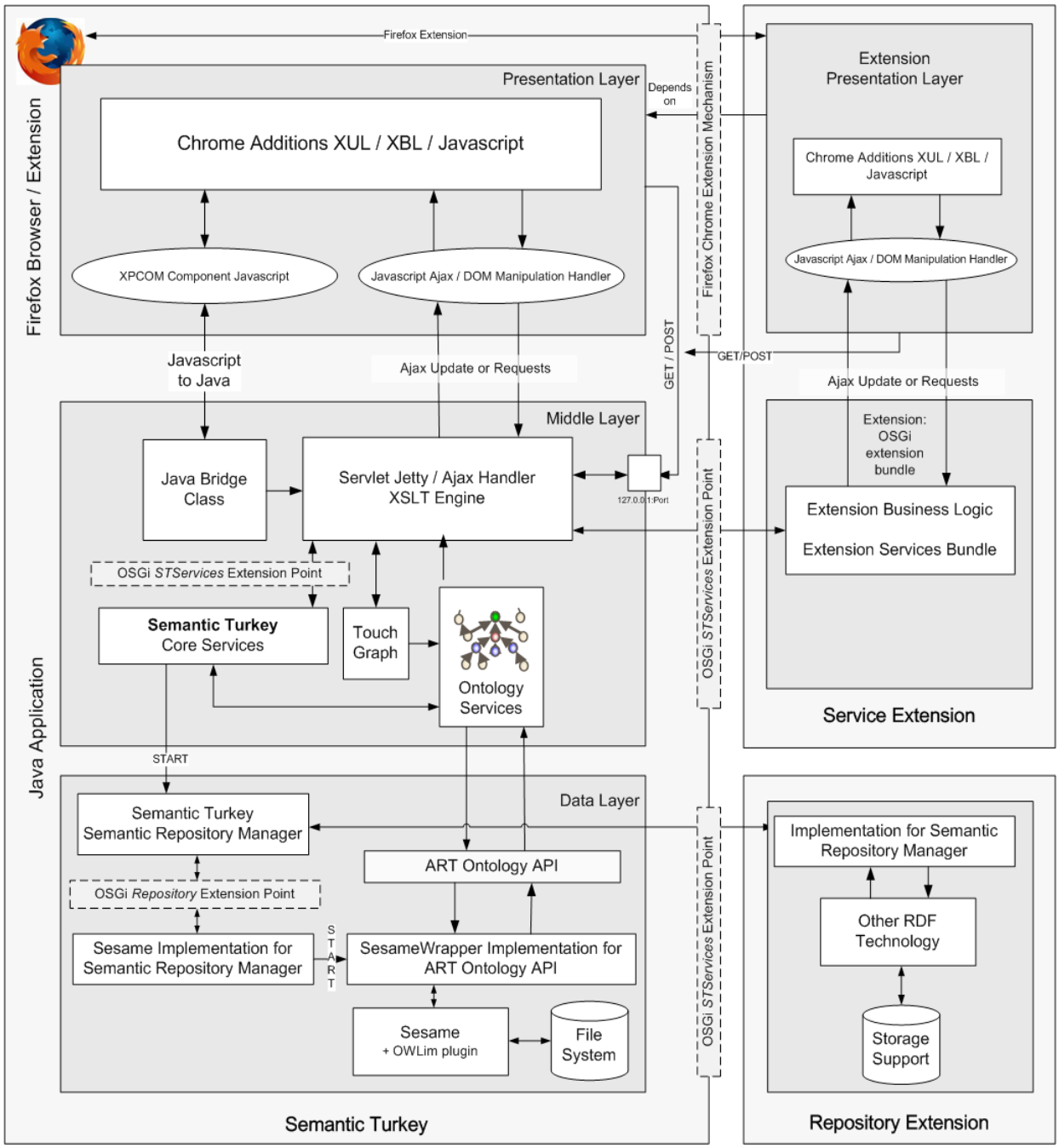


Figura 10: Architettura di Semantic Turkey

4.1.2 – Struttura dei sorgenti di Semantic Turkey

Prima di affrontare il tema dell'integrazione del parser nell'ontology editor, è utile analizzare brevemente l'organizzazione dei sorgenti di Semantic Turkey.

I tre tier dell'architettura sono realizzati attraverso i seguenti alberi di sorgenti:

- componente client: si tratta semplicemente di un insieme strutturato di script JavaScript e interfacce utente XUL, raggruppate nel “SemanticTurkeyUI”;
- componente server: realizzato mediante due progetti Maven:
    - “SemanticTurkeyBM”, che implementa il servlet container e lo interfaccia al livello di persistenza, oltre a fornire una serie di classi di utilità;
    - “SemanticTurkeySE”, che implementa nelle varie servlet la logica applicativa;
- meccanismo di persistenza: realizzato mediante due progetti Maven:
    - “OwlArtApi” (layer astratto, wrapper delle specifiche implementazioni);
    - “OwlArtApi–Sesame” (implementazione OWL ART API su Sesame2);

Come vedremo nel seguito, l'integrazione del parser è stata realizzata intervenendo solo sui sorgenti di “SemanticTurkeyUI” e “SemanticTurkeySE”; ci si è invece limitati ad analizzare i sorgenti di “SemanticTurkeyBM” (per la comprensione del flusso di chiamate e per l'uso delle classi di utilità) ed a consultare la JavaDoc di “OwlArtApi”(per l'implementazione della logica di elaborazione dell'albero prodotto dal parser).

4.3 – Integrazione di DL2OWL

Nei paragrafi seguenti si illustrano i dettagli dell'integrazione del parser in “SemanticTurkeyUI” ed in “SemanticTurkeySE”.

4.3.1 – Integrazione lato client

Prima di poter effettuare l'integrazione del codice JavaScript del parser, è stato necessario modificare l'interfaccia utente, aggiungendo gli elementi grafici ed interattivi, tramite i quali l'utente potesse:

- richiedere di applicare una restrizione ad una classe dell'ontologia (ovvero definire un assioma su essa);
- selezionare il tipo di assioma da definire;
- inserire la stringa DL che definisce l'assioma in termini di classi ed individui;
- inviare la stringa al componente server, una volta che il parser lato client ne abbia verificato la correttezza sintattica;
- visualizzare eventuali errori:
    - restituiti dal componente client a seguito della verifica sintattica;
    - restituiti dal componente server a seguito dell'elaborazione.

A tale scopo, facendo riferimento alla Figura 6: Semantic Turkey, si è proceduto come segue:

- è stato aggiunto un pulsante al "Class Panel" (evidenziato nel rettangolo blu), modificando il relativo XUL (*class.xul*) e aggiungendo l'event handler nel relativo JavaScript (*classEvents.js*);
- è stata creata una nuova finestra pop-up (evidenziata nel rettangolo rosso), creando un nuovo XUL (*dl2owl.xul*) ed il relativo JavaScript (*Dl2OwlSafe.js*), oltre alla funzione per l'invio della stringa DL al componente server (*SERVICE_Cls.jsm*).

L'integrazione del parser JavaScript è stata quindi relativamente semplice:

- sono stati aggiunti gli script generati (*Dl2OwlJsSafeLexer.js*, *Dl2OwlJsSafeParser.js*), il file dei token generato (*Dl2OwlJsSafe.tokens*) e la libreria runtime di ANTLR per JavaScript (*antlr3-all-patched.js*), alla quale era stata applicata la patch illustrata precedentemente;
- l'invio della stringa è stato subordinato all'invocazione del parser ed al risultato da questo ritornato: in caso di errore l'utente viene avvertito tramite pop-up (alert) e nessuna richiesta viene inoltrata al server.

Al fine di minimizzare l'impatto delle modifiche su "SemanticTurkeyUI", tutto il codice e gli elementi grafici aggiuntivi sono stati posizionati in una sottocartella "*dl2owl*" della cartella "*class*" in cui sono presenti gli analoghi elementi relativi al "Class Panel".

4.3.2 – Integrazione lato server

L'integrazione lato server è risultata invece leggermente più laboriosa, in quanto richiedeva diverse fasi:

- integrazione nella servlet del “riconoscimento” della chiamata lato client e relativo recupero dei parametri (entry point);
- invocazione del parser e generazione della struttura ad albero (AST) a partire dalla stringa DL ricevuta;
- elaborazione dell'albero ed invocazione delle OwlArtApi per la memorizzazione dell'assioma come insieme di triple RDF.

Nel seguito si illustrano separatamente tali fasi.

4.3.2.1 – Entry point

Come anticipato, tutte le servlet che implementano la logica applicativa fanno parte dell'albero di sorgenti "SemanticTurkeySE"; in particolare, tutte le operazioni sulle classi sono implementate nella classe "*Cls.java*", che si comporta a tutti gli effetti come una HttpServlet, pur derivando da una diversa gerarchia (ServiceExtension di OSGi).

L'aggancio all'entry point è stato realizzato, conformandosi alla struttura del codice preesistente, con un semplice "*if*" nel processamento della HttpServletRequest, come illustrato nel blocco di codice sottostante.

```
// DL2OWL METHODS
if (request.equals(Dl2Owl.addDlRestrictionRequest)) {
        String targetClassName = setHttpPar(Dl2Owl.targetClassNamePar);
        String owlAxiomType = setHttpPar(Dl2Owl.owlAxiomTypePar);
        String dlDescriptionToParse = setHttpPar(Dl2Owl.dlDescriptionToParsePar);

        checkRequestParametersAllNotNull(Dl2Owl.targetClassNamePar, Dl2Owl.owlAxiomTypePar,
Dl2Owl.dlDescriptionToParsePar);

        response = Dl2Owl.addRestriction(servletUtilities, targetClassName, owlAxiomType,
dlDescriptionToParse);
}

// GET CLASS METHODS
} else if (request.equals(getInstanceListRequest)) {
        String clsQName =
servletUtilities.removeInstNumberParentheses(setHttpPar(clsQNameField));
        checkRequestParametersAllNotNull(clsQNameField);

        response = getInstancesListXML(clsQName);
} else if (request.equals(getSuperClassesRequest)) {
...
```

Tutto il resto dell'integrazione, come pure le varie costanti letterali "`Dl2Owl.addDlRestrictionRequest`" e simili, sono state collocate in una classe separata (*Dl2Owl.java*) in modo da minimizzare l'impatto delle modifiche sul codice preesistente.

Come si vede dal codice sopra riportato, il metodo "`Dl2Owl.addRestriction(...)`" effettua tutta dell'elaborazione e ritorna direttamente una HttpReponse, opportunamente popolata (sia in caso di successo che di eccezione), al blocco dell'entry point, che a sua volta ritorna tale risposta al client.

Nel riquadro sottostante viene riportato il codice di tale metodo:

```
public static Response addRestriction(ServletUtilities servletUtilities, String
targetClassName, String restrictionType, String dlRestrictionToParse) {

	String strParsingResult = "";
	String strExecutionResult = "";
	ARTURIResource subject = null;

	// check restriction type
	if(!restrictionType.equals(dlRestrictionTypeParEquivalentClass) &&
			!restrictionType.equals(dlRestrictionTypeParSubClassOf) &&
			!restrictionType.equals(dlRestrictionTypeParDisjointWith)) {

		String msg = "Restrition type '" + restrictionType + "' not supported";

		return servletUtilities.createExceptionResponse(addDlRestrictionRequest, msg);
	}

	OWLModel ontModel = ProjectManager.getCurrentProject().getOntModel();


	// check target class
	String targetClassURI;
	try {
		targetClassURI = ontModel.expandQName(targetClassName);

		subject = ontModel.createURIResource(targetClassURI);

		boolean exists = ModelUtilities.checkExistingResource(ontModel, subject);
		if (!exists) {
			String msg = "there is NO resource with name '" + targetClassName + "', to
be used as target Class";

			return servletUtilities.createExceptionResponse(addDlRestrictionRequest,
msg);
		}
	} catch (ModelAccessException e) {
		return servletUtilities.createExceptionResponse(addDlRestrictionRequest, e);
	}


	// parse tree from DL restriction statement
	CommonTree tree = null;
	try {
		tree = parseDlRestriction(dlRestrictionToParse);
	} catch (Exception e) {
		return servletUtilities.createExceptionResponse(addDlRestrictionRequest,
e.toString());
	}


	// add restriction to ontModel from tree
	try {
	    if (restrictionType.equals(dlRestrictionTypeParEquivalentClass)) {
		// shall add a
		addGenericAxiom(subject, tree, ontModel, OWL.EQUIVALENTCLASS);

	    } else if (restrictionType.equals(dlRestrictionTypeParSubClassOf)){
		// shall add a
		addGenericAxiom(subject, tree, ontModel, RDFS.SUBCLASSOF);

	    } else if (restrictionType.equals(dlRestrictionTypeParDisjointWith)) {
		// shall add a
		addGenericAxiom(subject, tree, ontModel, OWL.DISJOINTWITH);
	    }
	} catch (Exception e) {
		return servletUtilities.createExceptionResponse(addDlRestrictionRequest,
e.toString());
	}

	// create XML response
	ResponseREPLY response =
ServletUtilities.getService().createReplyResponse(addDlRestrictionRequest, RepliesStatus.ok);
	Element dataElement = response.getDataElement();
```

```
        Element restrictionElement = XMLHelp.newElement(dataElement, addDlRestrictionRequest);

        Element targetClassNameElement = XMLHelp.newElement(restrictionElement,
targetClassNamePar);
        targetClassNameElement.setAttribute("name", targetClassName);

        Element dlRestrictionTypeElement = XMLHelp.newElement(restrictionElement,
owlAxiomTypePar);
        dlRestrictionTypeElement.setAttribute("type", restrictionType);

        Element dlRestrictionToParseElement = XMLHelp.newElement(restrictionElement,
dlDescriptionToParsePar);
        dlRestrictionToParseElement.setTextContent(dlRestrictionToParse);

        Element parsingResultElement = XMLHelp.newElement(restrictionElement, "ParsingResult");
        parsingResultElement.setTextContent(strParsingResult);

        Element executionResultElement = XMLHelp.newElement(restrictionElement,
"ExecutionResult");
        executionResultElement.setTextContent(strExecutionResult);

        return response;
}
```

4.3.2.2 – Invocazione del parser e generazione dell'AST

Questa operazione è invocata direttamente dal metodo "`Dl2Owl.addRestriction(...)`", tramite il metodo di supporto "`parseDlRestriction(...)`", che si riporta di seguito:

```
private static CommonTree parseDlRestriction(String dlRestrictionToParse) throws Exception {
       CommonTree tree=null;

       ANTLRStringStream input = new ANTLRStringStream(dlRestrictionToParse);
       Dl2OwlJavaSafeLexer lexer = new Dl2OwlJavaSafeLexer(input);
       TokenStream tokens = new CommonTokenStream(lexer);
       Dl2OwlJavaSafeParser parser = new Dl2OwlJavaSafeParser(tokens);

    try {
       axiom_return ret = parser.axiom();
       tree = (CommonTree) ret.getTree();
       logger.info("\nRESULT   :   " + AntlrV3Helper.printTreeHelper(tree));

       } catch (RecognitionException e) {
              String sErr = "";
              parser.emitErrorMessage(sErr);
              logger.error("\nERROR   :   " + sErr);

              throw e;
       }

    return tree;
}
```

Grazie all'override implementato in precedenza, l'operazione di parsing può effettivamente sollevare un'eccezione, che viene ritornata al metodo chiamante, in cui viene utilizzata per generare una HttpResponse che riporti l'errore (tramite il metodo di utilità, fornito da "SemanticTurkeyBM", "`servletUtilities.createExceptionResponse(...)`").

4.3.2.3 – Processamento dell'AST sull'ontologia

Come visibile nel metodo "`Dl2Owl.addRestriction(...)`", l'operazione di elaborazione dell'AST, struttura ad albero generata dal parser, viene delegata al metodo "`addGenericAxiom(...)`", riportato nel riquadro sottostante.

```
private static void addGenericAxiom (ARTResource subject, CommonTree tree, OWLModel ontModel,
String strAxiomPredicate) throws Exception {
        ARTURIResource predicate = ontModel.createURIResource(strAxiomPredicate);

        if (tree != null) {
                // root node AXIOM to be skipped
                CommonTree root = (CommonTree) tree.getChild(0);

                // generates the object from the tree
                ARTNode object = getRdfNodeFromTree(root, ontModel);

                // asserts target class (subject) is equiv./subcl./disj. (predic.)
                // with the object, built from the DL statement
                addTriple(ontModel, subject, predicate, object);
    }
}
```

Tale metodo si limita ad inserire nell'ontologia una sola tripla, quella relativa all'assioma, delegando a sua volta l'inserimento delle eventuali ulteriori triple al metodo "`getRdfNodeFromTree(...)`".

Tale approccio è giustificato dalla struttura stessa di un grafo RDF, formato da un insieme di triple soggetto-predicato-oggetto, come illustrato nella Figura 3: modello di una "tripla" RDF.

La tripla inserita da "`addGenericAxiom(...)`" è infatti quella che ha:

- per soggetto, la classe a cui applicare l'assioma;
- per predicato, il tipo di assioma da applicare (equivalentOf, subClassOf, disjointWith);
- per oggetto, il nodo (la risorsa) RDF generata dal metodo "`getRdfNodeFromTree(...)`".

Nel riquadro sottostante si riporta quest'ultimo metodo, che si occupa, ove necessario, di inserire nell'ontologia tutte le triple relative ad eventuali nodi RDF compositi (quelli legati alle eventuali "class descriptions" di natura ricorsiva).

```
private static ARTNode getRdfNodeFromTree(CommonTree tree, OWLModel ontModel) throws
Exception {

        ARTNode artNodeRet = null;

        String strNodeType = "";

        strNodeType = Dl2OwlJavaSafeParser.tokenNames[tree.getType()];

        // A -  classByName (type 1) :
        if (strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.URI_REF])) {
                artNodeRet = classByName(tree, ontModel);
        }

        // B - classByEnum (type 2) :
        if (strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.ENUM])) {
                artNodeRet = classByEnum(tree, ontModel);
        }

        //C - classByPropRestr (type 3) :
        if (strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.ALL_VALUES])
        || strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.SOME_VALUES])
        || strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.HAS_VALUE])
        || strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.HAS_CARD])
        || strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.MIN_CARD])
        || strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.MAX_CARD])
        ) {
                artNodeRet = classByPropRestr(tree, ontModel);
        }

        //D - classByBoole (type 4, 5 and 6) :
        if (strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.AND])
                || strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.OR])
                || strNodeType.equals(Dl2OwlJavaSafeParser.tokenNames[Dl2OwlJavaSafeParser.NOT])
        ) {
                artNodeRet = classByBoole(tree, ontModel);
        }

        // check one of the "if" above was executed
        if (artNodeRet == null) {
                throw new Exception("Exception : unrecognized tree or subtree:\n" +
AntlrV3Helper.printTreeHelper(tree));
        }
}
```

A titolo di esempio e per non appesantire la trattazione, si riportano i primi due metodi a loro volta chiamati: “`classByName(...)`” e “`classByEnum(...)`”.

Il metodo "`classByName(...)`" è il più semplice, in quanto deve limitarsi a restituire un nodo (una risorsa) molto semplice, essendo questa univocamente identificata con il suo URI:

```
private static ARTNode classByName(CommonTree tree, OWLModel ontModel) throws Exception {

	String targetClassURI;
	String strNodeValue = tree.getText();

	targetClassURI = ontModel.expandQName(strNodeValue);
	logger.debug("targetClassName: " + strNodeValue + " expanded in: " + targetClassURI);

	ARTURIResource res = ontModel.createURIResource(targetClassURI);

	return res;
}
```

Il metodo "`classByEnum(...)`" è leggermente più complesso e per questo consente di illustrare il meccanismo della costruzione di triple RDF "concatenate":

```
private static ARTNode classByEnum(CommonTree tree, OWLModel ontModel) throws Exception {

	ARTURIResource predOneOf = ontModel.createURIResource(OWL.ONEOF);

	ARTURIResource predType = ontModel.createURIResource(RDF.TYPE);
	ARTURIResource objClass = ontModel.createURIResource(OWL.CLASS);

	ARTURIResource predFirst = ontModel.createURIResource(RDF.FIRST);
	ARTURIResource predRest = ontModel.createURIResource(RDF.REST);

	ARTURIResource objNil = ontModel.createURIResource(RDF.NIL);

	// BNode to store the AND/OR
	ARTBNode subjOneOf = ontModel.createBNode();

	if (tree.getChildCount()>0) {

		ARTBNode prevOperand = ontModel.createBNode();

		// BOTTOM-UP LOOP to allow for #rest links
		// will loop from the last child, whose #rest will be linked to #nil
		// and up to the first child, whose #rest will be linked to the second element
		for (int i=tree.getChildCount()-1; i>=0; i--) {

			CommonTree tempChild = (CommonTree)tree.getChild(i);

			// BNode to store the #first and #rest links
			ARTBNode tempOperand = ontModel.createBNode();

			ARTNode tempObject = getRdfNodeFromIndividualLeaf(tempChild, ontModel);

			// assert list links :

			// 1 : assert the #first of the current BNode (tempOperand)
			addTriple(ontModel, tempOperand, predFirst, tempObject);

			// 2 : assert the #rest of the current BNode (tempOperand)
			if (i==tree.getChildCount()-1) {
				// last BNode has #nil as #rest
				addTriple(ontModel, tempOperand, predRest, objNil);
			} else {
				// other nodes have the previous (it's bottom-up) as #rest
				addTriple(ontModel, tempOperand, predRest, prevOperand);
			}
```

```
                        // update prevOperand
                        prevOperand = tempOperand;
                }

                // assert the BNode storing the ENUM is a class
                addTriple(ontModel, subjOneOf, predType, objClass);

                // assert the BNode storing the ENUM is a OWL.ONEOF
                addTriple(ontModel, subjOneOf, predOneOf, prevOperand);

        } else {
                throw new Exception("Exception : ENUMERATION node with NO CHILDREN!\n" +
AntlrV3Helper.printTreeHelper(tree));
    }

        // return the BNode storing the ENUM
        return subjOneOf;
}
```

5 – Conclusioni

Il risultato di questo lavoro consiste nell'implementazione di una funzionalità fondamentale per un ontology editor: la possibilità di definire nuove classi a partire da quelle esistenti, attraverso assiomi forniti dall'utente, sotto forma di espressioni in Description Logics.

Per realizzare tale funzionalità, è stato necessario:

- implementare un parser generato da grammatica, per l'analisi lessicale e sintattica della stringa rappresentante l'assioma;
- integrare tale parser nei componenti client e server dell'ontology editor, dai quali viene invocato;
- implementare nel componente server l'elaborazione dell'output prodotto dal parser, in modo da inserire l'assioma nell'ontologia in lavorazione.

A tale risultato si è giunti considerando come requisiti:

- gli standard W3C per la scrittura di ontologie, principalmente RDF(S) e OWL;
- la sintassi delle Description Logics per la specifica degli assiomi;
- la corrispondenza tra la sintassi DL ed i costrutti OWL;
- il confronto con le implementazioni realizzate in Top Braid Composer e Protégé.

In merito agli sviluppi futuri, alcune scelte progettuali, quali utilizzare un generatore di parser basato su grammatica e ridurre il “coupling” tra l'ontology editor ed il parser, consentono di replicare facilmente il risultato del presente lavoro, in modo da supportare sintassi alternative per la stringa dell'assioma, quali ad esempio la Manchester Syntax.

6 – Riferimenti

Di seguito i riferimenti divisi per argomento.

6.1 – Intelligenza artificiale, ingegneria della conoscenza, semantic web

[100] Stuart Russel, Peter Norvig
"Intelligenza artificiale, un approccio moderno"
Pearson – Prentice Hall (2003, 2005)

[101] Tim Berners-Lee (1998)
"Semantic Web Road Map - W3C Design Issues"
http://www.w3.org/DesignIssues/Semantic.html

[102] Tim Berners-Lee, James Hendler and Ora Lassila
"The Semantic Web"
Scientific American Magazine (2001)
http://www.sciam.com/article.cfm?id=the-semantic-web

[103] "OWL Web Ontology Language Use Cases and Requirements"
W3C Recommendation 10 February 2004
http://www.w3.org/TR/webont-req/#onto-def

[104] Tim Berners-Lee
"Semantic Web – XML2000"
http://www.w3.org/2000/Talks/1206-xml2k-tbl

[105] Charles Sanders Peirce (1935)
Collected Papers
Cambridge, Harvard University Press

[106] Ora Lassila, Deborah McGuinness
The Role of Frame-Based Representation on the Semantic Web
Nokia Research Center, Stanford University (2001)

[107] Tom R. Gruber
A translation approach to portable ontologies
Knowledge Acquisition (1993)

[108] IEEE Internet Computing
Special Issue on "Overcoming Information Overload Issues"
http://www.dbgroup.unimo.it/ic-overload/overload.htm

6.2 – Ontology editors

[200] Ontology editor, definizione ed esempi
http://en.wikipedia.org/wiki/Ontology_editor

[201] Top Braid Composer
Prodotto commerciale
http://www.topquadrant.com/products/TB_Composer.html

[202] Protégé
Implementazione opensource della Università di Stanford, CA, USA
http://protege.stanford.edu

[203] Semantic Turkey
Implementazione del gruppo ART dell'Università Tor Vergata
http://semanticturkey.uniroma2.it

[204] Semantic Turkey team
http://semanticturkey.uniroma2.it/aboutus/

[205] Mozilla Extension
https://developer.mozilla.org/en/Extensions

[206] OWL ART API
Abstraction layer over different RDF triple store technologies
http://art.uniroma2.it/owlart/

[207] Sesame
Open source Java framework for storage and querying of RDF data
http://www.openrdf.org/doc/sesame2/2.3.1/users/index.html

6.3 – Standard di base per il Semantic Web

[300] Cool URIs for the Semantic Web
W3C Interest Group Note 03 December 2008
http://www.w3.org/TR/cooluris

[301] Extensible Markup Language (XML) 1.0 (Fifth Edition)
W3C Recommendation 26 November 2008
http://www.w3.org/TR/xml

[302] Namespaces in XML 1.0 (Third Edition)
W3C Recommendation 8 December 2009
http://www.w3.org/TR/xml-names

[303] XML Schema Part 0: Primer Second Edition
W3C Recommendation 28 October 2004
http://www.w3.org/TR/xmlschema-0

6.4 – Description Logics

[400] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider
The Description Logic Hand Book: Theory, Implementation and Application
Cambridge University Press (2002)
http://www.cambridge.org/us/catalogue/catalogue.asp?isbn=9780511060632

[401] Marvin Minsky
"A Framework for Representing Knowledge"
McGraw-Hill, New York (1975)

[402] M. Ross Quillian
"Word Concepts: A Theory and Simulation of some Basic Semantic Capabilities"
Behavioral Science 12: 410-430 (1967)

6.5 – Linguaggi per ontologie

[500] RDF Primer
W3C Recommendation 10 February 2004
http://www.w3.org/TR/rdf-primer

[501] Resource Description Framework (RDF): Concepts and Abstract Syntax
W3C Recommendation 10 February 2004
http://www.w3.org/TR/rdf-concepts

[502] RDF Vocabulary Description Language 1.0: RDF Schema
W3C Recommendation 10 February 2004
http://www.w3.org/TR/rdf-schema

[503] OWL Web Ontology Language Overview
W3C Recommendation 10 February 2004
http://www.w3.org/TR/owl-features

[504] OWL Web Ontology Language Guide
W3C Recommendation 10 February 2004
http://www.w3.org/TR/owl-guide

[505] OWL Web Ontology Language Reference
W3C Recommendation 10 February 2004
http://www.w3.org/TR/owl-ref

[506] OWL Web Ontology Language: Semantics and Abstract Syntax
Section 2. Abstract Syntax
http://www.w3.org/TR/owl-semantics/syntax.html

[507] OWL 2 Web Ontology Language Primer
W3C Recommendation 27 October 2009
http://www.w3.org/TR/owl2-primer

[508] Manchester OWL Syntax
http://www.w3.org/TR/owl2-manchester-syntax

6.6 – Parser generator, project management

[600] Parsing
http://en.wikipedia.org/wiki/Parsing

[601] Dick Grune, Ceriel J.H. Jacobs
Parsing Techniques - A Practical Guide (1990)
Vrije Universiteit Amsterdam, The Netherlands

[602] Noam Chomsky
Three models for the description of languages
IRE Transactions on Information Theory (1956)

[603] Noam Chomsky
On certain formal properties of grammars
Information and Control (1959)

[604] Parser generators
http://en.wikipedia.org/wiki/Comparison_of_parser_generators

[605] Context-free grammar
http://en.wikipedia.org/wiki/Context-free_grammar

[606] Left-to-right Leftmost-derivation parser
http://en.wikipedia.org/wiki/LL_parser

[607] Left recursion
http://en.wikipedia.org/wiki/Left_recursion

[608] Removing Left Recursion in ANTLR
http://www.antlr.org/wiki/display/ANTLR3/Left-Recursion+Removal

[609] LL(*) parser genetator
http://www.antlr.org

[610] Wiki per documentazione ANTLR
http://www.antlr.org/wiki

[611] Terence Parr
The Definitive ANTLR Reference
The Pragmatic Bookshelf (2007)
http://www.pragprog.com/titles/tpantlr/the-definitive-antlr-reference

[612] ANTLR Interest – Mailing list
antlr-interest@antlr.org
http://www.antlr.org/pipermail/antlr-interest/

[613] Java Runtime Exception
http://java.sun.com/javase/6/docs/api/java/lang/RuntimeException.html

[614] The ANTLR GUI Development Environment
http://www.antlr.org/works/index.html

[615] Bug fix alla libreria JavaScript
http://code.google.com/p/antlr-javascript/issues/detail?id=17#c4
http://fisheye2.atlassian.com/changelog/antlr/?cs=6744

[616] Software project management and comprehension tool
http://maven.apache.org

[617] ANTLR V3 plugin for Maven
http://mojo.codehaus.org/antlr3-maven-plugin