Work in Progress ....

A Lexical Functional Grammar for RDF.

Motivation

In "Refactoring RDF/XML Syntax" clarity as to how XML documents are related to the grammar expressions has been given by the XML Infoset production rules.

It is also necessary to provide clarity as to which triples are generated.

It is generally better to provide declarative specifications rather than procedural specifications. Hence it seems appropriate to investigate whether there are declarative grammar technologies that are sufficiently powerful to demonstrate which triples are generated.

In computational linguistics there is a long tradition of using declarative grammars which have both a surface structure (corresponding to the well known parse tree of context free grammars) and a deep structure, which is a homomorphic image of the surface structure typically clarifying some basic linguistic operations such as relative clauses.

Much of RDF is similar to relative clauses in natural language. The subject of the triple appears from the context rather than explicitly.

Since, once upon a time, I was an expert in writing unification grammars (particularly Lexical Functional Grammars), I thought I would have a crack at doing one for RDF.

Lexical-Functional Grammar basics

A Lexical-Functional Grammar consists of a set of grammar rules, and a lexicon. Our treatment omits the lexicon.

Each grammar rule is an EBNF rule annotated with f-structure rules.

A complete analysis of an input string consists of a standard parse tree according to the EBNF with the input string as the concatenation of its leaves, and an "f-structure" (the deeper level of linguistic analysis) associated with each node in the parse tree.

The f-structure of the analysis is the f-structure corresponding to the top of the parse tree (the start symbol of the grammar).

The f-structure is a rooted directed acyclic functional graph with labelled edges. The vertices of the graph are either unlabelled or are atomic values. In the latter case, the vertex has no edges leaving it. The graph is functional in the sense that for any vertex and any edge label there is at most one edge with the given label leaving the given vertex. Hence an f-structure graph and a path expression of edge labels defines at most one vertex, found by starting at the root and following the edge with the next label in the path expression.

The annotations on the EBNF rules refer to the f-structures of the analysis. Two variable symbols are used in the annotations: ↑ refers to the f-structure corresponding to the left hand side of the rule, and ↓ refers to the f-structure of the annotated non-terminal symbol in the right hand side of the rule. Note that an LFG rule is hence two dimensional. From left to right we see the symbols of the context free grammar, and under each such symbol we get f-structure rules relating to that symbol. In particular, in say rule 6.1 below, when there is a kleene star or other meta symbol associated with the non-terminal symbol, then that meta symbol also applies to the f-structure rules.

So the first expansion 6.1 indicates that *for each* **description** node in the parse tree in this expansion the value of the *graph* edge of the **description** node's f-structure is a subset of the value of the *graph* edge of the **RDF** node's f-structure.

We note that the f-structure annotations describe the finished analysis, but do not specify how to make it.

LFG references:

Kaplan, Ronald M. and Joan Bresnan. 1982. Lexical-Functional Grammar: A formal system for grammatical representation. In Joan Bresnan, editor, The Mental Representation of Grammatical Relations. The MIT Press, Cambridge, MA, pages 173--281. Reprinted in Dalrymple, Kaplan, Maxwell, and Zaenen, eds., Formal Issues in Lexical-Functional Grammar, 29--130. Stanford: Center for the Study of Language and Information. 1995.

references listed here

http://clwww.essex.ac.uk/LFG/Introductions.html

particularly:

Wescoat, Michael. 1989. `Practical Instructions for Working with the Formalism of Lexical Functional Grammar'. MS, Xerox PARC. Available as a PostScript file

and

Neidle, Carol. 1994. `Lexical-Functional Grammar'. **Encyclopedia of Language and Linguistics**. New York: Pergamon Press. Available as a PostScript file

Also:

http://citeseer.nj.nec.com/ronald89formal.html

http://www.parc.xerox.com/istl/groups/nltt/medley/

http://www.parc.xerox.com/istl/groups/nltt/medley/screenshot.gif

Explanation of some of the approach

The goal is that the complete set of triples is the value of the *graph* edge of the f-structure.

Hence in some of the rules, e.g. 6.3, we assert that a particular triple is a member of a particular set.

We also assert that some sets are subsets of other sets.

The combination of these rules is intended to require all the desired triples to belong to the top-level set (the value of the *graph* edge).

Subject URI/ID

If there is an rdf:about or rdf:ID then this rule specifies the URI for the *subject* (of the *context*), otherwise the subject is left undefined. This undefined subject has to be interpreted as a prince node which is not part of this grammar. There is no genID or equivalent procedure specified.

Context

Each description/typedNode has a *context* edge. The value of this edge is stuff that influences each of the propAttr and propertyElt daughters of the description or typedNode. The context is shared with the daughters with a $\uparrow$ context = $\downarrow$ context rule.

BagID

Every triple generated by a **propertyElt** or a **propAttr** production is reified. However, its *reificationQuad* is not automatically included in the top level *graph*, but only in an intermediate set, the *reificationBag* which is a subset of the top level *graph* if the optional **bagIdAttr** non-terminal is present.

The *reificationQuad* is also a subset of the toplevel *graph* if an **idAttr** is present in production 6.12.

Fundamental Limitations

Being a declarative approach I do not believe this can address the rdf:li to rdf:_NNN rewrite issue.

I do not think it will possible to do rdf:aboutEach in this way.

Prolog Mapping

It is possible to map this into prolog. Executing it requires a small special purpose grammar compiler. It would parse rather strange XML strings with very little whitespace, and no use of xmlns and xml: attributes. In such a mapping the first rule might be:

rdf(R) --> "<rdf:RDF>", descriptionStar(R), "<rdf:RDF>".

descriptionStar(R) --> description(D), D:graph<R:graph, descriptionStar(R).

descriptionStar(R) --> [].

Here are the intended symbols, I hope they come out right on your browser.

Symbol	In words
↑	up arrow
↓	down arrow
⊂	Subset
∈	Set membership
→	left-to-right arrow

The Rules (incomplete)

6.1	**RDF**	→	"<*rdf*:RDF>"	**description*** ↓ graph ⊂ ↑ graph	"</*rdf*:RDF>"			
	RDF	→	**description** ↓ graph ⊂ ↑ graph					
6.3	**description**	→	"<*rdf*:Description "	**idAboutAttr**? ↑ context subject = ↓	**bagIdAttr**? ↑ context reificationBag ⊂ ↑ graph ↑ context bagSubject = ↓ triple(↓ , *rdf*:type, *rdf*:Bag) ∈ ↑ context reificationBag	**propAttr*** ↓ triple ∈ ↑ graph ↑ context = ↓ context	"/>"	
	description	→	"<*rdf*:Description "	**idAboutAttr**? ↑ context subject = ↓	**bagIdAttr**? ↑ context reificationBag ⊂ ↑ graph ↑ context bagSubject = ↓ t(↓ , *rdf*:type, *rdf*:Bag) ∈ ↑ context reificationBag	**propAttr*** ↓ triple ∈ ↑ graph ↑ context = ↓ context	">" **propertyElt*** ↓ triple ∈ ↑ graph ↓ graph ⊂ ↑ graph ↑ context = ↓ context	"<*rdf*:Description/>"
	description	→	**typedNode** ↑ = ↓					

6.5 **idAboutAttr** → **idAttr**
$\uparrow = \downarrow$

idAboutAttr → **aboutAttr**
$\uparrow = \downarrow$

idAboutAttr → **aboutEachAttr**
Not implemented

6.6 **idAttr** → " *rdf*:ID="' **IDsymbol** "'"
$\uparrow$ uri = "#" + $\downarrow$ name

6.7 **aboutAttr** → " *rdf*:about="' **URI-reference** "'"
$\uparrow = \downarrow$

6.9 **bagIdAttr** → " *rdf*:bagID="' **IDsymbol** "'"
$\uparrow$ uri = "#" + $\downarrow$ name

6.10 **propAttr** → **typeAttr**
$\uparrow = \downarrow$

propAttr →

propName	"=" **string**	"'"	
$\uparrow$ triple predicate = $\downarrow$	$\uparrow$ triple object = $\downarrow$ string	$\uparrow$ triple subject = $\uparrow$ context subject	t($\uparrow$ reification , *rdf*:type, *rdf*:Statement) $\in$ $\uparrow$ reificationQuad
t($\uparrow$ reification , *rdf*:predicate, $\downarrow$) $\in$ $\uparrow$ reificationQuad	t($\uparrow$ reification , *rdf*:object, $\downarrow$ string) $\in$ $\uparrow$ reificationQuad	$\uparrow$ reificationQuad $\subset$ $\uparrow$ context reificationBag	t($\uparrow$ reification , *rdf*:subject, $\uparrow$ context subject) $\in$ $\uparrow$ reificationQuad

6.11 **typeAttr** →

" *rdf*:type="'	"=" **URI-reference**	"'"	
$\uparrow$ triple predicate = *rdf*:type	$\uparrow$ triple object = $\downarrow$	$\uparrow$ triple subject = $\uparrow$ context subject	t($\uparrow$ reification , *rdf*:type, *rdf*:Statement) $\in$ $\uparrow$ reificationQuad
t($\uparrow$ reification , *rdf*:predicate, *rdf*:type) $\in$ $\uparrow$ reificationQuad	t($\uparrow$ reification , *rdf*:object, $\downarrow$) $\in$ $\uparrow$ reificationQuad	$\uparrow$ reificationQuad $\subset$ $\uparrow$ context reificationBag	t($\uparrow$ reification , *rdf*:subject, $\uparrow$ context subject) $\in$ $\uparrow$ reificationQuad

6.12 **propertyElt** →

"<" **propName**	**idAttr**?	">" **value**	"</" propName	">"
$\uparrow$ triple predicate = $\downarrow$	$\uparrow$ reification = $\downarrow$	$\uparrow$ triple object = $\downarrow$	$\uparrow$ triple subject = $\uparrow$ context subject	t($\uparrow$ reification , *rdf*:type, *rdf*:Statement) $\in$ $\uparrow$ reificationQuad
t($\uparrow$ reification , *rdf*:predicate, $\downarrow$) $\in$ $\uparrow$ reificationQuad	$\uparrow$ reificationQuad $\subset$ $\uparrow$ graph	t($\uparrow$ reification , *rdf*:object, $\downarrow$) $\in$ $\uparrow$ reificationQuad	$\uparrow$ reificationQuad $\subset$ $\uparrow$ context reificationBag	t($\uparrow$ reification , *rdf*:subject, $\uparrow$ context subject) $\in$ $\uparrow$ reificationQuad
		$\downarrow$ graph $\subset$ $\uparrow$ graph		