Title: # Converting SPARQL syntax to trees

Author: Fred Zemke
Date: September 28, 2006

## References

[SPARQL CR] "SPARQL query language for RDF", Candidate Recommendation, http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/

[rq24] "SPARQL query language for RDF", Editor's draft, http://www.w3.org/2001/sw/DataAccess/rq23/24
(the version I printed was dated 2006/06/12)

[First attempt] Fred Zemke, "An attempt at a formal semantics for SPARQL"

[Constructive/destructive]Fred Zemke, "SPARQL semantics: constructive or destructive?"

[Constructive mapping semantics]Fred Zemke, "Constructive mapping semantics for SPARQL"

[SQL/Framework] Jim Melton (ed), "ISO International Standard (IS) Database Language SQL - Part 1: SQL/Framework", ISO/IEC 9075-2:2003

## 1. Introduction

This paper continues the work begun in [Constructive mapping semantics]. That paper proposed a formal semantics for SPARQL starting from a tree representation of a SPARQL query. This paper proposes a technique for converting SPARQL syntax to such a tree. However, because my understanding of SPARQL continues to evolve, the trees proposed here are not precisely the same as the ones presumed in [Constructive mapping semantics].

## 2. Terminology

It will be convenient to have some terminology to talk about character strings that match BNF. I cite the following definitions from [SQL/Framework] Subclause 6.3.3.1 "Syntactic containment":

> Let <A>, <B>, and <C> be syntactic elements; let *A1*, *B1*, and *C1* respectively be instances of <A>, <B>, and <C>.
>
> In a Format, <A> is said to *immediately contain* <B> if <B> appears on the right-hand side of the BNF production rule for <A>. An <A> is said to *contain* or *specify* <C> if <A> immediately contains <C> or if <A> immediately contains a <B> that contains <C>.

> In SQL language, *A1* is said to *immediately contain B1* if <A> immediately contains <B> and *B1* is part of the text of *A1*. *A1* is said to *contain* or *specify C1* if *A1* immediately contains *C1* or if *A1* immediately contains *B1* and *B1* contains *C1*. If *A1* contains *C1*, then *C1* is *contained in A1* and *C1* is *specified by A1*.
>
> *A1* is said to contain *B1 with an intervening* <C> if *A1* contains *B1* and *A1* contains an instance of <C> that contains *B1*. *A1* is said to contain *B1 without an intervening* <C> if *A1* contains *B1* and *A1* does not contain an instance of <C> that contains *B1*.
>
> *A1 simply contains B1* if *A1* contains *B1* without an intervening instance of <A> or an intervening instance of <B>.

Adapting these definitions of "contain", "immediately contain" and "simply contain" to SPARQL BNF and language is straightforward. Usage in the SQL standard has evolved away from the word "specify", so I will use "contain" exclusively. Also, experience with the SQL standard shows that rules written using immediate containment are fragile, because rearranging the BNF can break immediate containment. In contrast, simple containment has been found to be rather robust and immune to changes in the BNF. Consequently, when a rule might be written with either immediate or simple containment, I prefer simple containment. However, cases arise in which only immediate containment is sufficiently precise.

# 3. Transformation of syntax to trees

To transform a SPARQL query to a tree, I proceed by recursion on substrings that match BNF nonterminals. If Q is a query string, let Tree(Q) denote the tree representation of Q.

## 3.1 Removing abbreviations

A query is processed by first removing the following abbreviated syntax:

1. Abbreviations for prefixes, as explained in [rq24] 3.1.1 "Syntax for IRIs".
2. Abbreviations for blank nodes, as explained in [rq24] 3.1.4 "Syntax for blank nodes". As a result, the only strings matching rule [66] BlankNode actually match rule [70] *BLANK_NODE_LABEL*.
3. Abbreviations for triple patterns, as explained in [rq24] 3.2 "Syntax for triple patterns". As a result, predicate-object lists, object lists, and RDF collections are expanded fully.
4. The keyword "a" is replaced by the full IRI, as explained in [rq24] 3.2.4 "rdf:type".

## 3.2 WhereClause

Let WC be a character string conforming to rule [13] WhereClause:

```
[13] WhereClause ::= 'WHERE'? GroupGraphPattern
```

Let GGP be the GroupGraphPattern simply contained in WC. Then Tree(WC) = Tree(GGP), i.e., the tree representation of WC is the same as the tree representation of GGP.

## 3.3 GroupGraphPattern, GraphPattern, and OptionalGraphPattern

GroupGraphPattern is defined by rule [19]:

```
[19] GroupGraphPattern ::= '{' GraphPattern '}'
```

### 3.3.1 Issues related to GroupGraphPattern

When defining the tree of a GroupGraphPattern, I believe there are two open issues that must be addressed:

1. Lee Feigenbaum raised the question, what is the scope of FILTER. The thread began with http://lists.w3.org/Archives/Public/public-rdf-dawg/2006JulSep/0186.html . I personally prefer the answer posed in the last paragraph of http://lists.w3.org/Archives/Public/public-rdf-dawg/2006JulSep/0228.html , that the scope of a FILTER is the GroupGraphPattern delimited by the nearest curly braces containing the FILTER. This paper implements that proposal.

2. First operand of OPTIONAL. This issue was first raised in my first paper [First attempt], which was attached to http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0170.html .

   Andy Seaborne in http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0175.html proposed that the best way to identify the first operand of OPTIONAL is to rearrange the grammar, so that rule [19] becomes

   ```
   [19'] GroupGraphPattern ::=
        '{' GraphPattern
            ( ( OptionalGraphPattern ('.')? )+ GraphPattern )?
        '}'
   ```

   and rule [23] becomes

   ```
   [23'] GraphPatternNotTriples ::=
        GroupOrUnionGraphPattern | GraphGraphPattern
   ```

   I initially embraced this solution; however, I now believe that it is not equivalent to the current grammar. For example, it will not recognize this query:

   ```
   WHERE { ?x :y :z OPTIONAL { ?x :u :v }
           ?x :y :z OPTIONAL { ?x :r :s } }
   ```

   Therefore the best way to identify the first operand of an OPTIONAL appears to be in http://lists.w3.org/Archives/Public/public-rdf-dawg/2006AprJun/0174.html . The proposal in this paper implements that algorithm.

### 3.3.2 Stage 1 analysis of GroupGraphPattern

GroupGraphPattern references rule [20] GraphPattern:

```
[20] GraphPattern ::= FilteredBasicGraphPattern
     ( GraphPatternNotTriples '.'? GraphPattern )?
```

This rule involves a recursion on the right, which can be replaced by an equivalent iteration:

```
[20'] GraphPattern ::= FilteredBasicGraphPattern
     ( GraphPatternNotTriples '.'?
       FilteredBasicGraphPattern )*
```

Let GGP be a GroupGraphPattern, and let GP be the GraphPattern simply contained in GGP. Using rule [20'] to parse GP, let n be the number of iterations of the final "star" quantifier; then n is the number of GraphPatternNotTriples simply contained in GP, and n+1 is the number of FilteredBasicGraphPatterns simply contained in GP. Let $NT_1$, . . ., $NT_n$ be the GraphPatternNotTriples, and let $FBGP_1$, . . . , $FBGP_{n+1}$ be the FilteredBasicGraphPatterns simply contained in GP. Thus GGP is

$$'\{' \; FBGP_1 \; NT_1 \; FBGP_2 \; . \; . \; . \; NT_n \; FBGP_{n+1} \; '\}'$$

### 3.3.3 Stage 2: pulling out FILTERs

FILTERs are embedded within FilteredBasicGraphPatterns. As explained under issue #1 above, I think that the scope of a FILTER should be a GroupGraphPattern. Consequently it will be necessary when building the tree for a GroupGraphPattern to reach down into the FilteredBasicGraphPatterns and pull out the FILTERs. FilteredBasicGraphPattern is defined by rule [21]:

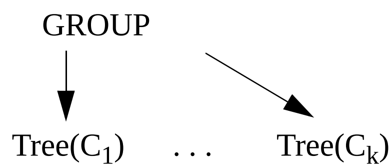
```
[21] FilteredBasicGraphPattern ::= BlockOfTriples?
     ( Constraint '.' FilteredBasicGraphPattern )?
```

This rule also has recursion on the right, which can be replaced by an equivalent iteration:

```
[21'] FilteredBasicGraphPattern ::= BlockOfTriples?
     ( Constraint '.' BlockOfTriples )*
```

Let k be the number of Constraints simply contained in GP. Let $C_1$, . . ., $C_k$ be the Constraints simply contained in GP. (This approach is simpler than resorting to double subscripts to drill down to Constraints within FilteredBasicGraphPatterns.)

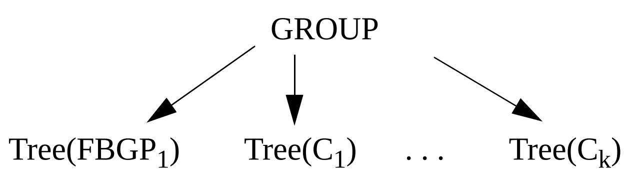
To begin the construction of Tree(GGP), we can collect all the Constraints as children of a GROUP node, like this:




Each FilteredBasicGraphPattern $FBGP_i$ can be processed as explained later to produce a tree Tree($FBGP_i$). These trees do not contain the trees for any FILTERs, which have conceptually been pulled out and promoted to be immediate children of the GROUP node, as shown above.

### 3.3.4 Stage 3: handling concatenated OPTIONALs

If n = 0 then the final tree for GGP is Tree(GGP) shown below:

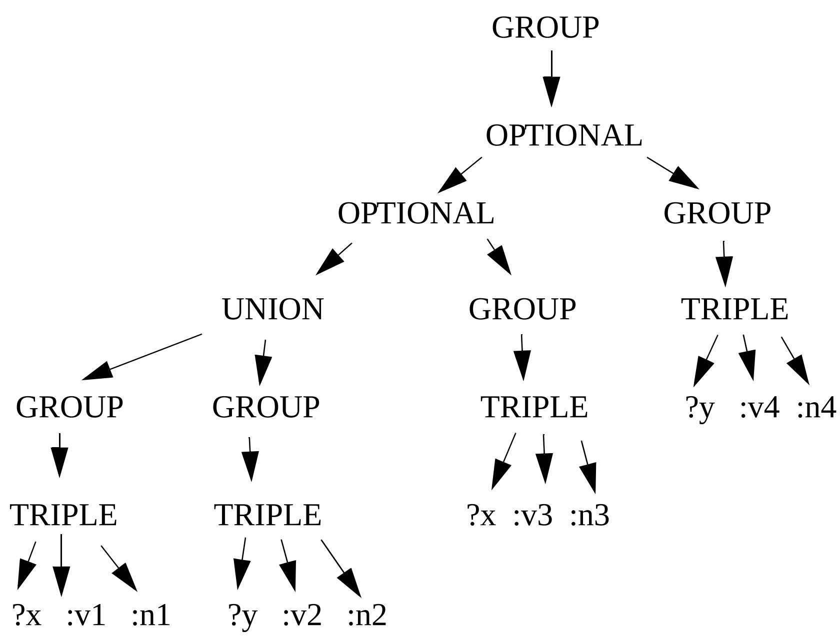



If n > 0, then it is possible that one or more of $NT_i$ is an OptionalGraphPattern; thus we encounter the need to address issue #2 above. The solution is rather messy because it is possible to transform into deep trees. For example

```
{ { ?x :v1 :n1 } UNION { ?y :v2 :n2 }
  OPTIONAL { ?x :v3 :n3 } OPTIONAL { ?y :v4 :n4 } }
```

**Converting SPARQL syntax to trees**

The objective is that this example should transform to this tree:




In this example,

$n = 3$,

$FBGP_1$ is empty,

$NT_1$ is { ?x :v1 :n1} UNION { ?y :v2 :n2 },

$FBGP_2$ is empty,

$NT_2$ is OPTIONAL { ?x :v3 :n3 },

$FBGP_3$ is empty,

$NT_3$ is OPTIONAL { ?y :v4 :n4 },

$FBGP_4$ is empty.

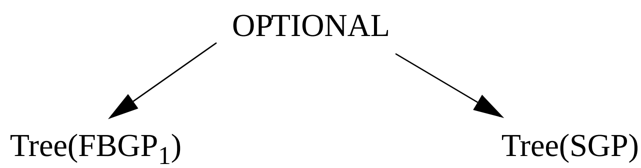
The following algorithm constructs a sequence of trees, CHILDREN. The algorithm proceeds by iterating on variable i, starting with i = 1 and running through i = n. The algorithm operates by appending a tree to CHILDREN, or by deleting the last element of CHILDREN. Initially CHILDREN consists of a single tree, CHILDREN = { Tree ($FBGP_1$) }. At the conclusion of the algorithm, CHILDREN will consist of the desired immediate children of the GROUP node corresponding to GGP.

For each i between 1 and n, perform the following rules:

1) If $NT_i$ is a GroupOrUnionGraphPattern or a GraphGraphPattern, then append Tree($NT_i$) to CHILDREN.

2) If $NT_i$ is an OptionalGraphPattern, then let SGP be the GroupGraphPattern simply contained in $NT_i$. (SGP for "supplementary graph pattern", my proposed term for the second operand of an OPTIONAL.)

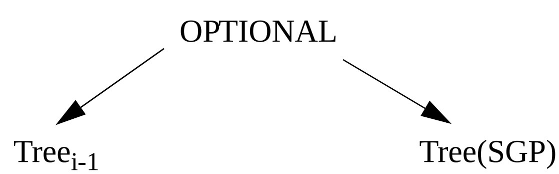
   a) If i = 1, then let $Tree_i$ be the following tree:




   Remove Tree($FBGP_1$), which is the only element of CHILDREN, from CHILDREN, and insert $Tree_1$.

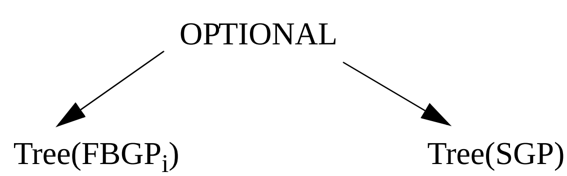
   b) If i > 1, then:

      i) If the last element of CHILDREN is EMPTY (this element is Tree($FBGP_i$)), then the penultimate element of CHILDREN is $Tree_{i-1}$. Construct the following tree:




      Remove the last two elements from CHILDREN (i.e., $Tree_{i-1}$ and the EMPTY that is Tree($FBGP_{i-1}$)) and insert $Tree_i$.

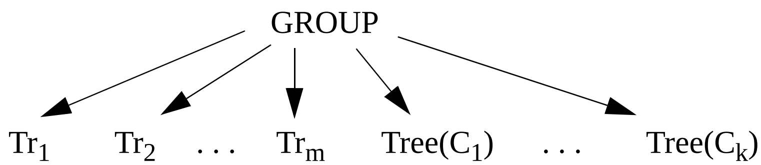
      ii) Otherwise, construct the following tree:




      Remove the last element from CHILDREN (i.e., Tree($FBGP_i$) and insert $Tree_i$.

3) Append Tree ($FBGP_{i+1}$) to CHILDREN.

After performing the preceding rules for all i between 1 and n, CHILDREN contains the desired immediate children of the GROUP node. Let CHILDREN = $\{Tr_1, \ldots, Tr_m\}$. Then Tree(GGP) is the following tree:




## 3.4 FilteredBasicGraphPattern

Let FBGP be a character string conforming to rule [21] FilteredBasicGraphPattern:

```
[21] FilteredBasicGraphPattern ::= BlockOfTriples?
     ( Constraint '.' FilteredBasicGraphPattern )?
```

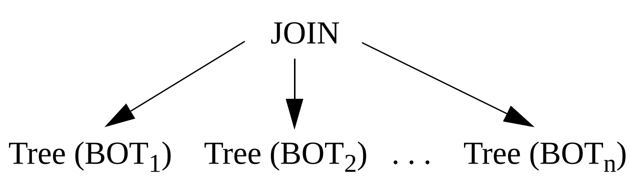
As explained above, we can remove the recursion by using this rule instead:

```
[21’] FilteredBasicGraphPattern ::= BlockOfTriples?
     ( Constraint '.' BlockOfTriples )*
```

All Constraints have already been handled by pulling them up to the GroupGraphPattern. What remains is zero or more BlockOfTriples. Let n be the number of BlockOfTriples simply contained in FBGP.

If n = 0, then Tree(FBGP) consists of a single node, labeled EMPTY.

Otherwise let the BlockOfTriples be $BOT_1, \ldots, BOT_n$. Then Tree(FBGP) is




## 3.5 BlockOfTriples

Let BOT be a string conforming to rule [22] BlockOfTriples

```
[22] BlockOfTriples ::= TriplesSameSubject
     ( '.' TriplesSameSubject? )*
```

Because all abbreviations have been removed (see Section 3.1 above), BOT also conforms to these rules:

```
[22a] BasicGraphPattern ::=
     TriplePattern ( '.' TriplePattern )*
```

```
[22b] TriplePattern ::= Subject Verb Object
```

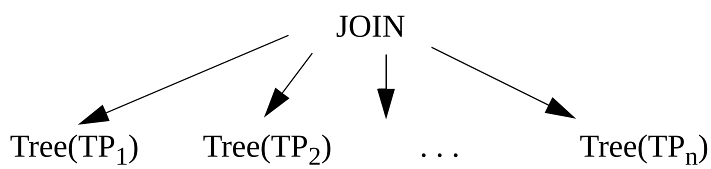
```
[22c] Subject ::= VarOrTerm
```

```
[36] Verb ::= VarOrIRIref | 'a'
```

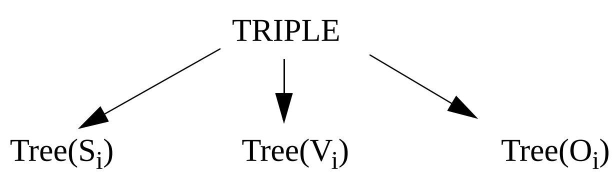
*[NOTE to the proposal reader: actually, the case ‘a’ cannot arise once abbreviations are removed, so we could replace rule [36] with something simpler.]*

```
[22d] Object ::= VarOrTerm
```

Let BOT consist of n TriplePattern’s, $TP_1, \ldots, TP_n$. Tree(BOT) is shown below:




For all i, let the Subject, Verb and Object simply contained in $TP_i$ be $S_i$, $V_i$ and $O_i$, respectively. Tree($TP_i$) is shown below:




## 3.6 Constraint

Constraint is defined by rule [27]:

```
[27] Constraint ::=
     'FILTER' ( BrackettedExpression | BuiltInCall
     | FunctionCall )
```

Let C be a Constraint. Let E be the simply contained BracketedExpression, BuiltInCall or FunctionCall. The tree Tree(C) is shown below:

$$\begin{tikzcd} \text{FILTER} \arrow[d] \\ \text{Tree(E)} \end{tikzcd}$$

### 3.6.1 BracketedExpression, Expression and ConditionalOrExpression

BracketedExpression is defined by rule [56]:

```
[56] BracketedExpression ::= '(' Expression ')'
```

The tree of a BracketedExpression is the same as the tree of the immediately contained Expression.

Expression is defined by rule [46]:

```
[46] Expression ::= ConditionalOrExpression
```

The tree of an Expression is the same as the tree of the immediately contained ConditionalOrExpression.

ConditionalOrExpression is defined by rule [47]:

```
[47] ConditionalOrExpression ::=
     ConditionalAndExpression
     ( '||' ConditionalAndExpression )*
```

Let COE be a CondtionalOrExpression. Let n be the number of simply contained ConditionalAndExpressions. Let these ConditionalAndExpressions be $COA_1, \ldots, COA_n$. If n = 1, then Tree(COE) = Tree($COA_1$). If n > 1, then the tree Tree(COE) is shown below:

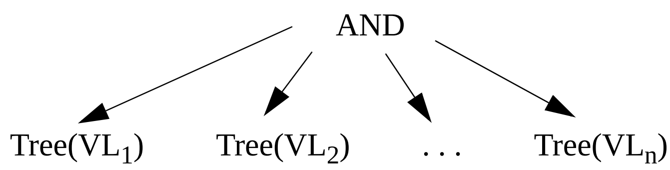
$$\begin{tikzcd} & & \text{OR} \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & \\ \text{Tree}(COA_1) & \text{Tree}(COA_2) & \ldots & \text{Tree}(COA_n) \end{tikzcd}$$

### 3.6.2 ConditionalAndExpression

ConditionalAndExpression is defined by rule [48]:

```
[48] ConditionalAndExpression ::=
     ValueLogical ( '&&' ValueLogical )*
```

Let COA be a CondtionalAndExpression. Let n be the number of simply contained ValueLogicals. Let these ValueLogicals be $VL_1, \ldots, VL_n$. If n = 1, then Tree(COA) = Tree($VL_1$). If n > 1, then the tree Tree(COA) is shown below:




### 3.6.3 ValueLogical and RelationalExpression

ValueLogical is defined by rule [49]:

```
[49] ValueLogical ::= RelationalExpression
```

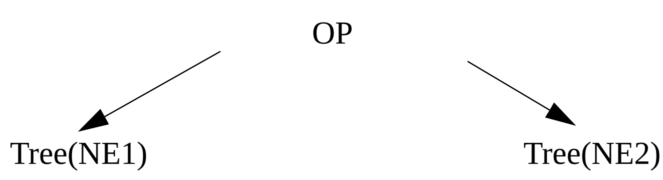
The tree of a ValueLogical is the same as the tree of the simply contained RelationalExpression.

RelationalExpression is defined by rule [50]:

```
[50] RelationalExpression   ::=
     NumericExpression
     (   '=' NumericExpression
       | '!=' NumericExpression
       | '<' NumericExpression
       | '>' NumericExpression
       | '<=' NumericExpression
       | '>=' NumericExpression )?
```

There are two cases:

1. If RelationalExpression immediately contains only one NumericExpression, then the tree of the RelationalExpression is the same as the tree of the NumericExpression.
2. If RelationalExpression immediately contains two NumericExpressions, then let them be NE1 and NE2, and let OP be the operator ('=', '!=', '<', '>', '<=', or '>=') immediately contained in the RelationalExpression. The tree of RelationalExpression is shown below:




Note that OP is not a keyword in the above tree; it is a symbol denoting one of the relational operator symbols '=', etc.

#### 3.6.4 Etc. for the rest of the expression syntax

The main point in continuing to flesh this out would be to reach function invocations, including especially BOUND, because there are semantic issues about the treatment of unbound variables in FILTER. An unbound variable in an invocation of BOUND is not an error. Andy Seaborne proposed that whether an unbound variable in an invocation of other functions is an error or not should be left to the function definition (I have not looked up the email message). Otherwise an unbound variable is an error.

## 3.7 GraphPatternNotTriples

GraphPatternNotTriples is defined by rule [23]:

```
[23] GraphPatternNotTriples ::= OptionalGraphPattern
     | GroupOrUnionGraphPattern | GraphGraphPattern
```

OptionalGraphPattern has already been handled with GroupGraphPattern. Otherwise, the tree representation of a GraphPatternNotTriples is the same as the tree representation of the simply contained GroupOrUnionGraphPattern or GraphGraphPattern.

## 3.8 GroupOrUnionGraphPattern

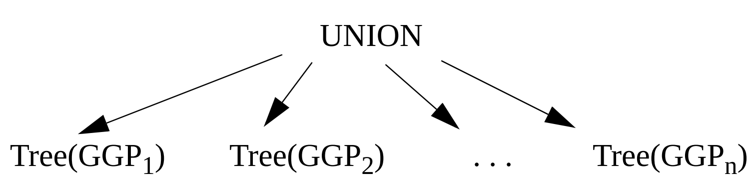
Let GOUGP be a character string conforming to rule [26] GroupOrUnionGraphPattern:

```
[26] GroupOrUnionGraphPattern ::= GroupGraphPattern
     ( 'UNION' GroupGraphPattern )*
```

There are two cases:

1. GOUGP does not immediately contain UNION. In that case the tree representation of GOUGP is the same as the tree representation of the only immediately contained GroupGraphPattern.

2. GOUGP immediately contains UNION. Let n be the number of GroupGraphPattern's immediately contained in GOUGP; let $GGP_1$, . . . , $GGP_n$ be these GroupGraphPattern's. Tree(GOUGP) is shown below:



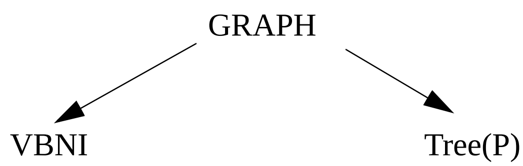

## 3.9 GraphGraphPattern

Let GGP be a character string conforming to rule [25] GraphGraphPattern:

**Converting SPARQL syntax to trees**

```
[25] GraphGraphPattern ::=
     'GRAPH' VarOrBlankNodeOrIRIref GroupGraphPattern
```

Let VBNI be the VarOrBlankNodeOrIRIref, and let P be the GroupGraphPattern.  Tree(GGP) is shown below:




***- End of paper -***