A Semantic Web Language Primer

Dr. A. Joseph Rockmore
jrockmore@asec-usa.com | 650/759-5399

This Primer describes how the Semantic Web languages defined by the World Wide Web Consortium (W3C) are used in expressing data and the meaning of that data. It is not meant to be a comprehensive explanation of the languages' syntax and semantics, nor a complete story of how the Semantic Web came to be and what it does. The former can be found on the W3C web site (start at http://www.w3.org/standards/semanticweb/), and there are several good books available for the latter.

RDF

The Resource Description Framework (RDF) language is used to express data about resources, where "resources" can be interpreted to be anything (a web page, a person, an idea, etc.). The basic building block is the *triple*, consisting of *subject*, *predicate*, *object*[1]. The *subject* is a URI, the *predicate* is some property that is defined for the type (class) of the subject, and the *object* is either a typed literal or the URI of some other subject.

Let's look at a couple of assertions that express data about resources[2]:

```
bb:YogiBerra rdf:type bio:Person  .
bb:YogiBerra bb:playsPosition bb:Catcher  .
bb:YogiBerra bb:careerHomeRuns 358 .
```

The first assertion says that a "resource," Yogi Berra, whose URI is defined in the `bb` namespace[3], is of type Person (where the meaning of Person is defined in the `bio` namespace), he played the catcher position (where the meaning of playsPosition and Catcher are defined in the `bb` namespace), and he had 358 career home runs (where the meaning of careerHomeRuns is defined in the `bb` namespace). The meaning of "type" is part of RDF, and thus is defined in the `rdf` namespace. This is what data looks like in RDF[4]: triples expressed as a subject, a predicate, and an object, separated by spaces, and concluded with a period[5].

In order to be able to interpret the meaning (semantics) of assertions such as these, it is necessary to define what the assertions are talking about – the meaning of the types (classes) and predicates. We already saw an example of asserting the type of a subject using:

```
rdf:type
```

---

[1] Others have used *object*, *property*, *value* or other similar terminology. Herein we follow W3C terms.
[2] Assertions about instances are indicated in italics herein.
[3] The examples herein presume a namespace, `bb`, that is used for data about baseball.
[4] The particular syntax used is explained below, along with alternates.
[5] We adopt the usual practice of naming classes and instances in upper camel case and properties in lower camel case.

when we asserted that Yogi Berra is a Person.

A predicate can be asserted by defining that its type is a property, using:

```
rdf:Property.
```

For example:

```
bb:playsPosition rdf:type rdf:Property  .
bb:careerHomeRuns rdf:type rdf:Property  .
```

Once defined, these assertions will provide the meaning to statements about instances, such as those above about Yogi Berra.

Notice the commonality of defining the semantics of the data by triples, and asserting instance data using triples. This is a particular strength of the W3C Semantic Web language family.

Reification is the process of making an assertion about an assertion. RDF supports reification, but the assertion about which the other assertion is made needs to be given a URI that refers to it. Explicit reification uses three properties, `rdf:subject`, `rdf:predicate`, and `rdf:object`, to define the three elements of a triple, and then assigns a URI to the triple[6]. For example,

```
news:12345 rdf:subject bb:WhiteyFord .
news:12345 rdf:predicate bb:playsPosition .
news:12345 rdf:object bb:Pitcher  .

news:NewYorkTimes news:reports news:12345 .
```

The last assertion says that the New York Times reported that Whitey Ford plays the pitcher position, which is different from asserting that Whitey Ford plays the pitcher position. Reification is an important concept, since assertions often need to be made as to the source, date, confidence, etc. of assertions in the domain of discourse. This type of data is often called pedigree or lineage or provenance data.

Syntaxes. There are three common syntaxes used for serialization of RDF triples.

N-Triples uses fully unabbreviated URI's for the subject, predicate, and object, followed by a period:

```
<http://www.baseball.org#NYYankees>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.baseball.org#Team> .
```

RDF/XML is an XML serialization that integrates best with HTML and the rest of the web infrastructure, defining namespaces and then using the `rdf:about` construct for the subject, followed by predicate tags containing the objects, such as:

---

[6] There is also a quad (vice triple) representation that is more efficient than explicit reified statements, but this is complex (it is easy to make statements that are not intended) so is not addressed herein.

```
<rdf:RDF
xmlns:bb="http://www.baseball.org#"
xmlns:rdf=" http://www.w3.org/1999/02/22-rdf-syntx-ns#"
xmlns:geo="http:// www.columbiagazetteer.org/cities#">
      <bb:Team rdf:about
      "http://www.baseball.org#NYYankees">
      <bb:yearFounded 1901 />
      <bb:playInCity rdf:resource=
         "http://www.columbiagazetteer.org/cities#NewYork"/>
      <bb:hasPlayer rdf:resource=
         "http://www.baseball.org#YogiBerra"/>
      </bb:Team>
</rdf:RDF>
```

Turtle[7] is a more compact serialization that defines namespaces (the prefix of a resource, which corresponds to a fully qualified URI) and then uses the prefix followed by a colon concatenated with the subject, predicate, and object, followed by a period:

```
@prefix bb: <http://www.baseball.org#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntx-ns#>
bb:NYYankees rdf:type bb:Team .
```

In Turtle, "a" can be used instead of `rdf:type`. With this usage, a triple is more "readable" as normal English:

```
bb:NYYankees a bb:Team .
```

Turtle allows several triples to be about the same subject (where the predicates and objects are delimited by semicolons), or about the same subject and predicate (where the objects are delimited by commas), respectively:

```
bb:NYYankees rdf:type bb:Team ;
      bb:yearFounded 1901 ;
      bb:playInCity geo:NewYork .

bb:NYYankees bb:hasPlayer bb:YogiBerra ,
      bb:JoeDiMaggio , bb:PhilRizzuto  .
```

Note that indenting and separate lines are for readability only, and have no meaning to an RDF parser.

RDF Schema (RDFS)

The Resource Description Framework Schema (RDFS) language is used to assert additional meaning (semantics) of data expressed in RDF. A fundamental construct is the class of a subject:

```
bb:Team rdf:type rdfs:Class .
```

[7] Another common syntax is N3, which is a superset of Turtle.

```
bb:Player rdf:type rdfs:Class .
bb:Infielder rdf:type rdfs:Class .
bb:Outfielder rdf:type rdfs:Class .
```

A class can be defined to be the subclass of another class. Subclasses (child classes) inherit all the properties of their superclasses (parent classes). So if we have defined a property on a class, such as `bb:playsPosition` defined on `bb:Player`, and then we assert subclasses:

```
bb:Infielder rdfs:subClassOf bb:Player .
bb:Outfielder rdfs:subClassOf bb:Player .
```

We can infer that `Infielders` and `Outfielders` have a property called `playsPosition`.

Every property is defined for instances of some class, and has as its values instances of some (possibly the same) class, or has as its values typed literals. The way to express these definitions is using the domain and range properties:

```
bb:playsPosition rdfs:domain bb:Player ;
      rdfs:range bb:Position .

bb:yearFounded rdfs:domain bb:Team ;
      rdfs:range xsd:gYear  .
```

The way to remember domain and range is that one can state in English, roughly, "domain property range," so that the example above would be "a Player playsPosition a Position" and a particular instance would be "Yogi Berra (a Player) playsPosition Catcher (a Position)." If the range is a typed literal, the allowable types are the XML Schema data types, including Booleans, binary data types, text types (strings, URI's, etc.), number types (integers, positive integers, floats, etc.), and date time types (dateTime, duration, month, etc.).

In a similar manner as with classes, properties can be defined to be subproperties of other properties:

```
bb:regularlyPlaysPosition rdfs:subPropertyOf
      bb:playsPosition  .
bb:occasionallyPlaysPosition rdfs:subPropertyOf
      bb:playsPosition  .
```

The usefulness of defining subproperties will become clearer when we discuss properties of properties in the OWL section below.

A few constructs are defined in RDFS with no semantics, to help people (not computers) in understanding. A printable or displayable name for a resource can be asserted using `rdfs:label`:

```
bb:YogiBerra rdfs:label "Lawrence Peter 'Yogi' Berra"  .
```

A resource can be referenced using `rdfs:seeAlso` and `rdfs:isDefinedBy` (which is a subproperty of `rdfs:seeAlso`). An arbitrary comment can be made using `rdfs:comment`.

Web Ontology Language (OWL)[8]

OWL defines certain properties of properties that allow inferences to be made about data. This is a very important part of the Semantic Web approach; we don't need to state additional information explicitly for every instance of a class if it can be defined as a property on the class and the additional information inferred.

An important reason why subproperties are included in OWL is so that the properties of the superproperty are inherited by the subproperty, and need not be separately asserted.

Properties can be asserted to be datatype properties, which have as their values an XML data type, or can be asserted to be object properties, which have as their values an instance of some class:

```
bb:yearFounded rdf:type owl:DataTypeProperty  .
bb:playsForTeam rdf:type owl:ObjectProperty  .
```

Common practice is to use these two OWL constructs instead of the `rdf:Property` construct, since they assert more information about the property.

Two properties can be inverses of one another:

```
bb:playsForTeam rdf:type rdf:Property ;
      rdfs:domain bb:Player; rdfs:range bb:Team .
bb:hasPlayer rdf:type rdf:Property  ;
      owl:inverseOf bb:playsForTeam  .
```

so if we assert:

```
bb:YogiBerra bb:playsForTeam bb:Yankees  .
```

then we can infer, without having had to explicitly state:

```
bb:Yankees bb:hasPlayer bb:YogiBerra  .
```

A property can be symmetric:

```
bb:teammateOf rdf:type owl:SymmetricProperty  .
```

so if we assert:

```
bb:YogiBerra bb:teammateOf bb:PhilRizzuto  .
```

then we can infer:

[8] Not all OWL constructs are described in this Primer.

```
bb:PhilRizzuto bb:teammateOf bb:YogiBerra  .
```

Properties can be transitive:

```
bb:teammateOf rdf:type owl:TransitiveProperty  .
```

so if we assert:

```
bb:YogiBerra bb:teammateOf bb:PhilRizzuto  .

bb:PhilRizzuto bb:teammateOf bb:JoeDiMaggio  .
```

then we can infer:

```
bb:YogiBerra bb:teammateOf bb:JoeDiMaggio  .
```

Properties can be functional, meaning that each instance has at most one value:

```
bb:hasCareerBattingAve rdf:type owl:FunctionalProperty  .

bb:YogiBerra bb:hasCareerBattingAve .285  .
```

Yogi cannot have two or more different career batting averages; he has only one.

Properties also can be inverse functional, meaning each value has only one instance:

```
bb:holdsRecord rdf:type owl:InverseFunctionalProperty  .

bb:TyCobb bb:hasCareerBattingAve .366  ;
          bb:holdsRecord "CareerBattingAverage" .
```

`bb:hasCareerBattingAve` is not inverse functional, since more than one player could have a career batting average of 0.285, like Yogi, but `bb:holdsRecord` is inverse functional, since only one player has the highest career batting average.

Properties can also have cardinality restrictions, asserting the minimum, maximum, or exact number of instances allowed:

```
bb:inningsPlayedInGame owl:minCardinality 5  .

bb:hasDefensivePlayers owl:cardinality 9  .

bb:hasOffensivePlayers owl:maxCardinality 4  .
```

OWL provides capabilities to make assertions about the equivalence, or lack thereof, between classes, properties, and instances. Two classes can be asserted to be the same:

```
bb:Player owl:equivalentClass sports:Pro  .
```

or they can be asserted to be different:

```
bb:Player owl:disjointWith bb:Manager  .
```

Two properties can be asserted to be the same:

```
bb:playsForTeam owl:equivalentProperty sports:memberOf  .
```

or they can be asserted to be different:

```
bb:playsForTeam owl:propertyDisjointWith bb:managesTeam  .
```

Two instances can be asserted to be the same:

```
bb:YogiBerra owl:sameAs sports:LawrencePBerra  .
```

or they can be asserted to be different:

```
bb:YogiBerra owl:differentFrom bb:DaleBerra  .
```

OWL has its own class construct, `owl:Class`. It is similar in concept to `rdfs:Class`, but whereas classes in RDFS can only be built through the subclass relationship, in OWL classes also can be built via enumeration, union, intersection, complement, etc.:

```
bb:League rdf:type owl:Class ;
      owl:oneOf (bb:AmericanLeague bb:NationalLeague)  .

bb:Infielder rdf:type owl:Class ;
      owl:unionOf (bb:FirstBaseMan
      bb:SecondBaseMan bb:ThirdBaseMan bb:ShortStop)  .

bb:TripleCrownWinner rdf:type owl:Class ;
      owl:intersectionOf
      (bb:BattingAveLeader bb:HomeRunLeader bb:RBILeader ) .
```

SPARQL

The query language for the Semantic Web is SPARQL[9]. Two forms of query are defined: the SELECT form and the CONSTRUCT form.

The basic SELECT form query in SPARQL is the *triple pattern*, which is a triple with zero (or more) of subject, predicate, or object as an unbound variable. Executing the query will return any triples that match the triple pattern, with all variables bound. Examples of SPARQL queries with the subject, predicate, and object as an unbound variable, respectively, are:

```
?team bb:hasPlayer bb:YogiBerra .
NYYankees ?r bb:YogiBerra .
NYYankees bb:hasPlayer ?player .
```

A triple store with a SPARQL query engine will return, respectively, all teams that Yogi Berra played for, all relationships Yogi had with the New York Yankees, and all players who played for the New York Yankees.

[9] SPARQL is a recursive acronym that stands for SPARQL Protocol and RDF Query Language.

Much more interesting and useful are SPARQL queries that are *graph patterns*. This query expresses a set of triple patterns, where any variable that appears in two or more triple patterns must bind to the same resource. The set of triple patterns is enclosed in braces. An example is:

```
{?team bb:hasPlayer ?player.
?player bb:playsPosition bb:Catcher .
?player bb:teammateOf bb:PhilRizzuto .}
```

This query will return any team/player graph where the player played on that team and played catcher, and was a teammate of Phil Rizzuto. So the graph with *bb:NYYankees* connected by the *bb:playsPosition* predicate to *bb:YogiBerra* is one set of bindings that meet the query criteria, but so is *bb:NYYankees* and *bb:BillDrescher.*

The basic CONSTRUCT form query in SPARQL is two (or more) graph patterns that produce new graphs, where any variable that appears in the graph patterns must bind to the same resource. For example:

```
CONSTRUCT (?s bb:hasPlayer ?o)
WHERE {?s bb:playInCity geo:NewYork .
       ?s bb:hasPlayer ?o .
       ?o rdf:playsPosition bb:Catcher .}
```

will return all team/player graphs where the team plays in New York (the Yankees and Mets) and has a player who is a catcher, including the graph connecting *bb:NYYankees* to *bb:YogiBerra*.

Another example that uses reification is:

```
CONSTRUCT (?s bb:hasPlayer ?o)
WHERE {?r rdf:subject ?s .
       ?r rdf:predicate bb:hasCatcher .
       ?r rdf:object ?o .
       ?r bb:year 1948 . }
```

which will return all team/catcher graphs connected by the `hasCatcher` predicate that were asserted to be true in 1948, including the graph connecting the New York Yankees to Yogi Berra.