Parsing OWL

An OWL-RDF parser takes an RDF-XML file and attempts to construct an OWL ontology that corresponds to the triples represented in the RDF. This page describes a basic strategy that could be used in such a parser. Note that this is not intended as a complete specification, but hopefully provides enough information to point the way towards how one would build a parser that will deal with a majority of (valid) OWL ontologies.

For example, we do not discuss the implementation or handling of `owl:imports` here, nor do we address in depth issues concerned with spotting some of the more obscure violations of the DL/Lite rules.

OWL in RDF

The OWL Semantics and Abstract Syntax (**S&AS**) document provides a characterisation of OWL ontologies in terms of an abstract syntax. This is a high level description of the way in which we can define the characteristics of classes and properties.

In addition, S&AS gives a mapping to RDF triples. This tells us how such an abstract description of an OWL ontology can be transformed to a collection of RDF triples (which can then be represented in a concrete fashion using, for example RDF-XML).

In order to parse an OWL-RDF file into some structure closer to the abstract syntax we need to reverse this mapping, i.e. determine what the class and property definitions were that lead to those particular triples. Note that this reverse mapping is not necessarily unique. For example, the following two ontology fragments:

```
Class( a )
Class( b )
SubClassOf( b a )
```

and

```
Class( a )
Class( b partial a )
```

both give rise to the same collection of triples under the mapping:

```
a rdf:type owl:Class
b rdf:type owl:Class
b rdfs:subClassOf a
```

For many purposes, e.g. species validation, this is not necessarily a problem. For other situations, e.g. where an editing tool is being used, we would at least expect a parser to be consistent in the strategy it employed to produce abstract syntax descriptions.

An arbitrary RDF graph may not necessarily correspond to an OWL Lite or DL ontology. In other words, there may not be an OWL Lite or DL ontology which when transformed using the mapping produces the given graph. This is what a species validator attempts to determine: if such an ontology exists. A parser (as described here) will go one step further and actually attempt to construct such an ontology.

Errors

There are, in general, two ways in which an RDF graph may fail to correspond to an OWL [Lite|DL] ontology.

1. There does not exist an OWL ontology in abstract syntax form that maps to the given triples.
2. There is an ontology in abstract syntax form that maps to the triples, but the ontology violates some of the restrictions for membership of the OWL [Lite|OWL] subspecies.

We might (loosely) describe the first as *external* errors, and the second as *internal* errors. Examples of *external* errors include:

- Using a URI reference in an `owl:Class` context (e.g. as the object of an `owl:someValuesFrom` property whose subject is an `owl:Restriction` which has an `owl:onProperty` property with an `owl:ObjectProperty` as its object) without explicitly including a statement that the URI reference is an `owl:Class` or `owl:Restriction`. The AS&S requires that all such usages are given an explicit typing.
- Using a malformed `owl:Restriction`, e.g. missing an `owl:onProperty` property.
- Using the wrong vocabulary, e.g. `rdf:Property` instead of the more specific `owl:ObjectProperty` and `owl:DatatypeProperty`.
- Violation of rules concerning structure sharing (see **below**).

Once we have an ontology in abstract form, we can then check for internal errors. For example, there are restrictions on the expressiveness that can be used in OWL Lite (no unions or enumerations and limited cardinality restrictions). The Lite and DL subspecies also have a constraint that effectively says that the collections of URI references of classes, individuals and properties must be disjoint. Thus in OWL Lite and DL we can not use metamodelling devices such as *classes as instances*.

The procedure described below is targeted primarily at parsing OWL DL ontologies. For example, whenever `rdfs:subPropertyOf` is used, OWL DL requires that the subject and object of the triple have corresponding types (e.g. both are either `owl:ObjectProperty` or `owl:DatatypeProperty`). If this is not the case, the parser will raise an error. An OWL Full parser should allow this (but it is not necessarily clear what the corresponding abstract syntax for such a construct would be).

Parser Implementation

The following discussion assumes that we have some implementation of a data structure representing the ontology which is close to the abstract syntax description (something along the lines of our proposed **OWL API**). We do not discuss the details of such an implementation here — hopefully the meaning of actions such as *add a class x* or *set the functional flag on a property* will be clear.

Streaming vs. non-streaming

Many XML parsers operate in a *streaming* fashion — elements are reported to the parser as they are encountered during the parse, and the file is processed incrementally. It is difficult to do this when parsing RDF models (or at least when performing a task such as producing an abstract syntax representation of an OWL ontology from a given RDF-XML file). The problem is that we have no guarantee of the order in which the triples in the graph are processed (and thus reported by the streaming parser). A particular syntactic construct may actually be split across several locations in the RDF file. In order to parse in a streaming fashion, we may have to make note of triples encountered earlier on and then come back to process them later. As a concrete example of this, consider a situation where an `owl:AnnotationPRoperty` is used to make an annotation about a particular individual:

```
AnnotationProperty( hasName )
Individual( fred hasName "Frederick" )
```

This results in the triples:

```
[1] hasName rdf:type owl:AnnotationProperty
[2] fred hasName "Frederick"
```

If we encounter `[1]` before `[2]` during the parse, we know that the property is an annotation property, and can thus process `[2]` as an annotation. If, however, we encounter `[2]` first, we do not know whether to process `[2]` as an annotation or a value on the individual. As there is no way of knowing whether or not `[1]` will occur until we have seen all the triples, we must wait until we have seen all triples before processing `[2]`.

Because of this, our strategy is that the parser does not attempt to process *anything* until all triples are available. Although it may be possible to process some information in a streaming manner, it reduces the conceptual complexity of the parser if we first collect the triples then process them. Note that this has ramifications on the resources that will be required when parsing — when parsing large RDF graphs, large amounts of memory may be needed.

If we are interested in detecting OWL DL ontologies, there are some things that *can* be done during the collection of triples — for example any node with `rdf:type owl:Restriction` must be a bnode. Thus if we encounter a triple:

```
x rdf:type owl:Restriction
```

where `x` is not a bnode, the triples cannot be the result of a transformation of an OWL Lite or DL ontology.

We assume that while parsing we have access to the objects in the ontology already created, e.g.. if an ObjectProperty `p` has been introduced we can get access to it. When we refer to, for example, *the ObjectProperty* `p`, we mean the ObjectProperty that has been defined with name `p`.

In addition, we assume that we can query the RDF graph to determine the presence or absence of particular arcs (e.g. precisely the kind of functionality provided by an RDF API such as **Jena**).

Using Triples

While processing the graph, we keep a record of any triples that have been *used* in the translation. For example, if there is a triple:

```
x rdf:type owl:Class
```

which results in the introduction of a class `x`.

```
Class( c )
```

then we consider that triple to have been *used*.

Named Objects

We first identify the name classes and properties that make up the ontology.

Classes

For any **non-bnode** `x` in the graph s.t. there is a triple:

```
x rdf:type owl:Class
```

introduce a new class `x`.

```
Class( c )
```

We will refer to any such classes that have been introduced in this manner as *named classes*.

Properties

Properties should all be introduced with an explicit type.

ObjectProperty

For any node `p` in the graph where there is one of the following triples:

```
p rdf:type owl:ObjectProperty
p rdf:type owl:TransitiveProperty
p rdf:type owl:InverseFunctionalProperty
p rdf:type owl:SymmetricProperty
```

introduce a new ObjectProperty `p`.

```
ObjectProperty( p )
```

In addition, if any of the latter three triples are present, the appropriate flag should be set on the property, e.g.:

```
ObjectProperty( p Transitive )
```

If there is also a triple of the form:

```
p rdf:type FunctionalProperty
```

then the property should be set as functional.

For any object property `p` dealt with as above, there may also be an (optional) triple:

```
p rdf:type rdf:Property
```

DatatypeProperty

For any node `p` in the graph where there is a triple:

```
q rdf:type owl:DataProperty
```

introduce a new DatatypeProperty `q`:

```
DatatypeProperty( q )
```

If there is also a triple of the form:

```
q rdf:type FunctionalProperty
```

then the property should be set as functional.

For any data property `p` dealt with as above, there may also be an (optional) triple:

```
p rdf:type rdf:Property
```

AnnotationProperty

For any node `a` in the graph where there is a triple:

```
a rdf:type owl:AnnotationProperty
```

introduce a new AnnotationProperty `a`.

```
AnnotationProperty( a )
```

For any annotation property `p` dealt with as above, there may also be an (optional) triple:

```
p rdf:type rdf:Property
```

Datatypes

For any node `d` in the graph where there is a triple:

```
d rdf:type rdfs:Datatype
```

introduce a new Datatype `d`.

```
Datatype( d )
```

There may also be an (optional) triple:

```
d rdf:type rdfs:Class
```

Axioms

Now that the named classes and properties have been identified, we can determine the axioms that have been asserted.

Property Axioms

Property axioms assert characteristics of properties.

Domain

For any triples of the form:

```
p rdfs:domain d
```

translate `d` to a class description, and add the resulting class description to the domains of the property `p`. If `p` is not a property, raise an error.

Range

For any triples of the form:

```
p rdfs:range r
```

if `p` is an ObjectProperty, then **translate** `r` to a class description, and add the resulting class description to the ranges of the property `p`. If `p` is a data property, convert `r` to a **data range** and add the result to the ranges of the property.

subProperty & equivalentProperty

For any triples of the form:

```
p rdfs:subPropertyOf q
```

or

```
p owl:equivalentProperty q
```

first check that either:

1. p and q are ObjectProperties;
   or
2. p and q are DatatypeProperties.

If so, add an axiom asserting that q is a superproperty or equivalent property of p as appropriate. If neither of the above are true, raise an error.

inverseOf

For any triples of the form:

```
p owl:inverseOf q
```

Check that p and q are ObjectProperties. If not, raise an error. If so, add q to the collection of inverses of p.

Class Definitions

We have to deal with any class definitions that occur in the ontology. For example, the following RDF fragment:

```
<class rdf:about="#a">
  <intersectionOf rdf:parseType="Collection">
    <class rdf:about="#b"/>
    <class rdf:about="#c"/>
  </intersectionOf>
</class>
```

arises when a class a has been given a complete definition involving an intersection.

For any named class x, do the following.

- For all triples:

  ```
  x owl:oneOf l
  ```

  l should be a node representing a **list** of individuals. Add the axiom:

  ```
  Class( x complete oneOf(i1 i2...in) )
  ```

  where i_1 i_2 ... i_n are the individuals in the list l. If l is not a list (of individuals), raise an error.

- For all triples:

  ```
  x owl:intersectionOf l
  ```

`l` should be a node representing a **list** of class descriptions. Add the axiom:

```
Class( x complete lt1 lt2...ltn )
```

where lt_1 lt_2 ... lt_n are the translated descriptions in the list `l`. If `l` is not a list (of class descriptions), raise an error.

- For all triples:

```
x owl:unionOf l
```

`l` should be a node representing a **list** of class descriptions. Add the axiom:

```
Class( x complete unionOf(lt1 lt2...ltn) )
```

where lt_1 lt_2 ... lt_n are the translated descriptions in the list `l`. If `l` is not a list (of class descriptions), raise an error.

- For all triples:

```
x owl:complementOf n
```

`n` should be a node representing a class description. Add the axiom:

```
Class( x complete complementOf( nt ) )
```

where `nt` is the translation of `n`. If `nt` is not a class description, raise an error.

Class Axioms

Class axioms can provide relationships and characteristics of arbitrary class descriptions.

SubClass

For all triples of the form:

```
c rdfs:subClassOf d
```

add a new axiom:

```
SubClassOf( ct dt )
```

where `ct` is the translation of `c` to a class description, and `dt` the translation of `d`. If `c` is a named class, then due to the **ambiguity** of the reverse mapping, an alternative here is to include the assertion as part of the definition of the class and add the axiom:

```
Class( c partial dt )
```

to the ontology. Note that in this case, if the class already has a partial description

in the ontology, e.g. there is an axiom:

```
Class( c partial e1 e2...en )
```

then we can simply add `dt` to this axiom to get:

```
Class( c partial e1 e2...en dt )
```

rather than introducing a new axiom.

EquivalentClass

See **below**.

DisjointClass

See **below**.

Individual Axioms

Individual axioms assert relationships about the equality and inequality of individuals.

Same

For all triples of the form:

```
x owl:sameAs y
```

where `x` and `y` are individualIDs, add individuals `x` and `y` (if necessary) and an axiom:

```
SameIndividual( x y )
```

Different

For all triples of the form:

```
x owl:differentFrom y
```

where `x` and `y` are individualIDs, add individuals `x` and `y` (if necessary) and an axiom:

```
DifferentIndividuals( x y )
```

AllDifferent

For all triples of the form:

```
x rdf:type owl:AllDifferent
```

where `x` is a bnode, there should also be a triple:

```
x owl:distinctMembers l
```

where `l` is a **list**. Add an axiom:

```
DifferentIndividuals( i1 i2...in )
```

where i_1 i_2 ... i_n are the individuals in the list `l`. If `l` is not a list (of individuals), `x` is not a bnode or the `owl:distinctMembers` triple is missing, raise an error.

Translating Lists

Lists are used in a number of places in OWL ontologies: for example to represent the arguments of boolean expressions or the individuals listed in an enumeration (one-of). For the purposes of producing a OWL ontology, order is not particularly important — the order of the operands in an intersection or union does not alter their semantics, so for simplicitly, we consider converting a node representing a list to a set of nodes. Lists are thus handled using the following simple recursive procedure.

To convert a node `l` s.t. there is a triple:

```
l rdf:type rdf:Nil
```

simply return the empty set.

For a node `l` s.t. there is a triple:

```
l rdf:type rdf:List
```

find the node `r` s.t. there is a triple:

```
l rdf:rest r
```

If such a node does not exist, or there are are multiple nodes which are the objects of such triples, raise an error. The node `r` should be a list node itself. Convert this node to a set of nodes `rs`. Now find the node s.t. there is a triple:

```
l rdf:first f
```

Again, there should be a single such node — if not, raise an error. Return the result of adding this node to the set `rs`.

For cases where we expect a list of class descriptions, we do the obvious thing, e.g. convert to a collection of nodes, then **translate** each node using the procedure described below.

For any node `l` which is used as a list (e.g. as the subject of a `rdf:first` or `rdf:rest`, the object of a `rdf:rest`, or in a place where a list is expected, there may be an (optional) triple:

```
l rdf:type rdf:List
```

If a list is circular, raise an error.

Translating Class Decriptions

If a node is used in particular contexts (e.g. as the subject or object of an `owl:subClassOf` triple) then we know that the node is intended to represent a class expression. In order to handle this, we define a procedure which takes a node in the RDF graph and yields a class expression.

If `n` is a named class, then return `n`.

If this is not the case, `n` must be the subject of the subject of an `rdf:type` triple with object `owl:Restriction` or be the subject of exactly one triple involving `owl:oneOf`, `owl:intersectionOf`, `owl:unionOf`, `owl:complementOf`. If not, raise an error.

The node may also be the subject of triple:

```
n rdf:type owl:Class
```

or

```
n rdf:type rdfs:Class
```

Translation then proceeds on a case-analysis of the particular triple found.

- If there is a triple:

  ```
  n rdf:type owl:Restriction
  ```

  then `n` needs to be translated as a restriction. There should be now be exactly one triple:

  ```
  n owl:onProperty p
  ```

  where `p` is an ObjectProperty or DatatypeProperty. If not, raise an error. In addition, `n` should be the subject of exactly one triple involving `owl:minCardinality`, `owl:maxCardinality` or `owl:cardinality`, `owl:someValuesFrom`, `owl:allValuesFrom` or `owl:hasValue`. If not, raise an error. Translation then again proceeds on a case-analysis of the type of the property and the triple it is involved in.

  - `n owl:cardinality k`

    Return a cardinality restriction whose numerical value is the non negative integer which should be the object of the cardinality triple, e.g.:

    ```
    restriction( p cardinality( k ) )
    ```

  - `n owl:minCardinality k`

    Return a cardinality restriction whose numerical value is the non

negative integer which should be the object of the cardinality triple, e.g.:

```
restriction( p minCardinality( k ) )
```

- `n owl:maxCardinality k`

  Return a cardinality restriction whose numerical value is the non negative integer which should be the object of the cardinality triple, e.g.:

  ```
  restriction( p maxCardinality( k ) )
  ```

- `n owl:someValuesFrom v`

  If `p` is an ObjectProperty, return:

  ```
  restriction( p someValuesFrom ( vt ) )
  ```

  where `vt` is the translation of v to a class description. If `p` is a DatatypeProperty, then return:

  ```
  restriction( p someValuesFrom ( vdt ) )
  ```

  where `vdt` is the translation of v to a **data range**.

- `n owl:allValuesFrom v`

  If `p` is an ObjectProperty, return:

  ```
  restriction( p allValuesFrom ( vt ) )
  ```

  where `vt` is the translation of v to a class description. If `p` is a DatatypeProperty, then return:

  ```
  restriction( p allValuesFrom ( vdt ) )
  ```

  where `vdt` is the translation of v to a **data range**.

- `n owl:hasValue v`

  If `p` is an ObjectProperty, return:

  ```
  restriction( p value ( v ) )
  ```

  where `v` is translation of v as an individual. If `p` is a DatatypeProperty, then return:

  ```
  restriction( p value ( vdt ) )
  ```

  where `vdt` is the translation of v as a data value.

- If there is a triple:

```
n owl:oneOf l
```

then `l` should be a **list** of individuals. Return:

```
oneOf(i1 i2...in)
```

where i_1 i_2 ... i_n are the individuals in the list `l`. If `l` is not a list, raise an error.

- If there is a triple:

```
n owl:intersectionOf l
```

return:

```
intersectionOf( lt1 lt2...ltn )
```

where lt_1 lt_2 ... lt_n are the translated descriptions in the list `l`. If `l` is not a list, raise an error.

- If there is a triple:

```
n owl:unionOf l
```

return:

```
unionOf( lt1 lt2...ltn )
```

where lt_1 lt_2 ... lt_n are the translated descriptions in the list `l`. If `l` is not a list, raise an error.

- If there is a triple:

```
n owl:complementOf m
```

return:

```
complementOf( mt )
```

where `mt` is the translation of `m` as a class description.

Translating Data Ranges

If `n` is an XML schema data type, then return that type.

If `n` is a datatype introduced as **above**, then return that datatype.

If there is a triple:

```
n owl:oneOf l
```

then `l` should be a **list** of data values. Return:

```
oneOf(d1 d2...dn)
```

where d_1 d_2 ... d_n are the data values in the list `l`. If `l` is not a list, raise an error.

Structure Sharing

S&AS includes the following comment relating to translation from abstract syntax to RDF graphs:

> *For many directives these transformation rules call for the transformation of components of the directive using other transformation rules. When the transformation of a component is used as the subject, predicate, or object of a triple, even an optional triple, the transformation of the component is part of the production (but only once per production) and the main node of that transformation should be used in the triple.*

In practice, this means that blank nodes (i.e. those with no name) which are produced during the transformation and represent arbitrary expressions in the abstract syntax form should not be "re-used".

Consider the following example:

```
Class(A partial intersectionOf(C D))
```

In this case, translation to an RDF graph would result in a blank node representing the intersection of C and D. This would then be used as the object of a `rdfs:subClassOf` triple with `A` as subject.

Now consider if the ontology also included a second axiom as below.

```
Class(A partial intersectionOf(C D))
Class(B partial intersectionOf(C D))
```

In this case, we are **not** allowed to "re-use" the blank node, but must instead produce a new node to represent the intersection being used in the definition of `B`, even though the expressions are identical.

There are, however, two cases where a blank node corresponding to an expression can be used in more than one place — when the translation results from an `EquivalentClasses` or `DisjointClasses` axiom. These are discussed in more detail below.

In order to check whether an RDF graph corresponds to an OWL [Lite|DL] ontology, we must check that the rules for structure sharing have not been violated. We describe strategies for doing this.

Marking Used Blank Nodes

We keep track of all the blank nodes that have been *used* during the parsing process. Effectively, this means that whenever we see a blank node that occurs as the object of a triple involving `owl:complementOf`, `rdf:type`, `owl:someValuesFrom`, `owl:allValuesFrom` or occuring as a value in a list which is the object of an `owl:intersectionOf` or `owl:unionOf` we first check to see whether the node has been *used*. If so, then structure sharing as occurred and the ontology is **not** in DL. If not, then we mark the node as *used* and carry on. Processing `owl:equivalentClass` and `owl:disjointWith` triples is slightly more complicated as the mapping rules permit us to share structure in particular ways.

EquivalentClass

In general, an equivalence axiom

```
EquivalentClasses( D1 D2...Dn )
```

is translated to a collection of nodes, one for each expression in the equivalence, and a number of `owl:equivalentClass` triples between these nodes such that those triples form a connected graph over the nodes. In other words, starting from any node in the collection, we can get to any other node in the collection along a path that only traverses `owl:equivalentClass` edges in either direction.

In practice, this means that a blank node may participate in more than one `owl:equivalentClass` triple (but note that it cannot also participate in other triples).

A possible strategy for dealing with `owl:equivalentClass` triples is as follows.

1. Collect all `owl:equivalentClass` triples that occur in the graph.
2. Partition the nodes that occur in these triples into sets, where each set consists of connected blank nodes and URI references connected to them, or pairs of URI references: if `n` and `m` are in a set, there is a path between them consisting only of `owl:equivalentClass` edges.
3. For each set of nodes $n_1\ n_2 \ldots n_n$, add an equivalence axiom:

   ```
   EquivalentClasses( tn1tn2...tnn )
   ```

   where tn_i is the translated description of n_i. In addition, if any of the n_i are blank nodes, check that they have not been *used*. If they have, this is **not** an OWL DL ontology. If they are not used, mark as *used*

An improvement to this strategy is to attempt to identify the situations where the `owl:equivalentClass` triple may have come from a class definition (recall the **ambiguity** of the mapping). To address this, if any of the node sets have size 2, and have been produced because of a single triple:

```
c owl:equivalentClass d
```

where `c` is a named class, then we translate the assertion as a definition of the class and add the axiom:

```
Class( c complete dt )
```

to the ontology. In order to correctly parse OWL Lite ontologies, this approach is essential, as it ensures that a situation such as:

```
c owl:equivalentClass _:a
_:a rdf:type owl:Restriction
_:a owl:onProperty p
_:a owl:minCardinality 0
```

is translated to a definition of the class rather than a class axiom (the resulting axiom would **not** be permitted in OWL Lite).

DisjointClass

The rules for `DisjointClasses` axioms tell us that an axiom:

```
DisjointClasses( D1 D2...Dn )
```

is translated to a collection of nodes, one for each expression in the equivalence, and a number of `owl:disjointWith` triples, such that every node in the collection is connected to every other node by *at least* one triple (in either direction). Again, this may lead to blank nodes being used in more than one place.

A possible strategy for dealing with `owl:disjointWith` triples is as follows:

- Collect all `owl:disjointWith` triples that occur in the graph.
- While there are blank nodes in the collection of nodes that we have not already dealt with, do the following:
  - Pick a blank node from the collection of nodes involved in those triples that we haven't already dealt with.
  - Gather together all the nodes n_1 $n_2 \ldots n_n$ that can be reached from n via a path that consists of `owl:disjointWith` triples, and which does not pass **through** a named class node — in other words the traversal stops when we reach a named node. Include n in this collection.
  - In order for the graph to be in OWL DL, the subgraph formed from these nodes considering `owl:disjointWith` edges must be fully connected: every node must have an edge to every other node. If this is not the case, the graph is not in DL.
  - Add a new disjoint axiom:

    ```
    EquivalentClasses( tn1tn2...tnn)
    ```

    where tn_i is the translated description of n_i. In addition, if any of the n_i are blank nodes, check that they have not been *used*. If they have, this is **not** an OWL DL ontology. If they are not used, mark as *used*.
- For any remaining pairs of nodes related by a triple:

  ```
  c owl:disjointWith d
  ```

  if the two nodes have not already been included in a single axiom produced by the process above, a new axiom:

  ```
  DisjointClasses( ct dt )
  ```

where `ct` is the translation of `c` to a class description, and `dt` the translation of `d`.

Tests for Structure Sharing

There are a number of tests in the OWL Test Cases which are designed to illustrate these issues, in particular:

- `owl:disjointWith` tests.
- **OWL DL Syntax** Tests.
- `owl:equivalentClass` tests.

Everything Else

Once all the triples that relate to primitive object definitions and axioms have been processed, (more or less) everything else is assumed to be a fact relating to individuals. For all remaining triples:

```
x p y
```

the action taken depends on the type of `p`. If no explicit type has been given for the property `p`, raise an error.

If `p` is an annotation property, then add an appropriate annotation to the object `x` (which should correspond to a named class, property or individual).

If `p` is an ObjectProperty, assume that the subject and object are individuals and add a fact:

```
Individual( x value( p y ) )
```

If `p` is an DatatypeProperty, assume that the subject is an individual and add a fact:

```
Individual( x value( p dy ) )
```

where `dy` is the translation of `y` to a data type value.

Error Recovery

There are many cases in the above discussion where errors may be raised — for example if properties are used without explicit typing. Strictly speaking, an OWL DL or Lite parser could choose to fail when encountering such situations. Of course, in practice, we might expect parsers to be more resilient and be able to recover. So for example, if the parser detects the following use of a property `p`:

```
x rdf:type owl:Thing
y rdf:type owl:Thing
x p y
```

it is reasonable to assume that the property `p` is intended to be an

`owl:ObjectProperty`. In this case, we might expect the parser to *assume* that `p` is an ObjectProperty and try and proceed with the parse (but would of course warn the user about the assumption being made).

Sean Bechhofer, University of Manchester, 10/09/03.

Last modified 17:16:58 10/09/03