A LaTeX-Style Syntax for OWL 2

Matteo Matassoni
University of Trento – DISI
#150629
<m.matassoni.1@studenti.unitn.it>

Contents

1 Preliminary Definitions

The grammar presented in this document uses the following two "special" terminal symbols, which affect the process of transforming an input sequence of characters into a sequence of regular (i.e., not "special") terminal symbols:

- whitespace is a nonempty sequence of space (U+20), horizontal tab (U+9), line feed (U+A), or carriage return (U+D) characters, and
- a comment is a sequence of characters that starts with the % (U+25) character and does not contain the line feed (U+A) or carriage return (U+D) characters.

2 General Definitions

$\langle$*nonNegativeInteger*$\rangle$ ::= a nonempty finite sequence of digits between 0 and 9

$\langle$*quotedString*$\rangle$ ::= a finite sequence of characters in which ‘’ (U+22) and \ (U+5C) occur only in pairs of the form \‘’ (U+5C, U+22) and \\ (U+5C, U+5C), enclosed in a pair of ‘’ (U+22) characters

$\langle$*languageTag*$\rangle$ ::= @ (U+40) followed a nonempty sequence of characters matching the langtag production from [2]

$\langle$*nodeID*$\rangle$ ::= a finite sequence of characters matching the BLANK_NODE_-LABEL production of [3]

$\langle$*fullIRI*$\rangle$ ::= an IRI as defined in [1], enclosed in a pair of $\langle$ *(U+3C) and* $\rangle$ (U+3E) characters

$\langle$*prefixName*$\rangle$ ::= a finite sequence of characters matching the as PNAME_NS production of [3] and not matching any of the keyword terminals of the syntax

$\langle$*abbreviatedIRI*$\rangle$ ::= a finite sequence of characters matching the PNAME_LN production of [3] and not matching any of the keyword terminals of the syntax

$\langle$*simpleIRI*$\rangle$ ::= a finite sequence of characters matching the PN_LOCAL production of [3] and not matching any of the keyword terminals of the syntax

$\langle$*IRI*$\rangle$:= $\langle$*fullIRI*$\rangle$ | $\langle$*abbreviatedIRI*$\rangle$ | $\langle$*simpleIRI*$\rangle$

3 Including additional input files

Inclusion of additional input files (both local and distributed) allows to organize the ontology in a modular fashion. To specify the input file the following format should be applied:

⟨*includeExternalFile*⟩ ::= '`\input`' '`{`' ⟨*INPUT_FILE*⟩ '`}`'

⟨*INPUT_FILE*⟩ ::= ⟨*URL*⟩ | ⟨*ABSOLUTE_FILEPATH*⟩

Please note that this command may occur anywhere inside the ontology document.

4 Ontologies

⟨*ontologyDocument*⟩ ::= ⟨*defaultPrefixDeclaration*⟩? ⟨*prefixDeclaration*⟩* ⟨*Ontology*⟩

⟨*defaultPrefixDeclaration*⟩ ::= '`\ns`' ⟨*fullIRI*⟩

⟨*prefixDeclaration*⟩ ::= '`\ns`' ⟨*prefixName*⟩ ⟨*fullIRI*⟩

⟨*Ontology*⟩ ::= '`\begin`' '`{`' '`ontology`' '`}`' ⟨*ontologyIRIs*⟩ (⟨*directlyImportsDocument*⟩
| ⟨*ontologyAnnotation*⟩)* ⟨*axioms*⟩ '`\end`' '`{`' '`ontology`' '`}`'

⟨*ontologyIRIs*⟩ ::= ['`[`' ⟨*ontologyIRI*⟩ ['`,`' ⟨*versionIRI*⟩] '`]`']

⟨*ontologyIRI*⟩ ::= ⟨*IRI*⟩

⟨*versionIRI*⟩ ::= ⟨*IRI*⟩

⟨*directlyImportsDocument*⟩ ::= '`\import`' ⟨*IRI*⟩

⟨*ontologyAnnotation*⟩ ::= ⟨*Annotation*⟩

⟨*axioms*⟩ ::= ⟨*Axiom*⟩*

Please note that prefixes: *owl:*, *rdf:*, *rdfs:*, *xml:* and *xsd:* are built-in and should not be explicitly declared.

Example 1.

The following code show an example of an ontology document, written with the proposed syntax.

```
% Default namespace
\ns <http://www.example.com/ontology1#>

\begin{ontology}[<http://www.example.com/ontology1>]
  \import <http://www.example.com/ontology2>
  \a{rdfs:label, "An example"}
  % These are comments and they will be skipped
  % during parsing
  % insert axioms below
  Child \cisa owl:Thing
\end{ontology}
```

5 Annotations

5.1 Annotations of Ontologies, Axioms, and other Annotations

⟨*Annotation*⟩ ::= ‘`\a`’ ‘`{`’ ⟨*AnnotationProperty*⟩ ‘`,`’ ⟨*AnnotationValue*⟩ ‘`}`’ ⟨*annotationAnnotations*⟩

⟨*annotationAnnotations*⟩ ::= [‘`[`’ ⟨*Annotation*⟩ (‘`,`’ ⟨*Annotation*⟩)* ‘`]`’]

⟨*AnnotationValue*⟩ ::= ⟨*AnonymousIndividual*⟩ | ⟨*IRI*⟩ | ⟨*Literal*⟩

5.2 Annotation Axioms

⟨*AnnotationAxiom*⟩ ::= ⟨*AnnotationAssertion*⟩
| ⟨*SubAnnotationPropertyOf*⟩
| ⟨*AnnotationPropertyDomain*⟩
| ⟨*AnnotationPropertyRange*⟩

5.2.1 Annotation Assertion

⟨*AnnotationAssertion*⟩ ::= ⟨*AnnotationSubject*⟩ ‘`\a`’ ‘`{`’ ⟨*AnnotationProperty*⟩ ‘`,`’ ⟨*AnnotationValue*⟩ ‘`}`’ ⟨*axiomAnnotations*⟩

⟨*AnnotationSubject*⟩ ::= ⟨*IRI*⟩ | ⟨*AnonymousIndividual*⟩

Example 2.

The following axiom assigns a human-readable comment to the IRI a:Person.

```
a:Person \a{rdfs:label, "Represents the set of all people
    ."}
```

Since the annotation is assigned to an IRI, it applies to all entities with the given IRI. Thus, if an ontology contains both a class and an individual a:Person, the above comment applies to both entities.

5.2.2 Annotation Subproperties

⟨*SubAnnotationPropertyOf*⟩ ::= ⟨*subAnnotationProperty*⟩ '`\aisa`' ⟨*superAnnotationProperty*⟩ ⟨*axiomAnnotations*⟩

⟨*subAnnotationProperty*⟩ ::= ⟨*AnnotationProperty*⟩

⟨*superAnnotationProperty*⟩ ::= ⟨*AnnotationProperty*⟩

5.2.3 Annotation Property Domain

⟨*AnnotationPropertyDomain*⟩ ::= ⟨*AnnotationProperty*⟩ '`\adomain`' ⟨*IRI*⟩ ⟨*axiomAnnotations*⟩

5.2.4 Annotation Property Range

⟨*AnnotationPropertyRange*⟩ ::= ⟨*AnnotationProperty*⟩ '`\arange`' ⟨*IRI*⟩ ⟨*axiomAnnotations*⟩

6 Entities, Literals, and Anonymous Individuals

6.1 Classes

⟨*Class*⟩ ::= ⟨*IRI*⟩

Example 3.

Classes a:Child and a:Person can be used to represent the set of all children and persons, respectively, in the application domain, and they can be used in an axiom such as the following one:

Each child is a person.

```
a:Child \cisa a:Person
```

6.2 Datatype

⟨*Datatype*⟩ ::= ⟨*IRI*⟩

Example 4.

The datatype xsd:integer denotes the set of all integers. It can be used in axioms such as the following one:

The range of the a:hasAge data property is xsd:integer.

```
a:hasAge \drange xsd:integer
```

6.3 ObjectProperty

⟨*ObjectProperty*⟩ ::= ⟨*IRI*⟩

Example 5.

The object property a:parentOf can be used to represent the parenthood relationship between individuals. It can be used in axioms such as the following one:

Peter is a parent of Chris.

```
a:parentOf(a:Peter, a:Chris)
```

6.4 DataProperty

⟨*DataProperty*⟩ ::= ⟨*IRI*⟩

Example 6.

The data property a:hasName can be used to associate a name with each person. It can be used in axioms such as the following one:

Peter's name is "Peter Griffin".

```
a:hasName(a:Peter, "Peter Griffin")
```

6.5 AnnotationProperty

⟨*AnnotationProperty*⟩ ::= ⟨*IRI*⟩

Example 7.

The comment provided by the following annotation assertion axiom might, for example, be used by an OWL 2 tool to display additional information about the IRI a:Peter.

This axiom provides a comment for the IRI a:Peter.

```
a:Peter \a{rdfs:comment, "The father of the Griffin
    family from Quahog."}
```

6.6 Individual

⟨*Individual*⟩ ::= ⟨*NamedIndividual*⟩ | ⟨*AnonymousIndividual*⟩

6.6.1 Named Individuals

⟨*NamedIndividual*⟩ ::= ⟨*IRI*⟩

Example 8.

The individual a:Peter can be used to represent a particular person. It can be used in axioms such as the following one:

Peter is a person.

```
a:Person(a:Peter)
```

6.6.2 Anonymous Individuals

⟨*NamedIndividual*⟩ ::= ⟨*nodeID*⟩

Example 9.

Anonymous individuals can be used, for example, to represent objects whose identity is of no relevance, such as the address of a person.

Peter lives at some (unknown) address.

```
a:livesAt(a:Peter, _:a1)
```

This unknown address is in the city of Quahog and...

```
a:city(_:a1, a:Quahog)
```

...in the state of Rhode Island.

```
a:state(_:a1, a:RI)
```

6.7 Literals

⟨*Literal*⟩ ::= ⟨*typedLiteral*⟩ | ⟨*stringLiteralNoLanguage*⟩ | ⟨*stringLiteralWithLanguage*⟩

⟨*typedLiteral*⟩ ::= ⟨*lexicalForm*⟩ ‘`[`’ ⟨*Datatype*⟩ ‘`]`’

⟨*lexicalForm*⟩ ::= ⟨*quotedString*⟩

⟨*stringLiteralNoLanguage*⟩ ::= ⟨*quotedString*⟩

⟨*stringLiteralWithLanguage*⟩ ::= ⟨*quotedString*⟩ ‘`[`’ ⟨*languageTag*⟩ ‘`]`’

Example 10.

“1”[xsd:integer] is a literal that represents the integer 1.

Example 11.

“Family Guy” is an abbreviation for “Family Guy@”[rdf:PlainLiteral] – a literal with the lexical form “Family Guy@” and the datatype rdf:PlainLiteral – which denotes a string “Family Guy” without a language tag.

Furthermore, “Padre de familia”[@es] is an abbreviation for the literal “Padre de familia@es”[rdf:PlainLiteral], which denotes a pair consisting of the string “Padre de familia” and the language tag es.

Example 12.

Even though literals “1”[xsd:integer] and “+1”[xsd:integer] are interpreted as the integer 1, these two literals are not structurally equivalent because their lexical forms are not identical. Similarly, “1”[xsd:integer] and “1”[xsd:positiveInteger] are not structurally equivalent because their datatypes are not identical.

6.8 Entity Declarations and Typing

⟨*Declaration*⟩ ::= ⟨*Entity*⟩ ⟨*axiomAnnotations*⟩

⟨*Entity*⟩ ::= ⟨*Class*⟩ ‘`\c`’
| ⟨*Datatype*⟩ ‘`\dt`’
| ⟨*ObjectProperty*⟩ ‘`\o`’
| ⟨*DataProperty*⟩ ‘`\d`’
| ⟨*AnnotationProperty*⟩ ‘`\a`’
| ⟨*NamedIndividual*⟩ ‘`\i`’

Example 13.

The following axioms state that the IRI a:Person is used as a class and that the IRI a:Peter is used as an individual.

```
a:Person \c
```

```
a:Peter \i
```

7 Property Expressions

7.1 Object Property Expressions

⟨*ObjectPropertyExpression*⟩ ::= ⟨*ObjectProperty*⟩
  | '(' ⟨*InverseObjectProperty*⟩ ')'

7.1.1 Inverse Object Properties

⟨*InverseObjectProperty*⟩ ::= '`\oinvof`' ⟨*ObjectProperty*⟩

Example 14.

Consider the ontology consisting of the following assertion.

Peter is Stewie's father.

```
a:fatherOf(a:Peter, a:Stewie)
```

This ontology entails that a:Stewie is connected by the following object property expression to a:Peter:

```
(\oinvof a:fatherOf)
```

7.2 Data Property Expressions

⟨*DataPropertyExpression*⟩ ::= ⟨*DataProperty*⟩

8 Data Ranges

⟨*DataRange*⟩ ::= ⟨*Datatype*⟩
  | ⟨*AtomicDataRange*⟩
  | '(' ⟨*NonAtomicDataRange*⟩ ')'

8.1 Atomic Data Ranges

⟨*AtomicDataRange*⟩ ::= ⟨*SequenceDataIntersectionOf*⟩
  | ⟨*SequenceDataUnionOf*⟩
  | ⟨*DataOneOf*⟩

8.1.1 Sequence Intersection of Data Ranges

⟨*SequenceDataIntersectionOf*⟩ ::= '`\drandof`' '`{`' ⟨*DataRange*⟩ ('`,`' ⟨*DataRange*⟩)+ '`}`'

Example 15.

Example 18 can be rewritten with the sequence-style notation:

```
\drandof{xsd:nonNegativeInteger, xsd:nonPositiveInteger}
```

8.1.2 Sequence Union of Data Ranges

⟨*SequenceDataUnionOf*⟩ ::= '`\drorof`' '`{`' ⟨*DataRange*⟩ ('`,`' ⟨*DataRange*⟩)+ '`}`'

Example 16.

Example 19 can be rewritten with the sequence-style notation:

```
\drorof{xsd:string, xsd:integer}
```

8.1.3 Enumeration of Literals

⟨*DataOneOf*⟩ ::= '`\droneof`' '`{`' ⟨*Literal*⟩ ('`,`' ⟨*Literal*⟩)* '`}`'

Example 17.

The following data range contains exactly two literals: the string "Peter" and the integer one.

```
\droneof{"Peter", "1"[xsd:integer]}
```

8.2 Non-atomic Data Ranges

⟨*NonAtomicDataRange*⟩ ::= ⟨*DataIntersectionOf*⟩
| ⟨*DataUnionOf*⟩
| ⟨*DataComplementOf*⟩
| ⟨*DatatypeRestriction*⟩

8.2.1 Intersection of Data Ranges

⟨*DataIntersectionOf*⟩ ::= ⟨*DataRange*⟩ '`\drand`' ⟨*DataRange*⟩

Example 18.

The following data range contains exactly the integer 0:

```
(xsd:nonNegativeInteger \drand xsd:nonPositiveInteger)
```

8.2.2 Union of Data Ranges

⟨*DataUnionOf*⟩ ::= ⟨*DataRange*⟩ '`\dror`' ⟨*DataRange*⟩

Example 19.

The following data range contains all strings and all integers:

```
(xsd:string \dror xsd:integer)
```

8.2.3 Complement of Data Ranges

⟨*DataComplementOf*⟩ ::= '`\drnot`' ⟨*DataRange*⟩

Example 20.

The following complement data range contains literals that are not positive integers:

```
(\drnot xsd:positiveInteger)
```

In particular, this data range contains the integer zero and all negative integers; however, it also contains all strings (since strings are not positive integers).

8.2.4 Datatype Restrictions

⟨*DatatypeRestriction*⟩ ::= ⟨*Datatype*⟩ '`\drres`' ⟨*DatatypeRestrictionExpression*⟩

⟨*DatatypeRestrictionExpression*⟩ ::= '`{`' ⟨*constrainingFacet*⟩ ⟨*restrictionValue*⟩
('`,`' ⟨*constrainingFacet*⟩ ⟨*restrictionValue*⟩)* '`}`'

⟨*constrainingFacet*⟩ ::= ⟨*IRI*⟩

⟨*restrictionValue*⟩ ::= ⟨*Literal*⟩

Example 21.

The following data range contains exactly the integers 5, 6, 7, 8, and 9:

```
(xsd:integer \drres{xsd:minInclusive "5"[xsd:integer],
    xsd:maxExclusive "10"[xsd:integer]})
```

9 Class Expressions

⟨*ClassExpression*⟩ ::= ⟨*Class*⟩
  | ⟨*AtomicClassExpression*⟩
  | '(' ⟨*NonAtomicClassExpression*⟩ ')'

9.1 Atomic Class Expression

9.1.1 Propositional Connectives and Enumeration of Individuals

⟨*AtomicClassExpression*⟩ :: =⟨*SequenceObjectIntersectionOf*⟩ | ⟨*SequenceObjectUnionOf*⟩
  | ⟨*ObjectOneOf*⟩
  | ⟨*ObjectSomeValueFrom*⟩ | ⟨*ObjectAllValueFrom*⟩ | ⟨*ObjectHasValue*⟩
  | ⟨*ObjectMinCardinality*⟩ | ⟨*ObjectMaxCardinality*⟩ | ⟨*ObjectExactCardinality*⟩
  | ⟨*DataSomeValueFrom*⟩ | ⟨*DataAllValueFrom*⟩ | ⟨*DataHasValue*⟩ |
  | ⟨*DataMinCardinality*⟩ | ⟨*DataMaxCardinality*⟩ | ⟨*DataExactCardinality*⟩

Sequence Intersection of Class Expressions

⟨*SequenceObjectIntersectionOf*⟩ ::= '`\candof`' '{' ⟨*ClassExpression*⟩ (',' ⟨*ClassExpression*⟩)+ '}'

Example 22.

Example 38 can be rewritten with the sequence-style notation:

```
\candof{a:Dog, a:CanTalk}
```

Sequence Union of Class Expressions

⟨*SequenceObjectUnionOf*⟩ ::= '`\corof`' '{' ⟨*ClassExpression*⟩ (',' ⟨*ClassExpression*⟩)+ '}'

Example 23.

Example 39 can be rewritten with the sequence-style notation:

```
\corof{a:Man, a:Woman}
```

Enumeration of Individuals

⟨*ObjectOneOf*⟩ ::= '`\ooneof`' '`{`' ⟨*Individual*⟩ ('`,`' ⟨*Individual*⟩)* '`}`'

Example 24.

Consider the ontology consisting of the following axioms.

The Griffin family consists exactly of Peter, Lois, Stewie, Meg, Chris, and Brian.

```
a:GriffinFamilyMember \ceq \ooneof{a:Peter, a:Lois, a:
    Stewie, a:Meg, a:Chris, a:Brian}
```

Quagmire, Peter, Lois, Stewie, Meg, Chris, and Brian are all different from each other.

```
\ialldiff{a:Quagmire, a:Peter, a:Lois, a:Stewie, a:Meg, a
    :Chris, a:Brian}
```

The class a:GriffinFamilyMember now contains exactly the six explicitly listed individuals. Since we also know that a:Quagmire is different from these six individuals, this individual is classified as an instance of the following class expression:

```
(\cnot a:GriffinFamilyMember)
```

9.1.2 Object Property Restrictions

Existential Quantification

⟨*ObjectSomeValuesFrom*⟩ ::= '`\oexists`' '`{`' ⟨*ObjectPropertyExpression*⟩ '`}`' '`{`' ⟨*ClassExpression*⟩ '`}`'

Example 25.

Consider the ontology consisting of the following axioms.

Peter is Stewie's father.

```
a:fatherOf(a:Peter, a:Stewie)
```

Stewie is a man.

```
a:Man(a:Stewie)
```

The following existential expression contains those individuals that are connected by the a:fatherOf property to individuals that are instances of a:Man; furthermore, a:Peter is classified as its instance:

```
\oexists{a:fatherOf}{a:Man}
```

Universal Quantification

⟨*ObjectAllValuesFrom*⟩ ::= '`\oforall`' '`{`' ⟨*ObjectPropertyExpression*⟩ '`}`' '`{`' ⟨*ClassExpression*⟩ '`}`'

Example 26.

Consider the ontology consisting of the following axioms.

Brian is a pet of Peter.

```
a:hasPet(a:Peter, a:Brian)
```

Brian is a dog.

```
a:Dog(a:Brian)
```

Peter has at most one pet.

```
\o[<=1]{a:hasPet}{a:Peter}
```

The following universal expression contains those individuals that are connected through the a:hasPet property only with individuals that are instances of a:Dog – that is, it contains individuals that have only dogs as pets:

```
\oforall{a:hasPet}{a:Dog}
```

The ontology axioms clearly state that a:Peter is connected by a:hasPet only to instances of a:Dog: it is impossible to connect a:Peter by a:hasPet to an individual different from a:Brian without making the ontology inconsistent. Therefore, a:Peter is classified as an instance of the mentioned class expression.

The last axiom – that is, the one stating that a:Peter has at most one pet – is critical for the inference from the previous paragraph due to the open-world semantics of OWL 2. Without this axiom, the ontology might not have listed all the individuals to which a:Peter is connected by a:hasPet. In such a case a:Peter would not be classified as an instance of the mentioned class expression.

Individual Value Restriction

⟨*ObjectHasValue*⟩ ::= '`\ohasvalue`' '`{`' ⟨*ObjectPropertyExpression*⟩ '`}`' '`{`' ⟨*Individual*⟩ '`}`'

Example 27.

Consider the ontology consisting of the following axioms.

Peter is Stewie's father.

```
a:fatherOf(a:Peter, a:Stewie)
```

The following has-value class expression contains those individuals that are connected through the a:fatherOf property with the individual a:Stewie; furthermore, a:Peter is classified as its instance:

```
\ohasvalue{a:fatherOf}{a:Stewie}
```

9.1.3 Object Property Cardinality Restrictions

Minimum Cardinality

⟨*ObjectMinCardinality*⟩ ::= '`\o[>=`' ⟨*nonNegativeInteger*⟩ '`]`' '`{`' ⟨*ObjectPropertyExpression*⟩ '`}`' ['`{`' ⟨*ClassExpression*⟩ '`}`']

Example 28.

Consider the ontology consisting of the following axioms.

Peter is Stewie's father.

```
a:fatherOf(a:Peter, a:Stewie)
```

Stewie is a man.

```
a:Man(a:Stewie)
```

Peter is Chris's father.

```
a:fatherOf(a:Peter, a:Chris)
```

Chris is a man.

```
a:Man(a:Chris)
```

Chris and Stewie are different from each other.

```
a:Chris \idiff a:Stewie
```

The following minimum cardinality expression contains those individuals that are connected by a:fatherOf to at least two different instances of a:Man:

```
\o[>=2]{a:fatherOf}{a:Man}
```

Since a:Stewie and a:Chris are both instances of a:Man and are different from each other, a:Peter is classified as an instance of this class expression. Due to the open-world semantics, the last axiom – the one stating that a:Chris and a:Stewie are different from each other – is necessary for this inference: without this axiom, it is possible that a:Chris and a:Stewie are actually the same individual.

Maximum Cardinality

⟨*ObjectMaxCardinality*⟩ ::= '`\o[<=`' ⟨*nonNegativeInteger*⟩ '`]`' '`{`' ⟨*ObjectPropertyExpression*⟩ '`}`' ['`{`' ⟨*ClassExpression*⟩ '`}`']

Example 29.

Consider the ontology consisting of the following axioms.

```
a:hasPet(a:Peter, a:Brian)
```

Peter has at most one pet.

```
\o[<=1]{a:hasPet}{a:Peter}
```

The following maximum cardinality expression contains those individuals that are connected by a:hasPet to at most two individuals:

```
\o[<=2]{a:hasPet}
```

Since a:Peter is known to be connected by a:hasPet to at most one individual, it is certainly also connected by a:hasPet to at most two individuals so, consequently, a:Peter is classified as an instance of this class expression. The example ontology explicitly names only a:Brian as being connected by a:hasPet from a:Peter, so one might expect a:Peter to be classified as an instance of the mentioned class expression even without the second axiom. This, however, is not the case due to the open-world semantics. Without the last axiom, it is possible that a:Peter is connected by a:hasPet to other individuals. The second axiom closes the set of individuals that a:Peter is connected to by a:hasPet.

Example 30.

Consider the ontology consisting of the following axioms.

Meg is a daughter of Peter.

```
a:hasDaughter(a:Peter, a:Meg)
```

Megan is a daughter of Peter.

```
a:hasDaughter(a:Peter, a:Megan)
```

Peter has at most one daughter.

```
\o[<=1]{a:hasDaughter}{a:Peter}
```

One might expect this ontology to be inconsistent: on the one hand, it says that a:Meg and a:Megan are connected to a:Peter by a:hasDaughter, but, on the other hand, it says that a:Peter is connected by a:hasDaughter to at most one individual. This ontology, however, is not inconsistent because the semantics of OWL 2 does not make the unique name assumption – that is, it does not assume distinct individuals to be necessarily different. For example, the ontology does not explicitly say that a:Meg and a:Megan are different individuals; therefore, since a:Peter can be connected by a:hasDaughter to at most one distinct individual, a:Meg and a:Megan must be the same. This example ontology thus entails the following assertion:

```
a:Meg \ieq a:Megan
```

One can axiomatize the unique name assumption in OWL 2 by explicitly stating that all individuals are different from each other. This can be done by adding the following axiom, which makes the example ontology inconsistent.

Peter, Meg, and Megan are all different from each other.

```
\ialldiff{a:Peter, a:Meg, a:Megan}
```

Exact Cardinality

⟨*ObjectExactCardinality*⟩ ::= '`\o[=`' ⟨*nonNegativeInteger*⟩ '`]`' '`{`' ⟨*ObjectPropertyExpression*⟩ '`}`' ['`{`' ⟨*ClassExpression*⟩ '`}`']

Example 31.

Consider the ontology consisting of the following axioms.

Brian is a pet of Peter.

```
a:hasPet(a:Peter, a:Brian)
```

Brian is a dog.

```
a:Dog(a:Brian)
```

Each pet of Peter is either Brian or it is not a dog.

```
\oforall{a:hasPet}{(\ooneof{a:Brian} \cor (\cnot a:Dog))
    }(a:Peter)
```

The following exact cardinality expression contains those individuals that are connected by a:hasPet to exactly one instance of a:Dog; furthermore, a:Peter is classified as its instance:

```
\o[=1]{a:hasPet}{a:Dog}
```

This is because the first two axioms say that a:Peter is connected to a:Brian by a:hasPet and that a:Brian is an instance of a:Dog, and the last axiom says that any individual different from a:Brian that is connected to a:Peter by a:hasPet is not an instance of a:Dog; hence, a:Peter is connected to exactly one instance of a:Dog by a:hasPet.

9.1.4 Data Property Restrictions

Existential Quantification

⟨*DataSomeValuesFrom*⟩ ::= '`\dexists`' '`{`' ⟨*DataPropertyExpression*⟩ '`}`' '`{`' ⟨*DataRange*⟩ '`}`'

Example 32.

Consider the ontology consisting of the following axioms.

Meg is seventeen years old.

```
a:hasAge(a:Meg, "17"[xsd:integer])
```

The following existential class expression contains all individuals that are connected by a:hasAge to an integer strictly less than 20 so; furthermore, a:Meg is classified as its instance:

```
\dexists{a:hasAge}{(xsd:integer \drres{xsd:maxExclusive
    "20"[xsd:integer])}
```

Universal Quantification

⟨*DataAllValuesFrom*⟩ ::= '`\dforall`' '`{`' ⟨*DataPropertyExpression*⟩ '`}`' '`{`' ⟨*DataRange*⟩ '`}`'

Example 33.

Consider the ontology consisting of the following axioms.

The ZIP code of _:a1 is the integer "02903".

```
a:hasZIP(_:a1, "02903"[xsd:integer])
```

Each object can have at most one ZIP code.

```
a:hasZIP \dfunc
```

In United Kingdom and Canada, ZIP codes are strings (i.e., they can contain characters and not just numbers). Hence, one might use the following universal expression to identify those individuals that have only integer ZIP codes (and therefore have non-UK and non-Canadian addresses):

```
\dforall{a:hasZIP}{xsd:integer}
```

The anonymous individual _:a1 is by the first axiom connected by a:hasZIP to an integer, and the second axiom ensures that _:a1 is not connected by a:hasZIP to other literals; therefore, _:a1 is classified as an instance of the mentioned class expression. The last axiom – the one stating that a:hasZIP is functional – is critical for the inference from the previous paragraph due to the open-world semantics of OWL 2. Without this axiom, the ontology is not guaranteed to list all literals that _:a1 is connected to by a:hasZIP; hence, without this axiom _:a1 would not be classified as an instance of the mentioned class expression.

Literal Value Restriction

⟨*DataHasValue*⟩ ::= '`\dhasvalue`' '`{`' ⟨*DataPropertyExpression*⟩ '`}`' '`{`' ⟨*Literal*⟩ '`}`'

Example 34.

Consider the ontology consisting of the following axioms.

Meg is seventeen years old.

```
a:hasAge(a:Meg, "17"[xsd:integer]})
```

The following has-value expression contains all individuals that are connected by a:hasAge to the integer 17; furthermore, a:Meg is classified as its instance:

```
\dhasvalue{a:hasAge}{"17"[xsd:integer]}
```

9.1.5 Data Property Cardinality Restrictions

Minimum Cardinality

⟨*DataMinCardinality*⟩ ::= '`\d[>=`' ⟨*nonNegativeInteger*⟩ '`]`' '`{`' ⟨*DataPropertyExpression*⟩ '`}`' ['`{`' ⟨*DataRange*⟩ '`}`']

Example 35.

Consider the ontology consisting of the following axioms.

Meg's name is "Meg Griffin".

```
a:hasName(a:Meg, "Meg Griffin")
```

Meg's name is "Megan Griffin".

```
a:hasName(a:Meg, "Megan Griffin")
```

The following minimum cardinality expression contains those individuals that are connected by a:hasName to at least two different literals:

```
\d[>=2]{a:hasName}
```

Different string literals are distinct, so "Meg Griffin" and "Megan Griffin" are different; thus, the individual a:Meg is classified as an instance of the mentioned class expression.

```
\d[>=2]{a:hasName}
```

Maximum Cardinality

⟨*DataMaxCardinality*⟩ ::= '`\d[<=`' ⟨*nonNegativeInteger*⟩ '`]`' '`{`' ⟨*DataPropertyExpression*⟩ '`}`' ['`{`' ⟨*DataRange*⟩ '`}`']

Example 36.

Consider the ontology consisting of the following axioms.

Each object can have at most one name.

```
a:hasName \dfunc
```

The following maximum cardinality expression contains those individuals that are connected by a:hasName to at most two different literals:

```
\d[<=2]{a:hasName}
```

Since the ontology axiom restricts a:hasName to be functional, all individuals in the ontology are instances of this class expression.

Exact Cardinality

⟨*DataExactCardinality*⟩ ::= '`\d[=`' ⟨*nonNegativeInteger*⟩ '`]`' '`{`' ⟨*DataPropertyExpression*⟩ '`}`' ['`{`' ⟨*DataRange*⟩ '`}`']

Example 37.

Consider the ontology consisting of the following axioms.

Brian's name is "Brian Griffin".

```
a:hasName(a:Brian, "Brian Griffin")
```

Each object can have at most one name.

```
a:hasName \dfunc
```

The following exact cardinality expression contains those individuals that are connected by a:hasName to exactly one literal:

```
\d[=1]{a:hasName}
```

Since the ontology axiom restricts a:hasName to be functional and a:Brian is connected by a:hasName to "Brian Griffin", it is classified as an instance of this class expression.

9.2 Non-atomic Class Expression

⟨*NonAtomicClassExpression*⟩ ::= ⟨*ObjectIntersectionOf*⟩ | ⟨*ObjectUnionOf*⟩ | ⟨*ObjectComplementOf*⟩ | ⟨*ObjectHasSelf*⟩

9.2.1 Propositional Connectives

Intersection of Class Expressions

⟨*ObjectIntersectionOf*⟩ ::= ⟨*ClassExpression*⟩ '`\cand`' ⟨*ClassExpression*⟩

Example 38.

Consider the ontology consisting of the following axioms.

Brian is a dog.

```
a:Dog(a:Brian)
```

Brian can talk.

```
a:CanTalk(a:Brian)
```

The following class expression describes all dogs that can talk; furthermore, a:Brian is classified as its instance.

```
(a:Dog \cand a:CanTalk)
```

Union of Class Expressions

⟨*ObjectUnionOf*⟩ ::= ⟨*ClassExpression*⟩ '`\cor`' ⟨*ClassExpression*⟩

Example 39.

Consider the ontology consisting of the following axioms.

Peter is a man.

```
a:Man(a:Peter)
```

Lois is a woman.

```
a:Woman(a:Loid)
```

The following class expression describes all individuals that are instances of either a:Man or a:Woman; furthermore, both a:Peter and a:Lois are classified as its instances:

```
(a:Man \cor a:Woman)
```

Complement of Class Expressions

⟨*ObjectComplementOf*⟩ ::= '`\cnot`' ⟨*ClassExpression*⟩

Example 40.

Consider the ontology consisting of the following axioms.

Nothing can be both a man and a woman.

```
a:Man \cdisjoint a:Woman
```

Lois is a woman.

```
a:Woman(a:Lois)
```

The following class expression describes all things that are not instances of a:Man:

```
(\cnot a:Man)
```

Since a:Lois is known to be a woman and nothing can be both a man and a woman, then a:Lois is necessarily not a a:Man; therefore, a:Lois is classified as an instance of this complement class expression.

Example 41.

OWL 2 has open-world semantics, so negation in OWL 2 is the same as in classical (first-order) logic. To understand open-world semantics, consider the ontology consisting of the following assertion.

Brian is a dog.

```
a:Dog(a:Brian)
```

One might expect a:Brian to be classified as an instance of the following class expression:

```
(\cnot a:Bird)
```

Intuitively, the ontology does not explicitly state that a:Brian is an instance of a:Bird, so this statement seems to be false. In OWL 2, however, this is not the case: it is true that the ontology does not state that a:Brian is an instance of a:Bird; however, the ontology does not state the opposite either. In other words, this ontology simply does not contain enough information to answer the question whether a:Brian is an instance of a:Bird or not: it is perfectly possible that the information to that effect is actually true but it has not been included in the ontology.

The ontology from the previous example (in which a:Lois has been classified as a:Man), however, contains sufficient information to draw the expected conclusion. In particular, we know for sure that a:Lois is an instance of a:Woman and that a:Man and a:Woman do not share instances. Therefore, any additional information that does not lead to inconsistency cannot lead to a conclusion that a:Lois is an instance of a:Man; furthermore, if one were to explicitly state that a:Lois is an instance of a:Man, the ontology would be inconsistent and, by definition, it then entails all possible conclusions.

9.2.2 Object Property Restrictions

Self-Restriction

⟨*ObjectHasSelf*⟩ ::= '`\ohasself`' ⟨*ObjectPropertyExpression*⟩

Example 42.

Consider the ontology consisting of the following axioms.

Peter likes Peter.

```
a:likes(a:Peter, a:Peter)
```

The following self-restriction contains those individuals that like themselves; furthermore, a:Peter is classified as its instance:

```
(\ohasself a:likes)
```

10 Axiom

⟨*Axiom*⟩ ::= ⟨*Declaration*⟩ | ⟨*ClassAxiom*⟩ | ⟨*ObjectPropertyAxiom*⟩
| ⟨*DataPropertyAxiom*⟩ | ⟨*DatatypeDefinition*⟩ | ⟨*HasKey*⟩
| ⟨*Assertion*⟩ | ⟨*AnnotationAxiom*⟩

⟨*axiomAnnotations*⟩ ::= [‘`[`’ ⟨*Annotation*⟩ (‘`,`’ ⟨*Annotation*⟩)* ‘`]`’]

10.1 Class Expression Axioms

⟨*ClassAxiom*⟩ ::= ⟨*NonSequenceClassAxiom*⟩ | ⟨*SequenceClassAxiom*⟩

10.1.1 Non-sequence Class Expression Axioms

⟨*NonSequenceClassAxiom*⟩ ::= ⟨*SubClassOf*⟩ | ⟨*EquivalentClasses*⟩ | ⟨*DisjointClasses*⟩
| ⟨*DisjointUnion*⟩

Subclass Axioms

⟨*SubClassOf*⟩ ::= ⟨*subClassExpression*⟩ ‘`\cisa`’ ⟨*superClassExpression*⟩ ⟨*axiomAnnotations*⟩

⟨*subClassExpression*⟩ ::= ⟨*ClassExpression*⟩

⟨*superClassExpression*⟩ ::= ⟨*ClassExpression*⟩

Example 43.

Consider the ontology consisting of the following axioms.

Each baby is a child.

```
a:Baby \cisa a:Child
```

Each child is a person.

```
a:Child \cisa a:Person
```

Stewie is a baby.

```
a:Baby(a:Stewie)
```

Since a:Stewie is an instance of a:Baby, by the first subclass axiom a:Stewie is classified as an instance of a:Child as well. Similarly, by the second subclass axiom a:Stewie is classified as an instance of a:Person. This style of reasoning can be applied to any instance of a:Baby and not just a:Stewie; therefore, one can conclude that a:Baby is a subclass of a:Person. In other words, this ontology entails the following axiom:

```
a:Baby \cisa a:Person
```

Example 44.

Consider the ontology consisting of the following axioms.

A person that has a child has either at least one boy or a girl.

```
a:PersonWithChild \cisa \oexists{a:hasChild}{(a:Boy \cor
    a:Girl)}
```

Each boy is a child.

```
a:Boy \cisa a:Child
```

Each girl is a child.

```
a:Girl \cisa a:Child
```

If some object has a child, then this object is a parent.

```
\oexists{a:hasChild}{a:Child} \cisa a:Parent
```

The first axiom states that each instance of a:PersonWithChild is connected to an individual that is an instance of either a:Boy or a:Girl. (Because of the open-world semantics of OWL 2, this does not mean that there must be only one such individual or that all such individuals must be instances of either a:Boy or of a:Girl.) Furthermore, each instance of a:Boy or a:Girl is an instance of a:Child. Finally, the last axiom says that all individuals that are connected by a:hasChild to an instance of a:Child are instances of a:Parent. Since this reasoning holds for each instance of a:PersonWithChild, each such instance is also an instance of a:Parent. In other words, this ontology entails the following axiom:

```
a:PersonWithChild \cisa a:Parent
```

Equivalent Classes

⟨*EquivalentClasses*⟩ ::= ⟨*ClassExpression*⟩ '`\ceq`' ⟨*ClassExpression*⟩ ⟨*axiomAnnotations*⟩

Example 45.

Consider the ontology consisting of the following axioms.

A boy is a male child.

```
a:Boy \ceq (a:Child \cand a:Man)
```

Chris is a child.

```
a:Child(a:Chris)
```

Chris is a man.

```
a:Man(a:Chris)
```

Stewie is a boy.

```
a:Boy(a:Stewie)
```

The first axiom defines the class a:Boy as an intersection of the classes a:Child and a:Man; thus, the instances of a:Boy are exactly those instances that are both an instance of a:Child and an instance of a:Man. Such a definition consists of two directions. The first direction implies that each instance of a:Child and a:Man is an instance of a:Boy; since a:Chris satisfies these two conditions, it is classified as an instance of a:Boy. The second direction implies that each a:Boy is an instance of a:Child and of a:Man; thus, a:Stewie is classified as an instance of a:Man and of a:Boy.

Example 46.

Consider the ontology consisting of the following axioms.

A mongrel owner has a pet that is a mongrel.

```
a:MongrelOwner \ceq \oexists{a:hasPet}{a:Mongrel}
```

A dog owner has a pet that is a dog.

```
a:DogOwner \ceq \oexists{a:hasPet}{a:Dog}
```

Functional-Style Syntax – *Each mongrel is a dog.*

```
a:Mongrel \cisa a:Dog
```

Peter is a mongrel owner.

```
a:MongrelOwner(a:Peter)
```

By the first axiom, each instance x of a:MongrelOwner must be connected via a:hasPet to an instance of a:Mongrel; by the third axiom, this individual is an instance of a:Dog; thus, by the second axiom, x is an instance of a:DogOwner. In other words, this ontology entails the following axiom:

```
a:MongrelOwner(a:DogOwner)
```

By the fourth axiom, a:Peter is then classified as an instance of a:DogOwner.

Disjoint Classes

⟨*DisjointClasses*⟩ : = ⟨*ClassExpression*⟩ '`\cdisjoint`' ⟨*ClassExpression*⟩ ⟨*axiomAnnotations*⟩

Example 47.

Consider the ontology consisting of the following axioms.

Nothing can be both a boy and a girl.

```
a:Boy \cdisjoint a:Girl
```

Stewie is a boy.

```
a:Boy(a:Stewie)
```

The axioms in this ontology imply that a:Stewie can be classified as an instance of the following class expression:

```
(\cnot a:Girl)
```

Furthermore, if the ontology were extended with the following assertion, the ontology would become inconsistent:

```
a:Girl(a:Stewie)
```

Disjoint Union of Class Expression

⟨*DisjointUnion*⟩ ::= ⟨*Class*⟩ '`\cdisjunion`' ⟨*disjointClassExpressions*⟩ ⟨*axiomAnnotations*⟩

⟨*disjointClassExpressions*⟩ ::= '`{`' ⟨*ClassExpression*⟩ ('`,`' ⟨*ClassExpression*⟩)* '`}`'

Example 48.

Consider the ontology consisting of the following axioms.

Each child is either a boy or a girl, each boy is a child, each girl is a child, and nothing can be both a boy and a girl.

```
a:Child \cdisjunion {a:Boy, a:Girl}
```

Stewie is a child.

```
a:Child(a:Stewie)
```

Stewie is not a girl.

```
(\cnot a:Girl)(a:Stewie)
```

By the first two axioms, a:Stewie is either an instance of a:Boy or a:Girl. The last assertion eliminates the second possibility, so a:Stewie is classified as an instance of a:Boy.

10.1.2 Sequence Class Expression Axioms

⟨*SequenceClassAxiom*⟩ ::= ⟨*SequenceEquivalentClasses*⟩ | ⟨*SequenceDisjointClasses*⟩

Sequence Equivalent Classes

⟨*SequenceEquivalentClasses*⟩ ::= '`\calleq`' '`{`' ⟨*ClassExpression*⟩ ('`,`' ⟨*ClassExpression*⟩)+ '`}`' ⟨*axiomAnnotations*⟩

Example 49.

Example 45 can be rewritten with the sequence-style notation:

```
\calleq{a:Boy, (a:Child \cand a:Man)}
```

Sequence Disjoint Classes

⟨*SequenceDisjointClasses*⟩ : = '`\calldisjoint`' '`{`' ⟨*ClassExpression*⟩ ('`,`' ⟨*ClassExpression*⟩)+ '`}`' ⟨*axiomAnnotations*⟩

Example 50.

Example 47 can be rewritten with the sequence-style notation:

```
\calldisjoint{a:Boy, a:Girl}
```

10.2 Object Property Axioms

⟨*ObjectPropertyAxiom*⟩ ::= ⟨*NonSequenceObjectPropertyAxiom*⟩ | ⟨*SequenceObjectPropertyAxiom*⟩

10.2.1 Non-sequence Object Property Axioms

⟨*NonSequenceObjectPropertyAxiom*⟩ ::= ⟨*SubObjectPropertyOf*⟩ | ⟨*EquivalentObjectProperties*⟩
| ⟨*DisjointObjectProperties*⟩ | ⟨*InverseObjectProperties*⟩ | ⟨*ObjectPropertyDomain*⟩
| ⟨*ObjectPropertyRange*⟩ | ⟨*FunctionalObjectProperty*⟩ | ⟨*InverseFunctionalObjectProperty*⟩
| ⟨*ReflexiveObjectProperty*⟩ | ⟨*IrreflexiveObjectProperty*⟩ | ⟨*SymmetricObjectProperty*⟩
| ⟨*AsymmetricObjectProperty*⟩ | ⟨*TransitiveObjectProperty*⟩

Object Subproperties

⟨*SubObjectPropertyOf*⟩ ::= ⟨*subObjectPropertyExpression*⟩ '`\oisa`' ⟨*superObjectPropertyExpression*⟩
⟨*axiomAnnotations*⟩

⟨*subObjectPropertyExpression*⟩ ::= ⟨*ObjectPropertyExpression*⟩ | ⟨*propertyExpressionChain*⟩

⟨*propertyExpressionChain*⟩ ::= '`\ochain`' '`{`' ⟨*ObjectPropertyExpression*⟩ ('`,`'
⟨*ObjectPropertyExpression*⟩)+ '`}`'

⟨*superObjectPropertyExpression*⟩ ::= ⟨*ObjectPropertyExpression*⟩

Example 51.

Consider the ontology consisting of the following axioms.

Having a dog implies having a pet.

```
a:hasDog \oisa a:hasPet
```

Functional-Style Syntax – *Brian is a dog of Peter.*

```
a:hasDog(a:Peter, a:Brian)
```

Since a:hasDog is a subproperty of a:hasPet, each tuple of individuals connected by the former property expression is also connected by the latter property expression. Therefore, this ontology entails that a:Peter is connected to a:Brian by a:hasPet; that is, the ontology entails the following assertion:

```
a:hasPet(a:Peter, a:Brian)
```

Example 52.

Consider the ontology consisting of the following axioms.

The sister of someone's mother is that person's aunt.

```
\ochain{a:hasMother, a:hasSister} \oisa a:hasAunt
```

Lois is the mother of Stewie.

```
a:hasMother(a:Stewie, a:Lois)
```

Carol is a sister of Lois.

```
a:hasSister(a:Lois, a:Carol)
```

The axioms in this ontology imply that a:Stewie is connected by a:hasAunt with a:Carol; that is, the ontology entails the following assertion:

```
a:hasAunt(a:Stewie, a:Carol)
```

Equivalent Object Properties

⟨*EquivalentObjectProperties*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\oeq`' ⟨*ObjectPropertyExpression*⟩ ⟨*axiomAnnotations*⟩

Example 53.

Consider the ontology consisting of the following axioms.

Having a brother is the same as having a male sibling.

```
a:hasBrother \oeq a:hasMaleSibling
```

Stewie is a brother of Chris.

```
a:hasBrother(a:Chris, a:Stewie)
```

Chris is a male sibling of Stewie.

```
a:hasMaleSibling(a:Stewie, a:Chris)
```

Since a:hasBrother and a:hasMaleSibling are equivalent properties, this ontology entails that a:Chris is connected by a:hasMaleSibling with a:Stewie – that is, it entails the following assertion:

```
a:hasMaleSibling(a:Chris, a:Stewie)
```

Furthermore, the ontology also entails that that a:Stewie is connected by a:hasBrother with a:Chris – that is, it entails the following assertion:

```
a:hasBrother(a:Stewie, a:Chris)
```

Disjoint Object Properties

⟨*DisjointObjectProperties*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\odisjoint`' ⟨*ObjectPropertyExpression*⟩ ⟨*axiomAnnotations*⟩

Example 54.

Consider the ontology consisting of the following axioms.

Fatherhood is disjoint with motherhood.

```
a:hasFather \odisjoint a:hasMother
```

Peter is Stewie's father.

```
a:hasFather(a:Stewie, a:Peter)
```

Lois is the mother of Stewie.

```
a:hasMother(a:Stewie, a:Lois)
```

In this ontology, the disjointness axiom is satisfied. If, however, one were to add the following assertion, the disjointness axiom would be invalidated and the ontology would become inconsistent:

```
a:hasMother(a:Stewie, a:Peter)
```

Inverse Object Properties

⟨*InverseObjectProperties*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\oinv`' ⟨*ObjectPropertyExpression*⟩ ⟨*axiomAnnotations*⟩

Example 55.

Consider the ontology consisting of the following axioms.

Having a father is the opposite of being a father of someone.

```
a:hasFather \oinv a:fatherOf
```

Peter is Stewie's father.

```
a:hasFather(a:Stewie, a:Peter)
```

Peter is Chris's father.

```
a:fatherOf(a:Peter, a:Chris)
```

This ontology entails that a:Peter is connected by a:fatherOf with a:Stewie – that is, it entails the following assertion:

```
a:fatherOf(a:Peter, a:Stewie)
```

Furthermore, the ontology also entails that a:Chris is connected by a:hasFather with a:Peter – that is, it entails the following assertion:

```
a:hasFather(a:Chris, a:Peter)
```

Object Property Domain

⟨*ObjectPropertyDomain*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\odomain`' ⟨*ClassExpression*⟩ ⟨*axiomAnnotations*⟩

Example 56.

Consider the ontology consisting of the following axioms.

Functional-Style Syntax – *Only people can own dogs.*

```
a:hasDog \odomain a:Person
```

Brian is a dog of Peter.

```
a:hasDog(a:Peter, a:Brian)
```

By the first axiom, each individual that has an outgoing a:hasDog connection must be an instance of a:Person. Therefore, a:Peter can be classified as an instance of a:Person; that is, this ontology entails the following assertion:

```
a:Person(a:Peter)
```

Domain axioms in OWL 2 have a standard first-order semantics that is somewhat different from the semantics of such axioms in databases and object-oriented systems, where such axioms are interpreted as checks. The domain axiom from the example ontology would in such systems be interpreted as a constraint saying that a:hasDog can point only from individuals that are known to be instances of a:Person; furthermore, since the example ontology does not explicitly state that a:Peter is an instance of a:Person, one might expect the domain constraint to be invalidated. This, however, is not the case in OWL 2: as shown in the previous paragraph, the missing type is inferred from the domain constraint.

Object Property Range

⟨*ObjectPropertyRange*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\orange`' ⟨*ClassExpression*⟩ ⟨*axiomAnnotations*⟩

Example 57.

Consider the ontology consisting of the following axioms.

The range of the a:hasDog property is the class a:Dog.

```
a:hasDog \orange a:Dog
```

Brian is a dog of Peter.

```
a:hasDog(a:Peter, a:Brian)
```

By the first axiom, each individual that has an outgoing a:hasDog connection must be an instance of a:Person. Therefore, a:Peter can be classified as an instance of a:Person; that is, this ontology entails the following assertion:

```
a:Dog(a:Brian)
```

Range axioms in OWL 2 have a standard first-order semantics that is somewhat different from the semantics of such axioms in databases and object-oriented systems, where such axioms are interpreted as checks. The range axiom from the example ontology would in such systems be interpreted as a constraint saying that a:hasDog can point only to individuals that are known to be instances of a:Dog; furthermore, since the example ontology does not explicitly state that a:Brian is an instance of a:Dog, one might expect the range constraint to be invalidated. This, however, is not the case in OWL 2: as shown in the previous paragraph, the missing type is inferred from the range constraint.

Functional Object Properties

⟨*FunctionalObjectProperty*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\ofunc`' ⟨*axiomAnnotations*⟩

Example 58.

Consider the ontology consisting of the following axioms.

Each object can have at most one father.

```
a:hasFather \ofunc
```

Peter is Stewie's father.

```
a:hasFather(a:Stewie, a:Peter)
```

Peter Griffin is Stewie's father.

```
a:hasFather(a:Stewie, a:Peter_Griffin)
```

By the first axiom, a:hasFather can point from a:Stewie to at most one distinct individual, so a:Peter and a:Peter_Griffin must be equal; that is, this ontology entails the following assertion:

```
a:Peter \ieq a:Peter_Griffin
```

One might expect the previous ontology to be inconsistent, since the a:hasFather property points to two different values for a:Stewie. OWL 2, however, does not make the unique name assumption, so a:Peter and a:Peter_Griffin are not necessarily distinct individuals. If the ontology were extended with the following assertion, then it would indeed become inconsistent:

```
a:Peter \idiff a:Peter_Griffin
```

Inverse-Functional Object Properties

⟨*InverseFunctionalObjectProperty*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\oinvfunc`' ⟨*axiomAnnotations*⟩

Example 59.

Consider the ontology consisting of the following axioms.

Each object can have at most one father.

```
a:fatherOf \oinvfunc
```

Peter is Stewie's father.

```
a:fatherOf(a:Peter, a:Stewie)
```

Peter Griffin is Stewie's father.

```
a:fatherOf(a:Peter_Griffin, a:Stewie)
```

By the first axiom, at most one distinct individual can point by a:fatherOf to a:Stewie, so a:Peter and a:Peter_Griffin must be equal; that is, this ontology entails the following assertion:

```
a:Peter \ieq a:Peter_Griffin
```

One might expect the previous ontology to be inconsistent, since there are two individuals that a:Stewie is connected to by a:fatherOf. OWL 2, however, does not make the unique name assumption, so a:Peter and a:Peter_Griffin are not necessarily distinct individuals. If the ontology were extended with the following assertion, then it would indeed become inconsistent:

```
a:Peter \idiff a:Peter_Griffin
```

Reflexive Object Properties

⟨*ReflexiveObjectProperty*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\oreflex`' ⟨*axiomAnnotations*⟩

Example 60.

Consider the ontology consisting of the following axioms.

Everybody knows themselves.

```
a:knows \oreflex
```

Peter is a person.

```
a:Person(a:Peter)
```

By the first axiom, a:Peter must be connected by a:knows to itself; that is, this ontology entails the following assertion:

```
a:knows(a:Peter, a:Peter)
```

Irreflexive Object Properties

⟨*IrreflexiveObjectProperty*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\oirreflex`' ⟨*axiomAnnotations*⟩

Example 61.

Consider the ontology consisting of the following axioms.

Nobody can be married to themselves.

```
a:marriedTo \oirreflex
```

If this ontology were extended with the following assertion, the irreflexivity axiom would be contradicted and the ontology would become inconsistent:

```
a:marriedTo(a:Peter, a:Peter)
```

Symmetric Object Properties

⟨*SymmetricObjectProperty*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\osym`' ⟨*axiomAnnotations*⟩

Example 62.

Consider the ontology consisting of the following axioms.

If x is a friend of y, then y is a friend of x.

```
a:friend \osym
```

Brian is a friend of Peter.

```
a:friend(a:Peter, a:Brian)
```

Since a:friend is symmetric, a:Peter must be connected by a:friend to a:Brian; that is, this ontology entails the following assertion:

```
a:friend(a:Brian, a:Peter)
```

Asymmetric Object Properties

⟨*AsymmetricObjectProperty*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\oasym`' ⟨*axiomAnnotations*⟩

Example 63.

Consider the ontology consisting of the following axioms.

If x is a parent of y, then y is not a parent of x.

```
a:parentOf \oasym
```

Peter is a parent of Stewie.

```
a:parentOf(a:Peter, a:Stewie)
```

If this ontology were extended with the following assertion, the asymmetry axiom would be invalidated and the ontology would become inconsistent:

```
a:parentOf(a:Stewie, a:Peter)
```

Transitive Object Properties

⟨*TransitiveObjectProperty*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '`\otrans`' ⟨*axiomAnnotations*⟩

Example 64.

Consider the ontology consisting of the following axioms.

If x is an ancestor of y and y is an ancestor of z, then x is an ancestor of z.

```
a:ancestorOf \otrans
```

Carter is an ancestor of Lois.

```
a:ancestorOf(a:Carter, a:Lois)
```

Lois is an ancestor of Meg.

```
a:ancestorOf(a:Lois, a:Meg)
```

Since a:ancestorOf is transitive, a:Carter must be connected by a:ancestorOf to a:Meg – that is, this ontology entails the following assertion:

```
a:ancestorOf(a:Carter, a:Meg)
```

10.2.2 Sequence Object Property Axioms

⟨*SequenceObjectPropertyAxiom*⟩ :: = ⟨*SequenceEquivalentObjectProperties*⟩
  | ⟨*SequenceDisjointObjectProperties*⟩

Sequence Equivalent Object Properties

⟨*SequenceEquivalentObjectProperties*⟩ ::= '`\oalleq`' '`{`' ⟨*ObjectPropertyExpression*⟩
  ('`,`' ⟨*ObjectPropertyExpression*⟩)+ '`}`' ⟨*axiomAnnotations*⟩

Example 65.

Example 53 can be rewritten with the sequence-style notation:

```
\oalleq{a:hasBrother, a:hasMaleSibling}
```

Sequence Disjoint Object Properties

⟨*SequenceDisjointObjectProperties*⟩ ::= '`\oalldisjoint`' '`{`' ⟨*ObjectPropertyExpression*⟩
  ('`,`' ⟨*ObjectPropertyExpression*⟩)+ '`}`' ⟨*axiomAnnotations*⟩

Example 66.

Example 54 can be rewritten with the sequence-style notation:

```
\oalldisjoint{a:hasFather, a:hasMother}
```

10.3 Data Property Axioms

⟨*DataPropertyAxiom*⟩ ::= ⟨*NonSequenceDataPropertyAxiom*⟩ | ⟨*SequenceDataPropertyAxiom*⟩

10.3.1 Non-sequence Object Property Axioms

⟨*NonSequenceDataPropertyAxiom*⟩ ::= ⟨*SubDataPropertyOf*⟩ | ⟨*EquivalentDataProperties*⟩
  | ⟨*DisjointDataProperties*⟩
  | ⟨*DataPropertyDomain*⟩ | ⟨*DataPropertyRange*⟩ | ⟨*FunctionalDataProperty*⟩

Data Subproperties

$\langle$*SubDataPropertyOf*$\rangle$ ::= $\langle$*subDataPropertyExpression*$\rangle$ '`\disa`' $\langle$*superDataPropertyExpression*$\rangle$ $\langle$*axiomAnnotations*$\rangle$

$\langle$*subDataPropertyExpression*$\rangle$:= $\langle$*DataPropertyExpression*$\rangle$

$\langle$*superDataPropertyExpression*$\rangle$:= $\langle$*DataPropertyExpression*$\rangle$

Example 67.

Consider the ontology consisting of the following axioms.

A last name of someone is his/her name as well.

```
a:hasLastName \disa a:hasName
```

Peter's last name is "Griffin".

```
a:hasLastName(a:Peter, "Griffin")
```

Since a:hasLastName is a subproperty of a:hasName, each individual connected by the former property to a literal is also connected by the latter property to the same literal. Therefore, this ontology entails that a:Peter is connected to "Griffin" through a:hasName; that is, the ontology entails the following assertion:

```
a:hasName(a:Peter, "Griffin")
```

Equivalent Data Properties

$\langle$*EquivalentDataProperties*$\rangle$ ::= $\langle$*DataPropertyExpression*$\rangle$ '`\deq`' $\langle$*DataPropertyExpression*$\rangle$ $\langle$*axiomAnnotations*$\rangle$

Example 68.

Consider the ontology consisting of the following axioms.

a:hasName and a:seLlama (in Spanish) are synonyms.

```
a:hasName \deq a:seLlama
```

Meg's name is "Meg Griffin".

```
a:hasName(a:Meg, "Meg Griffin")
```

Meg's name is "Megan Griffin".

```
a:seLlama(a:Meg, "Megan Griffin")
```

Since a:hasName and a:seLlama are equivalent properties, this ontology entails that a:Meg is connected by a:seLlama with “Meg Griffin” – that is, it entails the following assertion:

```
a:seLlama(a:Meg, "Meg Griffin")
```

Furthermore, the ontology also entails that a:Meg is also connected by a:hasName with “Megan Griffin” – that is, it entails the following assertion:

```
a:hasName(a:Meg, "Megan Griffin")
```

Disjoint Data Properties

⟨*DisjointDataProperties*⟩ ::= ⟨*DataPropertyExpression*⟩ ‘`\ddisjoint`’ ⟨*DataPropertyExpression*⟩ ⟨*axiomAnnotations*⟩

Example 69.

Consider the ontology consisting of the following axioms.

Someone’s name must be different from his address.

```
a:hasName \ddisjoint a:hasAddress
```

Peter’s name is “Peter Griffin”.

```
a:hasName(a:Peter, "Peter Griffin")
```

Peter’s address is Quahog, Rhode Island.

```
a:hasAddress(a:Peter, "Quahog, Rhode Island"})
```

In this ontology, the disjointness axiom is satisfied. If, however, one were to add the following assertion, the disjointness axiom would be invalidated and the ontology would become inconsistent:

```
a:hasAddress(a:Peter, "Peter Griffin")
```

Data Property Domain

⟨*DataPropertyDomain*⟩ ::= ⟨*DataPropertyExpression*⟩ ‘`\ddomain`’ ⟨*ClassExpression*⟩ ⟨*axiomAnnotations*⟩

Example 70.

Consider the ontology consisting of the following axioms.

Only people can have names.

```
a:hasName \ddomain a:Person
```

Peter's name is "Peter Griffin".

```
a:hasName(a:Peter, "Peter Griffin")
```

By the first axiom, each individual that has an outgoing a:hasName connection must be an instance of a:Person. Therefore, a:Peter can be classified as an instance of a:Person – that is, this ontology entails the following assertion:

```
a:Person(a:Peter)
```

Domain axioms in OWL 2 have a standard first-order semantics that is somewhat different from the semantics of such axioms in databases and object-oriented systems, where such axioms are interpreted as checks. Thus, the domain axiom from the example ontology would in such systems be interpreted as a constraint saying that a:hasName can point only from individuals that are known to be instances of a:Person; furthermore, since the example ontology does not explicitly state that a:Peter is an instance of a:Person, one might expect the domain constraint to be invalidated. This, however, is not the case in OWL 2: as shown in the previous paragraph, the missing type is inferred from the domain constraint.

Data Property Range

⟨*DataPropertyRange*⟩ ::= ⟨*DataPropertyExpression*⟩ '`\drange`' ⟨*DataRange*⟩ ⟨*axiomAnnotations*⟩

Example 71.

Consider the ontology consisting of the following axioms.

The range of the a:hasName property is xsd:string.

```
a:hasName \drange xsd:string
```

Peter's name is "Peter Griffin".

```
a:hasName(a:Peter, "Peter Griffin")
```

By the first axiom, each literal that has an incoming a:hasName link must be in xsd:string. In the example ontology, this axiom is satisfied. If, however, the ontology were extended with the following assertion, then the range axiom would imply that the literal "42"[xsd:integer] is in xsd:string, which is a contradiction and the ontology would become inconsistent:

```
a:hasName(a:Peter, "42"[xsd:integer])
```

Functional Data Properties

⟨*FunctionalDataProperty*⟩ ::= ⟨*DataPropertyExpression*⟩ '`\dfunc`' ⟨*axiomAnnotations*⟩

Example 72.

Consider the ontology consisting of the following axioms.

Each object can have at most one age.

```
a:hasAge \dfunc
```

Meg is seventeen years old.

```
a:hasAge(a:Meg, "17"[xsd:integer])
```

By the first axiom, a:hasAge can point from a:Meg to at most one distinct literal. In this example ontology, this axiom is satisfied. If, however, the ontology were extended with the following assertion, the semantics of functionality axioms would imply that "15"[xsd:integer] is equal to "17"[xsd:integer], which is a contradiction and the ontology would become inconsistent:

```
a:hasAge(a:Meg, "15"[xsd:integer])
```

10.3.2 Sequence Object Property Axioms

⟨*SequenceDataPropertyAxiom*⟩ ::= ⟨*SequenceEquivalentDataProperties*⟩
| ⟨*SequenceDisjointDataProperties*⟩

Equivalent Data Properties

⟨*SequenceEquivalentDataProperties*⟩ ::= '`\dalleq`' '`{`' ⟨*DataPropertyExpression*⟩ ('`,`' ⟨*DataPropertyExpression*⟩)+ '`}`' ⟨*axiomAnnotations*⟩

Example 73.

Example 68 can be rewritten with the sequence-style notation:

```
\dalleq{a:hasName, a:seLlama}
```

Disjoint Data Properties

⟨*SequenceDisjointDataProperties*⟩ ::= '`\dalldisjoint`' '`{`' ⟨*DataPropertyExpression*⟩ ('`,`' ⟨*DataPropertyExpression*⟩)+ '`}`' ⟨*axiomAnnotations*⟩

Example 74.

Example 69 can be rewritten with the sequence-style notation:

```
\dalldisjoint{a:hasName, a:hasAddress}
```

10.4 Datatype Definitions

⟨*DatatypeDefinition*⟩ := ⟨*Datatype*⟩ '`\dtdef`' ⟨*DataRange*⟩ ⟨*axiomAnnotations*⟩

Example 75.

Consider the ontology consisting of the following axioms.

a:SSN is a datatype.

```
a:SSN \d
```

A social security number is a string that matches the given regular expression.

```
a:SSN \dtdef (xsd:string \drres{xsd:pattern
    "[0-9]{3}-[0-9]{2}-[0-9]{4}"})
```

The range of the a:hasSSN property is a:SSN.

```
a:hasSSN \drange a:SSN
```

The second axiom defines a:SSN as an abbreviation for a datatype restriction on xsd:string. In order to satisfy the typing restrictions the first axiom explicitly declares a:SSN to be a datatype. The datatype a:SSN can be used just like any other datatype; for example, it is used in the third axiom to define the range of the a:hasSSN property. The only restriction is that a:SSN supports no facets and therefore cannot be used in datatype restrictions, and that there can be no literals of datatype a:SSN.

10.5 Keys

⟨*HasKey*⟩ := ⟨*ClassExpression*⟩ '`\key`' ⟨*HasKeyExpression*⟩ ⟨*axiomAnnotations*⟩

⟨*HasKeyExpression*⟩ ::= '`{`' [⟨*ObjectPropertyExpression*⟩ ('`,`' ⟨*ObjectPropertyExpression*⟩)*] '`}`' '`{`' [⟨*DataPropertyExpression*⟩ ('`,`' ⟨*DataPropertyExpression*⟩)*] '`}`'

Example 76.

Consider the ontology consisting of the following axioms.

Each object is uniquely identified by its social security number.

```
owl:Thing \key {}{a:hasSSN}
```

Peter's social security number is "123-45-6789".

```
a:hasSSN(a:Peter, "123-45-6789")
```

Peter Griffin's social security number is "123-45-6789".

```
a:hasSSN(a:Peter_Griffin, "123-45-6789")
```

The first axiom makes a:hasSSN the key for instances of the owl:Thing class; thus, only one individual can have a particular value for a:hasSSN. Since the values of a:hasSSN are the same for the individuals a:Peter and a:Peter_Griffin, these two individuals are equal – that is, this ontology entails the following assertion:

```
a:Peter \ieq a:Peter_Griffin
```

One might expect the previous ontology to be inconsistent, since the a:hasSSN has the same value for two individuals a:Peter and a:Peter_-Griffin. However, OWL 2 does not make the unique name assumption, so a:Peter and a:Peter_Griffin are not necessarily distinct individuals. If the ontology were extended with the following assertion, then it would indeed become inconsistent:

```
a:Peter \idiff a:Peter_Griffin
```

10.6 Assertion

$\langle \mathit{Assertion} \rangle$ ::= $\langle \mathit{NonSequenceAssertion} \rangle$ | $\langle \mathit{SequenceAssertion} \rangle$

$\langle \mathit{sourceIndividual} \rangle$ ::= $\langle \mathit{Individual} \rangle$

$\langle \mathit{targetIndividual} \rangle$ ::= $\langle \mathit{Individual} \rangle$

$\langle \mathit{targetValue} \rangle$ ::= $\langle \mathit{Literal} \rangle$

10.6.1 Non-sequence Assertion

$\langle \mathit{NonSequenceAssertion} \rangle$ ::= $\langle \mathit{SameIndividual} \rangle$ | $\langle \mathit{DifferentIndividuals} \rangle$ | $\langle \mathit{ClassAssertion} \rangle$
| $\langle \mathit{ObjectPropertyAssertion} \rangle$ | $\langle \mathit{NegativeObjectPropertyAssertion} \rangle$ |
| $\langle \mathit{DataPropertyAssertion} \rangle$ | $\langle \mathit{NegativeDataPropertyAssertion} \rangle$

Individual Equality

⟨*SameIndividual*⟩ ::= ⟨*Individual*⟩ '`\ieq`' ⟨*Individual*⟩ ⟨*axiomAnnotations*⟩

Example 77.

Consider the ontology consisting of the following assertion.

Functional-Style Syntax – *Meg and Megan are the same objects.*

```
a:Meg \ieq a:Megan
```

Meg has a brother Stewie.

```
a:hasBrother(a:Meg, a:Stewie)
```

Since a:Meg and a:Megan are equal, one individual can always be replaced with the other one. Therefore, this ontology entails that a:Megan is connected by a:hasBrother with a:Stewie – that is, the ontology entails the following assertion:

```
a:hasBrother(a:Megan, a:Stewie)
```

Individual Inequality

⟨*DifferentIndividuals*⟩ ::= ⟨*Individual*⟩ '`\idiff`' ⟨*Individual*⟩ ⟨*axiomAnnotations*⟩

Class Assertions

⟨*ClassAssertion*⟩ ::= ⟨*ClassExpression*⟩ '`(`' ⟨*Individual*⟩ '`)`' ⟨*axiomAnnotations*⟩

Example 78.

Consider the ontology consisting of the following assertion.

Brian is a dog.

```
a:Dog(a:Brian)
```

Each dog is a mammal.

```
a:Dog \cisa a:Mammal
```

The first axiom states that a:Brian is an instance of the class a:Dog. By the second axiom, each instance of a:Dog is an instance of a:Mammal. Therefore, this ontology entails that a:Brian is an instance of a:Mammal – that is, the ontology entails the following assertion:

```
a:Mammal(a:Brian)
```

Positive Object Property Assertions

⟨*ObjectPropertyAssertion*⟩ ::= ⟨*ObjectPropertyExpression*⟩ '(' ⟨*sourceIndividual*⟩
 ',' ⟨*targetIndividual*⟩ ')' ⟨*axiomAnnotations*⟩

Example 79.

Consider the ontology consisting of the following assertion.

Brian is a dog of Peter.

```
a:hasDog(a:Peter, a:Brian)
```

Objects that have a dog are dog owners.

```
\oexists{a:hasDog}{owl:Thing} \cisa a:DogOwner
```

The first axiom states that a:Peter is connected by a:hasDog to a:Brian. By the second axiom, each individual connected by a:hasDog to an individual is an instance of a:DogOwner. Therefore, this ontology entails that a:Peter is an instance of a:DogOwner – that is, the ontology entails the following assertion:

```
a:DogOwner(a:Peter)
```

Negative Object Property Assertions

⟨*NegativeObjectPropertyAssertion*⟩ ::= '!' ⟨*ObjectPropertyExpression*⟩ '(' ⟨*sourceIndividual*⟩
 ',' ⟨*targetIndividual*⟩ ')' ⟨*axiomAnnotations*⟩

Example 80.

Consider the ontology consisting of the following assertion.

Meg is not a son of Peter.

```
!a:hasSon(a:Peter, a:Meg)
```

The ontology would become inconsistent if it were extended with the following assertion:

```
a:hasSon(a:Peter, a:Meg)
```

Positive Data Property Assertions

⟨*DataPropertyAssertion*⟩ := ⟨*DataPropertyExpression*⟩ '(' ⟨*sourceIndividual*⟩ ','
⟨*targetValue*⟩ ')' ⟨*axiomAnnotations*⟩

Example 81.

Consider the ontology consisting of the following axiom.

Functional-Style Syntax – *Meg is seventeen years old.*

```
a:hasAge(a:Meg, "17"^^xsd:integer)
```

Objects that are older than 13 and younger than 19 (both inclusive) are teenagers.

```
\dexists{a:hasAge}{(xsd:integer \drres{xsd:minInclusive
    "13"[xsd:integer], xsd:maxInclusive "19"[xsd:integer
    ]})} \cisa a:Teenager
```

The first axiom states that a:Meg is connected by a:hasAge to the literal "17"[xsd:integer]. By the second axiom, each individual connected by a:hasAge to an integer between 13 and 19 is an instance of a:Teenager. Therefore, this ontology entails that a:Meg is an instance of a:Teenager – that is, the ontology entails the following assertion

```
a:Teenager(a:Meg)
```

Negative Data Property Assertions

⟨*NegativeDataPropertyAssertion*⟩ := '!' ⟨*DataPropertyExpression*⟩ '(' ⟨*sourceIndividual*⟩
',' ⟨*targetValue*⟩ ')' ⟨*axiomAnnotations*⟩

Example 82.

Consider the ontology consisting of the following axiom.

Meg is not five years old.

```
!a:hasAge(a:Meg, "5"[xsd:integer])
```

The ontology would become inconsistent if it were extended with the following assertion:

```
a:hasAge(a:Meg, "5"[xsd:integer]}
```

10.6.2 Sequence Assertion

⟨*SequenceAssertion*⟩ ::= ⟨*SequenceSameIndividual*⟩ | ⟨*SequenceDifferentIndividuals*⟩

Sequence Individual Equality

⟨*SequenceSameIndividual*⟩ ::= '`\ialleq`' '`{`' ⟨*Individual*⟩ ('`,`' ⟨*Individual*⟩)+ '`}`' ⟨*axiomAnnotations*⟩

Example 83.

Example 77 can be rewritten with the sequence-style notation:

```
\ialleq{a:Meg, a:Megan}
```

Individual Inequality

⟨*SequenceDifferentIndividuals*⟩ ::= '`\ialldiff`' '`{`' ⟨*Individual*⟩ ('`,`' ⟨*Individual*⟩)+ '`}`' ⟨*axiomAnnotations*⟩

Example 84.

Consider the ontology consisting of the following axioms.

Peter is Meg's father.

```
a:fatherOf(a:Peter, a:Meg)
```

Peter is Chris's father.

```
a:fatherOf(a:Peter, a:Chris)
```

Peter is Stewie's father.

```
a:fatherOf(a:Peter, a:Stewie)
```

Peter, Meg, Chris, and Stewie are all different from each other.

```
\ialldiff{a:Peter, a:Meg, a:Chris, a:Stewie}
```

The last axiom in this example ontology axiomatizes the unique name assumption (but only for the four names in the axiom). If the ontology were extended with the following axiom stating that a:fatherOf is functional, then this axiom would imply that a:Meg, a:Chris, and a:Stewie are all equal, thus invalidating the unique name assumption and making the ontology inconsistent.

```
a:fatherOf \ofunc
```

11 Sample Ontology

In this section a sample ontology is outlined.

```
% define base namespace for this ontology
\ns <http://basenamespace.owl#>

% define additional custom namespaces
% to refer to concept/property/object defined in a given
    namespace use a prefix notation ns:name, e.g., owl:Thing,
     owl:Nothing
\ns ns1: <http://www.namespace1.com/ns1#>
\ns ns2: <http://www.namespace2.com/ns2#>

\begin{ontology}
% import ontologies
\import <http://www.firstontology.org/first.owl>
\import <http://www.firstontology.org/second.owl>

% now you can specify the axioms of the ontology
Person \cisa owl:Thing
Person \a{rdfs:label, "Person"}
Person \a{rdfs:comment, "This is a class for representing
    people"}
Person \cisa \candof{\d[>=1]{hasName}, \d[=1]{hasSurname},
    \dexists{hasSex}{Sex}, \d[=1]{hasAge}}

% if some already existing ontologies are required to be
    reused without cutting-and-pasting them into the current
    ontology, the input command can be used
\input{/input.txt}

% hasName, hasSurname, and hasAge are datatype properties
hasName \ddomain Person
hasName \drange xsd:string
hasName \a{rdfs:comment, "Person's name"}
hasSurname \ddomain Person
hasSurname \drange xsd:string
hasSurname \a{rdfs:comment, "Person's surname"}
hasAge \ddomain Person
hasAge \drange xsd:integer

% let us define another datatype property by enumerating
    possible values
hasHairColor \ddomain Person
hasHairColor \drange \droneof{"Blonde", "Brown", "Red"}
```

```
% if we have another property hasLastName, we can define it
    as equal to the property hasSurname
hasLastName \deq hasSurname

% sex can be defined as a concept containing two objects,
    male and female
Sex \ceq \ooneof{M, F}

M \a{rdfs:label, "Male"}
F \a{rdfs:label, "Female"}

Father \cisa Person
Child \cisa (Person \cand \oexists{hasFather}{Father})

% if we want to reuse some concept defined in the other
    namespace
Father \cisa ns1:Father
ns1:isFatherOf \oinv hasFather

% property has father is functional
hasFather \ofunc

% let's populate the ontology with some individual data
Person(john)

% to express the fact that john and johny both refer to the
    same individual one can use the object equality construct
john \ieq johny

hasName(john, "John")
hasSurname(john, "Wild")
hasAge(john, "35"[xsd:integer])
hasSex(john, M)

Child(katty)
% to express the fact that john is different from katty one
    can use the object difference construct
katty \idiff john
hasName(katty, "Katty")
hasName(katty, "Wild")
hasSex(katty, F)
hasAge(katty, "3"[xsd:integer])

% John is Katty's father
hasFather(katty, john)

\end{ontology}
```

References

[1] M. Duerst and M. Suignard. RFC 3987: Internationalized Resource Identifiers (IRIs). RFC 3987 (Proposed Standard), see `http://www.ietf.org/rfc/rfc3987.txt`, January 2005.

[2] A. Phillips and M. Davis. BCP 47 – Tags for Identifying Languages. BCP 47 Standard, see `http://www.rfc-editor.org/rfc/bcp/bcp47.txt`, September 2006.

[3] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf. Latest version available as `http://www.w3.org/TR/rdf-sparql-query/`, January 2008.