Shape Expressions Language 2.1

Final Community Group Report 8 October 2019

This version:
http://shex.io/shex-semantics-20191008/

Latest published version:
http://shex.io/shex-semantics/

Editor's draft:
https://shexspec.github.io/spec/

Previous version:
http://shex.io/shex-semantics-20181122/

Test suite:
https://github.com/shexSpec/shexTest

Bug tracker:
File a bug (open bugs)

Editors:
Eric Prud'hommeaux (W3C/MIT)
Iovka Boneva (University of Lille)
Jose Emilio Labra Gayo (University of Oviedo)
Gregg Kellogg (Spec-Ops)

Participate:
GitHub shexSpec/spec
File a bug
Commit history
Pull requests

ShEx language:
Homepage
Issues
Gitter chat

Abstract

The Shape Expressions (ShEx) language describes RDF nodes and graph structures. A node constraint describes an RDF node (IRI, blank node or literal) and a shape describes the triples involving nodes in an RDF graph. These descriptions identify predicates and their associated cardinalities and datatypes. ShEx shapes can be used to communicate data structures associated with some process or interface, generate or validate data, or drive user interfaces.

This document defines the ShEx language. See the Shape Expressions Primer for a non-normative description of ShEx.

Status of This Document

This specification was published by the Shape Expressions Community Group. It is not a W3C Standard nor is it on the W3C Standards Track. Please note that under the W3C Community Contributor License Agreement (CLA) there is a limited opt-out and other conditions apply. Learn more about W3C Community and Business Groups.

ShEx 2.1 adds IMPORTS and language tag value sets to ShEx 2.1. No tests changed as a result of this and no implementations or applications are known to have been affected.

If you wish to make comments regarding this document, please raise them as GitHub issues. There are separate interfaces for specification, language and test issues. Only send comments to public-shex@w3.org (subscribe, archives) if you are unable to raise issues on GitHub. All comments are welcome.

Table of Contents

1. Introduction

The Shape Expressions (***ShEx***) language provides a structural schema for RDF data. This can be used to document APIs or datasets, aid in development of API-conformant messages, minimize defensive programming, guide user interfaces, or anything else that involves a machine-readable description of data organization and typing requirements.

ShEx describes RDF graph [RDF11-CONCEPTS] structures as sets of potentially connected Shapes. These constrain the triples involving nodes in an RDF graph. `Node Constraints` constrain RDF nodes by constraining their node kind (IRI, blank node or Literal), enumerating permissible values in value sets, specifying their datatype, and constraining value ranges of Literals. Additionally, they constrain lexical forms of Literals, IRIs and labeled blank nodes. Shape Expressions schemas share blank nodes with the constrained RDF graphs in the same way that graphs in RDF datasets [rdf11-concepts] share blank nodes.

ShEx can be represented in JSON structures (ShExJ) or a compact syntax (ShExC). The compact syntax is intended for human consumption; the JSON structure for machine processing. This document defines ShEx in terms of ShExJ and includes a section on the ShEx Compact Syntax (ShEx).

2. Notation

The JSON [rfc7159] Syntax serves as a serializable proxy for an abstract syntax.

RDF terms are represented as JSON-LD nodes.

- IRIs are represented as a JSON string consisting of the IRI string, e.g. `"http://example.org/resource"`
- Blank nodes are represented as a JSON string composed of the concatenation of "`_:`" and a ***blank node identifier***, e.g. `"_:blank3"`
- Literals are represented as a JSON objects following the composition rules for JSON-LD values, i.e.
  - literals with the datatype `http://www.w3.org/2001/XMLSchema#string` are represented with the value property, e.g. `{ "value": "abc" }`.
  - language-tagged strings are represented with an additional language property, e.g. `{ "value": "hello world", "langague": "en-US" }`
  - datatyped literals are represented with an additional datatype property, e.g. `{ "value": "123", "datatype": "http://www.w3.org/2001/XMLSchema#integer" }`

2.1 JSON Grammar

This specification uses a ***JSON grammar*** to describe the set of JSON documents that can be interpreted as a ShEx schema. ShEx data structures are represented as ***JSON objects*** with a member with the name "type" (i.e. an object with a type attribute):

```
{ "type": "typeName", member0…n }
```

These are expressed in JSON grammar as `typeName { member* }`. RFC7159 Section 2 provides syntactic constraints for JSON — the grammar constraining those to valid ShExJ constructs is composed of:

- typeName is the name of the typed data structure. Types are referenced in the definitions of object members and in the definitions of the semantics for those data structures.
- member* is a list of zero or more terminals or references to other typeExpressions.
- A typeExpression is one of:
  - typeName — an object of corresponding type
  - array: `[ typeExpression+ ]` — an array of one or more JSON values matching the typeExpression.
  - choice: `typeExpression1 | typeExpression2 | …` — a choice between two or more typeExpressions.
- Cardinalities are represented as by the strings ?, +, * following the notation in the XML specification[XML] or {m,} to indicate a that at least m elements are required.

The following examples are excerpts from the definitions below. In the JSON notation,

```
Schema { startActs:[SemAct+]? start:shapeExpr? imports:[IRI+]? shapes:[shapeExpr+]? }
```

signifies that a Schema has four optional components called startActs, start, imports and shapes:

- startActs is a list of one or more SemAct.
- start is a shapeExpr.
- imports is a list of one or more IRI.
- shapes is an array of shapeExpr.

```
shapeExpr = ShapeOr | ShapeAnd | ShapeNot
          | NodeConstraint | Shape | ShapeExternal ;
```

signifies that a shapeExpr is one of seven object types: ShapeOr | ShapeAnd | ….

```
NodeConstraint { nodeKind:("iri" | "bnode" | "nonliteral" | "literal")? xsFacet* }
xsFacet        = stringFacet | numericFacet ;
```

signifies that a NodeConstraint has a nodeKind of one of the four literals followed by any number of xsFacet and an xsFacet is either a stringFacet or a numericFacet.

2.2 References

ShExJ is a dialect of JSON-LD [JSON-LD] and the member ***id*** is used as a node identifier. An object may be represented inline or referenced by its id which may be either a blank node or an IRI.

```
ShapeOr         { id:shapeExprLabel? shapeExprs:[shapeExpr{2,}] }
shapeExprLabel  = IRIREF | BNODE ;
EachOf          { id:tripleExprLabel? expressions:[tripleExpr{2,}] ... }
tripleExprLabel = IRI | BNODE ;
```

The JSON structure may include references to shape expressions and triple expressions:

```
shapeExpr     = ShapeOr | ... | shapeExprRef ;
shapeExprRef  = shapeExprLabel ;
tripleExpr    = EachOf | ... | tripleExprRef ;
tripleExprRef = tripleExprLabel ;
```

An object with a circular reference must be referenced by an id. This example uses a nested shape reference on a value expression (defined below).

```
{ "type": "Schema", "shapes": [
    { "id": "http://schema.example/#IssueShape",
      "type": "Shape", "expression": {
        "type": "TripleConstraint", "predicate": "http://schema.example/#related",
        "valueExpr": "http://schema.example/#IssueShape", "min": 0 } } ] }
```

Not captured in this JSON syntax definition is the rule that every shapeExpr nested in a schema's shapes must have an id and no other shapeExpr may have an id. The JSON syntax definitition simplifies this by adding id:shapeExprLabel? to every shapeExpr. This example includes a nested shape. Nested shapes are not permitted to have ids.

```
{ "type": "Schema", "shapes": [
    { "id": "http://schema.example/#IssueShape",
      "type": "Shape", "expression": {
        "type": "TripleConstraint", "predicate": "http://schema.example/#submittedBy",
        "valueExpr": {
          "type": "Shape", "expression": {
            "type": "TripleConstraint", "predicate": "http://schema.example/#name",
            "valueExpr": {
              "type": "NodeConstraint", "nodeKind": "literal"
            } } } } } ] }
```

> NOTE
>
> The shape expressions in a schema are expressed as a list of shape expressions with id attributes because JSON-LD 1.0 has no provision for defining as JSON object as representing a map of URL→object. Drafts of JSON-LD 1.1 include this expressivity and, once JSON-LD 1.1 exits Candidate Recommendation, future versions of ShExJ will likely adopt it while requiring backard-compatibility with the list of shape expressions with id attributes.

2.3 Document style

JSON examples are rendered in a .json CSS style. Partial examples include ranges in a *.comment* CSS style to indicate text which would be substituted in a complete example. For example { "type": "ShapeAnd", "shapeExprs": [SE_1, ...] } indicates that both SE_1 and ... would be substituted in a complete example.

In javascript-enabled browsers, schemas with a json button can be converted between the JSON representation and the compact syntax by clicking the button. The button text indicates the currently shown representation. Selecting the example and pressing "j" or "c" converts the example to the JSON (ShExJ) or compact form (ShExC). Pressing "shift J" or "shift C" converts all such examples to ShExJ or ShExC.

2.4 Graph access

The validation process defined in this document relies on matching triple patterns in the form `(subject, predicate, object)` where each position may be supplied by a constant, a previously defined term, or the underscore "_", which represents a previously undefined element or wildcard. This corresponds to a ***SPARQL Triple Pattern*** where each "_" is replaced by a unique blank node. Matching such a triple pattern against a graph is defined by SPARQL Basic Graph Pattern Matching (BGP) with a BGP containing only that triple pattern.

2.5 Validation process

ShEx validation is defined in this document by the isValid function. This process takes as input a shapes schema, an RDF graph, and a ***fixed ShapeMap*** (abbreviated as "ShapeMap" in this document). ShEx validation results may be reported as a ***result ShapeMap*** [shape-map]. For illustration purposes in this specification, both the fixed ShapeMap input and the result ShapeMap output are represented in a table with four columns: node: a ShapeMap nodeSelector, shape: a ShapeMap shapeLabel, result: "pass" (for "conformant" status) or "fail" ("nonconformant" status), and an optional reason: an informal, human-readable explanation.

node	shape	result	reason
<node1>	<Shape1>	pass	
<node2>	<Shape1>	fail	no `ex:state` supplied.

3. Terminology

Shape expressions are defined using terms from RDF semantics [rdf11-mt]:

- ***Node***: one of ***IRI***, ***blank node***, ***Literal***
- ***Graph***: a set of ***Triples*** of (***subject***, ***predicate***, ***object***)

This specification makes use of the following namespaces:

foaf:
`http://xmlns.com/foaf/0.1/`

rdf:
`http://www.w3.org/1999/02/22-rdf-syntax-ns#`

rdfs:
`http://www.w3.org/2000/01/rdf-schema#`

shex:
`http://www.w3.org/ns/shex#`

xsd:
`http://www.w3.org/2001/XMLSchema#`

The following functions access the elements of an RDF graph G containing a node n:

- ***arcsOut***(G, n) is the set of triples in a graph G with subject n.
- ***predicatesOut***(G, n) is the set of predicates in arcsOut(G, n).
- ***arcsIn***(G, n) is the set of triples in a graph G with object n.
- ***predicatesIn***(G, n) is the set of predicates in arcsIn(G, n).
- ***neigh***(G, n) is the neighbourhood of the node n in the graph G.
  neigh(G, n) = arcsOut(G, n) ∪ arcsIn(G, n)
- ***predicates***(G, n) is the set of predicates in neigh(G, n).
  predicates(G, n) = predicatesOut(G, n) ∪ predicatesIn(G, n)

Consider the RDF graph G represented in Turtle:

```
PREFIX ex: http://schema.example/#
PREFIX inst: http://inst.example/#
PREFIX foaf: http://xmlns.com/foaf/
PREFIX xsd: http://www.w3.org/2001/XMLSchema#

inst:Issue1
    ex:state      ex:unassigned ;
    ex:reportedBy _:User2 .

_:User2
    foaf:name     "Bob Smith" ;
    foaf:mbox     <mailto:bob@example.org> .
```

There are two arcs out of `_:User2`; arcsOut(G, `_:User2`):

```
_:User2  foaf:name  "Bob Smith" .
_:User2  foaf:mbox  <mailto:bob@example.org> .
```

There is one arc into `_:User2`; arcsIn(G, `_:User2`):

```
inst:Issue1 ex:reportedBy  _:User2 .
```

There are three arcs in the neighbourhood of `_:User2` set, neigh(G, `_:User2`):

```
_:User2  foaf:name  "Bob Smith" .
_:User2  foaf:mbox  <mailto:bob@example.org> .
inst:Issue1 ex:reportedBy  _:User2 .
```

4. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, and *SHOULD NOT* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Conformance criteria are relevant to authors and authoring tool implementers. As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

- A ShExC document complies with this specification if it conforms to the grammar described in § 6. ShEx Compact syntax (ShExC) resulting in a valid ShExJ document.
- A ShExJ document complies with this specification if it is a valid JSON-LD document [JSON-LD], and conforms to the ShExJ syntax, as described in § A. ShEx JSON Syntax (ShExJ).
- JSON documents can be interpreted as ShExJ by following the normative statements in Section 4.8 Interpreting JSON as JSON-LD in [JSON-LD].
- Other RDF documents comply with this specification, if they are graph isomorphic[rdf11-concepts] to the RDF interpretation of some ShExJ document.

5. The Shape Expressions Language

A Shape Expressions (ShEx) schema is a collection of labeled Shapes and Node Constraints. These can be used to describe or test nodes in RDF graphs. ShEx does not prescribe a language for associating nodes with shapes but several approaches are described in the ShEx Primer.

5.1 Shapes Schema

A shapes schema is captured in a ***Schema*** object:

```
Schema { startActs:[SemAct+]? start:shapeExpr? imports:[IRI+]? shapes:[shapeExpr+]? }
```

where shapes is a mapping from shape label to shape expression.

```
{ "type": "Schema", "shapes": [                                    json
  { "id": "http://schema.example/#IssueShape", … },
  { "id": "_:UserShape", … },
  { "id": "http://schema.example/#EmployeeShape", … } ] }
```

5.2 Validation Definition

isValid: For a graph G, a schema Sch and a fixed ShapeMap ism, `isValid(G, Sch, ism)` indicates that for every RDFnode/shapeLabel pair (n, sl) in ism, the node n satisfies the shape expression identified by sl. The latter is captured by the expression `satisfies(n, s, G, Sch, completeTyping(G, Sch))`. The function `satisfies` is defined for every kind of shape expression.

The validation of an RDF graph G against a ShEx schema Sch is based on the existence of `completeTyping(G, Sch)`. For an RDF graph G and a shapes schema Sch, a ***typing*** is a set of pairs of the form (n, s) where n is a node in G and s is a Shape that appears in some shape expression in the shapes mapping of Sch. A ***correct typing*** is a typing such that for every RDFnode/shape pair (n,s) in typing, `matchesShape(n, s, G, Sch, typing)` holds. `completeTyping(G, Sch)` is a unique correct typing that exists for every graph and every ShEx schema that satisfies the schema requirements.

completeTyping: the definition of `completeTyping(G, Sch)` is based on a stratification of Sch. The number of ***strata*** of Sch is the number of maximal strongly connected components of the dependency graph of Sch. A ***stratification*** of a schema Sch with k strata is a function `stratum` that associates with every Shape in Sch a natural number between 1 and k such that:

- If s1 and s2 belong to the same maximal strongly connected component, then `stratum(s1)` = `stratum(s2)`.
- If there is a reference from s1 to s2 and s1 and s2 do not belong to the same maximal strongly connected component, then `stratum(s2)` < `stratum(s1)`.

The existence of a stratification for every schema is guaranteed by the negation requirement.

Given a stratification `stratum` of Sch with k strata, define inductively the series of k typings `completeTypingOn(1, G, Sch)` … `completeTypingOn(k, G, Sch)`.

- `completeTypingOn(1, G, Sch)` is the union of all correct typings that contain only RDFnode/shape pairs (n,s) with `stratum(s)` = 1;
- for every i between 2 and k, `completeTypingOn(i, G, Sch)` is the union of all correct typings that:
  - contain only RDFnode/shape pairs (n,s) with `stratum(s)` ≤ i
  - are equal to `completeTypingOn(i-1, G, Sch)` when restricted to their RDFnode/shape pairs (n1,s1) for which `stratum(s1)` < i.

Then `completeTyping(G, Sch)` = `completeTypingOn(k, G, Sch)`.

NOTE
The definition on strogly connected component and maximal strongly connected component of a graph can be found on Wikipedia https://en.wikipedia.org/wiki/Strongly_connected_component.

NOTE

The schema Sch might have several different stratifications but `completeTyping(G, Sch)` is the same for all these stratifications. This property is reminiscent of the use of stratified negiation in Datalog.

In order to decide isValid(Sch, G, m), it is sufficient to compute only a portion of completeTyping using an appropriate algorithm.

NOTE

Popular methods for constructing the input fixed ShapeMaps can be found on https://www.w3.org/2001/sw/wiki/ShEx/ShapeMap.

5.3 Shape Expressions

A ***shape expression*** is composed of four kinds of objects combined with the algebraic operators And, Or and Not:

- a node constraint (NodeConstraint) that defines the set of allowed values of a node. These include specification of RDF node kind, literal datatype, XML String and numeric facets and enumeration of value sets.
- a shape constraint (Shape) that defines a constraint on the allowed neighbourhood of a node, that is, the allowed triples that contain this node as subject or object.
- an external shape (ShapeExternal) which is an extension mechanism to externally define e.g. functional shapes or prohibitively large value sets.
- a shape reference (shapeExprLabel) identifies another shape in the schema or an imported schema.

5.3.1 JSON Syntax

```
shapeExpr = ShapeOr | ShapeAnd | ShapeNot | NodeConstraint | Shape | ShapeExternal | shapeExprRef ;
ShapeOr { id:shapeExprLabel? shapeExprs:[shapeExpr{2,}] }
ShapeAnd { id:shapeExprLabel? shapeExprs:[shapeExpr{2,}] }
ShapeNot { id:shapeExprLabel? shapeExpr:shapeExpr }
ShapeExternal { id:shapeExprLabel? }
shapeExprRef = shapeExprLabel ;
shapeExprLabel = IRIREF | BNODE ;
```

Examples of shape expressions:

```
{ "type": "Shape", … }
```
json

```
{ "type": "ShapeAnd", "shapeExprs": [
  { "type": "NodeConstraint", "nodeKind": "iri" },
  { "type": "ShapeOr", "shapeExprs": [
    "http://schema.example/#IssueShape",
    { "type": "ShapeNot", "shapeExpr": { "type": "Shape", … } }
    ] } ] }
```
json

In this ShapeOr's shapeExprs, "http://schema.example/#IssueShape" is a reference to the shape expression with the id "http://schema.example/#IssueShape".

5.3.2 Semantics

satisfies: The expression `satisfies(n, se, G, Sch, t)` indicates that a node n and a graph G satisfy a shape expression se with typing t for schema Sch.
notSatisfies: Conversely, `notSatisfies(n, se, G, Sch, t)` indicates that n and G do not satisfy se with the given typing t.

`satisfies(n, se, G,Sch, t)` is true if and only if:

- se is a NodeConstraint and `satisfies2(n, se)` as described below in Node Constraints. Note that testing if a node satisfies a node constraint does not require a graph or typing.
- se is a Shape and (n, se) belongs to t.
- se is a ShapeOr and there is some shape expression se2 in se.shapeExprs such that `satisfies(n, se2, G, Sch, t)`.
- se is a ShapeAnd and for every shape expression se2 in se.shapeExprs, `satisfies(n, se2, G, Sch, t)`.
- se is a ShapeNot and for the shape expression se2 at se.shapeExpr, `notSatisfies(n, se2, G, Sch, t)`.
- se is a ShapeExternal and implementation-specific mechansims not defined in this specification indicate success.
- se is a shapeExprRef and `satisfies(n, se2, G, Sch, t)` where se2 is the shape expression having se as id.

Given the three shape expressions SE_1, SE_2, SE_3 in a Schema Sch, such that:

- satisfies(n, SE_1, G, Sch, m)
- satisfies(n, SE_2, G, Sch, m)
- notSatisfies(n, SE_3, G, Sch, m)

the following hold:

- satisfies(n,
  `{ "type": "ShapeAnd", "shapeExprs": [ SE1, SE2 ] }`,
  G, Sch, m)
- satisfies(n,
  `{ "type": "ShapeOr", "shapeExprs": [ SE1, SE2, SE3 ] }`,
  G, Sch, m)
- notSatisfies(n,
  ```
  { "type": "ShapeNot", "shapeExpr": {
      { "type": "ShapeOr", "shapeExprs": [
          SE1,
          { "type": "ShapeAnd", "shapeExprs": [ SE2, SE3 ] }
        ] }
    } }
  ```
  ,
  G, Sch, m)

If Sch's shapes maps "http://schema.example/#shape1" to SE_1 then the following holds:

- satisfies(n,
  `http://schema.example/#shape1"`,
  G, Sch, m)

5.4 Node Constraints

NodeConstraint	{ id:shapeExprLabel? nodeKind:("iri" | "bnode" | "nonliteral" | "literal")? datatype:IRIREF? xsFacet* values:[valueSetValue+]? }
xsFacet	= stringFacet | numericFacet ;
stringFacet	= (length|minlength|maxlength):INTEGER | pattern:STRING flags:STRING? ;
numericFacet	= (mininclusive|minexclusive|maxinclusive|maxexclusive):numericLiteral
	|(totaldigits|fractiondigits):INTEGER ;
numericLiteral	= INTEGER | DECIMAL | DOUBLE ;
valueSetValue	= objectValue | IriStem | IriStemRange | LiteralStem | LiteralStemRange | Language | LanguageStem | LanguageStemRange ;
objectValue	= IRIREF | ObjectLiteral ;
ObjectLiteral	{ value:STRING language:STRING? type:STRING? }
IriStem	{ stem:IRIREF }
IriStemRange	{ stem:(IRIREF | Wildcard) exclusions:[IRIREF|IriStem +] }
LiteralStem	{ stem:STRING }
LiteralStemRange	{ stem:(STRING | Wildcard) exclusions:[STRING|LiteralStem +] }
Language	{ languageTag:LANGTAG }
LanguageStem	{ stem:LANGTAG }
LanguageStemRange	{ stem:(LANGTAG | Wildcard) exclusions:[LANGTAG|LanguageStem +] }
Wildcard	{ /* empty */ }

5.4.1 Semantics

For a node n and constraint nc, `satisfies2(n, nc)` if and only if for every nodeKind, datatype, xsFacet and values constraint value v present in nc `nodeSatisfies(n, v)`. The following sections define `nodeSatisfies` for each of these types of constraints:

- Node Kind Constraints
- Datatype Constraints
- XML Schema String Facet Constraints
- XML Schema Numeric Facet Constraints
- Values Constraints

5.4.2 Node Kind Constraints

For a node n and constraint value v, `nodeSatisfies(n, v)` if:

- v = "iri" and n is an IRI.
- v = "bnode" and n is a blank node.
- v = "literal" and n is a Literal.
- v = "nonliteral" and n is an IRI or blank node.

> NODE KIND EXAMPLE 1
>
> The following examples use a TripleConstraint object described later in the document. The

```
{ "type": "Schema", "shapes": [                                                    json
  { "id": "http://schema.example/#IssueShape",
    "type": "Shape", "expression": {
      "type": "TripleConstraint", "predicate": "http://schema.example/#state",
      "valueExpr": { "type": "NodeConstraint", "nodeKind": "iri" } } } ] }
```

```
<issue1> ex:state ex:HunkyDory .
<issue2> ex:taste ex:GoodEnough .
<issue3> ex:state "just fine" .
```

node	shape	result	reason
<issue1>	<IssueShape>	pass	
<issue2>	<IssueShape>	fail	expected 1 `ex:state` property.
<issue3>	<IssueShape>	fail	`ex:state` expected to be an IRI, literal found.

> Note that <issue2> fails not because of a nodeKind violation but instead because of a Cardinality violation described below.

5.4.3 Datatype Constraints

For a node n and constraint value v, `nodeSatisfies(n, v)` if n is an Literal with the datatype v and, if v is in the set of SPARQL operand data types[sparql11-query], an XML schema string with a value of the lexical form of n can be cast to the target type v per XPath Functions 3.1 section 19 Casting[xpath-functions]. The lexical form and numeric value (where

applicable) of all datatypes required by SPARQL XPath Constructor Functions *MUST* be tested for conformance with the corresponding XML Schema form. ShEx extensions *MAY* add support for other datatypes.

DATATYPE EXAMPLE 1

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#IssueShape",
    "type": "Shape", "expression": {
      "type": "TripleConstraint", "predicate": "http://schema.example/#submittedOn",
      "valueExpr": {
        "type": "NodeConstraint",
        "datatype": "http://www.w3.org/2001/XMLSchema#date"
      } } } ] }
```

```
<issue1> ex:submittedOn "2016-07-08"^^xsd:date .
<issue2> ex:submittedOn "2016-07-08T01:23:45Z"^^xsd:dateTime .
<issue3> ex:submittedOn "2016-07"^^xsd:date .
```

node	shape	result	reason
<issue1>	<IssueShape>	pass	
<issue2>	<IssueShape>	fail	`ex:submittedOn` expected to be an `xsd:date`, `xsd:dateTime` found.
<issue3>	<IssueShape>	fail	`2016-07` is not a valid `xsd:date`.

NOTE

In RDF 1.1, ***language-tagged strings***[rdf11-concepts] have the datatype `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`.

RDF 1.0 included RDF literals with no datatype or language tag. These are called "simple literals" in SPARQL11[sparql11-query]. In RDF 1.1, these literals have the datatype `http://www.w3.org/2001/XMLSchema#string`.

DATATYPE EXAMPLE 2

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#IssueShape",
    "type": "Shape", "expression": {
      "type": "TripleConstraint",
      "predicate": "http://www.w3.org/2000/01/rdf-schema#label",
      "valueExpr": {
        "type": "NodeConstraint",
        "datatype": "http://www.w3.org/1999/02/22-rdf-syntax-ns#langString"
      } } } ] }
```

```
<issue3> rdfs:label "emits dense black smoke"@en .
<issue4> rdfs:label "unexpected odor" .
```

node	shape	result	reason
<issue3>	<IssueShape>	pass	
<issue4>	<IssueShape>	fail	`rdfs:label` expected to be an `rdf:langString`, `xsd:string` found.

5.4.4 XML Schema String Facet Constraints

String facet constraints apply to the lexical form of the RDF Literals and IRIs and blank node identifiers (see note below regarding access to blank node identifiers).
Let lex =

- if the value n is an RDF Literal, the lexical form of the literal (see [rdf11-concepts] section 3.3 Literals).
- if the value n is an IRI, the IRI string (see [rdf11-concepts] section 3.2 IRIs).
- if the value n is a blank node, the blank node identifier (see [rdf11-concepts] section 3.4 Blank Nodes).

Let len = the number of unicode codepoints in lex
For a node n and constraint value v, `nodeSatisfies(n, v)`:

- for "`length`" constraints, v = len,
- for "`minlength`" constraints, v >= len,
- for "`maxlength`" constraints, v <= len,
- for "`pattern`" constraints, v is unescaped into a valid XPath 3.1 regular expression[xpath-functions-31] re and invoking `fn:matches(lex, re)` returns `fn:true`. If the flags parameter is present, it is passed as a third argument to `fn:matches`. The pattern may have XPath 3.1 regular expression escape sequences per the modified production [10] in section 5.6.1.1 as well as numeric escape sequences of the form 'u' HEX HEX HEX HEX or 'U' HEX HEX HEX HEX HEX HEX HEX HEX. Unescaping replaces numeric escape sequences with the corresponding unicode codepoint.

STRING FACETS EXAMPLE 1

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#IssueShape",
    "type": "Shape", "expression": {
      "type": "TripleConstraint",
      "predicate": "http://schema.example/#submittedBy",
      "valueExpr": { "type": "NodeConstraint", "minlength": 10 } } } ] }
```

```
<issue1> ex:submittedBy <http://a.example/bob> . # 20 characters
<issue2> ex:submittedBy "Bob" . # 3 characters
```

node	shape	result	reason
<issue1>	<IssueShape>	pass	
<issue2>	<IssueShape>	fail	`ex:submittedBy` expected to be >= 10 characters, 3 characters found.

NOTE

Access to blank node identifiers may be impossible or unadvisable for many use cases. For instance, the SPARQL Query and SPARQL Update languages treat blank nodes in the query, labeled or otherwise, as variables. Lexical constraints on blank node identifiers can only be implemented in systems which preserve such labels on data import.

STRING FACETS EXAMPLE 2

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#IssueShape",
    "type": "Shape", "expression": {
      "type": "TripleConstraint",
      "predicate": "http://schema.example/#submittedBy",
      "valueExpr": { "type": "NodeConstraint",
                     "pattern": "genuser[0-9]+", "flags": "i" }
} } ] }
```
json

```
<issue6> ex:submittedBy _:genUser218 .
<issue7> ex:submittedBy _:genContact817 .
```

node	shape	result	reason
<issue6>	<IssueShape>	pass	
<issue7>	<IssueShape>	fail	`_:genContact817` expected to match `genuser[0-9]+`.

When expressed as JSON strings, regular expressions are subject to the JSON string escaping rules.

STRING FACETS EXAMPLE 3

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#ProductShape",
    "type": "Shape", "expression": {
      "type": "TripleConstraint",
      "predicate": "http://schema.example/#trademark",
      "valueExpr": { "type": "NodeConstraint",
                     "pattern": "^/\\t\\\\uD835\uDCB8\\?$" }
} } ] }
```
json

```
<product6> ex:trademark "       \\c?" .
<product7> ex:trademark "\t\\\U0001D4B8?" . # Turtle literals have escape characters [tbnrf"'\].
<product8> ex:trademark "\t\\\U0001D4B8?" .
```

node	shape	result	reason
<product6>	<ProductShape>	pass	
<product7>	<ProductShape>	pass	
<product8>	<ProductShape>	fail	found "`\U0001D4B8`" instead of "*c*" (codepoint U+1D4B8).

5.4.5 XML Schema Numeric Facet Constraints

Numeric facet constraints apply to the numeric value of RDF Literals with datatypes listed in SPARQL 1.1 Operand Data Types[sparql11-query]. Numeric constraints on non-numeric values fail. `totaldigits` and `fractiondigits` constraints on values not derived from `xsd:decimal` fail.

Let num be the numeric value of n.
For a node n and constraint value v, `nodeSatisfies(n, v)`:

- for "`mininclusive`" constraints, v <= num,
- for "`minexclusive`" constraints, v < num,

- for "`maxinclusive`" constraints, v >= num,
- for "`maxexclusive`" constraints, v > num,
- for "`totaldigits`" constraints, v is less than or equals the number of digits in the XML Schema canonical form[xmlschema-2] of the value of n,
- for "`fractiondigits`" constraints, v is less than or equals the number of digits to the right of the decimal place in the XML Schema canonical form[xmlschema-2] of the value of n, ignoring trailing zeros.

The operators <=, <, >= and > are evaluated after performing numeric type promotion[xpath20].

NUMERIC FACETS EXAMPLE 1

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#IssueShape",
    "type": "Shape", "expression": {
      "type": "TripleConstraint",
      "predicate": "http://schema.example/#confirmations",
      "valueExpr": { "type": "NodeConstraint", "mininclusive": 1 } } } ] }
```

```
<issue1> ex:confirmations 1 .
<issue2> ex:confirmations 2^^xsd:byte .
<issue3> ex:confirmations 0 .
<issue4> ex:confirmations "ii"^^ex:romanNumeral .
```

node	shape	result	reason
<issue1>	<IssueShape>	pass	
<issue2>	<IssueShape>	pass	
<issue3>	<IssueShape>	fail	`0` is less than `1`.
<issue4>	<IssueShape>	fail	`ex:romanNumeral` is not a numeric datatype.

5.4.6 Values Constraint

The `nodeSatisfies` semantics for NodeConstraint values depends on a `nodeIn` function defined below.

For a node n and constraint value v, `nodeSatisfies(n, v)` if n matches some valueSetValue vsv in v. A term matches a valueSetValue if:

- vsv is an objectValue and n = vsv.
- vsv is a Language with languageTag lt and n is a language-tagged string with a language tag l and l = lt.
- vsv is a IriStem, LiteralStem or LanguageStem with stem st and nodeIn(n, st).
- vsv is a IriStemRange, LiteralStemRange or LanguageStemRange with stem st and exclusions excls and nodeIn(n, st) and there is no x in excls such that nodeIn(n, excl).
- vsv is a Wildcard with exclusions excls and there is no x in excls such that nodeIn(n, excl).

nodeIn: asserts that an RDF node n is equal to an RDF term s or is in a set defined by a IriStem, LiteralStem or LanguageStem.
The expression `nodeIn(n, s)` is satisfied if:

- n = s.
- s is a IriStem, LiteralStem or LanguageStem with stem st and:
  - s is a IriStem and n is an IRI and fn:starts-with(n, st).

- s is a LiteralStem and n is an RDF Literal with a lexical value l and fn:starts-with(l, st).
- s is a LanguageStem, n is a language-tagged string with a language tag l, st is a basic language range per Matching of Language Tags [rfc4647] section 2.1 and l matches st per the basic filtering scheme defined in [rfc4647] section 3.3.1.

VALUES CONSTRAINT EXAMPLE 1

NoActionIssueShape requires a state of Resolved or Rejected:

```
{ "type": "Schema", "shapes": [                    json
  { "id": "http://schema.example/#NoActionIssueShape",
    "type": "Shape", "expression": {
      "type": "TripleConstraint",
      "predicate": "http://schema.example/#state",
      "valueExpr": {
        "type": "NodeConstraint", "values": [
          "http://schema.example/#Resolved",
          "http://schema.example/#Rejected" ] } } } ] }
```

```
<issue1> ex:state ex:Resolved .
<issue2> ex:state ex:Unresolved .
```

node	shape	result	reason
<issue1>	<NoActionIssueShape>	pass	
<issue2>	<NoActionIssueShape>	fail	`ex:state` expected to be `ex:Resolved` or `ex:Rejected`, `ex:Unresolved` found.

VALUES CONSTRAINT EXAMPLE 2

An employee must have an email address that is the string "N/A" or starts with "engineering-" or "sales-" but not "sales-contacts" or "sales-interns":

```
{ "type": "Schema", "shapes": [                                                    json
  { "id": "http://schema.example/#EmployeeShape",
    "type": "Shape", "expression": {
      "type": "TripleConstraint",
      "predicate": "http://xmlns.com/foaf/0.1/mbox",
      "valueExpr": {
        "type": "NodeConstraint", "values": [
          {"value": "N/A"},
          { "type": "IriStem", "stem": "mailto:engineering-" },
          { "type": "IriStemRange", "stem": "mailto:sales-", "exclusions": [
              { "type": "IriStem", "stem": "mailto:sales-contacts" },
              { "type": "IriStem", "stem": "mailto:sales-interns" }
            ] }
        ] } } } ] }
```

```
<issue3> foaf:mbox "N/A" .
<issue4> foaf:mbox <mailto:engineering-2112@a.example> .
<issue5> foaf:mbox <mailto:sales-835@a.example> .
<issue6> foaf:mbox "missing" .
<issue7> foaf:mbox <mailto:sales-contacts-999@a.example> .
```

node	shape	result	reason
<issue3>	<EmployeeShape>	pass	
<issue4>	<EmployeeShape>	pass	
<issue5>	<EmployeeShape>	pass	
<issue6>	<EmployeeShape>	fail	`"missing"` is not in value set.
<issue7>	<EmployeeShape>	fail	`<mailto:sales-contacts-999@a.example>` is excluded.

VALUES CONSTRAINT EXAMPLE 3

An employee must not have an email address that starts with "engineering-" or "sales-":

json

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#EmployeeShape",
    "type": "Shape", "expression": {
      "type": "TripleConstraint",
      "predicate": "http://xmlns.com/foaf/0.1/mbox",
      "valueExpr": {
        "type": "NodeConstraint", "values": [
          { "type": "IriStemRange", "stem": {"type": "Wildcard"},
            "exclusions": [
              { "type": "IriStem", "stem": "mailto:engineering-" },
              { "type": "IriStem", "stem": "mailto:sales-" }
            ] }
        ] } } } ] }
```

```
<issue8> foaf:mbox 123 .
<issue9> foaf:mbox <mailto:core-engineering-2112@a.example> .
<issue10> foaf:mbox <mailto:engineering-2112@a.example> .
```

node	shape	result	reason
<issue8>	<EmployeeShape>	pass	
<issue9>	<EmployeeShape>	pass	
<issue10>	<EmployeeShape>	fail	<mailto:engineering-2112@a.example> is excluded.

A value set can have a single value in it. This is used to indicate that a specific value is required, e.g. that an ex:state must be equal to <http://schema.example/#Resolved> or the rdf:type of some node must be foaf:Person.

5.5 Shapes and Triple Expressions

Triple expressions are used for defining patterns composed of triple constraints. Shapes associate triple expressions with flags indicating whether triples match if they do not correspond to triple constraints in the triple expression. A ***triple expression*** is composed of TripleConstraint and tripleExprRef objects composed with grouping and choice operators.

5.5.1 JSON Syntax

```
Shape { id:shapeExprLabel? closed:BOOL? extra:[IRIREF+]? expression:tripleExpr? semActs:[SemAct+]?
        annotations:[Annotation+]? }
tripleExpr = EachOf | OneOf | TripleConstraint | tripleExprRef ;
EachOf { id:tripleExprLabel? expressions:[tripleExpr{2,}] min:INTEGER? max:INTEGER? semActs:[SemAct+]?
        annotations:[Annotation+]? }
OneOf { id:tripleExprLabel? expressions:[tripleExpr{2,}] min:INTEGER? max:INTEGER? semActs:[SemAct+]?
        annotations:[Annotation+]? }
TripleConstraint { id:tripleExprLabel? inverse:BOOL? predicate:IRIREF valueExpr:shapeExpr? min:INTEGER?
        max:INTEGER? semActs:[SemAct+]? annotations:[Annotation+]? }
tripleExprRef = tripleExprLabel ;
tripleExprLabel = IRIREF | BNODE ;
```

5.5.2 Semantics

The semantics of the `matchesShape` function are based on the `matches` function defined below. For a node n, shape S, graph G, a ShExSchema Sch, and typing m, `matchesShape(n, S, G, Sch, m)` if and only if:

- `neigh(G, n)` can be partitioned into two sets matched and remainder such that `matches(matched, expression, m)`. If expression is absent, remainder = `neigh(G, n)`.
  Let **outs** be the arcsOut in remainder: outs = remainder ∩ `arcsOut(G, n)`.
  Let **matchables** be the triples in outs whose predicate appears in a TripleConstraint in expression. If expression is absent, matchables = `Ø (the empty set)`.
  The complexity of partitioning is described briefly in the ShEx2 Primer.
- There is no triple in **matchables** which matches a TripleConstraint in expression.
  Let **unmatchables** be the triples in outs which are not in matchables. matchables ∪ unmatchables = outs.
- There is no triple in matchables whose predicate does not appear in extra.
- closed is false or unmatchables is empty.

matches: asserts that a triple expression is matched by a set of triples that come from the neighbourhood of a node in an RDF graph. The expression `matches(T, expr, m)` indicates that a set of triples T can satisfy these rules:

- expr has semActs and `matches(T, expr, m)` by the remaining rules in this list and the evaluation of semActs succeeds according to the section below on Semantic Actions.

  matches(T,
  `{ "type": "OneOf", "shapeExprs": [`te_1`, `te_2`, …], "min": 2, "max": 3, "semActs": [`$SemAct_1$`, `$SemAct_2$`, …] }` json ,
  m)
  evaluates as:
  matches(T,
  `{ "type": "OneOf", "shapeExprs": [`te_1`, `te_2`, …], "min": 2, "max": 3 }` json ,
  m)
  and semActsSatisfied([$SemAct_1$, $SemAct_2$, …])

- expr has a ***cardinality*** of min and/or max not equal to 1, where a max of -1 is treated as unbounded, and T can be partitioned into k subsets T_1, T_2,…T_k such that min ≤ k ≤ max and for each T_n, `matches(`T_n`, expr, m)` by the remaining rules in this list.

  matches(T,
  `{ "type": "OneOf", "shapeExprs": [`te_1`, `te_2`, …], "min": 2, "max": 3 }` json ,
  m)
  evaluates as:
  Let e = `{ "type": "OneOf", "shapeExprs": [`te_1`, `te_2`, …] }` json
  (matches(T_1, e, m) and matches(T_2, e, m)
  and T = T_1 ∪ T_2)
  or
  (matches(T_1, e, m) and matches(T_2, e, m) and matches(T_3, e, m)
  and T = T_1 ∪ T_2 ∪ T_3)

- expr is a OneOf and there is some shape expression se2 in shapeExprs such that `matches(T, se2, m)`.

matches(T,
```
{ "type": "OneOf", "shapeExprs": [
  { "type": "EachOf", "shapeExprs": [te3, te4, …] },
  { "type": "TripleExpression", "min": 1, "max": -1,
    "predicate": "http://xmlns.com/foaf/0.1/name" }
] }
```
json
,
m)
evaluates as:
matches(T,
```
{ "type": "EachOf", "shapeExprs": [te3, te4, …] }
```
json
,
m)
or matches(T,
```
{ "type": "TripleExpression", "min": 1, "max": -1,
   "predicate": "http://xmlns.com/foaf/0.1/name" }
```
json
,
m)

- expr is an EachOf and there is some partition of T into T_1, T_2,… such that for every expression $expr_1$, $expr_2$,… in shapeExprs, `matches(`T_n`,` $expr_n$`, m)`.

matches(T,
```
{ "type": "EachOf", "shapeExprs": [
  { "type": "TripleExpression",
    "predicate": "http://xmlns.com/foaf/0.1/givenName" },
  { "type": "TripleExpression",
    "predicate": "http://xmlns.com/foaf/0.1/familyName" }
] }
```
json
,
m)
evaluates as:
matches(T_1,
```
{ "type": "TripleExpression",
   "predicate": "http://xmlns.com/foaf/0.1/givenName" }
```
json
,
m)
and matches(T_2,
```
{ "type": "TripleExpression",
   "predicate": "http://xmlns.com/foaf/0.1/familyName" }
```
json
,
m)
and $T = T_1 \cup T_2$

- expr is a TripleConstraint and:
  - T is a set of one triple.
    Let **t** be the sole triple in T.
  - t's predicate equals expr's predicate.
    Let **value** be t's subject if inverse is true, else t's object.
  - if inverse is true, t is in arcsIn, else t is in arcsOut.
  - either
    - expr has no valueExpr

matches(T,
```
{ "type": "TripleExpression",                                    json
  "predicate": "http://xmlns.com/foaf/0.1/givenName" }
```
,
m)

holds if
- T has exactly one triple t.
- t has the predicate "http://xmlns.com/foaf/0.1/givenName"

- or satisfies(value, valueExpr, G, Sch, m).

matches(T,
```
{ "type": "TripleConstraint", "inverse": true,                   json
  "predicate": "http://purl.org/dc/elements/1.1/author",
  "valueExpr": "http://schema.example/#IssueShape" }
```
,
m)

holds if
- T has exactly one triple t.
- t has the predicate "http://purl.org/dc/elements/1.1/author"
- t has a subject n2
- The schema's shapes maps "http://schema.example/#IssueShape" to se2
- satisfies(n2, se2, G, Sch, m)

- expr is a tripleExprRef and satisfies(value, tripleExprWithId(tripleExprRef), G, Sch, Sch, m).
The tripleExprWithId function is defined in Triple Expression Reference Requirement below.

For the schema
```
{ "type": "Schema", "shapes": [                                  json
  { "id": "http://schema.example/#EmployeeShape",
    "type": "Shape", "expression": {
      "type": "EachOf", "expressions": [
        "http://schema.example/#nameExpr",
        { "type": "TripleConstraint",
          "predicate": "http://schema.example/#empID",
          "valueExpr": { "type": "NodeConstraint",
            "datatype": "http://www.w3.org/2001/XMLSchema#integer" } } ] } },
  { "id": "http://schema.example/#PersonShape",
    "type": "Shape", "expression": {
      "id": "http://schema.example/#nameExpr",
      "type": "TripleConstraint",
      "predicate": "http://xmlns.com/foaf/0.1/name" } } ] }
```
matches(T,
```
"http://schema.example/#PersonShape"
```
,
m)

holds if
- The schema has a shape se2 with the id "http://schema.example/#PersonShape"
- satisfies(n, se2, G, Sch, m)

5.6 ShEx Import

The presence of imports requires that:

- each IRI in imports be resolved and
- the returned representation of that IRI be interpreted as a ShEx S and
- each shapeExpr in S.shapes be in scope for resolving shape expression references and
- each tripleExpr with a tripleExprLabel be in scope for resolving triple expression references.

If any imported schema imports other schemas, shape and triple expression labels from those schemas are also in scope.

IMPORT EXAMPLE 1 - SHAPE AND TRIPLE EXPRESSIONS

schema1: json

```
{ "type": "Schema", "imports": ["http://schema.example/schema2"], "shapes": [
  { "id": "http://schema.example/#EmployeeShape",
    "type": "Shape", "expression": {
      "type": "EachOf", "expressions": [
        "http://schema.example/#nameExpr",
        { "type": "TripleConstraint",
          "predicate": "http://schema.example/#empID",
          "valueExpr": { "type": "NodeConstraint",
            "datatype": "http://www.w3.org/2001/XMLSchema#integer" } } ] } } ] }
```

schema2:

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#PersonShape",
    "type": "Shape", "expression": {
      "id": "http://schema.example/#nameExpr",
      "type": "TripleConstraint",
      "predicate": "http://xmlns.com/foaf/0.1/name" } } ] }
```

Both the shape expression `<PersonShape>` and the triple expression `<nameExpr>` are in scope.
schema2's `<nameExpr>` is referenced in **schema1**'s `<EmployeeShape>`

Redundant imports are treated as a single import. This includes circular imports:

IMPORT EXAMPLE 2 - CIRCULAR IMPORT

json

```
schema1:
{ "type": "Schema",
  "imports": ["http://schema.example/schema2", "http://schema.example/schema3"],
  "shapes": [
  { "id": "http://schema.example/schema1#S1",
    "type": "Shape", "expression": {
      "type": "TripleConstraint", "predicate": "http://schema.example/#p1",
      "valueExpr": "http://schema.example/schema1#S2"
    } } ] }
schema2:
{ "type": "Schema",
  "imports": ["http://schema.example/schema3"],
  "shapes": [
  { "id": "http://schema.example/schema1#S2",
    "type": "Shape", "expression": {
      "type": "TripleConstraint", "predicate": "http://schema.example/#p2",
      "valueExpr": "http://schema.example/schema1#S3"
    } } ] }
schema3:
{ "type": "Schema",
  "imports": ["http://schema.example/schema1"],
  "shapes": [
  { "id": "http://schema.example/schema1#S3",
    "type": "Shape", "expression": {
      "type": "TripleConstraint", "predicate": "http://schema.example/#p3",
      "valueExpr": "http://schema.example/schema1#S1", "min": 0,
    } } ] }
```

When some schema A imports schema B, B's start member is ignored.

IMPORT EXAMPLE 3 - IGNORED START IN IMPORT

json

```
schema1:
{ "type": "Schema",
  "imports": ["http://schema.example/schema2"],
  "shapes": [
  { "id": "http://schema.example/schema1#S1",
    "type": "Shape", "expression": {
      "type": "TripleConstraint", "predicate": "http://schema.example/#p1",
      "valueExpr": "http://schema.example/schema1#S2"
    } } ] }
schema2:
{ "type": "Schema",
  "start": "http://schema.example/schema1#S2",
  "shapes": [
  { "id": "http://schema.example/schema1#S2",
    "type": "Shape", "expression": {
      "type": "TripleConstraint", "predicate": "http://schema.example/#p2"
    } } ] }
```

`schema1` has no `start` even though it imports a schema with a `start`.

It is an error if `A` and `B` share any labels for shape expressions or triple expressions or if schema `B` has a startActs member.

IMPORT EXAMPLE 4 - ERRONEOUS IMPORT

json

```
schema1:
{ "type": "Schema",
  "imports": ["http://schema.example/schema2"],
  "shapes": [
  { "id": "http://schema.example/schema1#S1",
    "type": "Shape", "expression": {
      "type": "TripleConstraint", "predicate": "http://schema.example/#p1",
      "valueExpr": "http://schema.example/schema1#S2"
    } } ] }
schema2:
{ "type": "Schema",
"startActs": [ { "type": "semAct",
                 "name": "http://schema.example/schema1#A1" } ],
  "shapes": [
  { "id": "http://schema.example/schema1#S1",
    "type": "Shape", "expression": {
      "type": "TripleConstraint", "predicate": "http://schema.example/#p1",
      "valueExpr": "http://schema.example/schema1#S2"
    } },
  { "id": "http://schema.example/schema1#S2",
    "type": "Shape", "expression": {
      "type": "TripleConstraint", "predicate": "http://schema.example/#p2",
      "valueExpr": "http://schema.example/schema1#S3"
    } } ] }
```

This import fails because:

- `<http://schema.example/schema1#S1>` has conflicting definitions and
- an included schema has a `start` directive and
- the reference to `<http://schema.example/schema1#S3>` is not resolvable after imports.

5.7 Schema Requirements

The semantics defined above assume three structural requirements beyond those imposed by the grammar of the abstract syntax. These ensure referential integrity and eliminate logical paradoxes such as those that arrise through the use of negation. These are not constraints expressed by the schema but instead those imposed on the schema.

5.7.1 Schema Validation Requirement

A graph G is said to ***conform*** with a schema S with a ShapeMap m when:

1) Every, SemAct in the startActs of S has a successful evaluation of semActsSatisfied.

2) Every node n in m conforms to its associated shapeExprRefs se_n where for each shapeExprRef se_i in se_n:

2.1) se_i references a ShapeExpr in shapes, and

2.2) `satisfies(n, `se_i`, G, Sch, m)` for each shape se_i in se_n.

5.7.2 Shape Expression Reference Requirement

A shapeExprRef *MUST* appear in the schema's shapes map (or an imported schema's map) and the corresponding shape expression *MUST* be a Shape with a shapeExpr. The function `shapeExprWithId(shapeExprRef)` returns the shape expression with an id of shapeExprRef.

Additionally, a shapeExprLabel cannot refer to itself through a shape reference either directly or recursively. The ***shapeExprRef closure*** of a shape expression se is the set of shape expression labels used as references in se. The shapeExprLabel sl belongs to `shapeExprRefClosure(se)` if and only if:

- sl appears as an atomic shapeExprRef in se, or
- sl belongs to `shapeExprRefClosure(shapeExprWithId(sl2))` for some shapeExprLabel sl2 that belongs to `shapeExprRefClosure(se)`.

A shapes schema *MUST NOT* define a shape label sl that belongs to the shapeExprRef closure of its definition `shapeExprWithId(sl)`.

Following are two valid shapeExprRefs:

```json
{"type" : "Schema",
  "shapes" : [ {
    "id" : "http://schema.example/#PersonShape",
    "type" : "Shape",
    "expression" : {
      "type" : "TripleConstraint",
      "predicate" : "http://xmlns.com/foaf/0.1/name"
    }
  }, {
    "id" : "http://schema.example/#EmployeeShape",
    "type" : "ShapeAnd",
    "shapeExprs" : [ "http://schema.example/#PersonShape", {
      "type" : "Shape",
      "expression" : {
        "type" : "TripleConstraint",
        "predicate" : "http://schema.example/#employeeNumber"
      }
    } ]
  } ]
}
```

```json
{"type" : "Schema",
  "shapes" : [ {
    "id" : "http://schema.example/#PersonShape",
    "type" : "Shape",
    "expression" : {
      "type" : "TripleConstraint",
      "predicate" : "http://xmlns.com/foaf/0.1/name"
    }
  }, {
    "id" : "http://schema.example/#EmployeeShape",
    "type" : "Shape",
    "expression" : {
      "type" : "TripleConstraint",
      "predicate" : "http://schema.example/#dependent",
      "valueExpr" : "http://schema.example/#PersonShape",
      "min" : 0,
      "max" : -1
    }
  } ]
}
```

This shapeExprRef is invalid because there is no corresponding shape expression:

```json
{ "type":"Schema", "shapes": [
    { "id": "http://schema.example/#S1",
      "type":"Shape", "expression":
        "http://schema.example/#MissingShapeExpr"
} ] }
```

This shapeExprRef is invalid because the referenced object is a triple expression instead of a shape expression:

```json
{"type" : "Schema",
  "shapes" : [ {
    "id" : "http://schema.example/#CustomerShape",
    "type" : "Shape",
    "expression" : {
```

```
    "id" : "http://schema.example/#discountExpr",
    "type" : "TripleConstraint",
    "predicate" : "http://schema.example/#discount"
   }
  }, {
   "id" : "http://schema.example/#EmployeeShape",
   "type" : "Shape",
   "expression" : {
    "type" : "TripleConstraint",
    "predicate" : "http://schema.example/#contactFor",
    "valueExpr" : "http://schema.example/#discountExpr"
   }
  } ]
}
```

These shapeExprRefs are invalid because they recursively refer to each other.

```
{"type" : "Schema",                                                    json
  "shapes" : [ {
   "id" : "http://schema.example/#PersonShape",
   "type" : "ShapeAnd",
   "shapeExprs" : [ "http://schema.example/#EmployeeShape", {
    "type" : "Shape",
    "expression" : {
     "type" : "TripleConstraint",
     "predicate" : "http://xmlns.com/foaf/0.1/name"
    }
   } ]
  }, {
   "id" : "http://schema.example/#EmployeeShape",
   "type" : "ShapeAnd",
   "shapeExprs" : [ "http://schema.example/#PersonShape", {
    "type" : "Shape",
    "expression" : {
     "type" : "TripleConstraint",
     "predicate" : "http://schema.example/#employeeNumber"
    }
   } ]
  } ] }
```

5.7.3 Triple Expression Reference Requirement

An tripleExprRef *MUST* identify a triple expression in the schema. The function `tripleExprWithId(`tripleExprRef`)` returns the triple expression with the id tripleExprRef.

Additionally, a tripleExprLabel cannot refer to itself through a triple expression reference either directly or recursively. The ***tripleExprRef closure*** of a triple expression te is the set of triple expression labels used as references in te. The tripleExprLabel tl belongs to `tripleExprRefClosure(`te`)` if and only if:

- tl appears as an atomic tripleExprRef in te, or
- tl belongs to `tripleExprRefClosure(tripleExprWithId(`tl2`))` for some tripleExprLabel tl2 that belongs to `tripleExprRefClosure(`te`)`.

A shapes schema *MUST NOT* define a triple expression label tl that belongs to the tripleExprRef closure of its definition `tripleExprWithId(`tl`)`.

Following is a valid triple expression reference:

```
{ "type":"Schema", "shapes": [
    { "id": "http://schema.example/#PersonShape",
      "type":"Shape", "expression": {
        "id": "http://schema.example/#nameExpr",
        "type": "TripleConstraint",
        "predicate": "http://xmlns.com/foaf/0.1/name"
      } },
    { "id": "http://schema.example/#EmployeeShape",
      "type":"Shape", "expression": { "type":"EachOf", "expressions": [
        "http://schema.example/#nameExpr",
        { "type": "TripleConstraint",
          "predicate": "http://schema.example/#employeeNumber" }
] } } ] }
```
json

This triple expression reference is invalid because there is no corresponding triple expression:

```
{ "type":"Schema", "shapes": [
    { "id": "http://schema.example/#S1",
      "type":"Shape", "expression":
        "http://schema.example/#missingTripleExpr"
} ] }
```
json

This triple expression reference is invalid because the referenced object is a shape expression instead of a triple expression:

```
{ "type":"Schema", "shapes": [
    { "id": "http://schema.example/#CustomerShape",
      "type":"ShapeAnd", "shapeExprs": [ … ]
    },
    { "id": "http://schema.example/#PreferredCustomerShape",
      "type":"Shape", "expression": { "type":"EachOf", "expressions": [
        "http://schema.example/#CustomerShape",
        { "type": "TripleConstraint",
          "predicate": "http://schema.example/#discount" }
] } } ] }
```
json

5.7.4 Negation Requirement

A schema *MUST NOT* contain any Shape that has a negated reference to itself, either directly or transitively. This is formalized by the requirement that the dependency graph of a schema *MUST NOT* have a cycle that traverses some negated reference.

The set of ***atomic shapes*** of a shapeExpr se contains a Shape s if s or its id appears either directly or by shapeExprRef in se. That is, s belongs to `atomicShapes(se)` if and only if

- s appears as an atomic shape in se, or
- sid is the id of s and sid appears as an atomic shapeExprRef in se, or
- s belongs to `atomicShapes(se2)` for some shape expression se2 such that the id of se2 belongs to the shapeExprRefClosure of se.

The set of ***atomic triple constraints*** of a tripleExpr te includes every TripleConstraint tc that appears directly or by tripleExprRef in te. That is, tc belongs to `atomicTripleConstraints(te)` if and only if:

- tc is an atomic TripleConstraint in te, or
- te is an atomic TripleConstraint in `tripleExprWithId(tl)` for some tripleExprLabel tl that belongs to `tripleExprRefClosure(te)`.

The Shape s1 has a ***reference*** to the Shape s2 if

- s1.expression is present, and
- there is a triple constraint tc that belongs to `atomicTripleConstraints(s1.expression)`, and
- tc.valueExpr is present, and
- s2 belongs to `atomicShapes(tc.valueExpr)`.

The reference from s1 to s2 is a ***negated reference*** if

- s2 appears under an odd number of ShapeNot in the shapeExprRef closure of tc.valueExpr, or
- tc has predicate p and s1 has extra p.

The ***dependency graph*** of the schema Sch is the graph which vertices are all the Shapes that appear in some shape expression in the shapes of Sch, and that has two kinds of edges: negative and positive. There is a negative edge from s1 to s2 if s1 has a negated reference to s2. There is a positive edge from s1 to s2 if s1 has a reference but not a negated reference to s2.

Examples with ShapeNot

This negated self-reference violates the negation requirement.

```
{ "type": "Schema", "shapes": [
    { "id": "http://schema.example/#S",
      "type": "Shape",
      "expression": { "type": "TripleConstraint",
        "predicate": "http://schema.example/#p",
        "valueExpr": { "type": "ShapeNot",
          "shapeExpr": "http://schema.example/#S" } } }
  ] }
```

This indirect self-reference does not violate the negation requirement.

```
{ "type": "Schema",
  "shapes": [
    { "id": "http://schema.example/#US",
      "type": "Shape",
      "expression": { "type": "TripleConstraint",
        "predicate": "http://schema.example/#Up",
        "valueExpr": { "type": "ShapeNot",
          "shapeExpr": "http://schema.example/#UT" } } },
    { "id": "http://schema.example/#UT",
      "type": "Shape",
      "expression": { "type": "TripleConstraint",
        "predicate": "http://schema.example/#Uq",
        "valueExpr": "http://schema.example/#US" } }
  ] }
```

This negated, indirect self-reference violates the negation requirement.

```
{"type" : "Schema",
  "shapes" : [ {
    "id" : "http://schema.example/#S",
    "type" : "Shape",
    "expression" : {
      "type" : "TripleConstraint",
      "predicate" : "http://schema.example/#p",
      "valueExpr" : {
        "type" : "ShapeNot",
        "shapeExpr" : "http://schema.example/#T"
      }
    }
  } , {
    "id" : "http://schema.example/#T",
    "type" : "Shape",
    "expression" : {
      "type" : "TripleConstraint",
      "predicate" : "http://schema.example/#q",
      "valueExpr" : "http://schema.example/#S"
    }
  } ] }
```

This is a direct, negated self-reference of the shape with id ex:T and violates the negation requirement.

```
{"type" : "Schema",
  "shapes" : [ {
    "id" : "http://schema.example/#T",
```

```
    "type" : "Shape",
    "expression" : {
      "type" : "TripleConstraint",
      "predicate" : "http://schema.example/#p",
      "valueExpr" : "http://schema.example/#S"
    }
  } , {
    "id" : "http://schema.example/#S",
    "type" : "ShapeAnd",
    "shapeExprs" : [ {
      "type" : "ShapeNot",
      "shapeExpr" : "http://schema.example/#T"
    }, "http://schema.example/#U" ]
  }, {
    "id" : "http://schema.example/#U",
    "type" : "Shape"
  } ] }
```

This doubly-negated self-reference of ex:T does not violate the negation requirement.

```
{"type" : "Schema",                                    json
  "shapes" : [{
    "id" : "http://schema.example/#T",
    "type" : "Shape",
    "expression" : {
      "type" : "TripleConstraint",
      "predicate" : "http://schema.example/#p",
      "valueExpr" : "http://schema.example/#S"
    }
  } , {
    "id" : "http://schema.example/#S",
    "type" : "ShapeNot",
    "shapeExpr" : {
      "type" : "ShapeAnd",
      "shapeExprs" : [ {
        "type" : "ShapeNot",
        "shapeExpr" : "http://schema.example/#T"
      }, "http://schema.example/#U" ]
    }
  }, {
    "id" : "http://schema.example/#U",
    "type" : "Shape"
  } ] }
```

There is a cycle of negated references between the shape that defines ex:T and the shape that defines ex:U, so the negation requirement is violated.

```
{"type" : "Schema",                                    json
  "shapes" : [{
    "id" : "http://schema.example/#T",
    "type" : "Shape",
    "expression" : {
      "type" : "TripleConstraint",
      "predicate" : "http://schema.example/#p",
      "valueExpr" : {
        "type" : "ShapeNot",
        "shapeExpr" : "http://schema.example/#S"
      }
    }
  } , {
```

```
  "id" : "http://schema.example/#U",
  "type" : "Shape",
  "expression" : {
    "type" : "TripleConstraint",
    "predicate" : "http://schema.example/#q",
    "valueExpr" : "http://schema.example/#S"
  }
 }, {
  "id" : "http://schema.example/#S",
  "type" : "ShapeAnd",
  "shapeExprs" : [ {
    "type" : "ShapeNot",
    "shapeExpr" : "http://schema.example/#T"
  }, "http://schema.example/#U" ]
 } ] }
```

This satisfies the negation requirement, as ex:U does not refer to ex:T (compared to the previous example).

```
{"type" : "Schema",                                    json
  "shapes" : [{
    "id" : "http://schema.example/#T",
    "type" : "Shape",
    "expression" : {
      "type" : "TripleConstraint",
      "predicate" : "http://schema.example/#p",
      "valueExpr" : {
        "type" : "ShapeNot",
        "shapeExpr" : "http://schema.example/#S"
      }
    }
  } , {
    "id" : "http://schema.example/#U",
    "type" : "Shape",
    "expression" : {
      "type" : "TripleConstraint",
      "predicate" : "http://schema.example/#q"
    }
  }, {
    "id" : "http://schema.example/#S",
    "type" : "ShapeAnd",
    "shapeExprs" : [ {
      "type" : "ShapeNot",
      "shapeExpr" : "http://schema.example/#T"
    }, "http://schema.example/#U" ]
  } ] }
```

Examples with Shape.extra predicate

This self-reference on a predicate designated as extra violates the negation requirement:

```json
{ "type": "Schema", "shapes": [
    { "id": "http://schema.example/#S",
      "type": "Shape",
      "extra": [ "http://schema.example/#p" ], "expression":
      { "type": "TripleConstraint",
        "predicate": "http://schema.example/#p",
        "valueExpr": "http://schema.example/#S"
} } ] }
```

The same shape with a negated self-reference still violates the negation requirement because the reference occurs with a ShapeNot:

```json
{ "type": "Schema", "shapes": [
    { "id": "http://schema.example/#S",
      "type": "Shape",
      "extra": [ "http://schema.example/#p" ],
      "expression": {
        "type": "TripleConstraint",
        "predicate": "http://schema.example/#p",
        "valueExpr": {
          "type": "ShapeNot", "shapeExpr": "http://schema.example/#S"
} } } ] }
```

5.8 Semantic Actions

Semantic actions serve as an extension point for Shape Expressions. They appear in lists in Schema's startActs and Shape, OneOf, EachOf and TripleConstraint's semActs.

A semantic action is a tuple of an identifier and some optional code:

```
SemAct { name:IRIREF code:STRING? }
```

5.8.1 Semantics

The evaluation **semActsSatisfied** on a list of SemActs returns success or failure. The evaluation of an individual SemAct is implementation-dependent.

5.8.2 Use - informative

A practical evaluation of a SemAct will provide access to some context. For instance, the http://shex.io/extensions/Test/ extension requires access to the subject, predicate and object of a triple matching a TripleConstraint. These are used in a `print` function.

SEMANTIC ACTIONS EXAMPLE 1

```
{ "type": "Schema", "shapes": [
    { "id": "http://schema.example/#S1",
      "type": "Shape", "expression": {
        "type": "TripleConstraint", "predicate": "http://schema.example/#p1",
        "min": 1, "max": -1,
        "semActs": [
          { "type": "SemAct", "code": " print(s) ",
            "name": "http://shex.io/extensions/Test/" },
          { "type": "SemAct", "code": " print(o) ",
            "name": "http://shex.io/extensions/Test/" } ] } } ] }
```

json

```
<http://a.example/n1> <http://a.example/p1> <http://a.example/o1> .
<http://a.example/n2> <http://a.example/p1> "a", "b" .
<http://a.example/n3> <http://a.example/p2> <http://a.example/o2> .
```

node	shape	result	print arguments
<n1>	<S1>	pass	http://a.example/s1 http://a.example/o1
<n2>	<S1>	pass	http://a.example/s1 "a" http://a.example/s1 "b"
<n3>	<S1>	fail	

5.9 Annotations

Annotations provide a format-independent way to provide additional information about elements in a schema. They appear in lists in Shape, OneOf, EachOf and TripleConstraint's annotations.

```
Annotation { predicate:IRIREF object:objectValue }
```

5.9.1 Semantics - informative

Annotations do not affect whether a node conforms to some shape. Because they are part of the structure of the schema, they can be parsed in one ShEx format and emitted in that format or another.

ANNOTATIONS EXAMPLE 1

```
{ "type": "Schema", "shapes": [
    { "id": "http://schema.example/#IssueShape",
      "type": "Shape", "expression": {
        "type": "TripleConstraint",
        "predicate": "http://schema.example/#status",
        "annotations": [
           { "type": "Annotation",
             "predicate": "http://www.w3.org/2000/01/rdf-schema#comment",
             "object": {"value": "Represents reported software issues."} },
           { "type": "Annotation",
             "predicate": "http://www.w3.org/2000/01/rdf-schema#label",
             "object": {"value": "software issue"} } ] } } ] }
```

5.10 Validation Examples

The following examples demonstrate proofs for validations in the form of a nested list of invocations of the evaluation functions defined above.

5.10.1 Simple Examples

Schema:

S1

```
{ "type": "Schema", "shapes": [
    { "id": "http://schema.example/#IntConstraint",
      "type": "NodeConstraint",
      "datatype": "http://www.w3.org/2001/XMLSchema#integer"
    } ] }
```

nc1

Here the shape identified by `http://schema.example/#IntConstraint` is a shape expression consisting of a single NodeConstraint. Per Shape Expression Semantics, "30"^^<http://www.w3.org/2001/XMLSchema#integer> satisfies IntConstraint.

This document uses this nested tree convention to indicate that the dependency of an evaluation on those nested inside it. Nesting is expressed as indentation. Here, the evaluation of satisfies NodeConstraint ("30"^^xsd:integer, S1, G, m) depends on satisfies2 NodeConstraint ("30"^^xsd:integer, S1).

Validate "30"^^<http://www.w3.org/2001/XMLSchema#integer> as IntConstraint:
    satisfies NodeConstraint ("30"^^xsd:integer, S1, G, m)
      satisfies2 NodeConstraint ("30"^^xsd:integer, S1)

Validating a shape requires evaluating it's triple expression as well as the variables and functions neigh(G, n), matched, remainder, outs, matchables and unmatchables:

Schema:

S1
tc1

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#UserShape",
    "type": "Shape", "expression":
    { "type": "TripleConstraint",
      "predicate": "http://schema.example/#shoeSize"
      } } ] }
```

json

Data:

t1

```
BASE <http://a.example/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
<Alice> ex:shoeSize "30"^^xsd:integer .
```

Validate <Alice> as http://schema.example/#UserShape:

G = [t1] *The graph G consists of one triple.*
satisfies Shape (<Alice>, S1, G, m)
neigh(G, <Alice>) = [t1] */* The neighborhood around <Alice> consists of one triple. */*
matched = [t1] */* That triple is matched in the nested evaluation. */*
remainder = Ø */* The remainder is the empty set. */*
matches TripleConstraint ([t1], tc1, m)
outs = [t1] */* There is one arc out. */*
matchables = Ø */* There are no remaining arcs out of <Alice> with predicates appearing in tc1. */*
unmatchables = Ø */* There are no other arcs out of <Alice>. */*
closed is false */* The Shape's closed paramater has a value of false. */*

It is quite common that Shapes will constrain their nested TripleConstraints with NodeConstraints. Here is an example including that, extra triples and a closed shape:

Schema:

```
S1  { "type": "Schema", "shapes": [                                    json
tc1   { "id": "http://schema.example/#UserShape",
        "type": "Shape",
        "extra": ["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
        "expression": {
nc1       "type": "TripleConstraint",
          "predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "valueExpr":
          { "type": "NodeConstraint",
            "values": ["http://schema.example/#Teacher"]
          } } } ] }
```

Data:

```
t1  BASE <http://a.example/>
t2  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
t3  <Alice> ex:shoeSize "30"^^xsd:integer .
t4  <Alice> a ex:Teacher .
t5  <Alice> a ex:Person .
    <SomeHat> ex:owner <Alice> .
    <TheMoon> ex:madeOf <GreenCheese> .
```

Validate <Alice> as http://schema.example/#UserShape:

G = [t1,t2,t3,t4,t5]
satisfies Shape (<Alice>, S1, G, m)
neigh(G, <Alice>) = [t1,t2,t3,t4], matched = [t2], remainder = [t1,t3]
matches TripleConstraint ([t2], tc1, m)
satisfies NodeConstraint (ex:Teacher, nc1, G, m)
satisfies2 NodeConstraint (ex:Teacher, nc1)
outs = [t1,t2,t3]
matchables = [t3], unmatchables = [t1], closed is false

The non-empty matchables is permitted because the triple `t3` has a predicate which appears in the "extra" list: ["http://schema.example/#Teacher"].

5.10.2 Disjunction Example

Schema:

```
S1  { "type": "Schema", "shapes": [                                    json
te1   { "id": "http://schema.example/#UserShape",
tc1     "type": "Shape", "expression":
         {"type": "OneOf", "expressions": [
            { "type": "TripleConstraint",
              "predicate": "http://xmlns.com/foaf/0.1/name",
nc1           "valueExpr":
te2             { "type": "NodeConstraint", "nodeKind": "literal" } },
tc2         { "type": "EachOf", "expressions": [
              { "type": "TripleConstraint", "min": 1, "max": -1 ,
                "predicate": "http://xmlns.com/foaf/0.1/givenName",
nc2             "valueExpr":
tc3               { "type": "NodeConstraint", "nodeKind": "literal" } },
              { "type": "TripleConstraint",
                "predicate": "http://xmlns.com/foaf/0.1/familyName",
nc3             "valueExpr":
                  { "type": "NodeConstraint", "nodeKind": "literal" } }
            ] }
         ] }
 } ] }
```

Data:

```
t1  BASE <http://a.example/>
t2  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
t3  <Alice> foaf:givenName "Alice" .
t4  <Alice> foaf:givenName "Malsenior" .
t5  <Alice> foaf:familyName "Walker" .
t6  <Alice> foaf:mbox <mailto:alice@example.com> .
    <Bob> foaf:knows <Alice> .
    <Bob> foaf:mbox <mailto:bob@example.com> .
```

Per Shape Expression Semantics, <Alice> satisfies S1 with the simple ShapeMap

```
m:  { "http://a.example/Alice": "http://a.example/UserShape }
```

as seen in this validation.

Validate <Alice> as http://schema.example/#UserShape:

G = [t1,t2,t3,t4,t5,t6]
satisfies Shape (<Alice>, S1, G, m)
neigh(G, <Alice>) = [t1,t2,t3,t4,t5], matched = [t1,t2,t3], remainder = [t4,t5]
matches OneOf ([t1,t2,t3], te1, m)
matches EachOf ([t1,t2,t3], te2, m)
matches cardinality ([t1,t2], tc2, m)
matches TripleConstraint ([t1], tc2, m)
satisfies NodeConstraint ("Alice", nc2, G, m)
satisfies2 NodeConstraint ("Alice", nc2)
matches TripleConstraint ([t2], tc2, m)
satisfies NodeConstraint ("Malsenior", nc2, G, m)
satisfies2 NodeConstraint ("Malsenior", nc2)
matches TripleConstraint ([t3], tc3, m)
satisfies NodeConstraint ("Walker", nc3, G, m)
satisfies2 NodeConstraint ("Walker", nc3)
outs = [t4] /* t5 is in ArcsIn(G, <Alice>)>, t6 is not in neigh(G, <Alice>)>. */
matchables = Ø, unmatchables = [t5], closed is false

Replacing triples 1-3 with a single foaf:name property will also satisfy the schema.

Data:

```
t4  BASE <http://a.example/>
t5  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
t6  <Alice> foaf:mbox <mailto:alice@example.com> .
```

t7
```
<Bob> foaf:knows <Alice> .
<Bob> foaf:mbox <mailto:bob@example.com> .
<Alice> foaf:name "Alice Malsenior Walker" .
```

Validate <Alice> as http://schema.example/#UserShape:

G = [t4,t5,t6,t7]
satisfies Shape (<Alice>, S1, G, m)
  neigh(G, <Alice>) = [t4,t5,t7], matched = [t7], remainder = [t4,t5]
  matches OneOf ([t7], te1, m)
    matches TripleConstraint ([t7], tc1, m)
      satisfies NodeConstraint ("Walker", nc3, G, m)
        satisfies2 NodeConstraint ("Walker", nc3)
  outs = [t4]
  matchables = Ø, unmatchables = [t5], closed is false

Any mixure of `foaf:name` with `foaf:givenName` or `foaf:familyName` will fail to satisfy the schema as there will be a matchable triple t3 that's not used in the triple expression te1.

Data:

t3
t4
t5
t6
t7
```
BASE <http://a.example/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
<Alice> foaf:familyName "Walker" .
<Alice> foaf:mbox <mailto:alice@example.com> .
<Bob> foaf:knows <Alice> .
<Bob> foaf:mbox <mailto:bob@example.com> .
<Alice> foaf:name "Alice Malsenior Walker" .
```

Validate <Alice> as http://schema.example/#UserShape:

G = [t4,t5,t6,t7]
satisfies Shape (<Alice>, S1, G, m)
  neigh(G, <Alice>) = [t4,t5,t7], matched = [t7], remainder = [t4,t5]
  matches OneOf ([t7], te1, m)
    matches TripleConstraint ([t7], tc1, m)
      satisfies NodeConstraint ("Walker", nc3, G, m)
        satisfies2 NodeConstraint ("Walker", nc3)
  outs = [t4]
  matchables = [t3], unmatchables = [t5], closed is false

Adding a `foaf:familyName` to S1's extra would allow this graph to satisfy the schema.

S1
```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#UserShape",
    "type": "Shape", "extra": ["http://xmlns.com/foaf/0.1/familyName"] …
   } ] }
```

Closing S1 would also cause a validation failure if unmatchables were not empty:

S1
```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#UserShape",
    "type": "Shape", "closed": true …
   } ] }
```

G = [t4,t5,t6,t7]
satisfies Shape (<Alice>, S1, G, m)
  …
  unmatchables = [t5], closed is true

5.10.3 Dependent Shape Example

Schema:

S1
tc1
nc1
S2
tc2
nc2

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#IssueShape",
    "type": "Shape", "expression":
    { "type": "TripleConstraint",
      "predicate": "http://schema.example/#reproducedBy",
      "valueExpr":
      "http://schema.example/#TesterShape" } },
  { "id": "http://schema.example/#TesterShape",
    "type": "Shape", "expression":
    { "type": "TripleConstraint",
      "predicate": "http://schema.example/#role",
      "valueExpr":
      { "type": "NodeConstraint",
        "values": [ "http://schema.example/#testingRole" ] } } }
  ] }
```

json

Data:

t1
t2

```
PREFIX ex: <http://schema.example/#>
PREFIX inst: <http://inst.example/>
inst:Issue1 ex:reproducedBy inst:Tester2 .
inst:Tester2 ex:role ex:testingRole .
```

inst:Issue1 satisfies S1 with the ShapeMap

m:

```
{ "http://inst.example/Issue1": "http://schema.example/#IssueShape",
  "http://inst.example/Tester2": "http://schema.example/#TesterShape",
  "http://inst.example/Testgrammer23": "http://schema.example/#ProgrammerShape" }
```

Validate inst:Issue1 as http://schema.example/#IssueShape:

as seen in this evaluation:

G = [t1]
satisfies Shape (inst:Issue1, S1, G, m)
neigh(G, inst:Issue1) = [t1,t2], matched = [t1,t2], remainder = Ø
matches TripleConstraint ([t1], tc1, m)
satisfies NodeConstraint (inst:Tester2, nc1, G, m)
satisfies2 NodeConstraint (inst:Tester2, nc1)
satisfies Shape (inst:Tester2. S2, G, m)
neigh(G, inst:Tester2) = [t2], matched = [t2], remainder = Ø
matches TripleConstraint ([t2], tc2, m)
satisfies NodeConstraint (ex:testingRole, nc2, G, m)
satisfies2 NodeConstraint (ex:testingRole, nc2)
outs = Ø
matchables = Ø, unmatchables = Ø, closed is false
outs = Ø
matchables = Ø, unmatchables = Ø, closed is false

5.10.4 Recursion Example

Schema:

S1
tc1
nc1

```
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#IssueShape",
    "type": "Shape", "expression":
    { "type": "TripleConstraint", "min": 0, "max": -1,
      "predicate": "http://schema.example/#related",
      "valueExpr":
      "http://schema.example/#IssueShape"
    } } ] }
```

json

Data:

t1
t2
t3

```
PREFIX ex: <http://schema.example/#>
PREFIX inst: <http://inst.example/>
inst:Issue1 ex:related inst:Issue2 .
inst:Issue2 ex:related inst:Issue3 .
inst:Issue3 ex:related inst:Issue1 .
```

inst:Issue1 satisfies S1 with the ShapeMap

m:

```
{ "http://inst.example/Issue1": "http://schema.example/#IssueShape",
  "http://inst.example/Issue2": "http://schema.example/#IssueShape",
  "http://inst.example/Issue3": "http://schema.example/#IssueShape" }
```

Validate inst:Issue1 as http://schema.example/#IssueShape:

as seen in this evaluation:

G = [t1,t2,t3]
satisfies Shape (inst:Issue1, S1, G, m)
neigh(G, inst:Issue1) = [t1], matched = [t1], remainder = Ø
matches TripleConstraint ([t1], tc1, m)
satisfies NodeConstraint (inst:Issue2, nc1, G, m)
satisfies2 NodeConstraint (inst:Issue2, nc1)
satisfies Shape (inst:Issue2. S2, G, m)
neigh(G, inst:Issue2) = [t3], matched = [t3], remainder = Ø
matches TripleConstraint ([t3], tc3, m)
satisfies NodeConstraint (inst:Issue3, nc3, G, m)
satisfies2 NodeConstraint (inst:Issue3, nc3)
satisfies Shape (inst:Issue3. S2, G, m)
neigh(G, inst:Issue3) = [t3], matched = [t3], remainder = Ø
matches TripleConstraint ([t3], tc3, m)
satisfies NodeConstraint (inst:Issue1, nc3, G, m)
satisfies2 NodeConstraint (inst:Issue1, nc3)
This is known to be true or the initial typing would not be satisfied.
outs = Ø
matchables = Ø, unmatchables = Ø, closed is false
outs = Ø
matchables = Ø, unmatchables = Ø, closed is false
outs = Ø
matchables = Ø, unmatchables = Ø, closed is false

5.10.5 Simple Repeated Property Examples

Schema:

S1
te1
tc1
nc1
tc2
nc2

```
{ "type": "Schema", "shapes": [
    { "id": "http://schema.example/#TestResultsShape",
      "type": "Shape", "expression": {
        "type": "EachOf", "expressions": [
          { "type": "TripleConstraint", "min": 1, "max": -1,
            "predicate": "http://schema.example/#val",
            "valueExpr":
            { "type": "NodeConstraint",
              "values": [ {"value": "a"}, {"value": "b"}, {"value": "c"} ] } },
          { "type": "TripleConstraint", "min": 1, "max": -1,
            "predicate": "http://schema.example/#val",
            "valueExpr":
            { "type": "NodeConstraint",
              "values": [ {"value": "b"}, {"value": "c"}, {"value": "d"} ] } }
        ] } } ] }
```

Data:

t1
t2
t3
t4

```
BASE <http://a.example/>
PREFIX ex: <http://schema.example/#>
<s> ex:val "a" .
<s> ex:val "b" .
<s> ex:val "c" .
<s> ex:val "d" .
```

<s> satisfies S1 with:

m:

```
{ "http://a.example/s": "http://a.example/S1 }
```

Validate <s> as http://schema.example/#TestResultShape:

If tc1 consumes as many triples as it can, it consumes three and tc2 consumes one:

G = [t1,t2,t3,t4]
satisfies Shape (<s>, S1, G, m)
neigh(G, <s>) = [t1,t2,t3,t4], matched = [t1,t2,t3,t4], remainder = Ø
matches EachOf ([t1,t2,t3,t4], te1, m)
matches cardinality ([t1,t2,t3], tc1, m)
matches TripleConstraint ([t1], tc1, m)
satisfies NodeConstraint ("a", nc1, G, m)
satisfies2 NodeConstraint ("a", nc1)
matches TripleConstraint ([t2], tc1, m)
satisfies NodeConstraint ("b", nc1, G, m)
satisfies2 NodeConstraint ("b", nc1)
matches TripleConstraint ([t3], tc1, m)
satisfies NodeConstraint ("c", nc1, G, m)
satisfies2 NodeConstraint ("c", nc1)
matches cardinality ([t4], tc2, m)
matches TripleConstraint ([t4], tc2, m)
satisfies NodeConstraint ("d", nc2, G, m)
satisfies2 NodeConstraint ("d", nc2)
outs = Ø
matchables = Ø, unmatchables = Ø, closed is false

If we eliminate t4, either t2 or t3 must be allocated to tc2:

G = [t1,t2,t3]
satisfies Shape (<Alice>, S1, G, m)
neigh(G, <Alice>) = [t1,t2,t3], matched = [t1,t2,t3], remainder = Ø
matches EachOf ([t1,t2,t3], te1, m)
matches cardinality ([t1,t2], tc1, m)
matches TripleConstraint ([t1], tc1, m)
satisfies NodeConstraint ("a", nc1, G, m)
satisfies2 NodeConstraint ("a", nc1)
matches TripleConstraint ([t2], tc1, m)
satisfies NodeConstraint ("b", nc1, G, m)

```
        satisfies2 NodeConstraint ("b", nc1)
    matches cardinality ([t3], tc2, m)
      matches TripleConstraint ([t3], tc2, m)
        satisfies NodeConstraint ("d", nc2, G, m)
          satisfies2 NodeConstraint ("d", nc2)
  outs = Ø
  matchables = Ø, unmatchables = Ø, closed is false
```

5.10.6 Repeated Property With Dependent Shapes Example

Schema:

```
S1  { "type": "Schema", "shapes": [                                    json
te1   { "id": "http://schema.example/#IssueShape",
        "type": "Shape", "expression":
tc1     { "type": "EachOf", "expressions": [
            { "type": "TripleConstraint",
nc1           "predicate": "http://schema.example/#reproducedBy",
              "valueExpr":
tc2           "http://schema.example/#TesterShape" },
            { "type": "TripleConstraint",
nc2           "predicate": "http://schema.example/#reproducedBy",
              "valueExpr":
S2            "http://schema.example/#ProgrammerShape" }
        ] } },
tc3   { "id": "http://schema.example/#TesterShape",
        "type": "Shape", "expression":
        { "type": "TripleConstraint",
nc3       "predicate": "http://schema.example/#role",
          "valueExpr":
S3        { "type": "NodeConstraint",
            "values": [ "http://schema.example/#testingRole" ] } } },
tc4   { "id": "http://schema.example/#ProgrammerShape",
        "type": "Shape", "expression":
        { "type": "TripleConstraint",
nc4       "predicate": "http://schema.example/#department",
          "valueExpr":
          { "type": "NodeConstraint",
            "values": [ "http://schema.example/#ProgrammingDepartment" ] } } }
  ] }
```

Data:

```
t1  PREFIX ex: <http://schema.example/#>
t2  PREFIX inst: <http://inst.example/>
    inst:Issue1
      ex:reproducedBy inst:Tester2 ;
t3    ex:reproducedBy inst:Testgrammer23 .

    inst:Tester2
t4    ex:role ex:testingRole .
t5
    inst:Testgrammer23
      ex:role ex:testingRole ;
      ex:department ex:ProgrammingDepartment .
```

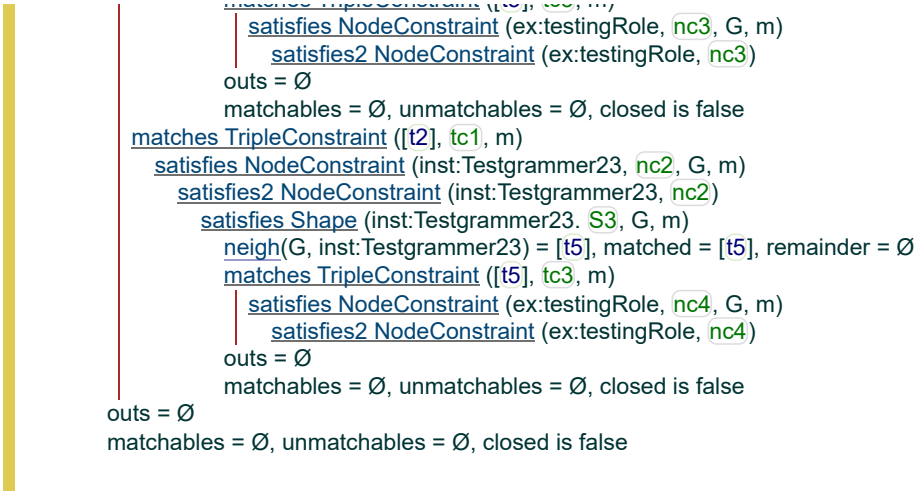
inst:Issue1 satisfies S1 with the ShapeMap

```
m:  { "http://inst.example/Issue1": "http://schema.example/#IssueShape",
      "http://inst.example/Tester2": "http://schema.example/#TesterShape",
      "http://inst.example/Testgrammer23": "http://schema.example/#ProgrammerShape" }
```

Validate inst:Issue1 as http://schema.example/#IssueShape:

as seen in this evaluation:

G = [t1,t2,t3,t4,t5]
satisfies Shape (inst:Issue1, S1, G, m)
neigh(G, inst:Issue1) = [t1,t2], matched = [t1,t2], remainder = Ø
matches EachOf ([t1,t2], te1, m)
matches TripleConstraint ([t1], tc1, m)
satisfies NodeConstraint (inst:Tester2, nc1, G, m)
satisfies2 NodeConstraint (inst:Tester2, nc1)
satisfies Shape (inst:Tester2. S2, G, m)
neigh(G, inst:Tester2) = [t3], matched = [t3], remainder = Ø
matches TripleConstraint ([t3], tc3, m)



5.10.7 Negation Example

Setting the maximum cardinality of a TripleConstraint with predicate p to zero (i.e. "max": 0 in ShExJ or `{0}` or `{0, 0}` in ShExC) asserts that matching nodes must have no triples with predicate p.

Schema:

```
S1  { "type": "Schema", "shapes": [                                    json
te1     { "id": "http://schema.example/#TestResultsShape",
tc1       "type": "Shape", "expression": {
            "type": "EachOf", "expressions": [
              { "type": "TripleConstraint", "min": 1, "max": -1,
nc1             "predicate": "http://schema.example/#p1",
                "valueExpr":
tc2             { "type": "NodeConstraint",
                  "values": [ {"value": "a"}, {"value": "b"} ] } },
              { "type": "TripleConstraint", "min": 1, "max": -1,
                "predicate": "http://schema.example/#p2", "min": 0, "max": 0 }
            ] } } ] }
```

Data:

```
t1  BASE <http://a.example/>
    PREFIX ex: <http://schema.example/#>
    <s> ex:p1 "a" .
```

<s> satisfies S1 with:

```
m:  { "http://a.example/s": "http://a.example/S1" }
```

Validate <s> as http://schema.example/#TestResultShape:

This is trivially satisfied by tc1 consuming one triple and tc2 consuming none:

G = [t1]
satisfies Shape (<s>, S1, G, m)
  neigh(G, <s>) = [t1], matched = [t1], remainder = Ø
  matches EachOf ([t1], te1, m)
    matches cardinality ([t1], tc1, m)
      matches TripleConstraint ([t1], tc1, m)
        satisfies NodeConstraint ("a", nc1, G, m)
          satisfies2 NodeConstraint ("a", nc1)
    matches cardinality ([], tc2, m)
      matches TripleConstraint ([], tc2, m)
  outs = Ø
  matchables = Ø, unmatchables = Ø, closed is false

If we add a t2 which matches tc2:

Data:

```
t1  BASE <http://a.example/>
t2  PREFIX ex: <http://schema.example/#>
    <s> ex:p1 "a" .
    <s> ex:p2 5 .
```

every partition fails, either because matchables is non-empty or because the maximum cardinality on tc2 is exceeded:

G = [t1]
satisfies Shape (<s>, S1, G, m)
  neigh(G, <s>) = [t1], matched = [t1], remainder = Ø
  matches EachOf ([t1], te1, m)
    matches cardinality ([t1], tc1, m)
      matches TripleConstraint ([t1], tc1, m)
        satisfies NodeConstraint ("a", nc1, G, m)
          satisfies2 NodeConstraint ("a", nc1)
    matches cardinality ([t2], tc2, m)
      matches TripleConstraint ([t2], tc2, m)
  outs = Ø
  matchables = Ø, unmatchables = Ø, closed is false

6. ShEx Compact syntax (ShExC)

The ShEx Compact Syntax expresses ShEx schemas in a compact, human-friendly form. Parsing ***ShExC*** transforms a ShExC document into an equivalent ShExJ structure. This is defined as a BNF which accepts ShExC followed by instructions for tranlating the rules in the BNF production into their corresponding ShExJ objects. For example, "`shapeExprDecl returns shapeExpression`" indicates that the result of matching the `shapeExprDecl` production is the object produced by parsing the `shapeExpression` production.

Semantic actions before the first shape expression declaration are startActs. After the first shape expression declaration, semantic actions are associated with the previous declaration.

Display grammar only

Below is the ShExC grammar following the notation in the XML specification[XML]:

[1]	`shexDoc`	::=	`directive* ((notStartAction | startActions) statement*)?`

followed by the associated ShExJ object(s):

```
Schema      {startActs:[SemAct+]? start:shapeExpr? imports:[IRIREF+]? shapes:
             [shapeExpr+]? }
```

and a description of the mapping of rules in the production to elements of the ShExJ object:

- startActs comes from startActions production.
- start comes from the start production.
- shapes come from the shapeExprDecl production.

[2]	`directive`	::=	`baseDecl | prefixDecl | importDecl`
[3]	`baseDecl`	::=	`"BASE" IRIREF`
[4]	`prefixDecl`	::=	`"PREFIX" PNAME_NS IRIREF`
[4½]	`importDecl`	::=	`"IMPORT" IRIREF`

"`IMPORT`" is described in ShEx Import.

[5]	`notStartAction`	::=	`start | shapeExprDecl`
[6]	`start`	::=	`"start" '=' inlineShapeExpression`
[7]	`startActions`	::=	`codeDecl+`
[8]	`statement`	::=	`directive | notStartAction`
[9]	`shapeExprDecl`	::=	`shapeExprLabel (shapeExpression | "EXTERNAL")`

If the "`EXTERNAL`" keyword is present, shapeExprDecl returns a ShapeExternal object:

```
ShapeExternal      {id:shapeExprLabel? }
```

otherwise shapeExprDecl returns shapeExpression.

Shape expressions are logical combinations of shape atoms. Inline variants of shape expressions are used in tripleConstraints and are not permitted to have annotations or semantic actions.

[10]	`shapeExpression`	::=	`shapeOr`
[11]	`inlineShapeExpression`	::=	`inlineShapeOr`
[12]	`shapeOr`	::=	`shapeAnd ("OR" shapeAnd)*`
[13]	`inlineShapeOr`	::=	`inlineShapeAnd ("OR" inlineShapeAnd)*`

If the right shapeAnd matches one or more times, the result is a ShapeOr object with shapeExprs containing the first shapeAnd followed by the ordered list from the second shapeAnd:

```
ShapeOr      {id:shapeExprLabel? shapeExprs:[shapeExpr{2,}] }
```

otherwise the result is the left shapeAnd.

[14]	shapeAnd	::=	shapeNot ("AND" shapeNot)*
[15]	inlineShapeAnd	::=	inlineShapeNot ("AND" inlineShapeNot)*

If the right shapeNot matches one or more times, the result is a ShapeAnd object with shapeExprs containing the first shapeNot followed by the ordered list from the second shapeNot:

```
ShapeAnd      {id:shapeExprLabel? shapeExprs:[shapeExpr{2,}] }
```

otherwise the result is the left shapeNot.

[16]	shapeNot	::=	"NOT"? shapeAtom
[17]	inlineShapeNot	::=	"NOT"? inlineShapeAtom

If the left "NOT" matches, the result is a ShapeNot object with shapeExpr containing the shapeAtom:

```
ShapeNot      {id:shapeExprLabel? shapeExpr:shapeExpr }
```

otherwise the result is the shapeAtom.

Shape atoms are shape references (indicated by "@"), definitions, or nested expressions.

[18]	shapeAtom	::=	nonLitNodeConstraint shapeOrRef? | litNodeConstraint | shapeOrRef nonLitNodeConstraint? | '(' shapeExpression ')' | '.'
[19]	shapeAtomNoRef	::=	nonLitNodeConstraint shapeOrRef? | litNodeConstraint | shapeDefinition nonLitNodeConstraint? | '(' shapeExpression ')' | '.'
[20]	inlineShapeAtom	::=	nonLitNodeConstraint inlineShapeOrRef? | litNodeConstraint | inlineShapeOrRef nonLitNodeConstraint? | '(' shapeExpression ')' | '.'

- If the matching production inludes both a node constraint (litNodeConstraint OR nonLitNodeConstraint) nc and a shapeOrRef, the result is a ShapeAnd object with shapeExprs containing the list of nc and a shapeOrRef.
- If the "(" shapeExpression ")" production matches, the result is the result of shapeExpression.
- If the "." production matches, the result is an empty shape: {"type": "Shape"}.

[21]	shapeOrRef	::=	shapeDefinition | shapeRef
[22]	inlineShapeOrRef	::=	inlineShapeDefinition | shapeRef
[23]	shapeRef	::=	ATPNAME_LN | ATPNAME_NS | '@' shapeExprLabel

- If the shapeDefinition production matches, the result is shapeDefinition.
- Otherwise, the result is a shapeExprRef to shapeExprLabel.

```
shapeExprRef      =shapeExprLabel ;
```

Node constraints identify a (possibly infinite) set of matching RDF nodes.

[24]	litNodeConstraint	::=	"LITERAL" xsFacet* | datatype xsFacet* | valueSet xsFacet* | numericFacet+

[25]	nonLitNodeConstraint	::=	nonLiteralKind stringFacet* | stringFacet+
	NodeConstraint		{id:shapeExprLabel? nodeKind:("iri" | "bnode" | "nonliteral" | "literal")? datatype:IRIREF? xsFacet* values:[valueSetValue+]? }
[26]	nonLiteralKind	::=	"IRI" | "BNODE" | "NONLITERAL"
[27]	xsFacet	::=	stringFacet | numericFacet
	xsFacet		=stringFacet | numericFacet ;
[28]	stringFacet	::=	stringLength INTEGER | REGEXP
[29]	stringLength	::=	"LENGTH" | "MINLENGTH" | "MAXLENGTH"
	stringFacet		=(length|minlength|maxlength):INTEGER | pattern:STRING flags:STRING? ;
[30]	numericFacet	::=	numericRange numericLiteral | numericLength INTEGER
[31]	numericRange	::=	"MININCLUSIVE" | "MINEXCLUSIVE" | "MAXINCLUSIVE" | "MAXEXCLUSIVE"
[32]	numericLength	::=	"TOTALDIGITS" | "FRACTIONDIGITS"
	numericFacet		=(mininclusive|minexclusive|maxinclusive|maxexclusive):numericLiteral | (totaldigits|fractiondigits):INTEGER ;

Shape defintions associate a triple expression with a closed flag and a list of partially constrained (extra) predicates. Any predicate appearing in a triple expression is fully constrained unless it appears in the list of extras.

[33]	shapeDefinition	::=	("extra" extraPropertySet | "CLOSED")* '{' tripleExpression? '}' annotation* semanticActions
[34]	inlineShapeDefinition	::=	("extra" extraPropertySet | "CLOSED")* '{' tripleExpression? '}'
	Shape		{id:shapeExprLabel? closed:BOOL? extra:[IRIREF+]? expression:tripleExpr? semActs:[SemAct+]? annotations:[Annotation+]? }

- closed is true if the "CLOSED" choice was matched one or more times.
- extra is the set of IRIs matching the extraPropertySet production.
- expression comes from the tripleExpression production.
- annotations is the set of Annotations matching the annotation production.
- semActs is the set of semantic actions matching the semanticActions production.

[35]	extraPropertySet	::=	"EXTRA" predicate+

Triple expressions are arrangements of triple constraints.

[36]	tripleExpression	::=	oneOfTripleExpr
[37]	oneOfTripleExpr	::=	groupTripleExpr | multiElementOneOf
[38]	multiElementOneOf	::=	groupTripleExpr ('|' groupTripleExpr)+

If the right groupTripleExpr matches one or more times, the result is a OneOf object with expressions containing the first groupTripleExpr followed by the ordered list from the second groupTripleExpr:

OneOf	{id:tripleExprLabel? expressions:[tripleExpr{2,}] min:INTEGER? max:INTEGER? semActs:[SemAct+]? annotations:[Annotation+]? }

otherwise the result is the left groupTripleExpr.

[40]	groupTripleExpr	::=	singleElementGroup | multiElementGroup
[41]	singleElementGroup	::=	unaryTripleExpr ';'?
[42]	multiElementGroup	::=	unaryTripleExpr (';' unaryTripleExpr)+ ';'?

If the right unaryTripleExpr matches one or more times, the result is a EachOf object with expressions containing the first unaryTripleExpr followed by the ordered list from the second unaryTripleExpr:

```
EachOf      {id:tripleExprLabel? expressions:[tripleExpr{2,}] min:INTEGER?
             max:INTEGER? semActs:[SemAct+]? annotations:[Annotation+]? }
```

otherwise the result is the left unaryTripleExpr.

[43] unaryTripleExpr ::= ('$' tripleExprLabel)? (tripleConstraint | bracketedTripleExpr) | include

[44] bracketedTripleExpr ::= '(' tripleExpression ')' cardinality? annotation* semanticActions

Triple constraints are matched against RDF triples.

[45] tripleConstraint ::= senseFlags? predicate inlineShapeExpression cardinality? annotation* semanticActions

```
TripleConstraint      {id:tripleExprLabel? inverse:BOOL? predicate:IRIREF
                       valueExpr:shapeExpr? min:INTEGER? max:INTEGER? semActs:[SemAct+]?
                       annotations:[Annotation+]? }
```

- inverse is true if the senseFlags matched "^".
- predicate comes from the predicate production.
- valueExpr comes from the inlineShapeExpression production. It it is an empty shape `{"type": "Shape"}`, valueExpr is not assigned.
- min comes from the cardinality production.
- max comes from the cardinality production.
- annotations is the set of Annotations matching the annotation production.
- semActs is the set of semantic actions matching the semanticActions production.

[46] cardinality ::= '*' | '+' | '?' | REPEAT_RANGE

In ShExJ, "*" is represented as -1, standing for the unbounded cardinality..

[47] senseFlags ::= '^'

Value sets identify ranges of RDF nodes by explicit inclusion or by range (indicated by "~"). Ranges may include exclusions, which may also be ranges but must not in turn contain exclusions. A valueSetValue may be an objectValue or one of IriStem, IriStemRange, LiteralStem, LiteralStemRange, LanguageStem, LanguageStemRange, .

[48] valueSet ::= '[' valueSetValue* ']'

[49] valueSetValue ::= iriRange | literalRange | languageRange | exclusion+

If "." matches and exclusion matches one or more times, all matched items must be consistently iri, literal, or language. valueSetValue returns either a IriStemRange, LiteralStemRange, or LanguageStemRange object with exclusions equal to the set of results of exclusion:

```
IriStemRange        {stem:(IRIREF | Wildcard) exclusions:[IRIREF|IriStem +] }
LiteralStemRange    {stem:(STRING | Wildcard) exclusions:[STRING|LiteralStem +] }
LanguageStemRange   {stem:(LANGTAG | Wildcard) exclusions:[LANGTAG|LanguageStem +] }
```

If "~" matches with no exclusion, valueSetValue returns a Wildcard object:

```
Wildcard      {/* empty */ }
```

[50] exclusion ::= '.' '-' (iri | literal | LANGTAG) '~'?

[51] iriRange ::= iri ('~' exclusion*)?

If iri matches with no "~", iriRange returns iri.

If iri and "~" match with no iriExclusion, iriRange returns a IriStem object:

```
IriStem     {stem:IRIREF }
```

If iri and "~" match and iriExclusion matches one or more times, iriRange returns a IriStemRange object with exclusions equal to the set of results of iriExclusion:

```
IriStemRange     {stem:(IRIREF | Wildcard) exclusions:[IRIREF|IriStem +] }
```

[52] iriExclusion ::= '-' iri '~'?

[53] literalRange ::= literal ('~' literalExclusion*)?

If literal matches with no "~", literalRange returns literal.

If literal and "~" match with no literalExclusion, literalRange returns a LiteralStem object:

```
LiteralStem     {stem:STRING }
```

If literal and "~" match and literalExclusion matches one or more times, literalRange returns a LiteralStemRange object with exclusions equal to the set of results of literalExclusion:

```
LiteralStemRange     {stem:(STRING | Wildcard) exclusions:[STRING|LiteralStem +] }
```

[54] literalExclusion ::= '-' literal '~'?

[55] languageRange ::= LANGTAG ('~' languageExclusion*)?
| '@' '~' languageExclusion*

If LANGTAG matches with no "~" match , languageRange returns a Language object with languageTag equal to LANGTAG:

```
Language     {languageTag:LANGTAG }
```

If LANGTAG and "~" match with no languageExclusion, languageRange returns a LanguageStem object:

```
LanguageStem     {stem:LANGTAG }
```

If LANGTAG and "~" match and languageExclusion matches one or more times, languageRange returns a LanguageStemRange object with exclusions equal to the set of results of languageExclusion:

```
LanguageStemRange     {stem:(LANGTAG | Wildcard) exclusions:[LANGTAG|LanguageStem +] }
```

If '@' '~' matched with no languageExclusion, languageRange returns a LanguageStemRange object with an empty stem:

```
LanguageStemRange     {stem: "" }
```

If '@' '~' matched and languageExclusion matches one or more times, languageRange returns a LanguageStemRange object with an empty stem ad exclusions equal to the set of results of languageExclusion:

```
LanguageStemRange     {stem: "" exclusions:[LANGTAG|LanguageStem +] }
```

[56] languageExclusion ::= '-' LANGTAG '~'?

Triple expressions can include the shapeExpression in a shapeExprDecl.

[57] include ::= '&' tripleExprLabel

Per the triple expression refrence requirement, tripleExprLabel property *MUST* appear in the schema's shapes map and the corresponding triple expression *MUST* be a Shape with a tripleExpr.

```
tripleExprRef     =tripleExprLabel ;
```

Triple expressions can include annotations in the form of a tuple of a predicate and an iri or literal.

[58] annotation ::= "//" predicate (iri | literal)

```
Annotation     {predicate:IRIREF object:objectValue }
```

Triple expressions can include semantic actions consisting of an iri and an optional code string.

[59] semanticActions ::= codeDecl*

[60] codeDecl ::= '%' iri (CODE | '%')

```
SemAct     {name:IRIREF code:STRING? }
```

The remaining productions come from the specifications for SPARQL and Turtle.

[13t]	literal	::=	rdfLiteral | numericLiteral | booleanLiteral
[61]	predicate	::=	iri | RDF_TYPE
[62]	datatype	::=	iri
[63]	shapeExprLabel	::=	iri | blankNode
[64]	tripleExprLabel	::=	iri | blankNode
[16t]	numericLiteral	::=	INTEGER | DECIMAL | DOUBLE
[65]	rdfLiteral	::=	langString | string ("^^" datatype)?
	returns: literal		The literal has a lexical form of the first rule argument, String. If the '^^' iri rule matched, the datatype is iri and the literal has no language tag. If the langString rule matched, the datatype is rdf:langString and the language tag is extracted from langTag. If neither matched, the datatype is xsd:string and the literal has no language tag.
[134s]	booleanLiteral	::=	"true" | "false"
	returns: literal		The literal has a lexical form of the true or false, depending on which matched the input, and a datatype of xsd:boolean.
[135s]	string	::=	STRING_LITERAL1 | STRING_LITERAL_LONG1 | STRING_LITERAL2 | STRING_LITERAL_LONG2
[66]	langString	::=	LANG_STRING_LITERAL1 | LANG_STRING_LITERAL_LONG1 | LANG_STRING_LITERAL2 | LANG_STRING_LITERAL_LONG2
[136s]	iri	::=	IRIREF | prefixedName
[137s]	prefixedName	::=	PNAME_LN | PNAME_NS
[138s]	blankNode	::=	BLANK_NODE_LABEL

Terminals

Terminals return:

- *the RDF abstract types IRI, lexical form, literal, **language tag**.*
- *a string of unicode codepoints for CODE.*
- *a `repeat range` for REPEAT_RANGE. A `repeat range` is a tuple of non-negative integers or a non-negative integer and a token for `*`.*

[67]	<CODE>	::=	"{" ([^%\\] | "\\" [%\\] | UCHAR)* "%" "}"
	returns: a string of unicode codepoints		The characters between "{" and "%}" are taken, with the numeric escape sequences unescaped, to form the unicode string of the IRI.
[68]	<REPEAT_RANGE>	::=	"{" INTEGER ("," (INTEGER | "*")?)? "}"
	returns: repeat range		The base-10 numeric values of INTEGER are taken or a non-negative integer and an * token if "*" was matched.
[69]	<RDF_TYPE>	::=	"a"
	returns: IRI		The iri http://www.w3.org/1999/02/22-rdf-syntax-ns# is returned.
[18t]	<IRIREF>	::=	"<" ([^#0000- <>\"{}|^`\\] | UCHAR)* ">"
	returns: IRI		The characters between "<" and ">" are taken, with the numeric escape sequences unescaped, to form the unicode string of the IRI. Relative IRI resolution is performed per Turtle Section 6.3.
[140s]	<PNAME_NS>	::=	PN_PREFIX? ":"

	returns: PREFIX		When used in a prefixDecl production, the prefix is a potentially empty unicode string matching the first argument of the rule and serves as a key into the prefixes map.
	returns: IRI		When used elsewhere, the `iri` is the value in the prefixes map corresponding to the first argument of the rule.
[141s]	<PNAME_LN>	::=	PNAME_NS PN_LOCAL
	returns: IRI		A potentially empty prefix is identified by the first token, PNAME_NS. The prefixes map *MUST* have a corresponding `namespace`. The unicode string of the IRI is formed by unescaping the reserved characters [rfc7159] in the second argument, PN_LOCAL, and concatenating this onto the `namespace`.
[70]	<ATPNAME_NS>	::=	"@" PNAME_NS
	returns: IRI		The `iri` is the value in the prefixes map corresponding to the second token of the rule.
[71]	<ATPNAME_LN>	::=	"@" PNAME_LN
	returns: IRI		A potentially empty prefix is identified by the second token, PNAME_NS. The prefixes map *MUST* have a corresponding `namespace`. The unicode string of the IRI is formed by unescaping the reserved characters [rfc7159] in the third token, PN_LOCAL, and concatenating this onto the `namespace`.

[72]	<REGEXP>	::=	'/' ([^/\\\n\r]
			  | '\\' [nrt\\\|.?*+(){}$-\[\]^/]
			  | UCHAR
			  )+ '/' [smix]*

			{pattern:STRING flags:STRING? }
	returns: JSON object		pattern is a unicode string formed from the characters between the outermost '/'s by unescaping matches of '\\' '/' in the terminal pattern as well as the numeric escape sequences matched by UCHAR. The remaining escape sequences are included verbatim in pattern, e.g. `^\/\t\\\U0001D4B8$` would become `^/\t\\\U0001D4B8$`. flags is a sequence of the characters [smix] if any were matched. Otherwise no flags attribute is returned.
[142s]	<BLANK_NODE_LABEL>	::=	"_:" (PN_CHARS_U | [0-9]) ((PN_CHARS | ".")* PN_CHARS)?
	returns: blank node		The characters following the "_:" form a blank node identifier. This corresponds to any blank node in the input dataset that had the same label.
[145s]	<LANGTAG>	::=	"@" ([a-zA-Z])+ ("-" ([a-zA-Z0-9])+)*
	returns: language tag		The characters following the @ form the unicode string of the language tag.
[19t]	<INTEGER>	::=	[+-]? [0-9]+
	returns: literal		The literal has a lexical form of the input string, and a datatype of `xsd:integer`.
[20t]	<DECIMAL>	::=	[+-]? [0-9]* "." [0-9]+
	returns: literal		The literal has a lexical form of the input string, and a datatype of `xsd:double`.
[21t]	<DOUBLE>	::=	[+-]? ([0-9]+ "." [0-9]* EXPONENT | "."? [0-9]+ EXPONENT)
	returns: literal		The literal has a lexical form of the input string, and a datatype of `xsd:double`.
[155s]	<EXPONENT>	::=	[eE] [+-]? [0-9]+
[156s]	<STRING_LITERAL1>	::=	"'" ([^'\\\n\r] | ECHAR | UCHAR)* "'"
	returns: lexical form		The characters between the outermost """s are taken, with numeric and string escape sequences unescaped, to form the unicode string of a lexical form.

[157s]	<STRING_LITERAL2>	::=	'"' ([^\"\\\n\r] | ECHAR | UCHAR)* '"'
	returns: lexical form		The characters between the outermost ""s are taken, with numeric and string escape sequences unescaped, to form the unicode string of a lexical form.
[158s]	<STRING_LITERAL_LONG1>	::=	"'''" (("'" | "''")? ([^\'\\] | ECHAR | UCHAR))* "'''"
	returns: lexical form		The characters between the outermost """s are taken, with numeric and string escape sequences unescaped, to form the unicode string of a lexical form.
[159s]	<STRING_LITERAL_LONG2>	::=	'"""' (('"' | '""')? ([^\"\\] | ECHAR | UCHAR))* '"""'
	returns: lexical form		The characters between the outermost """s are taken, with numeric and string escape sequences unescaped, to form the unicode string of a lexical form.
[73]	<LANG_STRING_LITERAL1>	::=	"'" ([^'\\\n\r] | ECHAR | UCHAR)* "'" LANGTAG
	returns: lexical form		The characters between the outermost ""s are taken, with numeric and string escape sequences unescaped, to form the unicode string of a lexical form. The trailing LANGTAG is used to create a language-tagged string.
[74]	<LANG_STRING_LITERAL2>	::=	'"' ([^\"\\\n\r] | ECHAR | UCHAR)* '"' LANGTAG
	returns: lexical form		The characters between the outermost ""s are taken, with numeric and string escape sequences unescaped, to form the unicode string of a lexical form. The trailing LANGTAG is used to create a language-tagged string.
[75]	<LANG_STRING_LITERAL_LONG1>	::=	"'''" (("'" | "''")? ([^\'\\] | ECHAR | UCHAR))* "'''" LANGTAG
	returns: lexical form		The characters between the outermost """s are taken, with numeric and string escape sequences unescaped, to form the unicode string of a lexical form. The trailing LANGTAG is used to create a language-tagged string.
[76]	<LANG_STRING_LITERAL_LONG2>	::=	'"""' (('"' | '""')? ([^\"\\] | ECHAR | UCHAR))* '"""' LANGTAG
	returns: lexical form		The characters between the outermost """s are taken, with numeric and string escape sequences unescaped, to form the unicode string of a lexical form. The trailing LANGTAG is used to create a language-tagged string.
[26t]	<UCHAR>	::=	"\\u" HEX HEX HEX HEX | "\\U" HEX HEX HEX HEX HEX HEX HEX HEX
[160s]	<ECHAR>	::=	"\\" [tbnrf\\\"\\']
[164s]	<PN_CHARS_BASE>	::=	[A-Z] | [a-z] | [#00C0-#00D6] | [#00D8-#00F6] | [#00F8-#02FF] | [#0370-#037D] | [#037F-#1FFF] | [#200C-#200D] | [#2070-#218F] | [#2C00-#2FEF] | [#3001-#D7FF] | [#F900-#FDCF] | [#FDF0-#FFFD] | [#10000-#EFFFF]
[165s]	<PN_CHARS_U>	::=	PN_CHARS_BASE | "_"
[167s]	<PN_CHARS>	::=	PN_CHARS_U | "-" | [0-9] | [#00B7] | [#0300-#036F] | [#203F-#2040]
[168s]	<PN_PREFIX>	::=	PN_CHARS_BASE ((PN_CHARS | ".")* PN_CHARS)?
[77]	<PN_LOCAL>	::=	(PN_CHARS_U | ":" | [0-9] | PLX) (PN_CHARS | "." | ":" | PLX)*
[170s]	<PLX>	::=	PERCENT | PN_LOCAL_ESC
[171s]	<PERCENT>	::=	"%" HEX HEX
[172s]	<HEX>	::=	[0-9] | [A-F] | [a-f]

[173s]	<PN_LOCAL_ESC>	::=	"\\" ("_" | "~" | "." | "-" | "!" | "$" | "&" | "'" | "(" | ")" | "*" | "+" | "," | ";" | "=" | "/" | "?" | "#" | "@" | "%")
[98]	PASSED TOKENS	::=	[\t\r\n]+ | "#" [^\r\n]* | "/*" ([^*] | '*' ([^/] | '\\/'))* "*/"

A. ShEx JSON Syntax (ShExJ)

This section aggregates the JSON grammar rules defined above and includes terminals referenced above.

A ShExJ document is a JSON-LD [JSON-LD] document which uses a proscribed structure to define a schema containing shape expressions and triple expressions. A ShExJ document *MAY* include an `@context` property referencing `http://www.w3.org/ns/shex.jsonld`. In the absense of a top-level `@context`, ShEx Processors *MUST* act as if a `@context` property is present with the value `http://www.w3.org/ns/shex.jsonld`.

A ShExJ document can also be thought of as the serialization of an RDF Graph using the Shape Expression Vocabulary [shex-vocab] which conforms to the shape defined in § B. ShEx Shape. Processors *MAY* interpret a ShExJ document as an RDF Graph. Processors may also transform arbitrary RDF Graphs conforming to § B. ShEx Shape into ShExJ using a mechanism not described within this specification.

In ShExJ, the unbounded cardinality constraint is `-1`, rather than `"*"`.

This is the complete grammar for ***ShExJ***.

```
Schema { "@context": "http://www.w3.org/ns/shex.jsonld"
         startActs:[SemAct+]? start:shapeExpr? imports:[IRIREF]? shapes:[shapeExpr+]? }
shapeExpr = ShapeOr | ShapeAnd | ShapeNot | NodeConstraint | Shape | ShapeExternal | shapeExprRef ;
ShapeOr { id:shapeExprLabel? shapeExprs:[shapeExpr{2,}] }
ShapeAnd { id:shapeExprLabel? shapeExprs:[shapeExpr{2,}] }
ShapeNot { id:shapeExprLabel? shapeExpr:shapeExpr }
ShapeExternal { id:shapeExprLabel? }
shapeExprRef = shapeExprLabel ;
shapeExprLabel = IRIREF | BNODE ;
NodeConstraint { id:shapeExprLabel? nodeKind:("iri" | "bnode" | "nonliteral" | "literal")? datatype:IRIREF?
         xsFacet* values:[valueSetValue]? }
xsFacet = stringFacet | numericFacet ;
stringFacet = (length|minlength|maxlength):INTEGER | pattern:STRING flags:STRING? ;
numericFacet = (mininclusive|minexclusive|maxinclusive|maxexclusive):numericLiteral
         |(totaldigits|fractiondigits):INTEGER ;
numericLiteral = INTEGER | DECIMAL | DOUBLE ;
valueSetValue = objectValue | IriStem | IriStemRange | LiteralStem | LiteralStemRange | Language |
         LanguageStem | LanguageStemRange ;
objectValue = IRIREF | ObjectLiteral ;
ObjectLiteral { value:STRING language:STRING? type:STRING? }
IriStem { stem:IRIREF }
IriStemRange { stem:(IRIREF | Wildcard) exclusions:[IRIREF|IriStem +] }
LiteralStem { stem:STRING }
LiteralStemRange { stem:(STRING | Wildcard) exclusions:[STRING|LiteralStem +] }
Language { languageTag:LANGTAG }
LanguageStem { stem:LANGTAG }
LanguageStemRange { stem:(LANGTAG | Wildcard) exclusions:[LANGTAG|LanguageStem +] }
Wildcard { /* empty */ }
Shape { id:shapeExprLabel? closed:BOOL? extra:[IRIREF]? expression:tripleExpr? semActs:[SemAct+]?
         annotations:[Annotation+]? }
tripleExpr = EachOf | OneOf | TripleConstraint | tripleExprRef ;
EachOf { id:tripleExprLabel? expressions:[tripleExpr{2,}] min:INTEGER? max:INTEGER? semActs:
         [SemAct+]? annotations:[Annotation+]? }
OneOf { id:tripleExprLabel? expressions:[tripleExpr{2,}] min:INTEGER? max:INTEGER? semActs:
         [SemAct+]? annotations:[Annotation+]? }
TripleConstraint { id:tripleExprLabel? inverse:BOOL? predicate:IRIREF valueExpr:shapeExpr? min:INTEGER?
         max:INTEGER? semActs:[SemAct+]? annotations:[Annotation+]? }
tripleExprRef = tripleExprLabel ;
tripleExprLabel = IRIREF | BNODE ;
SemAct { name:IRIREF code:STRING? }
Annotation { predicate:IRI object:objectValue }
```

Terminals These follow the rules for terminals in the XML 1.0 5th Edition

```
#Turtle IRIREF without enclosing "<>"s
IRIREF : (PN_CHARS | '.' | ':' | '/' | '\\' | '#' | '@' | '%' | '&' | UCHAR)* ;
#Turtle BLANK_NODE_LABEL
BNODE : '_:' (PN_CHARS_U | [0-9]) ((PN_CHARS | '.')* PN_CHARS)? ;
#JSON boolean values
BOOL : "true" | "false" ;
#Turtle INTEGER
INTEGER : [+-]? [0-9] + ;
#Turtle DECIMAL
DECIMAL : [+-]? [0-9]* '.' [0-9] + ;
#Turtle DOUBLE
DOUBLE : [+-]? ([0-9] + '.' [0-9]* EXPONENT | '.' [0-9]+ EXPONENT | [0-9]+ EXPONENT) ;
```

```
                # BCP47 Language-Tag
LANGTAG       : ([a-zA-Z])+('-'([a-zA-Z0-9])+)* ;
                # any JSON string
STRING        : .* ;
```

Components These terminals are referenced by other terminals but not by external productions.

```
PN_PREFIX     : PN_CHARS_BASE ((PN_CHARS | '.')* PN_CHARS)? ;
PN_CHARS_BASE :  [A-Z] | [a-z] | [\u00C0-\u00D6] | [\u00D8-\u00F6]
               | [\u00F8-\u02FF] | [\u0370-\u037D] | [\u037F-\u1FFF]
               | [\u200C-\u200D] | [\u2070-\u218F] | [\u2C00-\u2FEF]
               | [\u3001-\uD7FF] | [\uF900-\uFDCF] | [\uFDF0-\uFFFD]
               | [\u10000-\uEFFFF] ;
PN_CHARS      : PN_CHARS_U | '-' | [0-9] | '\u00B7' | [\u0300-\u036F] | [\u203F-\u2040] ;
PN_CHARS_U    : PN_CHARS_BASE | '_' ;
UCHAR         :  '\\u' HEX HEX HEX HEX
               | '\\U' HEX HEX HEX HEX HEX HEX HEX HEX ;
HEX           : [0-9] | [A-F] | [a-f] ;
EXPONENT      : [eE] [+-]? [0-9]+ ;
```

B. ShEx Shape

This section is non-normative.

The following schema describes the form of an RDF Graph conforming to a Shape Expression schema, consisten with the description of ShExJ.

EXAMPLE 1: ShExR Shape Expression Schema

```
PREFIX sx: <http://www.w3.org/ns/shex#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
BASE <http://www.w3.org/ns/shex#>
start=@<Schema>

<Schema> CLOSED {
  a [sx:Schema] ;
  sx:imports @<IriList1Plus>? ;
  sx:startActs @<SemActList1Plus>? ;
  sx:start @<shapeExpr>?;
  sx:shapes @<shapeExpr>*
}

<shapeExpr> @<ShapeOr> OR @<ShapeAnd> OR @<ShapeNot> OR @<NodeConstraint> OR @<Shape> OR @<ShapeExternal

<ShapeOr> CLOSED {
  a [sx:ShapeOr] ;
  sx:shapeExprs @<shapeExprList2Plus>
}

<ShapeAnd> CLOSED {
  a [sx:ShapeAnd] ;
  sx:shapeExprs @<shapeExprList2Plus>
}

<ShapeNot> CLOSED {
  a [sx:ShapeNot] ;
  sx:shapeExpr @<shapeExpr>
}

<NodeConstraint> CLOSED {
  a [sx:NodeConstraint] ;
  sx:nodeKind [sx:iri sx:bnode sx:literal sx:nonliteral]?;
  sx:datatype IRI ? ;
  &<xsFacets>  ;
  sx:values @<valueSetValueList1Plus>?
}

<Shape> CLOSED {
  a [sx:Shape] ;
  sx:closed [true false]? ;
  sx:extra IRI* ;
  sx:expression @<tripleExpression>? ;
  sx:semActs @<SemActList1Plus>? ;
  sx:annotation @<AnnotationList1Plus>? ;
}

<ShapeExternal> CLOSED {
  a [sx:ShapeExternal] ;
}

<SemAct> CLOSED {
  a [sx:SemAct] ;
  sx:name IRI ;
  sx:code xsd:string?
}
```

```
<Annotation> CLOSED {
  a [sx:Annotation] ;
  sx:predicate IRI ;
  sx:object @<objectValue>
}

#  <@stringFacet> OR <@numericFacet>
<facet_holder> { # hold labeled productions
  $<xsFacets> ( &<stringFacet> | &<numericFacet> )* ;
  $<stringFacet> (
      sx:length xsd:integer
    | sx:minlength xsd:integer
    | sx:maxlength xsd:integer
    | sx:pattern xsd:string ; sx:flags xsd:string?
  );
  $<numericFacet> (
      sx:mininclusive   @<numericLiteral>
    | sx:minexclusive   @<numericLiteral>
    | sx:maxinclusive   @<numericLiteral>
    | sx:maxexclusive   @<numericLiteral>
    | sx:totaldigits    xsd:integer
    | sx:fractiondigits xsd:integer
  )
}
<numericLiteral> xsd:integer OR xsd:decimal OR xsd:double

<valueSetValue> @<objectValue> OR @<IriStem> OR @<IriStemRange>
                               OR @<LiteralStem> OR @<LiteralStemRange>
                OR @<Language> OR @<LanguageStem> OR @<LanguageStemRange>
<objectValue> IRI OR LITERAL # rdf:langString breaks on Annotation.object
<Language> CLOSED { a [sx:Language]; sx:languageTag xsd:string }
<IriStem> CLOSED { a [sx:IriStem]; sx:stem xsd:string }
<IriStemRange> CLOSED {
  a [sx:IriStemRange];
  sx:stem xsd:string OR @<Wildcard>;
  sx:exclusion @<IriStemExclusionList1Plus>
}
<LiteralStem> CLOSED { a [sx:LiteralStem]; sx:stem xsd:string }
<LiteralStemRange> CLOSED {
  a [sx:LiteralStemRange];
  sx:stem xsd:string OR @<Wildcard>;
  sx:exclusion @<LiteralStemExclusionList1Plus>
}
<LanguageStem> CLOSED { a [sx:LanguageStem]; sx:stem xsd:string }
<LanguageStemRange> CLOSED {
  a [sx:LanguageStemRange];
  sx:stem xsd:string OR @<Wildcard>;
  sx:exclusion @<LanguageStemExclusionList1Plus>
}
<Wildcard> BNODE CLOSED {
  a [sx:Wildcard]
}

<tripleExpression> @<TripleConstraint> OR @<OneOf> OR @<EachOf> OR CLOSED {  }

<OneOf> CLOSED {
  a [sx:OneOf] ;
  sx:min xsd:integer? ;
  sx:max xsd:integer? ;
  sx:expressions @<tripleExpressionList2Plus> ;
```

```
  sx:semActs @<SemActList1Plus>? ;
  sx:annotation @<AnnotationList1Plus>?
}

<EachOf> CLOSED {
  a [sx:EachOf] ;
  sx:min xsd:integer? ;
  sx:max xsd:integer? ;
  sx:expressions @<tripleExpressionList2Plus> ;
  sx:semActs @<SemActList1Plus>? ;
  sx:annotation @<AnnotationList1Plus>?
}

<tripleExpressionList2Plus> CLOSED {
  rdf:first @<tripleExpression> ;
  rdf:rest @<tripleExpressionList1Plus>
}
<tripleExpressionList1Plus> CLOSED {
  rdf:first @<tripleExpression> ;
  rdf:rest  [rdf:nil] OR @<tripleExpressionList1Plus>
}

<TripleConstraint> CLOSED {
  a [sx:TripleConstraint] ;
  sx:inverse [true false]? ;
  sx:negated [true false]? ;
  sx:min xsd:integer? ;
  sx:max xsd:integer? ;
  sx:predicate IRI ;
  sx:valueExpr @<shapeExpr>? ;
  sx:semActs @<SemActList1Plus>? ;
  sx:annotation @<AnnotationList1Plus>?
}

<IriList1Plus> CLOSED {
  rdf:first IRI ;
  rdf:rest  [rdf:nil] OR @<IriList1Plus>
}

<SemActList1Plus> CLOSED {
  rdf:first @<SemAct> ;
  rdf:rest  [rdf:nil] OR @<SemActList1Plus>
}

<shapeExprList2Plus> CLOSED {
  rdf:first @<shapeExpr> ;
  rdf:rest  @<shapeExprList1Plus>
}
<shapeExprList1Plus> CLOSED {
  rdf:first @<shapeExpr> ;
  rdf:rest  [rdf:nil] OR @<shapeExprList1Plus>
}

<valueSetValueList1Plus> CLOSED {
  rdf:first @<valueSetValue> ;
  rdf:rest  [rdf:nil] OR @<valueSetValueList1Plus>
}

<AnnotationList1Plus> CLOSED {
  rdf:first @<Annotation> ;
```

```
  rdf:rest  [rdf:nil] OR @<AnnotationList1Plus>
}

<IriStemExclusionList1Plus> CLOSED {
  rdf:first IRI OR @<IriStem> ;
  rdf:rest  [rdf:nil] OR @<IriStemExclusionList1Plus>
}

<LiteralStemExclusionList1Plus> CLOSED {
  rdf:first xsd:string OR @<LiteralStem> ;
  rdf:rest  [rdf:nil] OR @<LiteralStemExclusionList1Plus>
}

<LanguageStemExclusionList1Plus> CLOSED {
  rdf:first xsd:string OR @<LanguageStem> ;
  rdf:rest  [rdf:nil] OR @<LanguageStemExclusionList1Plus>
}
```

IANA Considerations

This section has been submitted to the Internet Engineering Steering Group (IESG) for review, approval, and registration with IANA.

text/shex

Type name:
text

Subtype name:
shex

Required parameters:
None

Optional parameters:
None

Encoding considerations:
8-bit text

ShEx Compact Syntax (ShExC) is a text language which is encoded in UTF-8.

Security considerations:
Given that ShExC allows the substitution of long IRIs with short terms, ShExC documents may expand considerably when processed and, in the worst case, the resulting data might consume all of the recipient's resources. Applications should treat any data with due skepticism.

Interoperability considerations:
Not Applicable

Published specification:
http://shex.io/shex-semantics/

Applications that use this media type:
Any programming environment that requires the exchange of directed graphs. Implementations of ShEx have been created for JavaScript, Python, Ruby, and Java.

Fragment Identifier Considerations:
The structure of a ShEx schema is defined by its representation in JSON per ShEx JSON Syntax (ShExJ). The JSON-LD context <http://www.w3.org/ns/shex.jsonld> defines the RDF representation (ShExR) for every ShEx schema. A ShEx fragment identifies an instance of either the shapeExpr or tripleExpr ShExJ productions, as well as the RDF resource (see RDF 1.1 Concepts and Abstract Syntax §6 [RDF11-CONCEPTS]) in the corresponding ShExR.

Restrictions on Usage:
None

Provisional Registrations:
Not Applicable

Additional information:
Deprecated alias names for this type:
None

Magic number(s):

File extension(s):
.shex

Macintosh file type code(s):
TEXT

Intended usage:
Common

Other Information & Comments:
None

Contact Person:
Contact Name:
Eric Prud'hommeaux

Contact Email Address:
eric@w3.org

Change controller:
W3C

D. Security Considerations

Revealing the structure of an RDF graph can reveal information about the content of conformant data. For instance, a schema with a predicate to describe cancer stage indicates that conforming graphs describe patients with cancer.

The process of testing a graph's conformance to a schema may involve many detailed queries which could draw resources to respond to API calls or SPARQL queries.

ShEx has an extension mechanism which can, in principle, evalute arbitrary code, possibly as some trusted agent. Such extensions should not be executed if they don't come from a trusted source.

Since ShEx is intended to be a pure data exchange format for validating RDF graphs, the ShExJ serialization *SHOULD NOT* be passed through a code execution mechanism such as JavaScript's `eval()` function to be parsed. An (invalid) document may contain code that, when executed, could lead to unexpected side effects compromising the security of a system.

See also, § C. IANA Considerations.

E. References

E.1 Normative references

[ECMASCRIPT-6.0]
ECMA-262 6th Edition, The ECMAScript 2015 Language Specification. Allen Wirfs-Brock. Ecma International. June 2015. Standard. URL: http://www.ecma-international.org/ecma-262/6.0/index.html

[JSON-LD]
JSON-LD 1.0. Manu Sporny; Gregg Kellogg; Markus Lanthaler. W3C. 16 January 2014. W3C Recommendation. URL: https://www.w3.org/TR/json-ld/

[RDF11-CONCEPTS]
RDF 1.1 Concepts and Abstract Syntax. Richard Cyganiak; David Wood; Markus Lanthaler. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf11-concepts/

[rdf11-mt]
RDF 1.1 Semantics. Patrick Hayes; Peter Patel-Schneider. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf11-mt/

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[rfc4647]
Matching of Language Tags. A. Phillips; M. Davis. IETF. September 2006. Best Current Practice. URL: https://tools.ietf.org/html/rfc4647

[rfc7159]
The JavaScript Object Notation (JSON) Data Interchange Format. T. Bray, Ed.. IETF. March 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7159

[RFC8174]
Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://tools.ietf.org/html/rfc8174

[shape-map]
ShapeMap Structure and Language. Eric Prud'hommeaux; Thomas Baker. URL: http://shex.io/shape-map/

[shex-vocab]
Shape Expression Vocabulary. Gregg Kellogg. URL: http://www.w3.org/ns/shex#

[sparql11-query]
SPARQL 1.1 Query Language. Steven Harris; Andy Seaborne. W3C. 21 March 2013. W3C Recommendation. URL: https://www.w3.org/TR/sparql11-query/

[turtle]
RDF 1.1 Turtle. Eric Prud'hommeaux; Gavin Carothers. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/turtle/

[XML]
Extensible Markup Language (XML) 1.0 (Fifth Edition). Tim Bray; Jean Paoli; Michael Sperberg-McQueen; Eve Maler; François Yergeau et al. W3C. 26 November 2008. W3C Recommendation. URL: https://www.w3.org/TR/xml/

[xmlschema-2]
XML Schema Part 2: Datatypes Second Edition. Paul V. Biron; Ashok Malhotra. W3C. 28 October 2004. W3C Recommendation. URL: https://www.w3.org/TR/xmlschema-2/

[xpath-functions]
XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition). Ashok Malhotra; Jim Melton; Norman Walsh; Michael Kay. W3C. 14 December 2010. W3C Recommendation. URL: https://www.w3.org/TR/xpath-functions/

[xpath-functions-31]
XPath and XQuery Functions and Operators 3.1. Michael Kay. W3C. 21 March 2017. W3C Recommendation. URL: https://www.w3.org/TR/xpath-functions-31/

[xpath20]
XML Path Language (XPath) 2.0 (Second Edition). Anders Berglund; Scott Boag; Don Chamberlin; Mary Fernandez; Michael Kay; Jonathan Robie; Jerome Simeon et al. W3C. 14 December 2010. W3C Recommendation. URL: https://www.w3.org/TR/xpath20/

↥