Shape Expressions Language 2.next

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](../primer/index.html) for a non-normative description of ShEx.

RDF, Schema, Shape Expressions, Structure Definition, Structural Validation

This is an editor's draft of the Shape Expressions specification. ShEx 2.x differs significantly from the W3C ShEx Submission. The [July 2017 publication](http://shex.io/shex-semantics-20170713/) included a [definition of validation](#validation) which implied infinite recursion. This version explicitly includes recursion checks. 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](https://github.com/shexSpec/spec/issues), [language](https://github.com/shexSpec/shex/issues) and [test](https://github.com/shexSpec/shexTest/issues) issues. Only send comments to [public-shex@w3.org](mailto:public-shex@w3.org) ([subscribe](mailto:public-shex-request@w3.org?subject=subscribe), [archives](https://lists.w3.org/Archives/Public/public-shex/)) if you are unable to raise issues on GitHub. All comments are welcome.

## Conformance

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.

All ShEx documents MUST conform to the [Schema Requirements](#schema-requirements). Additional constraints for the specific types of ShEx documents (ShExC, ShExJ, and ShExR) follow:

* A ShExC document is a UTF-8 document which conforms to the grammar described in resulting in a valid ShExJ document.
* A ShExJ document is a valid JSON-LD document [[!JSON-LD]], and conforms to the ShExJ syntax, as described in .
* JSON documents can be interpreted as ShExJ by following the normative statements in Section 4.8 Interpreting JSON as JSON-LD in [[!JSON-LD]].
* A ShExR RDF document complies with this specification if it conforms to the schema in [ShExR.shex](#shexr).

## 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)](#shexc).

## Definitions, Acronyms, and Abbreviations

### Definitions

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)

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).
* def(Sch, label) is the decl.shapeExpr where decl.label = label. Sch must have exactly one def(Sch, label).

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 .

### Acronyms And Abbreviations

|  |  |
| --- | --- |
| BNF | Backus Naur Form |
| CSS | Cascading Stylesheets |
| IANA | Internet Assigned Numbers Authority |
| IRI | Internationalized Resource Identifier |
| RDF | Resource Description Framework |
| ShEx | Shape Expressions RDF schema language |
| ShExC | ShEx Compact syntax |
| ShExJ | ShEx JSON (or JSON-LD) syntax |
| ShExR | ShEx RDF syntax |
| SPARQL | RDF Query Language |
| URL | Uniform Resource Locator |
| UTF-8 | Unicode Transformation Format |
| XML | Extensible Markup Language |
| XPath | Path Language for XML |

## 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" }

### 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 typeExressions.
* 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.

The [executable JSON grammar for ShExJ](https://github.com/shexSpec/shexTest/blob/main/doc/ShExJ.jsg#L11) specifically disables the requirement for a matching "type" attribute in ObjectLiteral as "type" is instead used for the datatype of a [JSON-LD typed value](https://www.w3.org/TR/json-ld11/#dfn-typed-value).

### References

ShExJ is a dialect of JSON-LD [[!JSON-LD]] and the member id is used as a node identifier. An ShapeDecl or tripleExpr 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"  
 } } } } } ] }

### 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": [ SE1, … ] } indicates that both SE1 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.

### 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.

### Namespaces

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 Shape Expressions Language

A Shape Expressions (ShEx) schema is a collection of labeled Shapes and [Node Constraints](#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](../primer/index.html#associating-nodes-with-shapes).

### Shapes Schema

A shapes schema is captured in a Schema object with a list of Shape Declarationss:

|  |  |  |
| --- | --- | --- |
| Schema | { | imports:[IRIREF+]? startActs:[SemAct+]? start:shapeExpr? shapes:[ShapeDecl+]? } |
| ShapeDecl | { | id:shapeExprLabel abstract:BOOL? shapeExpr:shapeExpr | ShapeExternal } |

where shapes is a list of ShapeDecls.

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

ex:IssueShape { … }  
 \_:UserShape { … }  
 ex:EmployeeShape { … }

### Validation Definition

For a graph G, a schema Sch and a fixed ShapeMap ism, isValid(G, Sch, ism) indicates that for every shape association (node: n, shape: sl, exact: exact) in ism, the node n satisfies the shape expression identified by sl. If exact is true, the result is satisfies(n, def(s.label), G, Sch, completeTyping(G, Sch), neigh(n)) , otherwise, the result is satisfiesDescendant(n, def(s.label), G, Sch, completeTyping(G, Sch), neigh(n)). 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, l) where n is a node in G and l is a shape label that appears in the shape declarations of the schema. A correct typing is a typing such that for every RDF node/shape pair (n,l), satisfies(n, def(l), G, Sch, typing, neigh(n)) holds or satisfiesDescendant(n, def(l), G, Sch, typing, neigh(n)) holds. A completeTyping(G, Sch) is a unique correct typing that exists for every graph and every ShEx schema that satisfies the [schema requirements](#schema-requirements).

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 hierarchy and dependency graph of Sch. A stratification of a schema Sch with k strata is a function stratum that associates with every shape label from the shape declarations of Sch a natural number between 1 and k such that:

* If l1 and l2 belong to the same maximal strongly connected component, then stratum(l1) = stratum(l2).
* If there is a reference from l1 to l2 and l1 and l2 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](#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 RDF node/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 RDF node/shape pairs (n,s) with stratum(s) ≤ i
  + are equal to completeTypingOn(i-1, G, Sch) when restricted to their RDF node/shape pairs (n1,s1) for which stratum(s1) < i.

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

The definition of strongly connected component and maximal strongly connected component of a graph can be found at Wikipedia: <https://en.wikipedia.org/wiki/Strongly_connected_component>.

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 negation in Datalog.

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

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

### 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) 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) 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) is an extension mechanism allowing a ShapeDecl to denote an externally defined shapeExpr. It can be used to reference e.g. functional shapes or prohibitively large value sets.
* A shape reference (shapeExprLabel) identifies another shape in the schema or an [imported schema](#import).

#### JSON Syntax

|  |  |  |
| --- | --- | --- |
| shapeExpr | = | ShapeOr | ShapeAnd | ShapeNot | NodeConstraint | Shape | shapeExprRef ; |
| ShapeOr | { | shapeExprs:[shapeExpr{2,}] } |
| ShapeAnd | { | shapeExprs:[shapeExpr{2,}] } |
| ShapeNot | { | shapeExpr:shapeExpr } |
| ShapeExternal | { | } |
| ShapeExprRef | { | label:  shapeExprLabel exact:BOOL? } |
| shapeExprLabel | = | IRIREF | BNODE ; |

Examples of shape expressions:

{ "type": "Shape", … }

{ … }

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

IRI AND  
 (  
 @<http://schema.example/#IssueShape>  
 OR NOT { … }  
 )

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

#### Semantics

For a node n of the graph G, we define neigh(n) as the set of triples adjacent to n in the graph; these are the triples that have n either as subject or as object.

For a shape expression se we define its set of shapes nestedShapes(se) recursively on the structure of se:

* if se is a NodeConstraint, then nestedShapes(se) = emptyset
* if se is a Shape, then nestedShapes(se) = {se}
* if se is a ShapeNot, then nestedShapes(se) = shapes(se.shapeExpr)
* if se is a ShapeAnd or ShapeOr, then nestedShapes(se) is the union of the sets nestedShapes(se2) for all se2 in se.shapeExprs
* if se is a shapeExprRef with label lab, then nestedShapes(se) = nestedShapes(def(L))
* if se is a ShapeExternal, then nestedShapes(se) is the set of shapes denoted by se.

For shape expression labels label1, label2, we say that label2 directly extends label1 if

nestedShapes(def(label2))

contains a Shape s such that s.extends contains label2. The extension hierarchy graph of a shapes schema is a directed graph whose nodes are the shape expression labels of the schema and that has an edge from label2 to label1 whenever label2 directly extends label1.

satisfies: The expression satisfies(n, se, G, Sch, t, R) indicates that a node n, a subset R of neigh(n), and a graph G satisfy a shape expression se with typing t for schema Sch.  
satisfiesDescendant: The expression satisfiesDescendant(n, shapeExprLabel, G, Sch, t, R) indicates that n, a subset R of neigh(n), and G and some non-abstract child of shapeExprLabel in the extension hierarchy graph satisfies(n, child, G, Sch, t, R), with the given typing t.

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

* se is a NodeConstraint and [satisfies2](#satisfies2-NodeConstraint)(n, se) as described below in [Node Constraints](#node-constraints). Note that testing if a node satisfies a node constraint does not require a graph or typing.
* se is a Shape and matchesShape(n, S, G, Sch, m, R) is true.
* se is a ShapeOr and there is some shape expression se2 in se.shapeExprs such that satisfies(n, se2, G, Sch, t, R).
* se is a ShapeAnd and for every shape expression se2 in se.shapeExprs, satisfies(n, se2, G, Sch, t, R).
* se is a ShapeNot and for the shape expression se2 at se.shapeExpr, satisfies(n, se2, G, Sch, t, R) is false.
* se is a ShapeExternal and implementation-specific mechansims not defined in this specification indicate success.
* se is a ShapeExprRef. If ShapeExprRef.exact, satisfies(n, def(se.label), G, Sch, t, R), otherwise satisfiesDescendant(n, se.label, G, Sch, t, R).

Given the three shape expressions SE1, SE2, SE3 in a Schema Sch, such that:

* satisfies(n, SE1, G, Sch, m)
* satisfies(n, SE2, G, Sch, m)
* NOT satisfies(n, SE3, 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)
* NOT
* satisfies(
* 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 SE1 then the following holds:

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

In this example, EmployeeShape directly extends PersonShape and transitively extends EntityShape

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

ex:EntityShape {  
  
  
  
 ex:entityId .  
 }  
  
 ex:PersonShape EXTENDS @ex:EntityShape {  
  
  
  
  
 foaf:name .  
 }  
  
 ex:EmployeeShape EXTENDS @ex:PersonShape {  
  
  
  
  
 ex:employeeNumber .  
 }

In this example, UserShape directly extends PersonShape, and PersonShape directly references a conjunct which directly extends EntityShape. Through this, UserShape transitively extends EntityShape.

{ "type": "Schema",  
 "shapes": [  
 { "id": "http://schema.example/#EntityShape",  
 "type": "Shape",  
 "closed": true,  
 "expression": {  
 "type": "TripleConstraint",  
 "predicate": "http://schema.example/#entityId"  
 } },  
 { "id": "http://schema.example/#PersonShape",  
 "type": "ShapeAnd",  
 "shapeExprs": [  
 { "type": "Shape",  
 "extends": [ "http://schema.example/#EntityShape" ],  
 "closed": true,  
 "expression": {  
 "type": "TripleConstraint",  
 "predicate": "http://xmlns.com/foaf/0.1/name"  
 } },  
 { "type": "Shape",  
 "expression": {  
 "type": "TripleConstraint",  
 "predicate": "http://schema.example/#entityId",  
 "valueExpr": {  
 "type": "NodeConstraint",  
 "datatype": "http://www.w3.org/2001/XMLSchema#integer"  
 } } }  
 ] },  
 { "id": "http://schema.example/#UserShape",  
 "type": "Shape",  
 "extends": [ "http://schema.example/#PersonShape" ],  
 "expression": {  
 "type": "TripleConstraint",  
 "predicate": "http://schema.example/#userId"  
 } }  
] }

<#EntityShape> CLOSED {  
  
  
  
  
 :entityId .  
 }  
 <#PersonShape> EXTENDS @<#EntityShape> CLOSED {  
  
  
  
   
  
  
  
 foaf:name .  
 } AND {  
  
  
  
 :entityId  
  
  
 xsd:integer  
 }  
  
 <#UserShape> EXTENDS @<PersonShape>{  
  
   
  
  
 :userId .  
 }

### Node Constraints

|  |  |  |
| --- | --- | --- |
| NodeConstraint | { | 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 \*/ } |

#### 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](#nodeKind)
* [Datatype Constraints](#datatype)
* [XML Schema String Facet Constraints](#xs-string)
* [XML Schema Numeric Facet Constraints](#xs-numeric)
* [Values Constraints](#values)

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

ex:IssueShape {  
 ex:state 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](#matches-cardinality) violation described below.

#### Datatype Constraints

For a node n and constraint value v, nodeSatisfies(n, v) if n is a 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"  
 } } } ] }

ex:IssueShape {   
 ex:submittedOn xsd: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. |

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"  
 } } } ] }

ex:IssueShape {  
 rdfs:label rdf: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. |

#### 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](#blank-node-label) 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](https://www.w3.org/TR/xpath-functions-31/#matching-start-and-end) 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 } } } ] }

ex:IssueShape {  
 ex:submittedBy 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. |

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" }  
} } ] }

ex:IssueShape {  
 ex:submittedBy /genuser[0-9]+/i  
 }

<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\\?$" }  
} } ] }

ex:ProductShape {  
 ex:trademark /^\/\t\\\U0001D4B8\?$/  
 }

<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). |

#### 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 } } } ] }

ex:IssueShape {  
 ex:confirmations 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. |

#### Values Constraint

The nodeSatisfies semantics for NodeConstraint values depends on a nodeIn function [defined below](#nodeIn).

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](http://www.ietf.org/rfc/rfc4647.txt) [[!rfc4647]] section 2.1 and l matches st per the basic filtering scheme defined in [[!rfc4647]] section 3.3.1. The basic language range wildcard ("\*") is represented by an empty stem ("").

Values Constraint example 1

NoActionIssueShape requires a state of Resolved or Rejected:

{ "type": "Schema", "shapes": [  
 { "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" ] } } } ] }

ex:NoActionIssueShape {  
 ex:state [ ex:Resolved ex: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": [  
 { "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" }  
 ] }  
 ] } } } ] }

ex:EmployeeShape {  
 foaf:mbox [ "N/A"  
 <mailto:engineering->~  
 <mailto:sales->~  
 - <mailto:sales-contacts>~  
 - <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-":

{ "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-" }  
 ] }  
 ] } } } ] }

ex:EmployeeShape {  
 foaf:mbox [ . - <mailto:engineering->~ - <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.

### 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.

#### JSON Syntax

|  |  |  |
| --- | --- | --- |
| Shape | { | extends:[shapeExprRef]? 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 ; |

#### Semantics

The semantics of the matchesShape function are based on the matches function [defined below](#matches). A shape may have an expression. For the purposes of evaluation, we define an EmptyExpression which has no TripleConstraints.

parentShapeLabels is a function from a shape label to the set of shapeExprLabels parents of the labels in shape.extends as well as their parents in the extension hierarchy graph.

For a node n, shape S, graph G, a ShExSchema Sch, a typing m, and a subset R of neigh(n), matchesShape(n, S, G, Sch, m, R) if and only if:

* parents is the set parentShapeLabels(S). If s.extends does not exist, then parentShapeLabels(s) is the empty set.
* nCard is the length of parents.
* R can be partitioned into two sets matched and remainder
* matched is partitioned into nCard+1 parts R0, R1 ... RnCard such that
  + matches(R0, S.tripleExpr, m)
  + satisfies(n, constraint(L), G, Sch, m, matched)
  + for every i in 1..nCard, matches(Ri, mainShape(parentsi), m)
  + for every i in 1..nCard, satisfies(n, constraint(parenti), G, Sch, m, Ri ∪ Q) where Q is the union of all the Rj s.t. parentj is a parent of parenti.
* 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 one of the mainShape(parentsi) or in S.expression.
* There is no triple in matchables which matches a TripleConstraint in one of the mainShape(parentsi) nor one of the TripleConstraint in S.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.
* The complexity of partitioning is described briefly in the [ShEx2 Primer](../primer/#partition-complexity).

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](#semantic-actions).
* matches(
* T,
* { "type": "OneOf", "shapeExprs": [te1, te2, …], "min": 2, "max": 3,   
   "semActs": [SemAct1, SemAct2, …] }
* (te1 | te2, …) {2,3}    
   %<SemAct1>% %<SemAct2>% …
* ,  
  m)
* evaluates as:
* matches(
* T,
* { "type": "OneOf", "shapeExprs": [te1, te2, …], "min": 2, "max": 3 }
* (te1 | te2, …) {2,3}
* ,  
  m)
* and [semActsSatisfied](#semActsSatisfied)([SemAct1, SemAct2, …])
* 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 T1, T2,…Tk such that min ≤ k ≤ max and for each Tn, matches(Tn, expr, m) by the remaining rules in this list.
* matches(
* T,
* { "type": "OneOf", "shapeExprs": [te1, te2, …], "min": 2, "max": 3 }
* (te1 | te2, …) {2,3}
* ,  
  m)
* evaluates as:
* Let e =
* { "type": "OneOf", "shapeExprs": [te1, te2, …] }
* (te1 | te2, …)
* (matches(T1, e, m) and matches(T2, e, m)  
   and T = T1 ∪ T2)  
  or  
  (matches(T1, e, m) and matches(T2, e, m) and matches(T3, e, m)  
   and T = T1 ∪ T2 ∪ T3)
* 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" }  
  ] }
* (te3 ; te4 ; …)  
   | <http://xmlns.com/foaf/0.1/name> . +
* ,  
  m)
* evaluates as:
* matches(
* T,
* { "type": "EachOf", "shapeExprs": [te3, te4, …] }
* te3 ; te4 ; …
* ,  
  m)
* or matches(
* T,
* { "type": "TripleExpression", "min": 1, "max": -1,   
   "predicate": "http://xmlns.com/foaf/0.1/name" }
* <http://xmlns.com/foaf/0.1/name> . +
* ,  
  m)
* expr is an EachOf and there is some partition of T into T1, T2,… such that for every expression expr1, expr2,… in shapeExprs, matches(Tn, exprn, 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" }  
  ] }
* <http://xmlns.com/foaf/0.1/givenName> . ;  
    
  <http://xmlns.com/foaf/0.1/familyName> .
* ,  
  m)
* evaluates as:
* matches(
* T1,
* { "type": "TripleExpression",  
   "predicate": "http://xmlns.com/foaf/0.1/givenName" }
* <http://xmlns.com/foaf/0.1/givenName> .
* ,  
  m)
* and matches(
* T2,
* { "type": "TripleExpression",  
   "predicate": "http://xmlns.com/foaf/0.1/familyName" }
* <http://xmlns.com/foaf/0.1/familyName> .
* ,  
  m)
* and T = T1 ∪ T2
* 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",  
       "predicate": "http://xmlns.com/foaf/0.1/givenName" }
    - <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 expr.valueExpr is a shapeExprRef, then shapeExprRef.label is in m(value)
    - or expr.valueExpr is not a shapeExprRef, then satisfies(value, valueExpr, G, Sch, m, neigh(value)).
    - matches(
    - T,
    - { "type": "TripleConstraint", "inverse": true,  
       "predicate": "http://purl.org/dc/elements/1.1/author",  
       "valueExpr": "http://schema.example/#IssueShape" }
    - ^<http://purl.org/dc/elements/1.1/author>  
       @<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](#tripleExprRef-requirement) below.
* For the schema
* { "type": "Schema", "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" } } ] } },  
   { "id": "http://schema.example/#PersonShape",  
   "type": "Shape", "expression": {  
   "id": "http://schema.example/#nameExpr",  
   "type": "TripleConstraint",  
   "predicate": "http://xmlns.com/foaf/0.1/name" } } ] }
* <http://schema.example/#EmployeeShape> {  
   &<http://schema.example/#nameExpr> ;  
   <http://schema.example/#empID>  
   <http://www.w3.org/2001/XMLSchema#integer>  
    
    
    
  }  
  <http://schema.example/#PersonShape> {  
    
   $<http://schema.example/#nameExpr>  
   <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)

### 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:  
{ "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" } } ] }

schema1:    
<http://schema.example/#EmployeeShape> {  
 &<http://schema.example/#nameExpr> ;  
 <http://schema.example/#empID>  
 <http://www.w3.org/2001/XMLSchema#integer>  
  
  
  
  
}  
schema2:  
<http://schema.example/#PersonShape> {  
  
 $<http://schema.example/#nameExpr>  
 <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

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,  
 } } ] }

schema1:    
  
IMPORT <http://schema.example/schema2>  
IMPORT <http://schema.example/schema3>  
<http://schema.example/schema1#S1> {  
  
 <http://schema.example/#p1> @<http://schema.example/schema1#S2>  
  
  
schema2:  
  
IMPORT <http://schema.example/schema3>  
  
<http://schema.example/schema1#S2> {  
  
 <http://schema.example/#p2> @<http://schema.example/schema1#S3>  
  
  
schema3:  
  
IMPORT <http://schema.example/schema1>  
  
<http://schema.example/schema1#S3> {  
  
 <http://schema.example/#p3> @<http://schema.example/schema1#S1>?

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

Import example 3 - Ignored Start In Import

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:    
  
IMPORT <http://schema.example/schema2>  
  
<http://schema.example/schema1#S1> {  
  
 <http://schema.example/#p1> @<http://schema.example/schema1#S2>  
  
}  
schema2:  
  
start=@<http://schema.example/schema1#S2>  
  
<http://schema.example/schema1#S2> {  
  
 <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

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"  
 } } ] }

schema1:    
  
IMPORT <http://schema.example/schema2>  
  
<http://schema.example/schema1#S1> {  
  
 <http://schema.example/#p1> @<http://schema.example/schema1#S2>  
  
}  
schema2:  
  
%@<http://schema.example/schema1#A1>%  
  
  
<http://schema.example/schema1#S1> {  
  
 <http://schema.example/#p1> @<http://schema.example/schema1#S2>  
  
}  
<http://schema.example/schema1#S2> {  
  
 <http://schema.example/#p2> @<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.

### 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.

#### 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](#semActsSatisfied).
2. Every node n in m conforms to its associated shapeExprRefs sen where for each shapeExprRef sei in sen:
   1. sei references a ShapeExpr in shapes, and
   2. satisfies(n, sei, G, Sch, m) for each shape sei in sen.

#### Shape Expression Reference Requirement

A shapeExprRef MUST appear in the schema's shapes map (or an [imported schema's](#import) 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:

{"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"  
 }  
 } ]  
 } ]  
}

ex:PersonShape {  
  
  
  
 foaf:name .  
 }  
  
 ex:EmployeeShape  
  
 @ex:PersonShape AND {  
  
  
  
 ex:employeeNumber .  
 }

{"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  
 }  
 } ]  
}

ex:PersonShape {  
  
  
  
 foaf:name .  
 }  
  
 ex:EmployeeShape {  
  
  
  
 ex:dependent  
 @ex:PersonShape  
 \*  
  
 }

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

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

ex:S1 {  
  
 &ex:MissingShapeExpr  
 }

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

{"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"  
 }  
 } ]  
}

ex:CustomerShape {  
  
  
 $ex:discountExpr  
  
 ex:discount .  
 }  
  
 ex:EmployeeShape {  
  
  
  
 ex:contactFor  
 @ex:discountExpr  
 }

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

{"type" : "Schema",  
 "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"  
 }  
 } ]  
 } ] }

ex:PersonShape  
  
 @ex:EmployeeShape AND {  
  
  
  
 foaf:name .  
 }  
  
  
 ex:EmployeeShape  
  
 @ex:PersonShape AND {  
  
  
  
 ex:employeeNumber .  
 }

#### 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" }  
] } } ] }

ex:PersonShape {  
 $ex:nameExpr foaf:name .  
 }  
  
  
  
 ex:EmployeeShape {  
 &ex:nameExpr ;  
 ex:employeeNumber .  
 }

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"  
} ] }

ex:S1 {  
 &ex:missingTripleExpr  
 }

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" }  
] } } ] }

ex:CustomerShape {  
 …; …  
 }  
 ex:PreferredCustomerShape {  
 &ex:CustomerShape ;  
 ex:discount .  
 }

#### shapeExprRef non-abstract shape requirement

Every shapeExprRef |referer| MUST identify at least one non-abstract shape.

Following is a valid example with a shape with a shapeExprRef that references an abstract shape with two non-abstract descendants:

{"type" : "Schema",  
 "shapes" : [ {  
 "id": "http://schema.example/#IssueShape",  
 "type": "Shape",  
 "expression": {  
 "type": "TripleConstraint",  
 "predicate": "http://schema.example/#approvedBy",  
 "valueExpr": "http://schema.example/#EngineerShape"  
 }  
 }, {  
 "id" : "http://schema.example/#EntityShape",  
 "type" : "Shape", "abstract": true,  
 "expression" : {  
 "type" : "TripleConstraint",  
 "predicate" : "http://schema.example/#entityId"  
 }  
 }, {  
 "id" : "http://schema.example/#PersonShape",  
 "type" : "Shape",  
 "extends" : [ "http://schema.example/#EntityShape" ],  
 "expression" : {  
 "type" : "TripleConstraint",  
 "predicate" : "http://xmlns.com/foaf/0.1/name"  
 }  
 }, {  
 "id" : "http://schema.example/#EmployeeShape",  
 "type" : "Shape",  
 "extends" : [ "http://schema.example/#PersonShape" ],  
 "expression" : {  
 "type" : "TripleConstraint",  
 "predicate" : "http://schema.example/#employeeNumber"  
 }  
} ] }

ex:IssueShape {  
  
  
  
 ex:approvedBy  
 @ex:EntityShape  
 }  
  
 ABSTRACT ex:EntityShape {  
  
  
  
 ex:entityId .  
 }  
  
 ex:PersonShape EXTENDS @ex:EntityShape {  
  
  
  
  
 foaf:name .  
 }  
  
 ex:EmployeeShape EXTENDS @ex:EntityShape {  
  
  
  
  
 ex:employeeNumber .  
 }

This shapeExprRef is invalid because it references only abstract descendants:

{"type" : "Schema",  
 "shapes" : [ {  
 "id": "http://schema.example/#IssueShape",  
 "type": "Shape",  
 "expression": {  
 "type": "TripleConstraint",  
 "predicate": "http://schema.example/#approvedBy",  
 "valueExpr": "http://schema.example/#EngineerShape"  
 }  
 }, {  
 "id" : "http://schema.example/#EntityShape",  
 "type" : "Shape", "abstract": true,  
 "expression" : {  
 "type" : "TripleConstraint",  
 "predicate" : "http://schema.example/#entityId"  
 }  
 }, {  
 "id" : "http://schema.example/#PersonShape",  
 "type" : "Shape", "abstract": true,  
 "extends" : [ "http://schema.example/#EntityShape" ],  
 "expression" : {  
 "type" : "TripleConstraint",  
 "predicate" : "http://xmlns.com/foaf/0.1/name"  
 }  
 }, {  
 "id" : "http://schema.example/#EmployeeShape",  
 "type" : "Shape", "abstract": true,  
 "extends" : [ "http://schema.example/#PersonShape" ],  
 "expression" : {  
 "type" : "TripleConstraint",  
 "predicate" : "http://schema.example/#employeeNumber"  
 }  
} ] }

ex:IssueShape {  
  
  
  
 ex:approvedBy  
 @ex:EntityShape  
 }  
  
 ABSTRACT ex:EntityShape {  
  
  
  
 ex:entityId .  
 }  
  
 ABSTRACT ex:PersonShape EXTENDS @ex:EntityShape {  
  
  
  
  
 foaf:name .  
 }  
  
 ABSTRACT ex:EmployeeShape EXTENDS @ex:EntityShape {  
  
  
  
  
 ex:employeeNumber .  
 }

#### Negation Requirement

A schema MUST NOT contain any shapeExprLabel that has a negated reference to itself, either directly or transitively. This is formalized by the requirement that the hierarchy and 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 atomicTripleConstraints 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 expression s1 has a reference to the shape label l2 if

* there is a shape sh in atomicShapes(s1) and
* there is a triple constraint tc in atomicTripleConstraints(sh) and
* tc.valueExpr is present and
* tc.valueExpr contains a shape reference to l2.

The reference from s1 to l2 is a negated reference if the reference to l2 appears under an odd number of ShapeNot in tc.valueExpr.

The hierarchy and dependency graph of a schema is the graph whose nodes are the shape labels that appear in the shape declarations of the schema, and that has an edge from l1 to l2 if:

* the definition of l1 has a reference to l2, or
* there is an edge from l1 to l2 in the hieararchy graph, or
* there is an edge from l2 to l1 in the hieararchy graph.

The edge from l1 to l2 is negative if the definition of l1 has a negative reference to l2, otherwise the edge is positive.

### 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" } } }  
 ] }

ex:S {ex:p NOT @ex: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" } }  
 ] }

ex:US {ex:Up NOT @ex:UT}   
  
  
  
  
  
  
ex:UT { ex:Uq @ex: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"  
 }  
 } ] }

ex:S {  
  
  
  
 ex:p   
  
 NOT  
 @ex:T  
 }   
  
  
 ex:T {  
  
  
  
 ex:q  
 @ex: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"  
 } ] }

ex:T {  
  
  
  
 ex:p  
 @ex:S  
 }  
  
 ex:S  
  
  
 (NOT  
 @ex:T)  
 AND @ex:U  
  
 ex:U .

This doubly-negated self-reference of ex:T does not violate 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" : "ShapeNot",  
 "shapeExpr" : {  
 "type" : "ShapeAnd",  
 "shapeExprs" : [ {  
 "type" : "ShapeNot",  
 "shapeExpr" : "http://schema.example/#T"  
 }, "http://schema.example/#U" ]  
 }  
 }, {  
 "id" : "http://schema.example/#U",  
 "type" : "Shape"  
 } ] }

ex:T {  
  
  
  
 ex:p  
 @ex:S  
 }  
  
 ex:S  
 NOT  
  
  
 (  
 (NOT  
 @ex:T)  
 AND @ex:U )  
  
  
 ex:U .

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",  
 "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" ]  
 } ] }

ex:T {  
  
  
  
 ex:p  
  
 NOT  
 @ex:S  
 }  
  
  
 ex:U {  
  
  
  
 ex:q  
 @ex:S  
 }  
  
 ex:S  
  
  
 (NOT  
 @ex:T)  
 AND @ex:U

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

{"type" : "Schema",  
 "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" ]  
 } ] }

ex:T {  
  
  
  
 ex:p  
 NOT  
  
 @ex:S  
 }  
  
  
 ex:U {  
  
  
  
 ex:q .  
 }  
  
 ex:S  
  
  
 (NOT  
 @ex:T)  
 AND @ex:U

### Examples with Shape.extra predicate

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

{ "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"  
} } ] }

ex:S EXTRA ex:p {  
 ex:p @ex:S  
 }

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

{ "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"  
} } } ] }

ex:S EXTRA ex:p {  
 ex:p NOT @ex:S  
 }

#### Acyclic Extension Requirement

The extension hierarchy graph must be acyclic.

#### Extension Coherence

A shapeDecl D with label L and D.shapeExpr se is called extendable if it satisfies all of:

* it is of the form either `s` or ShapeAnd(s, se), where `s` is a Shape and se is a shapeExpr. In this case we denote `s` as `mainShape(L)` and se as `constraint(L)`:
* `def(L')` is an extendable shape expression for every `L'` in `s.extends` (note that this condition is trivially met when `s.extends` is empty),
* the set `predicates(se)` is included the union of the sets `predicates(mainShape(L'))` for all shape expression names `L'` that belong to `parentShapeLabels(L)`.

**Schema requirement** EXTENDS appears only in extendable shape expressions. That is, for every Shape `s` that appears in the schema, if `s.extends` is non empty and for every shapeExpr se in the schema, if s belongs to nestedShapes(se), then se is an extendable shape expression.

### 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? } |

#### Semantics

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

#### Use - informative

A practical evaluation of a SemAct will provide access to some context. For instance, the [http://shex.io/extensions/Test/](http://shex.io/extensions/Test/#def) 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/" } ] } } ] }

ex:S1 {  
 ex:p1 .+ %Test:{ print(s) %} %Test:{ print(o) %}  
 }

<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 |  |

### 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 } |

#### 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"} } ] } } ] }

ex:IssueShape {  
 ex:status .  
 // rdfs:comment "Represents reported software issues."  
 // rdfs:label "software issue"  
 }

### Validation Examples

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

#### Simple Examples

Schema:

S1  
  
nc1

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

ex:IntConstraint  
  
 xsd:integer

Here the shape identified by http://schema.example/#IntConstraint is a shape expression consisting of a single NodeConstraint. Per [Shape Expression Semantics](#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](#satisfies-NodeConstraint) ("30"^^xsd:integer, S1, G, m)
  + [satisfies2 NodeConstraint](#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"  
 } } ] }

ex:UserShape {  
 ex:shoeSize .  
 }

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](#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](#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  
tc1  
  
  
  
nc1

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

ex:UserShape EXTRA a {  
 a  
  
  
  
  
 [ex:Teacher]  
 }

Data:

t1  
t2  
t3  
t4  
t5

BASE <http://a.example/>  
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>  
<Alice> ex:shoeSize "30"^^xsd:integer .  
<Alice> a ex:Teacher .  
<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](#satisfies-Shape) (<Alice>, S1, G, m)
  + neigh(G, <Alice>) = [t1,t2,t3,t4], matched = [t2], remainder = [t1,t3]
  + [matches TripleConstraint](#matches-TripleConstraint) ([t2], tc1, m)
    - [satisfies NodeConstraint](#satisfies-NodeConstraint) (ex:Teacher, nc1, G, m)
      * [satisfies2 NodeConstraint](#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"].

#### Disjunction Example

Schema:

S1  
te1  
tc1  
  
  
nc1  
te2  
tc2  
  
  
nc2  
tc3  
  
  
nc3

{ "type": "Schema", "shapes": [  
 { "id": "http://schema.example/#UserShape",  
 "type": "Shape", "expression":  
 {"type": "OneOf", "expressions": [  
 { "type": "TripleConstraint",  
 "predicate": "http://xmlns.com/foaf/0.1/name",  
 "valueExpr":  
 { "type": "NodeConstraint", "nodeKind": "literal" } },  
 { "type": "EachOf", "expressions": [  
 { "type": "TripleConstraint", "min": 1, "max": -1 ,  
 "predicate": "http://xmlns.com/foaf/0.1/givenName",  
 "valueExpr":  
 { "type": "NodeConstraint", "nodeKind": "literal" } },  
 { "type": "TripleConstraint",  
 "predicate": "http://xmlns.com/foaf/0.1/familyName",  
 "valueExpr":  
 { "type": "NodeConstraint", "nodeKind": "literal" } }  
 ] }  
 ] }  
 } ] }

ex:UserShape  
 {  
 ( # extra ()s to clarify alignment with ShExJ  
 foaf:name  
  
  
 LITERAL |  
 ( # extra ()s to clarify alignment with ShExJ  
 foaf:givenName  
  
  
 LITERAL+ ;  
 foaf:familyName  
  
  
 LITERAL  
 )  
 )  
 }

Data:

t1  
t2  
t3  
t4  
t5  
t6

BASE <http://a.example/>  
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
<Alice> foaf:givenName "Alice" .  
<Alice> foaf:givenName "Malsenior" .  
<Alice> foaf:familyName "Walker" .  
<Alice> foaf:mbox <mailto:alice@example.com> .  
<Bob> foaf:knows <Alice> .  
<Bob> foaf:mbox <mailto:bob@example.com> .

Per [Shape Expression Semantics](#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](#satisfies-Shape) (<Alice>, S1, G, m)
  + neigh(G, <Alice>) = [t1,t2,t3,t4,t5], matched = [t1,t2,t3], remainder = [t4,t5]
  + [matches OneOf](#matches-OneOf) ([t1,t2,t3], te1, m)
    - [matches EachOf](#matches-EachOf) ([t1,t2,t3], te2, m)
      * [matches cardinality](#matches-cardinality) ([t1,t2], tc2, m)
        + [matches TripleConstraint](#matches-TripleConstraint) ([t1], tc2, m)

[satisfies NodeConstraint](#satisfies-NodeConstraint) ("Alice", nc2, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) ("Alice", nc2)

* + - * + [matches TripleConstraint](#matches-TripleConstraint) ([t2], tc2, m)

[satisfies NodeConstraint](#satisfies-NodeConstraint) ("Malsenior", nc2, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) ("Malsenior", nc2)

* + - * [matches TripleConstraint](#matches-TripleConstraint) ([t3], tc3, m)
        + [satisfies NodeConstraint](#satisfies-NodeConstraint) ("Walker", nc3, G, m)

[satisfies2 NodeConstraint](#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  
t5  
t6  
t7

BASE <http://a.example/>  
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
<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](#satisfies-Shape) (<Alice>, S1, G, m)
  + neigh(G, <Alice>) = [t4,t5,t7], matched = [t7], remainder = [t4,t5]
  + [matches OneOf](#matches-OneOf) ([t7], te1, m)
    - [matches TripleConstraint](#matches-TripleConstraint) ([t7], tc1, m)
      * [satisfies NodeConstraint](#satisfies-NodeConstraint) ("Walker", nc3, G, m)
        + [satisfies2 NodeConstraint](#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 is 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](#satisfies-Shape) (<Alice>, S1, G, m)
  + neigh(G, <Alice>) = [t4,t5,t7], matched = [t7], remainder = [t4,t5]
  + [matches OneOf](#matches-OneOf) ([t7], te1, m)
    - [matches TripleConstraint](#matches-TripleConstraint) ([t7], tc1, m)
      * [satisfies NodeConstraint](#satisfies-NodeConstraint) ("Walker", nc3, G, m)
        + [satisfies2 NodeConstraint](#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](#satisfies-Shape) (<Alice>, S1, G, m)
  + …
  + unmatchables = [t5], closed is true

#### 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" ] } } }  
 ] }

ex:IssueShape  
 {  
  
 ex:reproducedBy  
  
 @ex:TesterShape }  
ex:TesterShape  
 {  
  
 ex:role  
  
 [ex:testingRole] }

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](#satisfies-Shape) (inst:Issue1, S1, G, m)
  + neigh(G, inst:Issue1) = [t1,t2], matched = [t1,t2], remainder = Ø
  + [matches TripleConstraint](#matches-TripleConstraint) ([t1], tc1, m)
    - [satisfies NodeConstraint](#satisfies-NodeConstraint) (inst:Tester2, nc1, G, m)
      * [satisfies2 NodeConstraint](#satisfies2-NodeConstraint) (inst:Tester2, nc1)
        + [satisfies Shape](#satisfies-Shape) (inst:Tester2. S2, G, m)

neigh(G, inst:Tester2) = [t2], matched = [t2], remainder = Ø

[matches TripleConstraint](#matches-TripleConstraint) ([t2], tc2, m)

[satisfies NodeConstraint](#satisfies-NodeConstraint) (ex:testingRole, nc2, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) (ex:testingRole, nc2)

outs = Ø

matchables = Ø, unmatchables = Ø, closed is false

* + outs = Ø
  + matchables = Ø, unmatchables = Ø, closed is false

#### 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"  
 } } ] }

ex:IssueShape {  
  
  
 ex:related  
  
 @ex:IssueShape\*  
}

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](#satisfies-Shape) (inst:Issue1, S1, G, m)
  + neigh(G, inst:Issue1) = [t1], matched = [t1], remainder = Ø
  + [matches TripleConstraint](#matches-TripleConstraint) ([t1], tc1, m)
    - [satisfies NodeConstraint](#satisfies-NodeConstraint) (inst:Issue2, nc1, G, m)
      * [satisfies2 NodeConstraint](#satisfies2-NodeConstraint) (inst:Issue2, nc1)
        + [satisfies Shape](#satisfies-Shape) (inst:Issue2. S2, G, m)

neigh(G, inst:Issue2) = [t3], matched = [t3], remainder = Ø

[matches TripleConstraint](#matches-TripleConstraint) ([t3], tc3, m)

[satisfies NodeConstraint](#satisfies-NodeConstraint) (inst:Issue3, nc3, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) (inst:Issue3, nc3)

[satisfies Shape](#satisfies-Shape) (inst:Issue3. S2, G, m)

neigh(G, inst:Issue3) = [t3], matched = [t3], remainder = Ø

[matches TripleConstraint](#matches-TripleConstraint) ([t3], tc3, m)

[satisfies NodeConstraint](#satisfies-NodeConstraint) (inst:Issue1, nc3, G, m)

[satisfies2 NodeConstraint](#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

#### 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"} ] } }  
 ] } } ] }

<http://schema.example/#TestResultsShape>  
{  
  
  
 <http://schema.example/#val>  
  
  
 ["a" "b" "c"]+ ;  
  
 <http://schema.example/#val>  
  
  
 ["b" "c" "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](#satisfies-Shape) (<s>, S1, G, m)
  + neigh(G, <s>) = [t1,t2,t3,t4], matched = [t1,t2,t3,t4], remainder = Ø
  + [matches EachOf](#matches-EachOf) ([t1,t2,t3,t4], te1, m)
    - [matches cardinality](#matches-cardinality) ([t1,t2,t3], tc1, m)
      * [matches TripleConstraint](#matches-TripleConstraint) ([t1], tc1, m)
        + [satisfies NodeConstraint](#satisfies-NodeConstraint) ("a", nc1, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) ("a", nc1)

* + - * [matches TripleConstraint](#matches-TripleConstraint) ([t2], tc1, m)
        + [satisfies NodeConstraint](#satisfies-NodeConstraint) ("b", nc1, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) ("b", nc1)

* + - * [matches TripleConstraint](#matches-TripleConstraint) ([t3], tc1, m)
        + [satisfies NodeConstraint](#satisfies-NodeConstraint) ("c", nc1, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) ("c", nc1)

* + - [matches cardinality](#matches-cardinality) ([t4], tc2, m)
      * [matches TripleConstraint](#matches-TripleConstraint) ([t4], tc2, m)
        + [satisfies NodeConstraint](#satisfies-NodeConstraint) ("d", nc2, G, m)

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

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) ("a", nc1)

* + - * [matches TripleConstraint](#matches-TripleConstraint) ([t2], tc1, m)
        + [satisfies NodeConstraint](#satisfies-NodeConstraint) ("b", nc1, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) ("b", nc1)

* + - [matches cardinality](#matches-cardinality) ([t3], tc2, m)
      * [matches TripleConstraint](#matches-TripleConstraint) ([t3], tc2, m)
        + [satisfies NodeConstraint](#satisfies-NodeConstraint) ("d", nc2, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) ("d", nc2)

* + outs = Ø
  + matchables = Ø, unmatchables = Ø, closed is false

#### Repeated Property With Dependent Shapes Example

Schema:

S1  
te1  
  
tc1  
  
nc1  
  
tc2  
  
nc2  
  
S2  
  
tc3  
  
  
nc3  
  
S3  
  
tc4  
  
  
nc4

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

ex:IssueShape {  
  
  
  
 ex:reproducedBy  
  
 @ex:TesterShape;  
  
 ex:reproducedBy  
  
 @ex:ProgrammerShape  
}  
ex:TesterShape {  
  
  
 ex:role  
  
 [ex:testingRole]  
}  
ex:ProgrammerShape {  
  
  
 ex:department  
  
 [ex:ProgrammingDepartment]  
}

Data:

t1  
t2  
  
  
t3  
  
  
t4  
t5

PREFIX ex: <http://schema.example/#>  
PREFIX inst: <http://inst.example/>  
inst:Issue1  
 ex:reproducedBy inst:Tester2 ;  
 ex:reproducedBy inst:Testgrammer23 .  
  
inst:Tester2   
 ex:role ex:testingRole .  
  
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](#satisfies-Shape) (inst:Issue1, S1, G, m)
  + neigh(G, inst:Issue1) = [t1,t2], matched = [t1,t2], remainder = Ø
  + [matches EachOf](#matches-EachOf) ([t1,t2], te1, m)
    - [matches TripleConstraint](#matches-TripleConstraint) ([t1], tc1, m)
      * [satisfies NodeConstraint](#satisfies-NodeConstraint) (inst:Tester2, nc1, G, m)
        + [satisfies2 NodeConstraint](#satisfies2-NodeConstraint) (inst:Tester2, nc1)

[satisfies Shape](#satisfies-Shape) (inst:Tester2. S2, G, m)

neigh(G, inst:Tester2) = [t3], matched = [t3], remainder = Ø

[matches TripleConstraint](#matches-TripleConstraint) ([t3], tc3, m)

[satisfies NodeConstraint](#satisfies-NodeConstraint) (ex:testingRole, nc3, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) (ex:testingRole, nc3)

outs = Ø

matchables = Ø, unmatchables = Ø, closed is false

* + - [matches TripleConstraint](#matches-TripleConstraint) ([t2], tc1, m)
      * [satisfies NodeConstraint](#satisfies-NodeConstraint) (inst:Testgrammer23, nc2, G, m)
        + [satisfies2 NodeConstraint](#satisfies2-NodeConstraint) (inst:Testgrammer23, nc2)

[satisfies Shape](#satisfies-Shape) (inst:Testgrammer23. S3, G, m)

neigh(G, inst:Testgrammer23) = [t5], matched = [t5], remainder = Ø

[matches TripleConstraint](#matches-TripleConstraint) ([t5], tc3, m)

[satisfies NodeConstraint](#satisfies-NodeConstraint) (ex:testingRole, nc4, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) (ex:testingRole, nc4)

outs = Ø

matchables = Ø, unmatchables = Ø, closed is false

* + outs = Ø
  + matchables = Ø, unmatchables = Ø, closed is false

#### 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  
te1  
tc1  
  
  
nc1  
  
tc2

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

<http://schema.example/#TestResultsShape>  
{  
  
  
 <http://schema.example/#p1>  
  
  
 ["a" "b"] + ;  
  
 <http://schema.example/#p2> . {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](#satisfies-Shape) (<s>, S1, G, m)
  + neigh(G, <s>) = [t1], matched = [t1], remainder = Ø
  + [matches EachOf](#matches-EachOf) ([t1], te1, m)
    - [matches cardinality](#matches-cardinality) ([t1], tc1, m)
      * [matches TripleConstraint](#matches-TripleConstraint) ([t1], tc1, m)
        + [satisfies NodeConstraint](#satisfies-NodeConstraint) ("a", nc1, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) ("a", nc1)

* + - [matches cardinality](#matches-cardinality) ([], tc2, m)
      * [matches TripleConstraint](#matches-TripleConstraint) ([], tc2, m)
  + outs = Ø
  + matchables = Ø, unmatchables = Ø, closed is false

If we add a t2 which matches tc2:

Data:

t1  
t2

BASE <http://a.example/>  
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](#satisfies-Shape) (<s>, S1, G, m)
  + neigh(G, <s>) = [t1], matched = [t1], remainder = Ø
  + [matches EachOf](#matches-EachOf) ([t1], te1, m)
    - [matches cardinality](#matches-cardinality) ([t1], tc1, m)
      * [matches TripleConstraint](#matches-TripleConstraint) ([t1], tc1, m)
        + [satisfies NodeConstraint](#satisfies-NodeConstraint) ("a", nc1, G, m)

[satisfies2 NodeConstraint](#satisfies2-NodeConstraint) ("a", nc1)

* + - [matches cardinality](#matches-cardinality) ([t2], tc2, m)
      * [matches TripleConstraint](#matches-TripleConstraint) ([t2], tc2, m)
  + outs = Ø
  + matchables = Ø, unmatchables = Ø, closed is false

## 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](#prod-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](#prod-shapeExprDecl) are startActs. After the first [shape expression declaration](#prod-shapeExprDecl), semantic actions are associated with the previous declaration.

As with Turtle and SPARQL, ShExC offers URL resolution relative to a base per [[RFC3986]] and prefixes map to provide shorthand ways to write IRI identifiers.

Display grammar only

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

[1]

shexDoc

   ::=

[directive](#prod-directive)\* (([notStartAction](#prod-notStartAction) | [startActions](#prod-startActions)) [statement](#prod-statement)\*)?

followed by the associated ShExJ object(s):

Schema

{

startActs:[SemAct+]? start:shapeExpr? imports:[IRIREF+]? shapes:[ShapeDecl+]? }

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

* startActs comes from [startActions](#prod-startActions) production.
* start comes from the [start](#prod-start) production.
* shapes come from the [shapeExprDecl](#prod-shapeExprDecl) production.

[2]

directive

   ::=

[baseDecl](#prod-baseDecl) | [prefixDecl](#prod-prefixDecl) | [importDecl](#prod-importDecl)

[3]

baseDecl

   ::=

"BASE" [IRIREF](#term-IRIREF)

[4]

prefixDecl

   ::=

"PREFIX" [PNAME\_NS](#term-PNAME_NS) [IRIREF](#term-IRIREF)

[4½]

importDecl

   ::=

"IMPORT" [IRIREF](#term-IRIREF)

"IMPORT" is described in [ShEx Import](#import).

[5]

notStartAction

   ::=

[start](#prod-start) | [shapeExprDecl](#prod-shapeExprDecl)

[6]

start

   ::=

"start" '=' [inlineShapeExpression](#prod-inlineShapeExpression)

[7]

startActions

   ::=

[codeDecl](#prod-codeDecl)+

[8]

statement

   ::=

[directive](#prod-directive) | [notStartAction](#prod-notStartAction)

[9]

shapeExprDecl

   ::=

[shapeExprLabel](#prod-shapeExprLabel) ([shapeExpression](#prod-shapeExpression) | "EXTERNAL")

If the "EXTERNAL" keyword is present, [shapeExprDecl](#prod-shapeExprDecl) returns a ShapeDecl object whose shapeExpr is a ShapeExternal:

ShapeDecl

{

id:shapeExprLabel abstract:BOOL? shapeExpr: { type: "ShapeExternal" } }

* abstract is true if "ABSTRACT" was matched.

otherwise [shapeExprDecl](#prod-shapeExprDecl) returns a ShapeDecl object whose shapeExpr is a [shapeExpression](#prod-shapeExpression).

ShapeDecl

{

id:shapeExprLabel abstract:BOOL? shapeExpr: shapeExpr }

* abstract is true if "ABSTRACT" was matched.

Shape expressions are logical combinations of shape atoms. Inline variants of shape expressions are used in [tripleConstraint](#prod-tripleConstraint)s and are not permitted to have annotations or semantic actions.

[10]

shapeExpression

   ::=

[shapeOr](#prod-shapeOr)

[11]

inlineShapeExpression

   ::=

[inlineShapeOr](#prod-inlineShapeOr)

[12]

shapeOr

   ::=

[shapeAnd](#prod-shapeAnd) ("OR" [shapeAnd](#prod-shapeAnd))\*

[13]

inlineShapeOr

   ::=

[inlineShapeAnd](#prod-inlineShapeAnd) ("OR" [inlineShapeAnd](#prod-inlineShapeAnd))\*

If the right [shapeAnd](#prod-shapeAnd) matches one or more times, the result is a ShapeOr object with shapeExprs containing the first [shapeAnd](#prod-shapeAnd) followed by the ordered list from the second [shapeAnd](#prod-shapeAnd):

ShapeOr

{

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

otherwise the result is the left [shapeAnd](#prod-shapeAnd).

[14]

shapeAnd

   ::=

[shapeNot](#prod-shapeNot) ("AND" [shapeNot](#prod-shapeNot))\*

[15]

inlineShapeAnd

   ::=

[inlineShapeNot](#prod-inlineShapeNot) ("AND" [inlineShapeNot](#prod-inlineShapeNot))\*

If the right [shapeNot](#prod-shapeNot) matches one or more times, the result is a ShapeAnd object with shapeExprs containing the first [shapeNot](#prod-shapeNot) followed by the ordered list from the second [shapeNot](#prod-shapeNot):

ShapeAnd

{

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

otherwise the result is the left [shapeNot](#prod-shapeNot).

[16]

shapeNot

   ::=

"NOT"? [shapeAtom](#prod-shapeAtom)

[17]

inlineShapeNot

   ::=

"NOT"? [inlineShapeAtom](#prod-inlineShapeAtom)

If the left "NOT" matches, the result is a ShapeNot object with shapeExpr containing the [shapeAtom](#prod-shapeAtom):

ShapeNot

{

id:shapeExprLabel? shapeExpr:shapeExpr }

otherwise the result is the [shapeAtom](#prod-shapeAtom).

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

[18]

shapeAtom

   ::=

[nonLitNodeConstraint](#prod-nonLitNodeConstraint) [shapeOrRef](#prod-shapeOrRef)?  
| [litNodeConstraint](#prod-litNodeConstraint)  
| [shapeOrRef](#prod-shapeOrRef) [nonLitNodeConstraint](#prod-nonLitNodeConstraint)?  
| '(' [shapeExpression](#prod-shapeExpression) ')'  
| '.'

[19]

shapeAtomNoRef

   ::=

[nonLitNodeConstraint](#prod-nonLitNodeConstraint) [shapeOrRef](#prod-shapeOrRef)?  
| [litNodeConstraint](#prod-litNodeConstraint)  
| [shapeDefinition](#prod-shapeDefinition) [nonLitNodeConstraint](#prod-nonLitNodeConstraint)?  
| '(' [shapeExpression](#prod-shapeExpression) ')'  
| '.'

[20]

inlineShapeAtom

   ::=

[nonLitNodeConstraint](#prod-nonLitNodeConstraint) [inlineShapeOrRef](#prod-inlineShapeOrRef)?  
| [litNodeConstraint](#prod-litNodeConstraint)  
| [inlineShapeOrRef](#prod-inlineShapeOrRef) [nonLitNodeConstraint](#prod-nonLitNodeConstraint)?  
| '(' [shapeExpression](#prod-shapeExpression) ')'  
| '.'

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

[21]

shapeOrRef

   ::=

[shapeDefinition](#prod-shapeDefinition) | [shapeRef](#prod-shapeRef)

[22]

inlineShapeOrRef

   ::=

[inlineShapeDefinition](#prod-inlineShapeDefinition) | [shapeRef](#prod-shapeRef)

[23]

shapeRef

   ::=

   ( [ATPNAME\_LN](#term-ATPNAME_LN) | [ATPNAME\_NS](#term-ATPNAME_NS) | '@' [shapeExprLabel](#prod-shapeExprLabel) ) "EXACTLY"

* If the [shapeDefinition](#prod-shapeDefinition) production matches, the result is [shapeDefinition](#prod-shapeDefinition).
* Otherwise, the result is a ShapeExprRef to shapeExprLabel.

ShapeExprRef

{

label: shapeExprLabel }

* exact is true if the "EXACTLY" choice was matched.

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

[24]

litNodeConstraint

   ::=

   "LITERAL" [xsFacet](#prod-xsFacet)\*  
| [datatype](#prod-datatype) [xsFacet](#prod-xsFacet)\*  
| [valueSet](#prod-valueSet) [xsFacet](#prod-xsFacet)\*  
| [numericFacet](#prod-numericFacet)+

[25]

nonLitNodeConstraint

   ::=

[nonLiteralKind](#prod-nonLiteralKind) [stringFacet](#prod-stringFacet)\*  
| [stringFacet](#prod-stringFacet)+

NodeConstraint

{

id:shapeExprLabel? nodeKind:("iri" | "bnode" | "nonliteral" | "literal")? datatype:IRIREF? xsFacet\* values:[valueSetValue+]? }

[26]

nonLiteralKind

   ::=

"IRI" | "BNODE" | "NONLITERAL"

[27]

xsFacet

   ::=

[stringFacet](#prod-stringFacet) | [numericFacet](#prod-numericFacet)

xsFacet

=

stringFacet | numericFacet ;

[28]

stringFacet

   ::=

[stringLength](#prod-stringLength) [INTEGER](#term-INTEGER)  
| [REGEXP](#term-REGEXP)

[29]

stringLength

   ::=

"LENGTH" | "MINLENGTH" | "MAXLENGTH"

stringFacet

=

(length|minlength|maxlength):INTEGER | pattern:STRING flags:STRING? ;

[30]

numericFacet

   ::=

[numericRange](#prod-numericRange) [numericLiteral](#prod-numericLiteral)  
| [numericLength](#prod-numericLength) [INTEGER](#term-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

   ::=

([extraPropertySet](#prod-extraPropertySet) | "CLOSED")\* '{' [tripleExpression](#prod-tripleExpression)? '}' [annotation](#prod-annotation)\* [semanticActions](#prod-semanticActions)

[34]

inlineShapeDefinition

   ::=

([extraPropertySet](#prod-extraPropertySet) | "CLOSED")\* '{' [tripleExpression](#prod-tripleExpression)? '}'

Shape

{

extends:[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](#prod-extraPropertySet) production.
* expression comes from the [tripleExpression](#prod-tripleExpression) production.
* annotations is the set of Annotations matching the [annotation](#prod-annotation) production.
* semActs is the set of semantic actions matching the [semanticActions](#prod-semanticActions) production.

[35]

extraPropertySet

   ::=

"EXTRA" [predicate](#prod-predicate)+

Triple expressions are arrangements of triple constraints.

[36]

tripleExpression

   ::=

[oneOfTripleExpr](#prod-oneOfTripleExpr)

[37]

oneOfTripleExpr

   ::=

[groupTripleExpr](#prod-groupTripleExpr) | [multiElementOneOf](#prod-multiElementOneOf)

[38]

multiElementOneOf

   ::=

[groupTripleExpr](#prod-groupTripleExpr) ('|' [groupTripleExpr](#prod-groupTripleExpr))+

If the right [groupTripleExpr](#prod-groupTripleExpr) matches one or more times, the result is a OneOf object with expressions containing the first [groupTripleExpr](#prod-groupTripleExpr) followed by the ordered list from the second [groupTripleExpr](#prod-groupTripleExpr):

OneOf

{

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

otherwise the result is the left [groupTripleExpr](#prod-groupTripleExpr).

[40]

groupTripleExpr

   ::=

[singleElementGroup](#prod-singleElementGroup) | [multiElementGroup](#prod-multiElementGroup)

[41]

singleElementGroup

   ::=

[unaryTripleExpr](#prod-unaryTripleExpr) ';'?

[42]

multiElementGroup

   ::=

[unaryTripleExpr](#prod-unaryTripleExpr) (';' [unaryTripleExpr](#prod-unaryTripleExpr))+ ';'?

If the right [unaryTripleExpr](#prod-unaryTripleExpr) matches one or more times, the result is a EachOf object with expressions containing the first [unaryTripleExpr](#prod-unaryTripleExpr) followed by the ordered list from the second [unaryTripleExpr](#prod-unaryTripleExpr):

EachOf

{

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

otherwise the result is the left [unaryTripleExpr](#prod-unaryTripleExpr).

[43]

unaryTripleExpr

   ::=

   ('$' [tripleExprLabel](#prod-tripleExprLabel))? ([tripleConstraint](#prod-tripleConstraint) | [bracketedTripleExpr](#prod-bracketedTripleExpr))  
| [include](#prod-include)

[44]

bracketedTripleExpr

   ::=

'(' [tripleExpression](#prod-tripleExpression) ')' [cardinality](#prod-cardinality)? [annotation](#prod-annotation)\* [semanticActions](#prod-semanticActions)

Triple constraints are matched against RDF triples.

[45]

tripleConstraint

   ::=

[senseFlags](#prod-senseFlags)? [predicate](#prod-predicate) [inlineShapeExpression](#prod-inlineShapeExpression) [cardinality](#prod-cardinality)? [annotation](#prod-annotation)\* [semanticActions](#prod-semanticActions)

TripleConstraint

{

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

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

[46]

cardinality

   ::=

'\*' | '+' | '?' | [REPEAT\_RANGE](#term-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](#prod-valueSetValue)\* ']'

[49]

valueSetValue

   ::=

[iriRange](#prod-iriRange) | [literalRange](#prod-literalRange) | [languageRange](#prod-languageRange)  
| [exclusion](#prod-exclusion)+

If "." matches and [exclusion](#prod-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](#prod-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](#prod-exclusion), valueSetValue returns a Wildcard object:

Wildcard

{

/\* empty \*/ }

[50]

exclusion

   ::=

'.' ([iriExclusion](#prod-iriExclusion) | [literalExclusion](#prod-literalExclusion) | [languageExclusion](#prod-languageExclusion)) +

[50.1]

iriExclusion

   ::=

'-' [iri](#prod-iri) '~'?

[50.2]

literalExclusion

   ::=

'-' [literal](#prod-literal) '~'?

[50.3]

languageExclusion

   ::=

'-' [LANGTAG](#term-LANGTAG) '~'?

[51]

iriRange

   ::=

[iri](#prod-iri) ('~' [iriExclusion](#prod-iriExclusion)\*)?

If [iri](#prod-iri) matches with no "~", iriRange returns [iri](#prod-iri).

If [iri](#prod-iri) and "~" match with no [iriExclusion](#prod-iriExclusion), iriRange returns a IriStem object:

IriStem

{

stem:IRIREF }

If [iri](#prod-iri) and "~" match and [iriExclusion](#prod-iriExclusion) matches one or more times, iriRange returns a IriStemRange object with exclusions equal to the set of results of [iriExclusion](#prod-iriExclusion):

IriStemRange

{

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

[52]

iriExclusion

   ::=

'-' [iri](#prod-iri) '~'?

[53]

literalRange

   ::=

[literal](#prod-literal) ('~' [literalExclusion](#prod-literalExclusion)\*)?

If [literal](#prod-literal) matches with no "~", literalRange returns [literal](#prod-literal).

If [literal](#prod-literal) and "~" match with no [literalExclusion](#prod-literalExclusion), literalRange returns a LiteralStem object:

LiteralStem

{

stem:STRING }

If [literal](#prod-literal) and "~" match and [literalExclusion](#prod-literalExclusion) matches one or more times, literalRange returns a LiteralStemRange object with exclusions equal to the set of results of [literalExclusion](#prod-literalExclusion):

LiteralStemRange

{

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

[54]

literalExclusion

   ::=

'-' [literal](#prod-literal) '~'?

[55]

languageRange

   ::=

[LANGTAG](#term-LANGTAG) ('~' [languageExclusion](#prod-languageExclusion)\*)?  
| '@' '~' [languageExclusion](#prod-languageExclusion)\*

If [LANGTAG](#term-LANGTAG) matches with no "~" match , languageRange returns a Language object with languageTag equal to [LANGTAG](#term-LANGTAG):

Language

{

languageTag:LANGTAG }

If [LANGTAG](#term-LANGTAG) and "~" match with no [languageExclusion](#prod-languageExclusion), languageRange returns a LanguageStem object:

LanguageStem

{

stem:LANGTAG }

If [LANGTAG](#term-LANGTAG) and "~" match and [languageExclusion](#prod-languageExclusion) matches one or more times, languageRange returns a LanguageStemRange object with exclusions equal to the set of results of [languageExclusion](#prod-languageExclusion):

LanguageStemRange

{

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

If '@' '~' matched with no [languageExclusion](#prod-languageExclusion), languageRange returns a LanguageStemRange object with an empty stem:

LanguageStemRange

{

stem: "" }

If '@' '~' matched and [languageExclusion](#prod-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](#prod-languageExclusion):

LanguageStemRange

{

stem: "" exclusions:[LANGTAG|LanguageStem +] }

[56]

languageExclusion

   ::=

'-' [LANGTAG](#term-LANGTAG) '~'?

Triple expressions can include the [shapeExpression](#prod-shapeExpression) in a [shapeExprDecl](#prod-shapeExprDecl).

[57]

include

   ::=

'&' [tripleExprLabel](#prod-tripleExprLabel)

Per the [triple expression refrence requirement](#tripleExprRef-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](#prod-predicate) and an [iri](#prod-iri) or [literal](#prod-literal).

[58]

annotation

   ::=

"//" [predicate](#prod-predicate) ([iri](#prod-iri) | [literal](#prod-literal))

Annotation

{

predicate:IRIREF object:objectValue }

Triple expressions can include semantic actions consisting of an [iri](#prod-iri) and an optional code string.

[59]

semanticActions

   ::=

[codeDecl](#prod-codeDecl)\*

[60]

codeDecl

   ::=

'%' [iri](#prod-iri) ([CODE](#term-CODE) | '%')

SemAct

{

name:IRIREF code:STRING? }

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

[13t]

literal

   ::=

[rdfLiteral](#prod-rdfLiteral) | [numericLiteral](#prod-numericLiteral) | [booleanLiteral](#prod-booleanLiteral)

[61]

predicate

   ::=

[iri](#prod-iri) | [RDF\_TYPE](#term-RDF_TYPE)

[62]

datatype

   ::=

[iri](#prod-iri)

[63]

shapeExprLabel

   ::=

[iri](#prod-iri) | [blankNode](#prod-blankNode)

[64]

tripleExprLabel

   ::=

[iri](#prod-iri) | [blankNode](#prod-blankNode)

[16t]

numericLiteral

   ::=

[INTEGER](#term-INTEGER) | [DECIMAL](#term-DECIMAL) | [DOUBLE](#term-DOUBLE)

[65]

rdfLiteral

   ::=

[langString](#prod-langString) | [string](#prod-string) ("^^" [datatype](#prod-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](#term-STRING_LITERAL1) | [STRING\_LITERAL\_LONG1](#term-STRING_LITERAL_LONG1)  
| [STRING\_LITERAL2](#term-STRING_LITERAL2) | [STRING\_LITERAL\_LONG2](#term-STRING_LITERAL_LONG2)

[66]

langString

   ::=

[LANG\_STRING\_LITERAL1](#term-LANG_STRING_LITERAL1) | [LANG\_STRING\_LITERAL\_LONG1](#term-LANG_STRING_LITERAL_LONG1)  
| [LANG\_STRING\_LITERAL2](#term-LANG_STRING_LITERAL2) | [LANG\_STRING\_LITERAL\_LONG2](#term-LANG_STRING_LITERAL_LONG2)

[136s]

iri

   ::=

[IRIREF](#term-IRIREF) | [prefixedName](#prod-prefixedName)

[137s]

prefixedName

   ::=

[PNAME\_LN](#term-PNAME_LN) | [PNAME\_NS](#term-PNAME_NS)

[138s]

blankNode

   ::=

[BLANK\_NODE\_LABEL](#term-BLANK_NODE_LABEL)

### Terminals

Terminals return:

* the RDF abstract types IRI, lexical form, literal, language tag.
* a string of unicode codepoints for [CODE](#term-CODE).
* a repeat range for [REPEAT\_RANGE](#term-REPEAT_RANGE). A repeat range is a tuple of non-negative integers or a non-negative integer and a token for \*.

[67]

<CODE>

   ::=

"{" ([^%\\] | "\\" [%\\] | [UCHAR](#term-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](#term-INTEGER) ( "," ([INTEGER](#term-INTEGER) | "\*")? )? "}"

returns: repeat range

The base-10 numeric values of [INTEGER](#term-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](#term-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](#term-PN_PREFIX)? ":"

returns: PREFIX

When used in a [prefixDecl](#prod-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](#term-PNAME_NS) [PN\_LOCAL](#term-PN_LOCAL)

returns: IRI

A potentially empty prefix is identified by the first token, PNAME\_NS. The unicode string of the IRI is formed by unescaping the [reserved characters](https://tools.ietf.org/html/rfc3987#page-9) [[!rfc7159]] in the second argument, PN\_LOCAL, and concatenating this onto the namespace found in the prefixes map's entry for PNAME\_NS.

[70]

<ATPNAME\_NS>

   ::=

"@" [PNAME\_NS](#term-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](#term-PNAME_LN)

returns: IRI

A potentially empty prefix is identified by the second token, PNAME\_NS. The unicode string of the IRI is formed by unescaping the [reserved characters](https://tools.ietf.org/html/rfc3987#page-9) [[!rfc7159]] in the third token, PN\_LOCAL, and concatenating this onto the namespace found in the prefixes map's entry for PNAME\_NS.

[72]

<REGEXP>

   ::=

'/' ([^/\\\n\r]  
     | '\\' [nrt\\|.?\*+(){}$-\[\]^/]  
     | [UCHAR](#term-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](#term-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](#term-PN_CHARS_U) | [0-9]) (([PN\_CHARS](#term-PN_CHARS) | ".")\* [PN\_CHARS](#term-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](#term-EXPONENT) | "."? [0-9]+ [EXPONENT](#term-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](#term-ECHAR) | [UCHAR](#term-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](#term-ECHAR) | [UCHAR](#term-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](#term-ECHAR) | [UCHAR](#term-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](#term-ECHAR) | [UCHAR](#term-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](#term-ECHAR) | [UCHAR](#term-UCHAR))\* "'" [LANGTAG](#term-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](#term-LANGTAG) is used to create a language-tagged string.

[74]

<LANG\_STRING\_LITERAL2>

   ::=

'"' ([^\"\\\n\r] | [ECHAR](#term-ECHAR) | [UCHAR](#term-UCHAR))\* '"' [LANGTAG](#term-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](#term-LANGTAG) is used to create a language-tagged string.

[75]

<LANG\_STRING\_LITERAL\_LONG1>

   ::=

"'''" ( ("'" | "''")? ([^\\'\\] | [ECHAR](#term-ECHAR) | [UCHAR](#term-UCHAR)) )\* "'''" [LANGTAG](#term-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](#term-LANGTAG) is used to create a language-tagged string.

[76]

<LANG\_STRING\_LITERAL\_LONG2>

   ::=

'"""' ( ('"' | '""')? ([^\"\\] | [ECHAR](#term-ECHAR) | [UCHAR](#term-UCHAR)) )\* '"""' [LANGTAG](#term-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](#term-LANGTAG) is used to create a language-tagged string.

[26t]

<UCHAR>

   ::=

   "\\u" [HEX](#term-HEX) [HEX](#term-HEX) [HEX](#term-HEX) [HEX](#term-HEX)  
| "\\U" [HEX](#term-HEX) [HEX](#term-HEX) [HEX](#term-HEX) [HEX](#term-HEX) [HEX](#term-HEX) [HEX](#term-HEX) [HEX](#term-HEX) [HEX](#term-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](#term-PN_CHARS_BASE) | "\_"

[167s]

<PN\_CHARS>

   ::=

[PN\_CHARS\_U](#term-PN_CHARS_U) | "-" | [0-9]  
| [#00B7] | [#0300-#036F] | [#203F-#2040]

[168s]

<PN\_PREFIX>

   ::=

[PN\_CHARS\_BASE](#term-PN_CHARS_BASE) ( ([PN\_CHARS](#term-PN_CHARS) | ".")\* [PN\_CHARS](#term-PN_CHARS) )?

[77]

<PN\_LOCAL>

   ::=

([PN\_CHARS\_U](#term-PN_CHARS_U) | ":" | [0-9] | [PLX](#term-PLX)) (([PN\_CHARS](#term-PN_CHARS) | "." | ":" | [PLX](#term-PLX))\* ([PN\_CHARS](#term-PN_CHARS) | ":" | [PLX](#term-PLX)))?

[170s]

<PLX>

   ::=

[PERCENT](#term-PERCENT) | [PN\_LOCAL\_ESC](#term-PN_LOCAL_ESC)

[171s]

<PERCENT>

   ::=

"%" [HEX](#term-HEX) [HEX](#term-HEX)

[172s]

<HEX>

   ::=

[0-9] | [A-F] | [a-f]

[173s]

<PN\_LOCAL\_ESC>

   ::=

"\\" ( "\_" | "~" | "." | "-" | "!" | "$" | "&" | "'" | "(" | ")" | "\*" | "+" | "," | ";" | "=" | "/" | "?" | "#" | "@" | "%" )

[98]

PASSED TOKENS

   ::=

   [ \t\r\n]+  
| "#" [^\r\n]\*  
| "/\*" ([^\*] | '\*' ([^/] | '\\/'))\* "\*/"

## 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 . Processors MAY interpret a ShExJ document as an RDF Graph. Processors may also transform arbitrary RDF Graphs conforming to 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"? imports:[IRIREF+]? startActs:[SemAct+]? start:shapeExpr? shapes:[ShapeDecl+]? } |
| ShapeDecl | { | id:shapeExprLabel abstract:BOOL? shapeExpr:shapeExpr | ShapeExternal } |
| shapeExpr | = | ShapeOr | ShapeAnd | ShapeNot | NodeConstraint | Shape | ShapeExprRef ; |
| ShapeOr | { | shapeExprs:[shapeExpr{2,}] } |
| ShapeAnd | { | shapeExprs:[shapeExpr{2,}] } |
| ShapeNot | { | shapeExpr:shapeExpr } |
| ShapeExternal | { | } |
| ShapeExprRef | { | label:  shapeExprLabel exact:BOOL? } |
| 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 | EMPTY) } |
| LanguageStemRange | { | stem:(LANGTAG | EMPTY) exclusions:[LANGTAG|LanguageStem+]? } |
| Wildcard | { | /\* empty \*/ } |
| Shape | { | extends:[shapeExprRef]? 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](https://www.w3.org/TR/rdf11-concepts/#dfn-predicate):IRIREF valueExpr:shapeExpr? min:INTEGER? max:INTEGER? semActs:[SemAct+]? annotations:[Annotation+]? } |
| tripleExprRef | = | tripleExprLabel ; |
| tripleExprLabel | = | IRIREF | BNODE ; |
| SemAct | { | name:IRIREF code:STRING? } |
| Annotation | { | [predicate](https://www.w3.org/TR/rdf11-concepts/#dfn-predicate):[IRIREF](https://www.w3.org/TR/rdf11-concepts/#dfn-iri) object:objectValue } |
| # Terminals |  | These follow the [rules for terminals in the XML 1.0 5th Edition](http://www.w3.org/TR/2008/PER-xml-20080205/#sec-common-syn) |
|  | # | [Turtle IRIREF](http://www.w3.org/TR/turtle/#grammar-production-IRIREF) without enclosing "<>"s |
| IRIREF | : | (PN\_CHARS | '.' | ':' | '/' | '\\' | '#' | '@' | '%' | '&' | UCHAR)\* ; |
|  | # | [Turtle BLANK\_NODE\_LABEL](http://www.w3.org/TR/turtle/#grammar-production-BLANK_NODE_LABEL) |
| BNODE | : | '\_:' (PN\_CHARS\_U | [0-9]) ((PN\_CHARS | '.')\* PN\_CHARS)? ; |
|  | # | [JSON boolean values](https://tools.ietf.org/html/rfc7159#section-3) |
| BOOL | : | "true" | "false" ; |
|  | # | [Turtle INTEGER](http://www.w3.org/TR/turtle/#grammar-production-INTEGER) |
| INTEGER | : | [+-]? [0-9] + ; |
|  | # | [Turtle DECIMAL](http://www.w3.org/TR/turtle/#grammar-production-DECIMAL) |
| DECIMAL | : | [+-]? [0-9]\* '.' [0-9] + ; |
|  | # | [Turtle DOUBLE](http://www.w3.org/TR/turtle/#grammar-production-DOUBLE) |
| DOUBLE | : | [+-]? ([0-9] + '.' [0-9]\* EXPONENT | '.' [0-9]+ EXPONENT | [0-9]+ EXPONENT) ; |
|  | # | [BCP47 Language-Tag](https://tools.ietf.org/html/bcp47#section-2.1) |
| LANGTAG | : | [a-zA-Z]+ ('-' [a-zA-Z0-9]+)\* ; |
|  | # | any [JSON string](https://tools.ietf.org/html/rfc7159#section-7) |
| STRING | : | .\* ; |
|  | # | empty string |
| EMPTY | : | ^$ ; |
| # 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]+ ; |

## RDF Representation of ShEx (ShExR)

A ShExR graph is any RDF Graph which conforms to the following shapes schema and meets the [Schema Requirements](#schema-requirements). Every ShExR document is [graph isomorphic](https://www.w3.org/TR/rdf11-concepts/#dfn-graph-isomorphism)[[!rdf11-concepts]] to the [RDF serialization](https://www.w3.org/TR/json-ld11-api/#deserialize-json-ld-to-rdf-algorithm)[[!json-ld]] of some ShExJ document.

## 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)](#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](#dfn-shapeexpr) or [tripleExpr](#dfn-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

### 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, .