P3330™/D4
Draft Standard for Shape Expression Schemas

Developed by the

Standards Activities Board Standards Committee
of the
IEEE Computer Society

Approved <Date Approved>

IEEE SA Standards Board

Abstract: This standard defines the syntax of Shape Expression schemas represented in JavaScript Object Notation (JSON), Resource Description Framework (RDF) and plain text. This standard includes formal semantics for validation of RDF knowledge graphs using Shape Expressions. This validation process includes the definition of ShapeMaps to associate nodes in RDF graphs with labeled Shape Expressions. A test suite covers all aspects of syntax and validation.

Keywords: IEEE 3330™, RDF, Schema, Shape Expressions, Structure Definition, Structural Validation

---

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

PDF: ISBN 978-0-XXXX-XXXX-X STDXXXXX
Print: ISBN 978-0-XXXX-XXXX-X STDPDXXXXX

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE Standards documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page (https://standards.ieee.org/ipr/disclaimers.html), appear in all IEEE standards and may be found under the heading “Important Notices and Disclaimers Concerning IEEE Standards Documents.”

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents are developed within IEEE Societies and subcommittees of IEEE Standards Association (IEEE SA) Board of Governors. IEEE develops its standards through an accredited consensus development process, which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE standards are documents developed by volunteers with scientific, academic, and industry-based expertise in technical working groups. Volunteers involved in technical working groups are not necessarily members of IEEE or IEEE SA and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE makes no warranties or representations concerning its standards, and expressly disclaims all warranties, express or implied, concerning all standards, including but not limited to the warranties of merchantability, fitness for a particular purpose and non-infringement IEEE Standards documents do not guarantee safety, security, health, or environmental protection, or compliance with law, or guarantee against interference with or from other devices or networks. In addition, IEEE does not warrant or represent that the use of the material contained in its standards is free from patent infringement. IEEE Standards documents are supplied “AS IS” and “WITH ALL FAULTS.”

Use of an IEEE standard is wholly voluntary. The existence of an IEEE standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity, nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document should rely upon their own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: THE NEED TO PROCURE SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Translations

The IEEE consensus balloting process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English language version published by IEEE is the approved IEEE standard.

Use by artificial intelligence systems

In no event shall material in any IEEE Standards documents be used for the purpose of creating, training, enhancing, developing, maintaining, or contributing to any artificial intelligence systems without the express, written consent of IEEE SA in advance. "Artificial intelligence" refers to any software, application, or other system that uses artificial intelligence, machine learning, or similar technologies, to analyze, train, process, or generate content. Requests for consent can be submitted using the Contact Us form.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE SA Standards Board Operations Manual is not, and shall not be considered or inferred to be, the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE or IEEE SA. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that the presenter's views should be considered the personal views of that individual rather than the formal position of IEEE, IEEE SA, the Standards Committee, or the Working Group. Statements made by volunteers may not represent the formal position of their employer(s) or affiliation(s). News releases about IEEE standards issued by entities other than IEEE SA should be considered the view of the entity issuing the release rather than the formal position of IEEE or IEEE SA.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE or IEEE SA. However, **IEEE does not provide interpretations, consulting information, or advice pertaining to IEEE Standards documents**.

Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its Societies and subcommittees of the IEEE SA Board of Governors are not able to provide an instant response to comments or questions, except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in evaluating comments or revisions to an IEEE standard is welcome to join the relevant IEEE SA working group. You can indicate interest in a working group using the Interests tab in the Manage Profile & Interests area of the IEEE SA myProject system.[1] An IEEE Account is needed to access the application.

Comments on standards should be submitted using the Contact Us form.[2]

[1] Available at: https://development.standards.ieee.org/myproject-web/public/view.html#landing.
[2] Available at: https://standards.ieee.org/about/contact/.

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not constitute compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Data privacy

Users of IEEE Standards documents should evaluate the standards for considerations of data privacy and data ownership in the context of assessing and using the standards in compliance with applicable laws and regulations.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, neither IEEE nor its licensors waive any rights in copyright to the documents.

Photocopies

Subject to payment of the appropriate licensing fees, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400; https://www.copyright.com/. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every 10 years. When a document is more than 10 years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit IEEE Xplore or contact IEEE.[3] For more information about the IEEE SA or IEEE's standards development process, visit the IEEE SA Website.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE SA Website.[4] Search for standard number and year of approval to access the web page of the published standard. Errata links are located under the Additional Resources Details section. Errata are also available in IEEE Xplore. Users are encouraged to periodically check for errata.

Patents

IEEE standards are developed in compliance with the IEEE SA Patent Policy.[5]

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE SA Website at https://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

IMPORTANT NOTICE

Technologies, application of technologies, and recommended procedures in various industries evolve over time. The IEEE standards development process allows participants to review developments in industries, technologies, and practices, and to determine what, if any, updates should be made to the IEEE standard. During this evolution, the technologies and recommendations in IEEE standards may be implemented in ways not foreseen during the standard's development. IEEE standards development activities consider research and information presented to the standards development group in developing any safety recommendations. Other information about safety practices, changes in technology or technology implementation, or impact by peripheral systems also may be pertinent to safety considerations during implementation of the standard. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, data privacy, and interference protection practices and all applicable laws and regulations.

[3] Available at: https://ieeexplore.ieee.org/browse/standards/collection/ieee.
[4] Available at: https://standards.ieee.org/standard/index.html.
[5] Available at: https://standards.ieee.org/about/sasb/patcom/materials.html.

Participants

At the time this draft standard was completed, the Shape Expression Schemas Working Group had the following membership:

Kat Thornton, *Chair*
<Vice-chair Name>, *Vice Chair*

Participant1
Participant2
Participant3
Participant4
Participant5
Participant6
Participant7
Participant8
Participant9

The following members of the <individual/entity> Standards Association balloting group voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

[To be supplied by IEEE]

Balloter1
Balloter2
Balloter3
Balloter4
Balloter5
Balloter6
Balloter7
Balloter8
Balloter9

When the IEEE SA Standards Board approved this standard on <Date Approved>, it had the following membership:

[To be supplied by IEEE]

<Name>, *Chair*
<Name>, *Vice Chair*
<Name>, *Past Chair*
<Name>, *Secretary*

SBMember1
SBMember2
SBMember3
SBMember4
SBMember5
SBMember6
SBMember7
SBMember8
SBMember9

*Member Emeritus

Introduction

This introduction is not part of P3330/D4, Draft Standard for Shape Expression Schemas.

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).

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 included a definition of 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, language and test issues. Only send comments to public-shex@w3.org (subscribe, archives) if you are unable to raise issues on GitHub. All comments are welcome.

Contents

<After draft body is complete, select this text and click Insert Special->Add (Table of) Contents>

Draft Standard for Shape Expression Schemas

1. Overview

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

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

1.1 Scope

This standard defines the syntax of Shape Expression schemas represented in JavaScript Object Notation (JSON), Resource Description Framework (RDF) and plain text. This standard includes formal semantics for validation of RDF knowledge graphs using Shape Expressions. This validation process includes the definition of ShapeMaps to associate nodes in RDF graphs with labeled Shape Expressions. A test suite covers all aspects of syntax and validation.

1.2 Word Usage

The word shall indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (shall equals is required to).

The word should indicates that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required (should equals is recommended that).

The word may is used to indicate a course of action permissible within the limits of the standard (may equals is permitted to).

The word can is used for statements of possibility and capability, whether material, physical, or causal (can equals is able to).

2. Draft

This status of this document ED, it is NOT an IEEE specifiation.

2.1 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, IANA Considerations.

2.2 Normative references

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

[JSON-LD]
Manu Sporny; Gregg Kellogg; Markus Lanthaler. JSON-LD 1.0. W3C. 3 November 2020. W3C Recommendation. URL: https://www.w3.org/TR/json-ld/

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

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

[RFC3986]
T. Berners-Lee; R. Fielding; L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. IETF. January 2005. Internet Standard. URL: https://www.rfc-editor.org/rfc/rfc3986

[rfc4647]
A. Phillips, Ed.; M. Davis, Ed.. Matching of Language Tags. IETF. September 2006. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc4647

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

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

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

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

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

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

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

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

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

[xpath20]

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

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 included a definition of 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, language and test issues. Only send comments to public-shex@w3.org (subscribe, archives) if you are unable to raise issues on GitHub. All comments are welcome.

3. 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).

4. Definitions, Acronyms, and Abbreviations

4.1 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), the neighbourhood of the node n in a graph G, is the union of arcsOut(G, n) and arcsIn(G, n):
  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)`**.
- fixed ShapeMap is a list of pairs of RDF node label and shape expression label as defined in the [shape-map] specification.
- Given the graph G, schema Sch, and the fixed ShapeMap ism, validation(G, Sch, ism) is the process of assigning each node/shapeExpr pair in ism a status of **`conformant`** or **`nonconformant`** reflecting whether the node in G satisfies the shapeExpr in Sch.

Consider the RDF graph G represented in Turtle:

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

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

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

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

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

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

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

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

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

4.2 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

5. 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" }`

5.1 JSON Grammar

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

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

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

- **`typeName`** is the name of the typed data structure. Types are referenced in the definitions of object members and in the definitions of the semantics for those data structures.
- **`member*`** is a list of zero or more terminals or references to other typeExpressions.
- A **`typeExpression`** is one of:
  - **`typeName`** — an object of corresponding type

  - **array**: [typeExpression+]— an array of one or more JSON values matching the typeExpression.
  - **choice**: typeExpression1 | typeExpression2 | …— a choice between two or more 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.

Note

The executable JSON grammar for ShExJ 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.

5.2 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"
            } } } } } ] }
```

5.3 Document style

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

5.4 Graph access

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

5.5 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#`

6. The Shape Expressions Language

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

6.1 Shapes Schema

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

```
   Schema { "@context":"http://www.w3.org/ns/shex.jsonld"? import:[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", … } ] }
```

6.2 Validation Definition

For a graph G, a schema Sch and a fixed input 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.

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.

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)`.

Note

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.

Note

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

Note

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

6.3 Shape Expressions

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

- A node constraint (NodeConstraint) 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.

6.3.1 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", … } }
    ] } ] }
```

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

6.3.2 Semantics

For a node n of the graph G, neigh(G, n) is the set of triples in **G** 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(n, se)` as described below in Node Constraints. Note that testing if a node satisfies a node constraint does not require a graph or typing.
- se is a Shape and `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 SE_1, SE_2, SE_3 in a Schema Sch, such that:

- **satisfies(n, SE_1, G, Sch, m)**
- **satisfies(n, SE_2, G, Sch, m)**
- NOT **satisfies(n, SE_3, G, Sch, m)**

the following hold:

- satisfies(

  n,

  ```
  { "type": "ShapeAnd", "shapeExprs": [ SE1, SE2 ] }
  ```

  ,

  G, Sch, m)
- satisfies(

  n,

  ```
  { "type": "ShapeOr", "shapeExprs": [ SE1, SE2, SE3 ] }
  ```

  ,

  G, Sch, m)
- 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 SE_1 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"
    }
  } ] }
```

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"
    } }
] }
```

6.4 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 */ }
```

6.4.1 Semantics

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

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

6.4.2 Node Kind Constraints

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

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

Node Kind example 1

The following examples use a TripleConstraint object described later in the document. The

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

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

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

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

6.4.3 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"
      } } } ] }
```

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

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

Note

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

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

Datatype example 2

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

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

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

6.4.4 XML Schema String Facet Constraints

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

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

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

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

String Facets example 1

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

```
      "predicate": "http://schema.example/#submittedBy",
      "valueExpr": { "type": "NodeConstraint", "minlength": 10 } } } ] }
```

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

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

Note

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

String Facets example 2

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

```
<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\\?$" }
} } ] }
```

```
<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).
```

6.4.5 XML Schema Numeric Facet Constraints

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

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

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

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

Numeric Facets example 1

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

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

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

6.4.6 Values Constraint

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

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

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

- vsv is a Wildcard with exclusions excls and there is no x in excls such that nodeIn(n, excl).

nodeIn: asserts that an RDF node n is equal to an RDF term s or is in a set defined by a IriStem, LiteralStem or LanguageStem.

The expression **nodeIn(n, s)** is satisfied if:

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

```
<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" }
            ] }
        ] } } } ] }
```

```
<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-" }
            ] }
        ] } } } ] }
```

```
<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.

6.5 Shapes and Triple Expressions

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

6.5.1 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 ;
```

6.5.2 Semantics

The semantics of the **`matchesShape`** function are based on the **`matches`** function defined below. 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 $\mathbf{R_0}$, $\mathbf{R_1}$ ... $\mathbf{R_{nCard}}$ such that
    - **`matches(`**$\mathbf{R_0}$**`, S.tripleExpr, m)`**
    - satisfies(n, constraint(L), G, Sch, m, matched)
    - for every i in 1..nCard, **`matches(`**$\mathbf{R_i}$**`, mainShape(`**$\mathbf{parents_i}$**`), m)`**
    - for every i in 1..nCard, satisfies(n, constraint($parent_i$), G, Sch, m, $R_i \cup Q$) where Q is the union of all the R_j s.t. $parent_j$ is a parent of $parent_i$.
- 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($parents_i$) or in S.expression.
- There is no triple in matchables which matches a TripleConstraint in one of the mainShape($parents_i$) 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.

  Note

  The complexity of partitioning is described briefly in the ShEx2 Primer.

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

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

  matches(

  T,

  ```
  { "type": "OneOf", "shapeExprs": [te1, te2, …], "min": 2, "max": 3,
    "semActs": [SemAct1, SemAct2, …] }
  ```

  ,

  m)

  evaluates as:

  matches(

  T,

  ```
  { "type": "OneOf", "shapeExprs": [te1, te2, …], "min": 2, "max": 3 }
  ```

  ,

  m)

  and semActsSatisfied([$SemAct_1$, $SemAct_2$, …])

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

  matches(

  T,

  ```
  { "type": "OneOf", "shapeExprs": [te1, te2, …], "min": 2, "max": 3 }
  ```

  ,

  m)

  evaluates as:

  Let e =

  ```
  { "type": "OneOf", "shapeExprs": [te1, te2, …] }
  ```

  (matches(T_1, e, m) and matches(T_2, e, m) and $T = T_1 \cup T_2$)
  or
  (matches(T_1, e, m) and matches(T_2, e, m) and matches(T_3, e, m) and $T = T_1 \cup T_2 \cup T_3$)

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

  matches(

  T,

  ```
  { "type": "OneOf", "shapeExprs": [
    { "type": "EachOf", "shapeExprs": [te3, te4, …] },
    { "type": "TripleExpression", "min": 1, "max": -1,
  ```

```
    "predicate": "http://xmlns.com/foaf/0.1/name" }
] }
```

,
m)

evaluates as:

matches(

T,

```
{ "type": "EachOf", "shapeExprs": [te3, te4, …] }
```

,
m)

or matches(

T,

```
{ "type": "TripleExpression", "min": 1, "max": -1,
    "predicate": "http://xmlns.com/foaf/0.1/name" }
```

,
m)

- expr is an EachOf and there is some partition of T into T_1, T_2,… such that for every expression $expr_1$, $expr_2$,… in shapeExprs, `matches(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" }
] }
```

,
m)

evaluates as:

matches(

T_1,

```
{ "type": "TripleExpression",
    "predicate": "http://xmlns.com/foaf/0.1/givenName" }
```

,
m)

and matches(

T_2,

```
{ "type": "TripleExpression",
    "predicate": "http://xmlns.com/foaf/0.1/familyName" }
```

,
m)
and $T = T_1 \cup T_2$

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

      matches(
      T,

      ```
      { "type": "TripleExpression",
        "predicate": "http://xmlns.com/foaf/0.1/givenName" }
      ```

      ,
      m)
      holds if
      - T has exactly one triple t.
      - t has the predicate "`http://xmlns.com/foaf/0.1/givenName`"
    - or 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" }
      ```

      ,
      m)
      holds if
      - T has exactly one triple t.
      - t has the predicate "`http://purl.org/dc/elements/1.1/author`"
      - t has a subject n2
      - The schema's shapes maps "`http://schema.example/#IssueShape`" to se2
      - satisfies(n2, se2, G, Sch, m)
- expr is a tripleExprRef and `satisfies(value, tripleExprWithId(tripleExprRef), G, Sch, Sch, m)`.
  The `tripleExprWithId` function is defined in Triple Expression Reference Requirement below.

For the schema

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

matches(

T,

`"http://schema.example/#PersonShape"`

,

m)

holds if

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

6.6 ShEx Import

The presence of imports requires that:

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

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

Import example 1 - Shape and Triple Expressions

```
schema1:
{ "type": "Schema", "imports": ["http://schema.example/schema2"], "shapes": [
  { "id": "http://schema.example/#EmployeeShape",
    "type": "Shape", "expression": {
      "type": "EachOf", "expressions": [
        "http://schema.example/#nameExpr",
        { "type": "TripleConstraint",
          "predicate": "http://schema.example/#empID",
          "valueExpr": { "type": "NodeConstraint",
            "datatype": "http://www.w3.org/2001/XMLSchema#integer" } } ] } } ] }
schema2:
{ "type": "Schema", "shapes": [
  { "id": "http://schema.example/#PersonShape",
    "type": "Shape", "expression": {
```

```
      "id": "http://schema.example/#nameExpr",
      "type": "TripleConstraint",
      "predicate": "http://xmlns.com/foaf/0.1/name" } } ] }
```

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

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

Import example 2 - Circular Import

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

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

Import example 3 - Ignored Start In Import

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

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

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

Import example 4 - Erroneous Import

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

This import fails because:

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

6.7 Schema Requirements

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

6.7.1 Schema Validation Requirement

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

1. Every, SemAct in the startActs of S has a successful evaluation of semActsSatisfied.
2. Every node n in m conforms to its associated shapeExprRefs se_n where for each shapeExprRef se_i in se_n:
    a. se_i references a ShapeExpr in shapes, and
    b. `satisfies(n, se`$_i$`, G, Sch, m)` for each shape se_i in se_n.

6.7.2 Shape Expression Reference Requirement

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

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

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

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

Following are two valid shapeExprRefs:

```
{"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"
      }
    } ]
  } ]
}
```

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

```
  } ]
}
```

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

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

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

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

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

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

6.7.3 Triple Expression Reference Requirement

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

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

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

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

Following is a valid triple expression reference:

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

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"
} ] }
```

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" }
] } } ] }
```

6.7.4 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"
    }
} ] }
```

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"
      }
  } ] }
```

6.7.5 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.

6.7.6 Examples with ShapeNot

This negated self-reference violates the negation requirement.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.7.7 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"
} } ] }
```

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"
} } } ] }
```

6.7.8 Acyclic Extension Requirement

The extension hierarchy graph must be acyclic.

6.7.9 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.

6.8 Semantic Actions

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

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

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

6.8.1 Semantics

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

6.8.2 Use - informative

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

Semantic Actions example 1

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

```
<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/s1http://a.example/o1
<n2> | <S1>  | pass   | http://a.example/s1"a"http://a.example/s1"b"
<n3> | <S1>  | fail   |
```

6.9 Annotations

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

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

6.9.1 Semantics - informative

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

Annotations example 1

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

6.10 Validation Examples

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

6.10.1 Simple Examples

Schema:

```
    | { "type": "Schema", "shapes": [
S1  |     { "id": "http://schema.example/#IntConstraint",
    |       "type": "NodeConstraint",
nc1 |       "datatype": "http://www.w3.org/2001/XMLSchema#integer"
    |     } ] }
```

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

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

Validate "30"^^<http://www.w3.org/2001/XMLSchema#integer> as IntConstraint:

- satisfies NodeConstraint ("30"^^xsd:integer, S1, G, m)
  - satisfies2 NodeConstraint ("30"^^xsd:integer, S1)

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

Schema:

```
    | { "type": "Schema", "shapes": [
S1  |   { "id": "http://schema.example/#UserShape",
    |     "type": "Shape", "expression":
tc1 |     { "type": "TripleConstraint",
    |       "predicate": "http://schema.example/#shoeSize"
    |       } } ] }
```

Data:

```
   | BASE <http://a.example/>
   | PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
t1 | <Alice> ex:shoeSize "30"^^xsd:integer .
```

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

- G = [t1] The graph G consists of one triple.
- satisfies Shape (<Alice>, S1, G, m)
  - neigh(G, <Alice>) = [t1] /* The neighborhood around <Alice> consists of one triple. */
  - matched = [t1] /* That triple is matched in the nested evaluation. */
  - remainder = Ø /* The remainder is the empty set. */
  - matches TripleConstraint ([t1], tc1, m)
  - outs = [t1] /* There is one arc out. */

  - matchables = Ø /* There are no remaining arcs out of <Alice> with predicates appearing in tc1. */
  - unmatchables = Ø /* There are no other arcs out of <Alice>. */
  - closed is false /* The Shape's closed paramater has a value of false. */

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

Schema:

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

Data:

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

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

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

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

6.10.2 Disjunction Example

Schema:

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

Data:

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

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

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

as seen in this validation.

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

- G = [t1,t2,t3,t4,t5,t6]
- satisfies Shape (<Alice>, S1, G, m)
    - neigh(G, <Alice>) = [t1,t2,t3,t4,t5], matched = [t1,t2,t3], remainder = [t4,t5]
    - matches OneOf ([t1,t2,t3], te1, m)
        - matches EachOf ([t1,t2,t3], te2, m)
            - matches cardinality ([t1,t2], tc2, m)
                - matches TripleConstraint ([t1], tc2, m)
                    - satisfies NodeConstraint ("Alice", nc2, G, m)
                        - satisfies2 NodeConstraint ("Alice", nc2)
                - matches TripleConstraint ([t2], tc2, m)
                    - satisfies NodeConstraint ("Malsenior", nc2, G, m)
                        - satisfies2 NodeConstraint ("Malsenior", nc2)
            - matches TripleConstraint ([t3], tc3, m)
                - satisfies NodeConstraint ("Walker", nc3, G, m)

            - satisfies2 NodeConstraint ("Walker", nc3)
    - outs = [t4] /* t5 is in ArcsIn(G, <Alice>)>, t6 is not in neigh(G, <Alice>)>. */
    - matchables = Ø, unmatchables = [t5], closed is false

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

Data:

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

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

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

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

Data:

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

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

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

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

```
   | { "type": "Schema", "shapes": [
S1 |   { "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:

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

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

6.10.3 Dependent Shape Example

Schema:

```
    | { "type": "Schema", "shapes": [
S1  |   { "id": "http://schema.example/#IssueShape",
    |     "type": "Shape", "expression":
tc1 |     { "type": "TripleConstraint",
    |       "predicate": "http://schema.example/#reproducedBy",
    |       "valueExpr":
nc1 |       "http://schema.example/#TesterShape" } },
S2  |   { "id": "http://schema.example/#TesterShape",
    |     "type": "Shape", "expression":
tc2 |     { "type": "TripleConstraint",
    |       "predicate": "http://schema.example/#role",
    |       "valueExpr":
    |       { "type": "NodeConstraint",
nc2 |         "values": [ "http://schema.example/#testingRole" ] } } }
    |   ] }
```

Data:

```
   | PREFIX ex: <http://schema.example/#>
   | PREFIX inst: <http://inst.example/>
t1 | inst:Issue1 ex:reproducedBy inst:Tester2 .
t2 | inst:Tester2 ex:role ex:testingRole .
```

inst:Issue1 satisfies S1 with the ShapeMap

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

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

as seen in this evaluation:

- G = [t1]
- satisfies Shape (inst:Issue1, S1, G, m)
  - neigh(G, inst:Issue1) = [t1,t2], matched = [t1,t2], remainder = Ø

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

6.10.4 Recursion Example

Schema:

```
    | { "type": "Schema", "shapes": [
S1  |   { "id": "http://schema.example/#IssueShape",
    |     "type": "Shape", "expression":
tc1 |     { "type": "TripleConstraint", "min": 0, "max": -1,
    |       "predicate": "http://schema.example/#related",
nc1 |       "valueExpr": "http://schema.example/#IssueShape"
    |     } } ] }
```

Data:

```
   | PREFIX ex: <http://schema.example/#>
   | PREFIX inst: <http://inst.example/>
t1 | inst:Issue1 ex:related inst:Issue2 .
t2 | inst:Issue2 ex:related inst:Issue3 .
t3 | inst:Issue3 ex:related inst:Issue1 .
```

inst:Issue1 satisfies S1 with the ShapeMap

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

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

as seen in this evaluation:

- G = [t1,t2,t3]
- satisfies Shape (inst:Issue1, S1, G, m)
    - neigh(G, inst:Issue1) = [t1], matched = [t1], remainder = Ø
    - matches TripleConstraint ([t1], tc1, m)
        - satisfies NodeConstraint (inst:Issue2, nc1, G, m)
            - satisfies2 NodeConstraint (inst:Issue2, nc1)

        - satisfies Shape (inst:Issue2. S2, G, m)
            - neigh(G, inst:Issue2) = [t3], matched = [t3], remainder = Ø
            - matches TripleConstraint ([t3], tc3, m)
                - satisfies NodeConstraint (inst:Issue3, nc3, G, m)
                    - satisfies2 NodeConstraint (inst:Issue3, nc3)
                        - satisfies Shape (inst:Issue3. S2, G, m)

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

matches TripleConstraint ([t3], tc3, m)

satisfies NodeConstraint (inst:Issue1, nc3, G, m)

satisfies2 NodeConstraint (inst:Issue1, nc3)
This is known to be true or the initial typing would not be satisfied.

outs = Ø

matchables = Ø, unmatchables = Ø, closed is false

            - outs = Ø
            - matchables = Ø, unmatchables = Ø, closed is false

    - outs = Ø
    - matchables = Ø, unmatchables = Ø, closed is false

6.10.5 Simple Repeated Property Examples

Schema:

```
    | { "type": "Schema", "shapes": [
S1  |     { "id": "http://schema.example/#TestResultsShape",
    |       "type": "Shape", "expression": {
te1 |         "type": "EachOf", "expressions": [
tc1 |           { "type": "TripleConstraint", "min": 1, "max": -1,
    |             "predicate": "http://schema.example/#val",
    |             "valueExpr":
nc1 |             { "type": "NodeConstraint",
    |               "values": [ {"value": "a"}, {"value": "b"}, {"value": "c"} ] } },
tc2 |           { "type": "TripleConstraint", "min": 1, "max": -1,
    |             "predicate": "http://schema.example/#val",
    |             "valueExpr":
nc2 |             { "type": "NodeConstraint",
    |               "values": [ {"value": "b"}, {"value": "c"}, {"value": "d"} ] } }
    |         ] } } ] }
```

Data:

```
   | BASE <http://a.example/>
   | PREFIX ex: <http://schema.example/#>
t1 | <s> ex:val "a" .
t2 | <s> ex:val "b" .
t3 | <s> ex:val "c" .
t4 | <s> ex:val "d" .
```

<s> satisfies S1 with:

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

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

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

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

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

- G = [t1,t2,t3]
- satisfies Shape (<Alice>, S1, G, m)
  - neigh(G, <Alice>) = [t1,t2,t3], matched = [t1,t2,t3], remainder = Ø
  - matches EachOf ([t1,t2,t3], te1, m)
    - matches cardinality ([t1,t2], tc1, m)
      - matches TripleConstraint ([t1], tc1, m)
        - satisfies NodeConstraint ("a", nc1, G, m)
          - satisfies2 NodeConstraint ("a", nc1)
      - matches TripleConstraint ([t2], tc1, m)
        - satisfies NodeConstraint ("b", nc1, G, m)
          - satisfies2 NodeConstraint ("b", nc1)
    - matches cardinality ([t3], tc2, m)
      - matches TripleConstraint ([t3], tc2, m)
        - satisfies NodeConstraint ("d", nc2, G, m)
          - satisfies2 NodeConstraint ("d", nc2)
  - outs = Ø

- ○ matchables = Ø, unmatchables = Ø, closed is false

6.10.6 Repeated Property With Dependent Shapes Example

Schema:

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

Data:

```
   | PREFIX ex: <http://schema.example/#>
   | PREFIX inst: <http://inst.example/>
t1 | inst:Issue1
t2 |   ex:reproducedBy inst:Tester2 ;
   |   ex:reproducedBy inst:Testgrammer23 .
   |
t3 | inst:Tester2
   |   ex:role ex:testingRole .
   |
t4 | inst:Testgrammer23
t5 |   ex:role ex:testingRole ;
   |   ex:department ex:ProgrammingDepartment .
```

inst:Issue1 satisfies S1 with the ShapeMap

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

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

as seen in this evaluation:

- G = [t1,t2,t3,t4,t5]

- satisfies Shape (inst:Issue1, S1, G, m)
    - neigh(G, inst:Issue1) = [t1,t2], matched = [t1,t2], remainder = Ø
    - matches EachOf ([t1,t2], te1, m)
        - matches TripleConstraint ([t1], tc1, m)
            - satisfies NodeConstraint (inst:Tester2, nc1, G, m)
                - satisfies2 NodeConstraint (inst:Tester2, nc1)
                    - satisfies Shape (inst:Tester2. S2, G, m)
                        - neigh(G, inst:Tester2) = [t3], matched = [t3], remainder = Ø
                        - matches TripleConstraint ([t3], tc3, m)
                            - satisfies NodeConstraint (ex:testingRole, nc3, G, m)
                                - satisfies2 NodeConstraint (ex:testingRole, nc3)
                        - outs = Ø
                        - matchables = Ø, unmatchables = Ø, closed is false
        - matches TripleConstraint ([t2], tc1, m)
            - satisfies NodeConstraint (inst:Testgrammer23, nc2, G, m)
                - satisfies2 NodeConstraint (inst:Testgrammer23, nc2)
                    - satisfies Shape (inst:Testgrammer23. S3, G, m)
                        - neigh(G, inst:Testgrammer23) = [t5], matched = [t5], remainder = Ø
                        - matches TripleConstraint ([t5], tc3, m)
                            - satisfies NodeConstraint (ex:testingRole, nc4, G, m)
                                - satisfies2 NodeConstraint (ex:testingRole, nc4)
                        - outs = Ø
                        - matchables = Ø, unmatchables = Ø, closed is false
    - outs = Ø
    - matchables = Ø, unmatchables = Ø, closed is false

6.10.7 Negation Example

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

Schema:

```
    | { "type": "Schema", "shapes": [
S1  |     { "id": "http://schema.example/#TestResultsShape",
    |       "type": "Shape", "expression": {
te1 |         "type": "EachOf", "expressions": [
tc1 |           { "type": "TripleConstraint", "min": 1, "max": -1,
    |             "predicate": "http://schema.example/#p1",
    |             "valueExpr":
nc1 |             { "type": "NodeConstraint",
```

```
    |                "values": [ {"value": "a"}, {"value": "b"} ] } },
tc2 |            { "type": "TripleConstraint", "min": 1, "max": -1,
    |              "predicate": "http://schema.example/#p2", "min": 0, "max": 0 }
    |          ] } } ] }
```

Data:

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

<s> satisfies S1 with:

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

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

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

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

If we add a t2 which matches tc2:

Data:

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

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

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

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

7. ShEx Compact syntax (ShExC)

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

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

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.

```
[1]     shexDoc                  : directive*
                                   ((notStartAction | startActions) statement*)?
[2]     directive                : baseDecl | prefixDecl | importDecl
[3]     baseDecl                 : "BASE" IRIREF
[4]     prefixDecl               : "PREFIX" PNAME_NS IRIREF
[5]     importDecl               : "IMPORT" IRIREF
[6]     notStartAction           : start | shapeExprDecl
[7]     start                    : "start" '=' inlineShapeExpression
[8]     startActions             : annotation* codeDecl+
[9]     statement                : directive | notStartAction
[10]    shapeExprDecl            : shapeExprLabel (shapeExpression | "EXTERNAL")
[11]    shapeExpression          : shapeOr
[12]    inlineShapeExpression    : inlineShapeOr
[13]    shapeOr                  : shapeAnd ("OR" shapeAnd)*
[14]    inlineShapeOr            : inlineShapeAnd ("OR" inlineShapeAnd)*
[15]    shapeAnd                 : shapeNot ("AND" shapeNot)*
[16]    inlineShapeAnd           : inlineShapeNot ("AND" inlineShapeNot)*
[17]    shapeNot                 : "NOT"? shapeAtom
[18]    inlineShapeNot           : "NOT"? inlineShapeAtom
[19]    shapeAtom                :    nonLitNodeConstraint shapeOrRef?
                                  | litNodeConstraint
                                  | shapeOrRef nonLitNodeConstraint?
                                  | '(' shapeExpression ')'
                                  | '.'
[20]    shapeAtomNoRef           :    nonLitNodeConstraint shapeOrRef?
                                  | litNodeConstraint
                                  | shapeDefinition nonLitNodeConstraint?
                                  | '(' shapeExpression ')'
                                  | '.'
[21]    inlineShapeAtom          :    nonLitNodeConstraint inlineShapeOrRef?
                                  | litNodeConstraint
                                  | inlineShapeOrRef nonLitNodeConstraint?
                                  | '(' shapeExpression ')'
                                  | '.'
[22]    shapeOrRef               : shapeDefinition | shapeRef

[23]    inlineShapeOrRef         : inlineShapeDefinition | shapeRef
[24]    shapeRef                 : ATPNAME_LN | ATPNAME_NS | '@' shapeExprLabel
```

```
[25]    litNodeConstraint          :    "LITERAL" xsFacet*
                                     | datatype xsFacet*
                                     | valueSet xsFacet*
                                     | numericFacet+
[26]    nonLitNodeConstraint       :    nonLiteralKind stringFacet*
                                     | stringFacet+
[27]    nonLiteralKind             : "IRI" | "BNODE" | "NONLITERAL"
[28]    xsFacet                    : stringFacet | numericFacet
[29]    stringFacet                :    stringLength INTEGER
                                     | REGEXP
[30]    stringLength               : "LENGTH" | "MINLENGTH" | "MAXLENGTH"
[31]    numericFacet               :    numericRange numericLiteral
                                     | numericLength INTEGER
[32]    numericRange               :   "MININCLUSIVE" | "MINEXCLUSIVE"
                                     | "MAXINCLUSIVE" | "MAXEXCLUSIVE"
[33]    numericLength              : "TOTALDIGITS" | "FRACTIONDIGITS"
[34]    shapeDefinition            : (extraPropertySet | "CLOSED")*
                                     '{' tripleExpression? '}'
                                     annotation* semanticActions
[35]    inlineShapeDefinition      : (extraPropertySet | "CLOSED")*
                                     '{' tripleExpression? '}'
[36]    extraPropertySet           : "EXTRA" predicate+
[37]    tripleExpression           : oneOfTripleExpr
[38]    oneOfTripleExpr            : groupTripleExpr | multiElementOneOf
[39]    multiElementOneOf          : groupTripleExpr ('|' groupTripleExpr)+
[40]    groupTripleExpr            : singleElementGroup | multiElementGroup
[41]    singleElementGroup         : unaryTripleExpr ';'?
[42]    multiElementGroup          : unaryTripleExpr (';' unaryTripleExpr)+ ';'?
[43]    unaryTripleExpr            :    ('$' tripleExprLabel)?
                                       (tripleConstraint | bracketedTripleExpr)
                                     | include
[44]    bracketedTripleExpr        : '(' tripleExpression ')' cardinality?
                                     annotation* semanticActions
[45]    tripleConstraint           : senseFlags? predicate inlineShapeExpression
                                     cardinality? annotation* semanticActions
[46]    cardinality                : '*' | '+' | '?' | REPEAT_RANGE
[47]    senseFlags                 : '^'
[48]    valueSet                   : '[' valueSetValue* ']'
[49]    valueSetValue              :   iriRange | literalRange
                                     | languageRange | exclusion+
[50]    exclusion                  : '.'
                                     (iriExclusion | literalExclusion
                                      | languageExclusion)+
[51]    iriExclusion               : '-' iri '~'?
[52]    literalExclusion           : '-' literal '~'?
[53]    languageExclusion          : '-' LANGTAG '~'?
[54]    iriRange                   :    iri ('~' iriExclusion*)?
[55]    iriExclusion               : '-' iri '~'?
[56]    literalRange               :    literal ('~' literalExclusion*)?
[57]    literalExclusion           : '-' literal '~'?
[58]    languageRange              :    LANGTAG ('~' languageExclusion*)?
                                     | '@' '~' languageExclusion*
[59]    languageExclusion          : '-' LANGTAG '~'?
[60]    include                    : '&' tripleExprLabel
[61]    annotation                 : "//" predicate (iri | literal)
[62]    semanticActions            : codeDecl*
[63]    codeDecl                   : '%' iri (CODE | '%')
[13t]   literal                    : rdfLiteral | numericLiteral | booleanLiteral
[65]    predicate                  : iri | RDF_TYPE
[66]    datatype                   : iri
[67]    shapeExprLabel             : iri | blankNode
```

```
[68]    tripleExprLabel            : iri | blankNode
[16t]   numericLiteral             : INTEGER | DECIMAL | DOUBLE
[69]    rdfLiteral                 : langString | string ("^^" datatype)?
[134s]  booleanLiteral             : "true" | "false"
[135s]  string                     :    STRING_LITERAL1 | STRING_LITERAL_LONG1
                                     | STRING_LITERAL2 | STRING_LITERAL_LONG2
[70]    langString                 :    LANG_STRING_LITERAL1 | LANG_STRING_LITERAL_LONG1
                                     | LANG_STRING_LITERAL2 | LANG_STRING_LITERAL_LONG2
[136s]  iri                        : IRIREF | prefixedName
[137s]  prefixedName               : PNAME_LN | PNAME_NS
[138s]  blankNode                  : BLANK_NODE_LABEL
[71]    <CODE>                     : "{" ([^%\\] | "\\" [%\\] | UCHAR)* "%" "}"
[72]    <REPEAT_RANGE>             : "{" INTEGER ( "," (INTEGER | "*")? )? "}"
[73]    <RDF_TYPE>                 : "a"
[18t]   <IRIREF>                   : "<" ([^#0000- <>\"{}|^`\\] | UCHAR)* ">"
[140s]  <PNAME_NS>                 : PN_PREFIX? ":"
[141s]  <PNAME_LN>                 : PNAME_NS PN_LOCAL
[74]    <ATPNAME_NS>               : "@" PNAME_NS
[75]    <ATPNAME_LN>               : "@" PNAME_LN
[76]    <REGEXP>                   : '/' ([^/\\\n\r]
                                          | '\\' [nrt\\|.?*+(){}$-\[\]^/]
                                          | UCHAR
                                         )+ '/' [smix]*
[142s]  <BLANK_NODE_LABEL>         : "_:"
                                     (PN_CHARS_U | [0-9])
                                     ((PN_CHARS | ".")* PN_CHARS)?
[145s]  <LANGTAG>                  : "@" ([a-zA-Z])+ ("-" ([a-zA-Z0-9])+)*
[19t]   <INTEGER>                  : [+-]? [0-9]+
[20t]   <DECIMAL>                  : [+-]? [0-9]* "." [0-9]+
[21t]   <DOUBLE>                   :   [+-]? ([0-9]+ "." [0-9]* EXPONENT
                                     | "."? [0-9]+ EXPONENT)
[155s]  <EXPONENT>                 : [eE] [+-]? [0-9]+
[156s]  <STRING_LITERAL1>          : "'" ([^'\\\n\r] | ECHAR | UCHAR)* "'"
[157s]  <STRING_LITERAL2>          : '"' ([^\"\\\n\r] | ECHAR | UCHAR)* '"'
[158s]  <STRING_LITERAL_LONG1>     : "'''"
                                     ( ("'" | "''")? ([^\\'\\] | ECHAR | UCHAR) )*
                                     "'''"
[159s]  <STRING_LITERAL_LONG2>     : '"""'
                                     ( ('"' | '""')? ([^\"\\] | ECHAR | UCHAR) )*
                                     '"""'
[77]    <LANG_STRING_LITERAL1>     : "'" ([^'\\\n\r] | ECHAR | UCHAR)* "'" LANGTAG
[78]    <LANG_STRING_LITERAL2>     : '"' ([^\"\\\n\r] | ECHAR | UCHAR)* '"' LANGTAG
[79]    <LANG_STRING_LITERAL_LONG1>: "'''"
                                     ( ("'" | "''")? ([^\\'\\] | ECHAR | UCHAR) )*
                                     "'''" LANGTAG
[80]    <LANG_STRING_LITERAL_LONG2>: '"""'
                                     ( ('"' | '""')? ([^\"\\] | ECHAR | UCHAR) )*
                                     '"""' LANGTAG
[26t]   <UCHAR>                    :    "\\u" HEX HEX HEX HEX
                                     | "\\U" HEX HEX HEX HEX HEX HEX HEX HEX
[160s]  <ECHAR>                    : "\\" [tbnrf\\\"\\']
[164s]  <PN_CHARS_BASE>            :    [A-Z] | [a-z]
                                     | [#00C0-#00D6] | [#00D8-#00F6] | [#00F8-#02FF]
                                     | [#0370-#037D] | [#037F-#1FFF]
                                     | [#200C-#200D] | [#2070-#218F] | [#2C00-#2FEF]
                                     | [#3001-#D7FF] | [#F900-#FDCF] | [#FDF0-#FFFD]
                                     | [#10000-#EFFFF]
[165s]  <PN_CHARS_U>               : PN_CHARS_BASE | "_"
[167s]  <PN_CHARS>                 :    PN_CHARS_U | "-" | [0-9]
                                     | [#00B7] | [#0300-#036F] | [#203F-#2040]
[168s]  <PN_PREFIX>                : PN_CHARS_BASE ( (PN_CHARS | ".")* PN_CHARS )?
```

```
[81]     <PN_LOCAL>                  : (PN_CHARS_U | ":" | [0-9] | PLX)
                                       (
                                        (PN_CHARS | "." | ":" | PLX)*
                                        (PN_CHARS | ":" | PLX)
                                       )?
[170s]   <PLX>                       : PERCENT | PN_LOCAL_ESC
[171s]   <PERCENT>                   : "%" HEX HEX
[172s]   <HEX>                       : [0-9] | [A-F] | [a-f]
[173s]   <PN_LOCAL_ESC>              :   "\\" ( "_" | "~" | "." | "-" | "!" | "$" | "&"
                                       | "'" | "(" | ")" | "*" | "+" | "," | ";" | "="
                                       | "/" | "?" | "#" | "@" | "%" )
[82]     PASSED TOKENS               :    [ \t\r\n]+
                                       | "#" [^\r\n]*
                                       | "/*" ([^*] | '*' ([^/] | '\\/'))* "*/"
```

8. 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 RDF Representation of ShEx (ShExR). Processors *MAY* interpret a ShExJ document as an RDF Graph. Processors may also transform arbitrary RDF Graphs conforming to RDF Representation of ShEx (ShExR) 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 { 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:IRIREF
valueExpr:shapeExpr?
                     min:INTEGER? max:INTEGER?
                     semActs:[SemAct+]? annotations:[Annotation+]? }
     tripleExprRef = tripleExprLabel ;
   tripleExprLabel = IRIREF | BNODE ;
            SemAct { name:IRIREF code:STRING? }
        Annotation { predicate:IRIREF object:objectValue }
# Terminals:         These follow the rules for terminals in the XML 1.0 5th Edition
                   # Turtle IRIREF without enclosing "<>"s
            IRIREF : (PN_CHARS | '.' | ':' | '/' | '\\' | '#' | '@' | '%' | '&' | UCHAR)*
;
                   # Turtle BLANK_NODE_LABEL
             BNODE : '_:' (PN_CHARS_U | [0-9]) ((PN_CHARS | '.')* PN_CHARS)? ;
                   # JSON boolean values
              BOOL : "true" | "false" ;
                   # Turtle INTEGER
           INTEGER : [+-]? [0-9] + ;
                   # Turtle DECIMAL
           DECIMAL : [+-]? [0-9]* '.' [0-9] + ;
                   # Turtle DOUBLE
            DOUBLE : [+-]?
                     (  [0-9] + '.' [0-9]* EXPONENT
                      | '.' [0-9]+ EXPONENT
                      | [0-9]+ EXPONENT
                     ) ;
                   # BCP47 Language-Tag
           LANGTAG : [a-zA-Z]+ ('-' [a-zA-Z0-9]+)* ;
                   # any JSON string
            STRING : .* ;
                   # empty string
             EMPTY : ^$ ;
# Components:        Terminals 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]+ ;
```

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. Every ShExR document is graph isomorphic[rdf11-concepts] to the RDF serialization[json-ld] of some ShExJ document.

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

<#Schema> CLOSED {
  a [sx:Schema] ;
  sx:imports @<#IriList1Plus> ? ;
  sx:startActs @<#SemActList1Plus> ? ;
  sx:start @<#shapeDeclOrExpr> ? ;
  sx:shapes @<#ShapeDeclList1Plus> ?
}

<#shapeDeclOrExpr> @<#ShapeDecl> OR @<#shapeExpr>

<#ShapeDecl> CLOSED {
  a [sx:ShapeDecl] ;
  sx:abstract [true false] ? ;
  sx:shapeExpr @<#shapeExpr>
}

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

<#ShapeOr> CLOSED {
  a [sx:ShapeOr] ;
  sx:shapeExprs @<#shapeDeclOrExprList2Plus>
}

<#ShapeAnd> CLOSED {
  a [sx:ShapeAnd] ;
  sx:shapeExprs @<#shapeDeclOrExprList2Plus>
}

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

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

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

<#Shape> CLOSED {
  a [sx:Shape] ;
  sx:extends @<#shapeDeclOrExprList1Plus>? ;
  sx:closed [true false] ? ;
  sx:extra IRI * ;
  sx:expression @<#tripleExpression> ? ;
  sx:semActs @<#SemActList1Plus> ? ;
  sx:annotation @<#AnnotationList1Plus> ?
}

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

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

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

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

<#valueSetValue> @<#objectValue> OR @<#IriStem> OR @<#IriStemRange>
                               OR @<#LiteralStem> OR @<#LiteralStemRange>
                OR @<#Language> OR @<#LanguageStem> OR @<#LanguageStemRange>
<#objectValue> IRI OR LITERAL

<#IriStem> CLOSED { a [sx:IriStem] ; sx:stem xsd:string }
<#IriStemRange> CLOSED {
  a [sx:IriStemRange] ;
  sx:stem xsd:string OR @<#Wildcard> ;
  sx:exclusion @<#IriStemExclusionList1Plus>
}
```

```
<#LiteralStem> CLOSED { a [sx:LiteralStem] ; sx:stem xsd:string }
<#LiteralStemRange> CLOSED {
  a [sx:LiteralStemRange] ;
  sx:stem xsd:string OR @<#Wildcard> ;
  sx:exclusion @<#LiteralStemExclusionList1Plus>
}

<#Language> CLOSED { a [sx:Language] ; sx:languageTag xsd:string }
<#LanguageStem> CLOSED { a [sx:LanguageStem] ; sx:stem xsd:string }
<#LanguageStemRange> CLOSED {
  a [sx:LanguageStemRange] ;
  sx:stem xsd:string OR @<#Wildcard> ;
  sx:exclusion @<#LanguageStemExclusionList1Plus>
}

<#Wildcard> BNODE CLOSED {
  a [sx:Wildcard]
}

<#tripleExpression>
     @<#NotYetResolvedInclusion>
  OR @<#TripleConstraint>
  OR @<#OneOf>
  OR @<#EachOf>

<#NotYetResolvedInclusion> CLOSED {} # will have 1 incoming, 0 outgoing arcs

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

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

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

# RDF Lists

<#tripleExpressionList2Plus> CLOSED {
  rdf:first @<#tripleExpression> ;
```

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

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

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

<#ShapeDeclList1Plus> CLOSED {
  rdf:first @<#ShapeDecl> ;
  rdf:rest  [rdf:nil] OR @<#ShapeDeclList1Plus>
}

<#shapeDeclOrExprList2Plus> CLOSED {
  rdf:first @<#shapeDeclOrExpr> ;
  rdf:rest  @<#shapeDeclOrExprList1Plus>
}
<#shapeDeclOrExprList1Plus> CLOSED {
  rdf:first @<#shapeDeclOrExpr> ;
  rdf:rest  [rdf:nil] OR @<#shapeDeclOrExprList1Plus>
}

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

<#AnnotationList1Plus> CLOSED {
  rdf:first @<#Annotation> ;
  rdf:rest  [rdf:nil] OR @<#AnnotationList1Plus>
}

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

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

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

10. IANA Considerations

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

10.1 text/shex

Type name:
text

Subtype name:
shex

Required parameters:
None

Optional parameters:
None

Encoding considerations:
8-bit text

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

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

Interoperability considerations:
Not Applicable

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

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

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

Restrictions on Usage:
None

Provisional Registrations:
Not Applicable

Additional information:
Deprecated alias names for this type:
None

Magic number(s):
File extension(s):
.shex

Macintosh file type code(s):
TEXT

Intended usage:
Common

Other Information & Comments:
None

Contact Person:
Contact Name:
Eric Prud'hommeaux

Contact Email Address:
eric@w3.org

Change controller:
W3C

11.