Shape Expressions (ShEx) 2.1 Primer

Final Community Group Report 09 October 2019

This version:
http://shex.io/shex-primer-20191009

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

Latest editor's draft:
https://shexspec.github.io/primer/

Previous version:
http://shex.io/shex-primer-20170713

Editors:
Thomas Baker (Dublin Core Metadata Initiative)
Eric Prud'hommeaux (W3C/MIT)

Abstract

Shape Expressions (ShEx) is a language for describing RDF graph structures. A ShEx schema prescribes conditions that RDF data graphs must meet in order to be considered "conformant": which subjects, predicates, and objects may appear in a given graph, in what combinations and with what cardinalities and datatypes. In the ShEx model, an RDF graph is tested against a ShEx schema to yield a validation result that flags any parts of the data which do not conform. ShEx schemas are intended for use in validating RDF data, communicating interface parameters and data structures, generating user interfaces, and transforming RDF graphs into other data formats and structures. This primer introduces ShEx by means of annotated examples. Readers should already be familiar with the basic concepts of RDF. The primer is a companion to the full ShEx language specification [shex-semantics].

Status of This Document

This specification was published by the Shape Expressions Community Group. It is not a W3C Standard nor is it on the W3C Standards Track. Please note that under the W3C Community Final Specification Agreement (FSA) other conditions apply. Learn more about W3C Community and Business Groups.

This version of the document represents a Candidate Release, with stable features. Comments and implementations are solicited prior to an eventual Final 2.0 Release.

The Shape Expressions language is expected to remain stable with the exception of:

- addition of a UNIQUE function

If you wish to make comments regarding this document, please send them to public-shex@w3.org (subscribe, archives).

Table of Contents

1. Quick Start §

Shape Expressions (ShEx) is a language for describing RDF graph structures. A **ShEx schema** prescribes conditions that RDF data graphs must meet in order to be considered "conformant". In the ShEx model, a **shape map** specifies which nodes in an RDF graph will be tested against a ShEx schema. ShEx schemas are intended for use in validating instance data, communicating interface parameters and data structures, generating user interfaces, and transforming RDF graphs into other data formats and structures. This primer, a companion to the full ShEx language specification [shex-semantics], focuses on the common use case of validating instance data.

A ShEx schema is built on **node constraints** and **triple constraints** that define what it means for a given RDF data graph to conform. An RDF triple is the three-part data structure of subject, predicate, and object with which all RDF data is expressed, and an RDF node is the piece of data found in the subject or object position of a triple. (Readers unfamiliar these terms may want to consult an RDF primer.[rdf11-primer]) Node constraints and triple constraints are called "constraints" because they define, or "constrain", the set of RDF nodes and data triples that will pass a conformance test.

Picture an RDF database (graph) that carries information about enrollees in a school. Put yourself into the position of a data manager who wants to ensure that every student "record" in this graph reports a valid age and references one or two guardians, identified by IRI. The ShEx schema for accomplishing this has: one node constraint, school:enrolleeAge, for matching data nodes with an integer value between 13 and 20; one triple constraint, defined within the shape school:Enrollee, for matching one or two triples having the predicate ex:hasGuardian, and a second node constraint for specifying that the object (value) of the triple is an IRI. Data that conforms to these constraints will pass validation tests, and data that does not conform will fail.

Schema (ShExC)

```
PREFIX school: <http://school.example/#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://ex.example/#>

# Node constraint
school:enrolleeAge xsd:integer MinInclusive 13 MaxInclusive 20


school:Enrollee {
  # Triple constraint (including node constraint IRI)
  ex:hasGuardian IRI {1,2}
}
```

Passing Data (Turtle) try it: js | scala

```
PREFIX ex: <http://ex.example/#>
PREFIX inst: <http://example.com/users/>

inst:Student1 ex:hasGuardian
  inst:Person2, inst:Person3 .
```

Failing Data (Turtle) try it: js | scala

```
PREFIX ex: <http://ex.example/#>
PREFIX inst: <http://example.com/users/>

inst:Student2 ex:hasGuardian
  inst:Person4, inst:Person5, inst:Person6 .
```

The next example adds a triple constraint on the data predicate foaf:age, which must have a value matching the node constraint school:enrolleeAge, which is cited in the triple constraint by reference, indicated by the '@' symbol.

Schema (ShExC)

```
PREFIX ex: <http://ex.example/#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX school: <http://school.example/#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

school:enrolleeAge xsd:integer MinInclusive 13 MaxInclusive 20

school:Enrollee {
  foaf:age @school:enrolleeAge ;
  ex:hasGuardian IRI {1,2}
}
```

Passing Data (Turtle) try it: js | scala

```
PREFIX ex: <http://ex.example/#>
PREFIX inst: <http://example.com/users/>
PREFIX school: <http://school.example/#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

inst:Alice foaf:age 13 ;
  ex:hasGuardian inst:Person2, inst:Person3 .

inst:Bob foaf:age 15 ;
  ex:hasGuardian inst:Person4 .
```

Failing Data (Turtle) try it: js | scala

```
PREFIX ex: <http://ex.example/#>
PREFIX inst: <http://example.com/users/>
PREFIX school: <http://school.example/#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

inst:Claire foaf:age 12 ;
  ex:hasGuardian inst:Person5 .

inst:Don foaf:age 14 .
```

The four RDF data nodes inst:Alice, inst:Bob, inst:Claire, and inst:Don can be tested against the shape school:Enrollee by using either a **fixed shape map**, which selects nodes by simple enumeration, or a **query shape map**, which selects nodes matching a triple pattern. In the latter, the shape map keyword "FOCUS" identifies the node of the triple to be selected (here, the subject node), and the keyword "_" acts as a placeholder for any object. For readability, the IRIs are rendered here with prefixes.

Fixed Shape Map

```
inst:Alice @ school:Enrollee,
inst:Bob @ school:Enrollee,
inst:Claire @ school:Enrollee,
inst:Don @ school:Enrollee
```

Query Shape Map

```
{FOCUS foaf:age _} @ school:Enrollee
```

This example yields a validation result in the form of a **result shape map**. A validation result may be presented in different formats and with different levels of verbosity according to implementation requirements.

Result Shape Map

Node	Shape	Result	Reason
inst:Alice	school:Enrollee	pass	
inst:Bob	school:Enrollee	pass	
inst:Claire	school:Enrollee	fail	foaf:age 12 less than 13.
inst:Don	school:Enrollee	fail	No ex:hasGuardian supplied.

2. ShEx Model and Terminology §

2.1 Validating RDF Data §

In the Shape Expressions model, RDF data is seen from the standpoint of its structural components, or abstract syntax. An **RDF graph** is a collection of **triples**. A triple is a data structure composed of three **RDF terms**. An RDF term may be an **IRI**, **blank node (BNode)**, or **literal**. In a triple, RDF terms are arranged in a fixed order, or **directed arc**, from **subject** to **predicate** to **object**. A node in the subject position has an **outgoing arc** and a node in the object position has an **incoming arc**. RDF data may be serialized in any of several interchangeable concrete syntaxes designed for a variety of application requirements.

A **ShEx schema** is a collection of **shape expressions** that describe an RDF graph in terms of these abstract-syntactic components. A shape expression is a logical combination of **node constraints** and **shapes**. Node constraints define the characteristics of matching RDF nodes. A shape describes a collection of RDF triples touching a given RDF node in terms of **triple constraints**. Triple constraints specify matching RDF triples in terms of their predicates, direction (whether they are incoming or outgoing arcs with respect to a node), cardinality (how many triples should match), or value (characteristics of its subject or object node).

In the **ShEx model**, a given RDF data graph is tested against a ShEx schema to yield a **validation result**. In the example above, the RDF nodes inst:Alice, inst:Bob, inst:Claire and inst:Don are tested against the ShEx shape school:Enrollee. In the validation process, each of four nodes in the RDF data is treated, in turn, as a **focus node**, and triples involving that node are tested against a triple constraint which, in turn, includes the node constraint IRI. This validation process is controlled by a **shape map** that specifies how the constructs of a ShEx schema relate to the components of RDF data graphs. There are many ways to populate a shape map with nodes to be validated: through queries, APIs, protocols, or simple enumeration.

ShEx may be serialized using any of three interchangeable concrete syntaxes: Shape Expressions Compact Syntax or **ShExC**, a compact syntax meant for human eyes and fingers; **ShExJ**, a JSON-LD [json-ld] (Javascript) syntax meant for machine processing; and **ShExR**, the RDF interpretation of ShExJ expressed in RDF Turtle syntax [turtle]. The ShEx schemas in this primer may be viewed in ShExC syntax (by default) or JSON-LD syntax by pressing 'c' or 'j' in the browser or by clicking on the following radio buttons:

◉ ShExC ('c') ○ JSON ('j') .

2.2 Basic Terminology §

ShEx schema
: A collection of shape expressions.

shape expression
: Shapes and node constraints, possibly combined with AND, OR, and NOT expressions.

focus node
: An RDF data node of interest that is examined during validation.

shape
: A triple expression against which a focus node is tested to see whether all incoming or outgoing arcs, which match predicates in the triple expression, have the appropriate cardinality and values.

triple expression
: A collection of triple constraints which may be combined in groups or choices.

triple constraint
: A constraint with a given predicate which is tested against RDF triples with that predicate and the focus node as subject (or object, for inverse triple constraints).

node constraint
: A collection of constraints on a given focus node.

value constraint
: A shape expression used in a triple constraint. "Value constraint" (or "value constraint on a property") is a shorthand way of saying: "node constraint on the RDF node of the object associated in a triple with a given predicate". For inverse triple constraints, this shorthand is: "node constraint on the RDF node of the subject associated in a triple with a given predicate".

value set
: A set of values composed of RDF terms and substrings.

value
: A shorthand designation for the RDF node at the opposite end of an RDF data triple from a focus node. In typical usage, this is the object of a triple, or for inverse triple constraints, its subject.

constraint
: A restriction on the set of permissible nodes or triples.

shape map
: A collection of pairs of RDF data nodes and ShEx shapes that is used for invoking validation and reporting results.

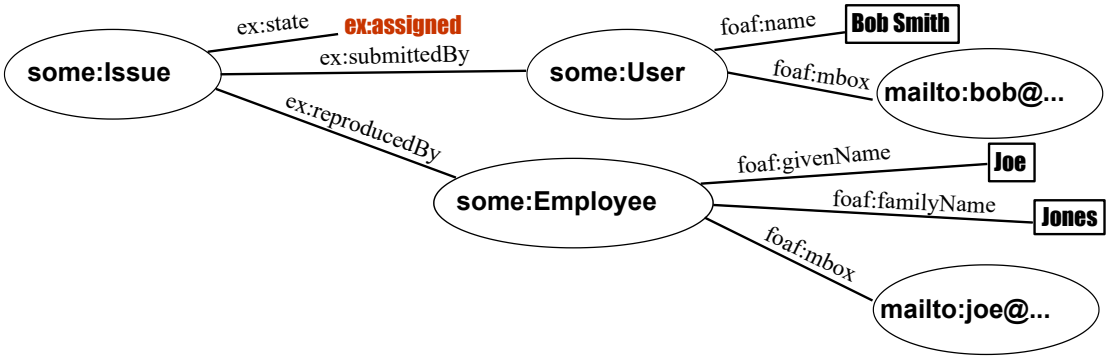
ShExC
: A compact syntax meant for human eyes and fingers.

ShExJ
: A JSON-LD [json-ld] representation meant for machine processing.

ShExR
: The RDF interpretation of ShExJ expressable in any RDF syntax.

3. ShEx Essentials §

This document uses a running example illustrated by the following graph in which an Issue is submitted by some person and may be reproduced by the same person or someone else. These issues can have a status of `unassigned` or `assigned`.



3.1 Node Constraints §

The following node constraints can be used alone, or in combination.

literal datatype
A node constraint that identifies the datatype of an RDF literal.

Schema (ShExC)
```
my:UserShape {
  foaf:name xsd:string
}
```

literal facet
A node constraint that applies XML Schema constraining facets, including **numeric facets**, which apply only to numeric RDF literals (`MinInclusive`, `MinExclusive`, `MaxInclusive`, `MaxExclusive`, `TotalDigits`, `FractionDigits`) and **string facets**, which apply to all RDF literals (`Length`, `MinLength`, `MaxLength`, `Pattern`). In the ShExC syntax, facet names are not case-sensitive.

Schema (ShExC)
```
my:UserShape {
  ex:shoeSize xsd:float MinInclusive 5.5 MaxInclusive 12.5
}
```

node kind
A node constraint that specifies whether an RDF data node is of kind `Literal`, `IRI`, `BNode`, or `NonLiteral`, a union of the kinds `IRI` and `BNode`. In the ShExC syntax, node kinds are not case-sensitive.

Schema (ShExC)
```
my:UserShape {
  foaf:mbox IRI
}
```

value set
Enumerates a set of specific expected values.

Schema (ShExC)
```
my:IssueShape {
  ex:state [ex:unassigned ex:assigned]
}
```

Schema (ShExC)
```
my:IssueShape {
  ex:state ["unassigned" "assigned"]
}
```

Schema (ShExC)
```
my:IssueShape {
  ex:state [0 1]
}
```

value shape
Asserts that the value is described by another shape, e.g.

Schema (ShExC)
```
my:IssueShape {
  ex:reportedBy @my:UserShape
}
```

3.2 Triple Constraints §

Triple constraints are evaluated against all of the triples in an RDF graph that touch a given data node. The RDF data node examined during validation is called the **focus node**. Triple constraints identify a predicate and describe matching triples as having the focus node as the subject (or object). (The exception is the inverse triple constraint, which describes matching triples as having the focus node as the object.) Triple constraints specify a cardinality (how many matching triples are

expected or permitted). They may include a value constraint, which is a node constraint describing the object of matching triples (or subject, in the case of inverse triple constraints).

The following simple example has one shape, my:UserShape, with a single triple constraint on the property foaf:name. A value constraint within the triple constraint says that the object of a foaf:name triple must be an RDF literal with a datatype of xsd:string. A conforming node in an RDF data graph will have exactly one such triple.

Schema (ShExC)

```
my:UserShape {
  foaf:name xsd:string
}
```

Passing Data (Turtle) try it: js | scala

```
inst:User1
    foaf:name  "Bob Smith"^^xsd:string .

inst:User2
    foaf:name  "Bob Smith" .
```

Failing Data (Turtle) try it: js | scala

```
inst:User3
    foaf:name  "Joe Jones"^^xsd:string ;
    foaf:name  "J. Jones"^^xsd:string .

inst:User4
    foaf:name  "Bob Smith"^^xsd:anyURI .
```

Result Shape Map

Node	Shape	Result	Reason
inst:User1	my:UserShape	pass	
inst:User2	my:UserShape	pass	Literals in Turtle have a default datatype of xsd:string.
inst:User3	my:UserShape	fail	Expected exactly one foaf:name arc.
inst:User4	my:UserShape	fail	Expected an xsd:string, not a xsd:anyURI.

The following regular expression conventions are used to specify cardinalities other than the default of "exactly one" (see the example in Quick Start, with cardinality "one or two").

- "+" - one or more
- "*" - zero or more
- "**?**" - zero or one
- "**{m}**" - exactly m
- "**{m,n}**" - at least m, no more than n

3.3 Grouping Triple Constraints in Shapes §

The examples above show just one triple constraint per shape. In practice, most shape declarations will include multiple constraints. In the ShExC syntax, triple constraints are separated by a semi-colon. In the following example, the shape my:IssueShape will match:

- exactly one data triple with predicate ex:state and a value in the value set ex:unassigned or ex:assigned,
- exactly one data triple with predicate ex:reportedBy and a value matching the shape labeled my:UserShape.

The shape labeled my:UserShape must have:

- exactly one triple constraint on predicate foaf:name with a value of the literal datatype xsd:string
- one or more triple constraints on predicate foaf:mbox with a `node kind` of IRI.

Schema (ShExC)

```
my:IssueShape {
  ex:state [ex:unassigned ex:assigned];
  ex:reportedBy @my:UserShape
}

my:UserShape {
  foaf:name xsd:string;
  foaf:mbox IRI+
}
```

Passing Data (Turtle) try it: js | scala

```
inst:Issue1 a ex:Issue ;
    ex:state        ex:unassigned ;
    ex:reportedBy   inst:User2 .

inst:User2 a foaf:Person ;
    foaf:name       "Bob Smith" ;
    foaf:mbox       <mailto:bob@example.org> ;
    foaf:mbox       <mailto:rs@example.org> .
```

Failing Data (Turtle) try it: js | scala

```
inst:Issue3 a ex:Issue ;
    ex:state        ex:unsinged ; # <-- typo
    ex:reportedBy   inst:User4 .

inst:User4 a foaf:Person ;
    foaf:name       "Bob Smith", "Robert Smith" ;
    foaf:mbox       <mailto:bob@example.org> ;
    foaf:mbox       <mailto:rs@example.org> .
```

Result Shape Map

Node	Shape	Result	Reason
inst:Issue1	my:IssueShape	pass	
inst:User2	my:UserShape	pass	
inst:Issue3	my:IssueShape	fail	ex:unsinged not in range of ex:status.
inst:User4	my:UserShape	fail	Max cardinality of foaf:name exceeded.

Note that a shape composed of triple expressions with a minimum cardinality of zero will trivially match any RDF node that does not have outgoing arcs.

Schema (ShExC)

```
my:IssueShape {
  ex:state [ex:unassigned ex:assigned];
  ex:reportedBy @my:UserShape
}

my:UserShape {
  foaf:name LITERAL?;
  foaf:mbox IRI*
}
```

Passing Data (Turtle) try it: js | scala

```
inst:Issue1 a ex:Issue ;
    ex:state        ex:unassigned ;
    ex:reportedBy   "Bob Smith" .
```

Result Shape Map

Node	Shape	Result	Reason
inst:Issue1	my:IssueShape	pass	
"Bob Smith"	my:UserShape	pass	Matched 0 foaf:name and 0 foaf:mbox arcs.

For both properties in my:UserShape, the minimum number of triples is zero. Any RDF data node will match this shape as long as it does not have conflicting ex:state or ex:reportedBy properties. Since no RDF literal may be the subject of RDF triples, any literal would satisfy this shape. This can lead to unintended results and confusing errors.

3.4 Combining Value Constraints §

Value constraints can be combined, for example to say that an issue conforms to a given shape and is identified by an IRI matching a certain pattern.

Schema (ShExC)

```
my:IssueShape {
  ex:state [ex:accepted ex:resolved];
  ex:reproducedBy @my:EmployeeShape
}

my:EmployeeShape IRI
 /^http:\/\/hr\.example\/id#[0-9]+/ {
  foaf:name LITERAL;
  ex:department [ex:ProgrammingDepartment]
}
```

Passing Data (Turtle) try it: js | scala

```
inst:Issue1
  ex:state ex:accepted ;
  ex:reproducedBy <http://hr.example/id#123> .

<http://hr.example/id#123>
  foaf:name "Bob Smith" ;
  ex:department ex:ProgrammingDepartment .
```

Failing Data (Turtle) try it: js | scala

```
inst:Issue1
  ex:state ex:accepted ;
  ex:reproducedBy <http://hr.example/id#abc> .

<http://hr.example/id#abc>
  foaf:name "Bob Smith" ;
  ex:department ex:ProgrammingDepartment .
```

Result Shape Map

Node	Shape	Result	Reason
inst:User1	my:UserShape	pass	
<http://hr.example/id#123>	my:EmployeeShape	pass	
inst:User3	my:UserShape	fail	Object of ex:reproducedBy does not match my:EmployeeShape.
<http://hr.example/id#abc>	my:EmployeeShape	fail	<http://hr.example/id#abc> does not match regular expression.

3.5 Nesting Shapes §

In the examples above, value expressions referenced shape expressions with the '@' symbol. It is also possible to write "anonymous" shapes directly in the value expression. In the example below, the value of the triple constraint on predicate ex:reportedBy is a shape comprised of triples constraints on the predicates foaf:name and foaf:mbox. In ShExC syntax, this anonymous shape is enclosed in curly brackets. In ShExJ, it is contained in the triple constraint's valueExpr property.

Schema (ShExC)

```
my:IssueShape {
  ex:state [ex:unassigned ex:assigned];
  ex:reportedBy {
    foaf:name LITERAL;
    foaf:mbox IRI+
  }
}
```

3.6 Expressions with Choices §

Most schema languages offer a way to express choices. In the following example, a user has either a simple name (foaf:name), or a composite name (foaf:familyName with one or more foaf:givenNames). The user must have exactly one foaf:mbox.

Schema (ShExC)

```
my:UserShape {
  (
     foaf:name LITERAL

   |
     foaf:givenName LITERAL+;
     foaf:familyName LITERAL
  );
  foaf:mbox IRI
}
```

Passing Data (Turtle) try it: js | scala

```
inst:User1 a foaf:Person ;
    foaf:name       "Alice Walker" ;
    foaf:mbox       <mailto:awalker@example.org> .

inst:User2 a foaf:Person ;
    foaf:givenName  "Robert" ;
    foaf:givenName  "Paris" ;
    foaf:familyName "Moses" ;
    foaf:mbox       <mailto:rpmoses@example.org> .
```

Failing Data (Turtle) try it: js | scala

```
inst:User3 a foaf:Person ;
    foaf:givenName  "Smith" ;
    foaf:mbox       <mailto:bobs@example.org> .

inst:User4 a foaf:Person ;
    foaf:name       "A" ;
    foaf:givenName  "B" ;
    foaf:familyName "C" ;
    foaf:mbox       <mailto:bobs@example.org> .

inst:User5 a foaf:Person ;
    foaf:name       "A" ;
    foaf:givenName  "B" ;
    foaf:mbox       <mailto:bobs@example.org> .
```

Result Shape Map

Node	Shape	Result	Reason
inst:User1	my:UserShape	pass	
inst:User2	my:UserShape	pass	
inst:User3	my:UserShape	fail	Expected a foaf:familyName arc.
inst:User4	my:UserShape	fail	Expected only one disjunction to pass.
inst:User5	my:UserShape	fail	Extra foaf:givenName arc.

3.7 Value Sets §

A **value set** is a value constraint for enumerating the set of permissible RDF nodes. The values can be IRIs, literals or language tags. In the example below, "`[@en @fr]`" matches any RDF literal with a language tag of "en" or "fr".

Schema (ShExC)

```
my:IssueShape {
  ex:state [ex:unassigned ex:assigned]
}
```

Schema (ShExC)

```
my:IssueShape {
  ex:state ["unassigned" "assigned"]
}
```

Schema (ShExC)

```
my:IssueShape {
  ex:label [@en @fr]
}
```

Fixed values are represented as value sets with one member. For example, a required type triple can be expressed in the following two ways using the ShExC syntax:

Schema (ShExC)

```
my:UserShape {
  rdf:type [foaf:Person]
}
```

Schema (ShExC)

```
my:UserShape {
  a [foaf:Person]
}
```

A value set can also contain **stems** for IRIs, strings and language tags. An IRI orx string stem matches any IRI or string that starts with the stem. A language tag stem matches any literal with a language tag that starts with the stem. These are flagged

as stems by appending a tilde ("~").

The last triple constraint in my:IssueShape below is an inverse triple constraint. These are described in the next section. This inverse triple constraint states that every my:IssueShape must have an incoming arc from one of two nodes: my:Product1 or my:Product2.

Schema (ShExC)

```
my:IssueShape {
  ex:status    [ excodes:~ auxterms:~ ];
  ex:mood      [ @en~ - @en-fr ];
 ^ex:hasIssue [ my:Product1 my:Product2 ]
}
```

Passing Data (Turtle) try it: js | scala

```
inst:Issue1 ex:status   excodes:resolved ;
            ex:mood     "hungry"@en-gb .
my:Product2 ex:hasIssue inst:Issue1 .
```

Failing Data (Turtle) try it: js | scala

```
inst:Issue2 ex:status   ex:done ;
            ex:mood     "angry"@en-fr .
my:Product1 ex:hasIssue inst:Issue2 .

inst:Issue3 ex:status  auxterms:done.
my:Product3 ex:hasIssue inst:Issue3 .
```

Result Shape Map

Node	Shape	Result	Reason
inst:Issue1	my:IssueShape	pass	
inst:Issue2	my:IssueShape	fail	ex:done not in range of ex:status. Excluded language tag for ex:mood.
inst:Issue3	my:IssueShape	fail	my:Product3 not in range of ex:hasIssue.

While IRI, string and language tag stems match any terms starting with the stem, one may wish to exclude specific terms or term stems. **Stem exclusions** are expressed by following a stem with one or more **exclusions**. An exclusion consists of a minus ("-") followed by an IRI, IRI stem, string, string stem, language tag, or language tag stem.

Schema (ShExC)

```
my:IssueShape {
  ex:status [
      excodes:~ - excodes:unassigned - excodes:assigned
      auxterms:~ - <http://aux.example/terms#med_>~
  ]
}
```

Data (Turtle) try it: js | scala

```
inst:Issue2 ex:status  excodes:resolved .

inst:Issue3 ex:status  excodes:assigned.

inst:Issue4 ex:status  auxterms:med_sniffles.

inst:Issue5 ex:status  ex:done.
```

Result Shape Map

Node	Shape	Result	Reason
inst:Issue2	my:IssueShape	pass	
inst:Issue3	my:IssueShape	fail	excodes:assigned excluded.
inst:Issue4	my:IssueShape	fail	auxterms:med_… terms excluded.
inst:Issue5	my:IssueShape	fail	ex:done not in range of ex:status.

A period (.) can be used with exclusions to say that any term is permitted except the exclusions.

Schema (ShExC)

```
my:IssueShape {
  ex:status [
    . - excodes:retracted - excodes:assigned
  ]
}
```

Passing Data (Turtle) try it: js | scala

```
inst:Issue1 ex:status  ex:random .
```

Failing Data (Turtle) try it: js | scala

```
inst:Issue2 ex:status  excodes:assigned.
```

Result Shape Map

Node	Shape	Result	Reason
inst:Issue1	my:IssueShape	pass	
inst:Issue2	my:IssueShape	fail	excodes:assigned is excluded.

4. Advanced Concepts §

4.1 Inverse Triple Constraints §

Placing a regular triple constraint on predicate ex:reportedIssue into my:UserShape effectively requires all users to have reported an issue.

Schema (ShExC)

```
my:IssueShape {
  ex:state [ex:unassigned ex:assigned]
}

my:UserShape {
  foaf:name LITERAL;
  foaf:mbox IRI+;
  ex:reportedIssue @my:IssueShape
}
```

Data (Turtle)

```
inst:Issue1 a ex:Issue ;
    ex:state        ex:unassigned .

inst:User2 a foaf:Person ;
    foaf:name       "Bob Smith" ;
    foaf:mbox       <mailto:bob@example.org> ;
    foaf:mbox       <mailto:rs@example.org> .

inst:User3 a foaf:Person ;
    foaf:name       "Bob Smith" ;
    ex:reportedIssue inst:Issue1 ;
    foaf:mbox       <mailto:bob@example.org> ;
    foaf:mbox       <mailto:rs@example.org> .
```

Result Shape Map

Node	Shape	Result	Reason
inst:Issue1	my:IssueShape	pass	
inst:User2	my:UserShape	fail	Expected ex:reportedIssue property.
inst:User3	my:UserShape	pass	

However, it may be more precise to require that for every issue, there is some user who reported it. This could be expressed in the data by describing every issue as being reported by (ex:reportedBy) a user. Alternatively, an issue could be described as the object of a triple with predicate ex:reportedIssue. A triple constraint for such an incoming arc may expressed in the ShExC syntax by prefixing the constraint with a caret (^) and in the ShExJ syntax with "inverse": true.

Schema (ShExC)

```
my:IssueShape {
  ex:state [ex:unassigned ex:assigned];
 ^ex:reportedIssue @my:UserShape
}

my:UserShape {
  foaf:name LITERAL;
  foaf:mbox IRI+
}
```

Passing Data (Turtle)

```
inst:Issue1 a ex:Issue ;
    ex:state        ex:unassigned .
inst:User1 a foaf:Person ;
    foaf:name       "Bob Smith" ;
    ex:reportedIssue inst:Issue1 ;
    foaf:mbox       <mailto:bob@example.org> ;
    foaf:mbox       <mailto:rs@example.org> .
```

Failing Data (Turtle)

```
inst:Issue2 a ex:Issue ;
    ex:state         ex:unassigned ;
    ex:reportedBy    inst:User2 .
inst:User2 a foaf:Person ;
    foaf:name        "Bob Smith" ;
    foaf:mbox        <mailto:bob@example.org> ;
    foaf:mbox        <mailto:rs@example.org> .

inst:Issue3 a ex:Issue ;
    ex:state         ex:unassigned ;
    ex:reportedIssue inst:User3 .
inst:User3 a foaf:Person ;
    foaf:name        "Bob Smith" ;
    foaf:mbox        <mailto:bob@example.org> ;
    foaf:mbox        <mailto:rs@example.org> .

inst:Issue4 a ex:Issue ;
    ex:state         ex:unassigned .
inst:User4 a foaf:Person ;
    foaf:name        "Robert" ;
    foaf:name        "Bob Smith" ;
    foaf:mbox        <mailto:bob@example.org> ;
    ex:reportedIssue inst:Issue4 .
```

Result Shape Map

Node	Shape	Result	Reason
inst:Issue1	my:IssueShape	pass	
inst:Issue2	my:IssueShape	fail	Missing incoming ex:reportedIssue property.
inst:Issue3	my:IssueShape	fail	ex:reportedIssue is in the wrong direction.
inst:Issue4	my:IssueShape	fail	Subject of ex:reportedIssue does not match my:UserShape;
inst:User4	my:UserShape	fail	Max cardinality of foaf:name exceeded.

4.2 Negative Triple Constraints §

A schema may specify triples that *must not* appear in the data. For example, suppose there were a need for free-standing issues -- issues having no ex:component relationships, incoming or outgoing, with any other issues. This can be expressed by setting the cardinality of triple constraints on the predicate ex:component to zero, meaning that they should match zero triples in the data. That the shape must also not be the target of incoming triples with the predicate ex:component is expressed by prefixing the triple constraint with a caret (^).

Schema (ShExC)

```
my:SolitaryIssueShape {
  ex:state [ex:unassigned ex:assigned];
  ex:component . {0} ;
 ^ex:component . {0}
}
```

Data (Turtle)

```
inst:Issue1 a ex:Issue ;
    rdfs:label      "smokes too much" ;
    ex:state        ex:unassigned .

inst:Issue2 a ex:Issue ;
    dc:creator      "Alice" ;
    ex:state        ex:unassigned ;
    ex:component    inst:Issue3 .

inst:Issue3 a ex:Issue ;
    rdfs:label      "smokes too little" ;
    ex:state        ex:unassigned .
```

Result Shape Map

Node	Shape	Result	Reason
inst:Issue1	my:SolitaryIssueShape	pass	
inst:Issue2	my:SolitaryIssueShape	fail	Expected zero outgoing ex:component arcs.
inst:Issue3	my:SolitaryIssueShape	fail	Expected zero incoming ex:component arcs.

4.3 Closing Shapes §

Services backed by an RDF triple store may simply accept and store any triples not described in the schema; in such a case, the schema may only identify triples that the service understands and manipulates. At the other extreme are services or databases that accept or emit *only* the data structures described in a schema. In a ShEx schema, a shape may be defined to match *only* RDF data nodes that have outgoing triples matching the given set of triple constraints and *no other* outgoing triples. A shape declaration can be qualified to mean "this set of outgoing triples and no others" by using the keyword CLOSED.

Schema (ShExC)

```
my:OpenUserShape {
  foaf:name xsd:string;
  foaf:mbox IRI
}

my:ClosedUserShape CLOSED {
  foaf:name xsd:string;
  foaf:mbox IRI
}
```

Data (Turtle)

```
inst:User1
    foaf:name       "Bob Smith" ;
    foaf:mbox       <mailto:rs@example.org> .

inst:User2 a foaf:Person ;
    foaf:name       "Bob Smith" ;
    foaf:mbox       <mailto:bob@example.org> .

inst:User2 foaf:knows inst:User1 .
```

Result Shape Map

Node	Shape	Result	Reason
inst:User1	my:OpenUserShape	pass	
inst:User2	my:OpenUserShape	pass	
inst:User1	my:ClosedUserShape	pass	
inst:User2	my:ClosedUserShape	fail	Unexpected rdf:type and foaf:knows arcs.

The triple inst:User2 foaf:knows inst:User1 makes inst:User2 non-conformant with my:ClosedUserShape, but it does not make inst:User1 non-conformant because CLOSED applies only to outgoing arcs.

4.4 Permitting Extra Properties §

If a shape contains a triple constraint with predicate P, the shape is said to "mention" P. By default, for an RDF data node to match that shape, every outgoing arc from that node that uses a mentioned predicate must match a triple constraint in the shape. This is called "closing a property". The following example asserts that every my:UserShape must have two rdf:types: ex:Employee and foaf:Person. The example data fails validation because ex:Manager is not a permitted object.

Schema (ShExC)

```
my:UserShape {
  a [ex:Employee];
  a [foaf:Person]
}
```

Failing Data (Turtle)

```
inst:User4 a foaf:Person, ex:Employee, ex:Manager.
```

The shape my:UserShape can be modified to accept any number of additional arcs with the predicate rdf:type by using the keyword EXTRA followed by the permitted predicate (here: a).

Schema (ShExC)

```
my:UserShape EXTRA a {
  a [ex:Employee];
  a [foaf:Person]
}
```

Passing Data (Turtle)

```
inst:User4 a foaf:Person, ex:Employee, ex:Manager.
```

4.5 Repeated Properties §

The use of generic predicates sometimes leads to their being used multiple times in the same shape. In the following example, we want to make sure that an issue that is ex:accepted or ex:resolved has been reproduced both by a tester and by a programmer.

Schema (ShExC)

```
my:IssueShape {
  ex:state [ex:accepted ex:resolved];
  ex:reproducedBy @my:TesterShape;
  ex:reproducedBy @my:ProgrammerShape
}

my:TesterShape {
  foaf:name xsd:string;
  ex:role [ex:testingRole]
}

my:ProgrammerShape {
  foaf:name xsd:string;
  ex:department [ex:ProgrammingDepartment]
}
```

Passing Data (Turtle)

```
inst:Issue1
  ex:state ex:accepted ;
  ex:reproducedBy inst:Tester2 ;
  ex:reproducedBy inst:Programmer3 .

inst:Tester2
  foaf:name "Uldis" ;
  ex:role ex:testingRole .

inst:Programmer3
  foaf:name "Liga" ;
  ex:department ex:ProgrammingDepartment .
```

Failing Data (Turtle)

```
inst:Issue1
  ex:state ex:accepted ;
  ex:reproducedBy inst:Tester2 ;
  ex:reproducedBy inst:Tester4 .

inst:Tester2
  foaf:name "Uldis" ;
  ex:role ex:testingRole .

inst:Tester4
  foaf:name "Liene" ;
  ex:role ex:testingRole .
```

Note that ShEx uses a partitioning strategy to find a solution whereby triples in the data are assigned to triple constraints in the schema. It is possible to construct schemas for which it is quite expensive to find a mapping from RDF data triple to ShEx triple constraint that satisfies the schema. In practical schemas, this is rarely a concern as the search space is quite small, however, certain mistakes in a schema can create a large search space.

- Accidentally duplicating many triple constraints in a shape causes the search space to explode. If you notice a validation process taking a long time or a lot of memory, look for duplicated chunks of the schema.
- For shapes with multiple triple constraints for the same predicate, try to minimize the overlap between the value expressions. For instance, if three types of inspection are necessary on a manufacturing checklist, use three different constraints for each of the inspection proprerties rather than requiring three different inspection properties with a value expression which is a union of all three types. This will make the validation process more efficient and will more effectively capture the business logic in the schema.

5. Shape Re-use §

In good programming spirit, ShEx provides mechanisms to factor and re-use shapes or schemas.

5.1 Reusing Triple Expressions §

When multiple shapes share common triple expressions, the triple expressions can be defined and labeled just once and referenced multiple times. In the example below, both users and employees are expected to have a foaf:name and foaf:mbox. my:UserShape declares my:entity (with a dollar-sign prefix) and my:EmployeeShape includes it (with an ampersand prefix).

Schema (ShExC)

```
my:UserShape {
  $my:entity (
    foaf:name LITERAL ;
    foaf:mbox IRI+
  ) ;
  ex:userID LITERAL
}

my:EmployeeShape {
  &my:entity ;
  ex:employeeID LITERAL
}
```

Data (Turtle) try it: js | scala

```
inst:Employee2 foaf:name "Bob" ;
  foaf:mbox <mailto:bob@example.com> ;
  ex:employeeID "e02" .

inst:Employee3 ex:employeeID "e03" .

inst:User1 foaf:name "Alice" ;
  foaf:mbox <mailto:alice@example.com> ;
  ex:userID "u01" .
```

Result Shape Map

Node	Shape	Result	Reason
inst:Employee2	my:EmployeeShape	pass	
inst:Employee3	my:EmployeeShape	fail	Expected foaf:name and foaf:mbox arcs.
inst:User1	my:UserShape	pass	

5.2 Importing schemas §

The IMPORT directive allows the importing schema to reference shape and triple expressions in imported schemas. In the schema above, my:EmployeeShape references a triple expression (my:entity) in my:UserShape. If my:UserShape were defined in another schema, the schema containing my:EmployeeShape could import it and still reference my:UserShape and my:entity. In the example below, my:UserShape is defined in a schema referenced by the URL <https://shex.io/examples/import/UserSchema>. Relative URLs are resolved against the base URL of their containing

document, which may be set by a BASE directive. Redunant imports are ignored which means that circular imports are not a problem.

Schema (ShExC)

```
BASE <https://shex.io/examples/import/>
IMPORT <UserSchema>

<EmployeeSchema#EmployeeShape> {
  &<UserSchema#entity> ;
  ex:employeeID LITERAL
}
```

Data (Turtle) try it: js | scala

```
inst:Employee2 foaf:name "Bob" ;
  foaf:mbox <mailto:bob@example.com> ;
  ex:employeeID "e02" .

inst:Employee3 ex:employeeID "e03" .

inst:User1 foaf:name "Alice" ;
  foaf:mbox <mailto:alice@example.com> ;
  ex:userID "u01" .
```

This import above loads the schema below. The &<UserSchema#entity> references the <#entity> triple expression in this schema:

Schema (ShExC)

```
<#UserShape> {
  &<UserSchema#entity> (
    foaf:name LITERAL ;
    foaf:mbox IRI+
  ) ;
  ex:userId LITERAL
}
```

The results are the same as if both documents were combined in one schema:

Result Shape Map

Node	Shape	Result	Reason
inst:Employee2	my:EmployeeShape	pass	
inst:Employee3	my:EmployeeShape	fail	Expected foaf:name and foaf:mbox arcs.
inst:User1	my:UserShape	pass	

6. Relationship of ShEx to RDF and OWL §

ShEx is designed to fill a long-recognized gap in Semantic Web technology. The Resource Description Framework language (RDF), and the more expressive Web Ontology Language (OWL), were designed for making statements about things "in the world" or, more precisely, about things in a conceptual caricature of the world. Things in that caricature may include anything from people, books, abstract ideas, and Web pages to planets or refrigerators. By design, RDF and OWL were optimized for aggregating information from multiple sources and for processing incomplete information. If a model in OWL says that a person has two biological parents, and only one parent is described in a given graph, an OWL processor will not report a mismatch between the model and the graph because the second parent is assumed to exist even if it is not described in the data. In other words, the graph *could* describe the second parent if more triples were supplied. In logic, this is called the "open world assumption".

In contrast, real-world data applications must often test the integrity of their data by flagging such omissions as non-conformant, and the ShEx language was designed for use in making such conformance tests. Where an RDF graph describes things in the caricature of the world, a ShEx schema describes things that are actually "in the data" of RDF data graphs. A shape expression refers to an RDF graph as a collection of abstract-syntactic entities: IRIs, blank nodes, literals, and triples, seen either as incoming arcs or outgoing arcs, with subjects, predicates, and objects. Inasmuch as a ShEx schema is tested

against a given RDF data graph, and does not consider potential but unknown data outside of that graph, the ShEx model of conformance testing follows the "closed world assumption". That said, a ShEx schema can specify a "closed" interpretation, meaning that data can conform only if it includes only triples specified in the schema, or an "open" interpretation, meaning that data triples not specified in the schema are simply ignored.

It should be noted that a ShEx schema is valid as an RDF expression in an open-world sense. A ShEx schema describes something in the world, where that "something" happens to be the set of abstract-syntactic components that comprise an RDF graph. However, such an interpretation is of no use in practical terms. The utility of a ShEx schema derives from its use for testing against the abstract-syntactic components "in the data" of an RDF graph to yield a conformance test result. It should also be noted that a ShEx schema is not an RDF schema, even though both describe RDF data. For historical reasons, "RDF Schema" is the name of a language for defining RDF vocabularies and for specifying semantic relationships between terms that can be used to infer relationships between RDF resources [rdf-schema]. A ShEx schema, in contrast, defines structural constraints, analogously to relational or XML schemas.

A. Namespaces in this Document §

Prefix	Namespace	
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#	
xsd:	http://www.w3.org/2001/XMLSchema#	
Example namespaces		Used in
foaf:	http://xmlns.com/foaf/0.1/	Friend Of A Friend vocabulary
ex:	http://ex.example/ns#	random application ontology
my:	http://my.example/ns#	shape declarations
inst:	http://inst.example/ns#	instance data

B. References §

B.1 Informative references §

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

[rdf-schema]
RDF Schema 1.1. Dan Brickley; Ramanathan Guha. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf-schema/

[rdf11-primer]
RDF 1.1 Primer. Guus Schreiber; Yves Raimond. W3C. 24 June 2014. W3C Note. URL: https://www.w3.org/TR/rdf11-primer/

[shex-semantics]
Shape Expressions Language 2.0. Eric Prud'hommeaux; Iovka Boneva; Jose Labra Gayo; Gregg Kellogg. URL: http://shex.io/shex-semantics/

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

↥