Shapes Constraint Language (SHACL)

W3C Recommendation 20 July 2017

This version:
https://www.w3.org/TR/2017/REC-shacl-20170720/

Latest published version:
https://www.w3.org/TR/shacl/

Latest editor's draft:
https://w3c.github.io/data-shapes/shacl/

Implementation report:
https://w3c.github.io/data-shapes/data-shapes-test-suite/

Previous version:
https://www.w3.org/TR/2017/PR-shacl-20170608/

Editors:
Holger Knublauch, TopQuadrant, Inc.
Dimitris Kontokostas, University of Leipzig

Repository:
GitHub
Issues

Test Suite:
SHACL Test Suite

Please check the **errata** for any errors or issues reported since publication.

See also **translations**.

Abstract

This document defines the SHACL Shapes Constraint Language, a language for validating RDF graphs against a set of conditions. These conditions are provided as shapes and other constructs expressed in the form of an RDF graph. RDF graphs that are used in this manner are called "shapes graphs" in SHACL and the RDF graphs that are validated against a shapes graph are called "data graphs". As SHACL shape graphs are used to validate that data graphs satisfy a set of conditions they can also be viewed as a description of the data graphs that do satisfy these conditions. Such descriptions may be used for a variety of purposes beside validation, including user interface building, code generation and data integration.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document was published by the RDF Data Shapes Working Group as a Recommendation. Comments regarding this document are welcome. Please send them to public-rdf-shapes@w3.org (subscribe, archives).

Please see the Working Group's implementation report.

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

Table of Contents

Document Outline

The introduction includes a Terminology section.

The sections 2 - 4 cover the SHACL Core language and may be read independently from the later sections.

The sections 5 and 6 are about the features that SHACL-SPARQL has in addition to the Core language. These advanced features are SPARQL-based constraints and constraint components.

The syntax of SHACL is RDF. The examples in this document use Turtle [turtle] and (in one instance) JSON-LD [json-ld]. Other RDF serializations such as RDF/XML may be used in practice. The reader should be familiar with basic RDF concepts [rdf11-concepts] such as triples and, for the advanced concepts of SHACL, with SPARQL [sparql11-query].

1. Introduction

This document specifies SHACL (Shapes Constraint Language), a language for describing and validating RDF graphs. This section introduces SHACL with an overview of the key terminology and an example to illustrate basic concepts.

1.1 Terminology

Throughout this document, the following terminology is used.

Terminology that is linked to portions of RDF 1.1 Concepts and Abstract Syntax is used in SHACL as defined there. Terminology that is linked to portions of SPARQL 1.1 Query Language is used in SHACL as defined there. A single linkage

is sufficient to provide a definition for all occurences of a particular term in this document.

Definitions are complete within this document, i.e., if there is no rule to make some situation true in this document then the situation is false.

Basic RDF Terminology
This document uses the terms ***RDF graph***, ***RDF triple***, ***IRI***, ***literal***, ***blank node***, ***node*** of an RDF graph, ***RDF term***, and ***subject***, ***predicate***, and ***object*** of RDF triples, and ***datatype*** as defined in RDF 1.1 Concepts and Abstract Syntax [rdf11-concepts]. ***Language tags*** are defined as in [BCP47].

Property Value and Path
A ***property*** is an IRI. An RDF term `n` has a ***value*** `v` for property `p` in an RDF graph if there is an RDF triple in the graph with subject `n`, predicate `p`, and object `v`. The phrase "Every value of P in graph G ..." means "Every object of a triple in G with predicate P ...". (In this document, the verbs *specify* or *declare* are sometimes used to express the fact that an RDF term has values for a given predicate in a graph.)
SPARQL property paths are defined as in SPARQL 1.1. An RDF term `n` has value `v` for SPARQL property path expression `p` in an RDF graph `G` if there is a solution mapping in the result of the SPARQL query `SELECT ?s ?o WHERE { ?s p' ?o }` on `G` that binds `?s` to `n` and `?o` to `v`, where `p'` is SPARQL surface syntax for `p`.

SHACL Lists
A ***SHACL list*** in an RDF graph `G` is an IRI or a blank node that is either `rdf:nil` (provided that `rdf:nil` has no value for either `rdf:first` or `rdf:rest`), or has exactly one value for the property `rdf:first` in `G` and exactly one value for the property `rdf:rest` in `G` that is also a SHACL list in `G`, and the list does not have itself as a value of the property path `rdf:rest+` in `G`.
The ***members*** of any SHACL list except `rdf:nil` in an RDF graph `G` consist of its value for `rdf:first` in `G` followed by the members in `G` of its value for `rdf:rest` in `G`. The SHACL list `rdf:nil` has no members in any RDF graph.

Binding, Solution
A ***binding*** is a pair (variable, RDF term), consistent with the term's use in SPARQL. A ***solution*** is a set of bindings, informally often understood as one row in the body of the result table of a SPARQL query. Variables are not required to be bound in a solution.

SHACL Subclass, SHACL superclass
A node `Sub` in an RDF graph is a ***SHACL subclass*** of another node `Super` in the graph if there is a sequence of triples in the graph each with predicate `rdfs:subClassOf` such that the subject of the first triple is `Sub`, the object of the last triple is `Super`, and the object of each triple except the last is the subject of the next. If `Sub` is a SHACL subclass of `Super` in an RDF graph then `Super` is a ***SHACL superclass*** of `Sub` in the graph.

SHACL Type
The ***SHACL types*** of an RDF term in an RDF graph is the set of its values for `rdf:type` in the graph as well as the SHACL superclasses of these values in the graph.

SHACL Class
Nodes in an RDF graph that are subclasses, superclasses, or types of nodes in the graph are referred to as ***SHACL class***.

SHACL Class Instance
A node `n` in an RDF graph `G` is a ***SHACL instance*** of a SHACL class `C` in `G` if one of the SHACL types of `n` in `G` is `C`.

SHACL Core and SHACL-SPARQL
The SHACL specification is divided into SHACL Core and SHACL-SPARQL. ***SHACL Core*** consists of frequently needed features for the representation of shapes, constraints and targets. All SHACL implementations must at least

implement SHACL Core. ***SHACL-SPARQL*** consists of all features of SHACL Core plus the advanced features of SPARQL-based constraints and an extension mechanism to declare new constraint components.

1.2 Document Conventions

Within this document, the following namespace prefix bindings are used:

Prefix	Namespace
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
sh:	http://www.w3.org/ns/shacl#
xsd:	http://www.w3.org/2001/XMLSchema#
ex:	http://example.com/ns#

Note that the URI of the graph defining the SHACL vocabulary itself is equivalent to the namespace above, i.e. it includes the `#`. References to the SHACL vocabulary, e.g. via `owl:imports` should include the `#`.

Throughout the document, color-coded boxes containing RDF graphs in Turtle will appear. These fragments of Turtle documents use the prefix bindings given above.

Example shapes graph

```
# This box represents an input shapes graph

# Triples that can be omitted are marked as grey e.g.
<s> <p> <o> .
```

Example data graph

```
# This box represents an input data graph.
# When highlighting is used in the examples:

# Elements highlighted in blue are focus nodes
ex:Bob a ex:Person .

# Elements highlighted in red are focus nodes that fail validation
ex:Alice a ex:Person .
```

Example validation results

```
# This box represents an output results graph
```

SHACL Definitions appear in blue boxes:

SPARQL or TEXTUAL DEFINITIONS

```
# This box contains SPARQL or textual definitions.
```

Grey boxes such as this include syntax rules that apply to the shapes graph.

`true` denotes the RDF term `"true"^^xsd:boolean`. `false` denotes the RDF term `"false"^^xsd:boolean`.

1.3 Conformance

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

The key words may, must, must not, and should are to be interpreted as described in [RFC2119].

This document defines the **SHACL Core** language, also referred to as just **SHACL**, as described in Part A, and the **SHACL-SPARQL** language that extends SHACL Core with constructs described in Part B. This specification describes conformance criteria for:

- **SHACL Core processors** as processors that support validation with the SHACL Core Language
- **SHACL-SPARQL processors** as processors that support validation with the SHACL-SPARQL Language

This document includes syntactic rules that shapes and other nodes need to fulfill in the shapes graph. These rules are typically of the form *A shape must have...* or *The values of X are literals* or *All objects of triples with predicate P must be IRIs*. The complete list of these rules can be found in the appendix. Nodes that violate any of these rules are called ***ill-formed***. Nodes that violate none of these rules are called ***well-formed***. A shapes graph is ill-formed if it contains at least one ill-formed node.

The remainder of this section is informative.

SHACL Core processors that do not also support SHACL-SPARQL ignore any SHACL-SPARQL constructs such as `sh:sparql` triples.

1.4 SHACL Example

This section is non-normative.

The following example data graph contains three SHACL instances of the class `ex:Person`.

Example data graph

```
ex:Alice
   a ex:Person ;
   ex:ssn "987-65-432A" .

ex:Bob
   a ex:Person ;
   ex:ssn "123-45-6789" ;
   ex:ssn "124-35-6789" .

ex:Calvin
   a ex:Person ;
   ex:birthDate "1971-07-07"^^xsd:date ;
   ex:worksFor ex:UntypedCompany .
```

The following conditions are shown in the example:

- A SHACL instance of `ex:Person` can have at most one value for the property `ex:ssn`, and this value is a literal with the datatype `xsd:string` that matches a specified regular expression.
- A SHACL instance of `ex:Person` can have unlimited values for the property `ex:worksFor`, and these values are IRIs and SHACL instances of `ex:Company`.
- A SHACL instance of `ex:Person` cannot have values for any other property apart from `ex:ssn`, `ex:worksFor` and `rdf:type`.

The aforementioned conditions can be represented as shapes and constraints in the following shapes graph:

Example shapes graph

```
ex:PersonShape
   a sh:NodeShape ;
   sh:targetClass ex:Person ;    # Applies to all persons
   sh:property [                 # _:b1
      sh:path ex:ssn ;           # constrains the values of ex:ssn
      sh:maxCount 1 ;
      sh:datatype xsd:string ;
      sh:pattern "^\\d{3}-\\d{2}-\\d{4}$" ;
   ] ;
   sh:property [                 # _:b2
      sh:path ex:worksFor ;
      sh:class ex:Company ;
      sh:nodeKind sh:IRI ;
   ] ;
   sh:closed true ;
   sh:ignoredProperties ( rdf:type ) .
```

The example below shows the same shape definition as a possible JSON-LD [json-ld] fragment. Note that we have left out a `@context` declaration, and depending on the `@context` the rendering may look quite different. Therefore this example should be understood as an illustration only.

Example shapes graph

```
{
   "@id" : "ex:PersonShape",
   "@type" : "NodeShape",
   "targetClass" : "ex:Person",
   "property" : [
      {
         "path" : "ex:ssn",
         "maxCount" : 1,
         "datatype" : "xsd:string" ,
         "pattern" : "^\\d{3}-\\d{2}-\\d{4}$"
      },
      {
         "path" : "ex:worksFor",
         "class" : "ex:Company",
         "nodeKind" : "sh:IRI"
      }
   ],
   "closed" : true,
   "ignoredProperties" : [ "rdf:type" ]
}
```

We can use the shape declaration above to illustrate some of the key terminology used by SHACL. The target for the shape `ex:PersonShape` is the set of all SHACL instances of the class `ex:Person`. This is specified using the property `sh:targetClass`. During the validation, these target nodes become focus nodes for the shape. The shape `ex:PersonShape` is a node shape, which means that it applies to the focus nodes. It declares constraints on the focus nodes, for example using the parameters `sh:closed` and `sh:ignoredProperties`. The node shape also declares two other constraints with the property `sh:property`, and each of these is backed by a property shape. These property shapes declare additional constraints using parameters such as `sh:datatype` and `sh:maxCount`.

Some of the property shapes specify parameters from multiple constraint components in order to restrict multiple aspects of the property values. For example, in the property shape for `ex:ssn`, parameters from three constraint components are used. The parameters of these constraint components are `sh:datatype`, `sh:pattern` and `sh:maxCount`. For each focus node the property values of `ex:ssn` will be validated against all three components.

SHACL validation based on the provided data graph and shapes graph would produce the following validation report. See the section Validation Report for details on the format.

Example validation results

```
[  a sh:ValidationReport ;
   sh:conforms false ;
   sh:result
   [  a sh:ValidationResult ;
      sh:resultSeverity sh:Violation ;
      sh:focusNode ex:Alice ;
      sh:resultPath ex:ssn ;
      sh:value "987-65-432A" ;
      sh:sourceConstraintComponent sh:RegexConstraintComponent ;
      sh:sourceShape ... blank node _:b1 on ex:ssn above ... ;
   ] ,
   [  a sh:ValidationResult ;
      sh:resultSeverity sh:Violation ;
      sh:focusNode ex:Bob ;
      sh:resultPath ex:ssn ;
      sh:sourceConstraintComponent sh:MaxCountConstraintComponent ;
      sh:sourceShape ... blank node _:b1 on ex:ssn above ... ;
   ] ,
   [  a sh:ValidationResult ;
      sh:resultSeverity sh:Violation ;
      sh:focusNode ex:Calvin ;
      sh:resultPath ex:worksFor ;
      sh:value ex:UntypedCompany ;
      sh:sourceConstraintComponent sh:ClassConstraintComponent ;
      sh:sourceShape ... blank node _:b2 on ex:worksFor above ... ;
   ] ,
   [  a sh:ValidationResult ;
      sh:resultSeverity sh:Violation ;
      sh:focusNode ex:Calvin ;
      sh:resultPath ex:birthDate ;
      sh:value "1971-07-07"^^xsd:date ;
      sh:sourceConstraintComponent sh:ClosedConstraintComponent ;
      sh:sourceShape sh:PersonShape ;
   ]
] .
```

The validation results are enclosed in a validation report. The first validation result is produced because `ex:Alice` has a value for `ex:ssn` that does not match the regular expression specified by the property `sh:regex`. The second validation result is produced because `ex:Bob` has more than the permitted number of values for the property `ex:ssn` as specified by the `sh:maxCount` of 1. The third validation result is produced because `ex:Calvin` has a value for `ex:worksFor` that does not have an `rdf:type` triple that makes it a SHACL instance of `ex:Company`. The forth validation result is produced because the shape `ex:PersonShape` has the property `sh:closed` set to `true` but `ex:Calvin` uses the property `ex:birthDate` which is neither one of the predicates from any of the property shapes of the shape, nor one of the properties listed using `sh:ignoredProperties`.

1.5 Relationship between SHACL and RDFS inferencing

SHACL uses the RDF and RDFS vocabularies, but full RDFS inferencing is not required.

However, SHACL processors may operate on RDF graphs that include entailments [sparql11-entailment] - either pre-computed before being submitted to a SHACL processor or performed on the fly as part of SHACL processing (without modifying either data graph or shapes graph). To support processing of entailments, SHACL includes the property `sh:entailment` to indicate what inferencing is required by a given shapes graph.

> The values of the property `sh:entailment` are IRIs. Common values for this property are covered by [sparql11-entailment].

SHACL implementations may, but are not required to, support entailment regimes. If a shapes graph contains any triple with the predicate `sh:entailment` and object `E` and the SHACL processor does not support `E` as an entailment regime for the given data graph then the processor must signal a failure. Otherwise, the SHACL processor must provide the entailments for

all of the values of `sh:entailment` in the shapes graph, and any inferred triples must be returned by all queries against the data graph during the validation process.

1.6 Relationship between SHACL and SPARQL

This section is non-normative.

For SHACL Core this specification uses parts of SPARQL 1.1 in non-normative alternative definitions of the semantics of constraint components and targets. While these may help some implementers, SPARQL is not required for the implementation of the SHACL Core language.

SHACL-SPARQL is based on SPARQL 1.1 and uses it as a mechanism to declare constraints and constraint components. Implementations that cover only the SHACL Core features are not required to implement these mechanisms.

SPARQL variables using the `$` marker represent external bindings that are pre-bound or, in the case of `$PATH`, substituted in the SPARQL query before execution (as explained in 6.3 Validation with SPARQL-based Constraint Components).

The definition of some constraints requires or is simplified through access to the shapes graph during query execution. SHACL-SPARQL processors may pre-bind the variable `shapesGraph` to provide access to the shapes graph. Access to the shapes graph is not a requirement for supporting the SHACL Core language. The variable `shapesGraph` can also be used in SPARQL-based constraints and SPARQL-based constraint components. However, such constraints may not be interoperable across different SHACL-SPARQL processors or not applicable to remote RDF datasets.

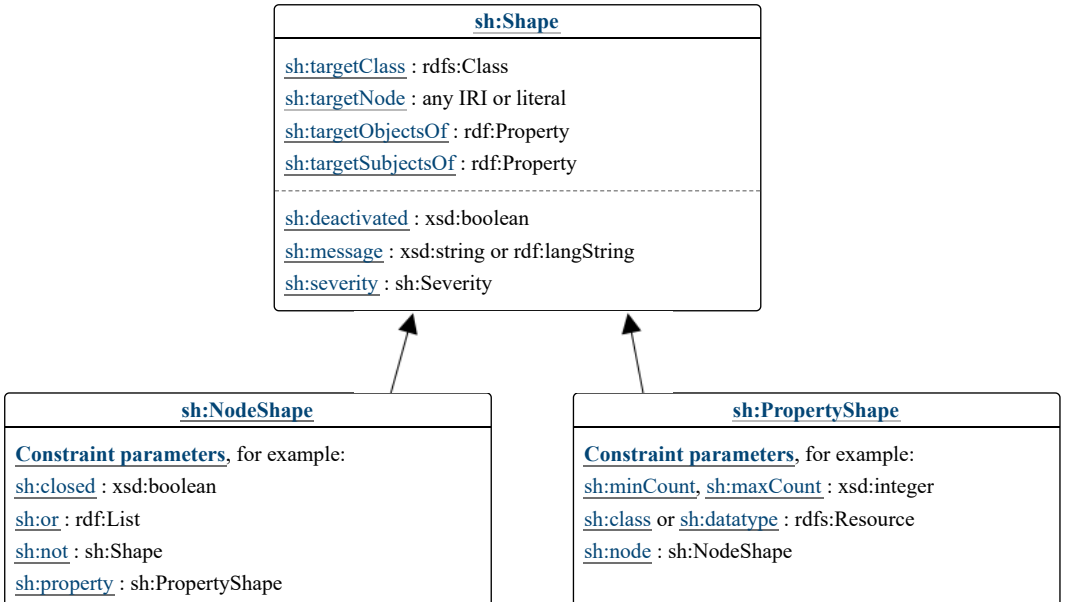
Note that at the time of writing, SPARQL EXISTS has been imperfectly defined and implementations vary. While a W3C Community Group is working on improving this situation, users of SPARQL are advised that the use of EXISTS may have inconsistent results and should be approached with care.

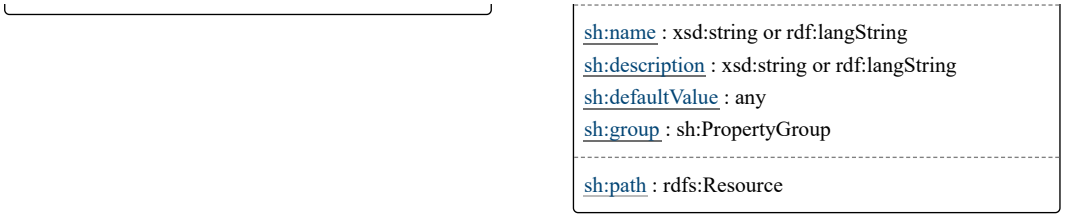
Part 1: SHACL Core

2. Shapes and Constraints

The following introduction is non-normative.

The following informal diagram provides an overview of some of the key classes in the SHACL vocabulary. Each box represents a class. The content of the boxes under the class name lists some of the properties that instances of these classes may have, together with their value types. The arrows indicate `rdfs:subClassOf` triples.





The Turtle serialization of the SHACL vocabulary contains the complete SHACL vocabulary.

2.1 Shapes

> A ***shape*** is an IRI or blank node `s` that fulfills at least one of the following conditions in the shapes graph:
>
> - `s` is a SHACL instance of `sh:NodeShape` or `sh:PropertyShape`.
> - `s` is subject of a triple that has `sh:targetClass`, `sh:targetNode`, `sh:targetObjectsOf` or `sh:targetSubjectsOf` as predicate.
> - `s` is subject of a triple that has a parameter as predicate.
> - `s` is a value of a shape-expecting, non-list-taking parameter such as `sh:node`, or a member of a SHACL list that is a value of a shape-expecting and list-taking parameter such as `sh:or`.

Note that the definition above does not include all of the syntax rules of well-formed shapes. Those are found throughout the document and summarized in Appendix B. Summary of SHACL Syntax Rules. For example, shapes that have literals as values for `sh:targetClass` are ill-formed.

Informally, a shape determines how to validate a focus node based on the values of properties and other characteristics of the focus node. For example, shapes can declare the condition that a focus node be an IRI or that a focus node has a particular value for a property and also a minimum number of values for the property.

The SHACL Core language defines two types of shapes:

- shapes about the focus node itself, called node shapes
- shapes about the values of a particular property or path for the focus node, called property shapes

`sh:Shape` is the SHACL superclass of those two shape types in the SHACL vocabulary. Its subclasses `sh:NodeShape` and `sh:PropertyShape` can be used as SHACL type of node and property shapes, respectively.

2.1.1 Constraints, Parameters and Constraint Components

Shapes can declare constraints using the parameters of constraint components.

A ***constraint component*** is an IRI. Each constraint component has one or more ***mandatory parameters***, each of which is a property. Each constraint component has zero or more ***optional parameters***, each of which is a property. The ***parameters*** of a constraint component are its mandatory parameters plus its optional parameters.

For example, the component `sh:MinCountConstraintComponent` declares the parameter `sh:minCount` to represent the restriction that a node has at least a minimum number of values for a particular property.

For a constraint component `C` with mandatory parameters `p1`, ... `pn`, a shape `s` in a shapes graph `SG` *declares* a ***constraint*** that has ***kind*** `C` with mandatory parameter values `<p1,v1>`, ... `<pn,vn>` in `SG` when `s` has `vi` as a value for `pi` in `SG`. For constraint components with optional parameters, the constraint declaration consists of the values that the shape has for all mandatory and optional parameters of that component.

Some constraint components declare only a single parameter. For example `sh:ClassConstraintComponent` has the single parameter `sh:class`. These parameters may be used multiple times in the same shape, and each value of such a parameter declares an individual constraint. The interpretation of such declarations is conjunction, i.e. all constraints apply. The

following example specifies that the values of `ex:customer` have to be SHACL instances of both `ex:Customer` and `ex:Person`.

Example shapes graph

```
ex:InvoiceShape
   a sh:NodeShape ;
   sh:property [
      sh:path ex:customer ;
      sh:class ex:Customer ;
      sh:class ex:Person ;
   ] .
```

> Some constraint components such as `sh:PatternConstraintComponent` declare more than one parameter. Shapes that have more than one value for any of the parameters of such components are ill-formed.

One way to bypass this syntax rule is to spread the constraints across multiple (property) shapes, as illustrated in the following example.

Example shapes graph

```
ex:MultiplePatternsShape
   a sh:NodeShape ;
   sh:property [
      sh:path ex:name ;
      sh:pattern "^Start" ;
      sh:flags "i" ;
   ] ;
   sh:property [
      sh:path ex:name ;
      sh:pattern "End$" ;
   ] .
```

Constraint components are associated with ***validators***, which provide instructions (for example expressed via SPARQL queries) on how the parameters are used to validate data. Validating an RDF term against a shape involves validating the term against each constraint where the shape has values for all mandatory parameters of the component of the constraint, using the validators associated with the respective component.

The list of constraint components included in SHACL Core is described in section 4. SHACL-SPARQL can be used to declare additional constraint components based on SPARQL.

2.1.2 Focus Nodes

An RDF term that is validated against a shape using the triples from a data graph is called a ***focus node***.

The remainder of this section is informative.

The set of focus nodes for a shape may be identified as follows:

- specified in a shape using target declarations
- specified in any constraint that references a shape in parameters of shape-expecting constraint parameters (e.g. `sh:node`)
- specified as explicit input to the SHACL processor for validating a specific RDF term against a shape

2.1.3 Targets

Target declarations of a shape in a shapes graph are triples with the shape as the subject and certain properties described in this document (e.g., `sh:targetClass`) as predicates. Target declarations can be used to produce focus nodes for a shape.

The ***target*** of a target declaration is the set of RDF terms produced by applying the rules described in the remainder of this section to the data graph. The ***target of a shape*** is the union of all RDF terms produced by the individual targets that are declared by the shape in the shapes graph.

SHACL Core includes the following kinds of targets: node targets, class-based targets (including implicit class-based targets), subjects-of targets, and objects-of targets.

The remainder of this introduction is informative.

RDF terms produced by targets are not required to exist as nodes in the data graph. Targets of a shape are ignored whenever a focus node is provided directly as input to the validation process for that shape. This includes the cases where the shape is a value of one of the shape-expecting constraint parameters (such as `sh:node`) and a focus node is determined during the validation of the corresponding constraint component (such as `sh:NodeConstraintComponent`). In such cases, the provided focus node does not need to be in the target of the shape.

2.1.3.1 Node targets (sh:targetNode)

> A ***node target*** is specified using the `sh:targetNode` predicate. Each value of `sh:targetNode` in a shape is either an IRI or a literal.

> TEXTUAL DEFINITION
>
> If `s` is a shape in a shapes graph `SG` and `s` has value `t` for `sh:targetNode` in `SG` then `{ t }` is a target from any data graph for `s` in `SG`.

The remainder of this section is informative.

With the example data below, only `ex:Alice` is the target of the provided shape:

Example shapes graph

```
ex:PersonShape
   a sh:NodeShape ;
   sh:targetNode ex:Alice .
```

Example data graph

```
ex:Alice a ex:Person .
ex:Bob a ex:Person .
```

The following query expresses a potential definition of node targets in SPARQL. The variable `targetNode` will be pre-bound to the given value of `sh:targetNode`. All bindings of the variable `this` from the solution become focus nodes.

> POTENTIAL DEFINITION IN SPARQL

```
SELECT DISTINCT ?this    # ?this is the focus node
WHERE {
   BIND ($targetNode AS ?this)    # $targetNode is pre-bound to ex:Alice
}
```

2.1.3.2 Class-based Targets (sh:targetClass)

> A ***class target*** is specified with the `sh:targetClass` predicate. Each value of `sh:targetClass` in a shape is an IRI.

> TEXTUAL DEFINITION

> If `s` is a shape in a shapes graph `SG` and `s` has value `c` for `sh:targetClass` in `SG` then the set of SHACL instances of `c` in a data graph `DG` is a target from `DG` for `s` in `SG`.

The remainder of this section is informative.

Example shapes graph

```
ex:PersonShape
   a sh:NodeShape ;
   sh:targetClass ex:Person .
```

Example data graph

```
ex:Alice a ex:Person .
ex:Bob a ex:Person .
ex:NewYork a ex:Place .
```

In this example, only `ex:Alice` and `ex:Bob` are focus nodes. Note that, according to the SHACL instance definition, all the `rdfs:subClassOf` declarations needed to walk the class hierarchy need to exist in the data graph. However, the `ex:Person a rdfs:Class` triple is not required to exist in either graphs.

In the following example, the selected focus node is only `ex:Who`.

Example data graph

```
ex:Doctor rdfs:subClassOf ex:Person .
ex:Who a ex:Doctor .
ex:House a ex:Nephrologist .
```

The following query expresses a potential definition of class targets in SPARQL. The variable `targetClass` will be pre-bound to the given value of `sh:targetClass`. All bindings of the variable `this` from the solutions become focus nodes.

> POTENTIAL DEFINITION IN SPARQL

```
SELECT DISTINCT ?this    # ?this is the focus node
WHERE {
   ?this rdf:type/rdfs:subClassOf* $targetClass .    # $targetClass is pre-bound to ex:Person
}
```

2.1.3.3 Implicit Class Targets

Informally, if a shape is also declared to be a class in the shapes graph then all SHACL instances of this class are a target for the shape.

> If `s` is a SHACL instance of `sh:NodeShape` or `sh:PropertyShape` in an RDF graph `G` and `s` is also a SHACL instance of `rdfs:Class` in `G` and `s` is not an IRI then `s` is an ill-formed shape in `G`.

> TEXTUAL DEFINITION
>
> If `s` is a SHACL instance of `sh:NodeShape` or `sh:PropertyShape` in a shapes graph `SG` and `s` is also a SHACL instance of `rdfs:Class` in `SG` then the set of SHACL instances of `s` in a data graph `DG` is a target from `DG` for `s` in `SG`.

The remainder of this section is informative.

In the following example, `ex:Alice` is a focus node, because it is a SHACL instance of `ex:Person` which is both a class and a shape in the shapes graph.

Example shapes graph

```
ex:Person
   a rdfs:Class, sh:NodeShape .
```

Example data graph

```
ex:Alice a ex:Person .
ex:NewYork a ex:Place .
```

2.1.3.4 Subjects-of targets (sh:targetSubjectsOf)

> A ***subjects-of target*** is specified with the predicate `sh:targetSubjectsOf`. The values of `sh:targetSubjectsOf` in a shape are IRIs.

> TEXTUAL DEFINITION
>
> If `s` is a shape in a shapes graph `SG` and `s` has value `p` for `sh:targetSubjectsOf` in `SG` then the set of nodes in a data graph `DG` that are subjects of triples in `DG` with predicate `p` is a target from `DG` for `s` in `SG`.

The remainder of this section is informative.

Example shapes graph

```
ex:TargetSubjectsOfExampleShape
   a sh:NodeShape ;
   sh:targetSubjectsOf ex:knows .
```

Example data graph

```
ex:Alice ex:knows ex:Bob .
ex:Bob ex:livesIn ex:NewYork .
```

In the example above, only `ex:Alice` is validated against the given shape, because it is the subject of a triple that has `ex:knows` as its predicate.

The following query expresses a potential definition of subjects-of targets in SPARQL. The variable `targetSubjectsOf` will be pre-bound to the given value of `sh:targetSubjectsOf`. All bindings of the variable `this` from the solutions become focus nodes.

> POTENTIAL DEFINITION IN SPARQL

```
SELECT DISTINCT ?this    # ?this is the focus node
WHERE {
   ?this $targetSubjectsOf ?any .    # $targetSubjectsOf is pre-bound to ex:knows
}
```

2.1.3.5 Objects-of targets (sh:targetObjectsOf)

> An ***objects-of target*** is specified with the predicate `sh:targetObjectsOf`. The values of `sh:targetObjectsOf` in a shape are IRIs.

> TEXTUAL DEFINITION
>
> If `s` is a shape in a shapes graph `SG` and `s` has value `p` for `sh:targetObjectsOf` in `SG` then the set of nodes in a data graph `DG` that are objects of triples in `DG` with predicate `p` is a target from `DG` for `s` in `SG`.

The remainder of this section is informative.

Example shapes graph

```
ex:TargetObjectsOfExampleShape
   a sh:NodeShape ;
   sh:targetObjectsOf ex:knows .
```

Example data graph

```
ex:Alice ex:knows ex:Bob .
ex:Bob ex:livesIn ex:NewYork .
```

In the example above, only `ex:Bob` is validated against the given shape, because it is the object of a triple that has `ex:knows` as its predicate.

The following query expresses a potential definition of objects-of targets in SPARQL. The variable `targetObjectsOf` will be pre-bound to the given value of `sh:targetObjectsOf`. All bindings of the variable `this` from the solutions become focus nodes.

> POTENTIAL DEFINITION IN SPARQL
>
> ```
> SELECT DISTINCT ?this    # ?this is the focus node
> WHERE {
>    ?any $targetObjectsOf ?this .    # $targetObjectsOf is pre-bound to ex:knows
> }
> ```

2.1.4 Declaring the Severity of a Shape

> Shapes can specify one value for the property `sh:severity` in the shapes graph. Each value of `sh:severity` in a shape is an IRI.

The values of `sh:severity` are called ***severities***. SHACL includes the three IRIs listed in the table below to represent severities. These are declared in the SHACL vocabulary as SHACL instances of `sh:Severity`.

Severity	Description
`sh:Info`	A non-critical constraint violation indicating an informative message.
`sh:Warning`	A non-critical constraint violation indicating a warning.
`sh:Violation`	A constraint violation.

The remainder of this section is informative.

The specific values of `sh:severity` have no impact on the validation, but may be used by user interface tools to categorize validation results. The values of `sh:severity` are used by SHACL processors to populate the `sh:resultSeverity` field of validation results, see section on severity in validation results. Any IRI can be used as a severity.

For every shape, `sh:Violation` is the default if `sh:severity` is unspecified. The following example illustrates this.

Example shapes graph

```
ex:MyShape
   a sh:NodeShape ;
   sh:targetNode ex:MyInstance ;
   sh:property [    # _:b1
      # Violations of sh:minCount and sh:datatype are produced as warnings
      sh:path ex:myProperty ;
      sh:minCount 1 ;
      sh:datatype xsd:string ;
      sh:severity sh:Warning ;
   ] ;
```

```
   sh:property [    # _:b2
      # The default severity here is sh:Violation
      sh:path ex:myProperty ;
      sh:maxLength 10 ;
      sh:message "Too many characters"@en ;
      sh:message "Zu viele Zeichen"@de ;
   ] .
```

Example data graph

```
ex:MyInstance
   ex:myProperty "http://toomanycharacters"^^xsd:anyURI .
```

Example validation results

```
[  a sh:ValidationReport ;
   sh:conforms false ;
   sh:result
   [  a sh:ValidationResult ;
      sh:resultSeverity sh:Warning ;
      sh:focusNode ex:MyInstance ;
      sh:resultPath ex:myProperty ;
      sh:value "http://toomanycharacters"^^xsd:anyURI ;
      sh:sourceConstraintComponent sh:DatatypeConstraintComponent ;
      sh:sourceShape _:b1 ;
   ] ,
   [  a sh:ValidationResult ;
      sh:resultSeverity sh:Violation ;
      sh:focusNode ex:MyInstance ;
      sh:resultPath ex:myProperty ;
      sh:value "http://toomanycharacters"^^xsd:anyURI ;
      sh:resultMessage "Too many characters"@en ;
      sh:resultMessage "Zu viele Zeichen"@de ;
      sh:sourceConstraintComponent sh:MaxLengthConstraintComponent ;
      sh:sourceShape _:b2 ;
   ]
] .
```

2.1.5 Declaring Messages for a Shape

> Shapes can have values for the property `sh:message`. The values of `sh:message` in a shape are either `xsd:string` literals or literals with a language tag. A shape should not have more than one value for `sh:message` with the same language tag.

If a shape has at least one value for `sh:message` in the shapes graph, then all validation results produced as a result of the shape will have exactly these messages as their value of `sh:resultMessage`, i.e. the values will be copied from the shapes graph into the results graph. (Note that in SHACL-SPARQL, SPARQL-based constraints and SPARQL-based constraint components provide additional means to declare such messages.)

The remainder of this section is informative.

The example from the previous section uses this mechanism to supply the second validation result with two messages. See the section on `sh:resultMessage` in the validation results on further details on how the values of `sh:resultMessage` are populated.

2.1.6 Deactivating a Shape

> Shapes can have at most one value for the property `sh:deactivated`. The value of `sh:deactivated` in a shape must be either `true` or `false`.

A shape that has the value `true` for the property `sh:deactivated` is called ***deactivated***. All RDF terms conform to a deactivated shape.

The remainder of this section is informative.

Use cases of this feature include shape reuse and debugging. In scenarios where shapes from other graphs or files are imported into a given shapes graph, `sh:deactivated` can be set to `true` in the local shapes graph for imported shapes to exclude shapes that do not apply in the current application context. This makes it possible to reuse SHACL graphs developed by others even if you disagree with certain assumptions made by the original authors. If a shape author anticipates that a shape may need to be disabled or modified by others, it is a good practice to use IRIs instead of blank nodes for the actual shapes. For example, a property shape for the property `ex:name` at the shape `ex:PersonShape` may have the IRI `ex:PersonShape-name`. Another typical use case of `sh:deactivated` is during the development and testing of shapes, to (temporarily) disable certain shapes.

The following example illustrates the use of `sh:deactivated` to deactivate a shape. In cases where shapes are imported from other graphs, the `sh:deactivated true` triple would be in the importing graph.

Example shapes graph

```
ex:PersonShape
   a sh:NodeShape ;
   sh:targetClass ex:Person ;
   sh:property ex:PersonShape-name .

ex:PersonShape-name
   a sh:PropertyShape ;
   sh:path ex:name ;
   sh:minCount 1 ;
   sh:deactivated true .
```

With the following data, no constraint violation will be reported even though the instance does not have any value for `ex:name`.

Example data graph

```
ex:JohnDoe a ex:Person .
```

2.2 Node Shapes

> A ***node shape*** is a shape in the shapes graph that is not the subject of a triple with `sh:path` as its predicate. It is recommended, but not required, for a node shape to be declared as a SHACL instance of `sh:NodeShape`. SHACL instances of `sh:NodeShape` cannot have a value for the property `sh:path`.

Informally, node shapes specify constraints that need to be met with respect to focus nodes. In contrast to property shapes they primarily apply to the focus node itself, not to its property values.

2.3 Property Shapes

> A ***property shape*** is a shape in the shapes graph that is the subject of a triple that has `sh:path` as its predicate. A shape has at most one value for `sh:path`. Each value of `sh:path` in a shape must be a well-formed SHACL property path. It is recommended, but not required, for a property shape to be declared as a SHACL instance of `sh:PropertyShape`. SHACL instances of `sh:PropertyShape` have one value for the property `sh:path`.

Informally, property shapes specify constraints that need to be met with respect to nodes that can be reached from the focus node either by directly following a given property (specified as an IRI) or any other SHACL property path, specified using `sh:path`.

Note that the definitions of well-formed property shapes and node shapes make these two sets of nodes disjoint.

The following example illustrates some syntax variations of property shapes.

Example shapes graph

```
ex:ExampleNodeShapeWithPropertyShapes
   a sh:NodeShape ;
   sh:property [
      sh:path ex:email ;
      sh:name "e-mail" ;
      sh:description "We need at least one email value" ;
      sh:minCount 1 ;
   ] ;
   sh:property [
      sh:path (ex:knows ex:email) ;
      sh:name "Friend's e-mail" ;
      sh:description "We need at least one email for everyone you know" ;
      sh:minCount 1 ;
   ] .

ex:ExamplePropertyShape
   a sh:PropertyShape ;
   sh:path ex:email ;
   sh:description "We need at least one email value" ;
   sh:minCount 1 .
```

2.3.1 SHACL Property Paths

SHACL includes RDF terms to represent the following subset of SPARQL property paths: `PredicatePath`, `InversePath`, `SequencePath`, `AlternativePath`, `ZeroOrMorePath`, `OneOrMorePath` and `ZeroOrOnePath`.

The following sub-sections provide syntax rules of well-formed SHACL property paths together with mapping rules to SPARQL 1.1 property paths. These rules define the ***path mapping*** `path(p,G)` in an RDF graph `G` of an RDF term `p` that is a SHACL property path in `G`. Two SHACL property paths are considered ***equivalent paths*** when they map to the exact same SPARQL property paths.

> A node in an RDF graph is a well-formed ***SHACL property path*** `p` if it satisfies exactly one of the syntax rules in the following sub-sections. A node `p` is not a well-formed SHACL property path if `p` is a blank node and any path mappings of `p` directly or transitively reference `p`.

The following example illustrates some valid SHACL property paths, together with their SPARQL 1.1 equivalents.

Example

```
SPARQL Property path: ex:parent
SHACL Property path: ex:parent

SPARQL Property path: ^ex:parent
SHACL Property path: [ sh:inversePath ex:parent ]

SPARQL Property path: ex:parent/ex:firstName
SHACL Property path: ( ex:parent ex:firstName )

SPARQL Property path: rdf:type/rdfs:subClassOf*
SHACL Property path: ( rdf:type [ sh:zeroOrMorePath rdfs:subClassOf ] )
```

```
SPARQL Property path: ex:father|ex:mother
SHACL Property path: [ sh:alternativePath ( ex:father ex:mother  ) ]
```

2.3.1.1 Predicate Paths

> A ***predicate path*** is an IRI.

If `p` is a predicate path then `path(p,G)` is a SPARQL `PredicatePath` with `p` as `iri`.

2.3.1.2 Sequence Paths

> A ***sequence path*** is a blank node that is a SHACL list with at least two members and each member is a well-formed SHACL property path.

If `p` is a sequence path in `G` with list members v_1, v_2, ..., v_n then `path(p,G)` is a SPARQL `SequencePath` of `path(`v_1`,G)` as `elt1`, and the results of the path mapping of the list node of v_2 as `elt2`.

Informal note: the nodes in such a SHACL list should not have values for other properties beside `rdf:first` and `rdf:rest`.

2.3.1.3 Alternative Paths

> An ***alternative path*** is a blank node that is the subject of exactly one triple in `G`. This triple has `sh:alternativePath` as predicate, `L` as object, and `L` is a SHACL list with at least two members and each member of `L` is a well-formed SHACL property path.

If `p` is an alternative path in `G` then, for the members of its SHACL list `L`: v_1, v_2, ..., v_n, `path(p,G)` is a SPARQL `AlternativePath` with `path(`v_1`,G)` as `elt1` followed by an `AlternativePath` for v_2 as `elt2`, ..., up to `path(`v_n`,G)`.

2.3.1.4 Inverse Paths

> An ***inverse path*** is a blank node that is the subject of exactly one triple in `G`. This triple has `sh:inversePath` as predicate, and the object `v` is a well-formed SHACL property path.

If `p` is an inverse path in `G` then `path(p,G)` is a SPARQL `InversePath` with `path(v,G)` as its `elt`.

2.3.1.5 Zero-Or-More Paths

> A ***zero-or-more path*** is a blank node that is the subject of exactly one triple in `G`. This triple has `sh:zeroOrMorePath` as predicate, and the object `v` is a well-formed SHACL property path.

If `p` is a zero-or-more path in `G` then `path(p,G)` is a SPARQL `ZeroOrMorePath` with `path(v,G)` as its `elt`.

2.3.1.6 One-Or-More Paths

> A ***one-or-more path*** is a blank node that is the subject of exactly one triple in `G`. This triple has `sh:oneOrMorePath` as predicate, and the object `v` is a well-formed SHACL property path.

If `p` is a one-or-more path in `G` then `path(p,G)` is a SPARQL `OneOrMorePath` with `path(v,G)` as its `elt`.

2.3.1.7 Zero-Or-One Paths

> A ***zero-or-one path*** is a blank node that is the subject of exactly one triple in `G`. This triple has `sh:zeroOrOnePath` as predicate, and the object `v` is a well-formed SHACL property path.

If `p` is a zero-or-one path in `G` then `path(p,G)` is a SPARQL `ZeroOrOnePath` with `path(v,G)` as its `elt`.

2.3.2 Non-Validating Property Shape Characteristics

This section is non-normative.

While the previous sections introduced properties that represent validation conditions, this section covers properties that are ignored by SHACL processors. The use of these so-called ***non-validating properties*** is entirely optional and not subject to formal interpretation contracts. They may be used for purposes such as form building or predictable printing of RDF files.

2.3.2.1 sh:name and sh:description

Property shapes may have one or more values for `sh:name` to provide human-readable labels for the property in the target where it appears. If present, tools should prefer those locally specified labels over globally specified labels at the `rdf:Property` itself. For example, if a form displays a node that is in the target of a given property shape with an `sh:name`, then the tool should use the provided name. Similarly, property shape may have values for `sh:description` to provide descriptions of the property in the given context. Both `sh:name` and `sh:description` may have multiple values, but should only have one value per language tag.

2.3.2.2 sh:order

Property shapes may have one value for the property `sh:order` to indicate the relative order of the property shape for purposes such as form building. The values of `sh:order` are decimals. `sh:order` is not used for validation purposes and may be used with any type of subjects. If present at property shapes, the recommended use of `sh:order` is to sort the property shapes in an ascending order, for example so that properties with smaller order are placed above (or to the left) of properties with larger order.

2.3.2.3 sh:group

Property shapes may link to an SHACL instance of the class `sh:PropertyGroup` using the property `sh:group` to indicate that the shape belongs to a group of related property shapes. Each group may have additional triples that serve application purposes, such as an `rdfs:label` for form building. Groups may also have an `sh:order` property to indicate the relative ordering of groups within the same form.

2.3.2.4 sh:defaultValue

Property shapes may have a single value for `sh:defaultValue`. The default value does not have fixed semantics in SHACL, but may be used by user interface tools to pre-populate input widgets. The value type of the `sh:defaultValue` should align with the specified `sh:datatype` or `sh:class` of the same shape.

The following example illustrates the use of these various features together.

Example shapes graph

```
ex:PersonFormShape
   a sh:NodeShape ;
   sh:property [
      sh:path ex:firstName ;
      sh:name "first name" ;
```

```
      sh:description "The person's given name(s)" ;
      sh:order 0 ;
      sh:group ex:NameGroup ;
   ] ;
   sh:property [
      sh:path ex:lastName ;
      sh:name "last name" ;
      sh:description "The person's last name" ;
      sh:order 1 ;
      sh:group ex:NameGroup ;
   ] ;
   sh:property [
      sh:path ex:streetAddress ;
      sh:name "street address" ;
      sh:description "The street address including number" ;
      sh:order 11 ;
      sh:group ex:AddressGroup ;
   ] ;
   sh:property [
      sh:path ex:locality ;
      sh:name "locality" ;
      sh:description "The suburb, city or town of the address" ;
      sh:order 12 ;
      sh:group ex:AddressGroup ;
   ] ;
   sh:property [
      sh:path ex:postalCode ;
      sh:name "postal code" ;
      sh:name "zip code"@en-US ;
      sh:description "The postal code of the locality" ;
      sh:order 13 ;
      sh:group ex:AddressGroup ;
   ] .

ex:NameGroup
   a sh:PropertyGroup ;
   sh:order 0 ;
   rdfs:label "Name" .

ex:AddressGroup
   a sh:PropertyGroup ;
   sh:order 1 ;
   rdfs:label "Address" .
```

A form building application may use the information above to display information as follows:

Name

first name: John

last name: Doe

Address

street address: 123 Silverado Ave

locality: Cupertino

zip code: 54321

3. Validation and Graphs

Validation takes a data graph and a shapes graph as input and produces a validation report containing the results of the validation. Conformance checking is a simplified version of validation, producing a boolean result. A system that is capable of performing validation is called a ***processor***, and the verb ***processing*** is sometimes used to refer to the validation process.

SHACL defines an RDF Validation Report Vocabulary that can be used by processors that produce validation reports as RDF results graphs. This specification uses the SHACL results vocabulary for the normative definitions of the validators associated with the constraint components. Only SHACL implementations that can produce all of the mandatory properties of the Validation Report Vocabulary are standards-compliant.

3.1 Shapes Graph

A ***shapes graph*** is an RDF graph containing zero or more shapes that is passed into a SHACL validation process so that a data graph can be validated against the shapes.

The remainder of this section is informative.

Shapes graphs can be reusable validation modules that can be cross-referenced with the predicate `owl:imports`. As a pre-validation step, SHACL processors should extend the originally provided shapes graph by transitively following and importing all referenced shapes graphs through the `owl:imports` predicate. The resulting graph forms the input shapes graph for validation and must not be further modified during the validation process.

In addition to shape declarations, the shapes graph may contain additional information for the SHACL processor such as `sh:entailment` statements.

3.2 Data Graph

Any RDF graph can be a ***data graph***.

The remainder of this section is informative.

A data graph is one of the inputs to the SHACL processor for validation. SHACL processors treat it as a general RDF graph and makes no assumption about its nature. For example, it can be an in-memory graph or a named graph from an RDF dataset or a SPARQL endpoint.

SHACL can be used with RDF graphs that are obtained by any means, e.g. from the file system, HTTP requests, or RDF datasets. SHACL makes no assumptions about whether a graph contains triples that are entailed from the graph under any RDF entailment regime.

The data graph is expected to include all the ontology axioms related to the data and especially all the `rdfs:subClassOf` triples in order for SHACL to correctly identify class targets and validate Core SHACL constraints.

3.3 Linking to shapes graphs (sh:shapesGraph)

> A data graph can include triples used to suggest one or more graphs to a SHACL processor with the predicate `sh:shapesGraph`. Every value of `sh:shapesGraph` is an IRI representing a graph that should be included into the shapes graph used to validate the data graph.

In the following example, a SHACL processor should use the union of `ex:graph-shapes1` and `ex:graph-shapes2` graphs (and their `owl:imports`) as the shapes graph when validating the given graph.

Example data graph

```
<http://example.com/myDataGraph>
   sh:shapesGraph ex:graph-shapes1 ;
   sh:shapesGraph ex:graph-shapes2 .
```

3.4 Validation

Validation is a mapping from some input to validation results, as defined in the following paragraphs.

Validation of a data graph against a shapes graph: Given a data graph and a shapes graph, the validation results are the union of results of the validation of the data graph against all shapes in the shapes graph.

Validation of a data graph against a shape: Given a data graph and a shape in the shapes graph, the validation results are the union of the results of the validation of all focus nodes that are in the target of the shape in the data graph.

Validation of a focus node against a shape: Given a focus node in the data graph and a shape in the shapes graph, the validation results are the union of the results of the validation of the focus node against all constraints declared by the shape, unless the shape has been deactivated, in which case the validation results are empty.

Validation of a focus node against a constraint: Given a focus node in the data graph and a constraint of kind `C` in the shapes graph, the validation results are defined by the validators of the constraint component `C`. These validators typically take as input the focus node, the specific values of the parameters of `C` of the constraint in the shapes graph, and the value nodes of the shape that declares the constraint.

During validation, the data graph and the shapes graph must remain immutable, i.e. both graphs at the end of the validation must be identical to the graph at the beginning of validation. SHACL processors must not change the graphs that they use to construct the shapes graph or the data graph, even if these graphs are part of an RDF store that allows changes to its stored graphs. SHACL processors may store the graphs that they create, such as a graph containing validation results, and this operation may change existing graphs in an RDF store, but not any of the graphs that were used to construct the shapes graph or the data graph. SHACL processing is thus idempotent.

3.4.1 Failures

Validation and conformance checking can result in a ***failure***. For example, a particular SHACL processor might allow recursive shapes but report a failure if it detects a loop within the data. Failures can also be reported due to resource exhaustion. Failures are signalled through implementation-specific channels.

3.4.2 Handling of Ill-formed Shapes Graphs

If the shapes graph contains ill-formed nodes, then the result of the validation process is *undefined*. A SHACL processor should produce a failure in this case. See also 3.6.1.3 Syntax Checking of Shapes Graph (sh:shapesGraphWellFormed).

3.4.3 Handling of Recursive Shapes

The following properties are the so-called ***shape-expecting constraint parameters*** in SHACL Core:

- sh:and
- sh:not
- sh:or
- sh:property
- sh:qualifiedValueShape
- sh:node
- sh:xone

The following properties are the so-called ***list-taking constraint parameters*** in SHACL Core:

- sh:and
- sh:in
- sh:languageIn
- sh:or
- sh:xone

A shape `s1` in an RDF graph `G` ***refers*** to shape `s2` in `G` if it has `s2` as value for some non-list-taking, shape-expecting parameter of some constraint component or `s2` as a member of the value for some list-taking, shape-expecting parameter of some constraint component. A shape in an RDF graph `G` is a ***recursive shape*** in `G` if it is related to itself by the transitive closure of the refers relationship in `G`.

The validation with recursive shapes is not defined in SHACL and is left to SHACL processor implementations. For example, SHACL processors may support recursion scenarios or produce a failure when they detect recursion.

The remainder of this section is informative.

The recursion policy above has been selected to support a large variety of implementation strategies. By leaving recursion undefined, implementations may chose to not support recursion so that they can issue a static set of SPARQL queries (against SPARQL end points) without having to support cycles. The Working Group is aware that other implementations may support recursion and that some shapes graphs may rely on these specific characteristics. The expectation is that future work, for example in W3C Community Groups, will lead to the definition of specific dialects of SHACL where recursion is well-defined.

3.5 Conformance Checking

A focus node ***conforms*** to a shape if and only if the set of result of the validation of the focus node against the shape is empty and no failure has been reported by it.

Conformance checking produces `true` if and only if a given focus node conforms to a given shape, and `false` otherwise.

Note that some constraint components of SHACL Core (e.g., those of `sh:not`, `sh:or` and `sh:node`) rely on conformance checking. In these cases, the validation results used to determine the outcome of conformance checking are separated from those of the surrounding validation process and typically do not end up in the same validation report (except perhaps as values of `sh:detail`).

3.6 Validation Report

The ***validation report*** is the result of the validation process that reports the conformance and the set of all ***validation results***. The validation report is described with the SHACL ***Validation Report Vocabulary*** as defined in this section. This vocabulary defines the RDF properties to represent structural information that may provide guidance on how to identify or fix violations in the data graph.

SHACL-compliant processors must be capable of returning a validation report with all required validation results described in this specification. SHACL-compliant processors may support optional arguments that make it possible to limit the number of returned results. This flexibility is for example needed in some large-scale dataset validation use cases.

The following graph represents an example of a validation report for the validation of a data graph that conforms to a shapes graph.

Example validation results

```
[  a sh:ValidationReport ;
   sh:conforms true ;
] .
```

The following graph represents an example of a validation report for the validation of a data graph that does not conform to a shapes graph. Note that the specific value of `sh:resultMessage` is not mandated by SHACL and considered implementation-specific.

Example validation results

```
[  a sh:ValidationReport ;
   sh:conforms false ;
   sh:result [
      a sh:ValidationResult ;
```

```
      sh:resultSeverity sh:Violation ;
      sh:focusNode ex:Bob ;
      sh:resultPath ex:age ;
      sh:value "twenty two" ;
      sh:resultMessage "ex:age expects a literal of datatype xsd:integer." ;
      sh:sourceConstraintComponent sh:DatatypeConstraintComponent ;
      sh:sourceShape ex:PersonShape-age ;
   ]
] .
```

3.6.1 Validation Report (sh:ValidationReport)

The result of a validation process is an RDF graph with exactly one SHACL instance of `sh:ValidationReport`. The RDF graph may contain additional information such as provenance metadata.

3.6.1.1 Conforms (sh:conforms)

Each SHACL instance of `sh:ValidationReport` in the results graph has exactly one value for the property `sh:conforms` and the value is of datatype `xsd:boolean`. It represents the outcome of the conformance checking. The value of `sh:conforms` is `true` if and only if the validation did not produce any validation results, and `false` otherwise.

3.6.1.2 Result (sh:result)

For every validation result that is produced by a validation process (except those mentioned in the context of conformance checking), the SHACL instance of `sh:ValidationReport` in the results graph has a value for the property `sh:result`. Each value of `sh:result` is a SHACL instance of the class `sh:ValidationResult`.

3.6.1.3 Syntax Checking of Shapes Graph (sh:shapesGraphWellFormed)

SHACL validation engines are not strictly required to check whether the shapes graph is well-formed. Implementations that do perform such checks (e.g., when the shapes graph is installed in the system, or before or during the validation) should use the property `sh:shapesGraphWellFormed` to inform the consumer of the validation report about this fact. If a SHACL instance of `sh:ValidationReport` in the results graph has `true` as the value for `sh:shapesGraphWellFormed` then the processor was certain that the shapes graph that was used for the validation process has passed all SHACL syntax rules (as summarized in B. Summary of SHACL Syntax Rules) during the validation process.

3.6.2 Validation Result (sh:ValidationResult)

SHACL defines `sh:ValidationResult` as a subclass of `sh:AbstractResult` to report individual SHACL validation results. SHACL implementations may use other SHACL subclasses of `sh:AbstractResult`, for example, to report successfully completed constraint checks or accumulated results.

All the properties described in the remaining sub-sections of this section can be specified in a `sh:ValidationResult`. The properties `sh:focusNode`, `sh:resultSeverity` and `sh:sourceConstraintComponent` are the only properties that are mandatory for all validation results.

3.6.2.1 Focus node (sh:focusNode)

Each validation result has exactly one value for the property `sh:focusNode` that is equal to the focus node that has caused the result. This is the focus node that was validated when the validation result was produced.

3.6.2.2 Path (sh:resultPath)

Validation results may have a value for the property `sh:resultPath` pointing at a well-formed SHACL property path. For results produced by a property shape, this SHACL property path is equivalent to the value of `sh:path` of the shape, unless stated otherwise.

3.6.2.3 Value (sh:value)

Validation results may include, as a value of the property `sh:value`, at most one RDF term that has caused the result. The textual definitions of the validators of the SHACL Core components specify how this value is constructed - often they are the value nodes that have violated a constraint.

3.6.2.4 Source (sh:sourceShape)

Validation results may include, as the only value of the property `sh:sourceShape`, the shape that the given `sh:focusNode` was validated against.

3.6.2.5 Constraint Component (sh:sourceConstraintComponent)

Validation results have exactly one value for the property `sh:sourceConstraintComponent` and this value is the IRI of the constraint component that caused the result. For example, results produced due to a violation of a constraint based on a value of `sh:minCount` would have the source constraint component `sh:MinCountConstraintComponent`.

3.6.2.6 Details (sh:detail)

The property `sh:detail` may link a (parent) result with one or more SHACL instances of `sh:AbstractResult` that can provide further details about the cause of the (parent) result. Depending on the capabilities of the SHACL processor, this may for example include violations of constraints that have been evaluated as part of conformance checking via `sh:node`.

3.6.2.7 Message (sh:resultMessage)

Validation results may have values for the property `sh:resultMessage`, for example to communicate additional textual details to humans. While `sh:resultMessage` may have multiple values, there should not be two values with the same language tag. These values are produced by a validation engine based on the values of `sh:message` of the constraints in the shapes graph, see Declaring Messages for a Shape. In cases where a constraint does not have any values for `sh:message` in the shapes graph the SHACL processor may automatically generate other values for `sh:resultMessage`.

3.6.2.8 Severity (sh:resultSeverity)

Each validation result has exactly one value for the property `sh:resultSeverity`, and this value is an IRI. The value is equal to the value of `sh:severity` of the shape in the shapes graph that caused the result, defaulting to `sh:Violation` if no `sh:severity` has been specified for the shape.

3.7 Value Nodes

The validators of most constraint components use the concept of ***value nodes***, which is defined as follows:

- For node shapes the value nodes are the individual focus nodes, forming a set with exactly one member.
- For property shapes with a value for `sh:path p` the value nodes are the set of nodes in the data graph that can be reached from the focus node with the path mapping of `p`. Unless stated otherwise, the value of `sh:resultPath` of each

validation result is a SHACL property path that represents an equivalent path to the one provided in the shape.

4. Core Constraint Components

This section defines the built-in SHACL Core constraint components that must be supported by all SHACL Core processors. The definition of each constraint component contains its IRI as well as a table of its parameters. Unless stated otherwise, all these parameters are mandatory parameters. Shapes that violate any of the syntax rules enumerated in those parameter tables are ill-formed.

Each constraint component also includes a textual definition, which describes the validator associated with the component. These textual definitions refer to the values of the parameters in the constraint by variables of the form `$paramName` where `paramName` is the part of the parameter's IRI after the `sh:` namespace. For example, the textual definition of `sh:ClassConstraintComponent` refers to the value of `sh:class` using the variable `$class`. Note that these validators define the *only* validation results that are being produced by the component. Furthermore, the validators always produce *new* result nodes, i.e. when the textual definition states that "...there is a validation result..." then this refers to a distinct new node in a results graph.

The remainder of this section is informative.

The choice of constraint components that were included into the SHACL Core was made based on the requirements collected by the [shacl-ucr] document. Special attention was paid to the balance between trying to cover as many common use cases as possible and keeping the size of the Core language manageable. Not all use cases can be expressed by the Core language alone. Instead, SHACL-SPARQL provides an extension mechanism, described in the second part of this specification. It is expected that additional reusable libraries of constraint components will be maintained by third parties.

Unless stated otherwise, the Core constraint components can be used both in property shapes and node shapes. Some constraint parameters have syntax rules attached to them that would make node shapes that use these parameters ill-formed. Examples of this include `sh:minCount` which is only supported for property shapes.

The SPARQL definitions in this section represent potential validators. They are included for illustration purposes only and have no formal status otherwise. Many constraint components are written as SPARQL ASK queries. These queries are interpreted against each value node, bound to the variable `value`. If an ASK query does not evaluate to `true` for a value node, then there is a validation result based on the rules outlined in the section on ASK-based validators. Constraint components that are described using a SELECT query are interpreted based on the rules outlined in the section on SELECT-based validators. In particular, for property shapes, the variable `PATH` is substituted with a path expression based on the value of `sh:path` in the shape. All SPARQL queries also require the variable bindings and result variable mapping rules detailed in the section on SPARQL-based Constraints. The variable `this` represents the currently validated focus node. Based on the parameter IRIs on the tables, pre-bound variables are derived using the syntax rules for parameter names.

4.1 Value Type Constraint Components

The constraint components in this section have in common that they can be used to restrict the type of value nodes. Note that it is possible to represent multiple value type alternatives using sh:or.

4.1.1 sh:class

The condition specified by `sh:class` is that each value node is a SHACL instance of a given type.

Constraint Component IRI: `sh:ClassConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:class`	The type of all value nodes. The values of `sh:class` in a shape are IRIs.

> TEXTUAL DEFINITION
>
> For each value node that is either a literal, or a non-literal that is not a SHACL instance of `$class` in the data graph, there is a validation result with the value node as `sh:value`.

The remainder of this section is informative.

Note that multiple values for `sh:class` are interpreted as a conjunction, i.e. the values need to be SHACL instances of all of them.

> POTENTIAL DEFINITION IN SPARQL (Must evaluate to true for each value node $value)

```
ASK {
   $value rdf:type/rdfs:subClassOf* $class .
}
```

Example shapes graph

```
ex:ClassExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:Bob, ex:Alice, ex:Carol ;
   sh:property [
      sh:path ex:address ;
      sh:class ex:PostalAddress ;
   ] .
```

Example data graph

```
ex:Alice a ex:Person .
ex:Bob ex:address [ a ex:PostalAddress ; ex:city ex:Berlin ] .
ex:Carol ex:address [ ex:city ex:Cairo ] .
```

4.1.2 sh:datatype

`sh:datatype` specifies a condition to be satisfied with regards to the datatype of each value node.

Constraint Component IRI: `sh:DatatypeConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:datatype`	The datatype of all value nodes (e.g., `xsd:integer`). The values of `sh:datatype` in a shape are IRIs. A shape has at most one value for `sh:datatype`.

> TEXTUAL DEFINITION
>
> For each value node that is not a literal, or is a literal with a datatype that does not match `$datatype`, there is a validation result with the value node as `sh:value`. The datatype of a literal is determined following the datatype function of SPARQL 1.1. A literal matches a datatype if the literal's datatype has the same IRI and, for the datatypes supported by SPARQL 1.1, is not an ill-typed literal.

The remainder of this section is informative.

The values of `sh:datatype` are typically datatypes, such as `xsd:string`. Note that using `rdf:langString` as value of `sh:datatype` can be used to test if value nodes have a language tag.

Example shapes graph

```
ex:DatatypeExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:Alice, ex:Bob, ex:Carol ;
   sh:property [
      sh:path ex:age ;
      sh:datatype xsd:integer ;
   ] .
```

Example data graph

```
ex:Alice ex:age "23"^^xsd:integer .
ex:Bob ex:age "twenty two" .
ex:Carol ex:age "23"^^xsd:int .
```

4.1.3 sh:nodeKind

`sh:nodeKind` specifies a condition to be satisfied by the RDF node kind of each value node.

Constraint Component IRI: `sh:NodeKindConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:nodeKind`	The node kind (IRI, blank node, literal or combinations of these) of all value nodes. The values of `sh:nodeKind` in a shape are one of the following six instances of the class `sh:NodeKind`: `sh:BlankNode`, `sh:IRI`, `sh:Literal` `sh:BlankNodeOrIRI`, `sh:BlankNodeOrLiteral` and `sh:IRIOrLiteral`. A shape has at most one value for `sh:nodeKind`.

> TEXTUAL DEFINITION
>
> For each value node that does not match `$nodeKind`, there is a validation result with the value node as `sh:value`. Any IRI matches only `sh:IRI`, `sh:BlankNodeOrIRI` and `sh:IRIOrLiteral`. Any blank node matches only `sh:BlankNode`, `sh:BlankNodeOrIRI` and `sh:BlankNodeOrLiteral`. Any literal matches only `sh:Literal`, `sh:BlankNodeOrLiteral` and `sh:IRIOrLiteral`.

The remainder of this section is informative.

> POTENTIAL DEFINITION IN SPARQL (Must evaluate to true for each value node $value)
>
> ```
> ASK {
>    FILTER ((isIRI($value) && $nodeKind IN ( sh:IRI, sh:BlankNodeOrIRI, sh:IRIOrLiteral ) ) ||
>       (isLiteral($value) && $nodeKind IN ( sh:Literal, sh:BlankNodeOrLiteral, sh:IRIOrLiteral )
>       (isBlank($value)   && $nodeKind IN ( sh:BlankNode, sh:BlankNodeOrIRI, sh:BlankNodeOrLitera
> }
> ```

The following example states that all values of `ex:knows` need to be IRIs, at any subject.

Example shapes graph

```
ex:NodeKindExampleShape
   a sh:NodeShape ;
   sh:targetObjectsOf ex:knows ;
   sh:nodeKind sh:IRI .
```

Example data graph

```
ex:Bob ex:knows ex:Alice .
   ex:Alice ex:knows "Bob" .
```

4.2 Cardinality Constraint Components

The following constraint components represent restrictions on the number of value nodes for the given focus node.

4.2.1 sh:minCount

`sh:minCount` specifies the minimum number of value nodes that satisfy the condition. If the minimum cardinality value is 0 then this constraint is always satisfied and so may be omitted.

Constraint Component IRI: `sh:MinCountConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:minCount`	The minimum cardinality. Node shapes cannot have any value for `sh:minCount`. A property shape has at most one value for `sh:minCount`. The values of `sh:minCount` in a property shape are literals with datatype `xsd:integer`.

> TEXTUAL DEFINITION
>
> If the number of value nodes is less than `$minCount`, there is a validation result.

The remainder of this section is informative.

Example shapes graph

```
ex:MinCountExampleShape
   a sh:PropertyShape ;
   sh:targetNode ex:Alice, ex:Bob ;
   sh:path ex:name ;
   sh:minCount 1 .
```

Example data graph

```
ex:Alice ex:name "Alice" .
ex:Bob ex:givenName "Bob"@en .
```

4.2.2 sh:maxCount

`sh:maxCount` specifies the maximum number of value nodes that satisfy the condition.

Constraint Component IRI: `sh:MaxCountConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:maxCount`	The maximum cardinality. Node shapes cannot have any value for `sh:maxCount`. A property shape has at most one value for `sh:maxCount`. The values of `sh:maxCount` in a property shape are literals with datatype `xsd:integer`.

> TEXTUAL DEFINITION
>
> If the number of value nodes is greater than `$maxCount`, there is a validation result.

The remainder of this section is informative.

Example shapes graph

```
ex:MaxCountExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:Bob ;
   sh:property [
      sh:path ex:birthDate ;
      sh:maxCount 1 ;
   ] .
```

Example data graph

```
ex:Bob ex:birthDate "May 5th 1990" .
```

4.3 Value Range Constraint Components

The following constraint components specify value range conditions to be satisfied by value nodes that are comparable via operators such as <, <=, > and >=. The following example illustrates a typical use case of these constraint components.

Example shapes graph

```
ex:NumericRangeExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:Bob, ex:Alice, ex:Ted ;
   sh:property [
      sh:path ex:age ;
      sh:minInclusive 0 ;
      sh:maxInclusive 150 ;
   ] .
```

Example data graph

```
ex:Bob ex:age 23 .
ex:Alice ex:age 220 .
ex:Ted ex:age "twenty one" .
```

4.3.1 sh:minExclusive

Constraint Component IRI: `sh:MinExclusiveConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:minExclusive`	The minimum exclusive value. The values of `sh:minExclusive` in a shape are literals. A shape has at most one value for `sh:minExclusive`.

> TEXTUAL DEFINITION
>
> For each value node `v` where the SPARQL expression `$minExclusive < v` does not return `true`, there is a validation result with `v` as `sh:value`.

The remainder of this section is informative.

The SPARQL expression produces an error if the value node cannot be compared to the specified range, for example when someone compares a string with an integer. If the comparison cannot be performed, then there is a validation result. This is different from, say, a plain SPARQL query, in which such errors would silently not lead to any results.

> POTENTIAL DEFINITION IN SPARQL (Must evaluate to true for each value node $value)

```
ASK {
   FILTER ($minExclusive < $value)
}
```

4.3.2 sh:minInclusive

Constraint Component IRI: `sh:MinInclusiveConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:minInclusive`	The minimum inclusive value. The values of `sh:minInclusive` in a shape are literals. A shape has at most one value for `sh:minInclusive`.

TEXTUAL DEFINITION

For each value node `v` where the SPARQL expression `$minInclusive <= v` does not return `true`, there is a validation result with `v` as `sh:value`.

The remainder of this section is informative.

POTENTIAL DEFINITION IN SPARQL (Must evaluate to true for each value node $value)

```
ASK {
   FILTER ($minInclusive <= $value)
}
```

4.3.3 sh:maxExclusive

Constraint Component IRI: `sh:MaxExclusiveConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:maxExclusive`	The maximum exclusive value. The values of `sh:maxExclusive` in a shape are literals. A shape has at most one value for `sh:maxExclusive`.

TEXTUAL DEFINITION

For each value node `v` where the SPARQL expression `$maxExclusive > v` does not return `true`, there is a validation result with `v` as `sh:value`.

The remainder of this section is informative.

POTENTIAL DEFINITION IN SPARQL (Must evaluate to true for each value node $value)

```
ASK {
   FILTER ($maxExclusive > $value)
}
```

4.3.4 sh:maxInclusive

Constraint Component IRI: `sh:MaxInclusiveConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
sh:maxInclusive	The maximum inclusive value. The values of sh:maxInclusive in a shape are literals. A shape has at most one value for sh:maxInclusive.

> TEXTUAL DEFINITION
>
> For each value node v where the SPARQL expression $maxInclusive >= v does not return true, there is a validation result with v as sh:value.

The remainder of this section is informative.

> POTENTIAL DEFINITION IN SPARQL (Must evaluate to true for each value node $value)
>
> ```
> ASK {
>    FILTER ($maxInclusive >= $value)
> }
> ```

4.4 String-based Constraint Components

The constraint components in this section have in common that they specify conditions on the string representation of value nodes.

4.4.1 sh:minLength

sh:minLength specifies the minimum string length of each value node that satisfies the condition. This can be applied to any literals and IRIs, but not to blank nodes.

Constraint Component IRI: sh:MinLengthConstraintComponent

Parameters:

Property	Summary and Syntax Rules
sh:minLength	The minimum length. The values of sh:minLength in a shape are literals with datatype xsd:integer. A shape has at most one value for sh:minLength.

> TEXTUAL DEFINITION
>
> For each value node v where the length (as defined by the SPARQL STRLEN function) of the string representation of v (as defined by the SPARQL str function) is less than $minLength, or where v is a blank node, there is a validation result with v as sh:value.

The remainder of this section is informative.

Note that if the value of sh:minLength is 0 then there is no restriction on the string length but the constraint is still violated if the value node is a blank node.

> POTENTIAL DEFINITION IN SPARQL (Must evaluate to true for each value node $value)
>
> ```
> ASK {
>    FILTER (STRLEN(str($value)) >= $minLength) .
> }
> ```

4.4.2 sh:maxLength

sh:maxLength specifies the maximum string length of each value node that satisfies the condition. This can be applied to any literals and IRIs, but not to blank nodes.

Constraint Component IRI: `sh:MaxLengthConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:maxLength`	The maximum length. The values of `sh:maxLength` in a shape are literals with datatype `xsd:integer`. A shape has at most one value for `sh:maxLength`.

TEXTUAL DEFINITION

For each value node `v` where the length (as defined by the SPARQL STRLEN function) of the string representation of `v` (as defined by the SPARQL str function) is greater than `$maxLength`, or where `v` is a blank node, there is a validation result with `v` as `sh:value`.

The remainder of this section is informative.

POTENTIAL DEFINITION IN SPARQL (Must evaluate to true for each value node $value)

```
ASK {
   FILTER (STRLEN(str($value)) <= $maxLength) .
}
```

Example shapes graph

```
ex:PasswordExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:Bob, ex:Alice ;
   sh:property [
      sh:path ex:password ;
      sh:minLength 8 ;
      sh:maxLength 10 ;
   ] .
```

Example data graph

```
ex:Bob ex:password "123456789" .
ex:Alice ex:password "1234567890ABC" .
```

4.4.3 sh:pattern

`sh:pattern` specifies a regular expression that each value node matches to satisfy the condition.

Constraint Component IRI: `sh:PatternConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:pattern`	A regular expression that all value nodes need to match. The values of `sh:pattern` in a shape are literals with datatype `xsd:string`. The values of `sh:pattern` in a shape are valid pattern arguments for the SPARQL REGEX function.
`sh:flags`	An optional string of flags, interpreted as in SPARQL 1.1 REGEX. The values of `sh:flags` in a shape are literals with datatype `xsd:string`.

TEXTUAL DEFINITION

For each value node that is a blank node or where the string representation (as defined by the SPARQL str function) does not match the regular expression `$pattern` (as defined by the SPARQL REGEX function), there is a validation

result with the value node as sh:value. If $flags has a value then the matching must follow the definition of the 3-argument variant of the SPARQL REGEX function, using $flags as third argument.

The remainder of this section is informative.

POTENTIAL DEFINITION IN SPARQL (Must evaluate to true for each value node $value)

```
ASK {
   FILTER (!isBlank($value) && IF(bound($flags), regex(str($value), $pattern, $flags), regex(str
}
```

Example shapes graph

```
ex:PatternExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:Bob, ex:Alice, ex:Carol ;
   sh:property [
      sh:path ex:bCode ;
      sh:pattern "^B" ;    # starts with 'B'
      sh:flags "i" ;       # Ignore case
   ] .
```

Example data graph

```
ex:Bob ex:bCode "b101" .
ex:Alice ex:bCode "B102" .
ex:Carol ex:bCode "C103" .
```

4.4.4 sh:languageIn

The condition specified by sh:languageIn is that the allowed language tags for each value node are limited by a given list of language tags.

Constraint Component IRI: sh:LanguageInConstraintComponent

Parameters:

Property	Summary and Syntax Rules
sh:languageIn	A list of basic language ranges as per [BCP47]. Each value of sh:languageIn in a shape is a SHACL list. Each member of such a list is a literal with datatype xsd:string. A shape has at most one value for sh:languageIn.

TEXTUAL DEFINITION

For each value node that is either not a literal or that does not have a language tag matching any of the basic language ranges that are the members of $languageIn following the filtering schema defined by the SPARQL langMatches function, there is a validation result with the value node as sh:value.

The remainder of this section is informative.

The following example shape states that all values of ex:prefLabel can be either in English or Māori.

Example shapes graph

```
ex:NewZealandLanguagesShape
   a sh:NodeShape ;
   sh:targetNode ex:Mountain, ex:Berg ;
   sh:property [
```

```
      sh:path ex:prefLabel ;
      sh:languageIn ( "en" "mi" ) ;
   ] .
```

From the example instances, ex:Berg will lead to constraint violations for all of its labels.

Example data graph

```
ex:Mountain
   ex:prefLabel "Mountain"@en ;
   ex:prefLabel "Hill"@en-NZ ;
   ex:prefLabel "Maunga"@mi .

ex:Berg
   ex:prefLabel "Berg" ;
   ex:prefLabel "Berg"@de ;
   ex:prefLabel ex:BergLabel .
```

4.4.5 sh:uniqueLang

The property sh:uniqueLang can be set to true to specify that no pair of value nodes may use the same language tag.

Constraint Component IRI: sh:UniqueLangConstraintComponent

Parameters:

Property	Summary and Syntax Rules
sh:uniqueLang	true to activate this constraint. The values of sh:uniqueLang in a shape are literals with datatype xsd:boolean. A property shape has at most one value for sh:uniqueLang. Node shapes cannot have any value for sh:uniqueLang.

> TEXTUAL DEFINITION
>
> If $uniqueLang is true then for each non-empty language tag that is used by at least two value nodes, there is a validation result.

The remainder of this section is informative.

Example shapes graph

```
ex:UniqueLangExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:Alice, ex:Bob ;
   sh:property [
      sh:path ex:label ;
      sh:uniqueLang true ;
   ] .
```

Example data graph

```
ex:Alice
   ex:label "Alice" ;
   ex:label "Alice"@en ;
   ex:label "Alice"@fr .

ex:Bob
   ex:label "Bob"@en ;
   ex:label "Bobby"@en .
```

4.5 Property Pair Constraint Components

The constraint components in this section specify conditions on the sets of value nodes in relation to other properties. These constraint components can only be used by property shapes.

4.5.1 sh:equals

`sh:equals` specifies the condition that the set of all value nodes is equal to the set of objects of the triples that have the focus node as subject and the value of `sh:equals` as predicate.

Constraint Component IRI: `sh:EqualsConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:equals`	The property to compare with. The values of `sh:equals` in a shape are IRIs.

> TEXTUAL DEFINITION
>
> For each value node that does not exist as a value of the property `$equals` at the focus node, there is a validation result with the value node as `sh:value`. For each value of the property `$equals` at the focus node that is not one of the value nodes, there is a validation result with the value as `sh:value`.

The remainder of this section is informative.

The following example illustrates the use of `sh:equals` in a shape to specify that certain focus nodes need to have the same set of values for `ex:firstName` and `ex:givenName`.

Example shapes graph

```
ex:EqualExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:Bob ;
   sh:property [
      sh:path ex:firstName ;
      sh:equals ex:givenName ;
   ] .
```

Example data graph

```
ex:Bob
   ex:firstName "Bob" ;
   ex:givenName "Bob" .
```

4.5.2 sh:disjoint

`sh:disjoint` specifies the condition that the set of value nodes is disjoint with the set of objects of the triples that have the focus node as subject and the value of `sh:disjoint` as predicate.

Constraint Component IRI: `sh:DisjointConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:disjoint`	The property to compare the values with. The values of `sh:disjoint` in a shape are IRIs.

> TEXTUAL DEFINITION

> For each value node that also exists as a value of the property `$disjoint` at the focus node, there is a validation result with the value node as `sh:value`.

The remainder of this section is informative.

> POTENTIAL DEFINITION IN SPARQL (Must return no results for the given $PATH)

```
SELECT DISTINCT $this ?value
WHERE {
   $this $PATH ?value .
   $this $disjoint ?value .
}
```

The following example illustrates the use of `sh:disjoint` in a shape to specify that certain focus nodes cannot share any values for `ex:prefLabel` and `ex:altLabel`.

Example shapes graph

```
ex:DisjointExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:USA, ex:Germany ;
   sh:property [
      sh:path ex:prefLabel ;
      sh:disjoint ex:altLabel ;
   ] .
```

Example data graph

```
ex:USA
   ex:prefLabel "USA" ;
   ex:altLabel "United States" .

ex:Germany
   ex:prefLabel "Germany" ;
   ex:altLabel "Germany" .
```

4.5.3 sh:lessThan

`sh:lessThan` specifies the condition that each value node is smaller than all the objects of the triples that have the focus node as subject and the value of `sh:lessThan` as predicate.

Constraint Component IRI: `sh:LessThanConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:lessThan`	The property to compare the values with. The values of `sh:lessThan` in a shape are IRIs. Node shapes cannot have any value for `sh:lessThan`.

> TEXTUAL DEFINITION
>
> For each pair of value nodes and the values of the property `$lessThan` at the given focus node where the first value is not less than the second value (based on SPARQL's `<` operator) or where the two values cannot be compared, there is a validation result with the value node as `sh:value`.

The remainder of this section is informative.

> POTENTIAL DEFINITION IN SPARQL (Must return no results for the given $PATH)

```
SELECT $this ?value
WHERE {
   $this $PATH ?value .
   $this $lessThan ?otherValue .
   BIND (?value < ?otherValue AS ?result) .
   FILTER (!bound(?result) || !(?result)) .
}
```

The following example illustrates the use of `sh:lessThan` in a shape to specify that all values of `ex:startDate` are "before" the values of `ex:endDate`.

Example shapes graph

```
ex:LessThanExampleShape
   a sh:NodeShape ;
   sh:property [
      sh:path ex:startDate ;
      sh:lessThan ex:endDate ;
   ] .
```

4.5.4 sh:lessThanOrEquals

`sh:lessThanOrEquals` specifies the condition that each value node is smaller than or equal to all the objects of the triples that have the focus node as subject and the value of `sh:lessThanOrEquals` as predicate.

Constraint Component IRI: `sh:LessThanOrEqualsConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:lessThanOrEquals`	The property to compare the values with. The values of `sh:lessThanOrEquals` in a shape are IRIs. Node shapes cannot have any value for `sh:lessThanOrEquals`.

TEXTUAL DEFINITION

For each pair of value nodes and the values of the property `$lessThanOrEquals` at the given focus node where the first value is not less than or equal to the second value (based on SPARQL's `<=` operator) or where the two values cannot be compared, there is a validation result with the value node as `sh:value`.

The remainder of this section is informative.

POTENTIAL DEFINITION IN SPARQL (Must return no results for the given $PATH)

```
SELECT $this ?value
WHERE {
   $this $PATH ?value .
   $this $lessThan ?otherValue .
   BIND (?value <= ?otherValue AS ?result) .
   FILTER (!bound(?result) || !(?result)) .
}
```

4.6 Logical Constraint Components

The constraint components in this section implement the common logical operators *and*, *or* and *not*, as well as a variation of *exclusive or*.

4.6.1 sh:not

sh:not specifies the condition that each value node cannot conform to a given shape. This is comparable to negation and the logical "not" operator.

Constraint Component IRI: sh:NotConstraintComponent

Parameters:

Property	Summary and Syntax Rules
sh:not	The shape to negate. The values of sh:not in a shape must be well-formed shapes.

> TEXTUAL DEFINITION
>
> For each value node v: A failure must be reported if the conformance checking of v against the shape $not produces a failure. Otherwise, if v conforms to the shape $not, there is validation result with v as sh:value.

The remainder of this section is informative.

The following example illustrates the use of sh:not in a shape to specify the condition that certain focus nodes cannot have any value of ex:property.

Example shapes graph

```
ex:NotExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:InvalidInstance1 ;
   sh:not [
      a sh:PropertyShape ;
      sh:path ex:property ;
      sh:minCount 1 ;
   ] .
```

Example data graph

```
ex:InvalidInstance1 ex:property "Some value" .
```

4.6.2 sh:and

sh:and specifies the condition that each value node conforms to all provided shapes. This is comparable to conjunction and the logical "and" operator.

Constraint Component IRI: sh:AndConstraintComponent

Parameters:

Property	Summary and Syntax Rules
sh:and	A SHACL list of shapes to validate the value nodes against. Each value of sh:and in a shape is a SHACL list. Each member of such list must be a well-formed shape.

> TEXTUAL DEFINITION
>
> For each value node v: A failure must be produced if the conformance checking of v against any of the members of $and produces a failure. Otherwise, if v does not conform to each member of $and, there is a validation result with v as sh:value.

The remainder of this section is informative.

Note that although sh:and has a SHACL list of shapes as its value, the order of those shapes does not impact the validation results.

The following example illustrates the use of `sh:and` in a shape to specify the condition that certain focus nodes have exactly one value of `ex:property`. This is achieved via the conjunction of a separate named shape (`ex:SuperShape`) which specifies the minimum count, and a blank node shape that additionally specifies the maximum count. As shown here, `sh:and` can be used to implement a specialization mechanism between shapes.

Example shapes graph

```
ex:SuperShape
   a sh:NodeShape ;
   sh:property [
      sh:path ex:property ;
      sh:minCount 1 ;
   ] .

ex:ExampleAndShape
   a sh:NodeShape ;
   sh:targetNode ex:ValidInstance, ex:InvalidInstance ;
   sh:and (
      ex:SuperShape
      [
         sh:path ex:property ;
         sh:maxCount 1 ;
      ]
   ) .
```

Example data graph

```
ex:ValidInstance
   ex:property "One" .

# Invalid: more than one property
ex:InvalidInstance
   ex:property "One" ;
   ex:property "Two" .
```

4.6.3 sh:or

`sh:or` specifies the condition that each value node conforms to at least one of the provided shapes. This is comparable to disjunction and the logical "or" operator.

Constraint Component IRI: `sh:OrConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:or`	A SHACL list of shapes to validate the value nodes against. Each value of `sh:or` in a shape is a SHACL list. Each member of such list must be a well-formed shape.

> TEXTUAL DEFINITION
>
> For each value node `v`: A failure must be produced if the conformance checking of `v` against any of the members produces a failure. Otherwise, if `v` conforms to none of the members of `$or` there is a validation result with `v` as `sh:value`.

The remainder of this section is informative.

Note that although `sh:or` has a SHACL list of shapes as its value, the order of those shapes does not impact the validation results.

The following example illustrates the use of sh:or in a shape to specify the condition that certain focus nodes have at least one value of ex:firstName or at least one value of ex:givenName.

Example shapes graph

```
ex:OrConstraintExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:Bob ;
   sh:or (
      [
         sh:path ex:firstName ;
         sh:minCount 1 ;
      ]
      [
         sh:path ex:givenName ;
         sh:minCount 1 ;
      ]
   ) .
```

Example data graph

```
ex:Bob ex:firstName "Robert" .
```

The next example shows how sh:or can be used in a property shape to state that the values of the given property ex:address may be either literals with datatype xsd:string or SHACL instances of the class ex:Address.

Example shapes graph

```
ex:PersonAddressShape
   a sh:NodeShape ;
   sh:targetClass ex:Person ;
   sh:property [
      sh:path ex:address ;
      sh:or (
         [
            sh:datatype xsd:string ;
         ]
         [
            sh:class ex:Address ;
         ]
      )
   ] .
```

Example data graph

```
ex:Bob ex:address "123 Prinzengasse, Vaduz, Liechtenstein" .
```

4.6.4 sh:xone

sh:xone specifies the condition that each value node conforms to *exactly one* of the provided shapes.

Constraint Component IRI: sh:XoneConstraintComponent

Parameters:

Property	Summary and Syntax Rules
sh:xone	A SHACL list of shapes to validate the value nodes against. Each value of sh:xone in a shape is a SHACL list. Each member of such list must be a well-formed shape.

> TEXTUAL DEFINITION
>
> For each value node `v` let `N` be the number of the shapes that are members of `$xone` where `v` conforms to the shape. A failure must be produced if the conformance checking of `v` against any of the members produces a failure. Otherwise, if `N` is not exactly `1`, there is a validation result with `v` as `sh:value`.

The remainder of this section is informative.

Note that although `sh:xone` has a SHACL list of shapes as its value, the order of those shapes does not impact the validation results.

The following example illustrates the use of `sh:xone` in a shape to specify the condition that certain focus nodes must either have a value for `ex:fullName` or values for `ex:firstName` and `ex:lastName`, but not both.

Example shapes graph

```
ex:XoneConstraintExampleShape
   a sh:NodeShape ;
   sh:targetClass ex:Person ;
   sh:xone (
      [
         sh:property [
            sh:path ex:fullName ;
            sh:minCount 1 ;
         ]
      ]
      [
         sh:property [
            sh:path ex:firstName ;
            sh:minCount 1 ;
         ] ;
         sh:property [
            sh:path ex:lastName ;
            sh:minCount 1 ;
         ]
      ]
   ) .
```

Example data graph

```
ex:Bob a ex:Person ;
   ex:firstName "Robert" ;
   ex:lastName "Coin" .

ex:Carla a ex:Person ;
   ex:fullName "Carla Miller" .

ex:Dory a ex:Person ;
   ex:firstName "Dory" ;
   ex:lastName "Dunce" ;
   ex:fullName "Dory Dunce" .
```

4.7 Shape-based Constraint Components

The constraint components in this section can be used to specify complex conditions by validating the value nodes against certain shapes.

4.7.1 sh:node

`sh:node` specifies the condition that each value node conforms to the given node shape.

Constraint Component IRI: `sh:NodeConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:node`	The node shape that all value nodes need to conform to. The values of `sh:node` in a shape must be well-formed node shapes.

> TEXTUAL DEFINITION
>
> For each value node `v`: A failure must be produced if the conformance checking of `v` against `$node` produces a failure. Otherwise, if `v` does not conform to `$node`, there is a validation result with `v` as `sh:value`.

The remainder of this section is informative.

In the following example, all values of the property `ex:address` must fulfill the constraints expressed by the shape `ex:AddressShape`.

Example shapes graph

```
ex:AddressShape
   a sh:NodeShape ;
   sh:property [
      sh:path ex:postalCode ;
      sh:datatype xsd:string ;
      sh:maxCount 1 ;
   ] .

ex:PersonShape
   a sh:NodeShape ;
   sh:targetClass ex:Person ;
   sh:property [   # _:b1
      sh:path ex:address ;
      sh:minCount 1 ;
      sh:node ex:AddressShape ;
   ] .
```

Example data graph

```
ex:Bob a ex:Person ;
   ex:address ex:BobsAddress .

ex:BobsAddress
   ex:postalCode "1234" .

ex:Reto a ex:Person ;
   ex:address ex:RetosAddress .

ex:RetosAddress
   ex:postalCode 5678 .
```

Example validation results

```
[  a sh:ValidationReport ;
   sh:conforms false ;
   sh:result [
      a sh:ValidationResult ;
      sh:resultSeverity sh:Violation ;
      sh:focusNode ex:Reto ;
      sh:resultPath ex:address ;
      sh:value ex:RetosAddress ;
      sh:resultMessage "Value does not conform to shape ex:AddressShape." ;
```

```
      sh:sourceConstraintComponent sh:NodeConstraintComponent ;
      sh:sourceShape _:b1 ;
   ]
] .
```

4.7.2 sh:property

`sh:property` can be used to specify that each value node has a given property shape.

Constraint Component IRI: `sh:PropertyShapeComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:property`	A property shape that all value nodes need to have. Each value of `sh:property` in a shape must be a well-formed property shape.

> TEXTUAL DEFINITION
>
> For each value node `v`: A failure must be produced if the validation of `v` as focus node against the property shape `$property` produces a failure. Otherwise, the validation results are the results of validating `v` as focus node against the property shape `$property`.

The remainder of this section is informative.

Note that there is an important difference between `sh:property` and `sh:node`: If a value node is violating the constraint, then there is only a single validation result for `sh:node` for this value node, with `sh:NodeConstraintComponent` as its `sh:sourceConstraintComponent`. On the other hand side, there may be any number of validation results for `sh:property`, and these will have the individual constraint components of the constraints in the property shape as their values of `sh:sourceConstraintComponent`.

Like with all other validation results, each time a property shape is reached via `sh:property`, a validation engine must produce *fresh* validation result nodes. This includes cases where the same focus node is validated against the same property shape although it is reached via different paths in the shapes graph.

4.7.3 sh:qualifiedValueShape, sh:qualifiedMinCount, sh:qualifiedMaxCount

`sh:qualifiedValueShape` specifies the condition that a specified number of value nodes conforms to the given shape. Each `sh:qualifiedValueShape` can have: one value for `sh:qualifiedMinCount`, one value for `sh:qualifiedMaxCount` or, one value for each, at the same subject.

Parameters:

Property	Summary and Syntax Rules
`sh:qualifiedValueShape`	The shape that the specified number of value nodes needs to conform to. The values of `sh:qualifiedValueShape` in a shape must be well-formed shapes. Node shapes cannot have any value for `sh:qualifiedValueShape`. This is a mandatory parameter of `sh:QualifiedMinCountConstraintComponent` and `sh:QualifiedMaxCountConstraintComponent`.
`sh:qualifiedValueShapesDisjoint`	This is an optional parameter of `sh:QualifiedMinCountConstraintComponent` and `sh:QualifiedMaxCountConstraintComponent`. If set to `true` then (for the counting) the value nodes must not conform to any of the sibling shapes. The values of `sh:qualifiedValueShapesDisjoint` in a shape are literals with datatype `xsd:boolean`.

sh:qualifiedMinCount	The minimum number of value nodes that conform to the shape. The values of sh:qualifiedMinCount in a shape are literals with datatype xsd:integer. This is a mandatory parameter of sh:QualifiedMinCountConstraintComponent.
sh:qualifiedMaxCount	The maximum number of value nodes that can conform to the shape. The values of sh:qualifiedMaxCount in a shape are literals with datatype xsd:integer. This is a mandatory parameter of sh:QualifiedMaxCountConstraintComponent.

TEXTUAL DEFINITION of Sibling Shapes

Let Q be a shape in shapes graph G that declares a qualified cardinality constraint (by having values for sh:qualifiedValueShape and at least one of sh:qualifiedMinCount or sh:qualifiedMaxCount). Let ps be the set of shapes in G that have Q as a value of sh:property. If Q has true as a value for sh:qualifiedValueShapesDisjoint then the set of ***sibling shapes*** for Q is defined as the set of all values of the SPARQL property path sh:property/sh:qualifiedValueShape for any shape in ps minus the value of sh:qualifiedValueShape of Q itself. The set of sibling shapes is empty otherwise.

TEXTUAL DEFINITION of sh:qualifiedMinCount

Let C be the number of value nodes v where v conforms to $qualifiedValueShape and where v does not conform to any of the sibling shapes for the *current* shape, i.e. the shape that v is validated against and which has $qualifiedValueShape as its value for sh:qualifiedValueShape. A failure must be produced if any of the said conformance checks produces a failure. Otherwise, there is a validation result if C is less than $qualifiedMinCount. The constraint component for sh:qualifiedMinCount is sh:QualifiedMinCountConstraintComponent.

TEXTUAL DEFINITION of sh:qualifiedMaxCount

Let C be as defined for sh:qualifiedMinCount above. A failure must be produced if any of the said conformance checks produces a failure. Otherwise, there is a validation result if C is greater than $qualifiedMaxCount. The constraint component for sh:qualifiedMaxCount is sh:QualifiedMaxCountConstraintComponent.

The remainder of this section is informative.

In the following example shape can be used to specify the condition that the property ex:parent has exactly two values, and at least one of them is female.

Example shapes graph

```
ex:QualifiedValueShapeExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:QualifiedValueShapeExampleValidResource ;
   sh:property [
      sh:path ex:parent ;
      sh:minCount 2 ;
      sh:maxCount 2 ;
      sh:qualifiedValueShape [
         sh:path ex:gender ;
         sh:hasValue ex:female ;
      ] ;
      sh:qualifiedMinCount 1 ;
   ] .
```

Example data graph

```
ex:QualifiedValueShapeExampleValidResource
   ex:parent ex:John ;
   ex:parent ex:Jane .
```

```
ex:John
   ex:gender ex:male .

ex:Jane
   ex:gender ex:female .
```

The following example illustrates the use of `sh:qualifiedValueShapesDisjoint` to express that a hand must have at most 5 values of `ex:property` (expressed using `sh:maxCount`), and exactly one of them must be an instance of `ex:Thumb` while exactly 4 of them must be an instance of `ex:Finger` but thumbs and fingers must be disjoint. In other words, on a hand, none of the fingers can also be counted as the thumb.

Example shapes graph

```
ex:HandShape
   a sh:NodeShape ;
   sh:targetClass ex:Hand ;
   sh:property [
      sh:path ex:digit ;
      sh:maxCount 5 ;
   ] ;
   sh:property [
      sh:path ex:digit ;
      sh:qualifiedValueShape [ sh:class ex:Thumb ] ;
      sh:qualifiedValueShapesDisjoint true ;
      sh:qualifiedMinCount 1 ;
      sh:qualifiedMaxCount 1 ;
   ] ;
   sh:property [
      sh:path ex:digit ;
      sh:qualifiedValueShape [ sh:class ex:Finger ] ;
      sh:qualifiedValueShapesDisjoint true ;
      sh:qualifiedMinCount 4 ;
      sh:qualifiedMaxCount 4 ;
   ] .
```

4.8 Other Constraint Components

This section enumerates Core constraint components that do not fit into the other categories.

4.8.1 sh:closed, sh:ignoredProperties

The RDF data model offers a huge amount of flexibility. Any node can in principle have values for any property. However, in some cases it makes sense to specify conditions on which properties can be applied to nodes. The SHACL Core language includes a property called `sh:closed` that can be used to specify the condition that each value node has values only for those properties that have been explicitly enumerated via the property shapes specified for the shape via `sh:property`.

Constraint Component IRI: `sh:ClosedConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:closed`	Set to `true` to close the shape. The values of `sh:closed` in a shape are literals with datatype `xsd:boolean`.
`sh:ignoredProperties`	Optional SHACL list of properties that are also permitted in addition to those explicitly enumerated via `sh:property`. The values of `sh:ignoredProperties` in a shape must be SHACL lists. Each member of such a list must be a IRI.

> TEXTUAL DEFINITION
>
> If `$closed` is `true` then there is a validation result for each triple that has a value node as its subject and a predicate that is not explicitly enumerated as a value of `sh:path` in any of the property shapes declared via `sh:property` at the current shape. If `$ignoredProperties` has a value then the properties enumerated as members of this SHACL list are also permitted for the value node. The validation result must have the predicate of the triple as its `sh:resultPath`, and the object of the triple as its `sh:value`.

The remainder of this section is informative.

The following example illustrates the use of `sh:closed` in a shape to specify the condition that certain focus nodes only have values for `ex:firstName` and `ex:lastName`. The "ignored" property `rdf:type` would also be allowed.

Example shapes graph

```
ex:ClosedShapeExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:Alice, ex:Bob ;
   sh:closed true ;
   sh:ignoredProperties (rdf:type) ;
   sh:property [
      sh:path ex:firstName ;
   ] ;
   sh:property [
      sh:path ex:lastName ;
   ] .
```

Example data graph

```
ex:Alice
   ex:firstName "Alice" .

ex:Bob
   ex:firstName "Bob" ;
   ex:middleInitial "J" .
```

4.8.2 sh:hasValue

`sh:hasValue` specifies the condition that at least one value node is equal to the given RDF term.

Constraint Component IRI: `sh:HasValueConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:hasValue`	A specific required value.

> TEXTUAL DEFINITION
>
> If the RDF term `$hasValue` is not among the value nodes, there is a validation result.

The remainder of this section is informative.

Example shapes graph

```
ex:StanfordGraduate
   a sh:NodeShape ;
   sh:targetNode ex:Alice ;
   sh:property [
```

```
      sh:path ex:alumniOf ;
      sh:hasValue ex:Stanford ;
   ] .
```

Example data graph

```
ex:Alice
   ex:alumniOf ex:Harvard ;
   ex:alumniOf ex:Stanford .
```

4.8.3 sh:in

`sh:in` specifies the condition that each value node is a member of a provided SHACL list.

Constraint Component IRI: `sh:InConstraintComponent`

Parameters:

Property	Summary and Syntax Rules
`sh:in`	A SHACL list that has the allowed values as members. Each value of `sh:in` in a shape is a SHACL list. A shape has at most one value for `sh:in`.

> TEXTUAL DEFINITION
>
> For each value node that is not a member of `$in`, there is a validation result with the value node as `sh:value`.

The remainder of this section is informative.

Note that matching of literals needs to be exact, e.g. `"04"^^xsd:byte` does not match `"4"^^xsd:integer`.

> POTENTIAL DEFINITION IN SPARQL (Must evaluate to true for each value node $value)
>
> ```
> ASK {
>    GRAPH $shapesGraph {
>       $in (rdf:rest*)/rdf:first $value .
>    }
> }
> ```

Example shapes graph

```
ex:InExampleShape
   a sh:NodeShape ;
   sh:targetNode ex:RainbowPony ;
   sh:property [
      sh:path ex:color ;
      sh:in ( ex:Pink ex:Purple ) ;
   ] .
```

Example data graph

```
ex:RainbowPony ex:color ex:Pink .
```

Part 2: SHACL-SPARQL

Part 1 of this specification introduced features that are built into the Core of SHACL. The goal of this Core is to provide a high-level vocabulary for common use cases to describe shapes. However, SHACL also provides mechanisms to go beyond

the Core vocabulary and represent constraints with greater flexibility. These mechanisms, called SHACL-SPARQL, are described in the following sections.

5. SPARQL-based Constraints

SHACL-SPARQL supports a constraint component that can be used to express restrictions based on a SPARQL SELECT query.

Constraint Component IRI: `sh:SPARQLConstraintComponent`

Parameters:

Property	Summary
`sh:sparql`	A SPARQL-based constraint declaring the SPARQL query to evaluate.

The syntax rules and validation process for SPARQL-based constraints are defined in the rest of this section.

5.1 An Example SPARQL-based Constraint

This section is non-normative.

The following example illustrates the syntax of a SPARQL-based constraint.

Example data graph

```
ex:ValidCountry a ex:Country ;
   ex:germanLabel "Spanien"@de .

ex:InvalidCountry a ex:Country ;
   ex:germanLabel "Spain"@en .
```

Example shapes graph

```
ex:LanguageExampleShape
   a sh:NodeShape ;
   sh:targetClass ex:Country ;
   sh:sparql [
      a sh:SPARQLConstraint ;   # This triple is optional
      sh:message "Values are literals with German language tag." ;
      sh:prefixes ex: ;
      sh:select """
         SELECT $this (ex:germanLabel AS ?path) ?value
         WHERE {
            $this ex:germanLabel ?value .
            FILTER (!isLiteral(?value) || !langMatches(lang(?value), "de"))
         }
         """ ;
   ] .
```

The target of the shape above includes all SHACL instances of `ex:Country`. For those nodes (represented by the variable `this`), the SPARQL query walks through the values of `ex:germanLabel` and verifies that they are literals with a German language code. The validation results for the aforementioned data graph is shown below:

Example validation results

```
[  a sh:ValidationReport ;
   sh:conforms false ;
   sh:result [
      a sh:ValidationResult ;
```

```
      sh:resultSeverity sh:Violation ;
      sh:focusNode ex:InvalidCountry ;
      sh:resultPath ex:germanLabel ;
      sh:value "Spain"@en ;
      sh:sourceConstraintComponent sh:SPARQLConstraintComponent ;
      sh:sourceShape ex:LanguageExampleShape ;
      # ...
   ]
] .
```

The SPARQL query returns result set solutions for all bindings of the variable `value` that violate the constraint. There is a validation result for each solution in that result set, applying the mapping rules explained later. In this example, each validation result will have the binding for the variable `this` as the `sh:focusNode`, `ex:germanLabel` as `sh:resultPath` and the violating value as `sh:value`.

The following example illustrates a similar scenario as above, but with a property shape.

Example shapes graph

```
ex:LanguageExamplePropertyShape
   a sh:PropertyShape ;
   sh:targetClass ex:Country ;
   sh:path ex:germanLabel ;
   sh:sparql [
      a sh:SPARQLConstraint ;   # This triple is optional
      sh:message "Values are literals with German language tag." ;
      sh:prefixes ex: ;
      sh:select """
         SELECT $this ?value
         WHERE {
            $this $PATH ?value .
            FILTER (!isLiteral(?value) || !langMatches(lang(?value), "de"))
         }
         """ ;
   ] .
```

5.2 Syntax of SPARQL-based Constraints

> Shapes may have values for the property `sh:sparql`, and these values are either IRIs or blank nodes. These values are called ***SPARQL-based constraints***.

> SPARQL-based constraints have exactly one value for the property `sh:select`. The value of `sh:select` is a literal of datatype `xsd:string`. The class `sh:SPARQLConstraint` is defined in the SHACL vocabulary and may be used as the type of these constraints (although no type is required). Using the prefix handling rules, the value of `sh:select` is a valid SPARQL 1.1 SELECT query. The SPARQL query derived from the value of `sh:select` projects the variable `this` in the SELECT clause.

The following two properties are similar to their use in shapes:

> SPARQL-based constraints may have values for the property `sh:message` and these are either `xsd:string` literals or literals with a language tag. SPARQL-based constraints may have at most one value for the property `sh:deactivated` and this value is either `true` or `false`.

SELECT queries used in the context of property shapes use a special variable named `PATH` as a placeholder for the path used by the shape.

> The only legal use of the variable `PATH` in the SPARQL queries of SPARQL-based constraints and SELECT-based validators is in the predicate position of a triple pattern. A query that uses the variable `PATH` in any other position is ill-

formed.

5.2.1 Prefix Declarations for SPARQL Queries

A shapes graph may include declarations of namespace prefixes so that these prefixes can be used to abbreviate the SPARQL queries derived from the same shapes graph. The syntax of such prefix declarations is illustrated by the following example.

Example shapes graph

```
ex:
   a owl:Ontology ;
   owl:imports sh: ;
   sh:declare [
      sh:prefix "ex" ;
      sh:namespace "http://example.com/ns#"^^xsd:anyURI ;
   ] ;
   sh:declare [
      sh:prefix "schema" ;
      sh:namespace "http://schema.org/"^^xsd:anyURI ;
   ] .
```

> The values of the property `sh:declare` are IRIs or blank nodes, and these values are called ***prefix declarations***. The SHACL vocabulary includes the class `sh:PrefixDeclaration` as type for such prefix declarations although no `rdf:type` triple is required for them. Prefix declarations have exactly one value for the property `sh:prefix`. The values of `sh:prefix` are literals of datatype `xsd:string`. Prefix declarations have exactly one value for the property `sh:namespace`. The values of `sh:namespace` are literals of datatype `xsd:anyURI`. Such a pair of values specifies a single mapping of a prefix to a namespace.

The recommended subject for values of `sh:declare` is the IRI of the named graph containing the shapes that use the prefixes. These IRIs are often declared as an instance of `owl:Ontology`, but this is not required.

Prefix declarations can be used by SPARQL-based constraints, the validators of SPARQL-based constraint components, and by similar features defined by SHACL extensions. These nodes can use the property `sh:prefixes` to specify a set of prefix mappings. An example use of the `sh:prefixes` property can be found in the example above.

> The values of `sh:prefixes` are either IRIs or blank nodes. A SHACL processor collects a set of prefix mappings as the union of all individual prefix mappings that are values of the SPARQL property path `sh:prefixes/owl:imports*/sh:declare` of the SPARQL-based constraint or validator. If such a collection of prefix declarations contains multiple namespaces for the same value of `sh:prefix`, then the shapes graph is ill-formed. (Note that SHACL processors may ignore prefix declarations that are never reached).

A SHACL processor transforms the values of `sh:select` (and similar properties such as `sh:ask`) into SPARQL by prepending `PREFIX` declarations for all prefix mappings. Each value of `sh:prefix` is turned into the `PNAME_NS`, while each value of `sh:namespace` is turned into the `IRIREF` in the `PREFIX` declaration. For the example shapes graph above, a SHACL-SPARQL processor would produce lines such as `PREFIX ex: <http://example.com/ns#>`. The SHACL-SPARQL processor must produce a failure if the resulting query string cannot be parsed into a valid SPARQL 1.1 query.

In the rest of this document, the `sh:prefixes` statements may have been omitted for brevity.

5.3 Validation with SPARQL-based Constraints

This section explains the validator of `sh:SPARQLConstraintComponent`. Note that this validator only explains one possible implementation strategy, and SHACL processors may choose alternative approaches as long as the outcome is equivalent.

> TEXTUAL DEFINITION

> There are no validation results if the SPARQL-based constraint has `true` as a value for the property `sh:deactivated`. Otherwise, execute the SPARQL query specified by the SPARQL-based constraint `$sparql` pre-binding the variables `this` and, if supported, `shapesGraph` and `currentShape` as described in 5.3.1 Pre-bound Variables in SPARQL Constraints ($this, $shapesGraph, $currentShape). If the shape is a property shape, then prior to execution ***substitute*** the variable `PATH` where it appears in the predicate position of a triple pattern with a valid SPARQL surface syntax string of the SHACL property path specified via `sh:path` at the property shape. There is one validation result for each solution that does not have `true` as the binding for the variable `failure`. These validation results must have the property values explained in 5.3.2 Mapping of Solution Bindings to Result Properties. A failure must be produced if and only if one of the solutions has `true` as the binding for `failure`.

5.3.1 Pre-bound Variables in SPARQL Constraints ($this, $shapesGraph, $currentShape)

When the SPARQL queries of SPARQL-based constraints and the validators of SPARQL-based constraint components are processed, the SHACL-SPARQL processor pre-binds values for the variables in the following table.

Variable	Interpretation
`this`	The focus node.
`shapesGraph` (Optional)	Can be used to query the shapes graph as in `GRAPH $shapesGraph { ... }`. If the shapes graph is a named graph in the same dataset as the data graph then it is the IRI of the shapes graph in the dataset. Not all SHACL-SPARQL processors need to support this variable. Processors that do not support the variable `shapesGraph` must report a failure if they encounter a query that references this variable. Use of `GRAPH $shapesGraph { ... }` should be handled with extreme caution. It may result in constraints that are not interoperable across different SHACL-SPARQL processors and that may not run on remote RDF datasets.
`currentShape` (Optional)	The current shape. Typically used in conjunction with the variable `shapesGraph`. The same support policies as for `shapesGraph` apply for this variable.

5.3.2 Mapping of Solution Bindings to Result Properties

The property values of the validation result nodes are derived by the following rules, through a combination of result solutions and the values of the constraint itself. The rules are meant to be executed from top to bottom, so that the first bound value will be used.

Property	Production Rules
`sh:focusNode`	1. The binding for the variable `this`
`sh:resultPath`	1. The binding for the variable `path`, if that is a IRI 2. For results produced by a property shape, a SHACL property path that is equivalent to the value of `sh:path` of the shape
`sh:value`	1. The binding for the variable `value` 2. The value node
`sh:resultMessage`	1. The binding for the variable `message` 2. For SPARQL-based constraints: The values of `sh:message` of the SPARQL-based constraint. For SPARQL-based constraint components: The values of `sh:message` of the validator of the SPARQL-based constraint component.

	3. For SPARQL-based constraint components: The values of `sh:message` of the SPARQL-based constraint component. These message literals may include the names of any SELECT result variables via `{?varName}` or `{$varName}`. If the constraint is based on a SPARQL-based constraint component, then the component's parameter names can also be used. These `{?varName}` and `{$varName}` blocks should be replaced with suitable string representations of the values of said variables.
`sh:sourceConstraint`	1. The SPARQL-based constraint, i.e. the value of `sh:sparql`

6. SPARQL-based Constraint Components

SPARQL-based constraints provide a lot of flexibility but may be hard to understand for some people or lead to repetition. This section introduces SPARQL-based constraint components as a way to abstract the complexity of SPARQL and to declare high-level reusable components similar to the Core constraint components. Such constraint components can be declared using the SHACL RDF vocabulary and thus shared and reused.

6.1 An Example SPARQL-based Constraint Component

This section is non-normative.

The following example demonstrates how SPARQL can be used to specify new constraint components using the SHACL-SPARQL language. The example implements `sh:pattern` and `sh:flags` using a SPARQL ASK query to validate that each value node matches a given regular expression. Note that this is only an example implementation and should not be considered normative.

Example shapes graph

```
sh:PatternConstraintComponent
   a sh:ConstraintComponent ;
   sh:parameter [
      sh:path sh:pattern ;
   ] ;
   sh:parameter [
      sh:path sh:flags ;
      sh:optional true ;
   ] ;
   sh:validator shimpl:hasPattern .

shimpl:hasPattern
   a sh:SPARQLAskValidator ;
   sh:message "Value does not match pattern {$pattern}" ;
   sh:ask """
      ASK {
         FILTER (!isBlank($value) &&
            IF(bound($flags), regex(str($value), $pattern, $flags), regex(str($value), $pattern)))
      }""" .
```

Constraint components provide instructions to validation engines on how to identify and validate constraints within a shape. In general, if a shape `S` has a value for a property `p`, and there is a constraint component `C` that specifies `p` as a parameter, and `S` has values for all mandatory parameters of `C`, then the set of these parameter values (including the optional parameters) declare a constraint and the validation engine uses a suitable validator from `C` to perform the validation of this constraint. In the example above, `sh:PatternConstraintComponent` declares the mandatory parameter `sh:pattern`, the optional parameter `sh:flags`, and a validator that can be used to perform validation against either node shapes or property shapes.

6.2 Syntax of SPARQL-based Constraint Components

> A ***SPARQL-based constraint component*** is an IRI that has SHACL type `sh:ConstraintComponent` in the shapes graph.

The mechanism to declare new constraint components in this document is limited to those based on SPARQL. However, then general syntax of declaring parameters and validators has been designed to also work for other extension languages such as JavaScript.

6.2.1 Parameter Declarations (sh:parameter)

> The parameters of a constraint component are declared via the property `sh:parameter`. The values of `sh:parameter` are called ***parameter declarations***. The class `sh:Parameter` may be used as type of parameter declarations but no such triple is required. Each parameter declaration has exactly one value for the property `sh:path`. At parameter declarations, the value of `sh:path` is an IRI.

The ***local name*** of an IRI is defined as the longest NCNAME at the end of the IRI, not immediately preceded by the first colon in the IRI. The ***parameter name*** of a parameter declaration is defined as the local name of the value of `sh:path`. To ensure that a correct mapping from parameters into SPARQL variables is possible, the following syntax rules apply:

> Every parameter name is a valid SPARQL VARNAME. Parameter names must not be one of the following: `this`, `shapesGraph`, `currentShape`, `path`, `PATH`, `value`. A constraint component where two or more parameter declarations use the same parameter names is ill-formed.

> The values of `sh:optional` must be literals with datatype `xsd:boolean`. A parameter declaration can have at most one value for the property `sh:optional`. If set to `true` then the parameter declaration declares an optional parameter. Every constraint component has at least one non-optional parameter.

> The class `sh:Parameter` is defined as a SHACL subclass of `sh:PropertyShape`, and all properties that are applicable to property shapes may also be used for parameters. This includes descriptive properties such as `sh:name` and `sh:description` but also constraint parameters such as `sh:class`. Shapes that do not conform with the constraints declared for the parameters are ill-formed. Some implementations may use these constraint parameters to prevent the execution of constraint components with invalid parameter values.

6.2.2 Label Templates (sh:labelTemplate)

> The property `sh:labelTemplate` can be used at any constraint component to suggest how constraints could be rendered to humans. The values of `sh:labelTemplate` are strings (possibly with language tag) and are called ***label templates***.

The remainder of this section is informative.

Label templates can include the names of the parameters that are declared for the constraint component using the syntaxes `{?varName}` or `{$varName}`, where `varName` is the name of the parameter name. At display time, these `{?varName}` and `{$varName}` blocks should be replaced with the actual parameter values. There may be multiple label templates for the same subject, but they should not have the same language tags.

6.2.3 Validators

For every supported shape type (i.e., property shape or node shape) the constraint component declares a suitable validator. For a given constraint, a validator is selected from the constraint component using the following rules, in order:

1. For node shapes, use one of the values of `sh:nodeValidator`, if present.

2. For property shapes, use one of the values of `sh:propertyValidator`, if present.
3. Otherwise, use one of the values of `sh:validator`.

If no suitable validator can be found, a SHACL-SPARQL processor ignores the constraint.

SHACL-SPARQL includes two types of validators, based on SPARQL SELECT (for `sh:nodeValidator` and `sh:propertyValidator`) or SPARQL ASK queries (for `sh:validator`).

6.2.3.1 SELECT-based Validators

> Validators with SHACL type `sh:SPARQLSelectValidator` are called ***SELECT-based validators***. The values of `sh:nodeValidator` must be SELECT-based validators. The values of `sh:propertyValidator` must be SELECT-based validators. SELECT-based validators have exactly one value for the property `sh:select`. The value of `sh:select` is a valid SPARQL SELECT query using the aforementioned prefix handling rules. The SPARQL query derived from the value of `sh:select` projects the variable `this` in its SELECT clause.

The remainder of this section is informative.

The following example illustrates the declaration of a constraint component based on a SPARQL SELECT query. It is a generalized variation of the example from 5.1 An Example SPARQL-based Constraint. That SPARQL query included two constants: the specific property `ex:germanLabel` and the language tag `de`. Constraint components make it possible to generalize such scenarios, so that constants get pre-bound with parameters. This allows the query logic to be reused in multiple places, without having to write any new SPARQL.

Example shapes graph

```
ex:LanguageConstraintComponentUsingSELECT
   a sh:ConstraintComponent ;
   rdfs:label "Language constraint component" ;
   sh:parameter [
      sh:path ex:lang ;
      sh:datatype xsd:string ;
      sh:minLength 2 ;
      sh:name "language" ;
      sh:description "The language tag, e.g. \"de\"." ;
   ] ;
   sh:labelTemplate "Values are literals with language \"{$lang}\"" ;
   sh:propertyValidator [
      a sh:SPARQLSelectValidator ;
      sh:message "Values are literals with language \"{?lang}\"" ;
      sh:select """
         SELECT DISTINCT $this ?value
         WHERE {
            $this $PATH ?value .
            FILTER (!isLiteral(?value) || !langMatches(lang(?value), $lang))
         }
         """
   ] .
```

Once a constraint component has been declared (in a shapes graph), its parameters can be used as illustrated in the following example.

Example shapes graph

```
ex:LanguageExampleShape
   a sh:NodeShape ;
   sh:targetClass ex:Country ;
   sh:property [
      sh:path ex:germanLabel ;
      ex:lang "de" ;
```

```
   ] ;
   sh:property [
      sh:path ex:englishLabel ;
      ex:lang "en" ;
   ] .
```

The example shape above specifies the condition that all values of `ex:germanLabel` carry the language tag `de` while all values of `ex:englishLabel` have `en` as their language. These details are specified via two property shapes that have values for the `ex:lang` parameter required by the constraint component.

6.2.3.2 ASK-based Validators

Many constraint components are of the form in which all value nodes are tested individually against some boolean condition. Writing SELECT queries for these becomes burdensome, especially if a constraint component can be used for both property shapes and node shapes. SHACL-SPARQL provides an alternative, more compact syntax for validators based on ASK queries.

> Validators with SHACL type `sh:SPARQLAskValidator` are called ***ASK-based validators***. The values of `sh:validator` must be ASK-based validators. ASK-based validators have exactly one value for the property `sh:ask`. The value of `sh:ask` must be a literal with datatype `xsd:string`. The value of `sh:ask` must be a valid SPARQL ASK query using the aforementioned prefix handling rules.

The remainder of this section is informative.

The ASK queries return `true` if and only if a given value node (represented by the pre-bound variable `value`) conforms to the constraint.

The following example declares a constraint component using an ASK query.

Example shapes graph

```
ex:LanguageConstraintComponentUsingASK
   a sh:ConstraintComponent ;
   rdfs:label "Language constraint component" ;
   sh:parameter [
      sh:path ex:lang ;
      sh:datatype xsd:string ;
      sh:minLength 2 ;
      sh:name "language" ;
      sh:description "The language tag, e.g. \"de\"." ;
   ] ;
   sh:labelTemplate "Values are literals with language \"{$lang}\"" ;
   sh:validator ex:hasLang .

ex:hasLang
   a sh:SPARQLAskValidator ;
   sh:message "Values are literals with language \"{$lang}\"" ;
   sh:ask """
      ASK {
         FILTER (isLiteral($value) && langMatches(lang($value), $lang))
      }
      """ .
```

Note that the validation condition implemented by an ASK query is "in the inverse direction" from its SELECT counterpart: ASK queries return `true` for value nodes that conform to the constraint, while SELECT queries return those value nodes that do not conform.

6.3 Validation with SPARQL-based Constraint Components

This section defines the validator of SPARQL-based constraint components. Note that this validator only explains one possible implementation strategy, and SHACL processors may choose alternative approaches as long as the outcome is equivalent.

As the first step, a validator must be selected based on the rules outlined in 6.2.3 Validators. Then the following rules apply, producing a set of solutions of SPARQL queries:

- For ASK-based validators: For each value node `v` where the SPARQL ASK query returns `false` with `v` pre-bound to the variable `value`, create one solution consisting of the bindings (`$this`, focus node) and (`$value`, `v`). Let `QS` be a list of these solutions.
- For SELECT-based validators: If the shape is a property shape, then prior to execution substitute the variable `PATH` where it appears in the predicate position of a triple pattern with a valid SPARQL surface syntax string of the SHACL property path specified via `sh:path` at the property shape. Let `QS` be the solutions produced by executing the SPARQL query.

The SPARQL query executions above must pre-bind the variables `this` and, if supported, `shapesGraph` and `currentShape` as described in 5.3.1 Pre-bound Variables in SPARQL Constraints ($this, $shapesGraph, $currentShape). In addition, each value of a parameter of the constraint component in the constraint must be pre-bound as a variable that has the parameter name as its name.

The production rules for the validation results are identical to those for SPARQL-based constraints, using the solutions `QS` as produced above.

Appendix

A. Pre-binding of Variables in SPARQL Queries

Some features of SHACL-SPARQL rely on the concept of pre-binding of variables as defined in this section.

> The definition of pre-binding used by SHACL requires the following restrictions on SPARQL queries. SHACL-SPARQL processors must report a failure when it is operating on a shapes graph that contains SHACL-SPARQL queries (via `sh:select` and `sh:ask`) that violate any of these restrictions. Note that the term *potentially pre-bound variables* includes the variables `this`, `shapesGraph`, `currentShape`, `value` (for ASK queries), and any variables that represent the parameters of the constraint component that uses the query.
>
> - SPARQL queries must not contain a `MINUS` clause
> - SPARQL queries must not contain a federated query (`SERVICE`)
> - SPARQL queries must not contain a `VALUES` clause
> - SPARQL queries must not use the syntax form `AS ?var` for any potentially pre-bound variable
> - Subqueries must return all potentially pre-bound variables, except `shapesGraph` and `currentShape` which are optional as already mentioned in 5.3.1 Pre-bound Variables in SPARQL Constraints ($this, $shapesGraph, $currentShape)

> DEFINITION: ***Values Insertion***
>
> For solution mapping `μ`, define `Table(μ)` to be the multiset formed from `μ`.
>
> ```
> Table(μ) = { μ }
> Card[μ] = 1
> ```
>
> Define the *Values Insertion* function `Replace(X, μ)` to replace each occurence `Y` of a Basic Graph Pattern, Property Path Expression, `Graph(Var, pattern)` in `X` with `join(Y, Table(μ))`.

DEFINITION: ***Pre-binding of variables***

The evaluation of the SPARQL Query `Q = (E, DS, QF)` with *pre-bound* variables `μ` is defined as the evaluation of SPARQL query `Q' = (Replace(E, μ), DS, QF)`.

B. Summary of SHACL Syntax Rules

This section enumerates all normative syntax rules of SHACL. This section is automatically generated from other parts of this spec and hyperlinks are provided back into the prose if the context of the rule in unclear. Nodes that violate these rules in a shapes graph are ill-formed.

Syntax Rule Id	Syntax Rule Text
SHACL-list	A SHACL list in an RDF graph `G` is an IRI or a blank node that is either `rdf:nil` (provided that `rdf:nil` has no value for either `rdf:first` or `rdf:rest`), or has exactly one value for the property `rdf:first` in `G` and exactly one value for the property `rdf:rest` in `G` that is also a SHACL list in `G`, and the list does not have itself as a value of the property path `rdf:rest+` in `G`.
entailment-nodeKind	The values of the property `sh:entailment` are IRIs.
shape	A shape is an IRI or blank node `s` that fulfills at least one of the following conditions in the shapes graph: • `s` is a SHACL instance of `sh:NodeShape` or `sh:PropertyShape`. • `s` is subject of a triple that has `sh:targetClass`, `sh:targetNode`, `sh:targetObjectsOf` or `sh:targetSubjectsOf` as predicate. • `s` is subject of a triple that has a parameter as predicate. • `s` is a value of a shape-expecting, non- list-taking parameter such as `sh:node`, or a member of a SHACL list that is a value of a shape-expecting and list-taking parameter such as `sh:or`.
multiple-parameters	Some constraint components such as `sh:PatternConstraintComponent` declare more than one parameter. Shapes that have more than one value for any of the parameters of such components are ill-formed.
targetNode-nodeKind	Each value of `sh:targetNode` in a shape is either an IRI or a literal.
targetClass-nodeKind	Each value of `sh:targetClass` in a shape is an IRI.
implicit-targetClass-nodeKind	If `s` is a SHACL instance of `sh:NodeShape` or `sh:PropertyShape` in an RDF graph `G` and `s` is also a SHACL instance of `rdfs:Class` in `G` and `s` is not an IRI then `s` is an ill-formed shape in `G`.
targetSubjectsOf-nodeKind	The values of `sh:targetSubjectsOf` in a shape are IRIs.
targetObjectsOf-nodeKind	The values of `sh:targetObjectsOf` in a shape are IRIs.
severity-maxCount	Shapes can specify one value for the property `sh:severity` in the shapes graph.
severity-nodeKind	Each value of `sh:severity` in a shape is an IRI.
message-datatype	The values of `sh:message` in a shape are either `xsd:string` literals or literals with a language tag.
deactivated-maxCount	Shapes can have at most one value for the property `sh:deactivated`.

deactivated-datatype	The value of `sh:deactivated` in a shape must be either `true` or `false`.
NodeShape-path-maxCount	SHACL instances of `sh:NodeShape` cannot have a value for the property `sh:path`.
PropertyShape	A property shape is a shape in the shapes graph that is the subject of a triple that has `sh:path` as its predicate.
path-maxCount	A shape has at most one value for `sh:path`.
path-node	Each value of `sh:path` in a shape must be a well-formed SHACL property path.
PropertyShape-path-minCount	SHACL instances of `sh:PropertyShape` have one value for the property `sh:path`.
path-metarule	A node in an RDF graph is a well-formed SHACL property path `p` if it satisfies exactly one of the syntax rules in the following sub-sections.
path-non-recursive	A node `p` is not a well-formed SHACL property path if `p` is a blank node and any path mappings of `p` directly or transitively reference `p`.
path-sequence	A sequence path is a blank node that is a SHACL list with at least two members and each member is a well-formed SHACL property path.
path-alternative	An alternative path is a blank node that is the subject of exactly one triple in `G`. This triple has `sh:alternativePath` as predicate, `L` as object, and `L` is a SHACL list with at least two members and each member of `L` is a well-formed SHACL property path.
path-inverse	An inverse path is a blank node that is the subject of exactly one triple in `G`. This triple has `sh:inversePath` as predicate, and the object `v` is a well-formed SHACL property path.
path-zero-or-more	A zero-or-more path is a blank node that is the subject of exactly one triple in `G`. This triple has `sh:zeroOrMorePath` as predicate, and the object `v` is a well-formed SHACL property path.
path-one-or-more	A one-or-more path is a blank node that is the subject of exactly one triple in `G`. This triple has `sh:oneOrMorePath` as predicate, and the object `v` is a well-formed SHACL property path.
path-zero-or-one	A zero-or-one path is a blank node that is the subject of exactly one triple in `G`. This triple has `sh:zeroOrOnePath` as predicate, and the object `v` is a well-formed SHACL property path.
shapesGraph-nodeKind	Every value of `sh:shapesGraph` is an IRI
class-nodeKind	The values of `sh:class` in a shape are IRIs.
datatype-nodeKind	The values of `sh:datatype` in a shape are IRIs.
datatype-maxCount	A shape has at most one value for `sh:datatype`.
nodeKind-in	The values of `sh:nodeKind` in a shape are one of the following six instances of the class `sh:NodeKind`: `sh:BlankNode`, `sh:IRI`, `sh:Literal` `sh:BlankNodeOrIRI`, `sh:BlankNodeOrLiteral` and `sh:IRIOrLiteral`.
nodeKind-maxCount	A shape has at most one value for `sh:nodeKind`.
minCount-scope	Node shapes cannot have any value for `sh:minCount`.
minCount-maxCount	A property shape has at most one value for `sh:minCount`.
minCount-datatype	The values of `sh:minCount` in a property shape are literals with datatype `xsd:integer`.

maxCount-scope	Node shapes cannot have any value for sh:maxCount.
maxCount-maxCount	A property shape has at most one value for sh:maxCount.
maxCount-datatype	The values of sh:maxCount in a property shape are literals with datatype xsd:integer.
minExclusive-nodeKind	The values of sh:minExclusive in a shape are literals.
minExclusive-maxCount	A shape has at most one value for sh:minExclusive.
minInclusive-nodeKind	The values of sh:minInclusive in a shape are literals.
minInclusive-maxCount	A shape has at most one value for sh:minInclusive.
maxExclusive-nodeKind	The values of sh:maxExclusive in a shape are literals.
maxExclusive-maxCount	A shape has at most one value for sh:maxExclusive.
maxInclusive-nodeKind	The values of sh:maxInclusive in a shape are literals.
maxInclusive-maxCount	A shape has at most one value for sh:maxInclusive.
minLength-datatype	The values of sh:minLength in a shape are literals with datatype xsd:integer.
minLength-maxCount	A shape has at most one value for sh:minLength.
maxLength-datatype	The values of sh:maxLength in a shape are literals with datatype xsd:integer.
maxLength-maxCount	A shape has at most one value for sh:maxLength.
pattern-datatype	The values of sh:pattern in a shape are literals with datatype xsd:string.
pattern-regex	The values of sh:pattern in a shape are valid pattern arguments for the SPARQL REGEX function.
flags-datatype	The values of sh:flags in a shape are literals with datatype xsd:string.
languageIn-node	Each value of sh:languageIn in a shape is a SHACL list.
languageIn-members-datatype	Each member of such a list is a literal with datatype xsd:string.
languageIn-maxCount	A shape has at most one value for sh:languageIn.
uniqueLang-datatype	The values of sh:uniqueLang in a shape are literals with datatype xsd:boolean.
uniqueLang-maxCount	A property shape has at most one value for sh:uniqueLang.
uniqueLang-scope	Node shapes cannot have any value for sh:uniqueLang.
equals-nodeKind	The values of sh:equals in a shape are IRIs.
disjoint-nodeKind	The values of sh:disjoint in a shape are IRIs.
lessThan-nodeKind	The values of sh:lessThan in a shape are IRIs.
lessThan-scope	Node shapes cannot have any value for sh:lessThan.
lessThanOrEquals-nodeKind	The values of sh:lessThanOrEquals in a shape are IRIs.
lessThanOrEquals-scope	Node shapes cannot have any value for sh:lessThanOrEquals.
not-node	The values of sh:not in a shape must be well-formed shapes.
and-node	Each value of sh:and in a shape is a SHACL list.
and-members-node	Each member of such list must be a well-formed shape.
or-node	Each value of sh:or in a shape is a SHACL list.

or-members-node	Each member of such list must be a well-formed shape.
xone-node	Each value of sh:xone in a shape is a SHACL list.
xone-members-node	Each member of such list must be a well-formed shape.
node-node	The values of sh:node in a shape must be well-formed node shapes.
property-node	Each value of sh:property in a shape must be a well-formed property shape.
qualifiedValueShape-node	The values of sh:qualifiedValueShape in a shape must be well-formed shapes.
qualifiedValueShape-scope	Node shapes cannot have any value for sh:qualifiedValueShape.
qualifiedValueShapesDisjoint-datatype	The values of sh:qualifiedValueShapesDisjoint in a shape are literals with datatype xsd:boolean.
qualifiedMinCount-datatype	The values of sh:qualifiedMinCount in a shape are literals with datatype xsd:integer.
qualifiedMaxCount-datatype	The values of sh:qualifiedMaxCount in a shape are literals with datatype xsd:integer.
closed-datatype	The values of sh:closed in a shape are literals with datatype xsd:boolean.
ignoredProperties-node	The values of sh:ignoredProperties in a shape must be SHACL lists.
ignoredProperties-members-nodeKind	Each member of such a list must be a IRI.
in-node	Each value of sh:in in a shape is a SHACL list.
in-maxCount	A shape has at most one value for sh:in.
sparql-nodeKind	Shapes may have values for the property sh:sparql, and these values are either IRIs or blank nodes.
SPARQLConstraint-select-count	SPARQL-based constraints have exactly one value for the property sh:select
SPARQLConstraint-select-datatype	The value of sh:select is a literal of datatype xsd:string.
select-query-valid	Using the prefix handling rules, the value of sh:select is a valid SPARQL 1.1 SELECT query.
select-query-this	The SPARQL query derived from the value of sh:select projects the variable this in the SELECT clause.
SPARQLConstraint-message-datatype	SPARQL-based constraints may have values for the property sh:message and these are either xsd:string literals or literals with a language tag.
SPARQLConstraint-deactivated-maxCount	SPARQL-based constraints may have at most one value for the property sh:deactivated
PATH-position	The only legal use of the variable PATH in the SPARQL queries of SPARQL-based constraints and SELECT-based validators is in the predicate position of a triple pattern.
declare-nodeKind	The values of the property sh:declare are IRIs or blank nodes
prefix-count	Prefix declarations have exactly one value for the property sh:prefix
prefix-datatype	The values of sh:prefix are literals of datatype xsd:string.
namespace-count	Prefix declarations have exactly one value for the property sh:namespace.
namespace-datatype	The values of sh:namespace are literals of datatype xsd:anyURI.

prefixes-nodeKind	The values of `sh:prefixes` are either IRIs or blank nodes.
prefixes-duplicates	A SHACL processor collects a set of prefix mappings as the union of all individual prefix mappings that are values of the SPARQL property path `sh:prefixes/owl:imports*/sh:declare` of the SPARQL-based constraint or validator. If such a collection of prefix declarations contains multiple namespaces for the same value of `sh:prefix`, then the shapes graph is ill-formed.
ConstraintComponent	A SPARQL-based constraint component is an IRI that has SHACL type `sh:ConstraintComponent` in the shapes graph.
Parameter-predicate-count	Each parameter declaration has exactly one value for the property `sh:path`
Parameter	At parameter declarations, the value of `sh:path` is an IRI.
parameter-name-VARNAME	Every parameter name is a valid SPARQL VARNAME.
parameter-name-not-in	Parameter names must not be one of the following: `this`, `shapesGraph`, `currentShape`, `path`, `PATH`, `value`.
parameter-name-unique	A constraint component where two or more parameter declarations use the same parameter names is ill-formed.
optional-datatype	The values of `sh:optional` must be literals with datatype `xsd:boolean`.
optional-maxCount	A parameter declaration can have at most one value for the property `sh:optional`.
ConstraintComponent-parameter	Every constraint component has at least one non-optional parameter.
Parameter-conformance	Shapes that do not conform with the constraints declared for the parameters are ill-formed.
labelTemplate-datatype	The values of `sh:labelTemplate` are strings (possibly with language tag)
nodeValidator-class	The values of `sh:nodeValidator` must be SELECT-based validators.
propertyValidator-class	The values of `sh:propertyValidator` must be SELECT-based validators.
SPARQLSelectValidator-select-count	SELECT-based validators have exactly one value for the property `sh:select`.
validator-class	The values of `sh:validator` must be ASK-based validators.
ask-count	ASK-based validators have exactly one value for the property `sh:ask`
ask-datatype	The value of `sh:ask` must be a literal with datatype `xsd:string`.
ask-sparql	The value of `sh:ask` must be a valid SPARQL ASK query using the aforementioned prefix handling rules.
pre-binding-limitations	The definition of pre-binding used by SHACL requires the following restrictions on SPARQL queries. SHACL-SPARQL processors must report a failure when it is operating on a shapes graph that contains SHACL-SPARQL queries (via `sh:select` and `sh:ask`) that violate any of these restrictions. Note that the term *potentially pre-bound variables* includes the variables `this`, `shapesGraph`, `currentShape`, `value` (for ASK queries), and any variables that represent the parameters of the constraint component that uses the query. • SPARQL queries must not contain a `MINUS` clause • SPARQL queries must not contain a federated query (`SERVICE`)

	• SPARQL queries must not contain a `VALUES` clause • SPARQL queries must not use the syntax form `AS ?var` for any potentially pre-bound variable • Subqueries must return all potentially pre-bound variables, except `shapesGraph` and `currentShape` which are optional as already mentioned in 5.3.1 Pre-bound Variables in SPARQL Constraints ($this, $shapesGraph, $currentShape)

C. SHACL Shapes to Validate Shapes Graphs

The following shapes graph is intended to enforce many of the syntactic constraints related to SHACL Core in this specification. As such, it can be understood as a machine-readable version of a subset of those constraints, and should be understood as normative. If differences are found between the constraints expressed here and elsewhere in this specification, that indicates an error in the following shapes graph. Please see the Errata Page for an enumeration and analysis of possible errors that have been reported. This shapes graph is available at http://www.w3.org/ns/shacl-shacl. That version may be more up-to-date than this specification as errata are noted against this specification.

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sh:   <http://www.w3.org/ns/shacl#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

@prefix shsh: <http://www.w3.org/ns/shacl-shacl#> .

shsh:
   rdfs:label "SHACL for SHACL"@en ;
   rdfs:comment "This shapes graph can be used to validate SHACL shapes graphs against a subset of
   sh:declare [
      sh:prefix "shsh" ;
      sh:namespace "http://www.w3.org/ns/shacl-shacl#" ;
   ] .


shsh:ListShape
   a sh:NodeShape ;
   rdfs:label "List shape"@en ;
   rdfs:comment "A shape describing well-formed RDF lists. Currently does not check for non-recursi
   rdfs:seeAlso <https://www.w3.org/TR/shacl/#syntax-rule-SHACL-list> ;
   sh:property [
      sh:path [ sh:zeroOrMorePath rdf:rest ] ;
      rdfs:comment "Each list member (including this node) must be have the shape shsh:ListNodeShap
      sh:hasValue rdf:nil ;
      sh:node shsh:ListNodeShape ;
   ] .

shsh:ListNodeShape
   a sh:NodeShape ;
   rdfs:label "List node shape"@en ;
   rdfs:comment "Defines constraints on what it means for a node to be a node within a well-formed
   sh:or ( [
            sh:hasValue rdf:nil ;
            sh:property [
               sh:path rdf:first ;
               sh:maxCount 0 ;
            ] ;
            sh:property [
               sh:path rdf:rest ;
               sh:maxCount 0 ;
```

```
            ] ;
         ]
         [
            sh:not [ sh:hasValue rdf:nil ] ;
            sh:property [
               sh:path rdf:first ;
               sh:maxCount 1 ;
               sh:minCount 1 ;
            ] ;
            sh:property [
               sh:path rdf:rest ;
               sh:maxCount 1 ;
               sh:minCount 1 ;
            ] ;
         ] ) .

shsh:ShapeShape
   a sh:NodeShape ;
   rdfs:label "Shape shape"@en ;
   rdfs:comment "A shape that can be used to validate syntax rules for other shapes."@en ;

   # See https://www.w3.org/TR/shacl/#shapes for what counts as a shape
   sh:targetClass sh:NodeShape ;
   sh:targetClass sh:PropertyShape ;
   sh:targetSubjectsOf sh:targetClass, sh:targetNode, sh:targetObjectsOf, sh:targetSubjectsOf ;
   sh:targetSubjectsOf sh:and, sh:class, sh:closed, sh:datatype, sh:disjoint, sh:equals, sh:flags,
      sh:ignoredProperties, sh:in, sh:languageIn, sh:lessThan, sh:lessThanOrEquals, sh:maxCount, sh
      sh:maxInclusive, sh:maxLength, sh:minCount, sh:minExclusive, sh:minInclusive, sh:minLength, s
      sh:not, sh:or, sh:pattern, sh:property, sh:qualifiedMaxCount, sh:qualifiedMinCount, sh:qualif
      sh:qualifiedValueShape, sh:qualifiedValueShapesDisjoint, sh:qualifiedValueShapesDisjoint, sh:

   sh:targetObjectsOf sh:node ;        # node-node
   sh:targetObjectsOf sh:not ;         # not-node
   sh:targetObjectsOf sh:property ;    # property-node
   sh:targetObjectsOf sh:qualifiedValueShape ; # qualifiedValueShape-node

   # Shapes are either node shapes or property shapes
   sh:xone ( shsh:NodeShapeShape shsh:PropertyShapeShape ) ;

   sh:property [
      sh:path sh:targetNode ;
      sh:nodeKind sh:IRIOrLiteral ;   # targetNode-nodeKind
   ] ;
   sh:property [
      sh:path sh:targetClass ;
      sh:nodeKind sh:IRI ;            # targetClass-nodeKind
   ] ;
   sh:property [
      sh:path sh:targetSubjectsOf ;
      sh:nodeKind sh:IRI ;            # targetSubjectsOf-nodeKind
   ] ;
   sh:property [
      sh:path sh:targetObjectsOf ;
      sh:nodeKind sh:IRI ;            # targetObjectsOf-nodeKind
   ] ;
   sh:or ( [ sh:not [
            sh:class rdfs:Class ;
            sh:or ( [ sh:class sh:NodeShape ] [ sh:class sh:PropertyShape ] )
         ] ]
         [ sh:nodeKind sh:IRI ]
        ) ;                           # implicit-targetClass-nodeKind

   sh:property [
      sh:path sh:severity ;
```

```
    sh:maxCount 1 ;                # severity-maxCount
    sh:nodeKind sh:IRI ;           # severity-nodeKind
] ;
sh:property [
    sh:path sh:message ;
    sh:or ( [ sh:datatype xsd:string ] [ sh:datatype rdf:langString ] ) ;   # message-datatype
] ;
sh:property [
    sh:path sh:deactivated ;
    sh:maxCount 1 ;                # deactivated-maxCount
    sh:in ( true false ) ;         # deactivated-datatype
] ;

sh:property [
    sh:path sh:and ;
    sh:node shsh:ListShape ;       # and-node
] ;
sh:property [
    sh:path sh:class ;
    sh:nodeKind sh:IRI ;           # class-nodeKind
] ;
sh:property [
    sh:path sh:closed ;
    sh:datatype xsd:boolean ;      # closed-datatype
    sh:maxCount 1 ;                # multiple-parameters
] ;
sh:property [
    sh:path sh:ignoredProperties ;
    sh:node shsh:ListShape ;       # ignoredProperties-node
    sh:maxCount 1 ;                # multiple-parameters
] ;
sh:property [
    sh:path ( sh:ignoredProperties [ sh:zeroOrMorePath rdf:rest ] rdf:first ) ;
    sh:nodeKind sh:IRI ;           # ignoredProperties-members-nodeKind
] ;
sh:property [
    sh:path sh:datatype ;
    sh:nodeKind sh:IRI ;           # datatype-nodeKind
    sh:maxCount 1 ;                # datatype-maxCount
] ;
sh:property [
    sh:path sh:disjoint ;
    sh:nodeKind sh:IRI ;           # disjoint-nodeKind
] ;
sh:property [
    sh:path sh:equals ;
    sh:nodeKind sh:IRI ;           # equals-nodeKind
] ;
sh:property [
    sh:path sh:in ;
    sh:maxCount 1 ;                # in-maxCount
    sh:node shsh:ListShape ;       # in-node
] ;
sh:property [
    sh:path sh:languageIn ;
    sh:maxCount 1 ;                # languageIn-maxCount
    sh:node shsh:ListShape ;       # languageIn-node
] ;
sh:property [
    sh:path ( sh:languageIn [ sh:zeroOrMorePath rdf:rest ] rdf:first ) ;
    sh:datatype xsd:string ;       # languageIn-members-datatype
] ;
sh:property [
    sh:path sh:lessThan ;
```

```
    sh:nodeKind sh:IRI ;            # lessThan-nodeKind
] ;
sh:property [
    sh:path sh:lessThanOrEquals ;
    sh:nodeKind sh:IRI ;            # lessThanOrEquals-nodeKind
] ;
sh:property [
    sh:path sh:maxCount ;
    sh:datatype xsd:integer ;       # maxCount-datatype
    sh:maxCount 1 ;                 # maxCount-maxCount
] ;
sh:property [
    sh:path sh:maxExclusive ;
    sh:maxCount 1 ;                 # maxExclusive-maxCount
    sh:nodeKind sh:Literal ;        # maxExclusive-nodeKind
] ;
sh:property [
    sh:path sh:maxInclusive ;
    sh:maxCount 1 ;                 # maxInclusive-maxCount
    sh:nodeKind sh:Literal ;        # maxInclusive-nodeKind
] ;
sh:property [
    sh:path sh:maxLength ;
    sh:datatype xsd:integer ;       # maxLength-datatype
    sh:maxCount 1 ;                 # maxLength-maxCount
] ;
sh:property [
    sh:path sh:minCount ;
    sh:datatype xsd:integer ;       # minCount-datatype
    sh:maxCount 1 ;                 # minCount-maxCount
] ;
sh:property [
    sh:path sh:minExclusive ;
    sh:maxCount 1 ;                 # minExclusive-maxCount
    sh:nodeKind sh:Literal ;        # minExclusive-nodeKind
] ;
sh:property [
    sh:path sh:minInclusive ;
    sh:maxCount 1 ;                 # minInclusive-maxCount
    sh:nodeKind sh:Literal ;        # minInclusive-nodeKind
] ;
sh:property [
    sh:path sh:minLength ;
    sh:datatype xsd:integer ;       # minLength-datatype
    sh:maxCount 1 ;                 # minLength-maxCount
] ;
sh:property [
    sh:path sh:nodeKind ;
    sh:in ( sh:BlankNode sh:IRI sh:Literal sh:BlankNodeOrIRI sh:BlankNodeOrLiteral sh:IRIOrLitera
    sh:maxCount 1 ;                 # nodeKind-maxCount
] ;
sh:property [
    sh:path sh:or ;
    sh:node shsh:ListShape ;        # or-node
] ;
sh:property [
    sh:path sh:pattern ;
    sh:datatype xsd:string ;        # pattern-datatype
    sh:maxCount 1 ;                 # multiple-parameters
    # Not implemented: syntax rule pattern-regex
] ;
sh:property [
    sh:path sh:flags ;
    sh:datatype xsd:string ;        # flags-datatype
```

```
      sh:maxCount 1 ;                # multiple-parameters
   ] ;
   sh:property [
      sh:path sh:qualifiedMaxCount ;
      sh:datatype xsd:integer ;      # qualifiedMaxCount-datatype
      sh:maxCount 1 ;                # multiple-parameters
   ] ;
   sh:property [
      sh:path sh:qualifiedMinCount ;
      sh:datatype xsd:integer ;      # qualifiedMinCount-datatype
      sh:maxCount 1 ;                # multiple-parameters
   ] ;
   sh:property [
      sh:path sh:qualifiedValueShape ;
      sh:maxCount 1 ;                # multiple-parameters
   ] ;
   sh:property [
      sh:path sh:qualifiedValueShapesDisjoint ;
      sh:datatype xsd:boolean ;      # qualifiedValueShapesDisjoint-datatype
      sh:maxCount 1 ;                # multiple-parameters
   ] ;
   sh:property [
      sh:path sh:uniqueLang ;
      sh:datatype xsd:boolean ;      # uniqueLang-datatype
      sh:maxCount 1 ;                # uniqueLang-maxCount
   ] ;
   sh:property [
      sh:path sh:xone ;
      sh:node shsh:ListShape ;       # xone-node
   ] .

shsh:NodeShapeShape
   a sh:NodeShape ;
   sh:targetObjectsOf sh:node ;       # node-node
   sh:property [
      sh:path sh:path ;
      sh:maxCount 0 ;                # NodeShape-path-maxCount
   ] ;
   sh:property [
      sh:path sh:lessThan ;
      sh:maxCount 0 ;                # lessThan-scope
   ] ;
   sh:property [
      sh:path sh:lessThanOrEquals ;
      sh:maxCount 0 ;                # lessThanOrEquals-scope
   ] ;
   sh:property [
      sh:path sh:maxCount ;
      sh:maxCount 0 ;                # maxCount-scope
   ] ;
   sh:property [
      sh:path sh:minCount ;
      sh:maxCount 0 ;                # minCount-scope
   ] ;
   sh:property [
      sh:path sh:qualifiedValueShape ;
      sh:maxCount 0 ;                # qualifiedValueShape-scope
   ] ;
   sh:property [
      sh:path sh:uniqueLang ;
      sh:maxCount 0 ;                # uniqueLang-scope
   ] .

shsh:PropertyShapeShape
```

```
   a sh:NodeShape ;
   sh:targetObjectsOf sh:property ;    # property-node
   sh:property [
      sh:path sh:path ;
      sh:maxCount 1 ;                  # path-maxCount
      sh:minCount 1 ;                  # PropertyShape-path-minCount
      sh:node shsh:PathShape ;         # path-node
   ] .

# Values of sh:and, sh:or and sh:xone must be lists of shapes
shsh:ShapesListShape
   a sh:NodeShape ;
   sh:targetObjectsOf sh:and ;         # and-members-node
   sh:targetObjectsOf sh:or ;          # or-members-node
   sh:targetObjectsOf sh:xone ;        # xone-members-node
   sh:property [
      sh:path ( [ sh:zeroOrMorePath rdf:rest ] rdf:first ) ;
      sh:node shsh:ShapeShape ;
   ] .


# A path of blank node path syntax, used to simulate recursion
_:PathPath
   sh:alternativePath (
      ( [ sh:zeroOrMorePath rdf:rest ] rdf:first )
      ( sh:alternativePath [ sh:zeroOrMorePath rdf:rest ] rdf:first )
      sh:inversePath
      sh:zeroOrMorePath
      sh:oneOrMorePath
      sh:zeroOrOnePath
   ) .

shsh:PathShape
   a sh:NodeShape ;
   rdfs:label "Path shape"@en ;
   rdfs:comment "A shape that can be used to validate the syntax rules of well-formed SHACL paths."
   rdfs:seeAlso <https://www.w3.org/TR/shacl/#property-paths> ;
   sh:property [
      sh:path [ sh:zeroOrMorePath _:PathPath ] ;
      sh:node shsh:PathNodeShape ;
   ] .

shsh:PathNodeShape
   sh:xone (                           # path-metarule
         [ sh:nodeKind sh:IRI ]          # 2.3.1.1: Predicate path
         [ sh:nodeKind sh:BlankNode ;    # 2.3.1.2: Sequence path
           sh:node shsh:PathListWithAtLeast2Members ;
         ]
         [ sh:nodeKind sh:BlankNode ;    # 2.3.1.3: Alternative path
           sh:closed true ;
           sh:property [
             sh:path sh:alternativePath ;
             sh:node shsh:PathListWithAtLeast2Members ;
             sh:minCount 1 ;
             sh:maxCount 1 ;
           ]
         ]
         [ sh:nodeKind sh:BlankNode ;    # 2.3.1.4: Inverse path
           sh:closed true ;
           sh:property [
             sh:path sh:inversePath ;
             sh:minCount 1 ;
             sh:maxCount 1 ;
           ]
```

```
        ]
        [ sh:nodeKind sh:BlankNode ;    # 2.3.1.5: Zero-or-more path
          sh:closed true ;
          sh:property [
            sh:path sh:zeroOrMorePath ;
            sh:minCount 1 ;
            sh:maxCount 1 ;
          ]
        ]
        [ sh:nodeKind sh:BlankNode ;    # 2.3.1.6: One-or-more path
          sh:closed true ;
          sh:property [
            sh:path sh:oneOrMorePath ;
            sh:minCount 1 ;
            sh:maxCount 1 ;
          ]
        ]
        [ sh:nodeKind sh:BlankNode ;    # 2.3.1.7: Zero-or-one path
          sh:closed true ;
          sh:property [
            sh:path sh:zeroOrOnePath ;
            sh:minCount 1 ;
            sh:maxCount 1 ;
          ]
        ]
      ) .

shsh:PathListWithAtLeast2Members
   a sh:NodeShape ;
   sh:node shsh:ListShape ;
   sh:property [
      sh:path [ sh:oneOrMorePath rdf:rest ] ;
      sh:minCount 2 ;    # 1 other list node plus rdf:nil
   ] .

shsh:ShapesGraphShape
   a sh:NodeShape ;
   sh:targetObjectsOf sh:shapesGraph ;
   sh:nodeKind sh:IRI .                # shapesGraph-nodeKind

shsh:EntailmentShape
   a sh:NodeShape ;
   sh:targetObjectsOf sh:entailment ;
   sh:nodeKind sh:IRI .                # entailment-nodeKind
```

D. Summary of SHACL Core Validators

This section enumerates all normative validators of SHACL Core. This section is automatically generated from other parts of this spec and hyperlinks are provided back into the prose if the context of the validator in unclear.

Validators by Constraint Component
sh:ClassConstraintComponent: For each value node that is either a literal, or a non-literal that is not a SHACL instance of `$class` in the data graph, there is a validation result with the value node as `sh:value`.
sh:DatatypeConstraintComponent: For each value node that is not a literal, or is a literal with a datatype that does not match `$datatype`, there is a validation result with the value node as `sh:value`. The datatype of a literal is determined following the datatype function of SPARQL 1.1. A literal matches a datatype if the literal's datatype has the same IRI and, for the datatypes supported by SPARQL 1.1, is not an ill-typed literal.
sh:NodeKindConstraintComponent: For each value node that does not match `$nodeKind`, there is a validation

result with the value node as sh:value. Any IRI matches only sh:IRI, sh:BlankNodeOrIRI and sh:IRIOrLiteral. Any blank node matches only sh:BlankNode, sh:BlankNodeOrIRI and sh:BlankNodeOrLiteral. Any literal matches only sh:Literal, sh:BlankNodeOrLiteral and sh:IRIOrLiteral.
sh:MinCountConstraintComponent: If the number of value nodes is less than $minCount, there is a validation result.
sh:MaxCountConstraintComponent: If the number of value nodes is greater than $maxCount, there is a validation result.
sh:MinExclusiveConstraintComponent: For each value node v where the SPARQL expression $minExclusive < v does not return true, there is a validation result with v as sh:value.
sh:MinInclusiveConstraintComponent: For each value node v where the SPARQL expression $minInclusive <= v does not return true, there is a validation result with v as sh:value.
sh:MaxExclusiveConstraintComponent: For each value node v where the SPARQL expression $maxExclusive > v does not return true, there is a validation result with v as sh:value.
sh:MaxInclusiveConstraintComponent: For each value node v where the SPARQL expression $maxInclusive >= v does not return true, there is a validation result with v as sh:value.
sh:MinLengthConstraintComponent: For each value node v where the length (as defined by the SPARQL STRLEN function) of the string representation of v (as defined by the SPARQL str function) is less than $minLength, or where v is a blank node, there is a validation result with v as sh:value.
sh:MaxLengthConstraintComponent: For each value node v where the length (as defined by the SPARQL STRLEN function) of the string representation of v (as defined by the SPARQL str function) is greater than $maxLength, or where v is a blank node, there is a validation result with v as sh:value.
sh:PatternConstraintComponent: For each value node that is a blank node or where the string representation (as defined by the SPARQL str function) does not match the regular expression $pattern (as defined by the SPARQL REGEX function), there is a validation result with the value node as sh:value. If $flags has a value then the matching must follow the definition of the 3-argument variant of the SPARQL REGEX function, using $flags as third argument.
sh:LanguageInConstraintComponent: For each value node that is either not a literal or that does not have a language tag matching any of the basic language ranges that are the members of $languageIn following the filtering schema defined by the SPARQL langMatches function, there is a validation result with the value node as sh:value.
sh:UniqueLangConstraintComponent: If $uniqueLang is true then for each non-empty language tag that is used by at least two value nodes, there is a validation result.
sh:EqualsConstraintComponent: For each value node that does not exist as a value of the property $equals at the focus node, there is a validation result with the value node as sh:value. For each value of the property $equals at the focus node that is not one of the value nodes, there is a validation result with the value as sh:value.
sh:DisjointConstraintComponent: For each value node that also exists as a value of the property $disjoint at the focus node, there is a validation result with the value node as sh:value.
sh:LessThanConstraintComponent: For each pair of value nodes and the values of the property $lessThan at the given focus node where the first value is not less than the second value (based on SPARQL's < operator) or where the two values cannot be compared, there is a validation result with the value node as sh:value.
sh:LessThanOrEqualsConstraintComponent: For each pair of value nodes and the values of the property $lessThanOrEquals at the given focus node where the first value is not less than or equal to the second value (based on SPARQL's <= operator) or where the two values cannot be compared, there is a validation result with the value node as sh:value.
sh:NotConstraintComponent: For each value node v: A failure must be reported if the conformance checking of v against the shape $not produces a failure. Otherwise, if v conforms to the shape $not, there is validation result with v as sh:value.
sh:AndConstraintComponent: For each value node v: A failure must be produced if the conformance checking of v against any of the members of $and produces a failure. Otherwise, if v does not conform to each member of $and,

there is a validation result with v as sh:value.
sh:OrConstraintComponent: For each value node v: A failure must be produced if the conformance checking of v against any of the members produces a failure. Otherwise, if v conforms to none of the members of $or there is a validation result with v as sh:value.
sh:XoneConstraintComponent: For each value node v let N be the number of the shapes that are members of $xone where v conforms to the shape. A failure must be produced if the conformance checking of v against any of the members produces a failure. Otherwise, if N is not exactly 1, there is a validation result with v as sh:value.
sh:NodeConstraintComponent: For each value node v: A failure must be produced if the conformance checking of v against $node produces a failure. Otherwise, if v does not conform to $node, there is a validation result with v as sh:value.
sh:PropertyConstraintComponent: For each value node v: A failure must be produced if the validation of v as focus node against the property shape $property produces a failure. Otherwise, the validation results are the results of validating v as focus node against the property shape $property.
sh:QualifiedMinCountConstraintComponent: Let C be the number of value nodes v where v conforms to $qualifiedValueShape and where v does not conform to any of the sibling shapes for the *current* shape, i.e. the shape that v is validated against and which has $qualifiedValueShape as its value for sh:qualifiedValueShape. A failure must be produced if any of the said conformance checks produces a failure. Otherwise, there is a validation result if C is less than $qualifiedMinCount. The constraint component for sh:qualifiedMinCount is sh:QualifiedMinCountConstraintComponent.
sh:QualifiedMaxCountConstraintComponent: Let C be as defined for sh:qualifiedMinCount above. A failure must be produced if any of the said conformance checks produces a failure. Otherwise, there is a validation result if C is greater than $qualifiedMaxCount. The constraint component for sh:qualifiedMaxCount is sh:QualifiedMaxCountConstraintComponent.
sh:ClosedConstraintComponent: If $closed is true then there is a validation result for each triple that has a value node as its subject and a predicate that is not explicitly enumerated as a value of sh:path in any of the property shapes declared via sh:property at the current shape. If $ignoredProperties has a value then the properties enumerated as members of this SHACL list are also permitted for the value node. The validation result must have the predicate of the triple as its sh:resultPath, and the object of the triple as its sh:value.
sh:HasValueConstraintComponent: If the RDF term $hasValue is not among the value nodes, there is a validation result.
sh:InConstraintComponent: For each value node that is not a member of $in, there is a validation result with the value node as sh:value.

E. Security and Privacy Considerations

This section is non-normative.

Like most RDF-based technologies, SHACL processors may operate on graphs that are combined from various sources. Some applications may have an open "linked data" architecture and dynamically assemble RDF triples from sources that are outside of an organization's network of trust. Since RDF allows anyone to add statements about any resource, triples may modify the originally intended semantics of shape definitions or nodes in a data graph and thus lead to misleading results. Protection against this (and the following) scenario can be achieved by only using trusted and verified RDF sources and eliminating the possibility that graphs are dynamically added via owl:imports and sh:shapesGraph.

SHACL-SPARQL includes all the security issues of SPARQL.

F. Acknowledgements

This section is non-normative.

Many people contributed to this specification, including members of the RDF Data Shapes Working Group. We especially thank the following:

Arnaud Le Hors (chair until end of 2016), Dean Allemang, Jim Amsden, Iovka Boneva, Olivier Corby, Karen Coyle, Richard Cyganiak, Michel Dumontier, Sandro Hawke, Holger Knublauch, Dimitris Kontokostas, Jose Labra, Pano Maria, Peter Patel-Schneider, Irene Polikoff, Eric Prud'hommeaux, Arthur Ryman (who also served as a co-editor until Feb 2016), Andy Seaborne, Harold Solbrig, Simon Steyskal, Ted Thibodeau, Bart van Leeuwen, Nicky van Oorschot

G. Revision History

This section is non-normative.

The detailed list of changes and their diffs can be found in the Git repository.

Summary of changes to this document since the Proposed Recommendation of 8 June 2017:

- Revision history was removed

H. References

H.1 Normative references

[BCP47]
Tags for Identifying Languages. A. Phillips; M. Davis. IETF. September 2009. IETF Best Current Practice. URL: https://tools.ietf.org/html/bcp47

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

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

[sparql11-entailment]
SPARQL 1.1 Entailment Regimes. Birte Glimm; Chimezie Ogbuji. W3C. 21 March 2013. W3C Recommendation. URL: https://www.w3.org/TR/sparql11-entailment/

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

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

H.2 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/

[shacl-ucr]
SHACL Use Cases and Requirements. Simon Steyskal; Karen Coyle. W3C. 22 January 2016. W3C Working Draft. URL: https://www.w3.org/TR/shacl-ucr/

↥