Tutorial on Jsontron for JSON Semantic Validation

Dr. Amer Ali Dr. Lixin Tao

School of Computer Science and Information Systems
Pace University
Westchester

November 14, 2018

Table of Contents

1 Introduction ... 2
1.1 Constraint Specification in Schematron ... 3
1.2 Element *schema* ... 3
1.3 Element *phase* ... 3
1.4 Element *pattern* ... 3
1.5 Element *rule* ... 4
1.6 Element *assert* or *report* ... 4
2 Setting Up and Running Jsontron ... 4
2.1 Installing *node.js* ... 4
2.2 Installing module *jsontron* ... 5
2.3 Module *jsontron* structure ... 6
2.4 Test run module *jsontron* ... 11
3 How to Specify a Semantic Rule ... 12
3.1 Prerequisites ... 12
3.2 Data ... 12
3.3 Simple Example ... 12
3.4 Adding multiple assert, rule, pattern and phase elements ... 24
3.5 Loan Data Main Example ... 27
3.6 Examples for phase, pattern, rule, assert, context, assert and report elements ... 30
3.7 Stack Overflow Meeting Times Dilemma Example ... 31
4 IBM Schematron Tutorial ... 33
4.1 Prerequisites ... 33
4.2 Schematron overview and example ... 34
4.3 Basics of rules, patterns, and assertions ... 35
4.4 Reports and communications control ... 55
4.5 Intermediate Schematron features ... 60
5 References ... 68

---

1 Introduction

Schematron is a rule-based XML validation schema language for making assertions about the presence or absence of patterns in XML trees. It is different from XML syntax schema languages like XML Schema, RELAX NG or DTD. Schematron is capable of specifying validation rules that syntax-based schema languages cannot. For instance, it can control the contents of an element via its siblings. The fundamental difference from other languages is that Schematron is not based on grammar, but it is based on finding tree patterns in XML documents and that makes it suitable for finding and validating structures that are difficult to be represented in grammar-based languages.

Rick Jelliffe [1] invented Schematron while working (1999-2001) at Academia Sinica, Taipei, Taiwan. He described Schematron as “a feather duster to reach the corners that other schema languages cannot reach”.

Schematron has been standardized by the ISO as “Information technology, Document Schema Definition Languages (DSDL), Part 3: Rule-based validation, Schematron (ISO/IEC 19757-3:2016)” [2]. ISO formally describes Schematron as below:

> “Considered as a document type, a Schematron schema contains natural-language assertions concerning a set of documents, marked up with various elements and attributes for testing these natural-language assertions, and for simplifying and grouping assertions.
>
> Considered theoretically, a Schematron schema reduces to a non-chaining rule system whose terms are Boolean functions invoking an external query language on the instance and other visible XML documents, with syntactic features to reduce specification size and to allow efficient implementation.
>
> Considered analytically, Schematron has two characteristic high-level abstractions: the pattern and the phase. These allow the representation of non-regular, non-sequential constraints that ISO/IEC 19757-2 [Regular-grammar-based validation — RELAX NG] [3] cannot specify and various dynamic or contingent constraints.”

Schematron is used in many use cases like business rules validation, data reporting, general validation, quality control, quality assurance, firewalling, filtering, constraint checking, naming and design rules checking, statistical consistency, data exploration, transformation testing, feature extraction, house-style-rules checking.

Schematron is used across many sectors and industries. Some of its users are:

- US National Emergency Management System (NEMSYS)
- NASA Planetary Data System
- National Environmental Information Exchange Network the US
- Australian Bureau of Meteorology

- Japanese Local Government
- Open Vulnerability and Assessment Language (OVAL) US Dept Homeland Security
- PEPPOL Pan-European Public Procurement Online
- European Commission e-trustex secure document transfer platform
- HM Revenue and Customs (PDF) the UK
- Aeronautical Information Exchange Rule Checker EUROCONTROL
- W3C Service Modeling Language SML
- Inline XBRL Validation
- ACORD Reinsurance and large commercial
- Associated Press
- US National Information Exchange Model Tools

1.1 Constraint Specification in Schematron

The power of Schematron lies in its simplicity and abstraction. There are only a few essential elements but they allow specifying constraints that are not possible in other schema languages. Below are essential building blocks of a Schematron schema:

```
schema
    title
    phase
    pattern+
        rule+
            (assert or report)+
```

1.2 Element *schema*

This is the top-level element of a Schematron schema. All other elements are enclosed inside the schema element. This element has several optional attributes like *title*, *schemaVersion*, *queryBinding* and *defaultPhase*. Some of these optional elements will be explained later.

1.3 Element *phase*

The is a higher level of abstraction and specifies a group of patterns to be activated to cater to variation in schemas. It supports progressive validation. ‘#ALL’ and ‘#DEFAULT’ are special phases that activate all and the default phase respectively. The phase element will be discussed more later.

1.4 Element *pattern*

The *pattern* element contains a set of *rule* elements. This is a higher level abstraction to encompass related rules. It has several optional attributes that will be discussed later.

1.5 Element *rule*

A *rule* element contains one or more assertions that need to be applied to a given context. The *rule* element has a required *context* attribute that returns the nodes on which the assertions need to be applied. A query language like XPath is used to select the nodes via the *context* expression.

1.6 Element *assert* or *report*

The *assert* and *report* elements contain a *test* attribute that is the condition to be tested on the *context* nodes. The content of the *assert* or *report* element is the message that is returned as a result of the test. The *assert* element will display the message if the test fails whereas the *report* element will display the message if the test passes. Similar to attribute context, the value of attribute *test* is expressed in a query language such as XPath. The value of attribute *context* is an XPath statement to express 'where' to test, and the value of attribute test is an XPath statement to express 'what' to test.

This tutorial introduces a Schematron-based semantic validator, *Jsontron*, explains how to set it up and its basic features through a series of hands-on labs in parallel to the IBM Schematron tutorial [4].

2 Setting Up and Running Jsontron

2.1 Installing *node.js*

Download and install *node.js* from https://nodejs.org/en/download/

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS
Recommended For Most Users

Current
Latest Features

Windows Installer | macOS Installer | Source Code

Windows Installer (.msi)	32-bit	64-bit	
Windows Binary (.zip)	32-bit	64-bit	
macOS Installer (.pkg)	64-bit		
macOS Binary (.tar.gz)	64-bit		
Linux Binaries (x64)	64-bit		
Linux Binaries (ARM)	ARMv6	ARMv7	ARMv8
Source Code	node-v10.13.0.tar.gz		

- You should see installer in your downloads folder or where you saved it.
- Double-click the installer and follow instructions to install *node*
- *NPM* (Node Package Manager) gets installed as part of *node* installation. You don't need to do anything extra for *npm* installation.

Open a command prompt and verify that *node* and *npm* have been properly installed

```
C:\>node -v
v10.13.0

C:\>npm -v
6.4.1
```

Note: Default installer puts *node* and *npm* on environment variable PATH so you can access them from anywhere.

Note: The default installer sets up a Node.js command prompt that you can access from Programs menu if you are using Windows. You can use a normal command prompt as well as long as *node* and *npm* are on PATH.

Setup *Node* and *NPM* on Ubuntu

- Run "sudo apt install nodejs" to install *node*.
- If you see error "E: Could not get lock /var/lib/dpkg/lock - open (11: Resource temporarily unavailable)", run "sudo rm /var/lib/apt/lists/lock" first.
- Run "sudo apt install npm" in install *npm*.
- Run "node –v" and "npm –v" to verify your installation.

```
pace@ubuntu18:~$ node -v
v8.10.0
pace@ubuntu18:~$ npm -v
3.5.2
```

2.2 Installing module *jsontron*

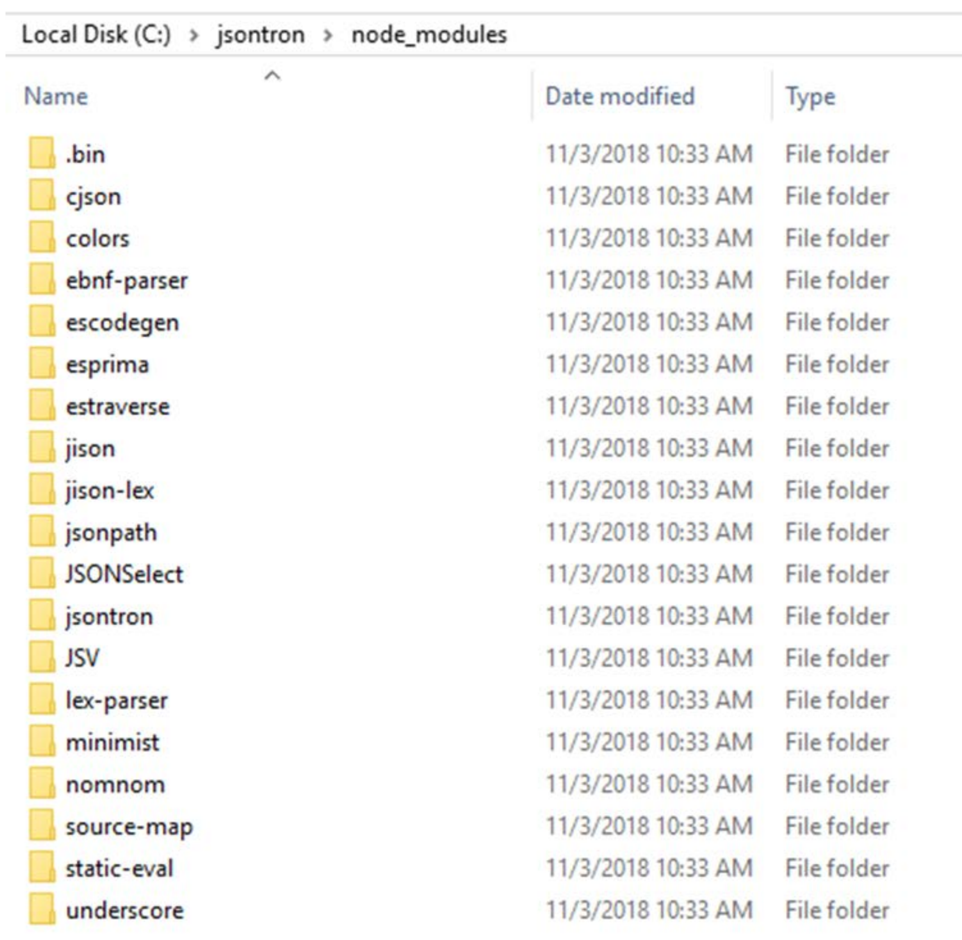
- Create a new folder *jsontron*.
- Open terminal window in folder *jsontron*, and run command: *npm i jsontron*

```
C:\>cd jsontron

C:\jsontron>npm i jsontron
npm WARN deprecated nomnom@1.5.2: Package no longer supported. Contact support@npmjs.com for more info.

> jsonpath@0.2.12 postinstall C:\jsontron\node_modules\jsonpath
> node lib/aesprim.js > generated/aesprim-browser.js

npm WARN saveError ENOENT: no such file or directory, open 'C:\jsontron\package.json'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open 'C:\jsontron\package.json'
npm WARN jsontron No description
npm WARN jsontron No repository field.
npm WARN jsontron No README data
npm WARN jsontron No license field.

+ jsontron@0.8.12
added 24 packages from 47 contributors and audited 29 packages in 4.266s
found 1 moderate severity vulnerability
  run `npm audit fix` to fix them, or `npm audit` for details

C:\jsontron>dir
 Volume in drive C has no label.
 Volume Serial Number is 92F0-27C6

 Directory of C:\jsontron

11/03/2018  10:33 AM    <DIR>          .
11/03/2018  10:33 AM    <DIR>          ..
11/03/2018  10:33 AM    <DIR>          node_modules
11/03/2018  10:33 AM             5,930 package-lock.json
               1 File(s)          5,930 bytes
               3 Dir(s)  4,787,364,188,160 bytes free
```

- You will see a new directory called node_modules created. That is where all node modules are installed.
- Later you can also update folder *jsontron* by running *npm update jsontron* in the current *node* root folder.

2.3 Module *jsontron* structure

- Go to folder *node_modules*, and you will see many modules installed in folder *node_modules* along with module *jsontron*. This is because *npm* automatically installs all the dependencies.



Local Disk (C:) › jsontron › node_modules

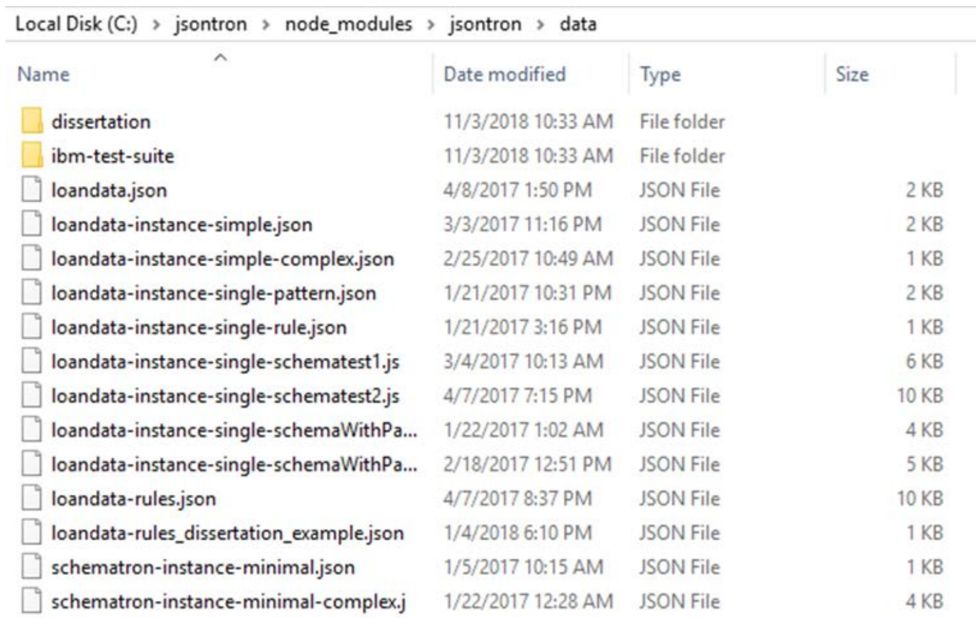
Name	Date modified	Type
.bin	11/3/2018 10:33 AM	File folder
cjson	11/3/2018 10:33 AM	File folder
colors	11/3/2018 10:33 AM	File folder
ebnf-parser	11/3/2018 10:33 AM	File folder
escodegen	11/3/2018 10:33 AM	File folder
esprima	11/3/2018 10:33 AM	File folder
estraverse	11/3/2018 10:33 AM	File folder
jison	11/3/2018 10:33 AM	File folder
jison-lex	11/3/2018 10:33 AM	File folder
jsonpath	11/3/2018 10:33 AM	File folder
JSONSelect	11/3/2018 10:33 AM	File folder
jsontron	11/3/2018 10:33 AM	File folder
JSV	11/3/2018 10:33 AM	File folder
lex-parser	11/3/2018 10:33 AM	File folder
minimist	11/3/2018 10:33 AM	File folder
nomnom	11/3/2018 10:33 AM	File folder
source-map	11/3/2018 10:33 AM	File folder
static-eval	11/3/2018 10:33 AM	File folder
underscore	11/3/2018 10:33 AM	File folder

- Go to folder *jsontron*, and you will find the following contents.

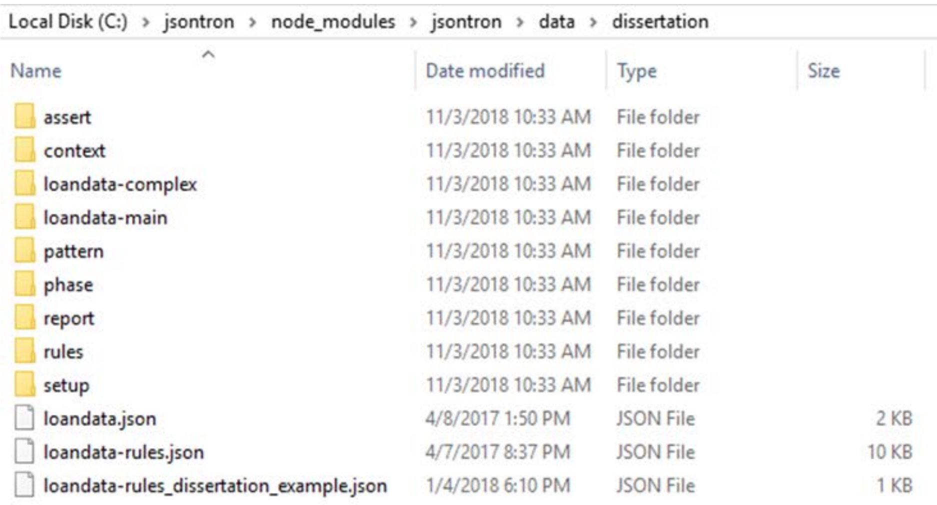
Local Disk (C:) › jsontron › node_modules › jsontron

Name	Date modified	Type	Size
.settings	11/3/2018 10:33 AM	File folder	
bin	11/3/2018 10:33 AM	File folder	
data	11/3/2018 10:33 AM	File folder	
docs	11/3/2018 10:33 AM	File folder	
examples	11/3/2018 10:33 AM	File folder	
lib	11/3/2018 10:33 AM	File folder	
schemas	11/3/2018 10:33 AM	File folder	
tests	11/3/2018 10:33 AM	File folder	
.npmignore	9/14/2018 10:57 PM	NPMIGNORE File	1 KB
.project	8/19/2018 12:07 AM	PROJECT File	1 KB
package.json	11/3/2018 10:33 AM	JSON File	2 KB
README.md	9/14/2018 10:19 PM	MD File	4 KB
scratch2.js	8/19/2018 7:24 PM	JavaScript File	1 KB
scratch3.js	10/6/2018 6:15 PM	JavaScript File	1 KB
scratch3-ibm.js	10/10/2018 3:06 PM	JavaScript File	2 KB

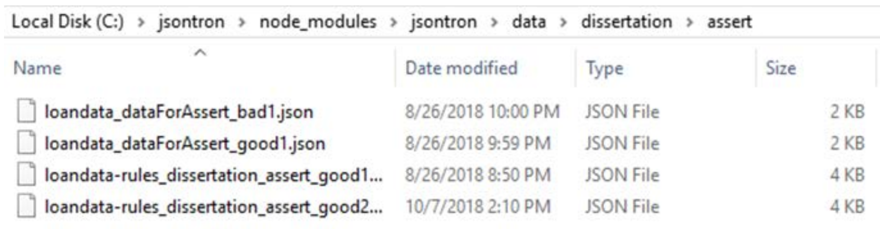
- Folder *data* contains all data for examples. The dissertation example data is in folder *dissertation* and IBM example data is in folder *ibm-test-suite*.



- In folder *dissertation*, you will find data for main use cases like the main example, assert, context, pattern, phase, and report.

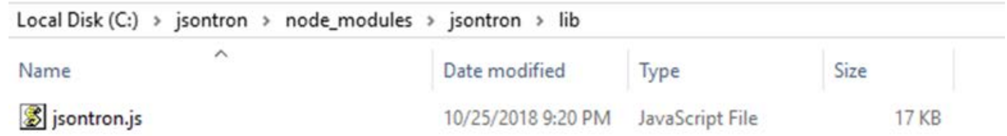


- If you go to one of the example folders like *assert*, you will see the good instance, bad instance and rules file.

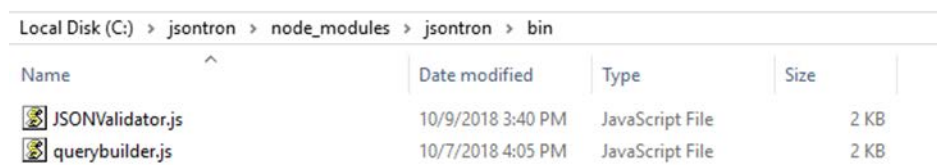


loandata-rules_* is rules file.
loandata-dataForAssert_bad* is instance document with bad data.
loandata-dataForAssert_good* is instance document with good data.

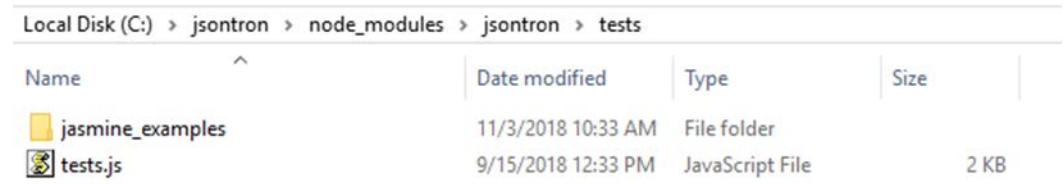
- Folder *jsontron/lib* contains the main source code for module *jsontron*



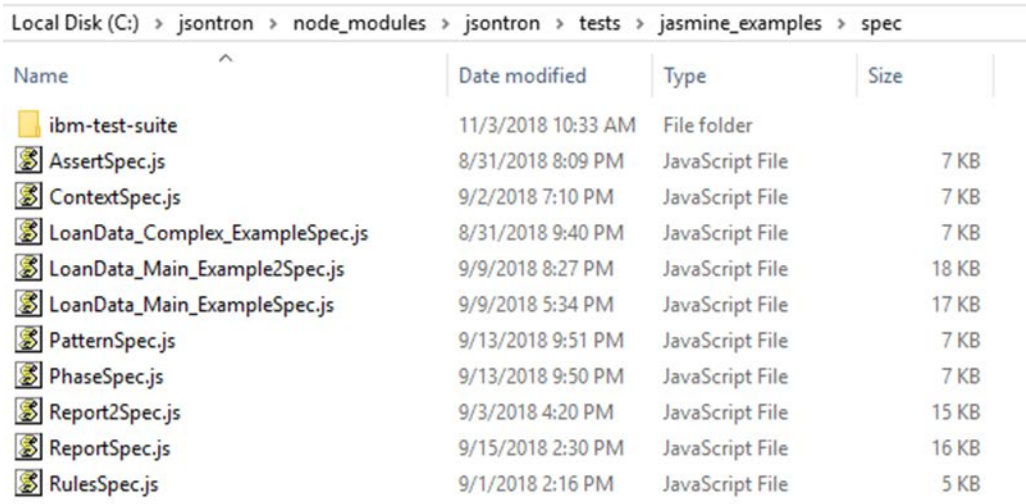
- Folder *jsontron/bin* contains the command-line invocation code of the program, *JSONValidator.js*.



- Folder *jsontron/tests* contains the command-line invocation scripts for all the tests.



- The data for tests are in folder jsontron/data.



2.4 Setting up jsontron environment variables on Windows

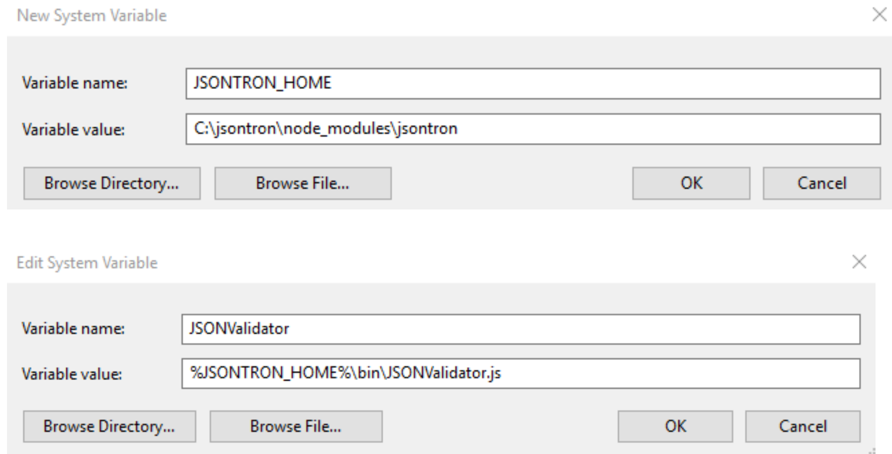
Please set up a new OS environment variable JSONTRON_HOME pointing to the root folder of the jsontron module. In our example, you can do so in the current Windows terminal window by running command

```
set JSONTRON_HOME= C:\jsontron\node_modules\jsontron
```

Please create a new environment variable JSONValidator and set its value as below.

```
set JSONValidator=%JSONTRON_HOME%\bin\JSONValidator.js
```

To avoid repeated setting up these two environment variables, add them in Windows *Environment Variables* pane:



Start a new Windows terminal window, run commands "echo %JSONTRON_HOME%" and "echo %JSONValidator%", make sure you get the same output as below:

```
C:\>echo %JSONTRON_HOME%
C:\jsontron\node_modules\jsontron

C:\>echo %JSONValidator%
C:\jsontron\node_modules\jsontron\bin\JSONValidator.js
```

2.5 Setting up jsontron environment variables on Linux

Assume you have set up **jsontron** in *~/jsontron*. Open terminal window in ~. Run *sudo gedit .bashrc* to add the following two lines at the bottom of file *~/.bashrc*:

```
export  JSONTRON_HOME=~/jsontron/node_modules/jsontron
export  JSONValidator=$JSONTRON_HOME/bin/JSONValidator.js
```

Save and exit *~/.bashrc*. Run *source ~/.bashrc* to active the edited environments. Verify your new environments:

```
pace@ubuntu18:~$ echo $JSONTRON_HOME
/home/pace/jsontron/node_modules/jsontron
pace@ubuntu18:~$ echo $JSONValidator
/home/pace/jsontron/node_modules/jsontron/bin/JSONValidator.js
```

2.6 Test run module *jsontron*

Change terminal window work folder to "jsontron/node_modules/jsontron/data/dissertation/pattern" by "cd %JSONTRON_HOME%\data\dissertation\pattern" (Windows) or "cd $JSONTRON_HOME/data/dissertation/pattern" (Linux).

On Windows, run the following semantic validation command:

```
node %JSONValidator% -i loandata_pattern_good1.json -r loandata-rules_dissertation_pattern_good1.json
```

```
C:\>cd %JSONTRON_HOME%\data\dissertation\pattern

C:\jsontron\node_modules\jsontron\data\dissertation\pattern>node  %JSONValidator%  -i  loandata_pattern_good1.json  -r
loandata-rules_dissertation_pattern_good1.json
Starting Semantic Validation .........
Parsing Pattern: precheck_pattern
Parsing Pattern: patternid1
Parsing Pattern: patternid2
Parsing Pattern: patternid3
4 Pattern(s) Requested. 4 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

On Linux, run the following semantic validation command:

node $JSONValidator -i loandata_pattern_good1.json -r loandata-rules_dissertation_pattern_good1.json

```
pace@ubuntu18:~$ cd  $JSONTRON_HOME/data/dissertation/pattern
pace@ubuntu18:~/jsontron/node_modules/jsontron/data/dissertation/pattern$ node
$JSONValidator  -i  loandata_pattern_good1.json  -r  loandata-rules_dissertation
_pattern_good1.json
Starting Semantic Validation .........
Parsing Pattern: precheck_pattern
Parsing Pattern: patternid1
Parsing Pattern: patternid2
Parsing Pattern: patternid3
4 Pattern(s) Requested. 4 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

In the remainder of this tutorial, we assume you are using a Linux system. You can easily revise our commands for Windows, as shown above.

3 How to Specify a Semantic Rule

In this tutorial, we will go through the steps you need to take to specify a Schematron rule for the system that we developed in this study.

3.1 Prerequisites

A working knowledge of JSON, JSONPath, and JavaScript is assumed.

3.2 Data

All the files for the examples in this tutorial are located at below location:
>> $JSONTRON_HOME/examples/main_features_examples

3.3 Simple Example

First let's use a simple example to construct a JSON Schematron rules file.

Let's assume a Bank gets JSON documents below for loan and wants to validate these documents for a few rules.

```
{
"loan_data":{
        "loans":[
                {
                        "loan_id":"1234567",
                        "loan_type":"FHA",
                        "customer_id":"JD689457",
                        "amount":500000,
                        "interest_rate":3.75,
                        "prime_rate":3.25,
                },
                        {
                        "loan_id":"1234568",
                        "loan_type":"Traditional",
                        "customer_id":"JD689457",
                        "amount":300000,
                        "interest_rate":3.75,
                        "prime_rate":3.25,
                }]}

}
```

Business Rules:

1. For all loan types, the interest rate cannot be less than 3.75%.

Specifying the Schematron Rules:

Below is a simple rule file that validates the first business rule.

```
{
"schema":{

"id":"Loan Data Rules",
"title":"Schematron Semantic Validation",
"schemaVersion":"ISO Schematron 2016",
"queryBinding":"jsonpath",

"defaultPhase":"phaseid1",

 "phase": [
           {
             "id":"phaseid1",
              "active":["patternid1"]
           }],
 "pattern": [
             {
               "id":"patternid1",
                "title":"Interest Rate Pattern",
                "abstract":false,
     "rule": [
              {
                "id":"RateRule1",
                "abstract":false,
                 "context": "$.loan_data.loans.*",

           "assert":[
                {
                    "id":"assertidINT21",
                    "test": "(jp.query(contextNode,'$..interest_rate') >= 3.75",
                    "message": "Assert 1: Interest Rate cannot be less than 3.75 Percent"
                 }

  ]}]}]}}
```

A few things to note about this Schematron Rules File:

- *schema* is the top-level element
- *schema* and *pattern* are the mandatory elements.
- *context* expression is behind the scene 2nd argument of jsonpath query() method jp.query(contextNode, '$.loan_data.loans.*')
- In the test expression, *contextNode* is a keyword in this implementation and represents the node-set returned from the *context* expression, and *jsonpath* methods are used as shown above.

Step 1: Specify Schema Element

Top level element in Schematron rules file is a ‘schema’ element that is an object element.

```
{
  "schema":{}
}
```

Step 2: Specify Schema Meta Data

Schema element has several meta data elements like ‘id’, ‘title’, ‘schemaVersion’, and ‘queryBinding’. These are optional elements but can hold important information.

Let’s add these optional elements to our schema:

```
{
    "schema":{

                   "id":"Loan Data Rules",
                   "title":"Schematron Semantic Validation",
                   "schemaVersion":"ISO Schematron 2016",
                   "queryBinding":"jsonpath"

           }
}
```

Step 3: Add a mandatory ‘pattern’ element

A pattern element contains a collection of rules. A schema can contain multiple patterns. We will use an array element to hold multiple pattern objects. Let’s first add an empty pattern array to our schema:

```
{
    "schema":{

                  "id":"Loan Data Rules",
                   "title":"Schematron Semantic Validation",
                   "schemaVersion":"ISO Schematron 2016",
                   "queryBinding":"jsonpath",

                   "pattern":[]

                   }
}
```

Step 4: Add our first pattern object into the pattern array

Unlike XML to which you can add multiple elements with same tag name, in JSON you need use an array to hold multiple elements.

In XML you can use something like:

```
<schema>
        <pattern> pattern 1 </pattern>
        <pattern> pattern 2 </pattern>
        <pattern> pattern 3 </pattern>
</schema>
```

In JSON, since the object contains key/value pairs, therefore, the key has to be unique. So you will use an array instead and each member of array will be a pattern object:

```
{  "schema": {
            "patter":[
                        {"patterned": "pattern1"},
                        {"patterned": "pattern2"},
                        {"patterned": "pattern3"}
                  ]
            }}
```

Now let's add our first pattern object to our collection of patterns:

```
{
    "schema":{

                "id":"Loan Data Rules",
                "title":"Schematron Semantic Validation",
                "schemaVersion":"ISO Schematron 2016",
                "queryBinding":"jsonpath",

                "pattern":[
                            {
                              "id":"patternid1",
                              "title":"Interest Rate Pattern",
                              "abstract":false
                             }

                        ]

                }
}
```

Step 5: Add a rule to the pattern

As mentioned above, a pattern is a collection of rules. We will again use an array to hold multiple rule elements for the same reason that we explained in the pattern array case. Let's add a rule array to our pattern.

```
{
    "schema":{

                 "id":"Loan Data Rules",
                "title":"Schematron Semantic Validation",
                "schemaVersion":"ISO Schematron 2016",
                "queryBinding":"jsonpath",

                "pattern":[
                                  {
                     "id":"patternid1",
                     "title":"Interest Rate Pattern",
                     "abstract":false,
```

```
                    "rule":[]
                  _}

                                   ]

                }
   }
```

Step 6: Add our first rule object to the rule array with meta data

Similar to a pattern array, a rule array contains a comma separated list of rule objects. A rule contains a 'context' and collection of assertions that are tested against the context. Let's add our first rule object with some meta data. We will discuss 'context' and assertion elements in the following sections.

```
   {
       "schema":{

                "id":"Loan Data Rules",
                 "title":"Schematron Semantic Validation",
                 "schemaVersion":"ISO Schematron 2016",
                 "queryBinding":"jsonpath",

                 "pattern":[
                                   {
                     "id":"patternid1",
                     "title":"Interest Rate Pattern",
                     "abstract":false,

                     "rule":[
                                   {
                                       "id":"RateRule1",
                                       "abstract":false


                                   }

                              ]
                   }

                                   ]

                }
   }
```

Step 7: Add a 'context' to the rule

A JSON instance document is a collection of contexts. In the rule element, we use a query language statement to select a node set from the instance document so that we can run our

assertions against that node set. This selected node set is called 'context'. Our business rule 1 at the beginning says:

"For all loan types, the interest rate cannot be less than 3.75%."

So here we have to select all loan elements/objects from the instance document so that we can check the interest rates of each loan. We will use the JSONPath query below to select all loans from the instance document:

```
"context": "$.loan_data.loans.*"
```

A few important things to note about the 'context' element:

- The value of the context element is the JSONPath query() method's second argument. So internally it will be used as:

```
var contextNode = jp.query(JSONInstanceDocument, "$.loan_data.loans.*")
```

where variable 'contextNode' will hold the nodeset returned, 'jp' is a JSONPath object, 'JSONInstanceDocument' is the instance document and our context query is the second argument to the jp.query() method.

- The query will return an array containing the nodes that were selected by the query. In this case it will return all elements of all the loans in the loan_data instance document. "$" represents root, that is similar to '/' in XPath queries.
- The context query can return an empty node set. If that is the case, as per Schematron specification, the implementation will return as if validation has passed.

```
{
    "schema":{

            "id":"Loan Data Rules",
            "title":"Schematron Semantic Validation",
            "schemaVersion":"ISO Schematron 2016",
            "queryBinding":"jsonpath",

             "pattern":[
                                {
                    "id":"patternid1",
                    "title":"Interest Rate Pattern",
                    "abstract":false,

                    "rule":[
                                {
                                    "id":"RateRule1",
                                     "abstract":false,

                                     "context": "$.loan_data.loans.*"
```

```
                                    }

                             ]
                    }

                                    ]

                 }
   }
```

Step 8: Add assertions to the rule

We have selected the context node set in the previous step. Now we need to add assertions that will be applied to the context node set. A rule can contain multiple assertions. So similar to pattern and rule elements, we will use an array element to hold the assertions. In Schematron there are two types of assertions; ‘assert’ and ‘report’. An ‘assert’ will print the message if the test fails whereas in case of ‘report’ it will print the message if the test passes. For simplicity’s sake, we have used only ‘assert’ element in this implementation as adding ‘report’ is trivial. Let’s first add an assert array to the rule element to hold multiple assertions.

```
   {
       "schema":{

             "id":"Loan Data Rules",
                 "title":"Schematron Semantic Validation",
                 "schemaVersion":"ISO Schematron 2016",
                 "queryBinding":"jsonpath",

                 "pattern":[
                        {
                      "id":"patternid1",
                      "title":"Interest Rate Pattern",
                      "abstract":false,

                      "rule":[
                        {
                         "id":"RateRule1",
                          "abstract":false,

                           "context": "$.loan_data.loans.*",

                           "assert":[]


                        }

                             ]
                    }

                                    ]
```

```
                    }
    }
```

Step 9: Add an assert object with test and message to assert array

Other than 'id', an assert object contains a 'test' and a 'message' element. Understanding how the test element works in conjunction with 'context' is the key to understand this specification and implementation. The 'context' element in step 7 above is purely a JSONPath expression. However, in case of the 'test' element in an assertion, you can use JSONPath expression or a plain JavaScript expression or a combination of both. In step 7 we selected all nodes for all the loan items. In this step we will select the 'interest_rate' elements and then test whether those are equal or greater than 3.75%. The test will look like below:

```
"test": "(jp.query(contextNode,'$..interest_rate')) >= 3.75"
```

A few important things to note about this test expression. The first portion of the test is a JSONPath query method. [`jp.query(contextNode,'$..interest_rate')`]. In this method 'contextNode' is a keyword in this implementation. It holds the output node set returned by the context expression from step 7. The method states that select interest_rate elements and then check whether each of those interest_rate elements have value greater than or equal to 3.75 [`>= 3.75`]. This portion of the test expression is plain JavaScript. So overall we use JSONPath query method to select all the interest_rate elements and then use JavaScript expression to implement our business rule. If the test returns false (the test fails) then a human readable 'message' is printed.

```
"message": "Assert assertidINT21: Interest Rate cannot be less than 3.75 Percent"
```

Another important thing to note about the test expression is that unlike the 'context' expression where only the second argument of query method is used, we use the full query method.

The reason for allowing JavaScript expressions and using full JSONPath methods is to overcome the problem we have with JSONPath that it has limited functions as compared to XPath. Now let's add our first assertion to the rule:

```
{
    "schema":{

                "id":"Loan Data Rules",
                    "title":"Schematron Semantic Validation",
                    "schemaVersion":"ISO Schematron 2016",
                    "queryBinding":"jsonpath",

                    "pattern":[
                            {
                        "id":"patternid1",
                        "title":"Interest Rate Pattern",
```

```
                        "abstract":false,

                        "rule":[
                          {
                           "id":"RateRule1",
                             "abstract":false,

                              "context": "$.loan_data.loans.*",

                              "assert":[
                {

                 "id":"assertidINT21",
                "test": "(jp.query(contextNode,'$..interest_rate') >= 3.75",
                "message": Assert assertidINT21: Interest Rate cannot be less than
3.75 Percent
                }

                                          ]

                          }

                                     ]
                       }

                                            ]
                    }  }
```

We have completed adding a pattern to our Schematron schema. Now let's add a 'phase' element to the schema.

Step 10: Add a phase element to the schema

A 'phase' is a mechanism in Schematron used to dynamically activate the patterns of choice. A phase element is an array similar to pattern, rule and assert as it may need to hold more than one phase objects. There are a few peculiar things about the phase element.

- #ALL is a special keyword that can be used at invocation time to activate all phases.
- #DEFAULT is a special keyword that can be used to activate the default phase
- 'defaultPhase' is the property in the schema to nominate a phase as a default phase

Now let's add a phase to the schema and make it as a default phase.

```
{
      "schema":{

                    "id":"Loan Data Rules",
                     "title":"Schematron Semantic Validation",
                     "schemaVersion":"ISO Schematron 2016",
                     "queryBinding":"jsonpath",
```

```
                    "defaultPhase":"phaseid1",

                     "phase": [
                                {
                                        "id":"phaseid1",
                                        "active":["patternid1"]
                                }],


                    "pattern":[
                                        {
                        "id":"patternid1",
                         "title":"Interest Rate Pattern",
                         "abstract":false,

                         "rule":[
                           {
                             "id":"RateRule1",
                              "abstract":false,

                               "context": "$.loan_data.loans.*",
                               "assert":[
                {
                    "id":"assertidINT21",
                    "test": "(jp.query(contextNode,'$..interest_rate') >= 3.75",
                    "message": "Assert 1: Interest Rate cannot be less than 3.75 Percent"
                                                        }
                                                        ]}]}]}}
```

Our simple schema is now complete. Let's run this schema with good and bad data.

The rules file and instance documents are located at:

```
>> $JSONTRON_HOME/examples/main_features_examples/simple_example
      loandata_rules_simple.json
      loandata_data_simple_good1.json
      loandata_rules_simple.json
```

We are now ready to run the simple example. Follow the below steps to run the example.

Make sure you are in the above *simple_example* folder.
Run: **node $JSONValidator -i loandata_data_simple_good1.json -r loandata_rules_simple.json**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/main_features_examples/simple_example
$ node $JSONValidator -i loandata_data_simple_good1.json -r loandata_rules_simple.json
Starting Semantic Validation .........
Parsing Pattern: patternid1
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

The instance document loandata_data_simple_good1.json has been successfully validated.

Now let's run the same rule against the bad instance document:
Run: **node $JSONValidator -i loandata_data_simple_bad1.json -r loandata_rules_simple.json**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/main_features_examples/simple_example
$ node $JSONValidator -i loandata_data_simple_bad1.json -r loandata_rules_simple.json
Starting Semantic Validation .........
Parsing Pattern: patternid1
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY ENABLING DE
BUG WITH -d OPTION ****
Completed Semantic Validation .........
```

The message tells us that the document is not valid against the rules. Let's use the debug option to get the detailed message:

Run: **node $JSONValidator -i loandata_data_simple_bad1.json -r loandata_rules_simple.json -d**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/main_features_examples/simple_example
$ node $JSONValidator -i loandata_data_simple_bad1.json -r loandata_rules_simple.json -d
Starting Semantic Validation .........
Parsing Pattern: patternid1
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY ENABLING DE
BUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 2
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
       ruleContext: [Array],
       assertionid: 'assertidINT21',
       assertionTest: '(jp.query(contextNode,\'$..interest_rate\')) >= 3.75',
       message: 'assertidINT21: Interest Rate cannot be less than 3.75 Percent',
       assertionValid: false },
     { schRule: [Object],
       ruleContext: [Array],
       assertionid: 'assertidINT21',
       assertionTest: '(jp.query(contextNode,\'$..interest_rate\')) >= 3.75',
       message: 'successful',
       assertionValid: true } ],
  finalValidationReport:
   [ { schRule: [Object],
       ruleContext: [Array],
       assertionid: 'assertidINT21',
       assertionTest: '(jp.query(contextNode,\'$..interest_rate\')) >= 3.75',
       message: 'assertidINT21: Interest Rate cannot be less than 3.75 Percent',
       assertionValid: false } ],
  valid: false }
```

A few things to note about the detailed report.

- It is generated by the *-d* option at the end of the command

- ‘Total Errors Found’ denotes the number of system errors like missing files or ill formed documents. These are not validation errors but program errors.
- ‘Total Warnings Found’ denotes any issue that are not fatal but need attention like a context expression returning empty node set.
- ‘Total Validations’ denotes how many assertion tests were executed against how many nodes. In this case there were two loan items against which one assertion was executed so total 2 validations. [2 nodes x 1 assert]. It includes all validations whether passed or failed. It is denoted by validations array in the report.
- ‘Total Failed Assertions” denotes the failed validations. It is denoted by the ‘finalValidationReport’ array in the report.
- The last line in the report is ‘valid’ Boolean attribute. The report has one passed validation and one failed validation, but overall the validation has failed so this attribute will return ‘false’. All validations have to pass and there shouldn’t be any warnings or errors for this ‘valid’ attribute to be set to ‘true’.

3.4 Adding multiple assert, rule, pattern and phase elements

As mentioned earlier, the *assert*, *rule*, *pattern* and *phase* elements are array elements and can contain multiple items of their own type respectively. In this example, we will extend the rules schema from example 3.3 and add an assert, a rule, a pattern, and a phase element to the schema.

Business Rules:

2. For all loan types, the prime rate cannot be more than 5%.
3. Traditional loan type’s maximum amount cannot be more than $1 million

Data:

The data for this example is found in the following folder:

```
>> $JSONTRON_HOME/examples/main_features_examples/multiple_elements_example
      loandata_rules_multi.json
      loandata_data_multi_good1.json
      loandata_rules_multi.json
```

Extending schema rules

We have two new business rules that we need to incorporate into our Schematron schema that we developed in example 3.3.

- For **business rule # 2**, since the context is the same so we will add another assert object in the same rule.
    - The **context** expression will be similar to example 3.3
      `"context": "$.loan_data.loans.*"`

- The **test** will be similar but instead of interest_rate, it will be applied to prime_rate nodes and upper limit for prime_rate will be 5% as per business rule
  `"test": "(jp.query(contextNode,'$..prime_rate')) <= 5"`

- For **business rule # 3**, since it is applicable to 'Traditional' loan types, therefore, we will put it in a separate pattern (patternid2) that in turn will contain a new rule with a new context and assertion.
  - The **context** expression will select the "Traditional" loans only instead of all loans
    `"context": "$.loan_data.loans[?(@.loan_type === 'Traditional')]"`

  - The **test** will now apply to the selected "Traditional" loan types only and will emit a message if it exceeds $1M as per business rule
    `"test": "(jp.query(contextNode,'$..amount') <= 1000000"`

- Since another pattern is added, we can separate all loans and traditional loan related patterns in different phases by adding another phase (phaseid2) to contain 'patternid2'

```
{
    "schema":{

            "id":"Loan Data Rules",
            "title":"Schematron Semantic Validation",
            "schemaVersion":"ISO Schematron 2016",
            "queryBinding":"jsonpath",
            "defaultPhase":"phaseid1",
            "phase": [
           {
                   "id":"phaseid1",
                   "active":["patternid1"]
           },
           {
                   "id":"phaseid2",
                   "active":["patternid2"]
           }],

            "pattern":[
                   {
                     "id":"patternid1",
                     "title":"Interest Rate Pattern",
                     "abstract":false,
                     "rule":[
                          {
                           "id":"RateRule1",
                           "abstract":false,
                           "context": "$.loan_data.loans.*",
                           "assert":[
                          {
                          "id":"assertidINT21",
                          "test": "(jp.query(contextNode,'$..interest_rate')) >= 3.75",
                          "message": "assertidINT21: Interest Rate cannot be less than 3.75 %"
                          },
                          {
                          "id":"assertidPRI21",
                          "test": "(jp.query(contextNode,'$..prime_rate')) <= 5",
                          "message": "assertidPRI21: Prime Rate cannot be more than 5 %"
                   }]}]},
```

```
                        {
                          "id":"patternid2",
                          "title":"Traditional Loan Max",
                          "abstract":false,
                          "rule":[
                          {
                          "id":"TRADMaxRule1",
                          "abstract":false,
                          "context": "$.loan_data.loans[?(@.loan_type === 'Traditional')]",

                          "assert":[
                                  {
                                  "id":"assertidINT21",
                                  "test": "(jp.query(contextNode,'$..amount')) <= 1000000",
                                  "message": "Assert 1: Traditional loan cannot be more than $1MM "
                                  }
                          ]}]}]}
    }
```

Let's run this example with good data.
Run: **node $JSONValidator -i loandata_data_multi_good1.json -r loandata_rules_multi.json**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/main_features_examples/multiple_elements_example
$ node $JSONValidator -i loandata_data_multi_good1.json -r loandata_rules_multi.json
Starting Semantic Validation .........
Parsing Pattern: patternid1
Parsing Pattern: patternid2
2 Pattern(s) Requested. 2 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Let's run the example with only one phase (phaseid1).
Run: **node $JSONValidator -i loandata_data_multi_good1.json -r loandata_rules_multi.json phaseid1**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/main_features_examples/multiple_elements_example
$ node $JSONValidator -i loandata_data_multi_good1.json -r loandata_rules_multi.json phaseid1
Starting Semantic Validation .........
Parsing Pattern: patternid1
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Same example with the second phase (phaseid2).
Run: **node $JSONValidator -i loandata_data_multi_good1.json -r loandata_rules_multi.json phaseid2**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/main_features_examples/multiple_elements_example
$ node $JSONValidator -i loandata_data_multi_good1.json -r loandata_rules_multi.json phaseid2
Starting Semantic Validation .........
Parsing Pattern: patternid2
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

The example with bad data where prime rate is expected to be 5 % or less.
Run: **node $JSONValidator -i loandata_data_multi_bad1.json -r loandata_rules_multi.json**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/main_features_examples/multiple_elements_example
$ node $JSONValidator -i loandata_data_multi_bad1.json -r loandata_rules_multi.json
Starting Semantic Validation .........
Parsing Pattern: patternid1
Parsing Pattern: patternid2
2 Pattern(s) Requested. 2 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY ENABLING DEBUG WITH -d
 OPTION ****
Completed Semantic Validation .........
```

The example with bad data where traditional loan is expected to be less than $1M but the file contains $3M.
Run: **node $JSONValidator -i loandata_data_multi_bad2.json -r loandata_rules_multi.json**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/main_features_examples/multiple_elements_example
$ node $JSONValidator -i loandata_data_multi_bad2.json -r loandata_rules_multi.json
Starting Semantic Validation .........
Parsing Pattern: patternid1
Parsing Pattern: patternid2
2 Pattern(s) Requested. 2 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY ENABLING DEBUG WITH -d
 OPTION ****
Completed Semantic Validation .........
```

3.5 Loan Data Main Example

This example contains many use cases that exhibit various capabilities of Schematron specification. Run it with various combinations and permutations to activate various phases.

Business Rules:

Semantic Validation

- If loan type is FHA, amount can't exceed 500K
- If loan type is FHA, mip_rate can't be 0 or less
- If loan type is traditional, amount can't exceed 1MM
- If loan type is jumbo, the amount can't be less than 1M
- Interest rate should at least be .25 % more than prime rate
- If loan type is not FHA, down payment can't be less than 20%
- If origination id is 'branch' then 'branch_id' should be present
- Customer id under loan and customer id under customer should match


```
{
"loan_data":{
"loans":[
 {
        "loan_id":"1234567",
        "loan_type":"FHA",
        "customer_id":"JD689457",
        "data_time":"20100601120000",
        "amount":500000,
        "interest_rate":3.75,
        "prime_rate":3.25,
        "mip_rate":1.5,
        "down_payment":5,
        "loan_restricted":false,
        "escrow":true,
        "origination_id":"branch",
        "branch_id":"5463",
        "electronic":true,
        "email":"john.doe@gmail.com",
        "customer":{
                "customer_id":"JD689457",
                "customer_fname":"John",
                "customer_lname":"Doe",
                "customer_address":" 4 Way Loop, New York, NY 10038"
    }
}]}
```


Data:

The data for this example is found in below folder:

```
>> $JSONTRON_HOME/examples/main_features_examples/loandata-main
      Loandata-rules-main.json
      Loandata-main.json
      Loandata-main-bad1.json
```

Specifying the semantic constraints based on the above business rules

Semantic Validation

- If loan type is FHA, amount can't exceed 500K
- If loan type is FHA, mip_rate can't be 0 or less
- If loan type is traditional, amount can't exceed 1MM
- If loan type is jumbo, the amount can't be less than 1M
- Interest rate should at least be .25 % more than prime rate
- If loan type is not FHA, down payment can't be less than 20%
- If origination id is 'branch' then 'branch_id' should be present
- Customer id under loan and customer id under customer should match

```
{
    "id":"rule22",
    "abstract":false,
    "context": "$.loan_data.loans[?(@.loan_type === 'FHA')]",
    "assert":[
      {
        "id":"assertid221",
        "test": "jp.query(contextNode,'$..amount') <= 500000",
        "message": "Assert 221: For FHA Loan, Amount cannot exceed $500K"
      },
      {
        "id":"assertid222",
        "test": "jp.query(contextNode,'$..mip_rate') > 0",
        "message": "Assert 222: For FHA loans, You must have MIP (Mortgage Insurance Premium)"
      }

  "context": "$.loan_data.loans[?(@.loan_type === 'Traditional')]",
  "assert":[
    {
      "id":"assertid31",
      "test": "jp.query(contextNode,'$..amount') <= 1000000",
      "message": "Assert 31: For Traditional Loan, Amount cannot exceed $1MM"
    }

    "context": "$.loan_data.loans[?(@.loan_type === 'Jumbo')]",
    "assert":[
      {
        "id":"assertid41",
        "test": "jp.query(contextNode,'$..amount') >= 1000000",
        "message": "Assert 41: For Jumbo Loan, Amount cannot be less than $1MM"
      }

  "context": "$.loan_data.loans.*",
  "assert":[
    {
      "id":"assertid81",
      "test": "(jp.query(contextNode,'$..interest_rate') - jp.query(contextNode,'$..prime_rate')) >= .25",
      "message": "Assert 81: Interest Rate should be atleast .25 points more than Prime Rate"
    }

  "context": "$.loan_data.loans[?(@.loan_type != 'FHA')]",
  "assert":[
    {
      "id":"assertid251",
      "test": "jp.query(contextNode,'$..down_payment') >= 20",
      "message": "Assert 251: For non-FHA loans, Minimum 20% downpayment is required"
    }

  "context": "$.loan_data.loans[?(@.origination_id === 'branch')]",
  "assert":[
    {
      "id":"assertid261",
      "test": "jp.query(contextNode,'$..branch_id') != ''",
      "message": "Assert 261: Missing Branch ID"
    }

  "context": "$.loan_data.loans.*",
  "assert":[
    {
      "id":"assertid271",
      "test": "jp.query(contextNode,'$[?(@.customer_id == @.customer.customer_id)]') != false",
      "message": "Assert 271: Customer ID mismatch"
    }
```

Run validation with good data.
Run: **node $JSONValidator -i loandata-main.json -r loandata-rules-main.json**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/main_features_examples/loandata-main
$ node $JSONValidator -i loandata-main.json -r loandata-rules-main.json
Starting Semantic Validation .........
Parsing Pattern: FHA_max_pattern
Parsing Pattern: FHA_MIP_pattern
Parsing Pattern: Traditional_max_pattern
Parsing Pattern: Jumbo_min_pattern
Parsing Pattern: NonFHA_down_pattern
Parsing Pattern: Branch_ID_pattern
Parsing Pattern: Customer_ID_pattern
Parsing Pattern: Interest_Min_pattern
8 Pattern(s) Requested. 8 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Run validation with bad data.
Run: **node $JSONValidator -i loandata-main-bad1.json -r loandata-rules-main.json**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/main_features_examples/loandata-main
$ node $JSONValidator -i loandata-main-bad1.json -r loandata-rules-main.json
Starting Semantic Validation .........
Parsing Pattern: FHA_max_pattern
Parsing Pattern: FHA_MIP_pattern
Parsing Pattern: Traditional_max_pattern
Parsing Pattern: Jumbo_min_pattern
Parsing Pattern: NonFHA_down_pattern
Parsing Pattern: Branch_ID_pattern
Parsing Pattern: Customer_ID_pattern
Parsing Pattern: Interest_Min_pattern
8 Pattern(s) Requested. 8 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY ENABLING D
EBUG WITH -d OPTION ****
Completed Semantic Validation .........
```

3.6 Examples for phase, pattern, rule, assert, context, assert and report elements

There are multiple examples available in the examples folder to test out main features of the Schematron implementation. Go to each folder and try out various examples.

Data:

There are multiple examples for each of the construct. The data for these examples are found in folders listed below. Go to each of these folders and run the examples to see how various constraints are expressed.

Theses examples are based on two main sets of data. The loan data that have been used so far will also be used in some of the examples here. Some examples also use ‘Store’ data set as well to shed light on nuances of Schematron implementation in JSON.

All the examples with good data, bad data and rule files are located at:

```
>> $JSONTRON_HOME/examples/main_features_examples
      /assert
      /context
      /pattern
      /phase
      /rules
      /report
```

3.7 Stack Overflow Meeting Times Dilemma Example

There was an unsolved question on Stack Overflow website about Schematron JSON implementation. The question was about making sure that when scheduling meetings, end time should not be before the start time. It was posted a few years ago on the site and was still unresolved. Jsontron successfully and easily solved the problem.

The original question and our solution are available at the following location:

https://stackoverflow.com/questions/28629107/json-is-there-an-equivalent-of-schematron-for-json-and-json-schema-that-is-a

Data:

The data for Stack Overflow example is located at:

```
>> $JSONTRON_HOME/examples/stackoverflow_example

      meeting-times-rules.json
      good-time.json
      bad-time.json
```

Run with good data:
Run: **node $JSONValidator -i good_time.json -r meeting-times-rules.json**

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/stackoverflow_examp
$ node $JSONValidator -i good_time.json -r meeting-times-rules.json
Starting Semantic Validation .........
Parsing Pattern: Meetingtimings
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Run with bad data and debug on:

```
pace@ubuntu18:~/jsontron/node_modules/jsontron/examples/stackoverflow_example
$ node $JSONValidator -i bad_time.json -r meeting-times-rules.json -d
Starting Semantic Validation .........
Parsing Pattern: Meetingtimings
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPOR
T BY ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 1
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
       ruleContext: [Array],
 Amazon ssertionid: 'start_stop_meeting_chec',
       assertionTest: 'jp.query(contextNode, \'$..starttime\') < jp.query(con
textNode, \'$..endtime\')',
       message: 'Meeting cannot end before it starts',
       assertionValid: false } ],
  finalValidationReport:
   [ { schRule: [Object],
       ruleContext: [Array],
       assertionid: 'start_stop_meeting_chec',
       assertionTest: 'jp.query(contextNode, \'$..starttime\') < jp.query(con
textNode, \'$..endtime\')',
       message: 'Meeting cannot end before it starts',
       assertionValid: false } ],
  valid: false }
```

4 IBM Schematron Tutorial

This tutorial is based on IBM staff Uche Ogbuji's XML Schematron tutorial [4].

4.1 Prerequisites

This tutorial assumes knowledge of JSON, JSON Schema, JSONPath, and JavaScript. If you are not familiar with these concepts, please take some basic tutorial first.

Below are some good resources to get started with these:

- https://www.w3schools.com/js/js_json_syntax.asp
- https://json-schema.org/understanding-json-schema
- https://www.w3schools.com/js/default.asp

Good understanding of JSONPath node module is very important for understanding the examples in this tutorial. JSONPath documentation is available at:

- https://www.npmjs.com/package/jsonpath

There is an online JSONPath evaluator available at below location:

- http://jsonpath.com

About the Schematron examples in this tutorial

The original XML instance documents translated into JSON instance document. All the Schematron rules were re-written in the JSON Schematron specification language.

Data:

All the files for this tutorial are available in the following folder:

```
>> $JSONTRON_HOME/examples/ibm-test-suite
```

The data is also available online on GitHub:

- https://github.com/amer-ali/jsontron/tree/master/jsontron/examples/ibm-test-suite

The naming convention used for the example files is similar to the original tutorial.

The examples are in respective folders and named with a prefix for the particular example like:

- **`eg3_1_good1.json`** [Instance document that should pass the validation]
- **`eg3_1_bad1.json`** [Instance document that should fail the validation]
- **`eg3_1-rules.json`** [The Schematron rules file. It is also a JSON document]

Reminder: not all the examples in the original tutorial are applicable in JSON format so you will see some missing examples in this implementation.

4.2 Schematron overview and example

Data:
All the files for this tutorial are available in the following folder:

`>> $JSONTRON_HOME/examples/ibm-test-suite/4.1`

Below examples are available with core features:
- `eg4_1-rules.json` [Schematron Rules File]
- `eg4_1_good1.json` [Valid Instance Document]
- `eg4_1_bad1.json` [Invalid Instance Document]

Commands:
Good data: **`>>node $JSONValidator -i eg4_1_good1.json -r eg4_1-rules.json`**
Bad data: **`>>node $JSONValidator -i eg4_1_bad1.json -r eg4_1-rules.json`**

Schematron is useful in the scenario where grammar/syntax-based schema languages like JSON Schema [5] are not suitable. It helps you define co-constraints.

The examples demonstrate the power of the Schematron using a hypothetical organization that handles the publishing of technical documents. The editors write the rules to ensure the submitted documents meet the editorial requirements.

If you have not done so, please read Section 2 and setup the 'jsontron' validator to run these examples.

Below is a sample JSON document that needs to be validated:

```
{
  "doc": {
        "prologue":{},
        "section":{}
  }
}
```

A sample rules file that ensures prologue and section elements in the document

In this example, there is one *pattern* that contains one *rule*. The rule has a *context* expression on line 25. This expression is a *jsonpath* expression. It is basically the second argument of the *jsonpath* query method. The validator will run this expression as below during runtime:

```
1 {
2 "schema":{
3
4     "id":"eg4_1",
5     "title":"Technical document schema",
6     "schemaVersion":"ISO Schematron 2016",
7     "queryBinding":"jsonpath",
8     "defaultPhase":"phaseid1",
9
10     "phase":[
11        {
12            "id":"phaseid1",
13            "active":["Major_elements"]
14        }
15    ],
16
17    "pattern":[
18      {
19        "id":"Major_elements",
20        "title":"Major elements Pattern",
21        "rule":[
22          {
23                "id":"Major_elementss_rule",
24                "abstract":false,
25                "context": "$..doc",
26                "assert":[
27                  {
28                    "id":"Major_elements_assert_prologue",
29                    "test":"jp.query(contextNode, '$..[?(@.prologue)]').length > 0",
30                    "message": "element must have a prologue"
31                  },
32                              {
33                    "id":"Major_elements_assert_section",
34                    "test":"jp.query(contextNode, '$..[?(@.section)]').length > 0",
35                    "message": "element must have a section"
36                  }
37                ]
38          }]}]}}
39
```

In this example, there is one pattern that contains one rule. The rule has a context expression on line 25. This expression is a *jsonpath* expression. It is basically the second argument of the *jsonpath* query method. The validator will run this expression as below during runtime:

```
var contextNode = jp.query(instanceDocument, "$..doc");
```

Please note that “contextNode” is used as a keyword in ‘jsontron’ implementation. It stores the node set that returns as a result of running ‘jsonpath’ query. The assertion tests are then run against each node of this node set. One critical thing to note is that *jsontron* encloses the result of the context expression in an array []. This is to avoid the inconsistency due to the fact that the *jsonpath* can sometimes return an object {}.

For more details, see sections on *Context* and *Assertion*.

4.3 Basics of rules, patterns, and assertions

Data: All the files for this tutorial are available in the following folder:

```
>> $JSONTRON_HOME/examples/ibm-test-suite/4.5
```

A pattern is a collection of rules. A rule is a collection of assertions. The assertion contains tests conducted against the nodes returned from running the context expression. For each pattern, the rules engine selects the nodes that will trigger the tests. Each pattern can contain multiple rules and each rule can contain multiple assertions. For the example above, the rules engine will select the ‘doc’ element by running below jsonpath query:

```
var contextNode = jp.query(schInstance, "$..doc");
```

and will return the following node set:

```
[ { prologue: {}, section: [ {}, {} ] } ]
```

Then the first assertion will check to see if the prologue element is present:

```
var prologue = jp.query(contextNode, '$..[?(@.prologue)]').length > 0;
```

It will return below output:

```
true
```

It will then run the second assertion checking the presence of ‘section’ element:

```
var section = jp.query(contextNode, '$..[?(@.section)]').length > 0;
```

```
true
```

If both assertions are true, that means the document is valid overall. You will get the following message:

```
**** THIS INSTANCE IS SEMANTICALLY VALID ****
```

Now if we have a bad document that has missing ‘prologue’ element:

```
{
  "doc": {
        "section":{}
  }
}
```

The second assertion will fail, as the prologue is missing, so the overall validation will fail.

```
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY ENABLING DEBUG WITH -d OPTION ****
```

Executing the sample schema

The sample schema is actually the example 4.5 from the IBM test suite. Below are the files:

- `eg4_5-rules.json` [Schematron Rules File]

- `eg4_5_good1.json` [Valid Instance Document]
- `eg4_5_bad1.json` [Invalid Instance Document. Missing ‘prologue’ element]

We are now ready to run the example. Follow the below steps to run the example.

Make sure you are in the right example folder:
`$JSONTRON_HOME/examples/ibm-test-suite/4.5`

```
>node $JSONValidator -i eg4_5_good1.json -r eg4_5-rules.json

Starting Semantic Validation .........
Parsing Pattern: Major_elements
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

where -i is the instance document and the -r is the rules document.

Now if you run with an invalid document that has ‘prologue’ element missing:

```
>node $JSONValidator -i eg4_5_bad1.json -r eg4_5-rules.json

Starting Semantic Validation .........
Parsing Pattern: Major_elements
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
```

If you run it in debug by adding -d at the end of the command, you will get below output:

```
>node $JSONValidator -i eg4_5_bad1.json -r eg4_5-rules.json -d

Starting Semantic Validation .........
Parsing Pattern: Major_elements
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 2
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
```

```
        ruleContext: [Object],
        assertionid: 'Major_elements_assert_prologue',
        assertionTest: 'jp.query(contextNode, \'$..[?(@.prologue)]\').length > 0',
        message: 'element must have a prologue',
        assertionValid: false },
      { schRule: [Object],
        ruleContext: [Object],
        assertionid: 'Major_elements_assert_section',
        assertionTest: 'jp.query(contextNode, \'$..[?(@.section)]\').length > 0',
        message: 'successful',
        assertionValid: true } ],
  finalValidationReport:
   [ { schRule: [Object],
        ruleContext: [Object],
        assertionid: 'Major_elements_assert_prologue',
        assertionTest: 'jp.query(contextNode, \'$..[?(@.prologue)]\').length > 0',
        message: 'element must have a prologue',
        assertionValid: false } ],
  valid: false }
```

As mentioned in the original tutorial, the examples in this tutorial are very simplistic. Same results can be achieved using the syntax-based schema languages like JSONSchema. The tutorial is meant for just getting familiar with features of Schematron so that more complicated rules can be developed later.

You can experiment with various files in the same folder (**4.5**) to test the various scenarios. Here are the examples in the folder:

- `4.5/eg4_5_good2.json` [Valid Instance Document with multiple 'section' elements]
- `4.5/eg4_5_bad2.json` [Invalid Instance Document. Missing 'section' element]
- `4.5/eg4_5_bad3.json` [Invalid Instance Document. Missing both 'section' and 'prologue' elements]

Now let's look at other examples one by one. We will follow the same sequence as original IBM tutorial to incrementally introduce various concepts.

Verifying that a particular element is a root

In example 3.1 we verify that 'doc' is always the root element. Below are the files for this example.

Data: All the files for this tutorial are available in the following folder:

`>> $JSONTRON_HOME/examples/ibm-test-suite/3.1`

- `eg3_1-rules.json` [Schematron Rules File]
- `eg3_1_good1.json` [Valid Instance Document]
- `eg3_1_bad1.json` [Invalid Instance Document. The root element is not 'doc']

Input valid instance document: **`eg3_1_good1.json`**

```
{

	"doc": {}

}
```

Input rules file: **`eg3_1-rules.json`**

```
{
"schema":{
	"id":"eg3_1",
	"title":"Technical document schema",
	"schemaVersion":"ISO Schematron 2016",
	"queryBinding":"jsonpath",
	"defaultPhase":"phaseid1",

	 "phase":[
	{
		"id":"phaseid1",
		"active":["Document_root"]
	}
		],

	"pattern":[
	    {"id":"Document_root",
	     "title":"pattern title",
	     "rule":[
	   {
	       "id":"doc_root",
	       "abstract":false,
	       "context": "$",
	        "assert":[
```

```
                  {
                     "id":"doc_root_assert",
                     "test":"Object.keys(jp.parent(contextNode, '$.*')[0]) == 'doc'",
                     "message": "Root element should be 'doc'."
                  }
      ]}]}]}}
```

Now let's run the example using jsontron:

```
>node $JSONValidator -i eg3_1_good1.json -r eg3_1-rules.json

Starting Semantic Validation .........
Parsing Pattern: Document_root
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Now with invalid instance document: eg3_1_bad1.json

```
{

      "bogus": {}

}
```

```
>node $JSONValidator -i eg3_1_bad1.json -r eg3_1-rules.json

Starting Semantic Validation .........
Parsing Pattern: Document_root
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
```

There are other examples you can experiment with:

- `eg3_1_bad2.json` [Invalid Instance Document. 'doc' element is not at the root]

Below are the *context* and *test* expressions that we will examine closely:

```
"context": "$",
"assert":[
  {
     "id":"doc_root_assert",
     "test":"Object.keys(jp.parent(contextNode, '$.*')[0]) == 'doc'",
     "message": "Root element should be 'doc'."
  }
,
```

Let's take a closer look at the *context* and *test* expressions. In the *context* expression, we just select the root element. However, the *test* expression is a little complex. Basically, it uses combination of 'jsonpath' parent() method and JavaScript Object.keys() method. Since this implementation allows using plain JavaScript expression due to the limitation of 'jsonpath' query language, therefore, you can achieve the same result in 'test' expression through multiple ways and using different methods and expressions. You will see some variations in later examples.

A little deeper anatomy of the context and test expressions.

Context expression '**$**' behind the scene translates into the following call:

```
var contextNode = jp.query(schInstance, "$");
```

It will return the following *context* node set

```
[ { doc: {} } ]
```

Note that the returned node-set is an array [] instead of object {}. As explained in detail before, this is due to the fact that jsonpath wraps the output in an array for consistency.

Now in order to test that the root is actually a *doc* element, we have to do a few things:

1. First retrieve the root element again by running jsonpath method *parent()* on the *contextNode* that was returned from the *context* expression.

```
var root = jp.parent(contextNode, '$.*');
```

```
[{ doc: {} }]
```

Notice that we used the jsonpath method *parent()* instead of a query method. We can run the query() method as well and we will show that little later.

2. Now using the plain old JavaScript array indexing, access the first element in the node set.

```
[{ doc: {} }].[0]
```

This will return the 'doc' object:

```
{ doc: {} }
```

Now retrieve the key of the object using the JavaScript Object.keys() method.

```
Object.keys({ doc: {} })
```

```
'doc' == 'doc'
```

This comparison will return 'true' in this case. So, combining all this in one statement:

```
test = Object.keys(jp.parent(contextNode, '$.*')[0]) == 'doc';
```

As mentioned above you can use jsonpath method *query()* as well to achieve the same result like below:

```
test = Object.keys(jp.query(contextNode, '$')[0][0]) == 'doc';
```

Note that when using method *query()* we have to use another index variable to access the 'doc' object. This is because of the fact that when jp.query() ran on *contextNode*, it added another wrapper around it so the object became [[{ doc: {} }]], therefore, we will have to use array[0][0] to access the doc object.

```
{ doc: {} }
```

It seems a bit complicated and takes some time to get used to, similar to XPath in XML, but once you get familiar with it, you will understand that it provides a lot of flexibility to express really sophisticated validation rules.

This is the hardest part to grasp for the new users, so take your time to understand this. If need be, go back to *jsonpath* and *JavaScript* tutorials as mentioned in pre-requisite section.

But the good news is that once you master these concepts, the rest of the tutorial will be a lot easier.

Validating the presence of elements

You can validate that certain elements are present. This schema checks that ‘doc’ elements have both ‘prologue’ and ‘section’ elements. This is similar to the sample schema we explained at the beginning of the tutorial. But here we will use simple test expressions. Below are the files used in this example

Data: All the files for this tutorial are available in the following folder:

>> $JSONTRON_HOME/examples/ibm-test-suite/3.2

- **eg3_2-rules.json** [Schematron Rules File]
- **eg3_2_good1.json** [Valid Instance Document]
- **eg3_2_bad1.json** [Invalid Instance Document. ‘prologue’ element is missing]
- **eg3_2_bad2.json** [Invalid Instance Document. ‘section’ element is missing]
- **eg3_2_bad3.json** [Invalid Instance Document. Both ‘prologue’ and ‘section’ elements are missing]

```
{
  "doc": {
             "prologue":{},
             "section":{}
        }
}
```

The rule looks like below:

```
"pattern":[
  {
    "id":"Major_Elements",
    "title":"Major Elements",
    "rule":[
      {
            "id":"prologue_section",
            "abstract":false,
            "context": "$..doc",
            "assert":[
              {
                 "id":"doc_prologue_assert",
                 "test":"contextNode[0].prologue != null",
                 "message": "The 'Doc' element should have 'prologue' child."
              },
              {
                 "id":"doc_section_assert",
                 "test":"contextNode[0].section != null",
                 "message": "The 'Doc' element should have 'section' child."
              }
      ......]
```

In this example, the context expresses “$..doc” will return the ‘doc’ element as below:

```
[ { prologue: {}, section: {} } ]
```

Next, we simply check if the first element of the array [0] has the *prologue* and *section* members. Internally two tests will be run, one for 'prologue' and one for 'section'

```
test = contextNode[0].prologue != null

test  = contextNode[0].section != null
```

Now let's run the example using jsontron:

```
>node $JSONValidator -i eg3_2_good1.json -r eg3_2-rules.json

Starting Semantic Validation .........
Parsing Pattern: Major_Elements
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Now with the bad data:

```
>node $JSONValidator -i eg3_2_bad1.json -r eg3_2-rules.json -d

Starting Semantic Validation .........
Parsing Pattern: Major_Elements
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Validations: 2
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { assertionid: 'doc_prologue_assert',
       assertionTest: 'contextNode[0].prologue != null',
       message: 'The \'Doc\' element should have \'prologue\' child.',
       assertionValid: false },
     { assertionid: 'doc_section_assert',
       assertionTest: 'contextNode[0].section != null',
       message: 'successful',
       assertionValid: true } ],
  finalValidationReport:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'doc_prologue_assert',
       assertionTest: 'contextNode[0].prologue != null',
```

```
        message: 'The \'Doc\' element should have \'prologue\' child.',
        assertionValid: false } ],
  valid: false }
```

Verifying that elements are where they are expected

To validate that an element appears only in a certain place, use this schema to check that the only ‘doc’ element is the root. This is similar to example 3_1 but in that example, we were verifying that the root element is a doc. But there could be other doc sub-elements. In this example, we are verifying that ‘doc’ element is allowed only as root. There shouldn’t be any other ‘doc’ element in the document other than root.

Data: All the files for this tutorial are available in the below folder:

`>> $JSONTRON_HOME/examples/ibm-test-suite/3.3`

The example files here are:

- **`eg3_3-rules.json`** [Schematron Rules File]
- **`eg3_3_good1.json`** [Valid Instance Document]
- **`eg3_3_bad1.json`** [Invalid Instance Document. The ‘doc’ is root element but there is another ‘doc’ sub-element]

The valid file:

```
{
  "doc": { "ok":{} }
}
```

The rule file snippet:

```
"pattern":[
  {
    "id":"Extraneous_docs",
    "title":"Extraneous Docs",
    "rule":[
      {
            "id":"extraneous_doc_rule",
            "abstract":false,
            "context": "$",
            "assert":[
              {
                 "id":"extraneous_doc_assert",
                 "test":"jp.query(contextNode,'$..doc').length ==1 && contextNode[0] == jp.parent(contextNode, '$..doc')",
                 "message": "The 'doc' element is only allowed at the document root."
              }
            ]
      }]}]}}
```

```
>node $JSONValidator -i eg3_3_good1.json -r eg3_3-rules.json

Starting Semantic Validation .........
Parsing Pattern: Extraneous_docs
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
```

```
Completed Semantic Validation .........
```

Now invalid document where there is a nested ‘doc’ element:

```
{
  "doc": { "doc":{} }
}
```

```
>node $JSONValidator -i eg3_3_bad1.json -r eg3_3-rules.json

Starting Semantic Validation .........
Parsing Pattern: Extraneous_docs
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
```

Output with debug enabled (-d)

```
>node $JSONValidator -i eg3_3_bad1.json -r eg3_3-rules.json -d

Starting Semantic Validation .........
Parsing Pattern: Extraneous_docs
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 1
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'extraneous_doc_assert',
       assertionTest: 'jp.query(contextNode,\'$..doc\').length ==1 && contextNode[0]
== jp.parent(contextNode, \'$..doc\')',
       message: 'The \'doc\' element is only allowed at the document root.',
       assertionValid: false } ],
  finalValidationReport:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'extraneous_doc_assert',
       assertionTest: 'jp.query(contextNode,\'$..doc\').length ==1 && contextNode[0]
== jp.parent(contextNode, \'$..doc\')',
        message: 'The \'doc\' element is only allowed at the document root.',
       assertionValid: false } ],
  valid: false }
```

Validating for a certain number of elements

You can validate whether there is a specific number of a particular element present.

To make it easier for readers to find the article in relevant contexts, the journal editors want to be sure that articles have at least three keywords in the prologue.

This schema enforces a minimum of three *keyword* children of the *prologue* element.

Data: All the files for this tutorial are available in the below folder:

>> $JSONTRON_HOME/examples/ibm-test-suite/3.6

Files in this example:

- **eg3_6-rules.json** [Schematron Rules File]
- **eg3_6_good1.json** [Valid Instance Document]
- **eg3_6_bad1.json** [Invalid Instance Document. Only has two keywords]

Valid document:

```
{
  "doc": {
    "prologue": {
      "title": "Faster than light travel",
      "subtitle": "From fantasy to reality",
      "keyword": [
        "tachyon",
        "higgs",
        "propulsion"
      ]
    }
  }
}
```

Rules file snippet. Simply select the keywords in the context and then count in the test.

```
"rule":[
  {
        "id":"Minimum_keywords_rule",
        "abstract":false,
        "context": "$..prologue.keyword",
        "assert":[
          {
              "id":"Minimum_keywords_assert",
              "test":"contextNode[0].length > 2",
              "message": "At least three keywords are required."
          }
```

You will get following validation result when running with good data.

```
>node $JSONValidator -i eg3_6_good1.json -r eg3_6-rules.json

Starting Semantic Validation .........
Parsing Pattern: Minimum_keywords
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Now invalid document with only two keywords instead of three.

```
>node $JSONValidator -i eg3_6_bad1.json -r eg3_6-rules.json

Starting Semantic Validation .........
Parsing Pattern: Minimum_keywords
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
```

Run the bad data example with debug option:

```
>node $JSONValidator -i eg3_6_bad1.json -r eg3_6-rules.json -d

Starting Semantic Validation .........
Parsing Pattern: Minimum_keywords
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 1
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Minimum_keywords_assert',
       assertionTest: 'contextNode[0].length > 2',
       message: 'At least three keywords are required.',
       assertionValid: false } ],
  finalValidationReport:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Minimum_keywords_assert',
       assertionTest: 'contextNode[0].length > 2',
```

```
      message: 'At least three keywords are required.',
      assertionValid: false } ],
  valid: false }
```

Validating presence and value of attributes

You can validate that an attribute appears, or that it has a certain value. This schema checks that an *author* element (child of the *prologue*) has an *e-mail* attribute and a *member* attribute. The latter indicates whether or not an author is a member of the technical association, and this schema checks that its value is “yes” or “no”.

Data: All the files for this tutorial are available in the following folder:

```
>> $JSONTRON_HOME/examples/ibm-test-suite/3.7
```

The files for this example are:

- **eg3_7-rules.json** [Schematron Rules File]
- **eg3_7_good1.json** [Valid Instance Document]
- **eg3_7_bad1.json** [Invalid Instance Document. Missing ‘email, attribute]

Valid document:

```
{
  "doc": {
    "prologue": {
      "title": "Faster than light travel",
      "subtitle": "From fantasy to reality",
      "author": {
        "member": "yes",
        "email": "cemereuwa@nasa.gov",
        "name": "Chikezie Emereuwa"
      }
    }
  }
}
```

Rules snippet. Two assertions. First checks ‘email’. Other, checks ‘member’ and ‘yes/no’.

```
"context": "$..author",
"assert":[
  {
    "id":"Author_attributes_assert_email",
    "test":"('email' in contextNode[0])",
    "message": "Author must have e-mail attribute."
  },
  {
    "id":"Author_attributes_assert_member",
    "test":"('member' in contextNode[0]) && ((jp.query(contextNode, '$..member'))[0] == ('yes')||(jp.query(contextNode, '$..member'))[0] == ('no'))",
    "message": "author must have member attribute with 'yes' or 'no' value."
  }
```

Let's run the example with valid data:

```
>node $JSONValidator -i eg3_7_good1.json -r eg3_7-rules.json

Starting Semantic Validation .........
Parsing Pattern: Author_attributes
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Now with an invalid document that has missing attribute 'email':

```
>node $JSONValidator -i eg3_7_bad1.json -r eg3_7-rules.json -d

Starting Semantic Validation .........
Parsing Pattern: Author_attributes
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 2
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Author_attributes_assert_email',
       assertionTest: '(\'email\' in contextNode[0])',
       message: 'Author must have e-mail attribute.',
       assertionValid: false },
     { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Author_attributes_assert_member',
```

```
       assertionTest: '(\'member\' in contextNode[0]) && ((jp.query(contextNode,
\'$..member\'))[0] == (\'yes\')||(jp.query(contextNode, \'$..member\'))[0] ==
(\'no\'))',
       message: 'successful',
       assertionValid: true } ],
  finalValidationReport:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Author_attributes_assert_email',
       assertionTest: '(\'email\' in contextNode[0])',
       message: 'Author must have e-mail attribute.',
       assertionValid: false } ],
  valid: false }
```

Simple validation of element content

To validate that an element has a certain value, use this schema to check that the the title element is not empty.

Data: All the files for this tutorial are available in the following folder:

```
>> $JSONTRON_HOME/examples/ibm-test-suite/3.8
```

The files for this example are:

- `eg3_8-rules.json` [Schematron Rules File]
- `eg3_8_good1.json` [Valid Instance Document]
- `eg3_8_bad1.json` [Invalid Instance Document. Empty 'title' attribute]

Valid document:

```
{
  "doc": {
    "prologue": {
      "title": "Faster than light travel",
      "subtitle": "From fantasy to reality"
    },
    "section":{}
  }
}
```

Rules file snippet. Just check the length of title's value:

```
"context": "$..title",
"assert":[
  {
     "id":"Useful_title_assert",
     "test":"contextNode[0].length > 0",
     "message": "Title may not be empty"
  }
```

Let's run the example with valid data:

```
>node $JSONValidator -i eg3_8_good1.json -r eg3_8-rules.json

Starting Semantic Validation .........
Parsing Pattern: Useful_title
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Now with an invalid document with the empty title:

```
>node $JSONValidator -i eg3_8_bad1.json -r eg3_8-rules.json

Starting Semantic Validation .........
Parsing Pattern: Useful_title
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
```

Now with an invalid document with the empty title:

```
>node $JSONValidator -i eg3_8_bad1.json -r eg3_8-rules.json -d

Starting Semantic Validation .........
Parsing Pattern: Useful_title
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 1
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Useful_title_assert',
       assertionTest: 'contextNode[0].length > 0',
       message: 'Title may not be empty',
       assertionValid: false } ],
  finalValidationReport:
   [ { schRule: [Object],
       ruleContext: [Object],
```

```
      assertionid: 'Useful_title_assert',
      assertionTest: 'contextNode[0].length > 0',
      message: 'Title may not be empty',
      assertionValid: false } ],
  valid: false }
```

Validating exclusivity of elements

You can validate that no unwanted elements are present. This schema checks that author elements only have a name, bio, and affiliation elements as children.

Data: All the files for this tutorial are available in the following folder:

```
>> $JSONTRON_HOME/examples/ibm-test-suite/3.9
```

The files are:

- `eg3_9-rules.json` [Schematron Rules File]
- `eg3_9_good1.json` [Valid Instance Document]
- `eg3_9_bad1.json` [Invalid Instance Document. Has 'age' as extra child for author]

Valid document:

```
{
  "doc": {
    "prologue": {
      "title": "Faster than light travel",
      "subtitle": "From fantasy to reality",
      "author": {
        "name": "Chikezie Emereuwa",
        "bio": "Chikezie Emereuwa is a quantum engineer and NASA researcher",
        "affiliation": "NASA"
      }
    },
    "section":{}
  }
}
```

Rules snippet. The context expression selects the author elements. The test expression then counts the number of children of author element and checks for the name, bio, and affiliation. If the count doesn't match that means there is an extraneous element.

```
"context": "$..author",
"assert":[
{
"id":"Author_elements_assert",

"test":"(jp.query(contextNode, '$..bio').length + jp.query(contextNode,
'$..affiliation').length + jp.query(contextNode, '$..name').length) ==
jp.query(contextNode, '$[0].*').length",
```

```
"message": "Only 'name', 'bio' and 'affiliation' elements are allowed as children of
'author'"
}
```

Let’s run the example with jsontron with valid data:

```
>node $JSONValidator -i eg3_9_good1.json -r eg3_9-rules.json

Starting Semantic Validation .........
Parsing Pattern: Author_elements
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Now with an invalid document that has an extra ‘age’ element:

```
>node $JSONValidator -i eg3_9_bad1.json -r eg3_9-rules.json -d

Starting Semantic Validation .........
Parsing Pattern: Author_elements
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 1
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Author_elements_assert',
       assertionTest: '(jp.query(contextNode, \'$..bio\').length +
jp.query(contextNode, \'$..affiliation\').length + jp.query(contextNode,
\'$..name\').length) == jp.query(contextNode, \'$[0].*\').length',
       message: 'Only \'name\', \'bio\' and \'affiliation\' elements are allowed as
children of \'author\'',
       assertionValid: false } ],
  finalValidationReport:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Author_elements_assert',
       assertionTest: '(jp.query(contextNode, \'$..bio\').length +
jp.query(contextNode, \'$..affiliation\').length + jp.query(contextNode,
\'$..name\').length) == jp.query(contextNode, \'$[0].*\').length',

       message: 'Only 'name', 'bio' and 'affiliation' elements are
allowed as children of 'author'',
```

```
      assertionValid: false } ],
  valid: false }
```

4.4 Reports and communications control

This section in the original tutorial mainly discusses some optional features of Schematron that were out of scope for this particular implementation. However, the core rules of these examples have still been implemented. In future, when the optional features are implemented then these examples can easily be tweaked accordingly.

Data: All the files for this tutorial are available in the following folder:

>> $JSONTRON_HOME/examples/ibm-test-suite/4.1

Below examples are available with core features:

- **eg4_1-rules.json** [Schematron Rules File]
- **eg4_1_good1.json** [Valid Instance Document]
- **eg4_1_bad1.json** [Invalid Instance Document]

Valid document:

```
{
  "doc": {
    "prologue": {
      "title": "Faster than light travel",
      "subtitle": "From fantasy to reality",
      "author": [
        {
          "member": "yes",
          "email": "cemereuwa@nasa.gov",
          "name": "Chikezie Emereuwa"
        },
        {
          "member": "yes",
          "email": "okey.agu@navy.mil",
          "name": "Okechukwu Agu"
        }
      ]
    },
    "section":{}
  }
}
```

This example actually has a better test than the original tutorial. In the original tutorial, it only checks if it contains '.mil' as there is no XPath function equivalent of endsWith(). Here we test if the email ends with '.mil' with JavaScript method endsWith().

```
                 "context": "$..author..email",
                 "assert":[
                   {
                      "id":"Military_authors_assert",
                      "test":"contextNode[0].endsWith('mil')",
                      "message": "Author appears to be military personnel"
```

Example 4_2 implements the rule to check the presence of elements

Data: All the files for this tutorial are available in the following folder:

>> $JSONTRON_HOME/examples/ibm-test-suite/4.2

This example tests the presence of the **'link'** in the text. Below are the files for it:

- **eg4_2-rules.json** [Schematron Rules File]
- **eg4_2_good1.json** [Valid Instance Document]
- **eg4_2_bad1.json** [Invalid Instance Document]

Valid document has a link:

```
{
  "doc": {
       "prologue":{},
    "section": {
      "text":"Placeholder for the emphasis text",
      "emphasis": {
        "link": "http://nasa.gov/ftl/paper.xml",
        "content": "actual content"
      }
    }
  }
}
```

The context expression selects all elements in the document. Then the 'test' checks to see if any of the elements is a 'link':

```
                 "context": "$..*",
                 "assert":[
                   {
                      "id":"Report_links_assert",
                      "test":"((jp.query(contextNode, '$..link')).length > 0)",
                      "message": "element has a link."
```

Example 4_3 implements the rule to check the sequence of elements.

Data: All the files for this tutorial are available in the following folder:

`>> $JSONTRON_HOME/examples/ibm-test-suite/4.3`

Below are the files for it:

- **`eg4_3-rules.json`** [Schematron Rules File]
- **`eg4_3_good1.json`** [Valid Instance Document]
- **`eg4_3_bad1.json`** [Invalid Instance Document]

The rule is that *title* must be immediately followed by *subtitle*. Below is the valid document where *subtitle* follows the *title* element:

```
{
  "doc": {
    "prologue": {
      "title": "Faster than light travel",
      "subtitle": "From fantasy to reality"
    },
    "section":{}
  }
}
```

The *context* expression first selects the *prologue* element. Then, Object.keys() JavaScript method is leveraged. Then the sequence is verified by checking the index of the subtitle and title.

```
                "context": "$..prologue",
                "assert":[
                  {
                     "id":"Title_with_subtitle_assert",
                     "test":"(((Object.keys(contextNode[0])).indexOf('subtitle')) -
  ((Object.keys(contextNode[0])).indexOf('title'))) == 1",

   "message": "Title must be immediately followed by subtitle"
                  }
                ]
```

```
>node $JSONValidator -i eg4_3_good1.json -r eg4_3-rules.json

Starting Semantic Validation .........
Parsing Pattern: Title_with_subtitle
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

Now let's validate the bad document where subtitle is missing altogether:

```
{
  "doc": {
    "prologue": {
      "title": "Faster than light travel"
    },
    "section":{}
  }
}
```

```
>node $JSONValidator -i eg4_3_bad1.json -r eg4_3-rules.json -d

Starting Semantic Validation .........
Parsing Pattern: Title_with_subtitle
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Validations: 1
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Title_with_subtitle_assert',
       assertionTest: '(((Object.keys(contextNode[0])).indexOf(\'subtitle\')) -
((Object.keys(contextNode[0])).indexOf(\'title\'))) == 1',
       message: 'Title must be immediately followed by subtitle',
       assertionValid: false } ],
  finalValidationReport:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Title_with_subtitle_assert',
       assertionTest: '(((Object.keys(contextNode[0])).indexOf(\'subtitle\')) -
((Object.keys(contextNode[0])).indexOf(\'title\'))) == 1',
       message: 'Title must be immediately followed by subtitle',
       assertionValid: false } ],
  valid: false }
```

Same is the result if try another bad document that has some other element after:

```
{
  "doc": {
    "prologue": {
      "title": "Faster than light travel",
      "midtitle": "Seriously",
      "subtitle": "From fantasy to reality"
```

```
    },
    "section":{}
  }
}
```

Example 4_4 is a variation of some of the earlier examples that we used to check the presence of an element.

Data: All the files for this tutorial are available in the following folder:

>> $JSONTRON_HOME/examples/ibm-test-suite/4.4

- **eg4_4-rules.json** [Schematron Rules File]
- **eg4_4_good1.json** [Valid Instance Document]
- **eg4_4_bad1.json** [Invalid Instance Document]

```
{
  "doc": {
      "prologue":{},
    "section": {
      "text":"Placeholder for the emphasis text",
      "emphasis": {
        "link": "http://nasa.gov/ftl/paper.xml",
        "content": "actual content"
      }
    }
  }
}
```

```
                "context": "$..*",
                "assert":[
                  {
                     "id":"Report_links_assert",
                     "test":"((jp.query(contextNode, '$..link')).length > 0)",
                     "message": "element has a link."
```

Example 4_5 has already been explained at the beginning of the tutorial

4.5 Intermediate Schematron features

In this section Example 5_1, Example 5_2 and Example 5_3 implement features that are out of scope for this implementation. But still, their core rules are implemented.

This section has two very interesting features that we will examine in the following sections:

Validating based on conditions in the document

"Very often you'll want to validate one part of a document based on what occurs in another part. This is something called a co-occurrence constraint. Syntax-based languages like JSON Schema cannot handle such validation at all, and RELAX NG can handle only limited examples, but Schematron provides extraordinary power for such validation tasks.

This schema checks that content by each author includes at least three sections, with the goal of encouraging longer submissions and discouraging people from padding the author list."

Data: All the files for this tutorial are available in the following folder:

```
>> $JSONTRON_HOME/examples/ibm-test-suite/5.4
```

Example 5_4 is variation implements this feature. Below are the files

- `eg5_4-rules.json` [Schematron Rules File]
- `eg5_4_good1.json` [Valid Instance Document]
- `eg5_4_bad1.json` [Invalid Instance Document]

```
        "context":"$..doc",

             "assert":[
              {
              "id":"Section_minimum_assert",
               "test":"((jp.query(contextNode, '$..section'))[0].length) >=
(((jp.query(contextNode, '$..author')).length) * 3)",
               "message": "There must be at least three sections for each author."
                }
```

"When using Schematron, don't think in terms of other schema languages, or you probably won't take advantage of all its power. Just think of what rules you'd like to express about the candidate document, and chances are you'll be able to find a way to express it using a combination of 'jsonpath' and JavaScript and thus in Schematron."

Valid example with three sections:

```
{
  "doc": {
    "prologue": {
      "title": "Faster than light travel",
      "subtitle": "From fantasy to reality",
      "author": {
        "member": "yes",
        "email": "cemereuwa@nasa.gov",
        "name": "Chikezie Emereuwa"
      }
    },
    "section":[
        {},{},{}
    ]
  }
}
```

Now let's run this example with good data using jsontron:

```
>node $JSONValidator -i eg5_4_good1.json -r eg5_4-rules.json -d

Starting Semantic Validation .........
Parsing Pattern: Section_minimum
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 1
Total Failed Assertions: 0
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Section_minimum_assert',
       assertionTest: '((jp.query(contextNode, \'$..section\'))[0].length) >=
(((jp.query(contextNode, \'$..author\')).length) * 3)',
       message: 'successful',
       assertionValid: true } ],
  finalValidationReport: [],
  valid: true }
```

Invalid example with two sections only:

```
{
  "doc": {
    "prologue": {
      "title": "Faster than light travel",
      "subtitle": "From fantasy to reality",
      "author": {
        "member": "yes",
        "email": "cemereuwa@nasa.gov",
        "name": "Chikezie Emereuwa"
      }
    },
    "section":[
        {},{}
    ]
  }
}
```

```
>node $JSONValidator -i eg5_4_bad1.json -r eg5_4-rules.json -d

Starting Semantic Validation .........
Parsing Pattern: Section_minimum
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE CONTAINS SEMANTIC VALIDATION ISSUES. PLEASE SEE FULL REPORT BY
ENABLING DEBUG WITH -d OPTION ****
Completed Semantic Validation .........
Total Errors Found: 0
Total Warnings Found: 0
Total Validations: 1
Total Failed Assertions: 1
Full Validation Report :
Report {
  errors: [],
  warnings: [],
  validations:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Section_minimum_assert',
       assertionTest: '((jp.query(contextNode, \'$..section\'))[0].length) >=
(((jp.query(contextNode, \'$..author\')).length) * 3)',
       message: 'There must be at least three sections for each author.',
       assertionValid: false } ],
  finalValidationReport:
   [ { schRule: [Object],
       ruleContext: [Object],
       assertionid: 'Section_minimum_assert',
       assertionTest: '((jp.query(contextNode, \'$..section\'))[0].length) >=
(((jp.query(contextNode, \'$..author\')).length) * 3)',
       message: 'There must be at least three sections for each author.',
       assertionValid: false } ],
  valid: false }
```

Phases

If we were to combine all the rules in this tutorial into one Schematron schema, it would be a large one. Schematron allows for modularity of schemata by allowing patterns to be organized into phases. A phase is a simple collection of patterns that are executed together. Some Schematron implementations allow you to select a particular phase to process. The following large sample schema incorporates several of the example rules from this tutorial and organizes them into phases.

Data: All the files for this tutorial are available in the below folder:

`>> $JSONTRON_HOME/examples/ibm-test-suite/5.5`

Below example files are available for testing the phase implementation:

- **`eg5_5-rules.json`** [Schematron Rules File]
- **`eg5_5_good1.json`** [Valid Instance Document]
- **`eg5_5_good2.json`** [Valid Instance Document]
- **`eg5_5_bad1.json`** [Valid Instance Document]
- **`eg5_5_bad2.json`** [Valid Instance Document]
- **`eg5_5_bad3.json`** [Valid Instance Document]
- **`eg5_5_bad4.json`** [Valid Instance Document]

Just to remind, the phases are specified as below:

```
"phase":[
        {
                "id":"quick-check",
                "active":["rightdoc"]
        },
```

and they are invoked as below from command line:

```
>node  $JSONValidator  -i eg5_5_good1.json  -r eg5_5-rules.json quick-check
```

```
{
"schema":{

        "id":"eg5_5",
        "title":"Technical document schema",
        "schemaVersion":"ISO Schematron 2016",
        "queryBinding":"jsonpath",
        "defaultPhase":"quick-check",

         "phase":[
        {
                "id":"quick-check",
                "active":["rightdoc"]
        },
        {
                "id":"full-check",
                "active":["rightdoc","extradocs","majelements"]
        },
        {
                "id":"extra-doc",
                "active":["extradocs"]
        }
    ],

        "pattern":[
                {
        "id":"majelements",
        "title":"Major elements Pattern",
        "rule":[
          {
                "id":"Major_elementss_rule",
                "abstract":false,
               "context": "$..doc",
               "assert":[
                 {
                    "id":"Major_elements_assert_prologue",
                    "test":"jp.query(contextNode, '$..[?(@.prologue)]').length > 0",
                    "message": "element must have a prologue"
                 },
                                  {
                    "id":"Major_elements_assert_section",
                    "test":"jp.query(contextNode, '$..[?(@.section)]').length > 0",
                    "message": "element must have a section"
                 }
               ]
          }]},
               {
```

```
        "id":"extradocs",
        "title":"Extraneous Docs",
        "rule":[
          {
                "id":"extraneous_doc_rule",
                "abstract":false,
                "context": "$",
                "assert":[
                  {
                     "id":"extraneous_doc_assert",
                     "test":"jp.query(contextNode,'$..doc').length ==1 && contextNode[0] ==
jp.parent(contextNode, '$..doc')",
                     "message": "The 'doc' element is only allowed at the document root."
                  }
                ]
          }]},
                {
        "id":"rightdoc",
        "title":"pattern title",
        "rule":[
          {
                "id":"rightdoc_rule",
                "abstract":false,
                "context": "$",
                "assert":[
                  {
                     "id":"doc_root_assert",
                     "test":"Object.keys(jp.parent(contextNode, '$.*')[0]) == 'doc'",
                     "message": "Root element should be 'doc'."
                  }
                ]
          }]}
          ]}}
```

- **Invoking a single phase:**
- 'quick-check' that has only one pattern

```
>node $JSONValidator -i eg5_5_good1.json -r eg5_5-rules.json quick-check

Starting Semantic Validation .........
Parsing Pattern: rightdoc
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

- **Invoking multiple patterns in one phase**

The phase 'full-check' has two patterns "rightdoc" and "majelements". Invoking full-check phase will process both patterns.

```
>node $JSONValidator -i eg5_5_good1.json -r eg5_5-rules.json full-check

Starting Semantic Validation .........
Parsing Pattern: majelements
Parsing Pattern: rightdoc
2 Pattern(s) Requested. 2 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

- **Invoking multiple phases**

The phase ‘quick-check’ has one pattern “rightdoc”, and phrase “extra-doc” has one pattern called “extradocs”. Invoking both phases will process both patterns.

```
>node $JSONValidator -i eg5_5_good1.json -r eg5_5-rules.json quick-check extra-doc


Starting Semantic Validation .........
Parsing Pattern: rightdoc
Parsing Pattern: extradocs
2 Pattern(s) Requested. 2 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

- **Invoking keyword #DEFAULT**

The default phase will be invoked by using this keyword. Remember the defaultPhase = ‘phase Name’ is mentioned in the schema.

```
>node $JSONValidator -i eg5_5_good1.json -r eg5_5-rules.json #DEFAULT


Starting Semantic Validation .........
Parsing Pattern: rightdoc
1 Pattern(s) Requested. 1 Pattern(s) Processed. 0 Pattern(s)  Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

- **Invoking all phases by keyword #ALL**

Keyword #ALL will invoke all phases. three patterns.

```
>node $JSONValidator -i eg5_5_good1.json -r eg5_5-rules.json #ALL


Starting Semantic Validation .........
Parsing Pattern: majelements
Parsing Pattern: extradocs
Parsing Pattern: rightdoc
3 Pattern(s) Requested. 4 Pattern(s) Processed. 0 Pattern(s) Ignored.
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

- **Invoking all phases by omitting the phase names**

If no phase is mentioned, all phases are processed

```
>node $JSONValidator -i eg5_5_good1.json -r eg5_5-rules.json


Starting Semantic Validation .........
Parsing Pattern: majelements
Parsing Pattern: extradocs
Parsing Pattern: rightdoc
3 Pattern(s) Requested. 3 Pattern(s) Processed. 0 Pattern(s)  Ignored..
**** THIS INSTANCE IS SEMANTICALLY VALID ****
Completed Semantic Validation .........
```

- **Handling invalid phases**

If an invalid phase is mentioned, all phases are processed. This is to ensure optimistic validation where inadvertent typo shouldn't let invalid data slip through the cracks.

```
> node $JSONValidator -i eg5_5_good1.json -r eg5_5-rules.json blah


Starting Semantic Validation .........
Parsing Pattern: majelements
Parsing Pattern: extradocs
Parsing Pattern: rightdoc
3 Pattern(s) Requested. 3 Pattern(s) Processed. 0 Pattern(s)  Ignored..
**** THIS INSTANCE IS SEMANTICALLY VALID ****
```

5 References

1. Rick Jelliffe, https://www.xml.com/authors/rick-jelliffe
2. ISO/IEC, Information technology, Document Schema Definition Languages (DSDL), Part 3: Rule-based validation, Schematron (ISO/IEC 19757-3:2016), https://www.iso.org/standard/55982.html
3. ISO/IEC, Information technology, Document Schema Definition Language (DSDL), Part 2: Regular-grammar-based validation, RELAX NG (ISO/IEC 19757-2:2008), https://www.iso.org/standard/52348.html
4. Uche Ogbuji, A hands-on introduction to Schematron, https://www6.software.ibm.com/developerworks/education/x-schematron/x-schematron-a4.pdf
5. JSON Schema, https://json-schema.org