Syntax

cswr.github.io/JsonSchema/spec/grammar

A more formal description of how JSON Schemas are built
In this section we define how every JSON Document can be built using a formal grammar. The purpose of this is to have a rigurous specification for JSON documents and to enable the community to solve ambiguities about each operator and type.

Notation

The Formal Grammar in this specification is given using a simple, visual-based Extended Backus-Naur Form (EBNF) notation, that we define below.

Each rule in the grammar defines one symbol, in the form

```
symbol := expression
```

For readability we always write non-terminal symbols in blackened font, such as **JSch** or **strRes**. The expression on the right hand side of these rules may match more than one string, and is constructed according to the following operators:

> `string` any non-blackened string that does not use `)` , `(` , `|` or `?` matches precisely against the string.

We also use brackets, as in `(expression)` , to specify that the expression inside them is a unit. We can combine units using the following operators

- `E?` : optional `E` , mathces `E` or nothing
- `A | B` : `A` or `B` , matches either `A` or `B`
- `A B` `A` concatenated with `B` , matches `A` followed by `B` . This operator has higher precedence over `|`
- `E*` : Matches zero or more ocurrences of `E` . Also has a higher precedence over `|`

Grammar

Formally we define a JSON Schema Document as a set of definitions and a JSON Schema. Each JSON Schema is threated as a set of restrictions that may apply to one or more types. To keep a clean and tidy grammar we divide each restriction in different sections, but as every grammar, the document is defined by the union of all these nested variables.

Json Documents and Schemas

Let **JDOC** be an arbitrary JSON Schema Document. We can define its syntax using the following grammar:

```
JSDoc := { ( id, )? ( defs, )? JSch }
id := "id": "uri"
defs := "definitions": { kSch (, kSch)*}
kSch := kword: { JSch }
JSch := ( res (, res)*)
res := type | strRes | numRes | arrRes | objRes | multRes | refSch | title |
description
type := "type" : ([typename (, typename)*] | typename)
typename := "string" | "integer" | "number" | "boolean" | "null" | "array" |
"object"
title := "title":  string
description := "description":  string
```

Here each **res** and **typename** must be different from each other(otherwise they would be superfluous). We must also note that each **kword** is representing a keyword that must be unique in the nest level that is occurs. Besides, **string** is any string to describe either the title or de description of the nested schema. Finally, a **uri** is any possible uri as defined in the standard. Next we specify the remaining restrictions: **strRes**, **numRes**, **arrRes**, **objRes** and **multRes**, as well as referred schemas **refSch**.

String Restrictions

```
strRes :=  minLen | maxLen | pattern
minLen := "minLength": n
maxLen := "maxLength": n
pattern := "pattern": "regExp"
```

Here **n** is a natural number and **r** is a regular expression.

Numeric Restrictions

```
numRes := min | max | multiple
min := "minimum": r (,exMin)?
exMin := "exclusiveMinimum": bool
max := "maximum": r (,exMax)?
exMax := "exclusiveMaximum": bool
multiple := "multipleOf": r   (r >= 0)
```

Here **r** is a decimal number and **bool** is either true or false.

Array Restrictions

```
 arrRes := items | additems | minitems | maxitems  | unique
 items := ( sameitems |  varitems )
 sameitems := "items": { JSch }
 varitems := "items": [{ JSch }(,{ JSch })*]
 additems :=  "additionalItems": (bool | { JSch })
 minitems := "minItems": n
 maxitems := "maxItems": n
 unique := "uniqueItems": bool
```

Here **n** is a natural number and **bool** is either true or false.

Object Restrictions

```
objRes := prop | addprop | req | minprop | maxprop | dep | pattprop
prop := "properties": { kSch (, kSch)*}
kSch := kword: { JSch }
addprop := "additionalProperties": (bool | { JSch })
req := "required": [ kword (, kword)*]
minprop := "minProperties": n
maxprop := "maxProperties": n
dep := "dependencies": { kDep (, kDep)*}
kDep := (kArr | kSch)
kArr := kword: [ kword (, kword)*]
pattprop := "patternProperties": { patSch (, patSch)*}
patSch := "regExp": { JSch }
```

Here **n** is a natural number, **bool** is either true or false and **regExp** is a regular expression. As above, each **kword** is representing a keyword that must be unique in the nest level that is occurs.

Multiple Restrictions

```
multRes := allOf | anyOf| oneOf | not | enum
anyOf := "anyOf": [ { JSch } (, { JSch }) * ]
allOf := "allOf": [ { JSch } (, { JSch }) * ]
oneOf := "oneOf": [ { JSch } (, { JSch }) * ]
not := "not": { JSch }
enum := "enum": [Jval (, Jval)*]
```

Here **Jval** is either a `string`, `number`, `array`, `object`, `bool` or a `null` value. Moreover each **Jval** must be different from each other(otherwise they would be superfluous).

Referred Schemas

Note that **uriRef** below is the same grammar we defined earlier for URIs.

```
refSch := "$ref": "uriRef"
uriRef := ( address )? ( # / JPointer )?
JPointer := ( / path )
path := ( unescaped | escaped )
escaped := ~0 | ~1
```

Where **unescaped** can be any character except for `/` and `~`. Also, **address** corresponds to any URI that does not use the `#` symbol, or more precisely to any URI-reference constructed using the following grammar, as defined in the official standard:

```
address = (scheme : )? hier-part (? query )
```

Well Formedness

The grammar above allow for some problematic schemas that need to be left out using a notion that we call *well formedness*.

As an example of a problematic schema, consider the following:

```
{
    "definitions": {
        "Schema1": {
            "not": {"$ref": "#/definitions/Schema1"}
        }
    },
    "$ref": "#/definitions/Schema1"

}
```

The above defines a Schema that is both S and not S at the same time!

Let **S** be a JSON Schema document, and let $\mathbf{S_1}$,...,$\mathbf{S_n}$ be all the schemas retrieved by any JSON Pointer inside **S**. The *reference graph* of S is a directed graph whose set of nodes is {$\mathbf{S_1}$,...,$\mathbf{S_n}$} and where there is an edge from $\mathbf{S_i}$ to $\mathbf{S_j}$ if **S_i** is a boolean combination of schemas and at least one of those schemas corresponds to the JSON Pointer that retrieves $\mathbf{S_j}$. For instance, the graph of the document above has only one node, corresponding to the subschema defined under `"Schema1"` , and the only edge is a self loop on this node. Edges are only added if $\mathbf{S_i}$ is a boolean combination of schemas, not if, for example, $\mathbf{S_i}$ is an object and a reference inside a `"properties"` keyword retrieves $\mathbf{S_j}$.

We say that **S** is a well formed Schema if the reference graph of **S** is acyclic.

We propose to add the well formedness condition to the Schema, and we assume that all schemas are well formed whenever we talk about conformance to this document.