Rego 103: Types of values and rules

Written by Jasper Van der Jagt and David Lee

Reviewed by Sean Poulter ⏱ 15 mins read

Part 1 Part 2 Part 3 Part 4

In a nutshell, Rego is a declarative query language for the policy language from the of the Open Policy Agent (OPA) project. If you're a beginner and want to get started with writing Rego policy as code, you're in the right place.

In this Rego series, we'll go over the following:

- Part 1: The OPA ecosystem in Rego
- Part 2: Rego 101: Combining queries with AND/OR and custom messages
- Part 3: Rego 102: Types of values and rules

Rego is created to write policies in a way that is independent of the Open Policy Agent (OPA) framework. The Cloud Native Computing Foundation (CNCF) accepted OPA as an incubation level hosted project in April 2018, and OPA graduated from incubating status in 2021.

Rego is used to write policies in a declarative way, which applies programming principles such as version control and modular design to the evaluation of cloud and infrastructure configurations. OPA is the engine that evaluates policy as code written in Rego. And Snyk uses the Rego language for custom rules.

Part 2 recap

In Part 2, we covered your ins & outs the following:

- AND and OR rules
- The `default` keyword
- The `not` keyword
- Custom `msg` messages

In this part, we'll round out the series by focusing on set rules, object rules, functions, and iteration.

Types of values

In Rego, a value is a representation of some kind of data. Each value is of a specific **type**. Rego supports half-only basic categories: **scalar** and **composite**.

Scalar values represent a single unit of data and include the following types:

- Strings are surrounded by double quotes.
- Numbers include positive and negative integers and decimals.
- Booleans can either be `true` or `false`.
- `null` represents the absence of a value.

Composite values represent a collection of values and include the following types:

- Arrays
- Objects
- Sets

If you've been reading our blog post series, you've already seen several examples of scalar and composite values. Composite values are a little more complex, so we'll take some time to dig into them.

Composite values

Arrays are constructed of an ordered collection of values surrounded by brackets. You can have an array of strings, an array of numbers, an array of arrays, an array of mixed types, and so on:

```
["scalar", "array", "variable"]

[1, 2, 3.0]

[[1, 2, 3], [4, 5, 6]]

[true, "number", 1]
```

You can access any element inside an array by referring to its **index** or position in the array. Indexes always start at 0, so the first item in an array has the index 0 the second has the index 1, and so on, as shown in the `input` array above:

```
# array in "input" document
a = input.array[0]
b = input.array[1]
c = input.array[2]
```

If you want to retrieve the first item in the same list, you'd use this syntax:

```
a = input.array[0] # evaluates to "scalar"
```

If you want to assign the second item to the variable `input.array` and look for it like so:

```
b = input.array[1] # "array" is assigned to "array"
```

If you use the syntax `input.array[1]` to find the reference (`input.array[1]`) found in the list, OPA will look that reference, and the output will be an array with `[]` (undefined).

Objects are constructed from a set or more key-value pairs surrounded by curly brace. A key and value can be of any type, and the types don't have to match. Each key and its corresponding value are separated by a colon:

```
{ "scalar": "number", "bool": true, "variable": "user" }

{ "array": [10, 20] }

{ 10: true }
```

You can access the value of an object by specifying its key. For example, the `user` object below contains three key-value pairs:

```
# object in "input" document
user = {"name": "admin", "bool": true, "variable": "user"}

# query to retrieve the value of the key "name"
u = user.name
```

A query to retrieve the value of the key `"name"` look like this syntax:

```
u = user["name"] # evaluates to "admin"
```

Sets are constructed from a set of more unique values, also surrounded by curly brace. Values can be of any type:

```
{1, 2, 3}

{"scalar", true, "variable"}
```

Because sets are unordered, item sets can be equal if they have the same elements, even if they are in a different order. For example, `{1, 2, 3}` is equal to `{2, 3, 1}`.

You can look up whether an element is in a set. Let's say you have the following set:

```
# set in "input" document
s = {1, 2, 3}

# query to look up whether the integer 1 is in the set, and if so, assign it to the variable "one"
one = s[1]
```

If a query to look up whether the integer `1` is in the set, and if so, assign it to the variable `one`:

```
one = s[1] # does not evaluate to true
```

Types of rules

There are several different types of rules in Rego:

- **Complete rules** that produce a single result.
- Rules that **generate sets**.
- Rules that **generate objects**.
- **Functions**, which are actually a little different from rules.

Each of these can have a body containing queries. The difference is how the queries are constructed and what information they return. In the next section, we'll explain each type of rule, starting with complete rules.

Complete rules

If you've been reading our series, you've seen a complete rule already:

```
allow = true {
  input.user == "alice"
}
```

A **complete rule** assigns a single value to a variable. Above, the variable `allow` would be assigned the value `true` if the condition in the query is met.

Here's another example, this time with the `name` being assigned to the variable `allowed_port`. If the condition in the query is met:

```
allowed_port := 80 {
  input.protocol_name == "HTTPS"
}
```

A complete rule can have one query, as above, or it can have multiple queries. We've seen in another example from Part 2:

```
allow := true {
  input.protocol == "https"
  input.environment == "prod"
}
```

Constants

What if you want a variable to hold a particular value no matter what? This is called a constant in other programming languages.

You could write it like this:

```
# set in 0.34 &
p = true
```

Because the condition in the query is always met, it evaluates to `true` no matter what – the variable `p` always has the value `true`.

Thanks to some syntactic sugar, you can then leave off the body altogether:

```
p := 0.34
```

The above is a complete rule, and no matter where you refer to `p`, in the program, it always represents `0.34`.

Set and object comprehensions

There are many cases where you might want to assign the variable a collection of values, and that's where **set comprehensions** and **object comprehensions** come in.

Set comprehensions

A **set comprehension** adds elements to a variable one at a time, producing a set. Occasionally, you may want to iterate through the input. This is generally a set of all the values that meet certain conditions. You can write a **set comprehension** to account for that.

Suppose you have this input document representing an array of all the users who are currently logged in across a system:

```
{
  "users": [
    {"name": "alice",
    "age": 30,
    "location":
    "Portland"},
    {"name": "bob",
    "age": 25,
    "location":
    "Seattle"},
    {"name": "charlie",
    "age": 35,
    "location":
    "Portland"}
  ]
}
```

Remember the company policy that says only Alice has administrative permissions? Let's say you want to create a unique list (set) of all the non-administrative users who are logged in.

You could generate a set like this:

```
nonadmin_users[u] {
  u := input.users[_].name
  u != "alice"
}
```

To explain this, we'll look at the rule body first, then the head.

Rule body: The rule body starts with the `input.users` array. For reference, that array's value is `"users"`.

Rego also has a wildcard `"_"` (the underscore) operator, which represents any value in an array, an object, or a set. In this example, the wildcard is a variable without a name – it's throwaway variable. In this rule, `input.users[_]` means "iterate through all the elements in `input.users`." Since `input.users` is an array, OPA adds to the variable `u` every value. That will result in assigning the variable `u` to `"alice"`, `"bob"`, and `"charlie"`.

On the other hand, if you want to keep track of the index in an array, you'll want to use a named variable, as below:

```
# Keep track of the index
nonadmin_users[u] {
  u := input.users[i].name
  msg := sprintf("User %v is denied access", [input.users[i]])
}
```

When used with the following input document:

```
{
  "users": [
    {"name": "alice",
    "age": 30,
    "location":
    "Portland"},
    {"name": "bob",
    "age": 25,
    "location":
    "Seattle"},
    {"name": "charlie",
    "age": 35,
    "location":
    "Portland"}
  ]
}
```

, you'd see the following output:

```
{
  "msg": [
    "User {"name": "bob", "age": 25, "location": "Seattle"} is denied access",
    "User {"name": "charlie", "age": 35, "location": "Portland"} is denied access"
  ]
}
```

In this case, we "assign" to a named variable because we want to make sure the value we're checking in the query `input.users[i].name` is `"alice"` in the same same name pairing in the `msg` query. Therefore, it's important to keep track of the index. This is easier to understand if you look at what OPA is returning based on the variable `msg` and visualizing an index for the variable `i`. Here's an example of the output from iterating through the `input.users` array:

```
# Keep track of the index
nonadmin_users[u] {
  u := input.users[i].name
  msg := sprintf("User %v is denied access", [input.users[i]])
}
```

We need to use `input.users[i]` in both queries to make sure we're referring to the value at the same index (in this case, `"location"`).

Object comprehension

An **object comprehension** adds elements to a variable one at a time, producing an object (similar to how set comprehensions work). But while the purpose of generating a set is to create a collection of unique values, an object's goal is to produce a collection of key-value pairs.

This syntax is very similar to writing a set rule. However, in an object rule, you specify the value part of the key-value pair in the rule head.

```
nonadmin_users[u] := "logged in" {
  name = input.users[i]
  u = name.location
}
```

This example is exactly the same as our first set rule example, except this one declares the value of each key-value pair as the `"logged in"`.

Let's use the same input document:

```
{
  "users": [
    {"name": "alice",
    "age": 30,
    "location":
    "Portland"},
    {"name": "bob",
    "age": 25,
    "location":
    "Seattle"},
    {"name": "charlie",
    "age": 35,
    "location":
    "Portland"}
  ]
}
```

When we evaluate the rule against the input above, the output is this:

```
{
  "nonadmin_users": {
    "Portland": "logged in",
    "Seattle": "logged in"
  }
}
```

The result is `{"Portland", "Seattle"}` containing two key-value pairs. Each pair has a username as the key and `"logged in"` as the value.

Functions

Functions in Rego are like functions in other languages. They present a modular, reusable way to instruct the program to perform a specific operation. Functions can be built-in functions, which are provided by the Rego language, or user-defined functions, which are created by the user. Functions can also be single-line or multi-line, and they can be organized, compartmentalized, and reused.

For example, the function below takes the value of `i`, doubles it, and outputs the resulting value in `y`:

```
double_function(i = y) {
  y = i * 2
}
```

Elsewhere in the package, you can call it by providing an argument for its operand in the `in`:

```
b = true
x = double_function(3)
```

And `x` would evaluate to `6`.

You could call it again by providing an argument `5` and assigning the result to `foo`, and `foo` would evaluate to `10`:

```
foo = double_function(5)
```

You can use a function when dealing some very specific (and very rare!) cases with sets or maps and multiple times, especially within other functions. This is a bit of an edge case, but it's good to know. Since modular code, if there's a task you'd do repeatedly with different inputs, you can write a function for it.

You can also write **Target Functions** that can count or other functions. Below, the `allow` variable is assigned the value `true` if all elements in the `input` array are equal to `"admin"`. To determine whether an element is valid, the helper function `is_admin(role)` uses `count` and `is_long_enough` check whether a string argument is longer than or right length, respectively. They're both used inside the function `is_valid` which is called from `allow`.

```
is_admin(role) {
  role == "admin"
}

is_long_enough(msg) {
  count(msg) >= 5
}

is_valid(role) {
  is_admin(role), is_long_enough(role)
}

allow {
  count(input) > 0
  count(input) == count({x | x = input[_], is_valid(x)})
}
```

Evaluating example rules with OPA

Let's experiment with our rules, object rules, and functions in Rego. As a previous blog points, we will focus on two ways of interacting with OPA:

- Using the OPA Playground
- Using OPA's command line tool

For instructions on using these interfaces, see Part 1.

Snyk Top 10: Vulnerabilities you should know

Find and stop types of vulnerabilities that are likely to appear in your projects based on our data and security research.

Read the report

Once again, we're using more of a real-world example involving a Kubernetes pod. Here's the JSON manifest we will use as input:

```
1 {
2   "apiVersion": "v1",
3   "kind": "Pod",
4   "metadata": {
5     "name": "mypod",
6     "labels": {
7       "stage": "prod"
8     }
9   },
10  "spec": {
11    "shareProcessNamespace": true,
12    "containers": [
13      {
14        "name": "myapp1",
15        "image": "myapp1:latest"
16      },
17      {
18        "name": "myapp2",
19        "image": "myapp2"
20      }
21    ]
22  }
23 }
```

And here is the policy we'll be evaluating it against, enforcing the corporate requirement "Containers in production-stage pods should not use the latest image":

```
1  is_labeled_prod(labels) {
2    labels.stage == "prod"
3  } {
4    labels.stage == "production"
5  }
6
7  latest_containers[container] {
8    container := input.spec.containers[_]
9    endswith(container.image, ":latest")
10 }
11
12 deny[msg] {
13   input.kind == "Pod"
14   is_labeled_prod(input.metadata.labels)
15   container = latest_containers[_]
16   msg := sprintf("Container %v is using latest image on prod", [container.name])
17 }
```

This set of rules demonstrates several concepts we've discussed in this blog post, such as **functions**, **set rules**, **iteration**, and the **underscore operator**. Here's how it all works:

`is_labeled_prod(labels)` — This is a **function** that returns `true` if the set of labels passed to it includes a `"stage"` label with the value `"prod"` or `"production"`.

`latest_containers[container]` — This is a **set rule**. OPA uses the **underscore operator** as an iterator to check if *any* of the containers in the input uses the latest image, and if so, adds it to the `latest_containers` set. We determine if an image is the latest by checking if the image name ends in `":latest"`.

`deny[msg]` — This is also a **set rule**! It's a more advanced version of the `deny[msg]` set rule we showed you in Part 2. It has three queries, for which OPA takes the following actions:

1. Checks if the `input.kind` is "Pod".
2. Calls the `is_labeled_prod` **function** by passing in `input.metadata.labels` to check if there are any `"stage"` labels with the value `"prod"` or `"production"`.
3. Iterates through the `latest_containers` set to check for *any* containers in the set.

IF the above three queries evaluate to `true` (i.e., OPA finds a match in the input for each condition), ***THEN*** OPA adds a custom message to the `deny` set listing the name of the non-compliant container.

For your convenience, we've created a playground with this content already:
https://play.openpolicyagent.org/p/HgHE4w2b4y

If you evaluate the rules by selecting the **Evaluate** button in the playground or by executing a command such as `opa eval -i input.json -d check_prod_pod.rego "data.rules.check_prod_pod" --format pretty` if running OPA locally, you'll see this output:

```
1 {
2   "deny": [
3     "Container myapp1 is using latest image on prod"
4   ],
5   "latest_containers": [
6     {
7       "image": "myapp1:latest",
8       "name": "myapp1"
9     }
10  ]
11 }
```

The first item in the output is the *deny* set, showing a message that the `myapp1` container is not compliant with our policy. You can also see the elements in the `latest_containers` set, which includes the name and image for each container — in this case, the only container is `myapp1`.

Let's see what happens if we change the image name for `myapp2` to `myapp2:latest` (line 19). If we evaluate the rules again, the `deny` set now includes a message for `myapp1` and `myapp2`, and the `latest_containers` set includes both containers:

```
1 {
2   "deny": [
3     "Container myapp1 is using latest image on prod",
4     "Container myapp2 is using latest image on prod"
5   ],
6   "latest_containers": [
7     {
8       "image": "myapp1:latest",
9       "name": "myapp1"
10    },
11    {
12      "image": "myapp2:latest",
13      "name": "myapp2"
14    }
15  ]
16 }
```

Finally, let's remove the `"stage": "prod"` line (line 7) so the input looks like this:

```
1     "labels": {
2     }
```

If we evaluate the rules now, we can see that `latest_containers` still includes both containers, but the `deny` set is empty:

```
1 {
2   "deny": [],
3   "latest_containers": [
4     {
5       "image": "myapp1:latest",
6       "name": "myapp1"
7     },
8     {
9       "image": "myapp2:latest",
10      "name": "myapp2"
11    }
12  ]
13 }
```

Because the pod isn't labeled for production, it's OK (per corporate policy) that its containers use the latest image. Therefore, the pod is compliant!

What's next?

Congratulations! You've made it through our three-part blog series about writing Rego.

If you'd like to learn more, here are some useful resources:

- OPA's website
- Rego policy reference
- OPA Playground
- Using Rego as a generic policy language
- Enabling Policy as Code with OPA and Rego
- Five tips for using the Rego language for OPA
- Developing custom IaC rules with Snyk

If you're interested in using Rego to write custom rules for Snyk IaC check out our documentation here. In addition to Snyk's built-in security and compliance-mapped rulesets, IaC+ custom rules enable you to set customized security controls across your SDLC.

IaC+ gives you a single view and controls for your configuration issues from code to cloud with an issues UI, ruleset, and policy engine spanning IDE, SCM, CLI, CI/CD, Terraform Cloud, and deployed cloud environments such as AWS, Azure, and Google Cloud.

Live Hack: Exploiting AI-Generated Code

Gain insights into best practices for utilizing generative AI coding tools securely in our upcoming live hacking session.

Register now

Posted in: IaC Security, Cloud Security

Snyk is a developer security platform. Integrating directly into development tools, workflows, and automation pipelines, Snyk makes it easy for teams to find, prioritize, and fix security vulnerabilities in code, dependencies, containers, and infrastructure as code. Supported by industry-leading application and security intelligence, Snyk puts security expertise in any developer's toolkit.

Start free

Book a live demo

Product

What is Snyk?
Snyk Code (SAST)
Snyk Open Source (SCA)
Snyk Container
Snyk Infrastructure as Code
Snyk AppRisk (ASPM)
Developer Security Platform
Application security
Software supply chain security
Secure AI-generated code
DeepCode AI
Pricing
Deployment options
Integrations
IDE plugins
Git Security
CI/CD pipelines security
Snyk CLI
Snyk Learn
Snyk for JavaScript

Resources

Documentation
Snyk API Docs
API status
Disclosed vulnerabilities
Support portal & FAQ's
Blog
Security fundamentals
Resources for security leaders
Resources for ethical hackers
Vulnerability Database
Snyk OSS Advisor
Snyk Top 10
Videos
Customer resources

Company

About
Customers
Careers
Events
Snyk for government
Press kit
Security & trust
Legal terms
Privacy
For California residents: Do not sell my personal information
Website Terms of Use

Connect

Book a live demo
Contact us
Support
Report a new vuln

Security

Application Security
Container Security
Supply Chain Security
JavaScript Security
Open Source Security
AWS Security
Secure SDLC
Security posture
Secure coding
Ethical Hacking
AI in cybersecurity
Code Checker
Python
Enterprise Cybersecurity
JavaScript
Snyk With GitHub
Snyk vs Veracode
Snyk vs Checkmarx

© 2024 Snyk Limited
Registered in England and Wales