```
│
│ This was checked by the validation rule at main.tf:21,3-13.
```

You can have multiple `validation` blocks in each variable to check multiple conditions:

```
variable "min_size" {
  description = "The minimum number of EC2 Instances in the ASG"
  type        = number

  validation {
    condition     = var.min_size > 0
    error_message = "ASGs can't be empty or we'll have an 
outage!"
  }

  validation {
    condition     = var.min_size <= 10
    error_message = "ASGs must have 10 or fewer instances to keep 
costs down."
  }
}
```

Note that `validation` blocks have a major limitation: the `condition` in a `validation` block can *only* reference the surrounding input variable. If you try to reference any other input variables, local variables, resources, or data sources, you will get an error. So while `validation` blocks are useful for basic input sanitization, they can't be used for anything more complicated: for example, you can't use them to do checks across multiple variables (such as "exactly one of these two input variables must be set") or any kind of dynamic checks (such as checking that the AMI the user requested uses the x86_64 architecture). To do these sorts of more dynamic checks, you'll need to use `precondition` and `postcondition` blocks, as described next.

Preconditions and postconditions

As of Terraform 1.2, you can add `precondition` and `postcondition` blocks to resources, data sources, and output variables to perform more dynamic checks. The `precondition` blocks are for catching errors before you run `apply`. For example, you could use a `precondition` block to do a more robust check that the `instance_type` the user passes in is in the AWS Free Tier. In the previous section, you did this check using a `validation` block and a hardcoded list of instance types, but these sorts of lists quickly go out of date. You can instead use the `instance_type_data` data source to always get up-to-date information from AWS:

```
data "aws_ec2_instance_type" "instance" {
  instance_type = var.instance_type
}
```

And then you can add a `precondition` block to the `aws_launch_configuration` resource to check that this instance type is eligible for the AWS Free Tier:

```
resource "aws_launch_configuration" "example" {
  image_id        = var.ami
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]
  user_data       = var.user_data

  # Required when using a launch configuration with an auto
scaling group.
  lifecycle {
    create_before_destroy = true
    precondition {
      condition     =
data.aws_ec2_instance_type.instance.free_tier_eligible
      error_message = "${var.instance_type} is not part of the
AWS Free Tier!"
    }
  }
}
```

Just like `validation` blocks, `precondition` blocks (and `postcondition` blocks, as you'll see shortly) include a `condition` that must evaluate to `true` or `false` and an `error_message` to show the user if the `condition` evaluates to `false`. If you now try to run `apply` with an instance type not in the AWS Free Tier, you'll see your error message:

```
$ terraform apply -var instance_type="m4.large"
│ Error: Resource precondition failed
│
│   on main.tf line 25, in resource "aws_launch_configuration"
"example":
│   18:     condition =
data.aws_ec2_instance_type.instance.free_tier_eligible
│     ├────────────────
│     │ data.aws_ec2_instance_type.instance.free_tier_eligible is
false
│
│ m4.large is not part of the AWS Free Tier!
```

The `postcondition` blocks are for catching errors after you run `apply`. For example, you can add a `postcondition` block to the `aws_autoscaling_group` resource to check that the ASG was deployed across more than one Availability Zone (AZ), thereby ensuring you can tolerate the failure of at least one AZ:

```
resource "aws_autoscaling_group" "example" {
  name                 = var.cluster_name
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = var.subnet_ids

  lifecycle {
    postcondition {
      condition     = length(self.availability_zones) > 1
      error_message = "You must use more than one AZ for high
availability!"
    }
  }

  # (...)
}
```

Note the use of the `self` keyword in the `condition` parameter. *Self expressions* use the following syntax:

```
self.<ATTRIBUTE>
```