Matchers

This section describes all the matchers currently recognized by the `forall` section.

has

```
has x, y, z
```

`has` passes when it finds a fact that matches the template. It can introduce new variables.

neg / missing

```
neg x, y, z
```

`neg` passes when it *cannot* find any facts that match the template. It **may not** introduce new variables (because it doesn't make say to say "let x equal something that doesn't exist").

maybe / optional

```
maybe x, y, z
```

`maybe` passes whether or not it finds a fact that matches the template. It is only useful if it introduces a new variable.

none / ncc

```
none {
  has x, y, z
  # ...
}
```

`none` creates a subchain of matchers and passes if that entire subchain yields an empty result set.

any

```
any {
  option {
    has x, y, z
    # ...
  }
  option {
    has x, y, z
    # ...
  }
}
```

`any` passes if any of the subchains in `option`s yield a non-empty result set.

assign

```
assign :X do |token|
   # ...
end
```

`assign` always passes. The argument must be a new variable that will be bound to the value returned by the block.

assuming

```
engine << rule("base") do
  # ...
end

engine << rule("specialization") do
  forall {
    assuming "base"
    # ...
  }
end
```

If several of your rules have a common prefix, `assuming` lets you extract it into a reusable base rule. The specialized rule will work as if the `forall` section of the base rule were included verbatim. This lets you keep your rules more compact and helps performance a bit, since some of the work can be reused.

The base rule must be installed prior to installing the specialized ones.

The rule compiler can collapse some simpler matchers like `has` into a single execution node if it detects that the declarations are identical, but for matchers taking code blocks, like `assert`, it is not possible, and `assuming` must be used.

Filter matchers

Filters are a category of matchers that block rule execution based on some predicate; they operate entirely on the token assembled so far.

same / eq / equal

```
same x, y
```

`same` passes if its arguments compare as equal using `#==`. It is obviously only useful if at least one of them is a variable.

diff / ne

```
diff x, y
```

`diff` passes if its arguments do not compare as equal using `#==`.

less

```
less x, y
```

`less` passes if `x < y`.

greater

```
greater x, y
```

`greater` passes if `x > y`.

assert

```
assert { |token|
  # ...
}

assert :A, :B, :C, ... do |a, b, c, ...|
  # ...
end
```

`assert` passes if the block returns a truthy value.