1. Welcome to the *GrapeVine* tutorial!

GrapeVine is a tool for graph transformations. ***GrapeVine*** can be used "under the hood" as a library in software projects, but it also comes with a computational notebook UI, which leverages [Gorilla].

A computational notebook consists of mulltiple *worksheets* (you are looking at one now). Each worksheet consists of *static* and *executable segments*.

1.1 Static segments

Static segments can contain text using markdown, html, $\LaTeX$, images and fancy things like Mermaid diagrams that can be edited online. See below for an example ER-diagram that shows the ***GrapeVine*** meta model (which you don't need to understand right now). If you want to edit the diagram, right-click opens in a new window with Mermaid live).

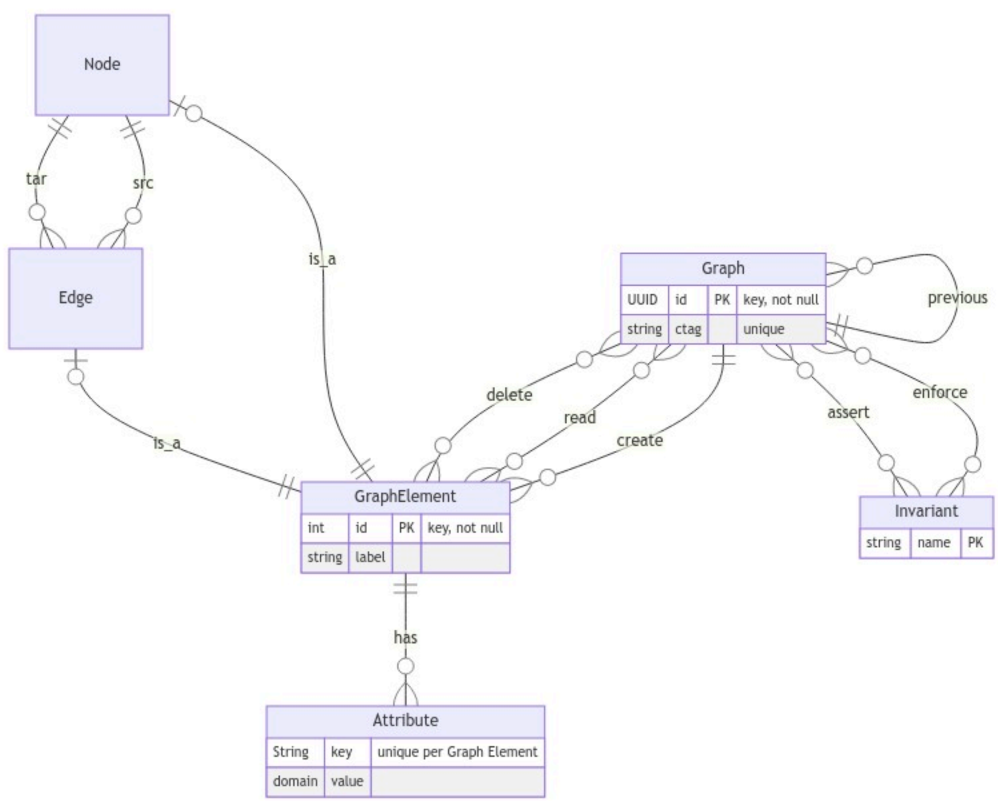


Fig. 1. ***GrapeVine*** Meta Model in Entity-Relationship Notation

1.2 Executable segments

Executable code is written in Clojure. (***GrapeVine*** code is a domain specific language extension of Clojure).

Executable segments are not automatically invoked when a worksheet is loaded.

Shift + enter evaluates a single code segment.

Ctrl + Shift + enter evaluates all code segments in a worksheet.

```
(+ 2 3)
```

```
5
```

1.3 Commands

Press ctrl+g twice in quick succession or click the menu icon (upper-right corner) for an overview on commands.

1.4 Using *GrapeVine* - Resources and namespaces

To make ***GrapeVine*** available, *its library must be imported.* Under the hood, ***GrapeVine*** connects to the graph database NEO4J. The simplest way to do the import is by using a `(use 'grape.core')` form, but it's best practice to define a separate namespace for each worksheet. Below creates a worksheet of name `tutorial`.

(**Remember: The statement below must be executed before any *GrapeVine*-related commands.** (Shift-enter) Establishing the session may take a few seconds. Wait until the green light is gone.)

Extend the statement below to import from other worksheets, if your project uses multiple worksheets.

```
(ns tutorial
  (:require [grape.core :refer :all]))
```

```
nil
```

2. Graphs and graph persistence

GrapeVine uses a database to store graphs. Graphs, once defined are persisted. Arbitrarily many graphs can be created. Morevover, graphs are maintained in a fully persistent data structures, i.e., *all* versions of a graph can be accessed and modified.

A new (empty) graph is created using the `(newgrape)` function:

```
(newgrape)
```

```
("3208917e-4a4f-409f-aa77-8ddb1d918e36")
```

As shown above, graphs are internally identified with a unique ID. That may not be convenient for the worksheet, but we can always define *vars* to refer to graphs. The statement below defines a *var* that refers to an empty graph.

Note: *if you are not familiar with Clojure, you can think of vars are similar to variables in other languages.*

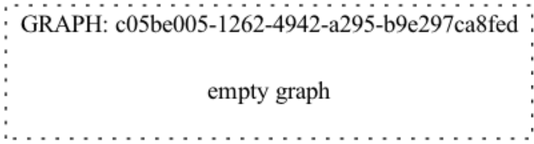
```
(def g0 (newgrape))
```

```
#'tutorial/g0
```

The built-in form `view` can be used to generate an image of the content of a graph. (These images are saved when the worksheet is saved.)

```
(view g0)
```

(



)

Unsurprisingly, the graph is empty. Perhaps suprisingly, it will always remain so. (Graphs are immutable in ***GrapeVine***).

2.1 Graph Enumerations (*Grapes*)

If you are very perceptive (and have some knowlege of Clojure data types), you may have noted that the above call to `newgrape` returned a *sequence* (with a single element) rather than a simple string (as indicated by the brackets `()`). Such a sequence is called a **GRAPH set Enumeration** or ***grape*** for short.

In ***GrapeVine***, all operations that produce graphs actually return *grapes* rather than graphs, even those ones that are guaranteed to generate a single graph only, such as the `newgrape` function.

Using a single data type for operations that can produce graphs facilitates composability of operations (as we will see later).

This technique of using a single data type is well known also from other domains, such as the data type of a "relation" in relational databases. (Even queries that are guaranteed to produce a single, unique result will still generate a relation with a single element.)

For simplicity and if the context is clear, we may continue to refer to ***grapes*** with a single element as "graphs".

3. Graph transformation rules

Graph transformation rules ("rules" for short) are defined in *hybrid* language (textual input with graphical view). There are three example rules below.

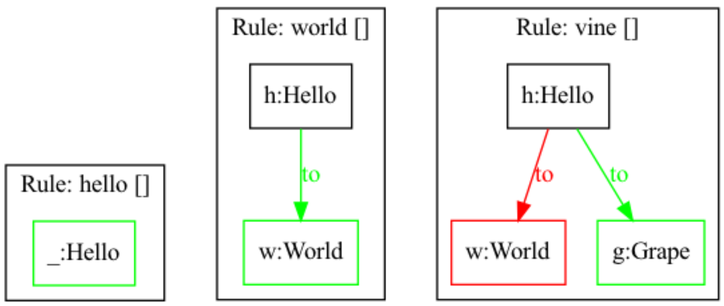
1. The first rule (**`hello`**) creates a new node of type *Hello*
2. The second rule (**`world`**) looks for an existing node of type `Hello` and connects it with a new node of type `World`.
3. The third rule (**`vine`**) looks for two nodes of type `Hello` and `World`, respectively, which are connected with a `to` edge and replaces the `World` node with a node of type `Grape`.

As we can see in the visualization, green colour marks new graph elements, red colour marks deleted ones, and black colour marks "preserved" elements.

> *Note: While graphs are persisted in the database, rules are not. If you want to execute any of these rules, you need to evaluate the code segment below first to declare them. Otherwise, the rules will not be known. Also, note that enclosing these rules in a `(list ...)` form has no significance and is purely cosmetic. The only reason for having it is to render multiple rules horizontally in the visualization. Feel free to delete it if you want vertical listing of output.*

```
(list

(rule hello []
   (create
     (node :Hello)))

(rule world []
   (read
      (node h:Hello))
   (create
      (node w:World)
      (edge :to h w )))

(rule vine []
   (read
      (node h:Hello)
      (node w:World)
      (edge e:to h w ))
   (delete w e)
   (create
      (node g:Grape)
      (edge :to h g)))
 )
```

(



)

4. Rule application

Applying a rule to a graph is as simple as calling a function with the rule's name with the graph as an argument. Note that this will produce a new graph (if the rule is applicable). (Graphs are immutable in *GrapeVine*, so `g0` will always remain an empty graph.)

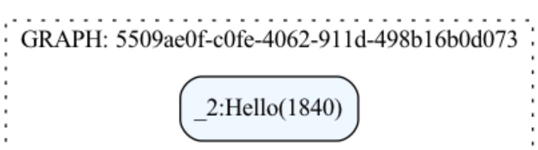
```
(hello g0)
```

```
("119b89cd-a377-4fce-872b-6b879f594e82")
```

Applying rule `hello` to graph `g0` worked, since it generated a new graph. If we want to visualize the resulting graph, we could copy the above result and use it as an argument for `viewgraph`, i.e., call `viewgraph '("7eff0750-abb7-428d-81ef-6c6483a17c3a")`. However, that's cumbersome. Of course, we could also just nest the functions:

```
(view (hello g0))
```

(



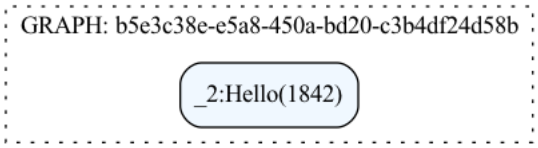
)

> Note: *The graph ids of the two applications of rule `hello` are differnt. Indeed we actually get a new graph each time a rule is applied.*

For convenience, ***GrapeVine*** also provides a shortcut symbol _ (underscore), to refer to the "last created graph" (resp. "last created ***grape***"). So the following also works:

```
(hello g0)
(view _)
```

(



)

> Note: *The shortcut (underscore) is like a global variable that is updated as a **side effect** when a new graph is produced. It should be clear that this breaks the functional computation paradigm otherwise used in **GrapeVine**. From a practical perspective this may mean that dynamic segments in a **GrapeVine** worksheet are no longer idempotent, but need to be executed in order to achieve a repeatable result.*

Now let's see the result of an unsuccessful rule application. Clearly, rule `world` requires the presence of a `Hello` node. So the following should not work:

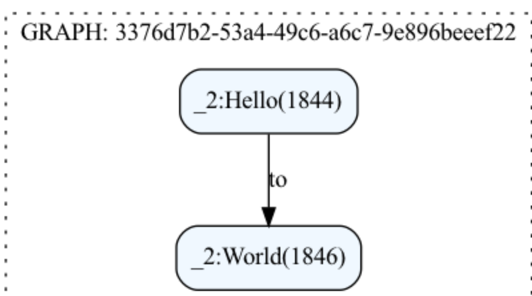
```
(world g0)
```

```
nil
```

However, rule `world` should be applicable to the graph that's produced when applying rule `hello` to graph `g0`:

```
(view (world (hello g0)))
```

(



)

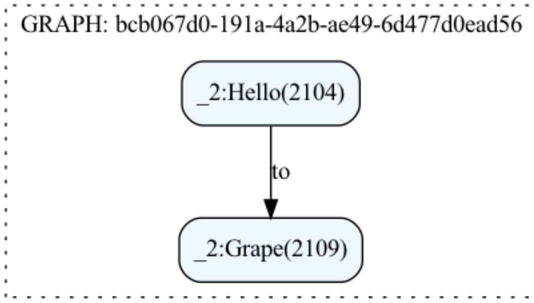
Deeply nested function may be hard to write and read. (Too many parentheses...) Consider this:

```
(view (vine (world (hello (newgrape)))))
```

```
()
```

Clojure provides ***threading macros***, to make functional composition more readable. The following expression is equivalent to the above:

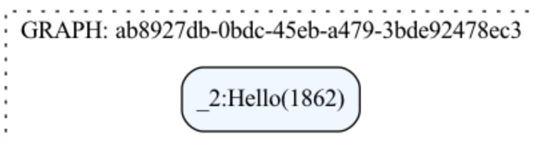
```
(-> (newgrape) hello world vine view)
```



4.1 Matching semantics

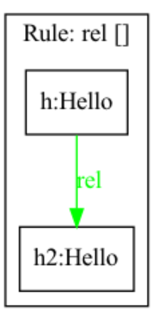
Rules may be defined to use ***homomorphic*** or ***isomorphic* matching** semantics. Homomorphic semantics is the default, and allows different graph elements in the rule to match the same graph elements in the graph that the rule is applied to. To understand this better, consider a graph (`g1`) with one `Hello` node:

```
(def g1 (-> (newgrape) hello))
(view g1)
```



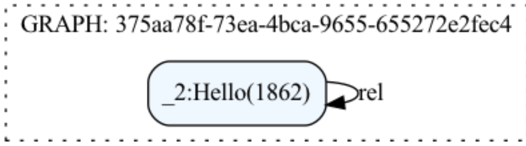
Now consider the following rule. It matches "two" `Hello` nodes with a new relationship `rel`.

```
(rule rel []
   (read
      (node h:Hello)
      (node h2:Hello))
   (create
      (edge e:rel h h2)))
```



Is this rule applicable to graph `g1`? The answer to this question depends on the matching semantics. If ***homomorphic*** matches are allowed, the two `Hello` nodes in rule `rel` may in fact be matched to the *same* node in the graph the rule is applied to. This would result in a rule with a `rel` edge to itself:

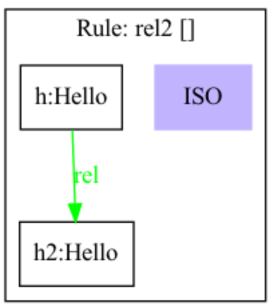
```
(view (rel g1))
```



( )

The above (homomorphic) matching semantics is the default. However, matching multiple nodes in a rule to a single node in the graph may not be desirable in some appliations. In this case, the rule designer can specify the rule to enforce ***isomorphic*** matching. The following rule `rel2` is identical to the above rule `rel`, except that it requires *isomorphic* matches.

```
(rule rel2 []
   (read :iso
       (node h:Hello)
       (node h2:Hello))
   (create
       (edge :rel h h2)))
```



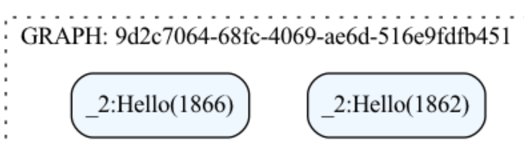
Applying that rule to our graph with a single `Hello` node should fail:

```
(rel2 g1)
```

```
nil
```

Let's add another `Hello` node:

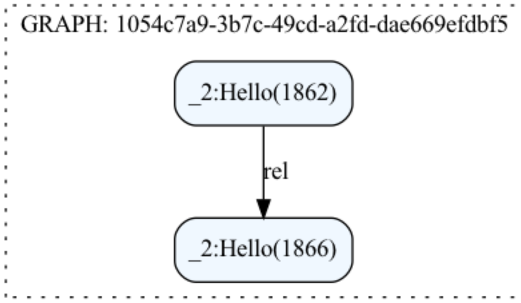
```
(def g2 (hello g1))
(view g2)
```



( )

We would now expect that an isomorphic match for rule `rel2` can be found:

```
(view (rel2 g2))
```



4.2 Non-determinism during matching

Consider the above example of graph `g2` again. Is there only one possible way to apply rule `rel2`? Indeed, there are **two** possible ways to apply that rule. The roles of the two nodes could also have been reversed, which would create the relationship in the *opposite* direction.

Applying a rule to a graph non-deterministically picks one out of possibly many valid matches.

4.3 Starred rule applications

In addition to simple rule applications, which non-deterministically pick one out of possibly many matches, ***GrapeVine*** provides **starred rule applications**. A starred rule application applies a rule so that *all* possible matches are performed. This generates a *set* of graphs, where each graph represents the result of a possible rule application. The output is a ***grape***, which contains more than a single graph. (See Section 2.1)

> *Note: this is the point in the tutorial when we start talking about "**grapes**" rather than graphs when referring to the input or output of operations.*

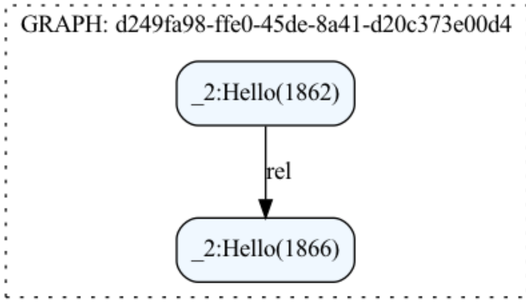
Invoking a starred rule is simply done by adding an asterisk after the rules name:

```
(rel2* g2)
```

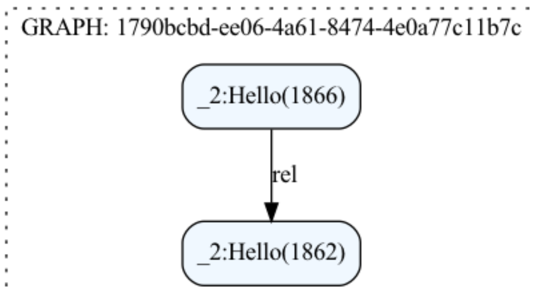
```
("fdcebbed-fa14-4c4f-be5e-0affc7ec949d" "fd2a630f-cc3a-4fc8-b8d0-ae0fb2d15033")
```

Indeed, we see that the *grape* produced by the starred application of rule `rel2` to our graph `g2` has two graphs. We can view the result by using `view`. (Yes, `view` works on *grapes* that contain more than a single graph).

```
(view (rel2* g2))
```



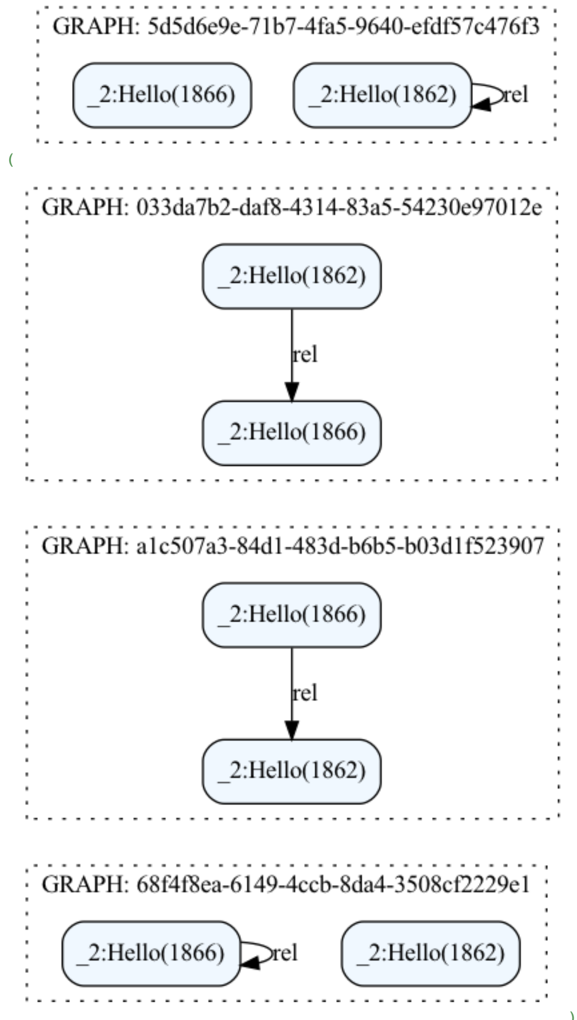
(



)

How many possible applications exist for our original (homomorphic) rule `rel`, when applied to graph `g2`? There should be four. Let's see...

```
(view (rel* g2))
```

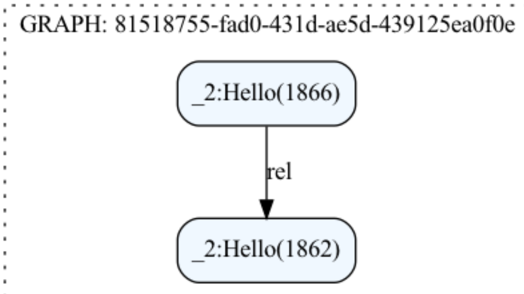


4.4. Sifting *grapes* for "distinct" graphs

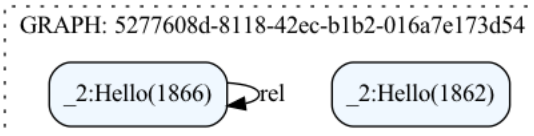
The above example shows a ***grape*** with four graphs. However, there are really only two *distinct* graphs in that set. In other words, there are two sets of graphs that can be seen as "equivalent" (when disregarding the node identifiers). We say that these graphs are identical up to isomorphism.

GrapeVine has the distinct operator (`dist`) operator to filter out such "duplications" (see below)

```
(-> g2 rel* dist view)
```



(



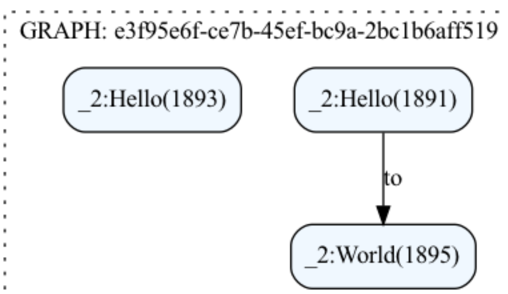
)

*Note: Since graph isomorphism checking is expensive, the current implementation of the `dist` operator is based a weaker notion of checking for graph bisimilarity. Two bisimilar graphs are likley isomorphic but not necessarily so. A future version of **GrapeVine** may do further isomorphism tests, similar to what has been implemented in GROOVE.*

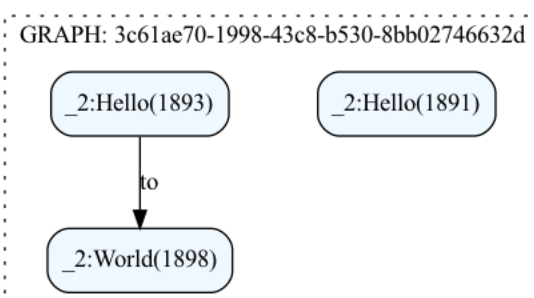
4.5 Applying rules to *grapes* (containing multiple graphs)

So far, we only applied rules to single graphs, i.e., ***grapes*** with a single element. However, rules can also be applied to ***grapes*** that enumerate multiple graphs. Consider the ***grape*** produced by the following statement. (We create a graph with two `Hello` nodes and then carry out a starred rule application of `world`.) The ***grape*** should have two graphs:

```
(-> (newgrape) hello hello world* view)
```



(

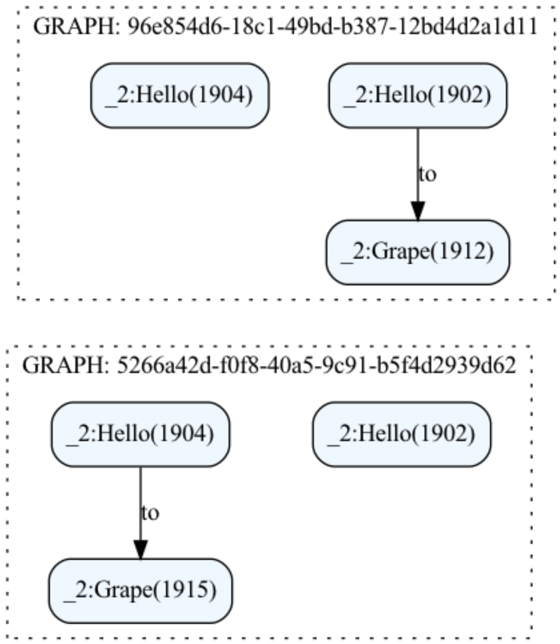


)

Applying rule `vine` to the above ***grape*** will apply the rule to *each* graph in the ***grape*** and produce a new ***grape*** that enumerates all resulting graphs. (This also works for starred rule applications.)

```
(-> (newgrape) hello hello world* vine view)
```

(



)

5. History Graphs, Traces and Derivations

As explained above, rules are applied to graphs (in ***grapes***) and are used to *derive* new graphs. This history of **derivations** of graphs can themselves be represented as a graph, where the nodes represent graphs and edges represent *occurrences* of graph transformations (i.e., rule applications).

5.1 History Graph

We define the **history graph** of a given graph g as the graph that contains all graphs (and derivations) that g depends on or that depend on g.

The above definition can easily be extended to ***grapes***: Since computations with ***grapes*** always start with a ***grape*** that contains a single graph only, the history graph of a ***grape*** is defined as the history graph of the unique original start graph that all graphs in the ***grape*** depend on.

We can generate the *history graph* of a ***grape*** using the `history` function:

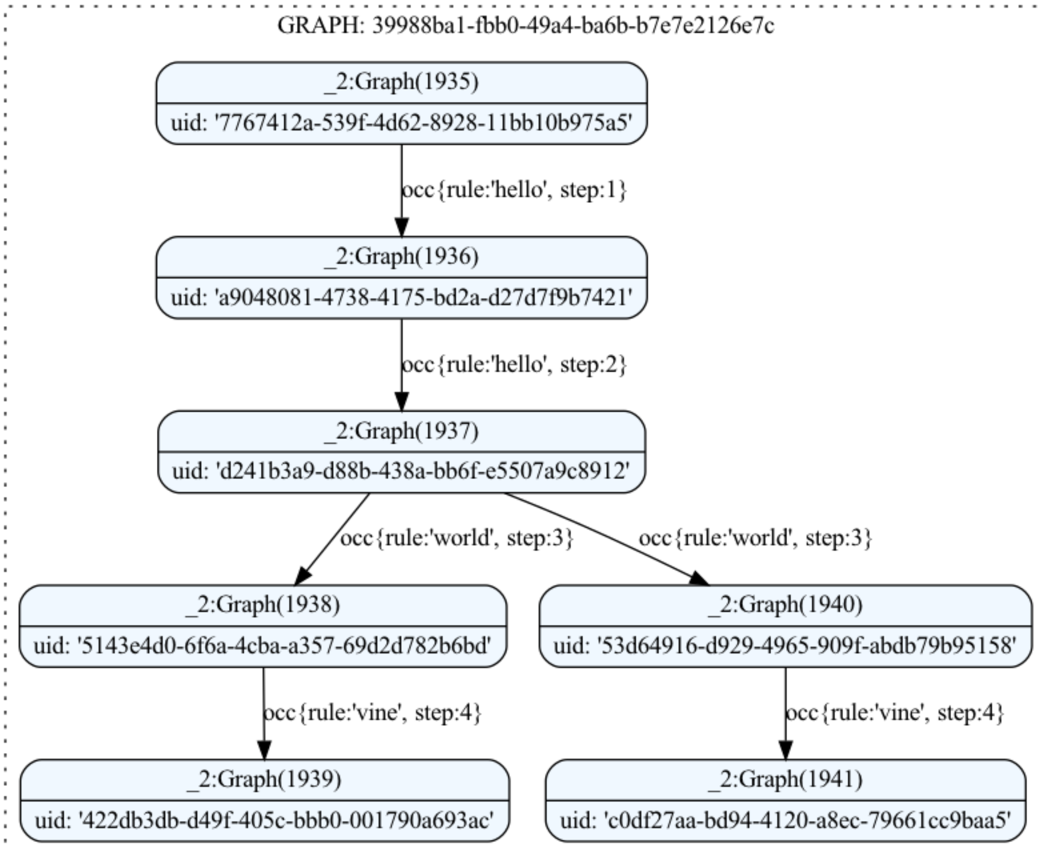
```
(def h (-> (newgrape) hello hello world* vine history))
```

```
#'tutorial/h
```

A history graph is just a regular graph and can be operated on accordingly. For example, we can view the history graph we just created using the view function.

```
(view h)
```

(



)

5.2 Derivation steps

The history shown above consists of four derivation steps:

1. Step 1: hello
2. Step 2: hello
3. Step 3: World* (two concurrent derivations)
4. Step 4: vine (two concurrent derivations)

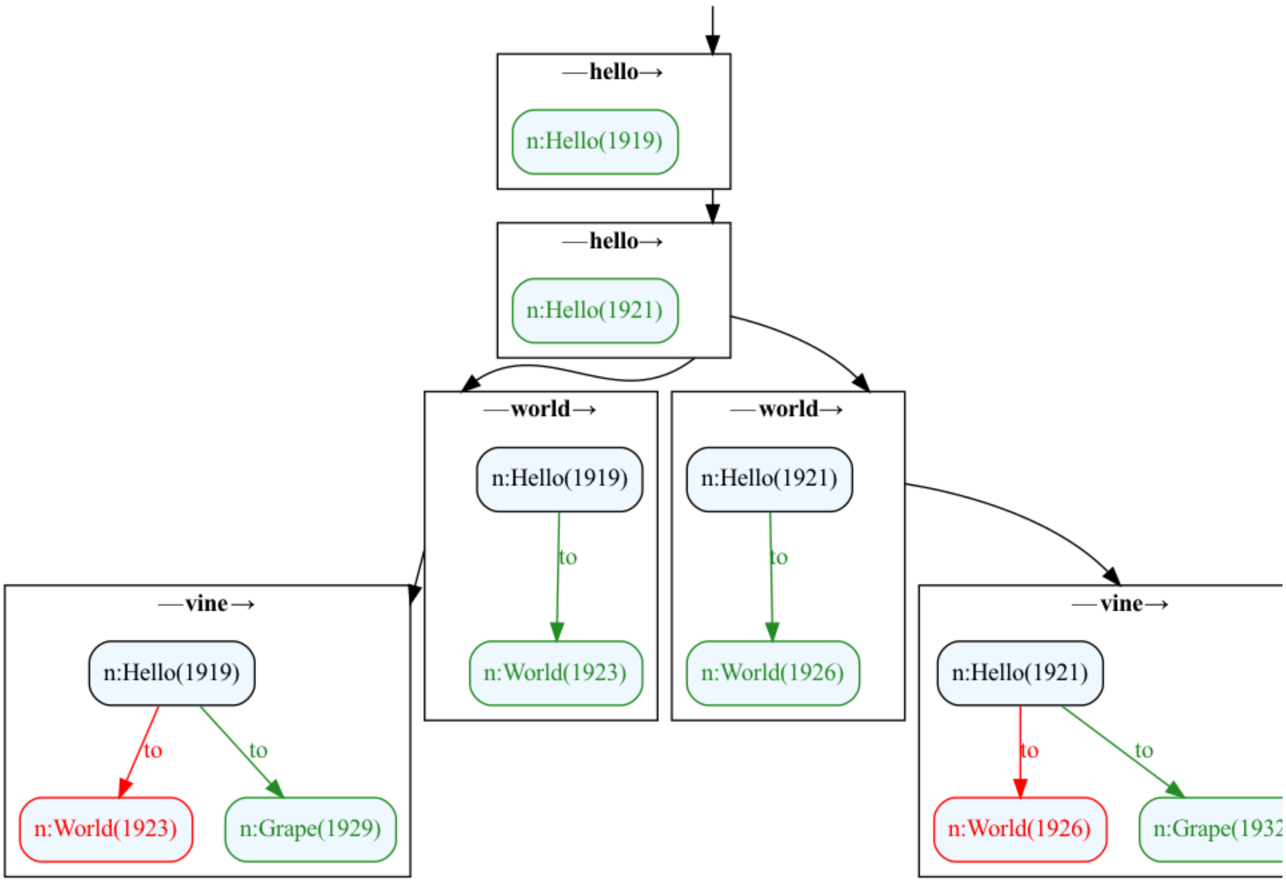
Each of these steps can be represented by a ***grape***. We can use the form `steps` to attain this sequence of ***grapes*** that characterize a history graph:

```
(steps h)
```

```
(("a9048081-4738-4175-bd2a-d27d7f9b7421") ("d241b3a9-d88b-438a-bb6f-e5507a9c8912") ("5143e4d0-6f6a-4cba-a357-69d2d782b6bd" "53d64916-d929-4965-909f-abdb79b95158") ("422db3db-d49f-405c-bbb0-001790a693ac") ("c0df27aa-bd94-4120-a8ec-79661cc9baa5"))
```

We can use the form `viewsteps` to visualize the details for each derivation.

```
(viewsteps (steps h))
```



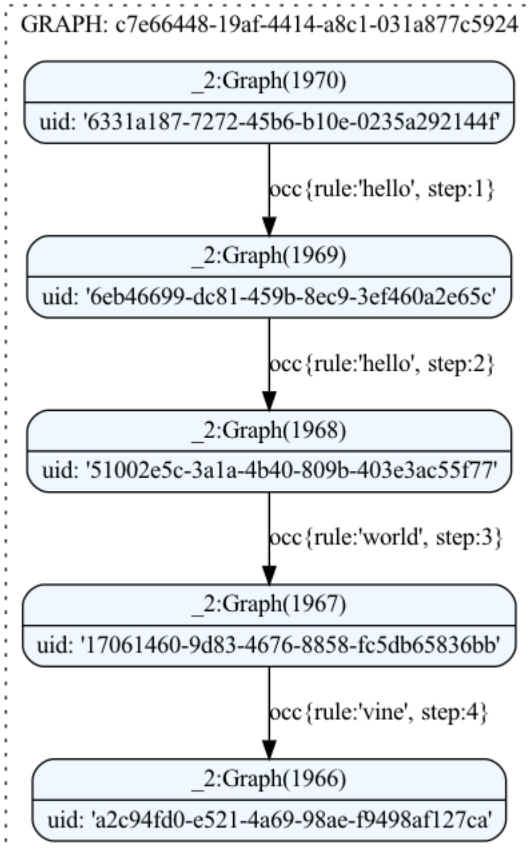
5.3 Traces

In many cases, we are not interested in the entire history graph of a ***grape***. Rather, we may just be interested the derivation steps that led to a particular graph. We call the (linear) sequence of derivation steps that let to a particular graph the **trace** of that graph.

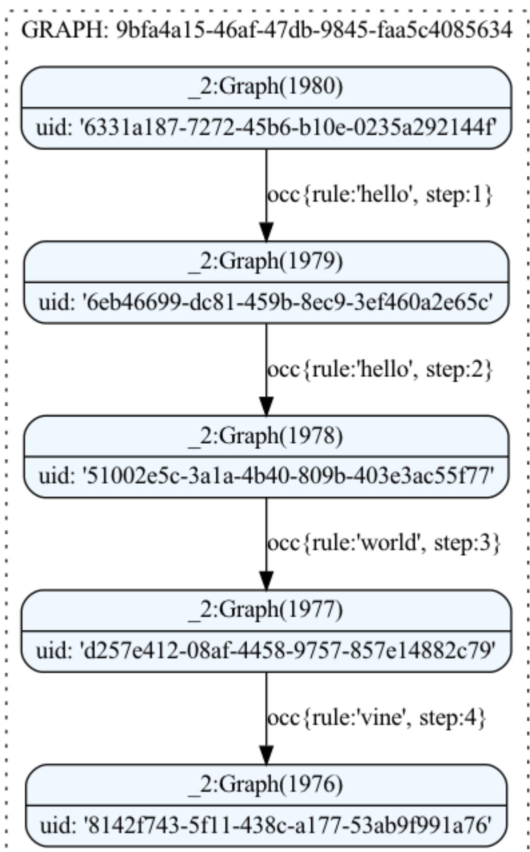
Anologously to history graphs, traces can be represented in graph form (showing linear chains of derivations).

For a given ***grape***, a set of traces for all included graphs can be computed using the `traces` form. As we can see above, there are two traces in our example:

```
(-> (newgrape) hello hello world* vine traces view )
```


GRAPH: c7e66448-19af-4414-a8c1-031a877c5924
_2:Graph(1970)
uid: '6331a187-7272-45b6-b10e-0235a292144f'
occ{rule:'hello', step:1}
_2:Graph(1969)
uid: '6eb46699-dc81-459b-8ec9-3ef460a2e65c'
occ{rule:'hello', step:2}
_2:Graph(1968)
uid: '51002e5c-3a1a-4b40-809b-403e3ac55f77'
occ{rule:'world', step:3}
_2:Graph(1967)
uid: '17061460-9d83-4676-8858-fc5db65836bb'
occ{rule:'vine', step:4}
_2:Graph(1966)
uid: 'a2c94fd0-e521-4a69-98ae-f9498af127ca'

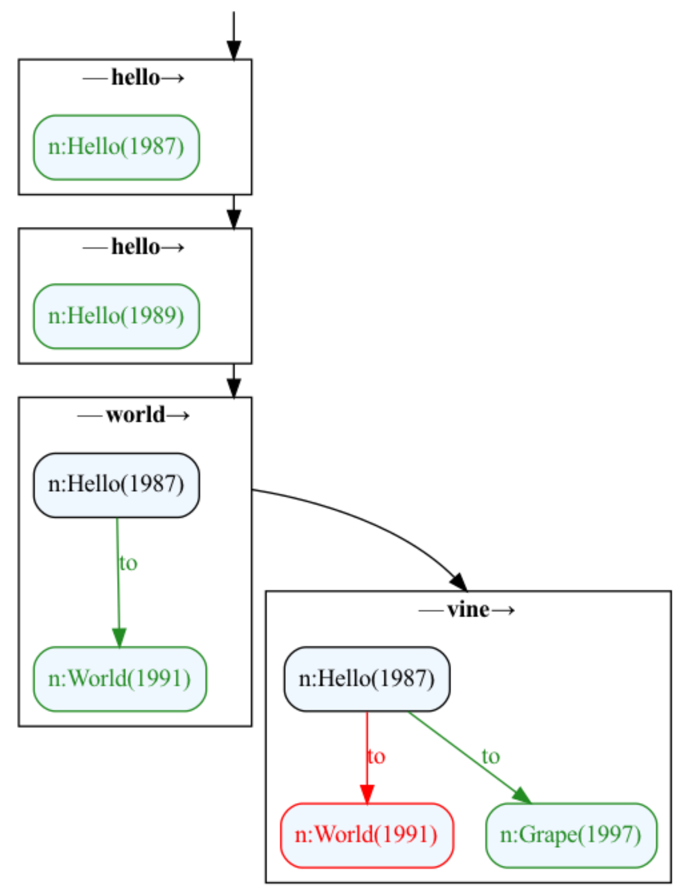
(


GRAPH: 9bfa4a15-46af-47db-9845-faa5c4085634
_2:Graph(1980)
uid: '6331a187-7272-45b6-b10e-0235a292144f'
occ{rule:'hello', step:1}
_2:Graph(1979)
uid: '6eb46699-dc81-459b-8ec9-3ef460a2e65c'
occ{rule:'hello', step:2}
_2:Graph(1978)
uid: '51002e5c-3a1a-4b40-809b-403e3ac55f77'
occ{rule:'world', step:3}
_2:Graph(1977)
uid: 'd257e412-08af-4458-9757-857e14882c79'
occ{rule:'vine', step:4}
_2:Graph(1976)
uid: '8142f743-5f11-438c-a177-53ab9f991a76'

)

Of course, we can also view details of the derivation steps on traces:

```
(-> (newgrape) hello hello world* vine traces steps viewsteps )
```



6. Graph Constraints

Graph constraints define conditions on the state of graphs. Graph constraints are defined based on a notion of an *atomic graph constraint* of form **if** X **then** C, where X describes a pattern to be searched for in a graph and C describes a necessary extension of that pattern for all found matches.

This concept of an atomic graph constraint (*atomic constraint* for short) has been formalized as $\forall(c : X \longrightarrow C)$ (see Orejas et al. for details).

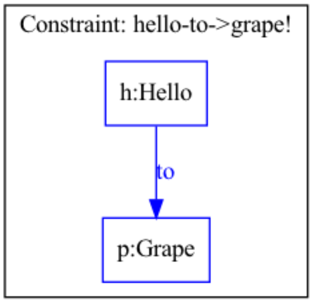
6.1 Basic constraints

An atomic constraint where $X = \emptyset$ is called a *basic atomic constraint* or *basic constraint* for short. It can be written as $\exists C$.

Consider the following basic constraint, which matches a `Hello` node that is connected to a `Grape` node via a `to` edge.

> Note: *By convention, names of constrains should end with a exclamation mark.*

```
(constraint hello-to->grape!
    (node h:Hello)
    (node p:Grape)
    (edge :to h p))
```



6.2 Using constraints to filter *grapes*

Like rules, constraints can be applied to *grapes* and filter out those graphs that do not satisfy the constraint. Applications can be positive or negated.

Let's try:

```
(hello-to->grape! (newgrape))
```

```
()
```

As expected, the above pattern cannot be found in an empty graph. Let's apply the constraint in negated form. (i.e., we assert that the "hello-to->grape!" pattern may not exist). This is done by appending a minus sign (**-**) to the call of the constraint.

```
(hello-to->grape!- (newgrape))
```

```
("f8f841b4-b2d9-4578-a859-55c1820e30a7")
```

We we see above that the empty graph satisfies this negated constraint.

Conversely, let's generate a graph that satisfies the positive constraint (see below):

```
(-> (newgrape) hello world vine hello-to->grape!)
```

```
("b572f89d-2f3f-4574-b26f-2388a5a68bfc")
```

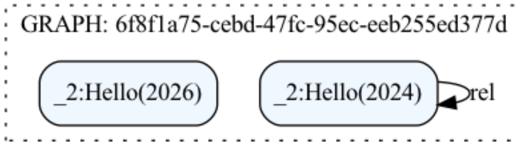
Observe that testing a graph constraint only returns a graph if it satisfies the constraint.

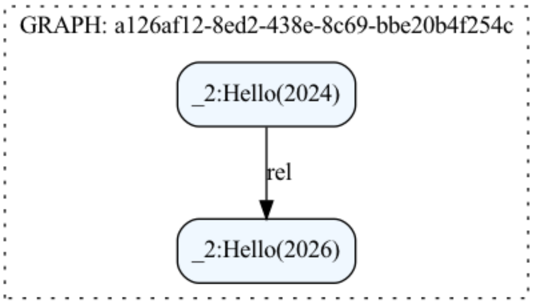
> *Graph constraints can act as filters on **grapes**. They filter out all graphs that do not satisfy the constraint.*

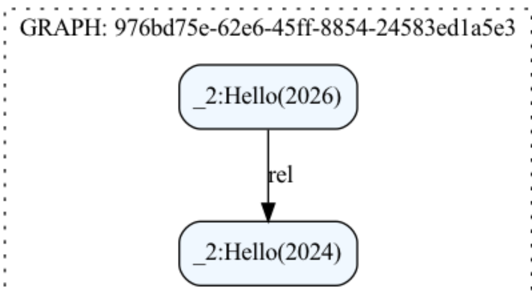
Let's see another example with the `rel` rule defined earlier. As previously noted, doing a starred application of that rule to a graph with two `Hello` nodes produces four possible derivations.

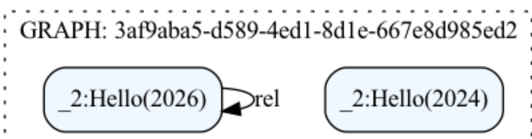
```
(-> (newgrape) hello hello rel* view)
```

(





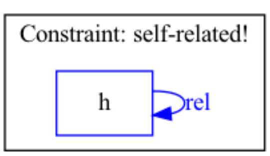




)

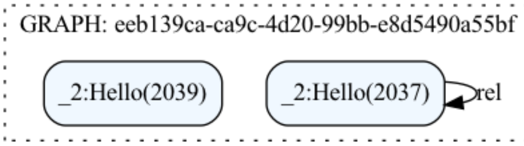
Now let's consider the following constraint `self-related!`, which matches a node that has a relationship with itself:

```
(constraint self-related!
        (node h)
        (edge e:rel h h ))
```

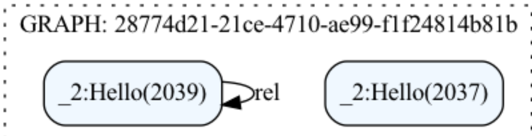


As we can see below, applying the graph constraint filters out those graphs that do not contain a "self-related" node:

```
(-> (newgrape) hello hello rel* self-related! view)
```



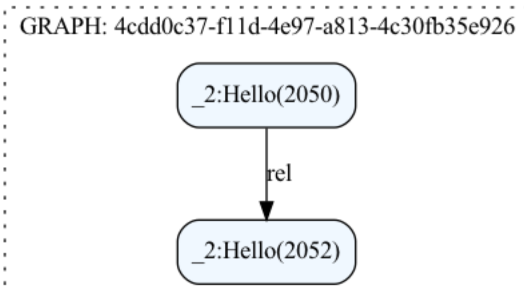
(



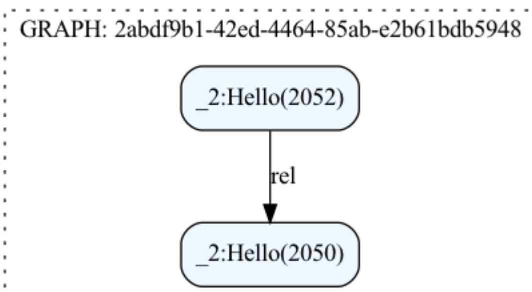
)

Conversely, applying the negated constraint exclude graphs that satisfy the constraint:

```
(-> (newgrape) hello hello rel* self-related!- view)
```



(



)

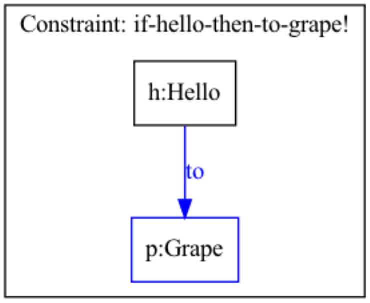
6.3 Atomic Constraints

As described above, atomic constraints have the general form **if X then C**. (Remember that basic constraints are a special case, were the pattern X is empty, intuitively: *if* TRUE *then* X)

Below is an example for an atomic constraint that is *not* a basic constraint. A graph satisfies this constraint if (and only if) all `Hello` nodes are connected to a `Grape` node by a `to` edge.

As we can see, the pattern that belongs to the "if" part is represented in black colour.

```
(cond-constraint if-hello-then-to-grape!
        (IF    (node h:Hello))
        (THEN  (node p:Grape)
                (edge :to h p)))
```



An empty graph satisfies this constraint:

```
(-> (newgrape) if-hello-then-to-grape!)
```

```
("7ae0c332-ca08-49a7-8a0c-6c0b9ce0aab4")
```

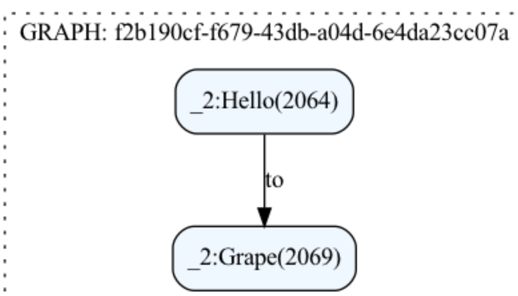
The following graph does not:

```
(-> (newgrape) hello world if-hello-then-to-grape!)
```

```
()
```

But this one does:

```
(-> (newgrape) hello world vine if-hello-then-to-grape! view)
```

(


)

6.4 Constraint clauses

In mathematical logic, a *clause* is a formula of the form $L_1 \vee ... \vee L_n$, where each L_i is a positive or negative literal.

GrapeVine provides the `constraint-clause` form for defining constraints in clausal form, where each literal is a positive or negative atomic constraint. For example, the form below defines a constraint claus (named `self-or-not-hello-to-grape!`) as a disjunction of the two atomic constraints `self-related!` and `if-hello-then-to-grape!`).

```
(constraint-clause  self-or-not-hello-to-grape!
                    [self-related! if-hello-then-to-grape!])
```

```
#'tutorial/self-or-not-hello-to-grape!
```

Constraints defined in clausal form can be applied to Grapes in the same way as atomic constraints, except that their negation is not defined. See below for an example.

```
(-> (newgrape) hello world vine self-or-not-hello-to-grape!)
```

```
("49590265-4add-4fb9-93fa-c120b79bef53")
```

7. Invariants

Above we have seen how constraints can be checked and enforced manually. Constraints can also be attached to graphs as *invariants*.

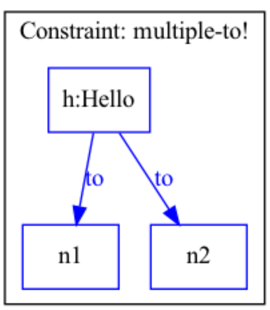
The definition of invariants can have one of two modes:

1. **enforced** invariants are guaranteed by disallowing any graph in the graph process that does not satisfy the invariant. (*This is mainly useful for restricting the behaviour of rules.*)
2. **asserted** invariants are checked upon each occurrence in the graph process and violations are reported through exceptions. (*This is mainly useful for model checking.*)

7.1 Enforced invariants

As an example, let's assume we want to allow only one single `to` edge from a `Hello` node. We can define the following constraint (and then use its negation as an enforced invariant on a graph).

```
(constraint multiple-to!
        (node h:Hello)
        (node n1)
        (node n2)
        (edge :to h n1)
        (edge :to h n2))
```



We can now create a graph where the negation of this constraint is an enforced invariant.

```
(def g (-> (newgrape) (enforce-invariant multiple-to!-)))
```

```
#'tutorial/g
```

Let's try to apply two times the `world` rule:

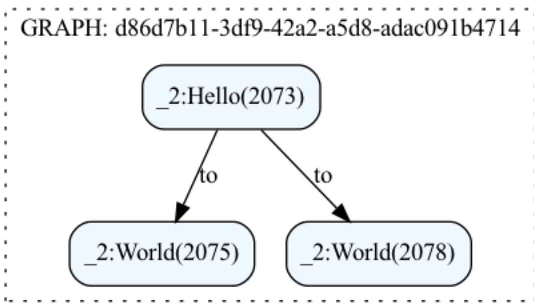
```
(-> g hello world world)
```

```
()
```

As we can see above, there is no result (i.e., the enforced invariant has prevented the second application of `world`).

In comparison, let's do the same on a graph that does not have the enforced invariant. The example below shows that without the enforced invariant, the operation succeeds.

```
(-> (newgrape) hello world world view)
```



7.2 Asserted invariants

In contrast the *enforced* invariants, *asserted* invariants are monitored but not enforced. Occurrences that violated asserted invariants are reported as exceptions. Lets consider the same example, but this time wiht an asserted invariant:

```
(def g (-> (newgrape) (assert-invariant multiple-to!-)))
```

```
#'tutorial/g
```

Let's cause a violation of the assertion:

```
(-> g hello world world)
```

8. Graph Queries

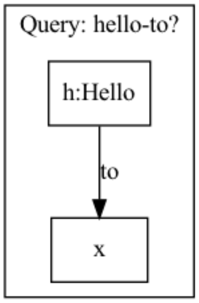
Graph queries are used to extract data from a graph (for further processing in the host language Clojjure) and to visualize parts of interest.

By convention and to distinguish queries from rules and constraints, names of queries should be appended with a question mark (**?**).

Like rules and constraints, queries can be applied to ***grapes***, but they return a relation for each graph in the ***grape***.

Consider the following query as an example:

```
(query hello-to? []
    (node h:Hello)
    (node x)
    (edge :to h x))
```




```
(-> (newgrape) hello world hello-to?)
```

```
(({:_g {:uid "0b0496a2-81f4-49e6-a08c-0cee51fa7a8c"}, :h {:_fp "CC3EED4CFB833776B2B02CD1766FC47C", :id
1623, :labels ("Hello" "__Node")}, :x {:_fp "13D744EF67DB02EC47A7D5261D2270A2", :id 1625, :labels
("__Node" "World")}, :_imQNiowEIc {:tar 1625, :src 1623, :_fps "CC3EED4CFB833776B2B02CD1766FC47C",
:_fpd true, :_fpt "13D744EF67DB02EC47A7D5261D2270A2", :_fp "5994B27BFABFBAE55A90D87A6375EDC3", :id
1626, :labels ("__Edge" "to")}}))
```

There is only one graph in the Grape that is fed to the query `hello-to?`, so the result is a set that contains only a single relation. Moreover, that relation only contains a single tuple. Extracting information of interest simply uses Clojure to navigate this set of relations. For example, the following extracts the ID of the first `h` node in the first graph:

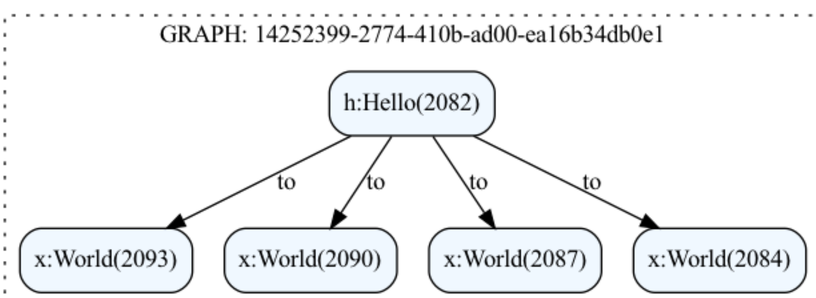
```
(-> (newgrape) hello world hello-to? first first :h :id)
```

```
525
```

8.1 Visualizing query results

Query results can be visualized using the `viewquery` form. Consider the following simple example of a query for "Hello" nodes and a graph that has two such nodes.

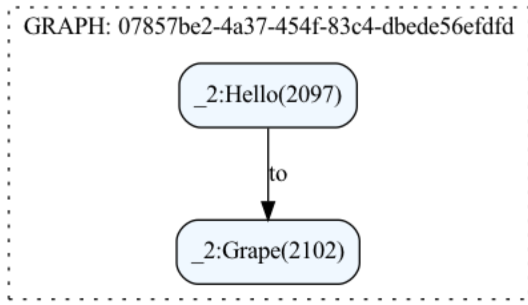
```
(-> (newgrape) hello world world world world hello-to? viewquery)
```

(


)

GrapVine comes with a built-in query `_any?` which matches any element in a graph (returning all elements as a result).

```
(-> (newgrape) hello world vine _any? viewquery)
```



Indeed, the function `view(g)` is merely a shortcut for writing `(viewquery (_any? g))`.

9. Transactions

The classical notion of transactions revolves around the ACID (atomic, consistent, isolated, durability) properties.

9.1 Durability

Since *GrapeVine* uses a database, all graph productions are automatically made durable. However, not all graphs may be of interest. Moreover, identifying graphs with automatically generated IDs is inconvenient.

Graphs can therefore be tagged with semantic names using the `commit` form. In the following example, we tag the resulting graph with the name "*mygraph*"

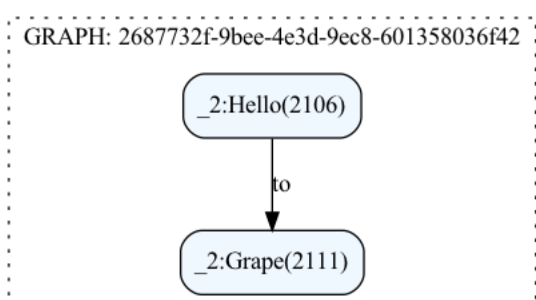
> **Note: `commit` tags an *individual* graph, that needs to be selected from a *grape*.** (In the example below, we use commit the first (only) graph in the ***grape***.)

```
(-> (newgrape) hello world vine)
(commit (first _) "mygraph")
```

```
"2687732f-9bee-4e3d-9ec8-601358036f42"
```

Once tagged, we can simply call the `graph` form to recall the graph by its semantic name, for example:

```
(-> (grape "mygraph") view)
```



The names given to graphs using the commit form must be unique. Trying to tag another graph with the same name will result in an error:

```
(-> (newgrape) hello first (commit "mygraph"))
```

```
Caught exception: org.neo4j.driver.exceptions.ClientException: Node(1651) already exists with label `__Graph`
and property `tag` = 'mygraph'
```

```
nil
```

Committed graphs can be "uncommitted" using the `uncommit` form:

```
(uncommit "mygraph")
```

```
"2687732f-9bee-4e3d-9ec8-601358036f42"
```

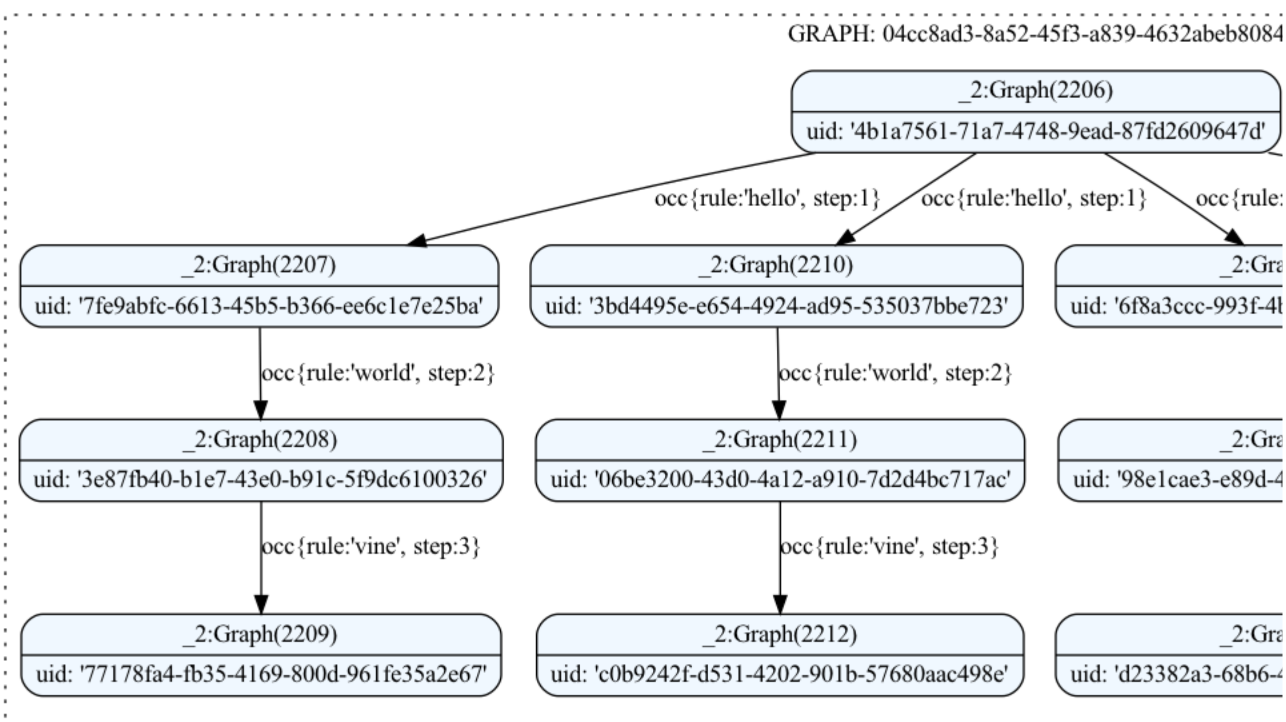
9.1.1 Rolling back uninteresting derivations

We may want to roll back uninteresting graph derivations, for example to free up resources or to prune the graph derivation history. We can do so by using the `rollback` form. **Warning**: calling rollback will remove *all* graphs that are not in the trace of any *committed* (tagged) graph.

The following example illustrates this behaviour. In the code block below, we create a graph history with four different branches, originating from the same empty start graph. We commit a tag for the end-result of one of these derivations.

```
(def start (newgrape))

(-> start hello world vine)
(-> start hello world vine)
(-> start hello world vine)
(-> start hello world vine)
(commit (first _) "mygraph")
(-> _ history view)
```

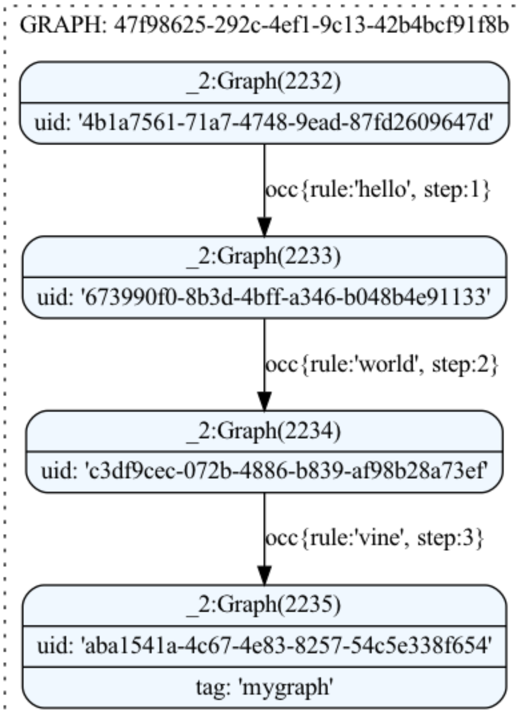
(



)

Calling `rollback` will remove all graphs that "do not lead up to" a committed (tagged) graph in the graph process. (see below)

```
(rollback)
(-> _ history view)
```



( )

10. Control structures

Control structures are the primitives we use to control the application of graph transformations, i.e., to write *graph programs*. Since ***GrapeVine*** is embedded into the Clojure, all the regular Clojure control structures are available to program with graph transformations. The ***GrapeVine*** approach is to leverage these existing control structures as much as possible rather than introducing new "syntactic sugar".

We have already seen that we use simple function calls for invoking graph transformations. Calling a rule. We have also seen that the Clojure *threading macro* (->) is well-suited for *sequential composition* of graph transformations, tests and queries. Of course, we could always alternative Clojure notation (as we did before as well), e.g, we could use the classical way of nesting function calls (i.e., (f1 (f2)))

The following table provides an overview of the typical ***GrapeVine*** control structures:

Operation	*GrapeVine* construct	Comments
transformation with ***r*** (non-deterministic)	`r`	or `(r ..)` if rule has parameters
transformation with rule ***r*** (deterministic)	`r*`	or `r*(..)` if rule has parameters
distinct - remove all duplicate graphs	`distinct`	
selection of graphs that satisfy constraint ***c***	`c`	

selection of graphs that satisfy constraint $\neg c$	`c-`	
constrain graphs with invariants $c1,..,Cn$	`(enforce-invariant [c1, .., cn])`	transformations that would violate invariants are disallowed
assert invariants $c1,..,cn$	`(assert-invariant [c1, .., cn])`	invariants are monitored and violations reported
unconstrain - remove an invariants $c1,..,Cn$ from graphs	`(remove-invariant [c1, .., cn])`	
sequence of operations $o1,..on$	`(-> o1 .. on)`	all alternatives are explored concurrently
alternative of operations $o1,..on$	`(||o1 .. on)`	
loop of operations $o1,..on$ until a graph is found that satisfies constraint c - or until no further progress is possible	`(->* c o1 .. on)`	filters out duplicate graphs (using the *distinct*)
loop! of operations $o1,..on$ until a graph is found that satisfies constraint c - or until no further progress is possible	`(->*! c o1 .. on)`	no restriction when exploring similar graphs

10.1 An example

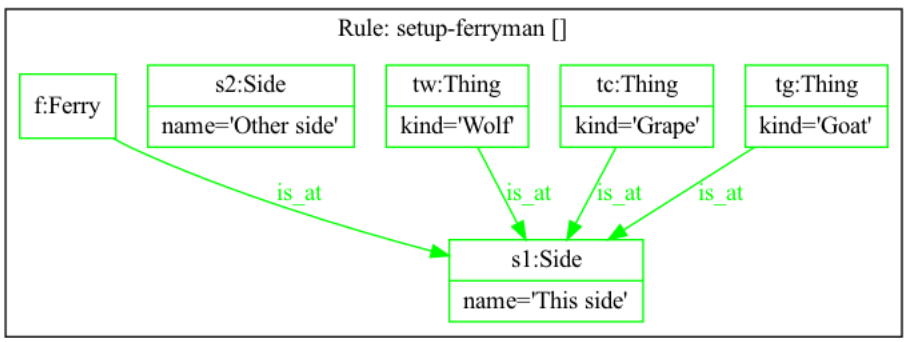
Consider the well-known Wolf-Goat-Cabbage (well, Wolf-Goat-Grape problem) as an example.

Below is specification of the problem. Rule `setup-ferryman` creates the starting situation. The ferryman can either take one item to the other side (rule `ferry_one_over`) or he can cross without cargo (rule `cross_empty`).

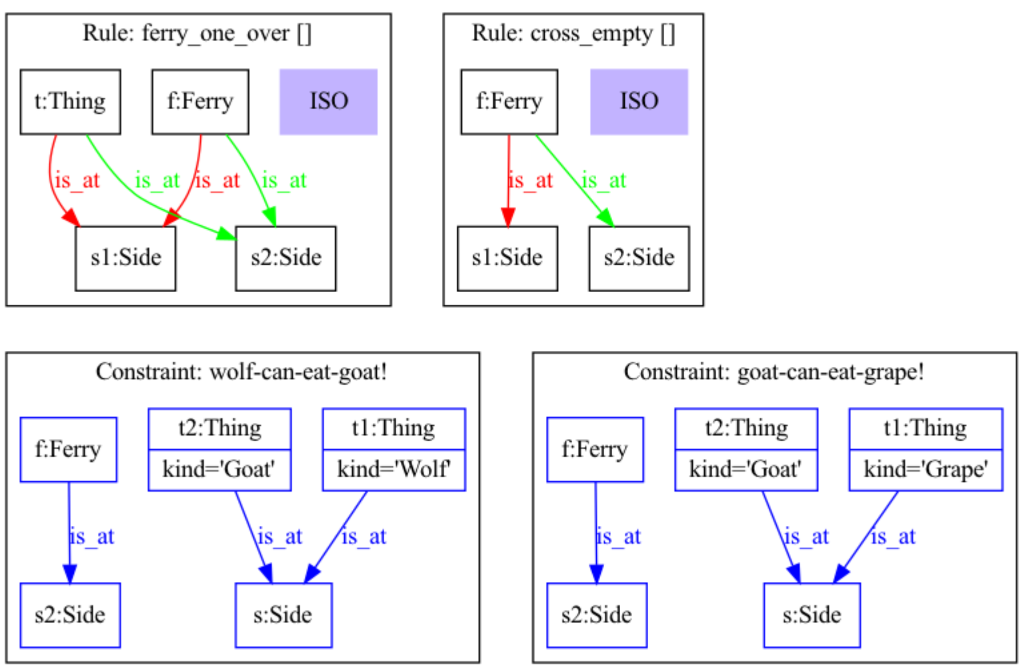
Then we have defined three graph queries. Two queries for testing the dangerious situations (wolf can eat goat or goat can eat grape) and one test to check whether we have solved the ridle (all on the other side).

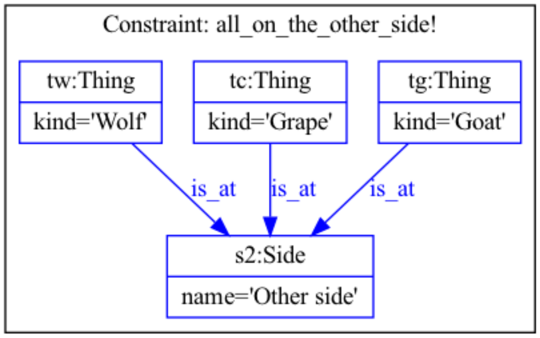
```
(list

(rule setup-ferryman []
  (create
    (node tg:Thing {:kind "'Goat'"})
    (node tc:Thing {:kind "'Grape'"})
    (node tw:Thing {:kind "'Wolf'"})
    (node s1:Side {:name "'This side'"})
    (node s2:Side {:name "'Other side'"})
    (node f:Ferry)
    (edge :is_at tg s1)
    (edge :is_at tc s1)
    (edge :is_at tw s1)
    (edge :is_at f s1)
    ))

(rule ferry_one_over []
  (read :iso
    (node s1:Side)
    (node s2:Side)
    (node f:Ferry)
    (node t:Thing)
    (edge et:is_at t s1)
    (edge e:is_at f s1))
  (delete e et)
  (create
    (edge :is_at f s2)
    (edge :is_at t s2)))

(rule cross_empty []
  (read :iso
    (node s1:Side)
    (node s2:Side)
    (node f:Ferry)
    (edge e:is_at f s1))
  (delete e)
  (create
    (edge :is_at f s2)))
```

```
(constraint wolf-can-eat-goat!
           (node t1:Thing {:kind "'Wolf'"})
           (node t2:Thing {:kind "'Goat'"})
           (node s:Side)
           (node s2:Side)
           (edge :is_at t1 s)
           (edge :is_at t2 s)
           (node f:Ferry)
           (edge :is_at f s2))

(constraint goat-can-eat-grape!
           (node t1:Thing {:kind "'Grape'"})
           (node t2:Thing {:kind "'Goat'"})
           (node s:Side)
           (node s2:Side)
           (edge :is_at t1 s)
           (edge :is_at t2 s)
           (node f:Ferry)
           (edge :is_at f s2))

(constraint all_on_the_other_side!
           (node tg:Thing {:kind "'Goat'"})
           (node tc:Thing {:kind "'Grape'"})
           (node tw:Thing {:kind "'Wolf'"})
           (node s2:Side {:name "'Other side'"})
           (edge :is_at tg s2)
           (edge :is_at tc s2)
           (edge :is_at tw s2))
)
```



(





)

Below is a program that demonstrates the use of the above control structures to solve the ferryman problem:

We begin as usual with creating our start graph (grape) (line 1).

Line 2 uses are new **looping** macro. It first takes a constraint to check whether the goal has already reached for *some* graph in the ***grape***. If no graph in the ***grape*** satisfies the constraint, the loop continues. (In other words, the looping macro has an "until" semantics.).

Line 3 explores all possible ferry moves in parallel.

Finally, the (negative) constraint checks in lines 3 and 4 filter out the dangerous situations.

```
1 (-> (newgrape) setup-ferryman
2     (->* all_on_the_other_side!
3          (|| ferry_one_over* cross_empty*)
4          wolf-can-eat-goat!-
5          goat-can-eat-grape!-))
```

```
("a6213cfb-e2b7-4dd5-a6de-34a399abdf70")
```

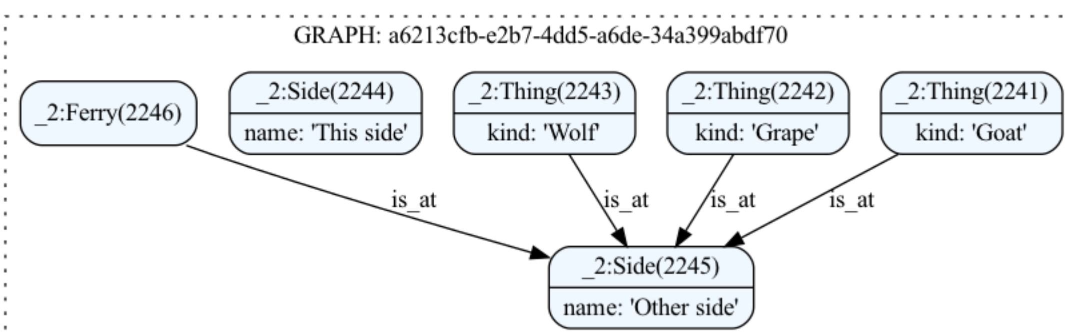
```
(set '(1 2 3 2))
```

```
#{1 3 2}
```

We can confirm that a solutions was indeed found.

```
(view _)
```

(



)

It should be clear that the above program does a *breadth-first exploration* of the possible moves for the ferryman until a solution is found.

What if there *is* no solution? Let's modify the program and only allow the ferryman to cross *with* cargo (i.e., no empty ferries). A naive breath-first exploration would run forever, since moves with the goat are always safe but will never achieve the desired solution. The result would be an infinite graph process. The program should never terminate (disregarding resource limitations). Let's try this out:

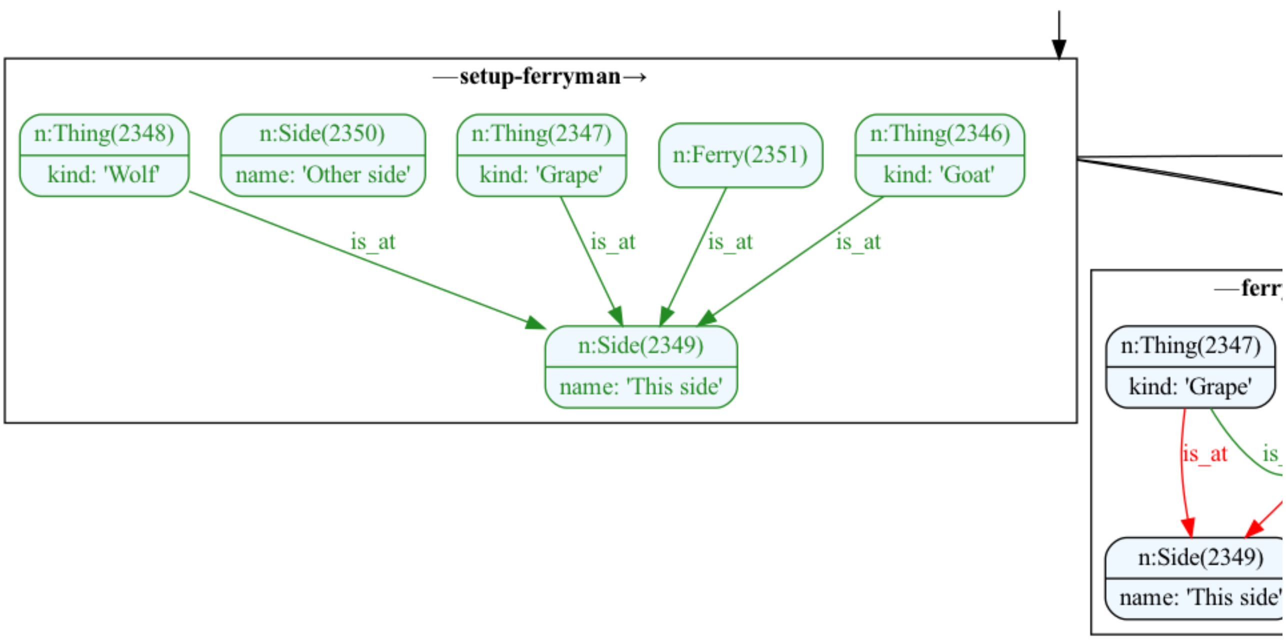
```
(-> (newgrape) setup-ferryman
    (->* all_on_the_other_side!
         ferry_one_over*
         wolf-can-eat-goat!-
         goat-can-eat-grape!-))
```

```
()
```

As we can see, the program does indeed terminate and returns an empty ***grape*** (no solution). The reason for this is that the ***GrapeVine*** looping macro compares graphs in the graph history for previously seen graphs and does not further explore graphs that are similar with graphs that have been seen before.

Let's have a look at the graph history for the above program:

```
(def start (-> (newgrape) setup-ferryman))
(->* start all_on_the_other_side!
         ferry_one_over*
         wolf-can-eat-goat!-
         goat-can-eat-grape!-)
(-> start history steps viewsteps)
```

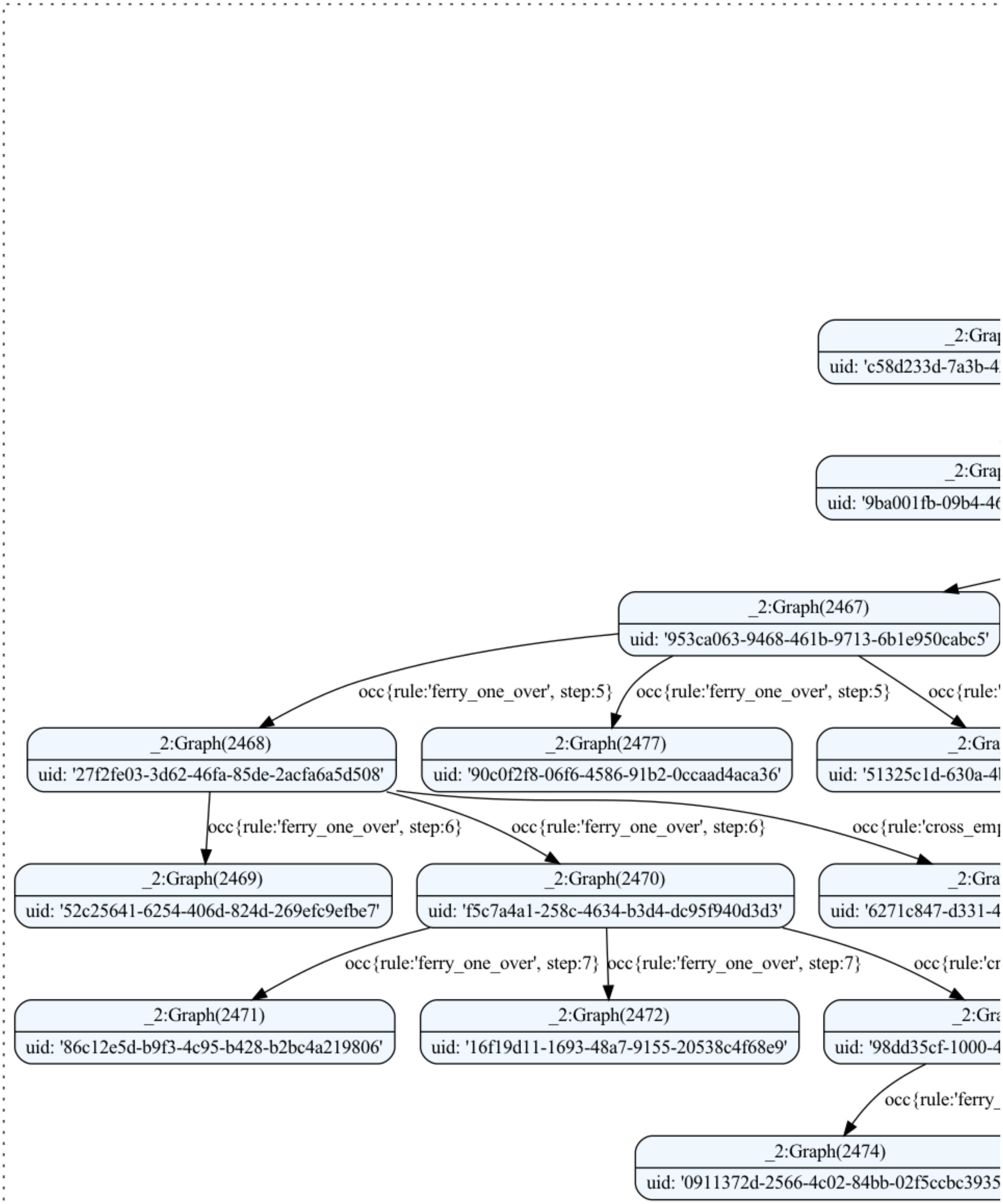


Indeed, there are three possible "cargo" moves of the ferry from the initial state. Two of them are "unsafe" and one of the (transporting the goat) is safe. From there, the only possible move is to transport the goat again. At that point, the resulting graph is identical to the start (up to isomorphism).

Let's look at the history of the original program again (which *has* a solution). Below we see that the breadth-first search does not explore graphs that have already been seen on the process (up to isomorphism).

```
(-> (newgrape) setup-ferryman
    (->* all_on_the_other_side!
         (|| ferry_one_over* cross_empty*)
         wolf-can-eat-goat!-
         goat-can-eat-grape!-))

(-> _ history view)
```

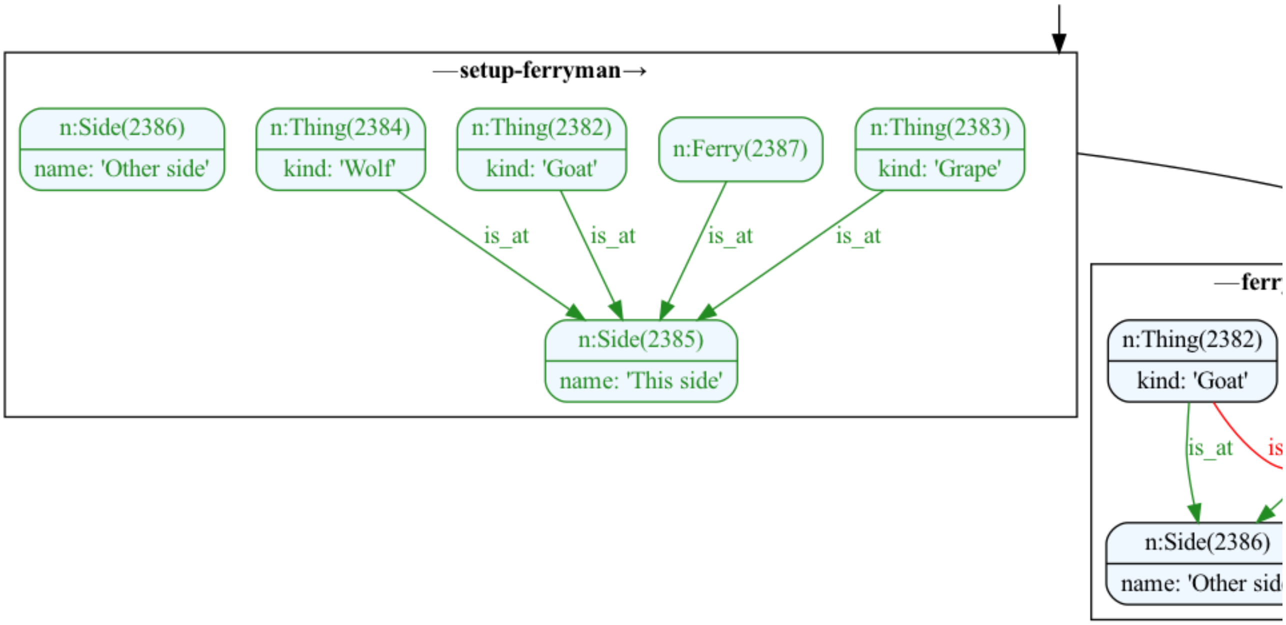
```
(
```




```
)
```

Viewing the details on the full history graph creates a big image. Rather, we are just interested in the trace that leads to the solution:

```
(-> _ traces steps viewsteps)
```


—setup-ferryman→
n:Side(2386)
name: 'Other side'
n:Thing(2384)
kind: 'Wolf'
n:Thing(2382)
kind: 'Goat'
n:Ferry(2387)
n:Thing(2383)
kind: 'Grape'
is_at
is_at
is_at
is_at
n:Side(2385)
name: 'This side'
n:Thing(2382)
kind: 'Goat'
is_at
n:Side(2386)

If, for some reason, we do not want to limit the application of graph productions to graphs that have already been seen in the graph process (up to isomorphism), we can use the alternative looping macro **(->!*)**. The example below shows that the resulting graph process is significantly larger.

> Note: *this will also take longer... - be patient*

```
(-> (newgrape) setup-ferryman
   (->*! all_on_the_other_side!
         (|| ferry_one_over* cross_empty*)
         wolf-can-eat-goat!-
         goat-can-eat-grape!-))
```

```
("bd5e8f82-b31e-4572-a2ae-0877691b94ce" "27477813-fe84-4316-83c3-029fb23678db")
```

Let's see how many graphs were produced in the above computation. Do do so, we can simply "flatten" the steps involved:

```
(-> _ history steps flatten count)
```

```
216
```

We see that 216 graphs were generated in the computation that does not check for graph similarity. This compares to 27 for the computation that does (see below):

```
(-> (newgrape) setup-ferryman
   (->* all_on_the_other_side!
         (|| ferry_one_over* cross_empty*)
         wolf-can-eat-goat!-
         goat-can-eat-grape!-))

(-> _ history steps flatten count)
```

```
27
```

Solving the Ferryman Problem with Invariants

Alternatively, we can solve the Ferryman Problem using invariants. We define avoidance of the two dangerous moves as enforced (negative) invariants.

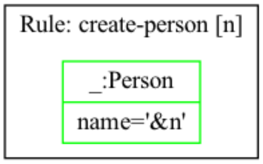
```
(-> (newgrape)
  (enforce-invariant wolf-can-eat-goat!- goat-can-eat-grape!-)
  setup-ferryman
  (->* all_on_the_other_side! (|| ferry_one_over* cross_empty*)))
```

```
("904dd24c-c1ff-4d95-b791-b8b85c0b2e4f")
```

11. Rules with parameters

Rules can be parameterized. For example, the following rule creates person nodes with a given name.

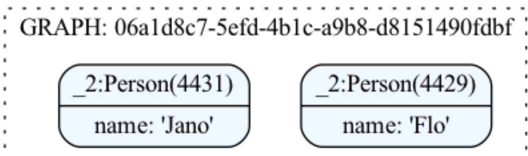
```
(rule create-person [n]
      (create (node :Person :asserts {:name "'&n'"})))
```



Formal parameters (like `n` above) must be actualized when the rule is applied. The rule application below creates two `Person` nodes with names "Flo" and "Jano", respectively.

Note that the expression `&n` is replaced with the value of the actual parameter `p` at rule execution time.

```
(-> (newgrape) (create-person "Flo") (create-person "Jano") view)
```

(


)