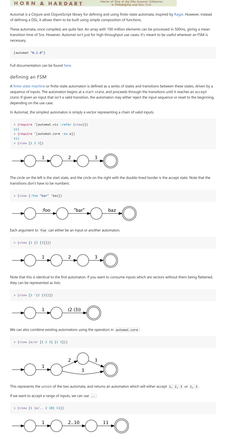
Werthen committed on GitHub Merge pull request #46 from hashafy/fix		Latest commit d3f4ba2e in 10 days
docs	fix image name	2 years ago
tests/automat	.gitignore fix file name for the other classes	2 months ago
src/automat	Pointed to dji	9 months ago
.gitignore	Clojure 1.7 works	9 months ago
.travis.yaml	remove java 6 (OpenJdk)(jar does not support it and java 6 is o	2 years ago
README.md	update readme	3 months ago
project.clj	mark 0.2.0	3 months ago

README.md



Automat is a Clojure and ClojureScript library for defining and using finite state automata, inspired by Rapid. However, instead of defining a DSL, it allows them to be built using ordinary Clojure compositions of functions.

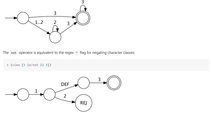
These automata, once compiled, are just functions, which can be used to process a sequence of inputs, resulting in a sequence of outputs. Because the automata keeps no state of its own, and proceeds through the transitions until it reaches an accept state, it gives an impression that isn't a valid transition, the automaton may either reject the input sequence or reset to the beginning, depending on the use case.

In Automat, the simplest automaton is simply a vector representing a chain of valid inputs.

```
[:require [automat.viz :refer [view]]
[automat.core :as a]]
[:require [automat.core :as a]]
a/c1 = [:a :b :c]
```

The circle on the left is the start state, and the circle on the right with the double-lined border is the accept state. Note that the transitions don't have to be numbers.

```
[:require [:foo "bar" "baz"]]
```



Each argument to `:foo` can either be an input or another automaton.

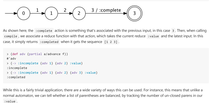
```
[:require [:1 [2 [3]]]]
```

Note that this is identical to the first automaton. If you want to consume inputs which are vectors without them being flattened, they can be represented as lists:

```
[:require [:1 '(:2 (:3))]]
```

We can also combine existing automatons using the operation in `automat.core`.

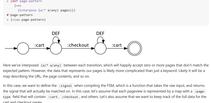
```
[:require [:or [1 2 3] [2 [3]]]]
```



This represents the union of the two automata, and returns an automaton which will either accept `1`, `2`, `3` or `2`, `3`.

If we want to accept a range of inputs, we can use `:..`:

```
[:require [:in [\. \# \$ \%]]]
```

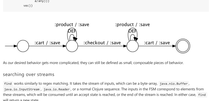


This will accept `\.`, `\#`, `\$`, `\%`, `\:`, and so on. If we subsequently want to narrow this, we can use `and`:

```
[:and
  [:in
    [:..
     \a \z]]
  [:in [\a \e \i \o \u]]]
```

This represents the intersection of two automata. In this case going as an automaton that either accepts `\a`, `\e`, `\i`, or `\o`, `\u`. Note that if the intersection is empty, this will give us an automaton that cannot accept anything.

```
[:and [:0 [:1 [:2 [:3 [:4 [:5]]]]]
      [:0 [:1 [:2 [:3 [:4 [:5]]]]]]]
```



This represents the difference between the automata in this case an automata that accepts `[1,2,3]`. Here the inputs `1`, `2`, `3`, the operations `:*`, `:+`, and `:?` behave as they do in regular expressions.

```
[:and [:0* [:1 [:2* [:3 [:4* [:5]]]]]
      [:0 [:1 [:2 [:3 [:4 [:5]]]]]]]
```

This gives us an automaton that accepts zero or one `:1` inputs, zero or more `:2` inputs, and one or more `:3` inputs.

```
[:and [:0 [:1 [:2 [:3]]]]
      [:0* [:1 [:2* [:3 [:4* [:5]]]]]]]
```

The `not` operator is equivalent to the regex `.` flag for negating character classes:

```
[:and [:in [\a\z\0 \1 \2]]
      [:not [:in [\0 \1 \2]]]]
```

This will optimize the FSM: emit code that processes `\a` and `\z` and `\0`. The resulting compiled FSM can be interacted with the advance, `fold`, `greedy-fold`. These operations are pure, and can be safely called on multiple threads.

Notice that when we visualize a compiled FSM, the states themselves are now labeled:

```
[:require [:automat.viz :refer [view]]
[:automat.core :as a]]
[:require [:automat.core :as a]]
[:and [:0 [:1 [:2 [:3]]]]
      [:0* [:1 [:2* [:3 [:4* [:5]]]]]]]
a/c = [:and [:0 [:1 [:2 [:3]]]]
            [:0* [:1 [:2* [:3 [:4* [:5]]]]]]]
a/c1 = (a/compile a/c)
[:accepted? true
:checkpoint nil
:state-index 0
:start-index 0
:stream-index 1
:value nil]
```

This returns a map representing the FSM state produced by the input.

field	description
accepted?	whether the FSM is in an accept state
state-index	the numeric identifier for the current FSM state, as shown by `view`
start-index	the stream index where the FSM last initiated
stream-index	the current stream index
checkpoint	the previous match, only non-when used with `greedy-fold`
value	the current reduce value, explained below

This map can be passed back into `advance` as the `state` parameter:

```
[:def a/c [:accept :a :b :c]
a/c = [:accept :a :b :c]
a/c1 = (a/advance a/c [:a :b :c])
[:accepted? true
:checkpoint nil
:state-index 1
:start-index 0
:stream-index 3
:value nil]
```

Notice that `advance` either accepts the map descriptor returned by a previous call to `advance`, or an arbitrary value, representing an initial value at the beginning of the FSM. The FSM reasons for this are shown below.

If we pass an input into `advance` that isn't accepted by the FSM, it will throw an exception. Alternately, we can pass an additional field into advance specifying what it should return if the input is rejected:

```
[:a/advance # nil a/c
ilegalArgumentException
[:accepted? false :error?
true]
a/c1 = (a/advance a/c a/c)
[:accepted? false
:checkpoint nil
:state-index 0
:start-index 0
:stream-index 1
:value nil]
```

Normal finite state automata are represented by simple integers in the `state-index`. The additional fields give us some additional capabilities of special interest is the `value` field, which allows us to treat the FSM as a pattern-matching reduction over a stream of values.

We can define reduction operations without our FSM using the `a` function.

```
[:def f (a/reduce
          [:0 [:1 :0 :1 :complete]
              [:reduce* [:complete (fn [state input] (conj state[:1]))]]]
a/c = (a/compile f)
[:0 [:1 :0 :1 :complete]
    [:reduce* [:complete (fn [state input] (conj state[:1]))]]]
```


We have two states, the `:complete` action is something that's associated with the previous input. In this case `:1`. Then, when calling `:complete`, we associate a reduce function with that action, which takes the current reduce `:value` and the latest input. In this case, it simply returns `:conjoined` where it gets the original `:1` [`:1`].

```
[:def a/c [:accept :a :b :c]
a/c = [:accept :a :b :c]
a/c1 (:0 [:reduce (fn [v] (inc v)) 0 :value]
[:0 [:reduce (fn [v] (inc v)) 0 :value]
a/c2 = (a/compile a/c1 [:a :b :c] :value)
:conjoined
```

While this is a fairly trivial application, there are a wide variety of ways this can be used. For instance, this means that unlike a normal automaton, we can let whether a list of parentheses are balanced, by tracking the number of un-closed parens in our reduce.

a short example

For a more real world use case, consider tracking browsing behavior on an online store. We want to see when the customer visits the cart, begins to checkout, but then returns back to the cart without having completed the checkout. Seeing this indecisive behavior, we can make a special offer.

This means that we want to track this pattern:

```
[:cart :checkout :cart]
```

However, the user may wander around a bit, so we want to allow for arbitrary pages in between:

```
[:def pages [:cart :checkout :cart]]
# pages
[:def page-pattern
      [:0*
        [:intersect [:in* #(deq1 pages)]]]
# page-pattern
[:0* [:0* [:intersect
           [:in* #(deq1 pages)]]]]
```


Here we've interpreted `[:in* #(deq1 pages)]` between each transition, which will happily accept zero or more pages that don't match the expected pattern. However, the data that represents our pages is likely more complicated than just a keyword. Likely it will be a map describing the URL, the page contents, and so on.

In this case, we want to define the `:original` when compiling the FSM, which is a function that takes the raw input, and returns the original that will be accepted by the automaton. In this case, we'll assume that each pageview is represented by a map with a `:page` type. Well that will contain `:cart`, `:checkout`, and others. Let's also assume that we want to keep track of the full state for the cart and checkout pages.

This is all fairly straightforward:

```
[:def page-pattern
      [:0* [:cart :checkout :cart]
           [:0* #(:in [:# [:cart :checkout]]
                  :original
                  (fn [state] [:# (:page state)]))]]
# page-pattern
[:0* [:0* [:cart :checkout :cart]
          [:0* #(:in [:# [:cart :checkout]]
                 :original
                 (fn [state] [:# (:page state)]))]]]
# pages
[:def pages [:cart :checkout :cart]]
# pages
[:def page-pattern
      [:0*
        [:intersect [:in* #(deq1 pages)]]]
# page-pattern
[:0* [:0* [:intersect
           [:in* #(deq1 pages)]]]]
```


First, we take each of the notable pages and add a `:cart` action to them. Then we intercept the add-cart machine to track. Then we define reduce functions that assoc the full page data under `:cart` or `:checkout`, and add `:original` to that we continue through the whole process.

Notice that the first action doesn't have an associated input, it simply happens upon entering the FSM. Notice too that the final input has two associated actions. This is perfectly safe, but the order in which the actions occur is not well-defined, so they should always be independent of each other. If the same action is defined twice on a given transition, it will only be performed once.

This is a fairly verbose way to define simple behavior, but it's worth noting that almost everything except for the `[:cart :checkout :cart]` sequence can be abstracted away into higher-order functions. Clojure's core specifically scaffolding, this can be a powerful declarative mechanism.

It's also easy to extend. Let's say that we want to save any `:product` pages the user visits, to better inform our offer. This is a fairly small modification:

```
[:def page-pattern
      [:0* [:cart :checkout :cart]
           [:0* #(:in [:# [:cart :checkout]]
                  :original
                  (fn
                    [:or
                     [:cart
                      [:0* [:product
                            a/any]]]
                     :any]))]]
```


As our desired behavior gets more complicated they can still be defined as small, composable pieces of behavior.

searching over streams

`:find` works similarly to regex matching. It takes the stream of inputs, which can be a byte array, `java.io.BufferedReader`, `java.io.InputStream`, `java.io.Reader`, or a normal Clojure sequence. The inputs in the FSM correspond to elements from the stream. Searches will be consumed until an accept state is reached, or the end of the stream is reached in either case. `:find` will return a map with the following keys:

If a match is found, `:accepted?` will be true, `:start-index` will be the point within the stream where the match begins, and `:stream-index` will be the point where the match ends.

```
[:find # nil [1 2 3]
[:accepted? true, :checkpoint nil, :start-index 0, :stream-index 3, :value nil]]
```

If `:accepted?` is false the `:start-index` will describe where the FSM last restarted. This state can be passed into `:find` with a new stream of inputs.

```
[:find # nil [1 2 3]
[:accepted? false, :checkpoint nil, :start-index 3, :stream-index 3, :value nil]]
```

In either case, `:state-index` will describe where we are in the actual compiled automaton. This can be combined by calling `automat.viz/view` on a compiled FSM.

When defining patterns that are subset of each other, such as `[:or [1 2] [1 2 3]]`, `:find` will always return upon reaching the shortest of the two. If we want to match on the longest, we can use `:greedy-find`. Instead finds their `:greedy-find` with the same parameters, but will be consumed until an accept state is reached, or the end of the stream is reached in either case. `:find` will return a map with the following keys:

If a match is found, `:accepted?` will be true, `:start-index` will be the point within the stream where the match begins, and `:stream-index` will be the point where the match ends.

```
[:greedy-find # nil [1 2 3]
[:accepted? true, :checkpoint nil, :start-index 0, :stream-index 3, :value nil]]
```

If `:accepted?` is false the `:start-index` will describe where the FSM last restarted. This state can be passed into `:find` with a new stream of inputs.

```
[:greedy-find # nil [1 2 3]
[:accepted? false, :checkpoint nil, :start-index 3, :stream-index 3, :value nil]]
```

In either case, `:state-index` will describe where we are in the actual compiled automaton. This can be combined by calling `automat.viz/view` on a compiled FSM.

When defining patterns that are subset of each other, such as `[:or [1 2] [1 2 3]]`, `:find` will always return upon reaching the shortest of the two. If we want to match on the longest, we can use `:greedy-find`. Instead finds their `:greedy-find` with the same parameters, but will be consumed until an accept state is reached, or the end of the stream is reached in either case. `:find` will return a map with the following keys:

When using `:find` for accumulation tools with `:find` or `:greedy-find`, it can be tedious to place an accumulation reduce function in between every transition. In these cases, we can simply use `:interpose #`:

```
[:def f (a/find
          [:0 [:1 :0 :1 :complete]
              [:reduce* [:complete (fn [state input] (conj state[:1]))]]]
a/c = (a/compile f)
[:0 [:1 :0 :1 :complete]
    [:reduce* [:complete (fn [state input] (conj state[:1]))]]]
[:def f (a/find
          [:0 [:1 :0 :1 :complete]
              [:reduce* [:complete (fn [state input] (conj state[:1]))]]]
:greedy-find
a/c = (a/compile f)
[:0 [:1 :0 :1 :complete]
    [:reduce* [:complete (fn [state input] (conj state[:1]))]]]
[:def f (a/find
          [:0 [:1 :0 :1 :complete]
              [:reduce* [:complete (fn [state input] (conj state[:1]))]]]
:interpose #(:reduce* [:complete (fn [state input] (conj state[:1]))])
a/c = (a/compile f)
[:0 [:1 :0 :1 :complete]
    [:reduce* [:complete (fn [state input] (conj state[:1]))]]]
```

interpose

Using `:interpose` to add a `:reduce` to a Clojure expression that then returns to an ClojureScript version of `automat.viz`.

Compiling FSMs can be slow if they have many states. While rarely a concern for JVM applications, quick startup time is often crucial for client-side JavaScript applications. If you find that compiling your ClojureScript FSMs is too slow, consider defining and precompiling them in Clojure.

```
[:ns viz.namespace
  [:require
    [:automat.core :as a]]
[:defmacro get-fsm []
  [:let [:fsm (:eval
                [:clojure.core
                 [:require
                  [:automat.viz :refer [:graphviz]]]
                 :graphviz(a/c)]
                :nil)]
    [:quote :fsm]]]
[:ns viz.namespace
  [:require
    [:automat.core :as a]
    [:graphviz.reader
      [:viz.namespace :refer [:get-fsm]]]]
[:def fsm
  [:clojure.core [:get-fsm] [:graphviz (:eval [:get-fsm])]]]
[:defn fsm-find [input]
  [:(:find fsm input)]]
[:fsm-find [:foo :bar :baz]
  [:# [:accepted? true, :checkpoint nil, :state-index 3, :start-index 0, :stream-index 3, :value [:foo :bar :baz]]]]
```

license

Copyright © 2014 Zachary Tellman

Distributed under the MIT License

Save · Why is this required?

Notes

Not enough notes added

Keyboard shortcuts

Remember collaboration settings

Show in-line collaboration

Show entire line at space