Extracted from:

Seven More Languages in Seven Weeks

Languages That Are Shaping the Future

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Seven More Languages in Seven Weeks

Languages That Are Shaping the Future

Bruce A. Tate, Fred Daoud, Ian Dees, and Jack Moffitt

Foreword by José Valim

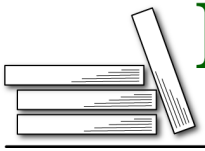
Edited by Jacquelyn Carter

Seven More Languages in Seven Weeks

Languages That Are Shaping the Future

Bruce A. Tate
Fred Daoud
Ian Dees
Jack Moffitt

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-941222-15-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2014

Day 3: Writing Stories with Logic

Over the last two days, you've seen a lot of what core.logic has to offer. Now it's time to put that knowledge to use in a larger and more practical example.

There are many problems that involve route planning. For example, how do you fly to a distant city? Sometimes there are direct flights, but sometimes the path involves multiple connections, different planes, and even several airlines. Alternatively, think of a truck making deliveries. After enumerating possible paths, you must then optimize for the shortest or the quickest.

If you think about it, generating a story is similar but more fun. Instead of connection cities, you have plot points. Routes through the plot make up the entire story, and as an author, you'll want to optimize to achieve the desired effect. Should the story be short? Should everyone die at the end?

We're going to build a story generator using the logic tools you've acquired so far. Although the end result may seem frivolous on the surface, the techniques we use are the same for the more mundane problems.

Before we get to the story generator, there's one last feature of core.logic that merits attention: finite domains.

Programming with Finite Domains

Logic programming is implemented behind the scenes with directed search algorithms. You specify constraints and the language searches for solutions that satisfy the criteria.

So far, we've been working with elements, lists, and maps in our logic programs. These structures may be infinite in size, but they are composed of a finite set of concrete elements. To search for solutions to `(membero q [1 2 3])`, core.logic only needs to look through all the elements.

What happens when we want to work with numbers? Imagine searching for integer solutions to `(<= q 1)`. There is an infinite number of answers. Even worse, there is an infinite number of possibilities to try, and depending on where you start and how you search, you may never find a solution.

The problem is tractable if we constrain `q` to the positive integers or any finite set of numbers. In core.logic we can make such constraints with finite domains. Finite domains add knowledge about the set of valid states in the search problem. Let's use our example `(<= q 1)`, but within a finite domain:

```
user=> (require '[clojure.core.logic.fd :as fd])
nil
```

```
user=> (run* [q]
  #_=>   (fd/in q (fd/interval 0 10))
  #_=>   (fd/<= q 1))
(0 1)
```

❶ This constraint establishes that `q` is in a given interval of numbers.

❷ The solution set is finite and found quickly because of the constrained domain.

Finite domains over numeric intervals also allow you to start doing mathematic operations on logic variables. We can ask core.logic to solve for every triple of distinct numbers whose sum is 100:

```
user=> (run* [q]
  #_=>   (fresh [x y z a]
  #_=>     (== q [x y z])
  #_=>     (fd/in x y z a (fd/interval 1 100))
  #_=>     (fd/distinct [x y z])
  #_=>     (fd/< x y)
  #_=>     (fd/< y z)
  #_=>     (fd/+ x y a)
  #_=>     (fd/+ a z 100)))
([1 2 97] [2 3 95] [1 3 96] [1 4 95] [3 4 93] [2 4 94] ...)
```

❶ `x`, `y`, and `z` are the three numbers we're solving for. `a` is just a temporary helper.

❷ All the logic variables are constrained to the finite range of 1 to 100.

❸ `fd/distinct` sets the constraint that none of the variables can be equal to another. This prevents solutions such as `[1 1 98]`.

❹ We constrain `x` to be less than `y`, and `y` less than `z`. If we failed to order the variables, then we'd have duplicate solutions such as `[6 28 66]` and `[66 28 6]`.

❺ There are 784 solutions. My machine found the entire set of solutions in 5 milliseconds. Core.logic is no slouch.

The code as written works quite well, although having to do arithmetic operations two numbers at a time with a temporary logic variable feels rather clumsy. Fortunately, core.logic provides some macro sugar to sprinkle on its math. `fd/eq` allows us to write normal expressions for our equations and transforms those expressions into code that creates the appropriate temporary logic vars and calls the appropriate finite domain functions.

```
user=> (run* [q]
  #_=>   (fresh [x y z]
  #_=>     (== q [x y z])
```

```
  #_=>     (fd/in x y z (fd/interval 1 100))
  #_=>     (fd/distinct [x y z])
  #_=>     (fd/< x y)
  #_=>     (fd/< y z)
  #_=>     (fd/eq
  #_=>       (= (+ x y z) 100))))
([1 2 97] [2 3 95] ...)
```

The last line is much simpler, and the resulting program is easier to read.

Take a minute to think about what's going on here. Macros are turning logic into normal syntax, finite domains are constraining the search problem to a small space, and the program isn't just finding a single solution but all possible solutions. Best of all, it's declarative and reads like a problem statement instead of a solution method.

Magical Stories

So far, the examples have been tailored to help you understand individual concepts in core.logic, and now we'll put what you've learned into practice with a more realistic and comprehensive example. In the end, even Harry Potter uses his spells for opening locks and other common tasks.

Our task will be one of path finding, subject to some constraints. Whether finding transit routes or scheduling deliveries, path planning problems abound, and logic programming excels at solving them.

Instead of finding a route for a delivery truck or solving for how to reach one city from another via available transit options, we'll be generating stories. From a database of plot elements, we can search for a story that reaches a certain end state. Following the path through the plot elements becomes a little narrative, controlled by logic and the destination you provide.

Getting Inspired

This example was inspired by a wonderful talk I saw at Strange Loop 2013. It's called "Linear Logic Programming" by Chris Martens.[a] She also co-wrote a paper on the same topic: "Linear Logic Programming for Narrative Generation."[b] Chris explains linear logic programming and then uses it to generate and explore narrative using Madame Bovary as a reference. I highly recommend investigating her work.

a. http://www.infoq.com/presentations/linear-logic-programming
b. https://www.cs.cmu.edu/~cmartens/lpnmr13.pdf

Core.logic will generate many possible stories for us, but it is up to us to pick out the ones that might be interesting. We'll use Clojure to postprocess pos-

sible stories to select ones that fit our criteria. For example, we may filter small stories out to get at more interesting, longer narratives.

Problem Details

Before we begin, we should define the problem a little better.

First, we need a collection of story elements and a method for moving from one element to another. We can easily store facts in a database about various plot points, but we'll need to create them. I'll let Hollywood do the hard work here and use the plot of the cult comedy movie *Clue*, a murder mystery loosely based on the board game where six guests, a butler, a maid, a cook, and the master of the house are in for a potentially deadly ride.

Here's a snippet of the story from the movie:

(1) Wadsworth opens the door to find a stranded motorist whose car broke down nearby. The motorist asks to use the phone, and Wadsworth escorts him to the lounge. The group locks the motorist in the lounge while they search for the killer in the rest of the house. (2) After some time, a policeman notices the abandoned car and starts to investigate. (3) Meanwhile, someone kills the motorist with the wrench.

We can simulate and manage moving from one plot point, (1), to another, (2), by the use of linear logic. Linear logic is an extension of the logic you're probably familiar with, which allows for the use and manipulation of resources. For example, a logical proposition may require and consume a particular resource. Instead of "A implies B," we say that "A consumes Z and produces B," where Z is some particular resource. In the previous snippet, (3) takes the motorist as input and produces a dead motorist. Similarly, (2) takes the motorist as input and produces a policeman.

We can craft a simple linear logic in core.logic. Each plot element will have some resource that it needs and some resource that it produces. We'll represent this as a two-element vector of the needed and produced resource. For example, `[:motorist :policeman]` means that for this story element to happen, we must have a `:motorist` available and it will create a `:policeman`. In the movie, a stranded motorist rings the doorbell for help, and later a policeman who discovers his car comes looking for him. Without the motorist, the policeman will never show up.

We'll have a starting state, which is a set of initial, available resources. A relation will govern selecting a legal story element given the state and moving to a new state. We'll control where the story goes by putting requirements on

the final state. For example, the story will finish when a particular character is caught or dies.

The final touch will be to print out the resulting stories in a readable form.

Story Elements

Our story elements need to contain the resource being consumed and the resource being produced. Additionally, we'll put in a string describing the element in prose that we'll make use of when printing out the narrative.

We'll need a large list of elements to make interesting stories, but the plot of *Clue* gives us plenty to work with. You might notice that several people can be murdered in multiple ways; *Clue* had three different endings, which are all represented.

Here are the first few elements of story-elements in story.clj:

minikanren/logical/src/logical/story.clj

```
(def story-elements
  [[:maybe-telegram-girl :telegram-girl
    "A singing telegram girl arrives."]
   [:maybe-motorist :motorist
    "A stranded motorist comes asking for help."]
   [:motorist :policeman
    "Investigating an abandoned car, a policeman appears."]
   [:motorist :dead-motorist
    "The motorist is found dead in the lounge, killed by a wrench."]
   [:telegram-girl :dead-telegram-girl
    "The telegram girl is murdered in the hall with a revolver."]
   [:policeman :dead-policeman
    "The policeman is killed in the library with a lead pipe."]
   [:dead-motorist :guilty-mustard
    "Colonel Mustard killed the motorist, his old driver during the war."]
   [:dead-motorist :guilty-scarlet
    "Miss Scarlet killed the motorist to keep her secrets safe."]
   ;; ...])
```

The structure is a vector of vectors. The inner vectors have three elements: the two resources and the string description. There are 27 elements in full story-elements, which is enough to get some interesting results.

We'll need to postprocess this to turn it into our story database in core.logic.

Building the Database and Initial State

Our goal is to turn the story-elements vector into a database of facts that core.logic can use. We'll use a simple relation, ploto, which will relate the input resource to the output. The end result we want would be equivalent to this:

```
(db-rel ploto a b)

(def story-db
  (db
   [ploto :maybe-telegram-girl :telegram-girl]
   [ploto :wadsworth :dead-wadsworth]
   ;; ...))
```

We'll use Clojure's reduce function to effect the transformation:

minikanren/logical/src/logical/story.clj
```
(db-rel ploto a b)

(def story-db
  (reduce (fn [dbase elems]
            (apply db-fact dbase ploto (take 2 elems)))
          (db)
          story-elements))
```

❶ Reducing functions take two arguments. The first is the accumulator that holds the initial, intermediate, or final result of the reduction. The second is the current element to reduce. Here we extend the database passed as the first argument with a new fact using the ploto relation and the first two elements of the story element vector.

❷ Our initial state is just a blank database.

❸ Running the reduction over all the story elements will turn our vector of vectors into a core.logic database of facts.

Now that we have our story elements, we need an initial state. This contains all the people who may appear and all the people who are already in the house who might later be killed. Note that only resources that are used in the story elements need to be listed.

minikanren/logical/src/logical/story.clj
```
(def start-state
  [:maybe-telegram-girl :maybe-motorist
   :wadsworth :mr-boddy :cook :yvette])
```

The story elements database and the initial state define all the data for our story. As you'll see shortly, the data used is much larger than the code we need to generate stories.

Plotting Along

The next task is to create a transition relation to move the story from one state to the next by selecting an appropriate story element. This is the workhorse of our generator:

minikanren/logical/src/logical/story.clj
```
(defn actiono [state new-state action]
  (fresh [in out temp]
    (membero in state)
    (ploto in out)
    (rembero in state temp)
    (conso out temp new-state)
    (== action [in out])))
```

❶ The in resource must be something from the current state. We can't use any of the story resources unless they are available.

❷ Once we have an in resource, ploto picks a corresponding resource to create in out.

❸ The resource is consumed as part of the story action, so we remove it from the state.

❹ The newly created resource is added to the state to produce the new state.

We can load logical.story in a REPL and experiment with actiono:

```
user=> (require '[logical.story :as story])
user=> (with-db story/story-db
  #_=>   (run* [q]
  #_=>     (fresh [action state]
  #_=>       (== q [action state])
  #_=>       (story/actiono [:motorist] state action))))
([[:motorist :policeman] (:policeman)]
 [[:motorist :dead-motorist] (:dead-motorist)])
```

This query uses a starting state of [:motorist] and asks for all the actions and their corresponding new states that are possible. Either a policeman can come looking for the stranded motorist, or the motorist can be murdered.

For generating our stories, we want to run the transitions backward. Starting from some goal conditions—resources we want to exist in the final state—we want to find a sequence of actions that will achieve the goals from the starting state.

minikanren/logical/src/logical/story.clj

```
(declare story*)

(defn storyo [end-elems actions]
  (storyo* (shuffle start-state) end-elems actions))

(defn storyo* [start-state end-elems actions]
  (fresh [action new-state new-actions]
    (actiono start-state new-state action)
    (conso action new-actions actions)
    (conda
     [(everyg #(membero % new-state) end-elems)
      (== new-actions [])]
     [(storyo* new-state end-elems new-actions)])))
```

❶ storyo simply calls storyo* so that the user doesn't have to pass in the initial state themselves. Shuffling the initial state will produce a randomized solution sequence.

❷ We transition to a new state by taking some action.

❸ We prepend the action we took onto the list of actions.

❹ everyg succeeds if the goal function provided as the first argument succeeds for every element of the collection in the second argument. If all our goal resources in end-elems are in new-state, our story is done. We set new-actions to the empty vector at the end since there won't be any more taken.

❺ If not all our goals have succeeded, we recursively call storyo* to keep going.

Let's play with storyo at the REPL to generate some simple stories:

```
user=> (with-db story/story-db
  #_=>   (run 5 [q]
  #_=>     (story/storyo [:dead-wadsworth] q)))
(([:wadsworth :dead-wadsworth])
 ([:maybe-motorist :motorist] [:wadsworth :dead-wadsworth])
 ([:maybe-telegram-girl :telegram-girl] [:wadsworth :dead-wadsworth])
 ([:maybe-motorist :motorist] [:motorist :policeman]
  [:wadsworth :dead-wadsworth])
 ([:maybe-motorist :motorist] [:motorist :dead-motorist]
  [:dead-motorist :guilty-mustard] [:wadsworth :dead-wadsworth]))
```

Core.logic has used our story database to generate five stories where Wadsworth ends up dead. Each solution is a list of actions, and if you squint at it and remember the story elements, you can figure out what is happening. For example, the last story in the list has a stranded motorist appearing, who is then murdered by Colonel Mustard, and then Wadsworth is killed in the hallway with the revolver.

We still have some work to do. The stories need to be human readable instead of the simple list of actions, and we'll need to filter stories to get more interesting results.