Uri's SM in Lisp: Version 0.0.4

Brian Beckman, Uri Ran

<2015-11-23 Mon>

Contents

1 Preliminaries

```
Emacs version: GNU Emacs 25.3.1 (x86_64-apple-darwin13.4.0, NS appkit-1265.21 Version 1
```

```
 of 2017-09-12
org version: 9.0.9
```

This is version 0.0.4 of Uri Ran's State-Machine demo.

1.1 How to Use this Document

This is a literate program. Code and documentation cohabit a single "org-mode" file called `sm3.org`. Typeset the document by running the emacs command `org-latex-export-to-pdf`. If you're a VIM user, use *Spacemacs*[1], a near-perfect emulation of VIM in emacs. See the section "Running Code" for instructions on running the code in this document.

1.2 Setup

2 Vertex Struct

A *vertex* or "state" in the state-machine diagram has an *entry action*, a *do-action*, an *exit-action*, and an *event table*. We prefer the word "vertex" to reduce ambiguity and overloading of the word "state."

Consider a unique variable or *token* called `*current-vertex*`. The value of this token represents the vertex that the state machine *inhabits*. The state machine may inhabit exactly one vertex at any time. Visualize the dynamics of the machine as a sequence of indivisible, atomic steps in which the token either stays within the current vertex or moves from one vertex to another in response to *events* or *inputs*. Events or inputs are denoted by *symbols* drawn from a finite *alphabet*, one at a time.

Entry actions run whenever the `*current-vertex*` token enters a vertex. Exit actions run when the `*current-vertex*` token leaves a vertex.

Do-actions concern *polling*, not further discussed here. Although we define do-actions here, we don't use them; they're a placeholder in this version.

An *event table* is an associative lookup table from event symbols to a list of triples. Each triple has a Boolean-valued *guard function*, a *transition-action* function, and a new vertex. The engine will `funcall` the guards in the list, in the order they're presentated, and accept the first triple whose guard returns `t`, which means *true* in Lisp. "Accepting the triple" means running the *transition action* and moving the token to the new vertex if the new vertex is non-nil (TODO: what if the new vertex is *nil*?). The exit action of the old vertex runs and then the entry action of the new vertex runs. If no guards return `t`, the engine logs a trace message and does nothing.

The following lisp code defines the vertex struct.

```
(defstruct vertex-t name entry-ac do-ac exit-ac evt-tbl)
```

`defstruct` gives us basic constructors and accessors, and that's all we need for this demonstration. Should we need more in the future, we can consider Lisp's `defclass`.

As a side-effect of calling `defstruct`, Lisp defines the following functions in our environment

- `(make-vertex-t`

[1] `http://spacemacs.org`

- `:name` <*name your vertex*>
- `:entry-ac` <*put a function value here*>
- `:do-ac` <*put another function value*>
- `:exit-ac` <*put another function value*>
- `:evt-tbl` <*put an event table, here*> `)`
- produces an instance of `vertex-t`; this is a *constructor* function

- `(vertex-t-name` <*some instance of vertex-t*> `)` produces the name of the vertex; this is like the dot notation in C / C++, *i.e.*, like `someInstance.name`
- `(vertex-t-entry-ac` <*some instance of vertex-t*> `)` produces the entry-action function value of the vertex; also like dot, just for a different instance variable
- `(vertex-t-do-ac` <*some instance of vertex-t*> `)` produces the do-action function value of the vertex; ditto
- `(vertex-t-exit-ac` <*some instance of vertex-t*> `)` produces the exit-action function value of the vertex; etc.

  It's possible to mutate the *instance variables* of a `defstruct`, but we don't need to do this here.

3 Utilities

3.1 Boolean fair coin

For fuzz testing.

```
(defun coin () (= 0 (random 2)))
```

3.2 Take & Drop

```
(defun take (seq n)
  "(take seq n) gives the first n elements of the seq. (take seq -n) gives the
  last n elements of the seq. This works on strings as well."
  (check-type seq sequence)
  (check-type n integer)
  (let ((l (length seq)))
    (cond ((>= n 0) (subseq seq 0 (min n l)))
          ((<  n 0) (subseq seq (max 0 (+ n l)) l))
          ((=  n 0) (subseq seq 0 0)))))

(defun drop (seq n)
  "(drop seq n) gives the seq with the first n elements removed. (drop seq -n)
  gives the seq with the last n elements removed. This works on strings as
  well."
```

```
  (let ((l (length seq)))
    (check-type seq sequence)
    (check-type n integer)
    (cond ((>= n 0) (subseq seq (min n l) l))
          ((<  n 0) (subseq seq 0 (max 0 (+ n l))))
          ((=  n 0)  seq))))

(defun str-last (str)
  "(str-last non-empty-string) produces the last character in a non-empty
  string."
  (check-type str string)
  (let ((l (length str)))
    (assert (> l 0))
    (subseq str (- l 1) l)))
```

3.3 Drawing to DOT

Borrowed from "Land of Lisp" by Conrad Barski, M.D.

3.3.1 TODO Robustify

a-la `http://tinyurl.com/y63ugo` and `http://tinyurl.com/j23lakq`

```
(defparameter *max-label-length* 30)

(defun dot-name (exp)
  (substitute-if #\_ (complement #'alphanumericp) (prin1-to-string exp)))

(defun dot-label (exp)
  (if exp
      (let ((s (write-to-string exp :pretty nil)))
        (if (> (length s) *max-label-length*)
            (concatenate 'string (subseq s 0 (- *max-label-length* 3)) "...")
            s))
      ""))

(defun nodes->dot (nodes)
  (mapc (lambda (node)
          (fresh-line)
          (princ (dot-name (car node)))
          (princ "[label=\"")
          (princ (dot-label node))
          (princ "\"];"))
        nodes))

(defun edges->dot (edges)
  (mapc (lambda (node)
```

```
         (mapc (lambda (edge)
                 (fresh-line)
                 (princ (dot-name (car node)))
                 (princ "->")
                 (princ (dot-name (car edge)))
                 (princ "[label=\"")
                 (princ (dot-label (cdr edge)))
                 (princ "\"];"))
               (cdr node)))
       edges))

(defun dgraph->dot (nodes edges)
  (princ "digraph{")
  (nodes->dot nodes)
  (edges->dot edges)
  (princ "}"))

(defun uedges->dot (edges)
  (maplist (lambda (lst)
             (mapc (lambda (edge)
                     (unless (assoc (car edge) (cdr lst))
                       (fresh-line)
                       (princ (dot-name (caar lst)))
                       (princ "--")
                       (princ (dot-name (car edge)))
                       (princ "[label=\"")
                       (princ (dot-label (cdr edge)))
                       (princ "\"];")))
                   (cdar lst)))
           edges))

(defun ugraph->dot (nodes edges)
  (princ "graph{")
  (nodes->dot nodes)
  (uedges->dot edges)
  (princ "}"))

(defun dot->png (fname thunk)
  (with-open-file (*standard-output* (concatenate 'string fname ".dot") :direction :out
    (funcall thunk))
  ;; (ext:shell (concatenate 'string "dot -Tpng -O " fname ".dot"))
  )

(defun dgraph->png (fname nodes edges)
  (dot->png fname
            (lambda ()
```

```
              (dgraph->dot nodes edges))))

(defun ugraph->png (fname nodes edges)
  (dot->png fname
            (lambda ()
              (ugraph->dot nodes edges))))
```

4 Action and Guard Functions

4.1 Actions

4.1.1 TODO Parameters or return values for actions?

4.1.2 TODO Contexts for actions and guards

4.1.3 Vertex Actions

For entry, polling (undefined) and exit, respectively.

Our actions just print to standard output because this is just a demo. They might do arbitrary side effects.

```
(defun vertex-1-entry () (print "vertex 1 entry"))
(defun vertex-2-entry () (print "vertex 2 entry"))
(defun vertex-3-entry () (print "vertex 3 entry"))
(defun vertex-4-entry () (print "vertex 4 entry"))

(defun vertex-1-do    () (print "vertex 1 do"))
(defun vertex-2-do    () (print "vertex 2 do"))
(defun vertex-3-do    () (print "vertex 3 do"))
(defun vertex-4-do    () (print "vertex 4 do"))

(defun vertex-1-exit  () (print "vertex 1 exit"))
(defun vertex-2-exit  () (print "vertex 2 exit"))
(defun vertex-3-exit  () (print "vertex 3 exit"))
(defun vertex-4-exit  () (print "vertex 4 exit"))
```

4.1.4 Edge Actions

When the engine takes a transition, moving the token from one vertex to another, it runs these functions.

```
(defun act-a () (print "action a" ))
(defun act-b () (print "action b" ))
(defun act-c () (print "action c" ))
(defun act-d () (print "action d" ))
(defun act-na() (print "action na"))
```

4.2 Guards (Boolean-Valued)

```
(defun guard-x     () (coin) )
(defun guard-y     () (coin) )
(defun guard-z     () (coin) )
(defun guard-true  () t      )
(defun guard-false () nil    )
(defun guard-na    () t      )
```

5 The Diagram

If `nym` is `"foo"`, we want functions `foo-entry`, `foo-do`, and `foo-exit` automatically assigned. The following macro expands into the boilerplate necessary for creating instances of `vertex-t`. These instances are stored in global *special variables* demarcated with asterisks, for example, `*vertex-1*`. Special variables are basically global variables, but there are some subtleties that don't concern us here.[2]

The macro works by defining some strings for the identifiers based off the `nym`, converting them to symbols, and writing out new code that defines, via `defvar`, a global variable that refers to an instance of `vertex-t` with *entry*, *do*, and *exit* actions defined according to the naming convention in the paragraph above.

5.1 How to Define a Vertex

```
(defparameter *vertices* nil)
(defmacro defvertex (nym evt-tbl)
  (let* ((dynvar (format nil "*~A*"     nym))
         (entry  (format nil "~A-entry" nym))
         (doo    (format nil "~A-do"    nym))
         (exit   (format nil "~A-exit"  nym))
         (vtxsym (with-input-from-string (s dynvar) (read s))))
    `(progn
       (defparameter ,vtxsym
         (make-vertex-t
          :name     (format nil "~A" ,nym)
          :entry-ac (function ,(with-input-from-string (s entry) (read s)))
          :do-ac    (function ,(with-input-from-string (s doo  ) (read s)))
          :exit-ac  (function ,(with-input-from-string (s exit ) (read s)))
          :evt-tbl  ,evt-tbl))
       (push ,vtxsym *vertices*))))
```

5.2 Vertices in Our Diagram

Notice that the new vertices named in the event table are unevaluated symbols. That's because we want to refer to them before they're defined. We know their names at the time we write the

[2] `http://www.flownet.com/ron/specials.pdf`

table, but they don't always have values. This is a good way to avoid forward referencing and resolution. Evaling the symbols at transition time is preferable.

```
(defvertex "vertex-1"
    '((ev-2 (guard-true act-c  *vertex-3* ))
      (ev-3 (guard-x    act-na *vertex-1* ))  ))
(defvertex "vertex-2"
    '((ev-4 (guard-true act-na *vertex-1* ))
      (ev-6 (guard-x    act-c  *vertex-4* ))  ))
(defvertex "vertex-3"
    '((ev-1 (guard-x    act-na nil        )
            (guard-y    act-b  *vertex-1* )
            (guard-z    act-na *vertex-1* ))
      (ev-5 (guard-na   act-d  *vertex-4* ))  ))
(defvertex "vertex-4"
    '((ev-3 (guard-y    act-d  *vertex-2* ))
      (ev-6 (guard-x    act-c  *vertex-3* ))  ))
```

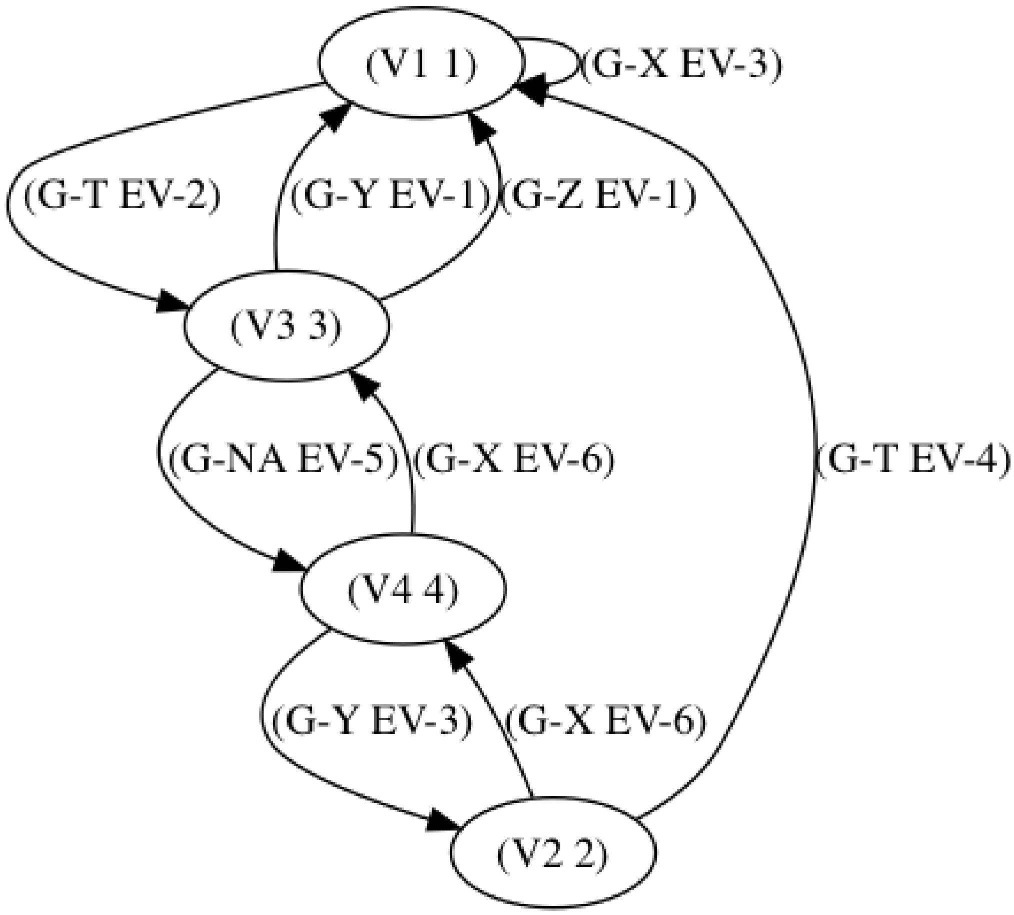
5.3 Drawing The Diagram

```
(defparameter *wizard-nodes*
  '((v1 1)
    (v2 2)
    (v3 3)
    (v4 4)))

(defparameter *wizard-edges*
  '((v1 (v3 g-t  ev-2)
        (v1 g-x  ev-3))
    (v2 (v1 g-t  ev-4)
        (v4 g-x  ev-6))
    (v3 (v1 g-y  ev-1)
        (v1 g-z  ev-1)
        (v4 g-na ev-5))
    (v4 (v2 g-y  ev-3)
        (v3 g-x  ev-6))))

(dgraph->png "wizard" *wizard-nodes* *wizard-edges*)
```

```
dot -Tpng -O wizard.dot
```



5.4 Simulating Transitions in the Diagram

5.4.1 The Vertex Token

At any time, the state-machine is "in" a vertex (or state). This means that the value of `*current-vertex*` is the particular vertex instance. We call `*current-vertex*` the *vertex token*. Visualize a token on a gaming board moving from one vertex to another.

```
(defparameter *current-vertex* *vertex-1*)
```

5.4.2 The Engine

1. Eval-First-Admissible-Triple

   This function implements the token-moving strategy discussed above and returns the current value of the token `*current-vertex*`, whether it's changed or not.

   When the new-vertex is `nil`, the `*current-vertex*` does not change and the action functions do not run, even if the guard is true.

```
(defun eval-first-admissible-triple (triples)
  (cond (triples
         (let* ((triple     (first  triples))
                (guard      (first  triple))
                (action     (second triple))
                (new-vertex (eval (third  triple))))
           (if (and (funcall guard) new-vertex)
               (progn
                 (funcall (vertex-t-exit-ac *current-vertex*))
                 (funcall action)
                 (setf *current-vertex* new-vertex)
                 (funcall (vertex-t-entry-ac *current-vertex*)))
               (progn
                 (format t "~%~A: guard failed; trying next guard"
                         (vertex-t-name *current-vertex*))
                 (eval-first-admissible-triple (rest triples))))))
        (t (format t "~%~A: all guards failed; doing nothing"
                   (vertex-t-name *current-vertex*))))
  *current-vertex*)
```

2. SM-Engine

   This takes an event symbol, does lookup in the diagram, and performs the indicated transition.

```
(defun sm-engine (event-symbol)
  (let ((line (rest (assoc event-symbol
                           (vertex-t-evt-tbl *current-vertex*)))))
    (if line
        (eval-first-admissible-triple line)
        (progn
          (format t "~%~A: event ~W not found; doing nothing"
                  (vertex-t-name *current-vertex*)
                  event-symbol)
          *current-vertex*))))
```

6 Running the Code

This document contains actual, live code. You can run the code in two ways: inside org mode or by extracting (tangling) the code and running it at the command line.

6.1 Setting up Two Good Lisps

Install SBCL (Steel Bank Common Lisp) for running this code in the editor or a REPL, and ECL (Embeddable Common Lisp) for generating C code. On a mac, this is trivial with homebrew:

```
brew install sbcl
brew install ecl
```

You will need SLIME in Emacs or Spacemacs to run the code in this file directly. Just Google any of these things you don't recognize. I recommend Spacemacs because it has high-fidelity VIM emulation.

To find out whether you have slime, type `M-x slime`. If you don't have it, get it.

6.2 Running Code Directly

Once you have SLIME running in Emacs, type `M-x slime` to start the REPL, then type `M-x org-babel-execute-buffer` to run all the code in this file. At the very end of this file, you will see a few unit tests. Put the cursor in that code block and type `C-c C-c` repeatedly to run the unit tests over and over. The results will be slightly different each time because the guard functions do coin flips. I have tried to arrange the unit tests so that the last value always prints `t`, short for `true`.

6.3 Extracting Code From This File

Type `M-x org-babel-tangle` and you should get a file named `sm.lisp`.

6.4 Generating, Inspecting, Running C code

After extracting code, run ECL at the command prompt:

```
$ ecl -load make.lisp
```

Watch all the pretty messages go by, then type

```
(quit)
```

to leave the ECL REPL, then

```
$ ./sm
```

to run the generated code. You should see exactly the same output as you would get from the last section below.

6.4.1 **TODO** Create Deeply Embedded C

The generated code is in the files `sm.c`, `sm.h`, and `sm.data`. The generated code pretty much just calls the ECL runtime kernel. This is a *shallow embedding* of the lisp code in C. A *deep embedding* would write C code that bypasses lisp-specific helpers and more directly express the model. Bypassing a lisp runtime means that we can avoid exposure to garbage collection and other potential hazards in the lisp implementation.

A good way to produce a deep embedding will be through macros. The deeply embedded code should be comparable to the code that Uri wrote by hand.

6.5 Interactively

To run in an external REPL, paste the following code into the REPL (and remove the quote, of course). Don't try to run this code from org-mode; it will deadlock with SLIME as they contend over who has the terminal.

```
'(let ((ev 1024))
   (loop while (> ev 0) do
     (format t "~%Enter an event number > 0, 0 to quit: ")
     (setf ev (read))
     (format t "~%~A: searching for event ~A"
             (vertex-t-name *current-vertex*)
             (format nil "ev-~A" ev))
     (if (numberp ev)
         (progn
           (with-input-from-string (s (format nil "ev-~A" ev))
                (sm-engine (read s nil 0))))
         (format t "~%~A: failure: type-of ev wasn't a number, but a ~A"
                 (vertex-t-name *current-vertex*)
                 (type-of ev)))))
```

6.6 Unit Tests, Exhaustive Tests

Because the current guards are random, exhaustively testing them isn't as trivial as enumeration.

Run the following unit test repeatedly; it can be a little different each time, but the machine should always end up in vertex 3.

```
(print (equal *current-vertex* *vertex-1*))
(print (eq *current-vertex* *vertex-1*))
(print (eq (sm-engine 'ev-1) *vertex-1*))
(print (eq (sm-engine 'ev-4) *vertex-1*))
(print (eq (sm-engine 'bogus) *vertex-1*))
(print (eq (sm-engine 'ev-3) *vertex-1*))
(print (eq (sm-engine 'ev-2) *vertex-3*))
```

```
T
T
vertex-1: event EV-1 not found; doing nothing
T
vertex-1: event EV-4 not found; doing nothing
T
vertex-1: event BOGUS not found; doing nothing
T
"vertex 1 exit"
"action na"
```

```
"vertex 1 entry"
T
"vertex 1 exit"
"action c"
"vertex 3 entry"
T
```

Emacs 24.3.1 (Org mode 8.0.7)