CA-DSL Initial Planning Document

Purpose and Concepts

Our programming language is designed to ease the ability to describe and visualize cellular automata. There are currently a variety of different format strings for certain specific classes of cellular automata and some visualizers that can visualize them, but we want a general approach that can enable for the construction of new classes of cellular automata in a simple, but systematic manner.

A *cellular automata* is a collection of *cells* which each have a *state*. At each time step, all the cells transition between states according to *rules*. Rules define the start and end states of the transition and the condition when the transition should occur. The condition is dependent on the state of cells in the *neighborhood* of a cell. In *totalistic cellular automata*, it is only the number of cells in each state which the condition depends on. In *non-totalistic cellular automata*, the positioning of the cell in each state is also factored into the condition. We want to make expressing totalistic cellular automata simple, while still making it possible to express non-totalistic cellular automata.

The concept of a neighborhood requires the introduction of a *topology*. A topology represents how cells are connected. In the canonical cellular automata, Conway's Game of Life, the topology is a flat 2D infinite plane. We can't model an infinite plane with a finite computer, so we intend to have the ability to specify certain cells which are frozen in one state, thus creating a wall around a finite portion of the plane. Our DSL will also enable modeling of topologies composed of hexagonal cells as well as looped topologies. Our dream is that we can model a cellular automata on a soccer ball.

Example Programs

Conway's Game of Life

```
(define conways
    (rule
        [(0 -> 1) (3 in 1)]
        [(1 -> 1) ((2 3) in 1)]))


(define world
    (make-world 100 100 (random 50 50)))

(run conways world)
```

Wireworld

```
(define wireworld ;; https://conwaylife.com/wiki/OCA:WireWorld
    (rule #:default identity
        [('head -> 'tail)]
        [('tail -> 'conductor)]
        [('conductor -> 'head) ((1 2) in 'head)]))
```

Star Wars

```
(define star-wars ;; https://conwaylife.com/wiki/OCA:Star_Wars
    (rule #:default add1
        [(0 -> 0) (except 2 in 1)]
        [(1 -> 1) ((3 4 5) in 1)]
        [(3 -> 0)]))
```

Rule 110 (Example of 1D Rule)

```
(define rule-110 ;; https://en.wikipedia.org/wiki/Rule_110
   (rule #:neighborhood '(-1, 1)
      [(0 -> 1) (1 is 1)]
      [(1 -> 1) (except 2 in 1)]))
```

Predators and Prey

```
(define predators-and-prey
   (rule #:default 'dead
      [('dead -> 'prey) (and (none in 'predator) (any in 'prey))]
      [('prey -> 'prey) (and (any in 'dead) (none in 'predator))]
      [('prey -> 'predator) (all in 'prey)]
      [('predator -> 'predator) (any in 'prey)]))
```

Defining Topologies

Below is an example of defining a 2D topology, which wraps around in the x dimension. We'd like to add syntactic sugar and built in functions to make defintions like this much simpler. We will also be providing these functions as a part of the library.

```
(define-type Pos (Pair Integer Integer))
(define-type Offset (Union Pos Symbol))
(define-type Maybe-Pos (Union (Pair Integer Integer) Void))

(: offset-to-pos : (Offset -> Pos))
(define (offset-to-pos offset)
  (match offset
    [(cons x y) (cons x y)]
    ['above (cons 0 1)]
    ['below (cons 0 -1)]
    ['right (cons 1 0)]
    ['left (cons -1 0)]))
```

```
(: cartesian-topology : (Pos Offset -> Pos))
(define (cartesian-topology cell offset)
  (let ([off-pos (offset-to-pos offset)])
    (cons (+ (car cell) (car off-pos))
          (+ (cdr cell) (cdr off-pos)))))

(: add-x-edges : (Pos Offset -> Pos) Integer Integer -> (Pos Offset ->
Maybe-Pos))
(define (add-x-edges topology x-min x-max)
  (lambda ([cell : Pos] [offset : Offset])
    (let ([out-cell (topology cell offset)])
        (when (and (<= (car out-cell) x-max) (>= (car out-cell) x-min))
              out-cell))))
```

Grammars and Signatures

```
<RULE> ::= (rule <NEIGHBORHOOD> <DEFAULT> <BRANCH> ... )

<NEIGHBORHOOD> ::=
               | #:neighborhood <INLINE-NEIGHBORHOOD>

<DEFAULT> ::=
           | #:default <expr>

<BRANCH> ::= [<TRANSITION> <COND>]
        | [<TRANSITION>]
        | [<STATE> <expr>]



<TRANSITION> ::= (<STATE> -> <STATE>)
             | (* -> <STATE>)

<COND> ::= <expr>
        | <exc?> <COUNTS> in <STATE>
        | <EXC?> <COUNTS> from <INLINE-NEIGHBORHOOD> are <STATE>
        | <EXC?> <NEIGHBOR> is <STATE>

<COUNTS> ::= (<natural> <natural> ...)
         | <natural>
         | none
         | any
         | all

<INLINE-NEIGHBORHOOD> ::= <list of NEIGHBOR>

<NEIGHBOR> ::= <OFFSET>
            | absolute <CELL>

<EXC?> ::=
```

```
        | except

<STATE> ::= <expr>

<CELL> ::= <expr>



<OFFSET> ::= <expr>


<INLINE-NEIGHBORHOOD> ::= <list>
```

```
(define-type (StateMap C S) (HashTable C S))
(define-type (Topology C O) (C O -> C))
(define-type (ActiveFilter C) (C -> Bool))
(define-type (ColorMap S) (S -> Color))
(define-type (Rule C S) ((World C Any S) -> (StateMap C S)))
(deine-type (Renderer C O S) ((World C O S -> Image))
(define-type (World C O S) (All (C O S) (Struct (StateMap C S) (Topology C
O) (ActiveFilter C))))

;; which is a struct representing the current state of the cells, the
topology of the how the cells are connected, and a predicate to determine
whether a cell is fixed in a state or can change. Type parameters C, O, and
S represent the type of the cells, the offsets, and the states
respectively.

;; Renderer for a 2d cells represented by cartesian coordinates.
(All (S) (ColorMap S) -> ((World Posn Any S) -> Image))
(define (render-2d color-map) ... )

;; (All (C O S) (World C O S) (Rule C S) (Renderer C O S) -> (World C O S))
;; (define (run world rule renderer) ...)


## Milestones

- Create a 2d renderer which can map a world where cells are represented as
posns to an image.
- Create the `make-2d-topology` function which can create a topology by
assembling components like a cartesian grid, boundaries outside of which
are invalid coordinates, and linking edges.
- Create a function which can create a rectangular grid of cells of a fixed
state represented as a StateMap.
- Create a barebones `run` function which can render a cellular automata
with no rules, just all cells staying in the same state by passing in a
simple 3x3 rectangular topology with walls on the edges and using the
identity function for the rule, and then creating our 2d renderer which can
```

```
take in a mapping of posn -> state and render using a default color scheme.
- Create a random function to create a StateMap with random states.
- Create the `rule` macro without support for branches nor support for the
#:neighborhood named argument. We will just show that it can change from
one state to another on the first generation and then stay there.
- Make a function that creates a hollow rectangle of frozen cells.
- Implement support for specifying a neighborhood.
- Implement the branch portion of the `rule` macro without support for
conds
- Implement support for < cond>
- Add support for hex grids
- Add support for linking edges of the topology
- Add library functions to translate from existing cellular automata
specification strings to our DSL
- Make the second argument to our render2d optional, and just use a default
color scheme.
- Add support for the renderer indicating connected edges
- Add a 3d renderer for Posns in 3d
```