```
1
2  #lang racket
3
4
5  (require racket/format "components.rkt")
6  (provide simple-inc)
7
8
9  (define (turing name state-init state-term blank rules symbols)
10   (define state  state-init)
11   (define head   500)
12   (define bottom 500)
13   (define top    500)
14   (define step   0)
15   (define tape   (make-vector 1000))
16   (define led-tape (make-vector 11))
17   (define running #f)
18
19   (displayln "machine: ")
20   (displayln name)
21
22   (define (initialize)
23     (let create-leds
24       ((idx 0)
25        (pin-idx 100)
26        (end (vector-length led-led-tape)))
27       (when (< idx end)
28         (vector-set! led-tape idx (mk-led pin-idx))
29         (create-leds (+ idx 1) (+ pin-idx 1) end)))
30     (reset '(1 1 1)))
31
32   (define (update-leds!) ; led -vector updaten
33     (let update
34       ((led-idx 0)
35        (end (vector-length led-tape))
36        (tape-idx (- head 5)))
37       (when (< led-idx end)
38           (let ((tape-value (vector-ref tape tape-idx))
39                 (led (vector-ref led-tape led-idx)))
40             ((led 'update!) tape-value)
41             (update (+ led-idx 1) end (+ tape-idx 1))))))
42
43
44
45
46   (define (reset symbols) ; reset tape
47     (set! state state-init)
48     (set! head 500)
49     (set! bottom 500)
50     (set! top 500)
51     (set! step 0)
52     (set! tape (make-vector 1000))
53     (do ()((null? symbols))
54       (vector-set! tape top (car symbols))
55       (set! symbols (cdr symbols))
56       (set! top (+ top 1)))
57     (vector-set! tape top blank)
58     (update-leds!))
59
60
61
62   (define (show)
63     (display " step: ")  (display (~a step  #:align 'right #:width 2  #:pad-string
63 " "))
64     (display " state: ") (display (~a state #:align 'left  #:width 5  #:pad-string
64 " "))
65     (display " tape: ")
```

```
    (do ((pos bottom (+ pos 1))) ((> pos top))
      (display (if (= pos head) "[" " "))
      (display (vector-ref tape pos))
      (display (if (= pos head) "]" " ")))
    (newline))

  (define (left)
    (set! head (- head 1))
    (when (< head bottom)
      (set! bottom head))
    (update-leds!))

  (define (right)
    (set! head (+ head 1))
    (when (> head top)
      (set! top head))
    (update-leds!))

  (define (write/erase!)
    (let ((old (vector-ref tape head)))
      (if (or (eq? old blank) (eq? old 0))
          (vector-set! tape head 1)
          (vector-set! tape head 0))
      (update-leds!)))

  (define (try rules)
    (cond
      ((not (null? rules))
       (let* ((rule          (car rules))
              (state-before  (vector-ref rule 0))
              (symbol-before (vector-ref rule 1)))
         (if (and (eq? state-before state)
                  (eq? symbol-before (vector-ref tape head)))
             (let ((symbol-after (vector-ref rule 2))
                   (action       (vector-ref rule 3))
                   (state-after  (vector-ref rule 4)))
               (vector-set! tape head symbol-after)
               (case action
                 ((left)
                  (set! head (- head 1))
                  (cond
                    ((< head bottom)
                     (set! bottom head))))
                 ((right)
                  (set! head (+ head 1))
                  (cond
                    ((> head top)
                     (set! top head)))))
               (set! state state-after))
             (try (cdr rules)))))))


  (define (next-step)
    (when (not (eq? state state-term))
      (set! step (+ step 1))
      (try rules)
      (update-leds!)))

  (define (run)
    (if (eq? state state-term)
        (begin (show) (update-leds!))
        (if running
            (begin
              (next-step)
              (show)
              (sleep 1.5)(run))
            (run))))
```

```
133
134
135
136   (lambda (m)
137     (cond ((eq? m 'init) (initialize))
138           ((eq? m 'run) (set! running #t)(run))
139           ((eq? m 'pause) (set! running #f))
140           ((eq? m 'step) (next-step))
141           ((eq? m 'left) (left))
142           ((eq? m 'right) (right))
143           ((eq? m 'reset!) (reset '(1 1 1)))
144           ((eq? m 'write/erase!) (write/erase!))
145           ((eq? m 'show) (show))
146           (else (error "unknown message -- turing-machine" m))))
147
148
149   )
150
151
152
153 (define simple-inc
154   (turing "Simple incrementer"
155           'q0
156           'qf
157           'B
158           '(#(q0 1 1 right q0)
159             #(q0 B 1 stay  qf))
160           '(1 1 1)))
161
162
163 (define move-left (mk-button 26))
164 (define write/erase (mk-button 28))
165 (define move-right (mk-button 98))
166 (define pause (mk-button 27))
167 (define step (mk-button 29))
168 (define run/cont (mk-button 97))
169 (define buzzer (mk-led 99))
170 (define (rest) (gpio-delay-ms 500))
171
172 (simple-inc 'init)
173
174 (thread (lambda () (let listen
175                      ()
176                        (cond ((move-left 'pushed?) (displayln "left pushed")
176 (buzzer 'bip)(simple-inc 'left)
177                                                    (rest)(listen))
178                              ((move-right 'pushed?) (displayln "right pushed")
178 (buzzer 'bip)(simple-inc 'right)
179                                                     (gpio-delay-ms 1000)(listen))
180                              ((write/erase 'pushed?) (displayln "write pushed")
180 (buzzer 'bip)(simple-inc 'write/erase!)
181                                                      (rest) (listen))
182                              ((pause 'pushed?) (displayln "pause pushed") (buzzer
182 'bip)(simple-inc 'pause)
183                                                  (rest)(listen))
184                              ((step 'pushed?)(displayln "step pushed")(buzzer
184 'bip)(simple-inc 'step)
185                                                (rest)(listen))
186                              ((run/cont 'pushed?) (displayln "run pushed")(buzzer
186 'bip)(simple-inc 'run)
187                                                   (rest)(listen))
188                              (else (listen))))))
```