Conway's game of Life

```
PROC calculate.next.state(CHAN link[],
                          VALUE in[], state, VAR next.state)
PROC broadcast.present.state(CHAN link[], VALUE out[], state)
PROC cell(CHAN link[], VALUE in[], out[], CHAN control, sense)
PROC initialize(VALUE x, y, VAR in[], out[])

PROC move.cursor(CHAN screen, VALUE x, y)
PROC clear.screen(CHAN screen)

PROC initialize.display(CHAN screen)
PROC clean.up.display(CHAN screen)
PROC display.state(CHAN screen, VALUE x, y, state)

PROC generation(CHAN screen, control[], sense[], VAR active)

PROC edit(CHAN keyboard, screen, control[])

PROC display.activity(CHAN screen, VALUE activity)
PROC new.activity(VAR activity, VALUE ch)
PROC controller(CHAN keyboard, screen, control[], sense[])

PAR
  controller( ... )                 -- control process
  PAR x = [0 FOR array.width]           -- board
    PAR y = [0 FOR array.height]
      SEQ
        initialize( ... )
        cell( ... )
```

```
DEF dead = 0,  alive = NOT dead :      -- possible states of each cell


DEF radius     = 1 ,             -- radius of the ‘sphere of influence’
    diameter   = (2 * radius) + 1 ,
    neighbours = (diameter * diameter) - 1 :
                                 -- consequent number of neighbours




PROC calculate.next.state(CHAN link[],
                          VALUE in[], state, VAR next.state) =
  VAR count :  -- number of living neighbours
  SEQ
    VAR state.of.neighbour[neighbours] :
    SEQ
      PAR i = [0 FOR neighbours]             -- receive present state
        link[in[i]] ? state.of.neighbour[i]  --   from each neighbour
      count := 0
      SEQ i = [0 FOR neighbours]
        IF
          state.of.neighbour[i] = alive
            count := count + 1               -- count the number alive
          state.of.neighbour[i] = dead
            SKIP
    IF
      count < 2                    -- if too few
        next.state := dead         --    die from isolation
      count = 2                    -- if exactly two
        next.state := state        --    this cell is stable
      count = 3                    -- if exactly three
        next.state := alive        --    give birth if dead
      count > 3                    -- if too many
        next.state := dead   :     --    die from overcrowding




PROC broadcast.present.state(CHAN link[], VALUE out[], state) =
  -- satisfy each neighbour's need to know this cell's state
  PAR i = [0 FOR neighbours]
    link[out[i]] ! state                   :
```

```
DEF set.state = 1, ask.state = 2, terminate = 3 :


PROC cell(CHAN link[], VALUE in[], out[], CHAN control, sense) =
  -- calculate the state of a single cell on the board
  VAR state, instruction :
  SEQ
    state := dead                  -- the whole board starts off dead
    control ? instruction
    WHILE instruction <> terminate
      SEQ
        IF                         -- on instruction
          instruction = set.state
            control ? state        --     accept a new state
          instruction = ask.state
            VAR next.state :
            SEQ                    --     or calculate the next state
              PAR
                broadcast.present.state(link, out, state)
                SEQ
                  calculate.next.state(link, in, state, next.state)
                  sense ! (state <> next.state); next.state
                                   --     announce it to the controller
              state := next.state  --     and move on a generation
        control ? instruction                                  :
```

```
DEF array.width = 50,
    array.height = 20 :

DEF number.of.cells = array.height * array.width ,
    number.of.links = neighbours * number.of.cells :

PROC initialize(VALUE x, y, VAR in[], out[]) =
  -- initialize the link indirection arrays for the cell at x,y
  SEQ delta.x = [-radius FOR diameter]           -- offset of neighbour
    SEQ delta.y = [-radius FOR diameter]         -- in two dimensions
      VAR direction :   -- -4 <= direction <= +4
      SEQ
        direction := delta.x + (diameter * delta.y)
        IF
          direction <> 0
            VAR index, process :
            SEQ
              -- select outgoing channel in this direction
              process := x + (array.width * y)
              index    := (neighbours + direction) \ (neighbours + 1)
              out[index] := index + (neighbours * process)
              -- and select the corresponding incoming channel
              process := ((x + delta.x + array.width) \ array.width) +
                  (array.width *
                        ((y + delta.y + array.height) \ array.height))
              index    := (neighbours - direction) \ (neighbours + 1)
              in[index] := index + (neighbours * process)

            direction = 0       -- this cell is not its own neighbour
              SKIP                                    :
```

```
DEF control = NOT ((NOT 0) << 5),  escape = control /\ '[' :


PROC move.cursor(CHAN screen, VALUE x, y) =
  -- move to column x of line y (of a VT52)
  screen ! escape; 'Y'; '*s' + y; '*s' + x :

PROC clear.screen(CHAN screen) =
  -- clear the screen (of a VT52)
  screen ! escape; 'H' ; escape ; 'J' :


PROC initialize.display(CHAN screen) =
  -- display an entirely dead board
  clear.screen(screen)  :

PROC clean.up.display(CHAN screen) =
  -- move away from board
  move.cursor(screen, 0, array.height)  :

PROC display.state(CHAN screen, VALUE x, y, state) =
  -- display the state of one cell
  SEQ
    move.cursor(screen, x, y)
    IF
      state = alive             -- live cells show as an asterisk
        screen ! '**'
      state = dead             -- dead ones as a blank space
        screen ! '*s'                         :


PROC generation(CHAN screen, control[], sense[], VAR active) =
  -- cause the colony on the board to move on one generation
  SEQ
    SEQ cell = [0 FOR number.of.cells]     -- invite each cell
      control[cell] ! ask.state            --     to make progress
    active := FALSE
    SEQ cell = [0 FOR number.of.cells]     -- for each cell
      VAR changed, next.state :
      SEQ
        sense[cell] ? changed; next.state  --     receive its new state
        IF
          changed                          --     and display it
            SEQ
              display.state(screen, cell \ array.width,
                                    cell / array.width, next.state)
              active := TRUE
          NOT changed
            SKIP                         :
```

```
PROC edit(CHAN keyboard, screen, control[]) =
  -- modify the colony on the board

  DEF ctrl = NOT ((NOT 0) << 5),  otherwise = TRUE :
  DEF left.key  = ctrl /\ 'H',    right.key = ctrl /\ 'L',
      up.key  = ctrl /\ 'K',      down.key  = ctrl /\ 'J',
      uproot.key = '*s',          plant.key = '**' :

  VAR x, y, editing, ch :
  SEQ
    x := array.width / 2       -- set co-ordinates of cursor
    y := array.height / 2      -- to the centre of the board
    editing := TRUE
    WHILE editing
      SEQ
        move.cursor(screen, x, y)
        keyboard ? ch
        IF
          (ch = left.key) AND (x > 0)
            x := x - 1
          (ch = right.key) AND (x < (array.width - 1))
            x := x + 1
          (ch = up.key) AND (y > 0)
            y := y - 1
          (ch = down.key) AND (y < (array.height - 1))
            y := y + 1
          (ch = uproot.key) OR          -- change the state of the
                     (ch = plant.key)   -- cell under the cursor
            VAR state :
            SEQ
              state := (dead /\ (ch = uproot.key)) \/
                       (alive /\ (ch = plant.key))
              control[x + (array.width * y)] ! set.state; state
              display.state(screen, x, y, state)
          (ch = 'q') OR (ch = 'Q')
            editing := FALSE
          otherwise -- ignore anything that is not understood
            SKIP                                  :
```

```
DEF idle           = 1,
    editing        = 2,
    single.stepping = 3,
    free.running   = 4,
    terminated     = 5 :

PROC display.activity(CHAN screen, VALUE activity) =
  -- display state of the controller
  SEQ
    move.cursor(screen, array.width + 1, array.height / 2)
         -- move to a place off the right of the board
    IF
      activity = idle
        write.string(screen, "Idle")
      activity = editing
        write.string(screen, "Edit")
      activity = single.stepping
        write.string(screen, "Step")
      activity = free.running
        write.string(screen, "Busy")
      activity = terminated
        write.string(screen, "Done")                :


PROC new.activity(VAR activity, VALUE ch) =
  IF                            -- type 'ch' on the keyboard ...
    (ch = 'q') OR (ch = 'Q')       -- ... Q to finish the program
      activity := terminated
    (ch = 's') OR (ch = 'S')       -- ... S to halt evolution
      activity := idle
    (ch = 'e') OR (ch = 'E')       -- ... E to start editing
      activity := editing
    (ch = 'r') OR (ch = 'R')       -- ... R to start evolution
      activity := free.running
    otherwise                      -- ... or anything else for
                                   --    just one generation
      activity := single.stepping  :
```

```
PROC controller(CHAN keyboard, screen, control[], sense[]) =
  -- control the activity of the colony on the board
  -- under direction from the keyboard
  VAR activity :
  SEQ
    activity := idle
    initialize.display(screen)
    WHILE activity <> terminated
      SEQ
        display.activity(screen, activity)
        VAR ch :
        PRI ALT
          (activity <> editing) & keyboard ? ch
                  -- if not editing, type to change activity
            new.activity(activity, ch)
          (activity = editing) & SKIP
            SEQ
              edit(keyboard, screen, control)
              activity := idle
          (activity = free.running) OR      -- if evolving
          (activity = single.stepping) &
                  SKIP                      -- but nothing typed
            VAR changing :
            SEQ
              generation(screen, control, sense, changing)
                                            -- move on a generation
              IF
                (activity = single.stepping) OR (NOT changing)
                  activity := idle
                (activity = free.running) AND changing
                  SKIP
    display.activity(screen, activity)
    SEQ cell = [0 FOR number.of.cells]
      control[cell] ! terminate
    clean.up.display(screen)                           :
```

```
CHAN link[number.of.links],
     control[number.of.cells],
     sense[number.of.cells] :
PAR
  controller(keyboard, screen, control, sense)     -- control process
  PAR x = [0 FOR array.width]                      -- board
    PAR y = [0 FOR array.height]
      VAR in[neighbours], out[neighbours] :
      SEQ
        initialize(x, y, in, out)
        cell(link, in, out,
               control[x+(array.width*y)], sense[x+(array.width*y)])
```