8 Conway's game of Life

Lest you be misled by the name, *Life* is neither a competitive game between several players, nor yet a solitaire game in which a player competes against the collusion between the rules and the roll of the dice. The game is more a simulation, in which the evolution of a system is fully determined by a set of rules.

To be precise, Life is played on an infinite square board: that means that there are a number of squares, or *cells*, each of which has four immediate neighbours and four diagonal neighbours, in the fashion of a chess board. That the board is infinite means simply that every cell in which you will be interested is one with a full complement of neighbours, so that you need never worry about what happens at the edges. There will be only a finite number of interesting cells to think about at any one time. Each cell may be in one of two states: occupied (*alive*) or unoccupied (*dead*), and only finitely many will be alive at any time.

The rules describe the succession of states of each cell in terms of earlier states of that cell and of its eight near neighbours. Each cell passes through a sequence of generations, with the state of the cell in the next generation being determined by its state in this generation, and by the number of cells adjacent to it which are alive in this generation. If a cell is currently alive, and if it has less than two live neighbours, it is deemed to die of loneliness, and will be dead in the next generation. A live cell with two or three neighbours alive in the same generation survives into the next generation, but if it has four or more contemporaries, it will be dead from overcrowding by the next generation. A dead cell with exactly three live neighbours in this generation will give birth and be alive in the next generation, otherwise it will remain barren.

Notice that the rules determine the state of the whole board in the next generation in terms of its state in the present generation. Moreover, the rules are expressed in purely local terms, and the property of Life that makes it interesting is that these local rules can control the evolution of global structures. A number of patterns of live cells are known to pass through cycles of growth and decline, some are known to grow without limit, whilst others die out.

Although the rules of evolution are simple, applying them to a pattern large enough to be interesting, for more than one or two generations, is a tedious business. Machine assistance makes it possible to watch the long term development of substantial colonies, and Life was once a popular way of consuming otherwise unused machine cycles! More practically, a Life board is a particularly simple and symmetrical example of a systolic cellular array. These are studied by VLSI designers seeking algorithms with fast but simple implementations in highly parallel hardware. A systolic array is characterized by the achievement of global co-operation through many simultaneous calculations organized by local communications. Ideally, the components of the array are, like the cells of a Life board, all of a few basic types, have a small finite amount of state, and need never know where they are in the array.

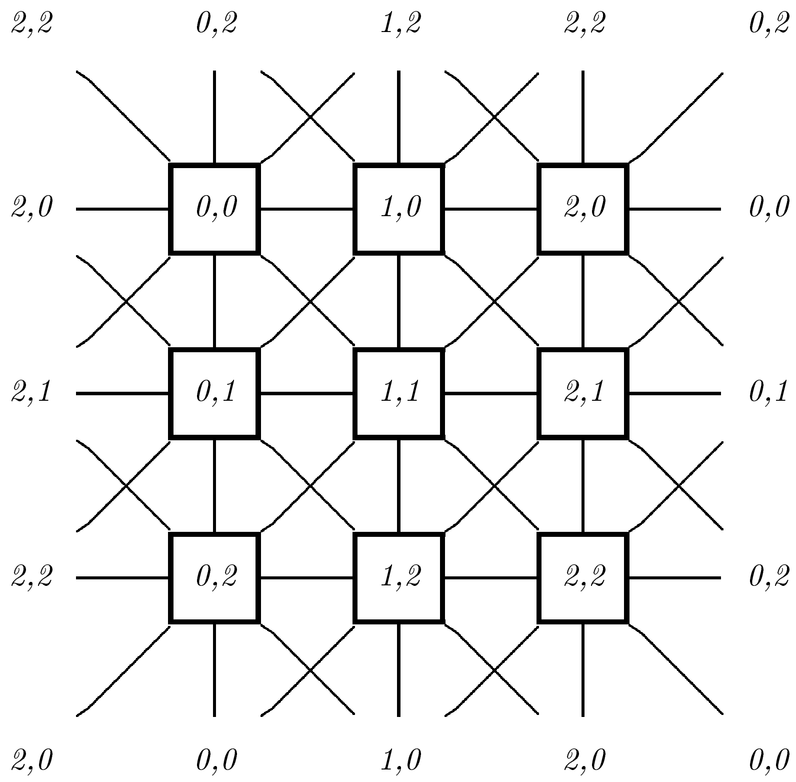
The program described here is, as with the parallel sorter, a simulation in two parts: there is a plane of parallel processes in which the cells of a Life board are represented, one cell to a process; added to this is an essentially sequential mechanism for guiding and watching the evolution of the colony. Perhaps it is worth pointing out at the outset that the resulting program, run on a single processor, is far from the fastest way of playing Life. There are, for example, a number of optimizations that require each process to have a more global view of the state of the board, and naturally give rise to a sequential program. This program is here for two reasons: firstly as an intricate example of the interconnection of processes, showing how to separate this from the workings of the processes themselves; secondly, it is an example of a general method of adding global synchronization to a loosely coupled system in order to observe its behaviour.

The Life board

There is no problem in selecting a representation the board. Each cell of the board has a state, so is represented by a process which administers the variable in which that state is stored. There is no reason why each of these processes should not be identical. Each cell is distinguished only by the particular eight other cells which are close enough to influence its state in the next generation. The neighbours of a cell process are connected to it, each by a pair of channels, one in each direction.

The first problem that arises is one of representing an infinite board on what must necessarily be a finite array of processes. As suggested earlier, the requirement of an infinite board is made so that the behaviour of a cell will not be influenced by its being at an edge of the board. Unless a colony grows without limit, or moves *en masse* in some direction, a finite board will do, since the evolution of a colony is unaffected by any amount of dead space around it.

One solution, and the one that I have adopted here, is to take a finite sized board and wrap it around a torus, so that the cells on the top edge have neighbours on the bottom edge, and those on the right have neighbours on the left.



There are now no edges to worry about. You may think of this toroidal board in either of two ways. Looking at it as a flat board with tricky edges, it correctly implements the rules of healthy living until one or more of the edge cells gives birth, from which point on it is possible for things to go wrong, with miraculous conceptions and unexplained deaths happening in ways not predicted by the rules. Another way of thinking about it is that the toroidal board behaves as if it were a fragment of a truly infinite flat board on which the real finite colony that you can see is repeated, in the fashion of a wallpaper pattern, at regular intervals in the horizontal and vertical directions. The boundary effects are now explicable, since they are the effects (predicted by the rules) of a neighbouring copy of the colony coming close enough to influence the visible part of the board.

To be more definite about the program, it consists of a rectangular array of cell processes

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

PAR x = [0 FOR array.width]
  PAR y = [0 FOR array.height]
    ... process representing cell x, y
```

The neighbours of cell x, y are those indexed

$$\big\langle ((x \pm 1) + \mathit{array.width}) \backslash \mathit{array.width}, \\ ((y \pm 1) + \mathit{array.height}) \backslash \mathit{array.height} \big\rangle$$

with the remainder operator taking care of the proximity of cells at opposite edges of the board. The numerator of the remainder operator has to be made non-negative, since in *occam* it is defined that the value of `((-1)\w)` is minus one.

The next thing to decide is the arrangement of the channels connecting these processes. As in the matrix multiplier example, it would be possible to allocate one channel array to account for all of the data flowing in each compass direction. The result would be that each cell process would be connected to eight individually named channels carrying data inwards, and eight individually named channels carrying data outwards. This is to ignore the symmetry with which the rules of living treat the neighbours of a process. A cell does not discriminate between its neighbours according to their compass direction, but treats them uniformly. The symmetry should be represented by a `FOR` loop in the cell processes, the body being executed eight times, once for each neighbour. That suggests that an array of eight channels is needed, indexed by the eight directions.

Since there are, in *occam*, neither channel variables nor channel pointers, the only neat solution to this problem is to allocate all of the channels from a single large array. Each cell then needs to be told which eight subscripts it should use to select its incoming channels, and which eight to select its outgoing ones.

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

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

PROC initialize(VALUE x, y, VAR in[], out[]) =
  ... initialize in[...] and out[...]

PROC cell(CHAN link[], VALUE in[], out[]) =
  ... cell using link[in[...]] and link[out[...]]

CHAN link[number.of.links] :
PAR x = [0 FOR array.width]
  PAR y = [0 FOR array.height]
    VAR in[neighbours], out[neighbours] :
      SEQ
        initialize(x, y, in, out)
        cell(link, in, out)
```

Perhaps this is the place to note that I remain unsatisfied by this solution because of the generality of the variable arrays *in*[ ] and *out*[ ]. A mechanical checker, such as might be a part of an *occam* compiler, is unlikely to be able to verify that the *cell* makes only legal use of the *link* channels, since the uses appear to be dynamically determined. A mechanically checkable program would most probably have to recompute the subscripts at the point of use. It is because the effort of recomputing complex subscript expressions would dominate all of the other activity in the program that I have adopted this solution.

The remainder of the board configuration is in the initialization of the indirection arrays, *in*[ ] and *out*[ ]. To do this, an enumeration of the processes and the channels must be chosen. I have chosen to count the processes in the usual way: along the rows then down the colums, from zero at process x zero, y zero in the top left

```
this.process := x + (array.width * y)
```

and to allocate the first eight channels to carry data out of the first process, the next eight out of the next process, and so on. This accounts for all of the channels exactly once, as every channel is outward bound from some process.

To settle on a particular enumeration of channels, the eight neighbours of a process must be put in some order. I choose order of increasing *direction* as computed by the loop

```
SEQ delta.x = [-radius FOR diameter]
  SEQ delta.y = [-radius FOR diameter]
    VAR direction :
    SEQ
      direction := delta.x + (diameter * delta.y)
      ... consider neighbour x+delta.x  y+delta.y
```

which is (except at the top and left edges) the order of increasing process number. The direction of a neighbour characterizes it, and lies in the range

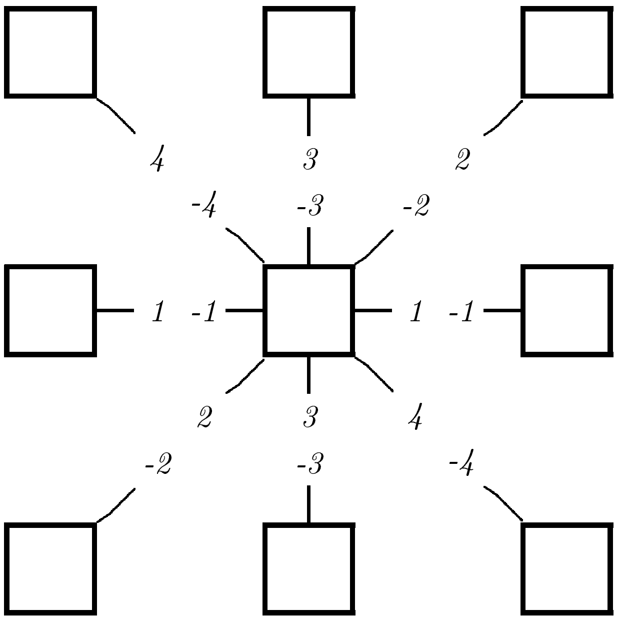
$$-(\mathit{neighbours}/2) \quad \leq \quad \mathit{direction} \quad \leq \quad +(\mathit{neighbours}/2)$$

with the zero value corresponding to the cell at x, y itself. To fill an array with the *neighbour* consecutive subscripts of outward going channels the non-zero values of direction must be mapped onto consecutive indices for *out*, and a group of eight consecutive channel numbers

```
IF
  direction <> 0
    VAR this.index :
    SEQ
      this.index := (neighbours + direction) \
                                        (neighbours + 1)
      out[this.index] := this.index +
                              (neighbours * this.process)
  direction = 0
    SKIP
```

The value of *this.index* constructed in this way ranges from zero to $\mathit{neighbours} - 1$, taking on each value exactly once, in the course of a scan of the neighbours.

Now the question arises of which are the correct subscripts to use to select the incoming links. Incoming links at *this.process* are, if looked at from the other end, the outgoing links from the neighbours of *this.process*.



The simplest, brute force, solution to the problem of enumerating them is to put yourself in the position of the process at the other end of each channel, and to ask which link that process would be using to talk to *this.process*, as one of its neighbours. The process at x, y is a neighbour of each of its own neighbours; in particular it is the neighbour in the $-direction$ direction of the process which is its neighbour in the $+direction$ direction. (You can see this because *direction* is linear in *delta.x* and in *delta.y*.) This means that the inward channel from the neighbour in the $+direction$ direction is the one that, at that other process would be described as the outward channel in the $-direction$ direction.

```
SEQ
  other.x := (x + delta.x + array.width)\array.width
  other.y := (y + delta.y + array.height)\array.height
  other.process := other.x + (array.width * other.y)
  other.index   := (neighbours - direction) \
                                         (neighbours + 1)
  in[other.index] := other.index +
                             (neighbours * other.process)
```

I have gathered these fragments in the process *initialize* which appears in the appendix, leaving only the process *cell* to be coded. This is comparatively simple: recording the state of the cell, whether it is *dead* or *alive*, and controlling its evolution

```
DEF dead = 0, alive = NOT dead :

PROC cell(CHAN link[], VALUE in[], out[]) =

  PROC broadcast.present.state(CHAN link[],
                                VALUE out[], state) =
    ... tell neighbours about the state of this cell

  PROC calculate.next.state(CHAN link[],
                 VALUE in[], state, VAR next.state) =
    ... evolve in keeping with the rules

  VAR state :
  SEQ
    state := ...              -- set an initial state
    WHILE TRUE
      VAR next.state :
      SEQ
        PAR
          broadcast.present.state(link, out, state)
          calculate.next.state(link,
                                in, state, next.state)
        state := next.state           :
```

I will postpone the matter of what initial value is given to *state*, which determines the type of colony being watched. The second half of this chapter will describe an editor that allows the user to set an initial configuration from the keyboard.

In each generation, each *cell* must learn the state of each of its neighbours. There is a corresponding obligation on a *cell* to tell each of its neighbours about its own current state

```
PROC broadcast.present.state(CHAN link[],
                              VALUE out[], state) =
  PAR i = [0 FOR neighbours]
    link[out[i]] ! state                 :
```

To calculate its next state, the *cell* counts the number of adjacent occupied cells, and applies the local rule

```
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]
        link[in[i]] ? state.of.neighbour[i]
      count := 0
      SEQ i = [0 FOR neighbours]
        IF
          state.of.neighbour[i] = alive
            count := count + 1
          state.of.neighbour[i] = dead
            SKIP
    IF
      count < 2     -- death from isolation
        next.state := dead
      count = 2     -- this cell is stable
        next.state := state
      count = 3     -- stable if alive, a birth if dead
        next.state := alive
      count > 3     -- death from overcrowding
        next.state := dead   :
```

Notice that although the input processes are written in a parallel `FOR` loop, the counting of live neighbours has to be sequential, since the *count* variable must not be shared. Whilst the simplest of mechanical checkers would be justified in drawing the programmer's attention to the shared array *state.of.neighbour*[ ], it is clear that no element of the array is shared.

Observation and control

As with the parallel sorter, having completed the highly parallel part of the program, I have still to design a means for controlling and watching what happens. The observation will impose more synchronization on the array of cells: there is, so far, nothing to prevent widely separated processes from working on as widely separated generations, but the display should be capable of showing the state of one generation at a time, across the whole of the board.

There are three intructions that the controlling process will need to issue to each cell on the board: it may ask for the cell to assume a new state, so as to initialize, and subsequently edit, the state of the board; it may instruct the cell to evolve for one generation; and it may tell the cell process to terminate.

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

In response to instructions to evolve, the cell should yield up its new state. To carry these messages, a channel is needed into each cell, and one from each cell, and the *cell* process must be recoded to expect and comply with the instructions received on its *control* channel:

```
PROC cell(CHAN link[], VALUE in[], out[],
                                CHAN control, sense) =
  VAR state, instruction :
  SEQ
    state := dead    -- the whole board starts off dead
    control ? instruction
    WHILE instruction <> terminate
      SEQ
        IF
          instruction = set.state
            control ? state
          instruction = ask.state
            VAR next.state :
            SEQ
              PAR
                broadcast.present.state(link, out,
                                                 state)
                SEQ
                  calculate.next.state(link, in,
                                     state, next.state)
                  sense ! (state <> next.state);
                          next.state
              state := next.state
        control ? instruction                      :
```

This coding of *cell* expects a sequence of instructions, each of which is either a *set.state* followed by a new state to be assumed, or is an *ask.state* which causes the cell to evolve and report its new state. The whole sequence is followed by a single instruction to terminate. At the end of each generation, the *cell* process sends not only its new state, but also an indication of whether the state has changed in this generation. That makes the task of the controlling process simpler.

Since there is only one terminal keyboard and one terminal screen involved, the controller is essentially a sequential process, with each of the *control* and *sense* channels connected to it

```
CHAN link[number.of.links],
     control[number.of.cells], sense[number.of.cells] :
PAR
  controller(keyboard, screen, control, sense)
  PAR x = [0 FOR array.width]
    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)])
```

The *controller* is written as a state machine, whose state is changed by characters typed on the keyboard

```
PROC new.activity(VALUE ch, VAR activity) =
  ... select a new activity
```

Under the control of input from the terminal keyboard, it calls processes either to modify the state of cells on the board by issuing *set.state* instructions

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

or to drive the whole board through the evolution of a single generation by scanning the board, issuing *ask.state* instructions, and reading back the new states.

```
PROC generation(CHAN screen, control[], sense[],
                                          VAR active) =
  ... cause the colony on the board to move on a step
```

The *active* parameter returns an indication of whether any changes have happened during the generation.

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

PROC controller(CHAN keyboard, screen,
                              control[], sense[]) =
  VAR activity :
  SEQ
    activity := idle
    initialize.display(screen)
    WHILE activity <> terminated
      SEQ
        display.activity(screen, activity)
        VAR ch :
        PRI ALT

          (activity <> editing) & keyboard ? ch
            new.activity(activity, ch)

          (activity = editing) & SKIP
            SEQ
              edit(keyboard, screen, control)
              activity := idle

          (activity = free.running) OR
                  (activity = single.stepping) & SKIP
            VAR changing :
            SEQ
              generation(screen, control,
                                       sense, changing)
              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)                       :
```

The normal activity of the controller, *free.running*, is to cause a sequence of invocations of *generation* so that the colony is continually evolving. The alternative has to be asymmetric, because a sequence of calls to *generation* might otherwise go on indefinitely without ever allowing pending keyboard input to be accepted.

Between any two generations, the keyboard has an opportunity to change the activity, by calling *new.activity*

```
PROC new.activity(VAR activity, VALUE ch) =
  -- type 'ch' on the keyboard ...
  IF
    (ch = 'q') OR (ch = 'Q')
                          -- ... Q to finish 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 :
```

The *single.step* activity, entered by typing almost anything on the keyboard, causes an evolution of precisely one generation. This makes it easier to follow the details of a history. The code of the board is entirely unaffected by the detailed design of the single stepping mechanism, and even of the details of the editor.

Recall that each cell starts a step of its evolution in response to an *ask.state* instruction, on its *control* channel. It cannot complete the advance unless its neighbours are also on the move, so the *generation* process starts each cell on the board, and then collects a new state from each of the cells.

```
PROC generation(CHAN screen, control[], sense[],
                                        VAR active) =
  SEQ
    SEQ cell = [0 FOR number.of.cells]
      control[cell] ! ask.state
    active := FALSE
    SEQ cell = [0 FOR number.of.cells]
      VAR changed, next.state :
      SEQ
        sense[cell] ? changed; next.state
        IF
          changed
            SEQ
              display.state(screen, cell \ array.width,
                         cell / array.width, next.state)
              active := TRUE
          NOT changed
            SKIP                    :
```

When the new states are gathered, any changes are notified on the display. A note is also kept of whether *any* of the cells has changed state, because a colony which does not change in one generation has become stable, and will never change again. If the colony becomes stable, the *active* parameter to *generation* is returned as `FALSE` and the controller becomes *idle*.

The details of the display are all contained in the process to set up the screen, to display the state of a cell, to display the activity of the controller, and to clear up the screen after the program has terminated. Assuming that the board is some tens of cells on a side, I have allocated consecutive screen locations to adjacent cells

```
PROC display.state(CHAN screen, VALUE x, y, state) =
  SEQ
    move.cursor(screen, x, y)
    IF
      state = alive
        screen ! '**'
      state = dead
        screen ! '*s'                          :
```

A live cell shows as an asterisk, and a dead cell as a blank space.

To make the initial screen consistent with the initial state of the board, which is entirely dead, it suffices to clear the screen

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

and to clean up at the end of the program, the cursor is moved to the left of the line below the image of the board

```
PROC clean.up.display(CHAN screen) =
  move.cursor(screen, 0, array.height)  :
```

Assuming that there is some spare room on the screen to the right of the image of the board, the activity of the controller can be displayed there

```
PROC display.activity(CHAN screen, VALUE activity) =
  SEQ
    move.cursor(screen, array.width+1, array.height/2)
    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")                :
```

The terminal specific declarations are just those of the process invoked to alter the screen:

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

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

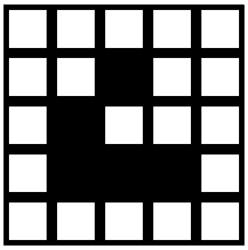
All that remains is to supply an editor. Here is a simple process that allows a cursor to be moved around the board image, and allows the state of the cell under the cursor to be set

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

  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
    y := array.height / 2
    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 (ch = plant.key)
            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
            SKIP                                    :
```

Editing continues until a character ‘Q’ is typed. The cursor control keys move the cursor vertically and horizontally over the board, the space bar kills the occupant of a cell, and the asterisk key plants a new occupant. For simplicity, any other character, or an attempt to pass over the boundary of the board image is ignored without comment.

Life

A brief word seems to be in order about the history of *Life* itself. It first became widely known through Martin Gardner's column *Mathematical Games* in the *Scientific American* magazine, in October 1970 (pp. 120–123) and May 1971 (pp. 112–117). The former article explains the rules, and introduces some of the jargon of the subject: for example, the speed of light, which is one cell width per generation, the greatest rate at which information can pass across the board; and the glider, a small, fixed size, moving colony



The glider is one of the small, simple colonies whose evolution is fully known: it moves across the board in the direction in which it appears to be pointing, at a quarter of the speed of light, passing through a fixed sequence of four distinct forms.

The second article describes more complicated examples, drawn from the readers' experience of wasting both machine cycles and mathematical ingenuity. Here you will find the curiosities of the subject: Garden of Eden colonies, which are ones that cannot possibly have come about as a result of an evolutionary advance from a former state; the glider gun, a huge structure which grows without limit, by firing an unending stream of gliders from one of its extremities; and a glider-gobbler which, although stable in itself, can also swallow a stream of gliders such as that given off by the gun, to no ill effect. There are viruses, which disrupt regular structures, and regular structures which can recover their symmetry after withstanding a virus attack.