13

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, like the parallel sorter of Chapter 12, 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.

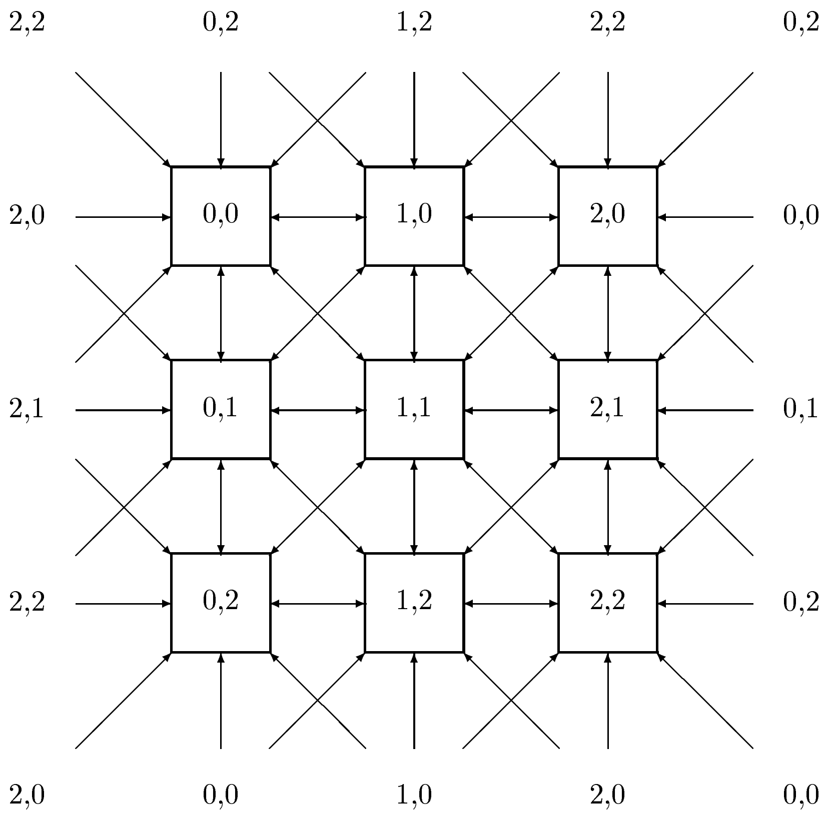
The Life board

There is no problem in selecting a representation the board. Each cell of the board has a state, and 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 along which to exchange information about states, 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 is 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, as in Figure 13.1. There are now no edges about which to worry.

Figure 13.1 A toroidal board with circumference 3



You can 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 in parallel

```
VAL INT array.width IS ... :
VAL INT array.height IS ... :

PAR x = 0 FOR array.width
  PAR y = 0 FOR array.height
    ... process representing cell (x, y)
```

The neighbours of cell $\langle x, y\rangle$ are those with an ordinate in the set

$$\{((x-1)+\mathit{array.width})\backslash \mathit{array.width},\ x,\ (x+1)\backslash \mathit{array.width}\}$$

and abscissa among the elements of

$$\{((y-1)+\mathit{array.height})\backslash \mathit{array.height},\ y,\ (y+1)\backslash \mathit{array.height}\}$$

The remainder operator is used to take 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 $(-1)\ \backslash\ n$ is always 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. This 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, both for input and for output.

However, since the channels need to be shared with a number of the other component processes, and it is not possible to produce a table of channels drawn from different sources, this can be achieved for only one type of communication. Each process may 'own' the channels on which it outputs, but it would then have to listen to those 'belonging' to the eight neighbours. A single large three-dimensional array, indexed by the x and y co-ordinates of the transmitting cell and by the eight directions, is ideal for allocating output channels; some arithmetic is required to determine which component of which sub-arrays are allocated to each process for input.

Consistent usage of shared arrays requires that this arithmetic selects distinct channels in each process. You should certainly be able to check this disjointness yourself, although it might be too tricky for a mechanical checker, a part of an occam 2 compiler. The identifiers involved are either constants (*array.width*, *array.height*) or will be the bound variables of constant-bounded FOR loops (x, y and the direction d), but a compiler which attempts to check the definition and the calls of *cell* independently of each other will almost certainly fail.

Rather than recalculate the indices of the neighbours in each direction each time, which would dominate the rest of the calculations involved, it is preferable to build them into tables *nx* and *ny* for each process, as in Process 13.1.

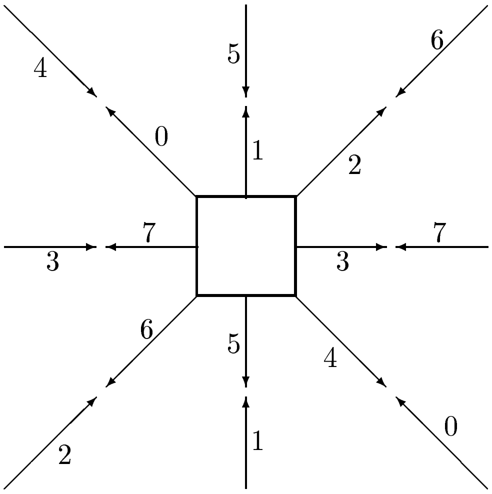
Process 13.1

```
VAL INT radius     IS 1 :   -- of the 'sphere of influence'
VAL INT diameter   IS (2 * radius) + 1 :
VAL INT neighbours IS (diameter * diameter) - 1 :

VAL INT number.of.cells IS array.height * array.width :
VAL INT number.of.links IS neighbours * number.of.cells :

PROC cell([][][]CHAN OF STATE link,
          VAL INT x, y, VAL []INT nx, ny )
  -- cell using link[x][y][d] for output
  --        and link[nx[d]][ny[d]][d] for input
  ...
:

[array.width][array.height][neighbours]CHAN OF STATE link :
PAR x = 0 FOR array.width
  PAR y = 0 FOR array.height
    VAL [neighbours]INT nx IS ... :
    VAL [neighbours]INT ny IS ... :
    cell(link, x, y, nx, ny)
```

The remaining question is which are the correct subscripts to use to select the incoming links. First numbers between 0 and $neighbours - 1$ must be assigned to the directions. The precise enumeration does not particularly matter; that adopted here is to number them clockwise from top-left.

Incoming links at $\langle x, y\rangle$ are, if looked at from the other end, the outgoing links from its neighbours. If the output of each cell is to be sent in direction d along a channel with final index d, then it is not the cell in direction d to which this cell must listen on the input channel with final index d; rather, as can be seen from Figure 13.2, it is the diametrically opposite cell. Indeed, precisely because the outgoing channel array $link_{x,y}$ will be treated uniformly by *cell*, it is really the selection of the incoming channels

Figure 13.2 The numbering of directions



which fixes the meaning of the direction numbers.

Having calculated the indices of the adjacent rows and columns, the tabulation of *nx* and *ny* is really no more than reading the appropriate horizontal and vertical displacements for each *d* corresponding to the outer ring of arrows on the diagram. This gives us Process 13.2 as the body of the program.

Process 13.2

```
PAR x = 0 FOR array.width
  PAR y = 0 FOR array.height
    VAL INT left  IS ((x - 1) + array.width)  \ array.width :
    VAL INT right IS  (x + 1)                 \ array.width :
    VAL INT up    IS  (y + 1)                 \ array.height :
    VAL INT down  IS ((y - 1) + array.height) \ array.height :
    VAL [neighbours]INT nx IS
          [ right, x,    left, left, left, x,  right, right ] :
    VAL [neighbours]INT ny IS
          [ down,  down, down, y,    up,   up, up,    y     ] :
    cell(link, x, y, nx, ny)
```

With the framework now fixed, the code of the *cell* process is quite simple. It records the state of the cell, whether it is dead or alive, and controls the evolution of the state between generations. Since the state has only two possible values, it may as well be modelled with a variable of type `BOOL`, using the following definitions to give Process 13.3:

```
PROTOCOL STATE IS BOOL :
VAL BOOL alive IS TRUE :
VAL BOOL dead IS NOT alive :
```

Process 13.3

```
PROC cell([][][]CHAN OF STATE link,
          VAL INT x, y, VAL []INT nx, ny )
  PROC broadcast.present.state([][][]CHAN OF STATE link,
                               VAL INT x, y, VAL BOOL state )
    ... tell neighbours about the state of this cell
  :
  PROC calculate.next.state([][][]CHAN OF STATE link,
                            VAL []INT nx, ny,
                            VAL BOOL state, BOOL next.state )
    ... evolve in keeping with the rules
  :
  BOOL state :
  SEQ
    ... set the state to some appropriate value
    WHILE TRUE
      BOOL next.state :
      SEQ
        PAR
          broadcast.present.state(link, x, y, state)
          calculate.next.state(link, nx, ny,
                               state, next.state )
        state := next.state
:
```

The initial value given to the cell's *state* variable determines the type of colony being watched. A later section 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 (Process 13.4).

Process 13.4

```
PROC broadcast.present.state([][][]CHAN OF STATE link,
                             VAL INT x, y, VAL BOOL state )
  PAR d = 0 FOR neighbours
    link[x][y][d] ! state
:
```

To calculate its next state, the *cell* counts the number of adjacent cells which have told it they are occupied, and applies the local rule, as in Process 13.5. 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 array *state.of.neighbour* is shared between parallel processes, it is clear that no element of the array is shared, and this can be checked by the compiler as d is a bound variable in a `FOR` loop with constant bounds.

Process 13.5

```
PROC calculate.next.state([][][]CHAN OF STATE link,
                          VAL []INT nx, ny,
                          VAL BOOL state, BOOL next.state )
  INT count :       -- number of living neighbours
  SEQ
    [neighbours]BOOL state.of.neighbour :
    SEQ
      PAR d = 0 FOR neighbours
        link[nx[d]][ny[d]][d] ? state.of.neighbour[d]
      count := 0
      SEQ d = 0 FOR neighbours
        IF
          state.of.neighbour[d] = alive
            count := count + 1
          state.of.neighbour[d] = 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
:
```

Observation and control

As in Chapter 12, having completed the highly parallel part of the program, it remains 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 instructions 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. In response to instructions to evolve, the cell should yield up its new state. In fact, at the end of each generation, the *cell* process will send 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. 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 (Process 13.6). 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 *evolve* which causes the cell to evolve and report its new state. The whole sequence is followed by a single instruction to terminate.

Process 13.6

```
PROTOCOL COMMAND
  CASE
    set.state; BOOL
    evolve
    terminate
:

PROTOCOL RESPONSE IS BOOL; BOOL :


PROC cell([][][]CHAN OF STATE link,
          VAL INT x, y, VAL []INT nx, ny,
          CHAN OF COMMAND control,
          CHAN OF RESPONSE sense         )
  BOOL state, not.finished :
  SEQ
    state := dead    -- the whole board starts off dead
    not.finished := TRUE
    WHILE not.finished
      control ? CASE

        set.state; state
          SKIP       -- state has been set to the new value

        evolve
          BOOL next.state :
          SEQ
            PAR
              broadcast.present.state(link, x, y, state)
              SEQ
                calculate.next.state(link, nx, ny,
                                     state, next.state )
                sense ! (state <> next.state); next.state
            state := next.state

        terminate
          not.finished := FALSE
:
```

Since there is only one terminal keyboard and one terminal screen involved, the *controller* is essentially a sequential process, with all of the *control* and *sense* channels connected to it. The new program has the structure shown in Process 13.7.

Process 13.7

```
[array.width][array.height][neighbours]CHAN OF STATE link :
[array.width][array.height]CHAN OF COMMAND control :
[array.width][array.height]CHAN OF RESPONSE sense :
PAR
  controller(keyboard, screen, control, sense)
  PAR x = 0 FOR array.width
    PAR y = 0 FOR array.height
      ... declarations of nx and ny :
      cell(link, x, y, nx, ny, control[x][y], sense[x][y])
```

The *controller* (Process 13.8) is written as a state machine; its state may be represented by a variable of type INT, and set to values determined by characters typed on the keyboard

```
INT FUNCTION new.activity(VAL BYTE char)
  ... select a new activity
:
```

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

```
PROC editor(CHAN OF BYTE keyboard, screen,
            [][]CHAN OF COMMAND 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 *evolve* instructions, and reading back the new states

```
PROC generation(CHAN OF BYTE screen,
                [][]CHAN OF COMMAND control,
                [][]CHAN OF RESPONSE sense,
                BOOL 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.

Process 13.8

```
VAL INT idle         IS 0 :  -- controller activity values
VAL INT editing      IS 1 :
VAL INT single.step  IS 2 :
VAL INT free.running IS 3 :
VAL INT terminated   IS 4 :

PROC controller(CHAN OF BYTE keyboard, screen,
                [][]CHAN OF COMMAND control,
                [][]CHAN OF RESPONSE sense    )
  INT activity :
  SEQ
    activity := idle
    initialize.display(screen)
    WHILE activity <> terminated
      SEQ
        display.activity(screen, activity)
        BYTE char :
        PRI ALT
          (activity <> editing) & keyboard ? char
            activity := new.activity(char)
          (activity <> idle) & SKIP
            CASE activity
              editing
                SEQ
                  editor(keyboard, screen, control)
                  activity := idle
              free.running, single.step
                BOOL changing :
                SEQ
                  generation(screen, control, sense, changing)
                  IF
                    (activity = single.step) OR (NOT changing)
                      activity := idle
                    (activity = free.running) AND changing
                      SKIP
    display.activity(screen, activity)
    PAR x = 0 FOR array.width
      PAR y = 0 FOR array.height
        control[x][y] ! 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. As it is, between any two generations the keyboard is given an opportunity to change to the *new.activity* which Process 13.9 associates with the character typed.

Process 13.9

```
INT FUNCTION new.activity(VAL BYTE char)
  INT activity :
  VALOF
    CASE char     -- typed on the keyboard ...
      'q', 'Q'              -- ... Q to finish program
        activity := terminated
      's', 'S'              -- ... S to halt evolution
        activity := idle
      'e', 'E'              -- ... E to start editing
        activity := editing
      'r', 'R'              -- ... R to start evolution
        activity := free.running
      ELSE  -- ... or anything else for one generation
        activity := single.step
    RESULT activity
:
```

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. Notice that 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 *evolve* instruction on its *control* channel. It cannot complete the advance unless its neighbours also move, so the *generation* process, Process 13.10, starts each cell on the board, and only then collects a new state from each of the cells. When the new states are gathered, any changes are notified on the display. 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 never changes 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 procedures 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, it is reasonable to allocate consecutive screen locations to adjacent cells, and so use the co-ordinates of the cell as those of its representation on the screen (Process 13.11). 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 (Process 13.12), 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 (Process 13.13). Assuming that there is some

Process 13.10

```
PROC generation(CHAN OF BYTE screen,
                [][]CHAN OF COMMAND control,
                [][]CHAN OF RESPONSE sense,
                BOOL active                )
  SEQ
    PAR x = 0 FOR array.width
      PAR y = 0 FOR array.height
        control[x][y] ! evolve
    active := FALSE
    SEQ x = 0 FOR array.width
      SEQ y = 0 FOR array.height
        BOOL changed, next.state :
        SEQ
          sense[x][y] ? changed; next.state
          IF
            changed
              SEQ
                display.state(screen, x, y, next.state)
                active := TRUE
            NOT changed
              SKIP
:
```

Process 13.11

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

Process 13.12

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

Process 13.13

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

Process 13.14

```
PROC display.activity(CHAN OF BYTE screen,
                      VAL INT activity    )
  SEQ
    move.cursor(screen, array.width+1, array.height/2)
    CASE activity
      idle
        write.string(screen, "Idle")
      editing
        write.string(screen, "Edit")
      single.step
        write.string(screen, "Step")
      free.running
        write.string(screen, "Busy")
      terminated
        write.string(screen, "Done")
:
```

Process 13.15

```
PROC move.cursor(CHAN OF BYTE screen, VAL INT x, y)
  -- move to column x of line y (of an ANSI terminal screen)
  CHAN OF DATA.ITEM c :
  PAR
    write.formatted(screen, "*#1B[%d;%dH", c)
    SEQ
      c ! data.int; x+1
      c ! data.int; y+1
:
```

Process 13.16

```
PROC clear.screen(CHAN OF BYTE screen)
  -- clear the whole of the screen (of an ANSI terminal)
  write.string(screen, "*#1B[2J")
:
```

spare room on the screen to the right of the image of the board, the activity of the controller can be displayed there (Process 13.14).

The terminal-specific declarations are just those of the procedures invoked to alter the screen. Process 13.15 and Process 13.16 are suitable definitions for an ANSI terminal.

All that remains is to supply an editor. Process 13.17 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. Editing continues until the 'Q' key 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. This means that although ANSI cursor keys generate a control sequence – for example the down-arrow key sends escape and open bracket (or an equivalent eight-bit control character), and 'B' – it is sufficient to interpret only the terminating character ('A' to 'D').

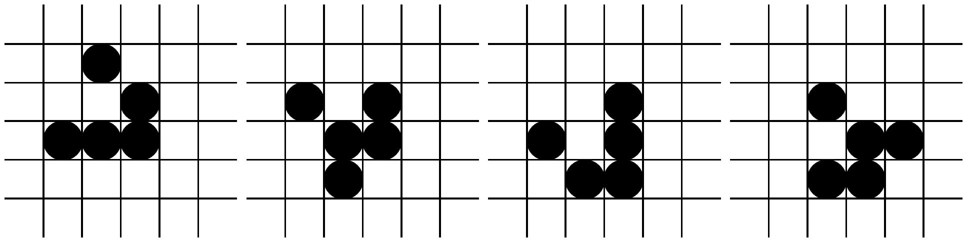
Process 13.17

```
PROC editor(CHAN OF BYTE keyboard, screen,
            [][]CHAN OF COMMAND control   )
  INT x, y :
  BOOL editing :
  SEQ
    -- initialize co-ordinates to centre of board
    x, y := array.width / 2, array.height / 2
    editing := TRUE
    WHILE editing
      BYTE char :
      SEQ
        move.cursor(screen, x, y)
        keyboard ? char
        CASE char
          'A'        -- move up, if possible
            y := max(y - 1, 0)
          'B'        -- move down, if possible
            y := min(y + 1, array.height - 1)
          'C'        -- move right, if possible
            x := min(x + 1, array.width - 1)
          'D'        -- move left, if possible
            x := max(x - 1, 0)
          '*s', '**'
            VAL BOOL state IS (char = '**') = alive :
            PAR
              control[x][y] ! set.state; state
              display.state(screen, x, y, state)
          'q', 'Q'
            editing := FALSE
          ELSE
            SKIP     -- ignore anything else
:
```

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[16, 17]. 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 (Figure 13.3).

Figure 13.3 The glider



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 the 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 off an unending stream of gliders; 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 infection.