Conway's game of life (http://en.wikipedia.org/wiki/Conway's_Game_of_Life) is a classic example of a cellular automaton. This zero-player game involves a universe on an orthogonal grid of cells, each cell can be dead or alive. Cells interact with all their eight neighbours. To evolve forward in time, the following rules are applies:

1. Any living cell with fewer than two live neighbours dies from boredom (well under-population if you go by Conway).
2. Any living cell two or three live neighbours lives on to the next generation.
3. Any living cell with more than three neighbour dies from overcrowding.
4. Any dead cell with exactly three living neighbours comes back to life by reproduction.

```fortran
program life

  implicit none
  integer, parameter :: ni=200, nj=200, nsteps = 500
  integer :: i, j, n, im, ip, jm, jp, nsum, isum
  integer, allocatable, dimension(:,:) :: old, new
  real :: arand

  ! allocate arrays, including room for ghost cells
  allocate(old(0:ni+1,0:nj+1), new(0:ni+1,0:nj+1))

  ! initialize elements of old to 0 or 1
  do j = 1, nj
    do i = 1, ni
      call random_number(arand)
      old(i,j) = nint(arand)
    endo, j
  enddo

  ! iterate
  time_iteration: do n = 1, nsteps
    ! corner boundary conditions
    old(0,0) = old(ni,nj)
    old(0,nj+1) = old(ni,1)
    old(ni+1,nj+1) = old(1,1)
    old(ni+1,0) = old(1,nj)
    ! left-right boundary conditions
    old(1:ni,0) = old(1:ni,nj)
  endtime_iteration
```

---

The Game of Life

```
!----------------------
!  Conway Game of Life
!    serial version
!----------------------

program life

  implicit none
  integer, parameter :: ni=200, nj=200, nsteps = 500
  integer :: i, j, n, im, ip, jm, jp, nsum, isum
  integer, allocatable, dimension(:,:) :: old, new
  real :: arand

  ! allocate arrays, including room for ghost cells
  allocate(old(0:ni+1,0:nj+1), new(0:ni+1,0:nj+1))

  ! initialize elements of old to 0 or 1
  do j = 1, nj
    do i = 1, ni
      call random_number(arand)
      old(i,j) = nint(arand)
    endo, j
  enddo

  ! iterate
  time_iteration: do n = 1, nsteps
    ! corner boundary conditions
    old(0,0) = old(ni,nj)
    old(0,nj+1) = old(ni,1)
    old(ni+1,nj+1) = old(1,1)
    old(ni+1,0) = old(1,nj)
    ! left-right boundary conditions
    old(1:ni,0) = old(1:ni,nj)
  endtime_iteration
```
old(1:ni,nj+1) = old(1:ni,1)

! top-bottom boundary conditions

old(0,1:nj) = old(ni,1:nj)
old(ni+1,1:nj) = old(1,1:nj)

do j = 1, nj
    do i = 1, ni

        im = i - 1
        ip = i + 1
        jm = j - 1
        jp = j + 1
        nsum = old(im,jp) + old(i,jp) + old(ip,jp) &
               + old(im,j )             + old(ip,j ) &
               + old(im,jm) + old(i,jm) + old(ip,jm)

        select case (nsum)
        case (3)
            new(i,j) = 1
        case (2)
            new(i,j) = old(i,j)
        case default
            new(i,j) = 0
        end select

    enddo
endo
endo

! copy new state into old state

old = new
rewind(1)
!
!
do i = 1, ni
    do j = 1, nj
        if (new(i,j)>0) write(1,"(3i4)") i,j,new(i,j)
    enddo
endo
endo!
endo time_iteration

! Iterations are done; sum the number of live cells
!
isum = sum(new(1:ni,1:nj))
!
! Print final number of live cells.

write(*,"(/'Number of live cells = ', i6/)") isum
deallocate(old, new)

end program life

Exercise:

1. Read the code and make sure you understand roughly what it does.
2. Compile and run
3. Visualize this map with gnuplot using script live4.gpl (https://wiki.ucl.ac.uk/display/GradPrgCrs/gnuplot+scripts):

   gnuplot -persist live4.gpl

Now go to this website:


and find some good patterns. Code them into your world by hand in a text editor and load them. See if they work as they should. Gosper glider gun is especially nice to try.