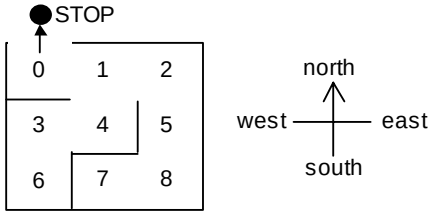
Tutorial 6

6.1 The figure below depicts a maze. Write a description of the maze in FSP which, using deadlock analysis, finds the shortest path out of the maze starting at any square.



(Hint: At each numbered square in the maze, a directional action can be used to indicate an allowed path to another square.)

6.2 One solution to the Dining Philosophers problem permits only four philosophers to sit down at the table at the same time. Specify a `BUTLER` process which, by counting, when composed with the model of section 6.2 permits a maximum of four philosophers to engage in the `sitdown` action before an `arise` action occurs. Show that this system is deadlock-free.

6.3 Using the Java timed wait primitive:

```
public final void wait(long timeout)
                          throws InterruptedException
```

modify the `Fork` monitor such that after a wait of 1 second, the call to `get` times out and returns the result false. The `Philosopher` should release the other fork if it holds it and try again if the timeout occurs. Observe the behaviour of the resulting system.

6.4 It is possible for the following system to deadlock. Explain how this deadlock occurs and relate it to one of the four necessary and sufficient conditions for deadlock to occur.

```
Alice = (call.bob -> wait.chris -> Alice).
Bob   = (call.chris -> wait.alice -> Bob).
Chris = (call.alice -> wait.bob -> Chris).
||S   = (Alice || Bob || Chris) /{call/wait}.
```

The following model attempts to fix the problem by allowing Alice, Bob and Chris to timeout from a call attempt. Is a deadlock still possible? If so, describe how the deadlock can occur and give an execution trace leading to the deadlock.

```
Alice = (call.bob -> wait.chris -> Alice
        | timeout.alice -> wait.chris ->Alice).
Bob   = (call.chris -> wait.alice -> Bob
        | timeout.bob -> wait.alice ->Bob).
Chris = (call.alice -> wait.bob -> Chris
        | timeout.chris -> wait.bob ->Chris).

||S = (Alice || Bob || Chris) /{call/wait}.
```