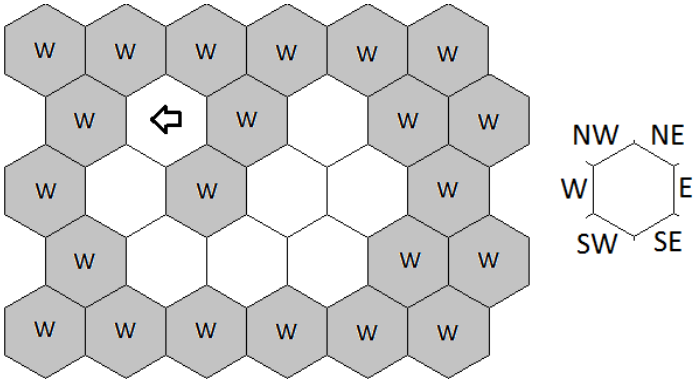
Maze Robot Simulation

Consider that you are the programmer of a robot simulator. The environment of the robot is a hexagonal grid. Each grid field can be a wall or an open space with optional number of marks. The robot stands on exactly one field in the maze which is not a wall. The state of the robot is defined by the maze, by the robot's coordinates in the maze, and by the robot's orientation in the maze. The orientation of the robot can be: East, West, NorthEast, SouthEast, NorthWest and SouthWest.



The robot can do following actions.

Step
: The robot moves exactly by one field in the direction of its orientation. This action can be done only if there is no wall in front of the robot. It is always guaranteed by the maze structure, that there is no possibility to get the robot outside the maze by this action.

Turn left
: The robot turns left. This action changes robot's orientation (from NW to W, from W to SW, from SW to SE, from SE to E, from E to NE and from NE to NW).

Put mark
: The robot puts the mark on the field where it is staying.

Get mark
: The robot gets the mark from the field where it is staying. This action can be done only if there is at least one mark on the field.

The programming language of the robot has several commands and forms:

The language has the following four commands: `step`, `turn-left`, `put-mark`, and `get-mark`. These commands correspond to the previously defined actions.

The language has two forms. The first form is sequence of commands or forms or procedure calls. This sequence is a simple Scheme list. If the sequence is an empty list, it means that the robot does nothing (something like NOP (No OPeration)). Any non-empty sequence is interpreted sequentially. The first element in that sequence is interpreted first and then the second element ... Every element can be: form or command or procedure call.

The second form is if condition: `(if <condition> <positive-branch> <negative-branch>)`. If the condition is true then the positive branch is interpreted. Otherwise the negative branch is interpreted. The condition can be: `west?`, `mark?`, or `wall?`.

- `west?` is true only if the current robot's orientation is west. Otherwise `west?` is false.
- `mark?` is true only if the field, where the robot is currently staying, has at least one mark. Otherwise `mark?` is false.
- `wall?` is true only if the action step is not possible for the current robot state. Otherwise `wall?` is false.

A program for the robot is a list of procedure definitions: `(procedure <name> <body>)`. Procedure definitions cannot be nested. The <name> means the name of the procedure. The <body> can be a form, a command or a procedure call. The procedure call is done by calling the name of the procedure in the same way as the command call. This can be done only if there is a definition of the procedure in the program. If the call of procedure is done then the interpretation continues on the next form, command or procedure call behind the last procedure call point. This behavior is similar to Scheme function call.

Your task is to write the simulator in Scheme (by R5RS Scheme standard) of the robot's program: `(simulate <state> <expr> <program> <limit>)` where

<state>
: is a state of the robot of the form: `(<maze> (<coordinate-x> <coordinate-y>) <orientation>)` where

<maze>
: is a list of lists with the same length. Each element can be `w` as wall or non-negative number as the number of marks on this field.

<coordinate-y>
: means the index (indexed from 0) of list in <maze> where the robot is staying.

<coordinate-x>
: means the index (indexed from 0) of element in list of <maze> where the robot is staying.

<orientation>
: can be: `west`, `northwest`, `northeast`, `east`, `southeast` and `southwest`.

<expr>
: can content form, command, or procedure call.

<program>
: can content list of procedure definitions.

<limit>
: is a number which means maximum nested procedure calls. If a new procedure call exceeds the limit then this call cannot be done.

The simulate function returns `(<action sequence> (<maze> (<coordinate-x> <coordinate-y>) <orientation>))` where <action sequence> is a list of actions that were applied during the simulation and `(<maze> (<coordinate-x> <coordinate-y>)) <orientation>` is the last state of the robot.

If some command or form or procedure call cannot be done then the simulator ends and returns the last legal state of the robot.

Example:

Consider the following part of Scheme program:

```
(define get-maze
'(
(w   w   w   w   w   w)
  (w   0   w   0   w   w)
(w   0   w   0   0   w)
  (w   0   0   0   w   w)
(w   w   w   w   w   w)
)
)
(define right-hand-rule-prg
  '(
    (procedure start
      ( turn-right
        (if wall?
           ( turn-left
             (if wall?
                 (turn-left
                     (if wall?
                        turn-left
                        step
                     )
                 )
                 step
              )
           )
           step
        )
        put-mark
        start
      )
    )
    (procedure turn-right (turn-left turn-left turn-left turn-left turn-
left))
  )
)
```

Now we call:

```
>  (simulate (list get-maze (list 1 1) 'west) 'start right-hand-rule-prg 3)
```

And we obtain (according to the task specification):

```
(
 (turn-left turn-left turn-left turn-left turn-left turn-left turn-left step
put-mark turn-left turn-left turn-left turn-left turn-left turn-left turn-
left step put-mark)
 (((w   w   w   w   w   w)
     (w   0   w   0   w   w)
   (w   1   w   0   0   w)
     (w   1   0   0   w   w)
   (w   w   w   w   w   w))
 (1 3) southeast))
```