Population Evaluator

This task is a second step on the path to making a system, which automatically creates programs for a robot in a maze via genetic programming. At the end, the programs will be created from example pairs of the state of the maze before and after the execution of the program. The best program is the shortest and quickest program that can transform any input maze to the desired output maze.

The basic law driving the evolution is the higher chance of survival of the best individuals. The individuals in our case are the programs for the robot simulator. Your second assignment is to implement a function, which will choose the best programs from a given population and sort them according to their quality.

Input

`(evaluate <prgs> <pairs> <threshold> <stack_size>)` where

`<prgs>`
: is a list of programs for the robot (as in Task 1). Each program includes a procedure named "start", which is the initial expression for the simulation;

`<pairs>`
: is a list of pairs of states, including the position and the orientation of the robot;

`<threshold>`
: is lower bounds on the quality of the program in order to appear in the output;

`<stack_size>`
: is the limit on the robot simulator stack size (see Task 1).

Output

A sorted list of pairs `((<value> <program>) ... (<value> <program>))` including only the programs, which made it under the threshold.

The value of a program has four components that are used in lexicographical order for the final ordering. The components in the decreasing order of importance are the following:

- **Manhattan distance** between the marks in the state produced by the program and the marks in the desired state of the maze. I.e. the sum of the absolute values of the differences between number of marks present in the output maze and the number of marks present at the same position in the desired maze. The distance between the marks on the following two mazes is 4.

  ```
  ((w w w w w)   ((w w w w w)
   (w 1 w 1 w)    (w 0 w 1 w)
   (w 3 0 0 w)    (w 1 0 1 w)
   (w w w w w))   (w w w w w))
  ```

- **Robot configuration distance** is the Manhattan distance between the robot final position and the desired final position, plus one, if the robot has wrong orientation.

  The distance of `(<maze> (3 2) north) and (<maze> (1 1) north)` is 3.

  The distance of `(<maze> (3 1) north) and (<maze> (1 3) south)` is 5.

- **The length of the program** is the number of commands, tests, procedure calls and procedure declarations in the whole program. We do not count the keywords "procedure" and "if" nor empty sequences. The length of the example "right-hand-rule-prg" from Task 1 is 17. It has 3 tests, 10 commands, 2 procedure calls and 2 procedure declarations.
- **Number of steps** (commands) needed by the program to produce the final configuration. It is the number of commands in the output of the simulation for the program. It is 12 in the example from Task 1.

The final value of a program is a list of four numbers, each representing the sum of the corresponding component above (the length of the program is of course computed only once) over all the input pairs of states.

The role of the threshold is to remove obviously wrong programs and to make the evaluation more efficient. As soon as it is clear that an individual will not make it to the output set, the evaluator should stop evaluating (e.g. simulating) the individual in order not to waste computational resources. The threshold for each of the fitness function components is applied separately. It means that a program which has any of the components of the fitness function higher than the threshold will not appear in the output. Note that the threshold is applied to the sum of the evaluations for all pair not to each evaluation separately.

Example:

Consider the following part of Scheme program:

```
(define prgs
'(
   (
      (procedure start
         (turn-right (if wall? (turn-left
             (if wall? (turn-left (if wall? turn-left step)) step)) step) step)
                put-mark start )
      )
      (procedure turn-right (turn-left turn-left turn-left))
  )
  (
      (procedure start  (put-mark (if wall? turn-left step) start))
  )
  (
      (procedure start (step step step put-mark))
  )
)
)

(define pairs
'(
  (
   (((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) south)

   (((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))
     (1 1) north)
   )
   (
   (((w w w w w w)
     (w 0 w 0 w w)
     (w 0 w 2 0 w)
     (w 1 3 0 w w)
     (w w w w w w))
     (3 3) north)

   (((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))
     (1 1) north)
  ))
 )
```

Now we call:

```
>  (evaluate prgs pairs '(20 20 20 20) 5)
```

And we obtain (according to the task specification):

```
(
((8 5 5 2) ((procedure start (step step step put-mark))))
((18 7 6 20) ((procedure start (put-mark (if wall? turn-left step) start))))
)
```

Example 2 (input,output), Example 3 (input,output), Example 4 (input,output), Example 5 (input,output), Example 6 (input,output), Example 7 (input,output),