# Automatic Code Synthesis

The third Scheme assignment is to complete the automatic code synthesis framework using the population evaluation function from Assignment 2 as your solution quality evaluator. The task is to write a function, which searches for the best program that performs a task given as pairs of the initial and the desired state of the maze. The ideal program would transform all the initial states to the desired states in the minimal number of steps.

## Input

`(evolve <pairs> <threshold> <stack_size>)` where

`<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 Assignment 2).

The inputs are exactly the same as in Assignment 2, with the exception of missing list of programs.

## Output

`(<value> <program>)` where

`<value>`
: is the value of the output program in the sense of Assignment 2;

`<program>`
: is the best program found for the given task;

The output is **NOT the regular output** of the function. Your function will be executed for a specific amount of time and then killed by the system. The output used by the system for evaluation will be the last line of the standard output produced by your program. Hence, you should always output the best found solution using display and then newline and flush-output function.

We do not put any hard restrictions on how your method works, but we suggest you to use genetic programming. In order to implement the genetic programming algorithm, you need to design a suitable initial population and implement the operations of selection, mutation and crossover.

**Selection** selects the individuals that will participate in creating the next generation. Generally, the better individuals should have better chance to be selected. On the other hand, even the worse should be selected with non-zero probability to keep the population diverse and avoid local minima.

**Mutation** is an operation on single individual of the genetic algorithm (program). It slightly randomly modifies the individual. For example, it can substitute one command/test for another, remove or add random command.

**Crossover** is an operation on two selected individuals, which produces two new individuals as their combination. The basic selection could choose a sub-tree in each program and swap them.

The genetic programming evolves programs using the following high level algorithm:

1. create an initial population
2. evaluate all individuals in the population
3. if desired quality of the solution is reached then stop
4. select individuals participating on the new population
5. use mutation, crossover, and copy to create a new population
6. goto 2

In order to be able to evolve any program, it is important to design the mutation and crossover operations in a way that allows any program to be created from the initial population with non-zero probability.

The R5RS norm does not specify any built-in random number generator, which is crucial in genetic programming. You can use the following implementation taken from MIT-Scheme.

```
(define (congruential-rng seed)
  (let ((a 16807 #|(expt 7 5)|#)
        (m 2147483647 #|(- (expt 2 31) 1)|#))
    (let ((m-1 (- m 1)))
      (let ((seed (+ (remainder seed m-1) 1)))
        (lambda (b)
          (let ((n (remainder (* a seed) m)))
            (set! seed n)
            (quotient (* (- n 1) b) m-1)))))))
(define random (congruential-rng 12345))
```

The function random expects one integer argument and returns a random number between 0 and the argument. Each consecutive call of the function returns next random number.

## Example:

Consider the following part of Scheme program:

```
(define s0
'((
(w w w)
(w 1 w)
(w w w)
)
(1 1) south
))

(define t0
'((
(w w w)
(w 1 w)
(w w w)
)
(1 1) south
))

(define s1
'((
(w w w)
(w 1 w)
(w 0 w)
(w w w)
)
(1 1) west
))

(define t1
'((
(w w w)
(w 1 w)
(w 0 w)
(w w w)
)
(1 1) west
))

(define s2
'((
(w w w)
(w 0 w)
(w 0 w)
(w 0 w)
(w 1 w)
(w 0 w)
(w 0 w)
(w w w)
)
(1 1) south
))

(define t2
'((
(w w w)
(w 0 w)
(w 0 w)
(w 0 w)
(w 0 w)
(w 1 w)
(w 0 w)
(w 0 w)
(w w w)
)
(1 7) south
))
```

Now we call:

```
(evolve `((,s0 ,t0) (,s1 ,t1) (,s2 ,t2)) `(10000 10000 20 20) 10)
```

And we can obtain during some time period (according some algorithm):

```
((0 6 1 0) ((procedure start ())))
((0 6 1 0) ((procedure 1 ())))
((0 6 1 0) ((procedure start ())))
((0 3 8 3) ((procedure 1 ()) (procedure start ((if mark? () (1)) 1 step step step))))
((0 0 13 14) ((procedure start ((if mark? () (1)) 1 step step step)) (procedure 1 (put-mark get-mark step step step))))
((0 0 11 14) ((procedure 1 (put-mark get-mark step step step)) (procedure start ((if mark? () (1)) 1 step))))
((0 0 5 6) ((procedure 1 step) (procedure start (step start))))
((0 0 3 6) ((procedure start (step start))))
```

The last line of the output above has the final importance. If the first two numbers are zeros then the problem was successfuly solved.

### Example 1 (in, out), Example 2 (in, out), Example 3 (in, out), Example 4 (in, out)