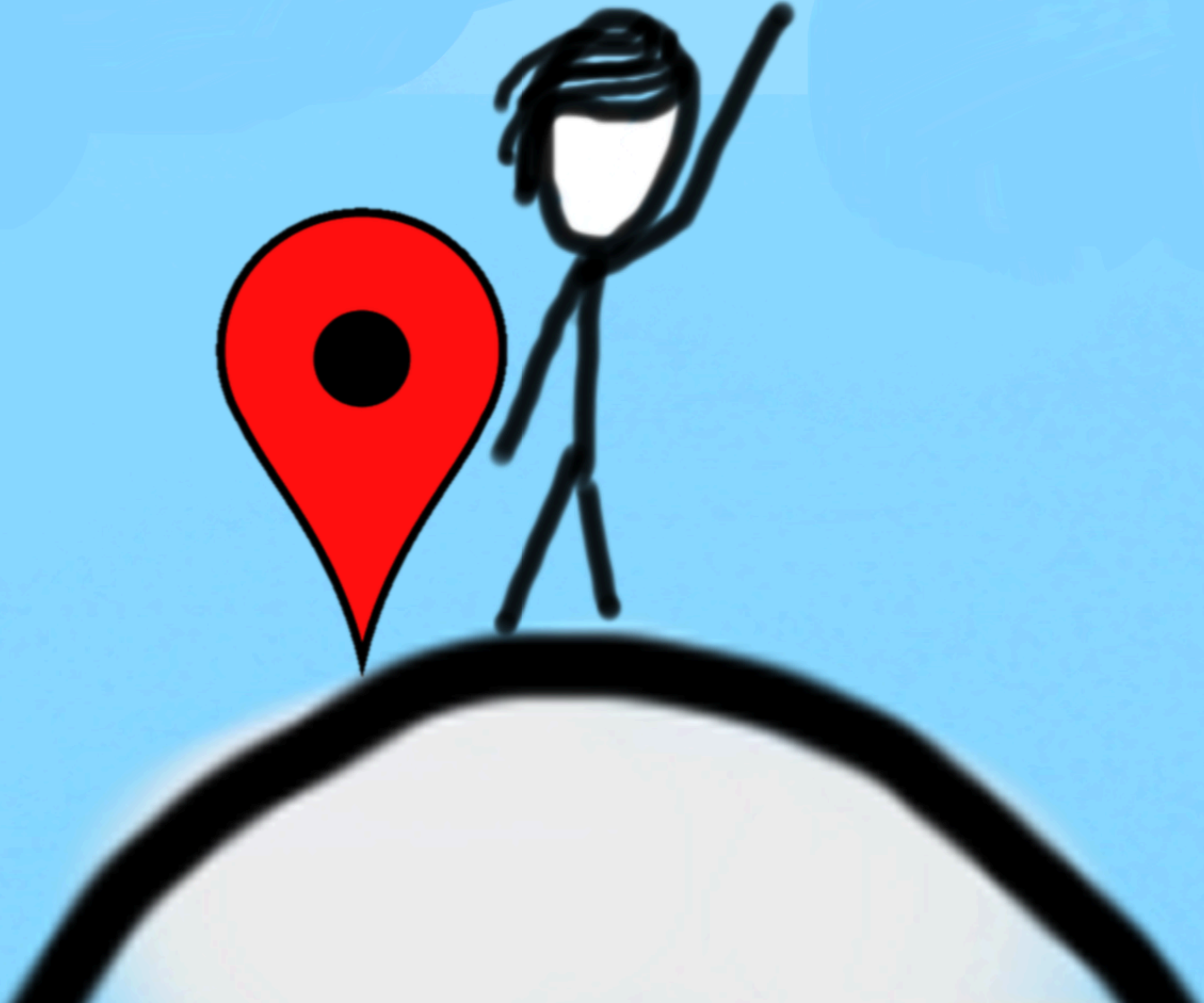
Decomposition

Chris Gregg

CS106A, Stanford University

Today’s Goal:

- Be able to approach a problem “top down” by using *decomposition* (also known as *top down refinement*)

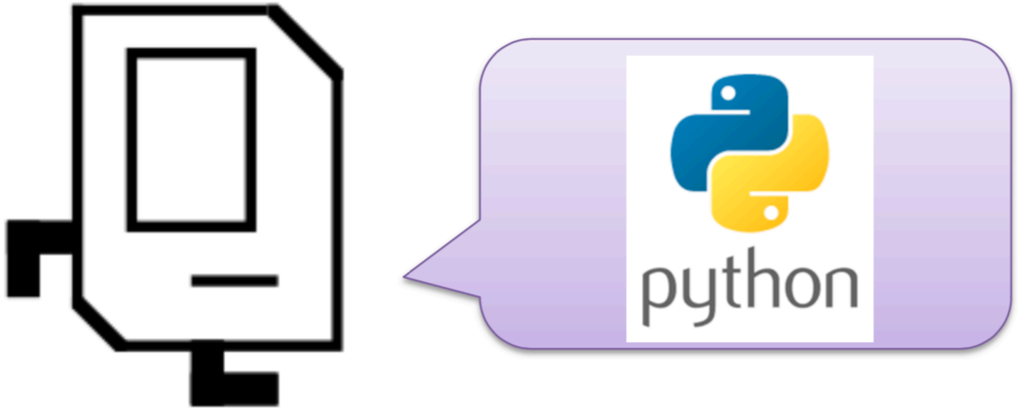


Today's Plan:

- Decomposition
- double_beepers()
- Infinite loops (oops!)
- roomba_karel()

Quick Review

- Karel the Robot:



- Functions:

```
def main():
    goToMoon()

def go_to_moon():
    build_spaceship() # a few more steps

def build_spaceship():
    # todo
    put_beeper()
```

Quick Review

- For loops:

```
def main():
    # repeats the body 99 times
    for i in range(99):
        # the “body”
        put_beeper()
```

- While loops:

```
def main():
    # while condition holds runs body
    # checks condition after body completes
    while front_is_clear():
        move()
```

Quick Review

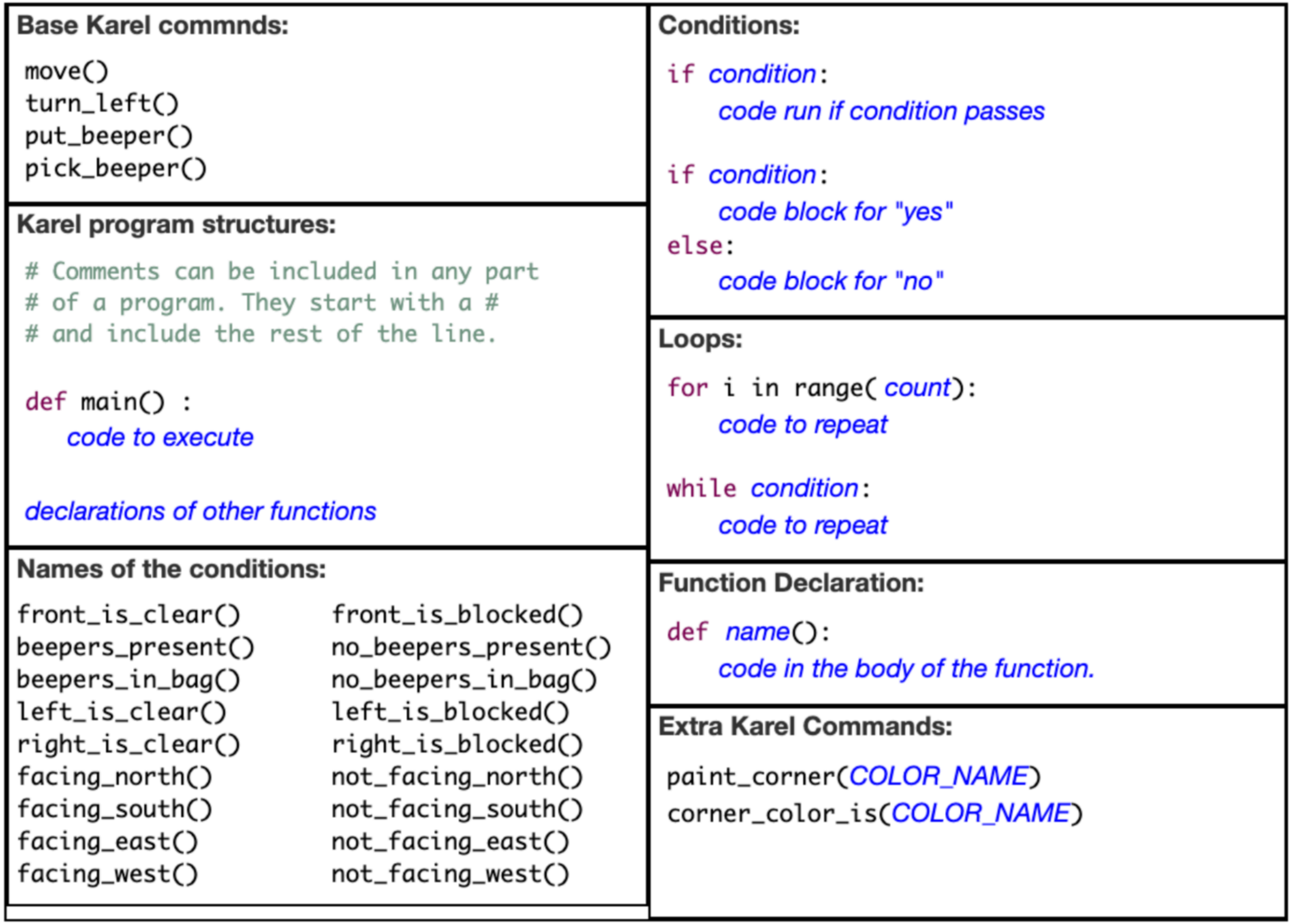
- If statement:

```
def main():
    # If the condition holds, runs body
    if front_is_clear():
        move()
```

- If / Else statement:

```
def main():
    # If the condition holds,
    if beepers_present():
        # do this
        pick_beeper()
    else :
        # otherwise, do this
        put_beeper()
```

Karel Reference



Base Karel commnds:

```
move()
turn_left()
put_beeper()
pick_beeper()
```

Karel program structures:

```
# Comments can be included in any part
# of a program. They start with a #
# and include the rest of the line.

def main() :
    code to execute

declarations of other functions
```

Names of the conditions:

```
front_is_clear()        front_is_blocked()
beepers_present()       no_beepers_present()
beepers_in_bag()        no_beepers_in_bag()
left_is_clear()         left_is_blocked()
right_is_clear()        right_is_blocked()
facing_north()          not_facing_north()
facing_south()          not_facing_south()
facing_east()           not_facing_east()
facing_west()           not_facing_west()
```

Conditions:

```
if condition:
    code run if condition passes

if condition:
    code block for "yes"
else:
    code block for "no"
```

Loops:

```
for i in range( count):
    code to repeat

while condition:
    code to repeat
```

Function Declaration:

```
def name():
    code in the body of the function.
```

Extra Karel Commands:

```
paint_corner(COLOR_NAME)
corner_color_is(COLOR_NAME)
```

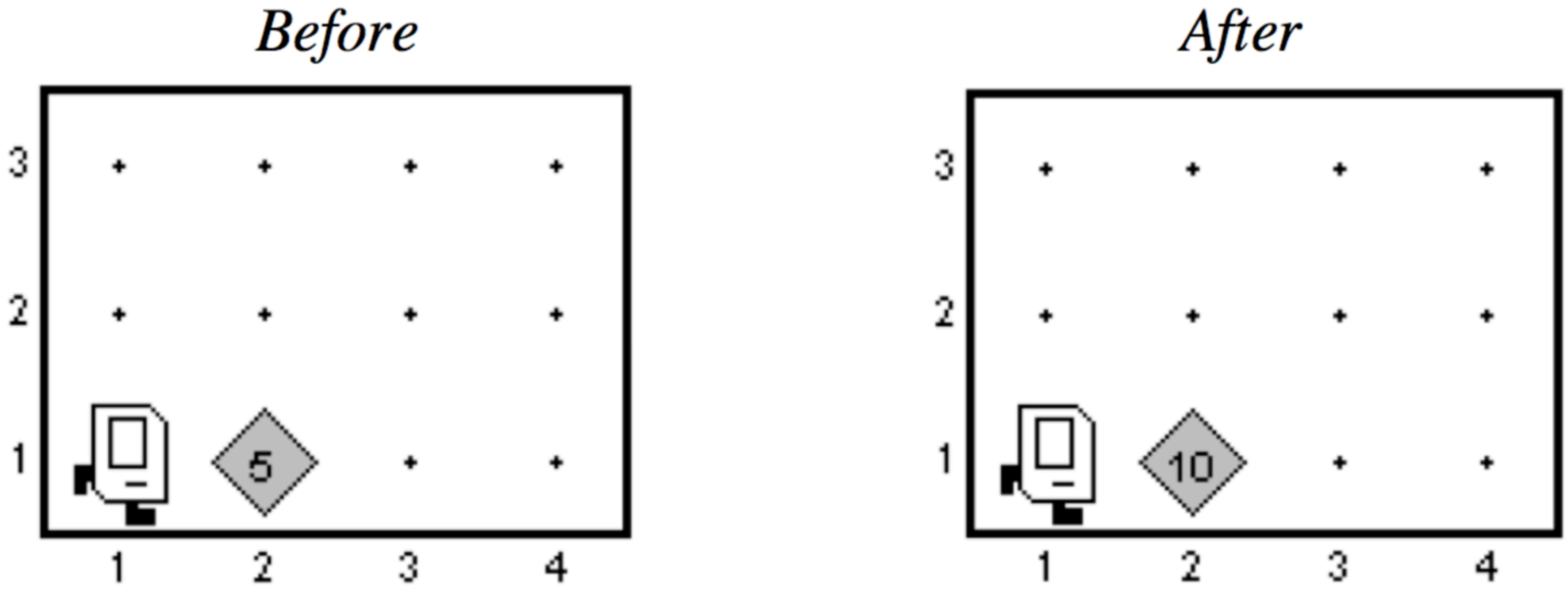
What is Beethoven doing now?

Decomposing.

- In programming, *decomposition* is the art of breaking a problem down into manageable parts that are clear, understandable, and easy to debug and maintain. Another term for decomposition is *factoring*.
- Instead of a big, monolithic program, a well-decomposed program has small functions and easily understood parts.
  - Each function should have one purpose, or be made up of smaller functions that each have a single purpose
- Each function within a larger function should be able to stand on its own
  - This makes debugging easier, and it means that we can debug functions separately

Example: What is your morning routine?

More Karel: `double_beepers()`



How can we go from *Before* to *After* with Karel?

This is not trivial!

Muhammed ibn Musa Al Kwarizmi

Our algorithm must work for *any* number of starting beepers!

http://web.stanford.edu/class/cs106a/apps/karelide/#/double

What does this program do?

```python
from karel.stanfordkarel import *

def main():
    move()
    while beepers_present():
        pick_beeper()
        move()
        put_beeper()
        put_beeper()
        turn_around()
        move()
        turn_around()

    move()
    while beepers_present():
        pick_beeper()
        turn_around()
        move()
        turn_around()
        put_beeper()
        move()

    turn_around()
    move()
    turn_around()
    turn_around()
    move()
    turn_around()

def turn_around():
    turn_left()
    turn_left()

if __name__ == "__main__":
    run_karel_program()
```

Full `double_beepers()` program (next slide has utility functions):

```
from karel.stanfordkarel import *

# File: double.py
# ----------------------------
# Practice decomposition and stepwise
# refinement
def main():
   """
   Big idea: make a pile with double beepers next to the first pile
   and then move that pile back to the original beepers location
   """
   move()
   # step 1: Make a double pile next to the first
   make_double_pile_nextdoor()

   # step 2: Move the pile back to the original location
   move_pile_backwards()

   # OBO: move karel back one spot
   move_backwards()
```

```

# pre-condition: Karel is on top of a pile of beepers
# post-condition: Karel is in front of a pile of
beepers with twice the
#                 original amount that is next to the
original spot.
#                 No more beepers are on the original
location
def make_double_pile_nextdoor():
   while beepers_present():
      pick_beeper()
      move()
      put_beeper()
      put_beeper()
      move_backwards()

# pre-condition: Karel is in front of a pile of
beepers
# post-condition: Karel has moved the pile backwards
and is
#                 on top of it
def move_pile_backwards():
   move()
   while beepers_present():
      pick_beeper()
      move_backwards()
      put_beeper()
      move()
   move_backwards()
```

Full `double_beepers()` program (next slide has utility functions):

```
# ---------------------------- #
#     Utility functions        #
# ---------------------------- #

# a classic...
def move_backwards():
   turn_around()
   move()
   turn_around()

# another classic...
def turn_around():
   turn_left()
   turn_left()

# rememeber lecture 1? fond memories...
def turn_right():
   for i in range(3):
      turn_left()
```

What does this program do?

This is DoubleBeepers!

- It’s harder to understand because it hasn’t been decomposed.
- It would be infinitely easier to modify the DoubleBeepers we worked on instead of this one, because the decomposed version is that much more clear.

```
from karel.stanfordkarel import *

def main():
    move()
    while beepers_present():
        pick_beeper()
        move()
        put_beeper()
        put_beeper()
        turn_around()
        move()
        turn_around()

    move()
    while beepers_present():
        pick_beeper()
        turn_around()
        move()
        turn_around()
        put_beeper()
        move()

    turn_around()
    move()
    turn_around()
    turn_around()
    move()
    turn_around()

def turn_around():
    turn_left()
    turn_left()

if __name__ == "__main__":
    run_karel_program()
```

Pro Tips

- A good function should do “one conceptual thing.”
- All functions and variables should be descriptive enough so that someone reading your code can have a good idea about what it does simply from the name.
- Good functions should be less than ten lines and no more than three levels of indentation.
- Functions should be reusable (within reason) and easy to modify.
- Functions should be well commented, but not over-commented.

There are two types of programs:
One is so complex that there is nothing obvious wrong with it.
One is so clear that there is obviously nothing wrong with it.

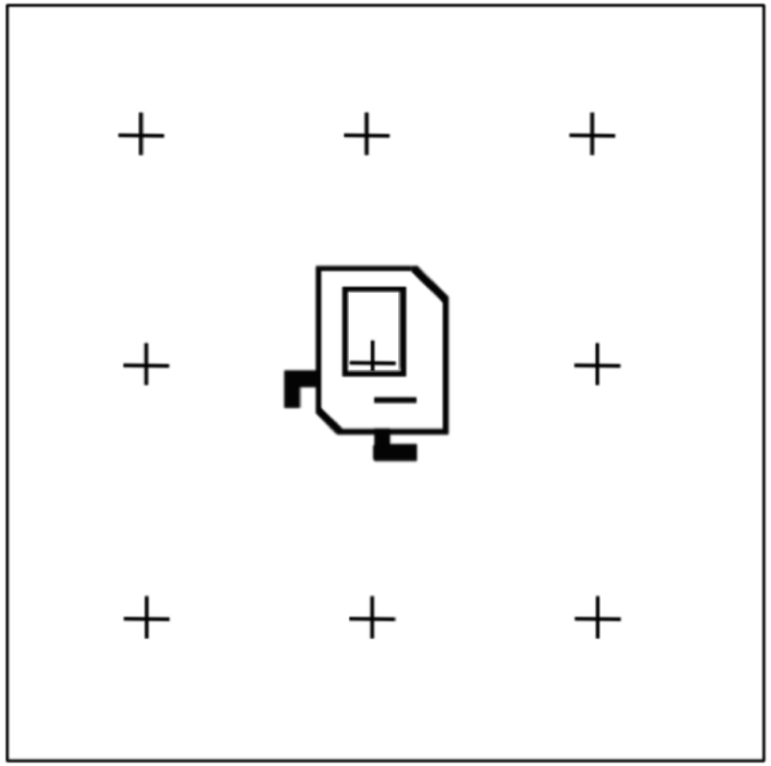
Infinite loops (oops!)

Why did the computer scientist die in the shower?

The bottle of shampoo said, *Lather, rinse, repeat*.

```
def turn_to_wall():
    while left_is_clear():
        turn_left()
```

What happens in the program when Karel is in this state?

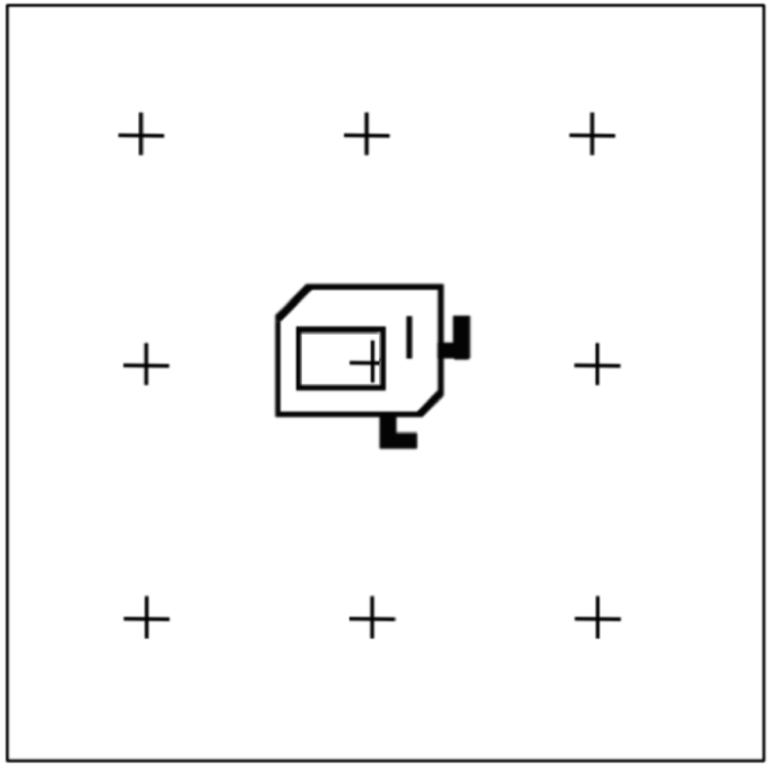


Infinite loops (oops!)

Why did the computer scientist die in the shower?

The bottle of shampoo said, *Lather, rinse, repeat*.

```
def turn_to_wall():
    while left_is_clear():
        turn_left()
```



What happens in the program when Karel is in this state?

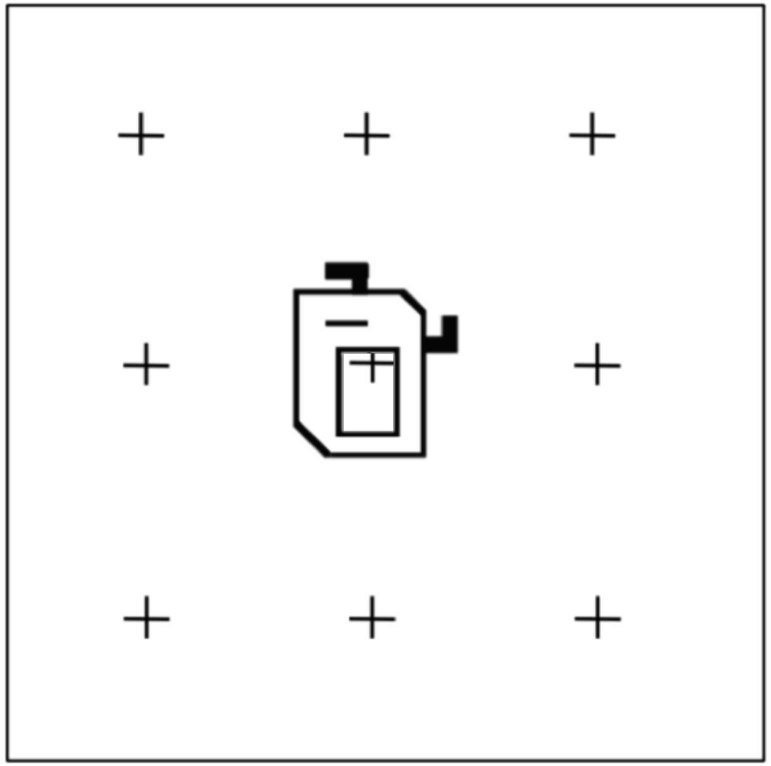
Infinite loops (oops!)

Why did the computer scientist die in the shower?

The bottle of shampoo said, *Lather, rinse, repeat*.

```
def turn_to_wall():
    while left_is_clear():
        turn_left()
```

What happens in the program when Karel is in this state?

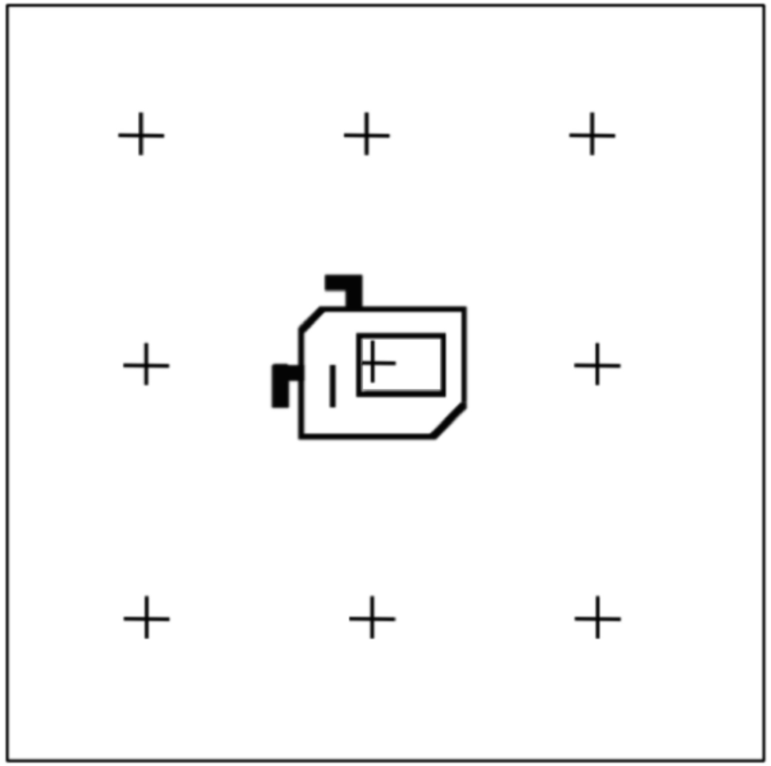


Infinite loops (oops!)

Why did the computer scientist die in the shower?

The bottle of shampoo said, *Lather, rinse, repeat*.

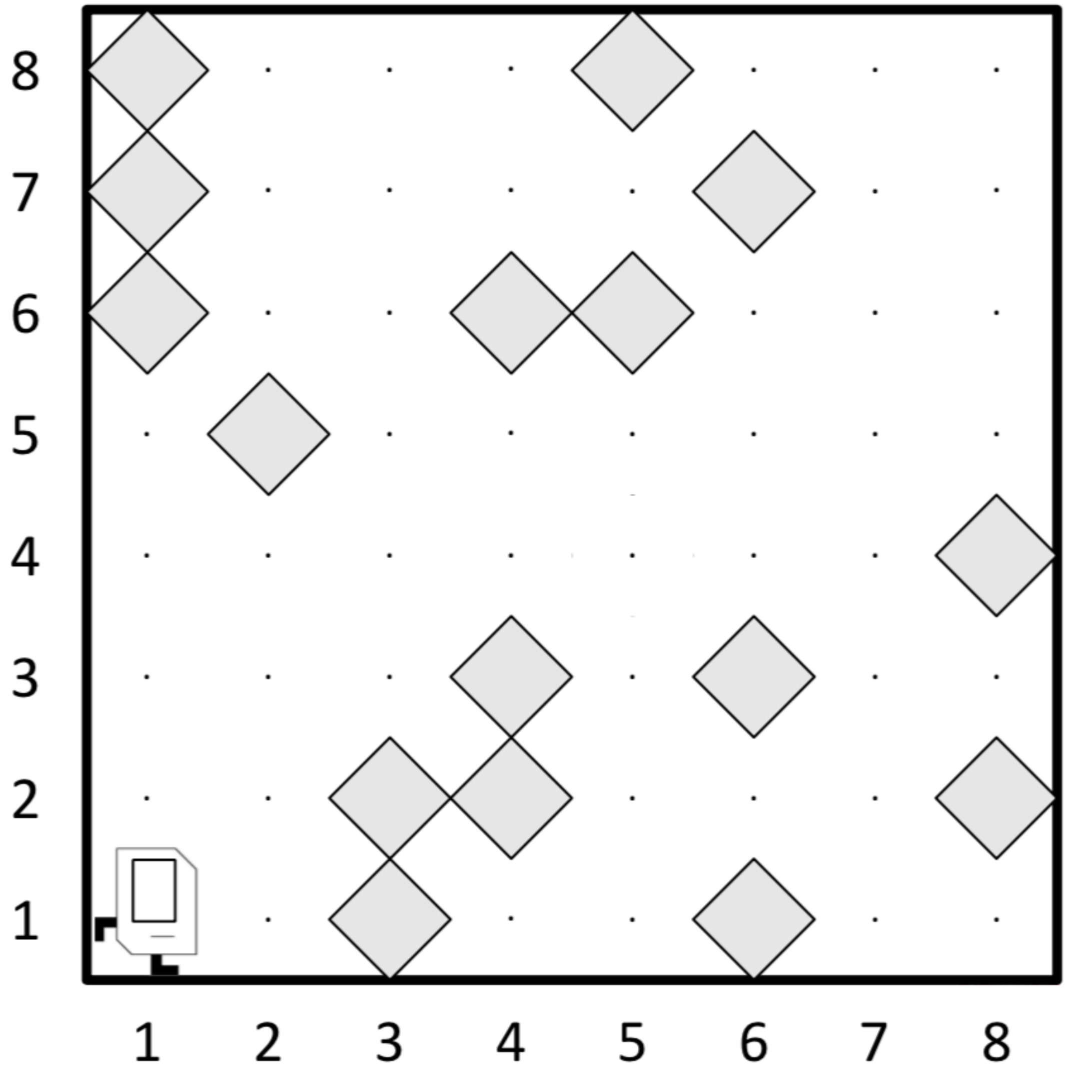
```
def turn_to_wall():
    while left_is_clear():
        turn_left()
```



What happens in the program when Karel is in this state?

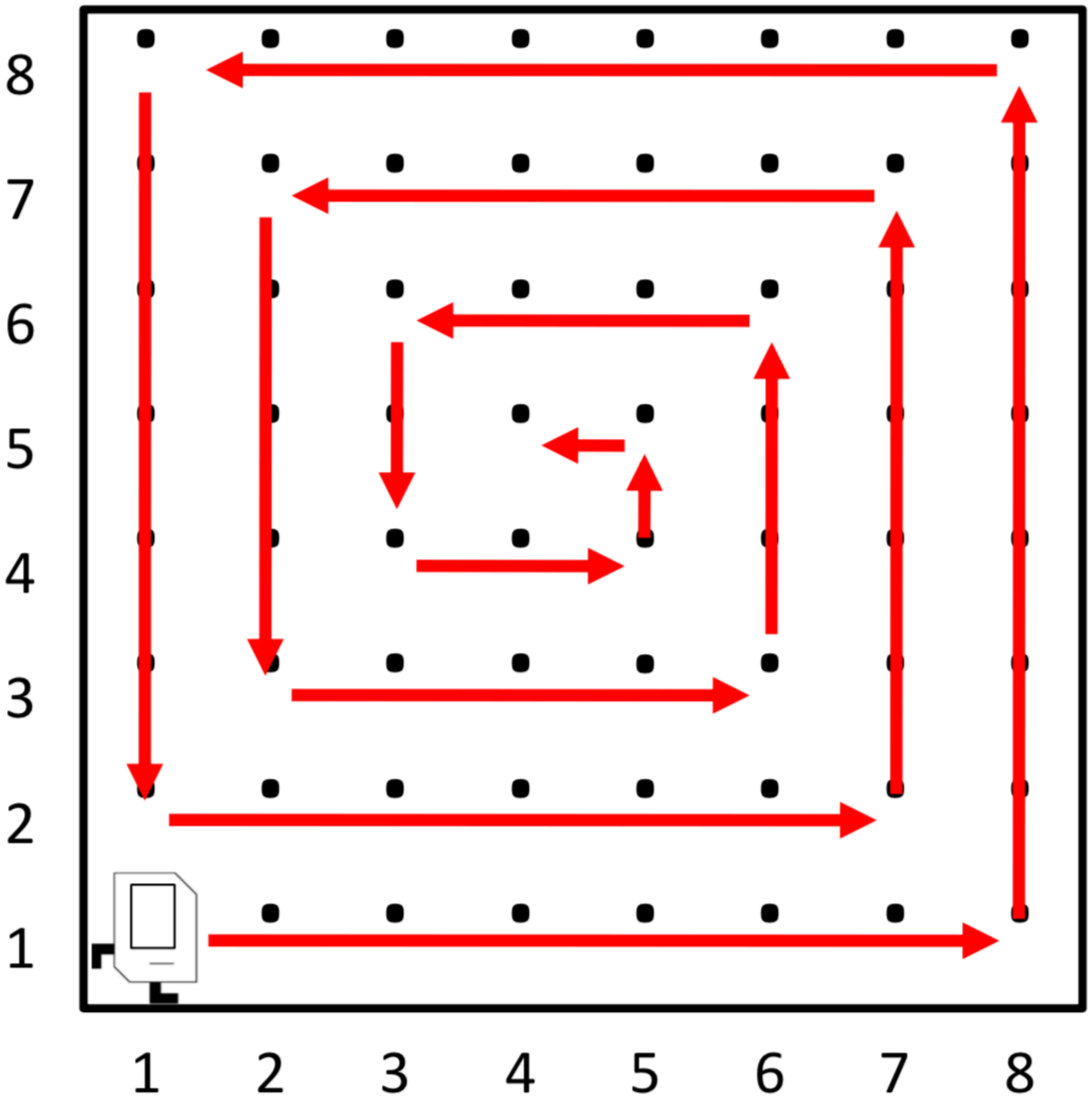
roomba_karel

- Write a Roomba Karel that sweeps the entire world of all beepers.
  - Karel starts at (1,1) facing East.
  - The world is rectangular, and some squares contain beepers.
  - There are no interior walls.
  - When the program is done, the world should contain 0 beepers.
  - Karel's ending location does not matter.
- How should we approach this tricky problem?



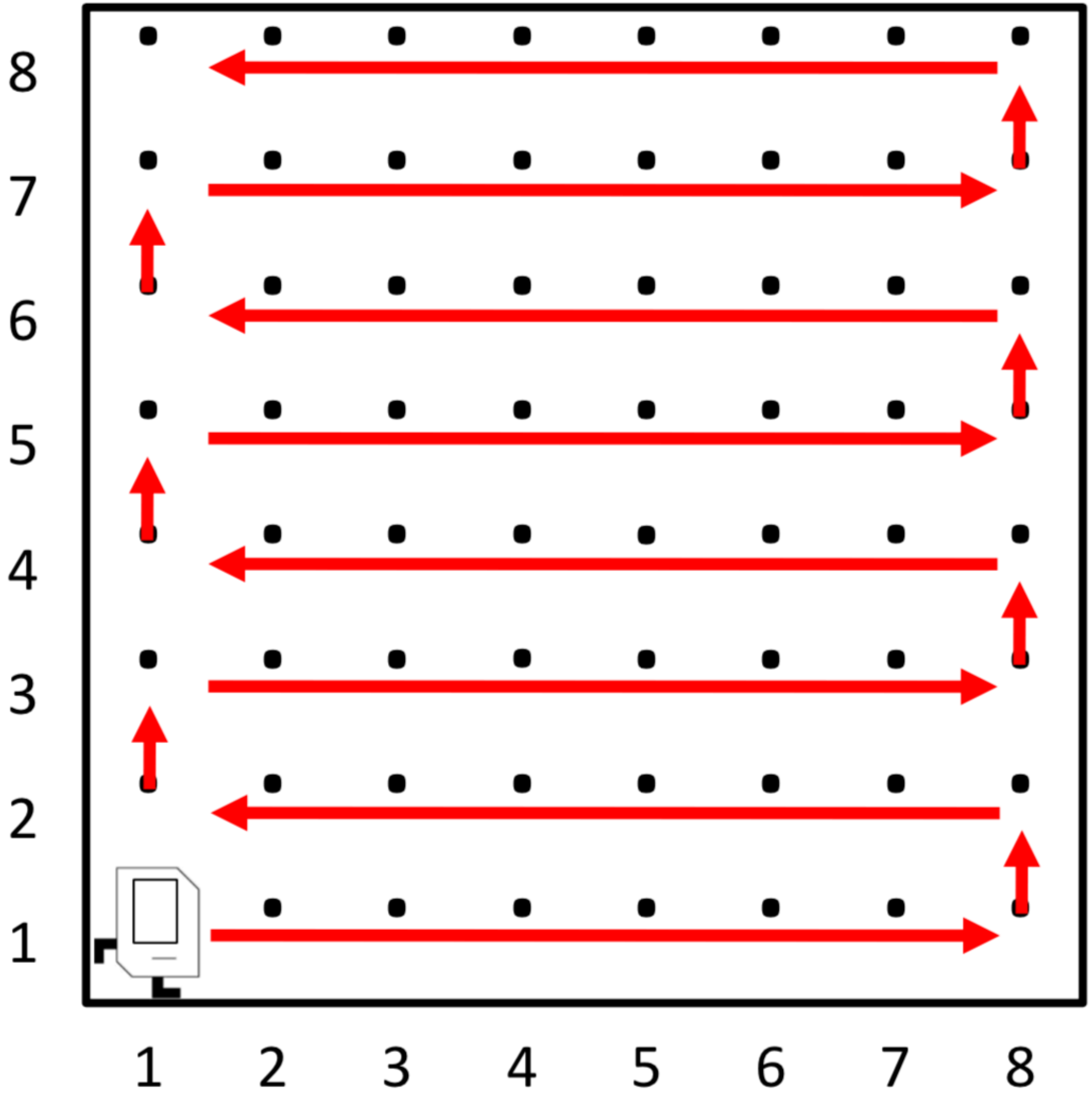
`roomba_karel`

Possible algorithm 1



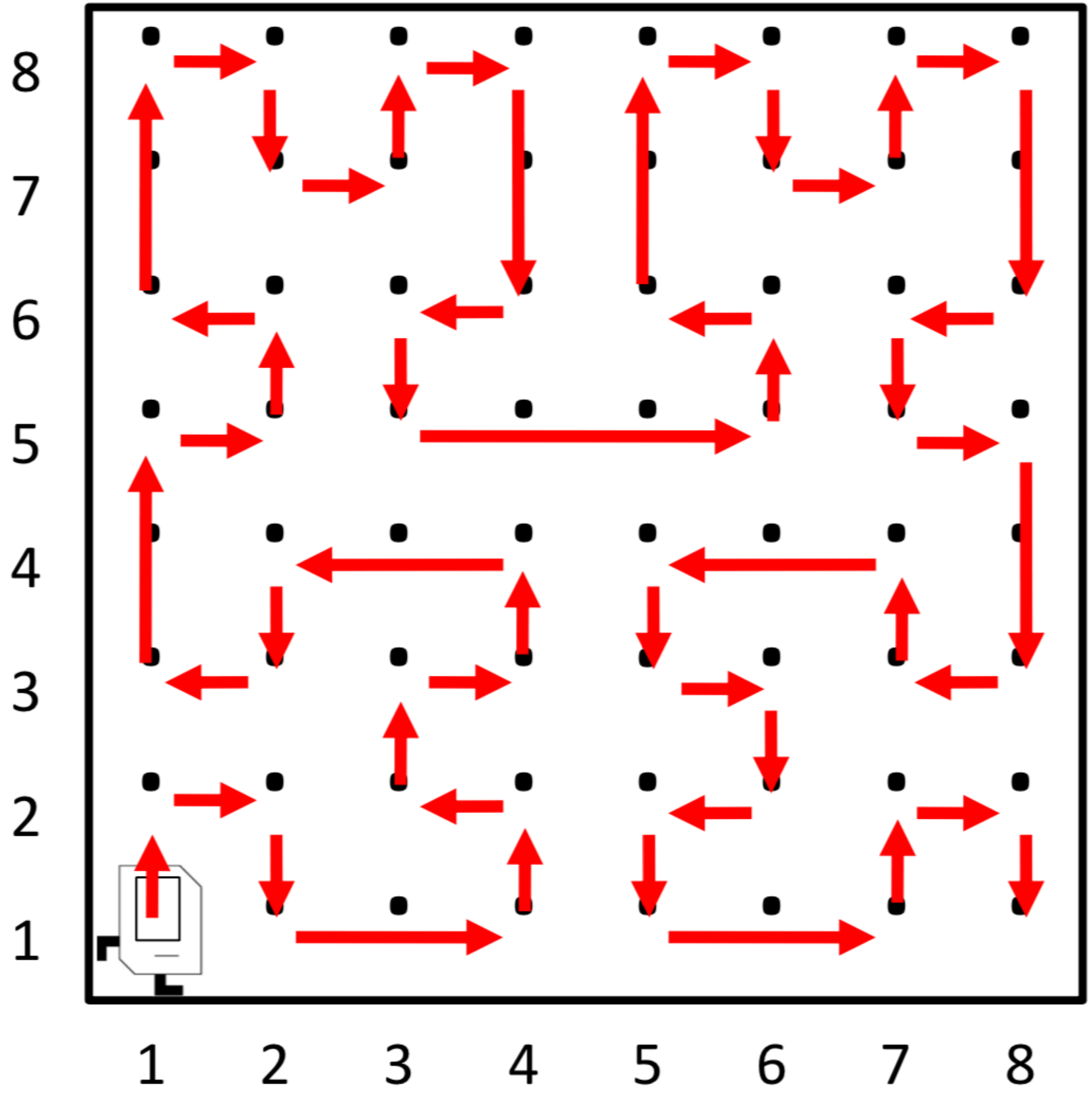
roomba_karel

Possible algorithm 2



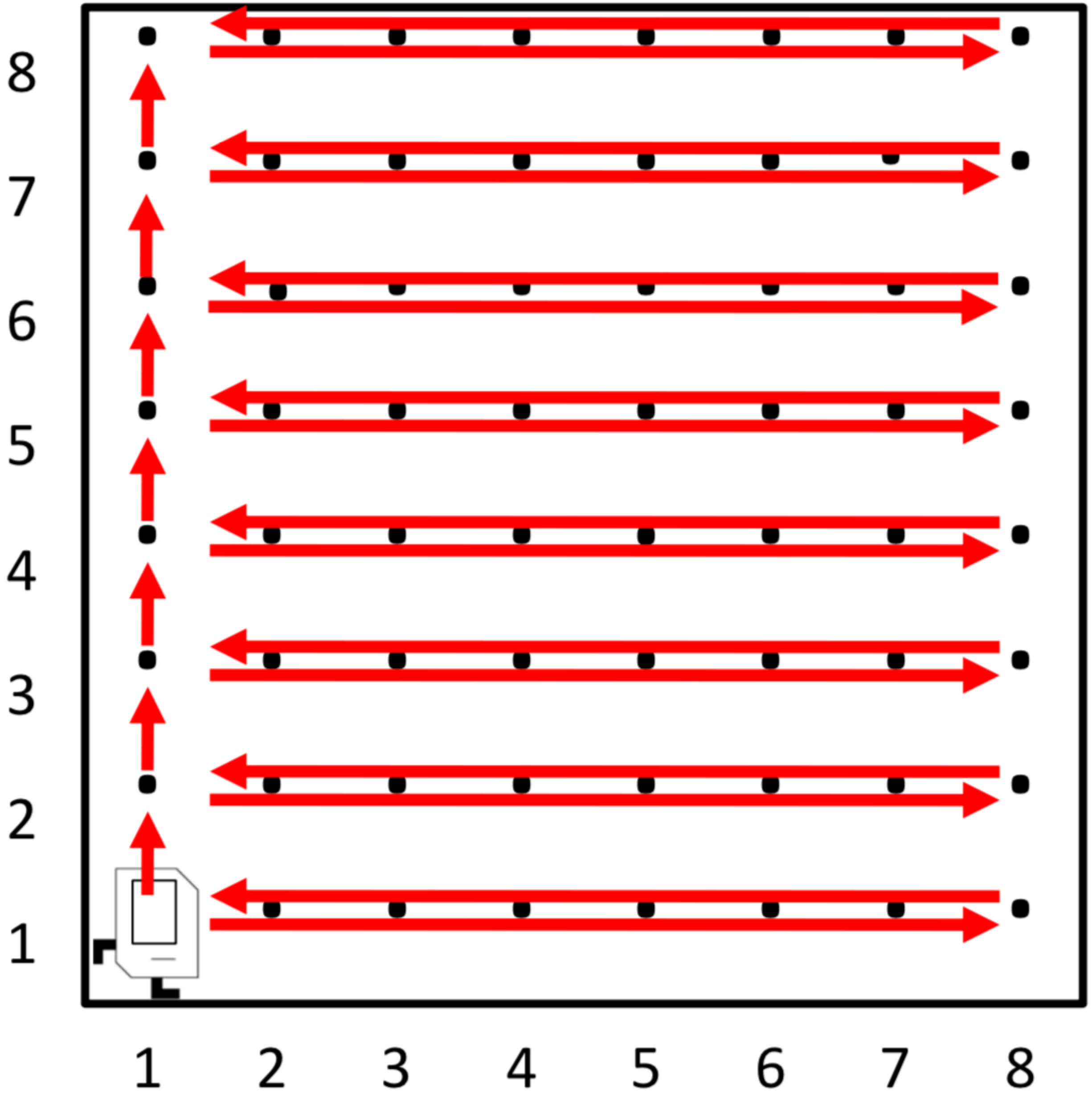
`roomba_karel`

Possible algorithm 3

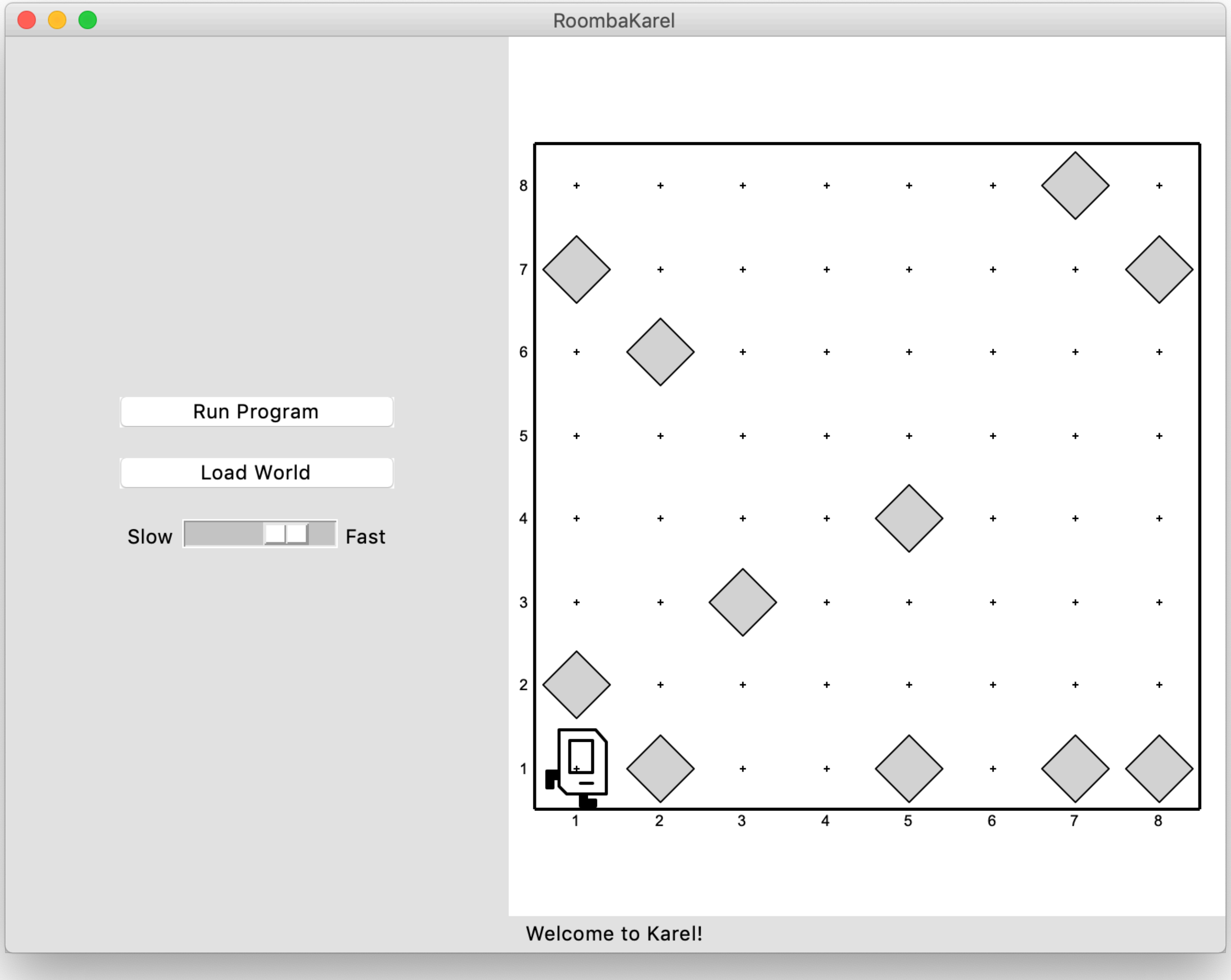


roomba_karel

Possible algorithm 4



roomba_karel



Welcome to Karel!