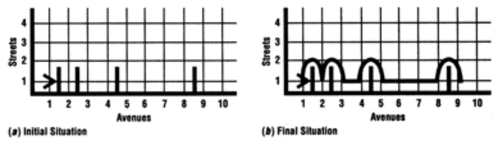
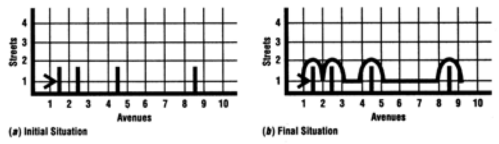
## Mile-long hurdle race

**Suppose that we want to program Reeborg to run a one-mile long hurdle race, where vertical wall sections represent hurdles. The hurdles are only one block high and are randomly placed between any two corners in the race course. One possible race course is shown below.**

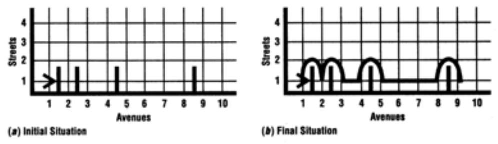


## Strategy?

**Reeborg *could* run this race by jumping up between every pair of corners. Is this an appropriate strategy?**

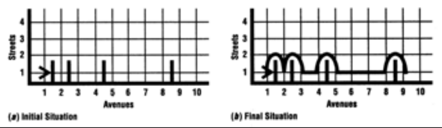


**Jumping up between corners, whether a hurdle was between them or not, would slow Reeborg down. Instead, program Reeborg to move straight ahead when it can, and to jump over hurdles only when it must. The program could then consist of 8 `advance_a_corner()` instructions. The definition of `advance_a_corner()` can be written using stepwise refinement as follows:**



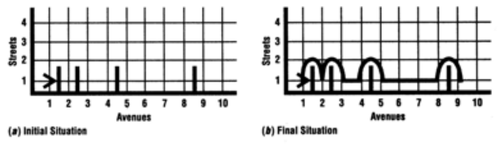
```
def advance_a_corner():
    if front_is_clear():
        move()
    else
        jump_hurdle()
```

**Notice we have used a new command `jump_hurdle()`that must now be defined.**



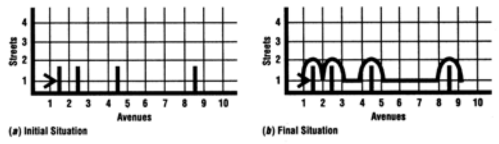
**We continue our refinement by defining `jump_hurdle()` to be**

```
def jump_hurdle():
    jump_up()
    move()
    jump_down()
```



**To finish the problem, we write `jump_up()`and `jump_down()`**

```
def jump_up():          def jump_down():
    turn_left()             turn_right()
    move()                  move()
    turn_right()            turn_left()
```



## Instructions that repeat

- Reeborg often has to repeat instructions. For example, to run the hurdle race, the instruction `advance_a_corner()` had to be repeated 8 times.
- There are several statements built into Reeborg's vocabulary that allow one or a suite of instructions to be repeated.

## `for`

`for` is used when it is necessary to have Reeborg perform an instruction (or suite of instructions) a certain number of times.

We previously handled this problem by writing the instruction as many times as needed. The new instruction has the following form:

```
for x in range(iteration-amount):
    <loop body suite of statement(s)>
```

where *`iteration-amount`* is the number of times the loop body statements are to be repeated.

For example, the solution of the hurdle race problem can now be written as one of the following:

| revised | previously |
|---|---|
| for x in range(8):<br>    advance_a_corner()<br>turn_off() | advance_a_corner()<br>advance_a_corner()<br>advance_a_corner()<br>advance_a_corner()<br>advance_a_corner()<br>advance_a_corner()<br>advance_a_corner()<br>advance_a_corner()<br>turn_off() |

Example:

```
def turn_right():
    for x in range(3):
        turn_left()
```

The function turn_left() is repeated three times, which is equivalent to a right turn.

## `while`

There are many situations where Reeborg needs to repeat an instruction but it is not yet known how often. For example, if we wish for Reeborg to pick up a pile of beepers of arbitrary size, he needs to repeatedly execute the pick_beeper() command, but because we do not know in advance the number of beepers in the pile, we do not know exactly how often to execute that command.

The WHILE statement is made-to-order for this situation: you can use it to tell Reeborg to repeat something while a certain predicate is True; for example to pick up beepers while there are any left to pick up.

```
while predicate:
        <loop body suite of statement(s)>
```

The *predicate* that appears in the WHILE statement comes from the same list of predicates that Reeborg can use in an IF statement.

- Reeborg executes a WHILE by first checking the predicate.
- If the predicate is True, then the loop body is executed and Reeborg loops back to the predicate to check it again.
- This continues over and over as long as the predicate evaluates to True.
- If the predicate evaluates to False, Reeborg is finished with the WHILE statement and begins executing the instruction that immediately follows the WHILE statement.

- NOTE: If the predicate is initially False the statement(s) in the loop body will not be executed at all.
- For this reason, WHILE loops are sometimes called *zero-or-more times* loops.

Examples:

```
def clear_corner_of_beepers():
     while on_beeper():
         pick_beeper()
```

Reeborg would pick up all beepers on the current corner, regardless how many there are (if it is a finite number, at least).

## Building a WHILE Loop

- **Step 1:** Identify the one test that must be True when Reeborg is finished with the loop.

  In the above problem, Reeborg must pick all beepers on the corner. If we consider only tests that involve beepers, we can choose among four:

  `carrying_beepers`, `not carrying_beepers`, `on_beeper()`, and `not on_beeper()`

  Which one is the test we want?

- **Step 2:** Use the opposite form of the test identified in step 1 as the loop predicate.

  This step implies that we should use `on_beeper()`. The WHILE instruction continues to execute the loop body as long as the test is True and stops when it is False.

- Step 3: Do whatever is required before or after the WHILE is executed to ensure we solve the given problem.

  In this example, we have to do nothing before or after the loop. However in other situations, we may miss one iteration of the loop and have to “clean things up,” which can be done either before or after the WHILE.

- Step 4: Do the minimum that is needed to ensure that the test eventually evaluates to False so that the WHILE loop stops.
  - Something within the body of the loop must allow the test eventually to evaluate to False or the loop will run forever.
  - This implies that there must be some instruction (or sequence or instructions) within the loop that is related to the test.
  - Thus in our example, because we are testing for `on_beeper()` we must pick one (and only one) beeper somewhere in the loop.

Apply these four steps to a new problem. Reeborg is somewhere in the world facing south. One beeper is on every corner between Reeborg’s current position and the southern boundary wall. There is no beeper on the corner on which he is currently standing. Write a new instruction, `clear_beepers_to_wall`, to pick all the beepers.

To solve any problem, ask questions:

What do we know about Reeborg’s initial situation?

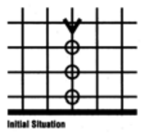


Figure 5-1 Pick All Beepers

- Reeborg is facing south
- Reeborg is an unknown distance from the southern boundary wall
- Each corner between Reeborg and the southern boundary wall has one beeper.

Does any of this information provide insight toward a solution?

- Yes – Reeborg can travel forward until it reaches the southern boundary wall. It can pick a beeper from each corner as it travels.

What Reeborg instruction can we use to keep Reeborg traveling southward until it reaches the southern boundary wall?

- Since traveling to the southern boundary wall requires an unknown number of move instructions, we can use a WHILE loop.

## Four Step Process

- Step 1: Identify the one test that must be True when Reeborg is finished with the loop.
  - Reeborg will be at the southern boundary wall, so the test `not front_is_clear` will be True.

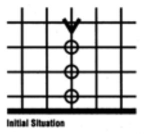


Figure 5-1 Pick All Beepers

- Step 2: Use the opposite form of the test identified in step 1.
  - The opposite of `not front_is_clear` is simply `front_is_clear`

- Step 3: Do whatever is required before or after the WHILE is executed to ensure we solve the given problem. As Reeborg is already facing south, we do not have to do anything.

- **Step 4:** **Do the minimum that is needed to ensure that the test eventually evaluates to False so that the WHILE loop stops.**

**Reeborg must move forward one block then pick a beeper.**

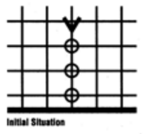


**Figure 5–1** Pick All Beepers

**Based on this discussion, we can write the following new instruction:**

```
def clear_beepers_to_wall():
    while front_is_clear():
        move()
        pick_beeper()
```

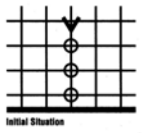


**Figure 5–1** Pick All Beepers

**A while loop can occur in a while loop:**

```
def pick_all_beepers_to_wall():
     while on_beeper():
         pick_beeper()
     while front_is_clear():
         move()
         while on_beeper():
             pick_beeper()
```

**The logic of these *nested* WHILE statements has Reeborg pick up all beepers between him and the wall ahead of him, including beepers on Reeborg's beginning street corner. Reeborg stops in front of the wall.**