Logic and Functional Programming

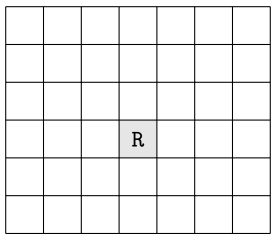
Labwork 11: Searching in an infinite space

Mircea Marin

See project assignment.

Formalizing the problem

The search space is an unbound grid of square rooms, like in figure below:



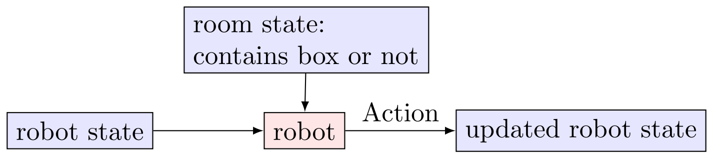
and the robot starts moving north/south/east/west from a room, called the **origin** (the gray square).

Rooms are locations where boxes are placed, or where the robot can move.

- They can be represented as pairs of integers `(X,Y)`.
- We can assume that the origin is `(0,0)`.

What should be the state of the robot?

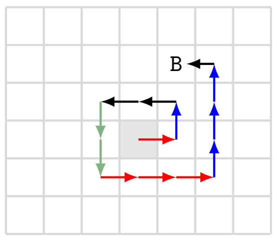
The *state* (or *program data*) of the robot is a data structure where the robot stores information that helps him perform the required task: to deliver the required number of boxes to the origin. Informally, the robot state are the things that the robot should remember in order to finish his job.



The robot is asked to find a specified number of boxes in the infinite space of rooms. To find them, he must explore all rooms.

Finding the first box

The robot must move on a path that visits all rooms (otherwise, he may never find that box). There are many possibilities to do so. One possibility is to move in a spiral, as illustrated below:



Illustrated example when first box is at location `(1,2)`.

Moves to east are red ($\rightarrow$).	Action: `move(east)`
Moves to north are blue ($\uparrow$).	Action: `move(north)`
Moves to west are black ($\leftarrow$).	Action: `move(west)`
Moves to south are green ($\downarrow$).	Action: `move(south)`

When the robot finds the box, he must pick it up (Action: `pickBox`) and start moving with it to the origin. The movement on this spiral can be described with the predicate

```
nextPair((X,Y),(X1,Y1))
```

from labwork 3.(b), which holds if `(X1,Y1)` is the location immediately after `(X,Y)` on this spiral path. To implement this predicate it is useful to note that:

- `(X1,Y1)` is `(X+1,Y)` if and only if $\mathtt{Y} \leq 0$ and $\mathtt{Y} \leq \mathtt{X} \leq -\mathtt{Y}$.

  In this case, the robot moves east (red arrow).
- `(X1,Y1)` is `(X,Y+1)` if and only if $\mathtt{X} > 0$ and $-\mathtt{X} < \mathtt{Y} < \mathtt{X}$.

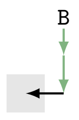
  In this case, the robot moves north (blue arrow).
- `(X1,Y1)` is `(X-1,Y)` if and only if $\mathtt{Y} > 0$ and $-\mathtt{Y} < \mathtt{X} \leq \mathtt{Y}$.

  In this case, the robot moves west (black arrow).
- `(X1,Y1)` is `(X,Y-1)` if and only if $\mathtt{X} < 0$ and $\mathtt{X} < \mathtt{Y} \leq -\mathtt{X}$.

  In this case, the robot moves south (green arrow).

Bringing a box back to the origin

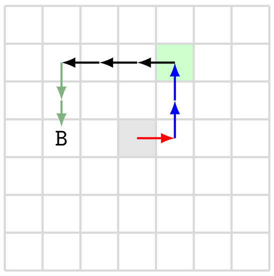
To move fast to the origin, the robot should know how far he is from the origin. If he knows his current location `(X,Y)`, he can move to origin in $|\mathtt{X}| + |\mathtt{Y}|$ steps. For example, he can move the box from `(1,2)` to `(0,0)` in 3 steps, and deliver the box in one final step:



```
moveWithBox(south)
moveWithBox(south)
moveWithBox(west)
deliverBox
```

Finding another box (not the first box)

The robot can remember where he found the last box. He can go there directly, and start moving from there in a spiral to find the next box. For example, if the next box is at position `(-2,0)`, he can move as follows:



Illustrated example when the next box is at `(-2,0)`. The last visited room is the green one.

A suitable structure for the state of the robot

We noticed that it is useful to keep track of:

- The number of boxes that the robot must deliver to the origin.
- The current location of the robot.
- If the robot has a box in his arms or not.
- The last location visited by the robot.

We can represent this state as a tuple

```
(N,(X,Y),S,(Xl,Yl))
```

where: `N` is the number of boxes that the robot must deliver; `(X,Y)` is the current location of the robot; ; `S` is `true` if the robot holds a box, and `false` otherwise; and `(Xl,Yl)` is the last location visited by the robot.

If the robot is asked to deliver `N` boxes, his initial state should be

```
(N,(0,0),false,(0,0)).
```

Implementation considerations

Predicate

```
perform(+ProgramData, +ContainsBox, -Action, -ProgramDataUpdated)
```

should select what action to do by looking at the values of the input arguments `ProgramData` and `ContainsBox`. There should be only one clause for every possible action of the robot. To make the computation more efficient, you should use the cut operator (`!`) to confirm the choice of a rule for a particular action (see Lecture 11). The following two examples illustrate how to use the cut operator to confirm the choice of a rule for a particular action.

Example: action `done`

Action `done` should be performed only if the robot is in the origin, it carries no boxes, and it has no more boxes to deliver. For this action we can define the corresponding rule

```
perform((0,(0,0),false,P),_,done,(0,(0,0),false,P)):-!.
```

Example: moving back to the last visited room

We can decide[1] that if the robot carries no box and is not in the last visited room then he should go and start searching boxes from there.

```
perform((N,(X ,Y),false,(Xl,Yl)),_,move(east),
        (N,(X1,Y),false,(Xl,Yl))):-X<Xl,!,X1 is X+1.
perform((N,(X ,Y),false,(Xl,Yl)),_,move(west),
        (N,(X1,Y),false,(Xl,Yl))):-X>Xl,!,X1 is X-1.
perform((N,(X,Y), false,(X,Yl)),_,move(north),
        (N,(X,Y1),false,(Xl,Yl))):-Y<Yl,!,Y1 is Y+1.
...
```

[1]We noticed that this is a good search strategy for boxes.