Learning to Program with F#
Exercises
Department of Computer Science
University of Copenhagen

Jon Sporring, Martin Elsman, Torben Mogensen, Christina Lioma

September 30, 2022

In the following, you are to work with the abstract datatype known as a stack. A stack is like a stack of plates in a cafeteria, they are placed in stack, and you can take the top plate or place a plate on the top, but you cannot access a plate in the middle of a stack before you have removed all above. Stacks typically comes with the following functions:

```
type element // an element on the stack such as a plate
type stack // a stack of elements
init: () -> stack // create an empty stack
// return the top element and the resulting stack
pop: stack -> element*stack
// put an element on a stack and return the resulting stack
push: element stack -> stack
```

In this exercise, you are to work with stacks in F#.

0.1 Canvas

0.1.1 Opgave(r)

0.1.1: (a) Given a source and target grid point, write the function

```
dist: p1: pos -> p2: pos -> int
```

which calculates the squared distance between positions p_1 and p_2. I.e., if $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ then $\text{dist}(p_1, p_2) = (x_2 - x_1)^2 + (y_2 - y_1)^2$.

(b) Given a source and a target and `dist`, write the function

```
candidates: src: pos -> tg: pos -> pos list
```

which returns the list of candidate next positions, which brings the robot closer to its target. I.e., if $\text{src} = (x, y)$, then the function must consider all the neighbouring positions, $\{(x+1, y), (x-1, y), (x, y+1), (x, y-1)\}$, and return those whose distance is equal to or smaller than $\text{dist}(\text{src}, \text{tg})$. This can be done with `List.filter`.

(c) Given a source and a target and by use of `candidates` the above functions, write a recursive function

```
routes: src: pos -> tg: pos -> pos list list
```

which calculates the list of all the shortest routes from `src` to `tg`. For example, the list of shortest routes from $(3, 3)$ to $(1, 1)$ are

```
[[(3, 3); (2, 3); (1, 3); (1, 2); (1, 1)];
 [(3, 3); (2, 3); (2, 2); (1, 2); (1, 1)];
 [(3, 3); (2, 3); (2, 2); (2, 1); (1, 1)];
 [(3, 3); (3, 2); (2, 2); (1, 2); (1, 1)];
 [(3, 3); (3, 2); (2, 2); (2, 1); (1, 1)];
 [(3, 3); (3, 2); (3, 1); (2, 1); (1, 1)]]
```

Beware, this list grows fast, the further the source and target is from each other, so you will be wise to only work with short distances. This can be done with a recursive function and a `List.map` of a `List.map`.

(d) Consider now a robot, which also can move diagonally. Extend `candidate` to also consider the diagonal positions $\{(x+1,y+1),(x+1,y-1),(x-1,y+1),(x-1,y-1)\}$, and update `routes` to return the list of shortest routes only. For example, the shortest routes from $(3,4)$ to $(1,1)$ should be

```
[[(3, 4); (2, 3); (1, 2); (1, 1)];
 [(3, 4); (2, 3); (2, 2); (1, 1)];
 [(3, 4); (3, 3); (2, 2); (1, 1)]]
```

but not necessarily in that order.

(e) Optional: Make a Canvas program, which draws routes, and apply it to the routes found above.