Report of Functional Programming

You ZUO

May 6, 2019

Contents

Chapter 1

Conception of data structure

The data structure of the `antenna` is the most crucial step since it influences a lot our decisions later, I considered the antenna like a wheel with a "head" which is like an arrow pointing to the element that we can access right now.

1.1 Data structure of one antenna

At the very first of time, the functions of the antenna reminded me of how integers are constructed for most of the machine languages, in general, it has integers from -256 to 255 which are put in one "wheel," and 255 and -256 are next to each other. For an antenna, we have 27 characters which 26 of them are English capital letters and the last one a blank. So I considered the blank in the middle, then for letters from 'A' to 'M,' they were arranged to the right of the blank ' ' in order, and the letters from 'Z' to 'N' in the left, and here is a demonstration diagram for it:

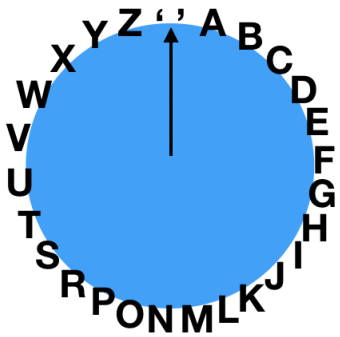

Figure 1.1: Conception of data structure

Sending a command by an antenna is like finding a convenient method to get access to the character that we need on it, which is some kinds of finding the shortest distance of two characters. So it must be figured out a way to make calculations for the characters. As a

result, I made every element in the antenna a combination of the character and an integer as its key number.

After that, I noticed that even though every antenna has a different element of "head" currently, it has exactly the same structure for this kind of relation of mapping. So for each antenna what really matters is the element "head" to inform the user that the element can be accessed. In this consideration, we can easily set the foundational structure of our antenna as a fixed list as it:

Listing 1.1: foundational structure of antenna

```
let antenne = (−1,'Z')::(−2,'Y')::(−3,'X')::(−4,'W')::(−5,'V')::(−6,'U')
              ::(−7,'T')::(−8,'S')::(−9,'R')::(−10,'Q')::(−11,'P')::(−12,'O')
              ::(−13,'N')::(0,' ')::(1,'A')::(2,'B')::(3,'C')::(4,'D')
              ::(5,'E')::(6,'F')::(7,'G')::(8,'H')::(9,'I')::(10,'J')
              ::(11,'K')::(12,'L')::(13,'M')::[];;
```

So based on this, I considered explicitly that every antenna has actually only one "head," and we can change it to other elements in the list above when we need to access different characters. Here I have an example for creating an antenna in the first phase:

Listing 1.2: example to creat an antenna

```
let ant_init = (0,' ');;
```

As you can see, here 0 and ' ' have the same signification, from listing 1.1 we can see that 0 is the key number of the character blank, so actually, I could have made the structure of the antenna less redundant. In this case, I made changes for phase 2 in which we have more quantity of antennas.

1.2 Data structure of antennas

Here we have more antennas in the second question, which means that we should give a number to each of them and note down their "head" at the same time. At that time, I realize that I may make the wrong decision for the structure for one antenna because actually for the character and the corresponding number, we needed just one of them. Therefore, I took off the character in the couple and remained only the corresponding of it. Furthermore, I gave a number for each antenna from 0 to n-1 which n is the total number of our antennas. According to these the data structure for each antenna is a combination of its identifier and the corresponding number of the character on its "head".

Listing 1.3: example to creat n antennas

```
val creer_n_ants : int -> (int * int) list = <fun>
# creer_n_ants 7;;
- : (int * int) list =
[(0, 0); (1, 0); (2, 0); (3, 0); (4, 0); (5, 0); (6, 0)]
```

For the example above, I created a list of couples which each couple represents an antenna. The first integer of the couple means the number or the identifier of the antenna, the second the key number for the character on its "head." In the beginning, all antennas have blank as its "head" which has key number 0 according to the foundational structure of our antenna.

1.3 Structure of interactive system

Plus, I designed an interactive system with users, which makes it easier to make a series of continuous manipulations. Here I give a flow chart and a test for it:

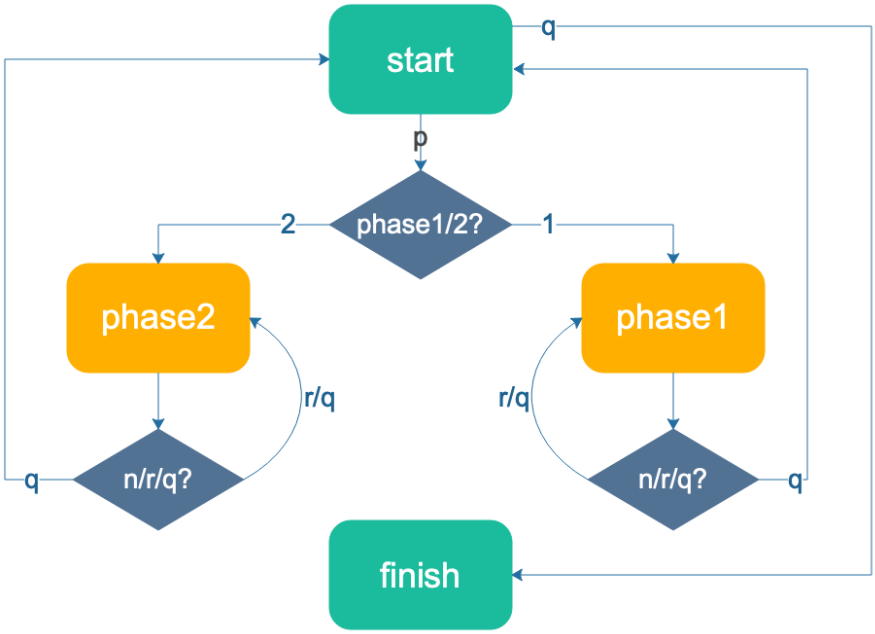

Figure 1.2: Interactive system

Welcome to the extraterrestrial message-sending system! Choose:
q: to quit the system
p: to send a message

Please choose the phase (1/2):

You are now in phase 1, please choose the instruction:
n: to initialize this(these) antenna(s) to send another message
r: to continue using this(these) antenna(s) for another message
q: to quit phase 1

q
Thank you, goodbye!

Chapter 2

Technical problems

2.1 Print character as integer

In the second part, we are supposed to print the commands which consist of the change of antenna and the manipulations for one antenna selected. So, it is necessary to declare the number of the aimed antenna when changing it. However, before that, I put all my commands calculated from the given message in a list of characters, in which way an integer like 1 would be transferred as '\001', so when it comes to print my commands, this character would be considered as vide because of '\0' in the first position.

Listing 2.1: problem when printing an int as char

```
# let a = char_of_int 1;;
val a : char = '\001'
# let _ = Printf.printf "%c" a;;
- : unit = ()
```

To solve this problem, I noticed that for the commands, the identifier of the antenna must be printed after character "S." So we can be sure that the character after "S" must be an integer, for this reason, I converted every time the character after "S" into an integer by using the function int_of_char.

Listing 2.2: function to print the commands

```
let rec affiche_cmds = fun cmds ->
  match cmds with
  | [] -> Printf.printf "\n"
  | e::l ->
    if e = 'S' then (
      match l with
      | [] -> raise (Error "Mistaken commands!")
      | n::l' -> let _ = Printf.printf "%c%i" e (int_of_char n) in affiche_cmds l'
    )
    else let _ = Printf.printf "%c" e in affiche_cmds l;;
```

2.2 Order conflict of definition of functions

In my program, I designed an interactive system with users which requires them to make their choices every time to continue the later manipulation as they want. So the functions for this interactive system should always be called by other functions, so I met a problem with the order of definition of functions: So I decided to transfer the function as an

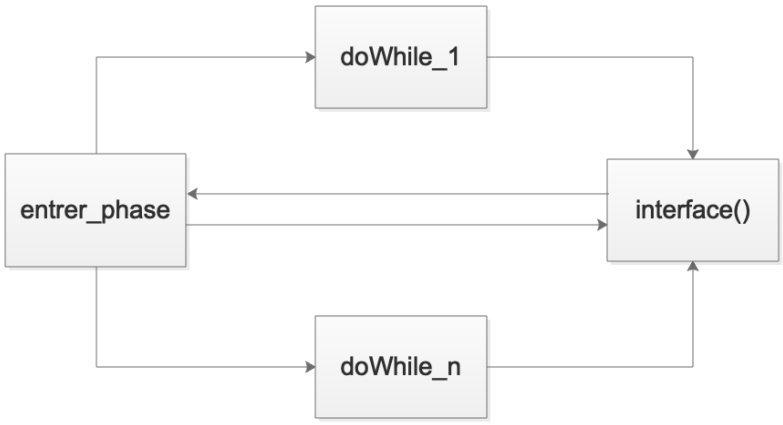

Figure 2.1: conflict of function call

argument instead of using the function call directly. That's to say my functions which need to call the function "interact()" knows only the type of this argument but not the precise definition of it.

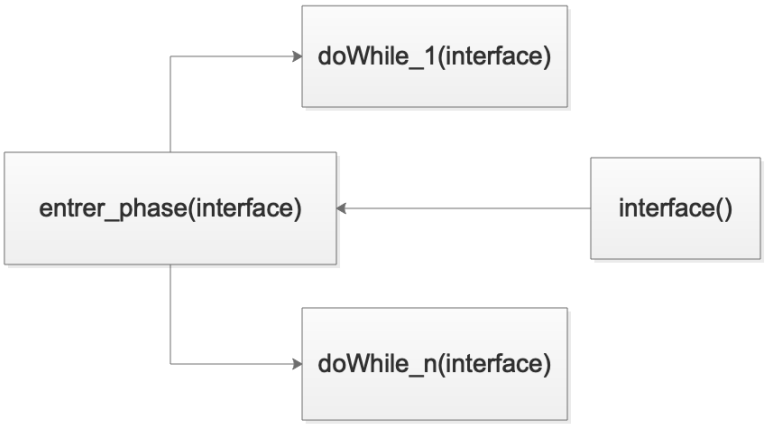

Figure 2.2: trasfer the fonction as an argument instead of calling it

Chapter 3

Limitations of programme

3.1 Resolution of optimazation

For the second part of the project, it requires to figure out a solution to optimization. It is quite easy to think about finding the quickest way for every step, which means to compare the "distance" of the character to send with its corresponding number on every antenna. Then pick up the antenna which has the least movements to set this character as its "head" and finally, we send this character out and continue another tour like it if the message has not been completely translated into commands.

However, in this way, I have ignored the fact that the optimal local solution may not be the optimal global solution. I want to use a particular example to explain: if we have at least two antennas and a message to send as CDCDCDCDCD, and if we send this message with optimal local solution, we will first turn the number 0 antenna to letter 'C' and send it by giving the commands as NNNE.

For the second character 'D' in our message, if we look for the antenna with the closest "distance" from its "head" to letter "D," with no doubt that we will get antenna 0 this time since "C" is closer to "D" than blank. So the next instruction will be NE. So according to this, we can quickly get the whole commands to send the message, which will be:

Listing 3.1: local resolution

```
Please input the message:
2
CDCDCDCDCD
NNNENEPENEPENEPENEPENE
time: 86 seconds.
```

But consider if we fix the "heads" of two of antennas in the first two rounds, and only switch between these two antennas to send the message, in this case, we will get the commands as:

Listing 3.2: global resolution

```
#   let cmds = parse_input1();;
NNNES1NNNNES0ES1ES0ES1ES0ES1ES0ES1E
val cmds : char list =
  ['N'; 'N'; 'N'; 'E'; 'S'; '1'; 'N'; 'N'; 'N'; 'N'; 'E'; 'S'; '0'; 'E'; 'S';
   '1'; 'E'; 'S'; '0'; 'E'; 'S'; '1'; 'E'; 'S'; '0'; 'E'; 'S'; '1'; 'E'; 'S';
   '0'; 'E'; 'S'; '1'; 'E']
# affiche_t 0 cmds;;
time: 80 seconds.
```

3.2 Reuse of plural antennas

As I have mentioned, I designed an interactive system and at that time I conceived what they may be able to do after sending one message in a certain phase. It's proposed to set a choice for them to quit the system, as above, but what if they want to continue to send another message? Here as for me, I considered two situations:

1. to send another message using a new group of antennas
2. to send another message with the former antennas, which means the antennas will retain the final condition after sending the last message

It works perfectly for the first phase, but seemingly it will take so many modifications or additional work to get it done, but I didn't have enough time then so I'm so sorry if you find it functions not at all for the reuse of antennas in phase 2.