LINGI1131 - Project 2017
Captain Sonar

Hélène Verheaghe & Sana Imtiaz

March 2017 - Version 2

1 Introduction

You are asked to implement a customised version of the popular game, Captain Sonar. In Captain Sonar, each player is in charge of a submarine and is trying to locate enemy submarine(s) in order to blow it out of the water before they can do the same to him. The game can be played in two modes: turn-by-turn or simultaneous. In the latter set-up, all players take their actions simultaneously while trying to track what the opponents are doing.

You are the captain of your submarine, responsible for moving the submarine and announcing some details of this movement. And since we are putting all responsibility on the captain, you also have to man the sonar in order to listen to the opposing captain's orders and try to decipher where that submarine might be in the water. Also, you need to keep track of the munitions room to prepare missiles and mines that will allow for combat!

2 Program description

This section will describe the expected architecture of your program and the motivations behind it.

2.1 Modularity

The organisation of the program will be highly modular: using functors and with given interfaces. The groups following the guidelines will be able to easily make their submarine fight against another submarine without many changes.

If enough groups follow the guidelines, we may do a little tournament with the best submissions. So try to follow them ;)

The next subsection will describe the different components of the program.

2.2 Types

To ease the modularity, we define some types to respect. There are defined with EBNF grammar.

```
<id>            ::= null | id(id:<idNum> color:<color>) name:Name)
<idNum>         ::= 1 | 2 | ... | Input.nbPlayer
<color>         ::= red | blue | green | yellow | white | black
                    | c(<colorNum> <colorNum> <colorNum>)
<colorNum>      ::= 0 | 1 | ... | 255

<position>      ::= pt(x:<row> y:<column>)
<row>           ::= 1 | 2 | ... | Input.nRow
<column>        ::= 1 | 2 | ... | Input.nColumn

<carddirection> ::= east | north | south | west
```

```
<direction>     ::= <carddirection> | surface

<item>          ::= null | mine | missile | sonar | drone
<fireitem>      ::= null | mine(<position>) | missile(<position>) | <drone> | sonar
<drone>         ::= drone(row <x>) | drone(column <y>)
<mine>          ::= null | <position>
```

More information for $< color >$:
`https://mozart.github.io/mozart-v1/doc-1.4.0/mozart-stdlib/wp/qtk/html/node36.html#misc.color`

2.3 Parameters of the game (`Input.oz`)

The game has many parameters, leading to many different games. Following are the parameters and their descriptions (the `Input.oz` file contains the setup parameters of the game) :

- `isTurnByTurn` : true means the game will be turn by turn, false means the game will be simultaneous
- description of the map :
  - `nRow` : the number of rows of the map
  - `nColumn` : the number of columns of the map
  - `map` : the description of the map, 0 meaning water, 1 meaning island. For example, the following map description give a map with 5 rows and 10 columns.

    ```
    NRow = 5
    NColumn = 10
    Map = [[0 0 0 0 0 0 0 0 0 0]
      [0 0 0 0 0 0 0 0 0 0]
      [0 0 0 1 1 0 0 0 0 0]
      [0 0 1 1 0 0 1 0 0 0]
      [0 0 0 0 0 0 0 0 0 0]]
    ```
- Description of the players :
  - `nbPlayer` : give the number of player
  - `players` : describe the types of every player (AI or human)
  - `colors` : associate a color for every player
  - `maxDamage` : if the total damage taken by the submarine reaches this number, the submarine is destroyed
- `turnSurface` : the number of turns (in turn by turn) or seconds (in simultaneous) the submarine has to wait before continuing playing
- Loading parameters :
  - `missile` : the number of loading charges needed to make a missile
  - `mine` : the number of loading charges needed to make a mine
  - `sonar` : the number of loading charges needed to make a sonar
  - `drone` : the number of loading charges needed to make a drone
- Bound of distances for weapons : Distances are computed following the following formula

$$D = |x_1 - x_2| + |y_1 - y_2|$$

  - `minDistanceMine` : the minimum distance from the position to place a mine

- `maxDistanceMine` : the maximum distance from the position to place a mine
- `minDistanceMissile` : the minimum distance from the position to explode a missile
- `maxDistanceMissile` : the maximum distance from the position to explode a missile

- Thinking parameters (simultaneous game only) :
  - `thinkMin` : minimum time (in ms) used when thinking
  - `thinkMax` : maximum time (in ms) used when thinking

Guidelines for modularity : Don't change any names, change only the values :)

2.4 Game controller (`Main.oz`)

You can see the game controller as the neutral third party nation delegated by the marine laws to be the referee in the submarine war.

He is there to ensure that submarines play by the rules and none of them performs the allowed operations in the wrong order. He manages the broadcasting radio to ensure none of the submarines illegally share information to make sub-coalitions.

He is also responsible to send the information to the GUI, allowing the marine law judges to enjoy the spectacle and control the global behaviours of the battle following the marine laws.

To ensure fairness and equity, the game controller is the one in charge of creating the GUI port and giving the radio to the submarines (creating their port objects).

```
functor
import
    GUI
    Input
    PlayerManager
define
    ...
in
    ...
end
```

What he should do, in the big lines :

1. Create the port for the GUI and launch its interface
2. Create the port for every player using the PlayerManager and assign a unique id between 1 and Input.nbPlayer ($< idnum >$). The ids are given in the order they are defined in the input file
3. Ask every player to set up (choose its initial point, they all are at the surface at this time)
4. When every player has set up, launch the game (either in turn by turn or in simultaneous mode, as specified by the input file)

Let's now explain the two game modes.

2.4.1 Turn by turn

The order of play follows the natural order of IDs (from player id 1 to player id Input.nbPlayer).

The game controller will trigger the action of every submarine in turns in order to guarantee every player plays in turn and every turn follows the right order of actions. The game controller also ensures that the number of surface turns are followed.

The possible actions for one turn of a player are summarised here :

1. Check if the submarine can play (if he is not at surface anymore). If the submarine is at the surface, go direct to point 9.

2. If this is the first round, or if in the previous turn the submarine was surface, send the dive message to the submarine

3. Ask the submarine to choose its direction. If the direction isn't going surface, continue at point 5.

4. Surface has been chosen, the turn is stopped and counted as first turn passed at surface. The game controller broadcasts, on the radio, the information that this player has made surface. The information is also broadcast to the GUI. The submarine stays a total of Input.turnSurface turns at the surface before continuing. Go to 9 to finish the turn

5. The chosen direction is broadcast trough the radio and the GUI is also informed.

6. The submarine is now authorised to charge an item. If the answer contains information about a new item produced, the information is broadcast through the radio.

7. The submarine is now authorised to fire an item. If the answer contains information about a fired item, the information is broadcast through the radio.

8. The submarine is now authorised to explode a mine. If the answer contains information about blowing a mine up, the information is broadcast trough the radio.

9. The turn is finished for this submarine

2.4.2 Simultaneous actions

In this case, the game controller launches a thread for each submarine. Each submarine loops over the following actions :

1. If this is the beginning, or the surface has ended, send the dive message to the submarine

2. Simulate thinking

3. Ask the submarine to choose its direction. If the direction isn't going surface, continue at point 5.

4. Surface has been chosen, the thread delay for Input.turnSurface seconds observed and the GUI is notified. Then go back to point 1.

5. The chosen direction is broadcast trough the radio and the GUI is also informed.

6. Simulate thinking

7. The submarine is now authorised to charge an item. If the answer contains information about a new item produced, the information is broadcast through the radio.

8. Simulate thinking

9. The submarine is now authorised to fire an item. If the answer contains information about a fired item, the information is broadcast through the radio.

10. Simulate thinking

11. The submarine is now authorised to explode a mine. If the answer contains the information about blowing a mine up, the information is broadcast trough the radio.

12. Loop of actions is finished, go back to 1

2.5 The graphical interface (`GUI.oz`)

This functor only has one export (`portWindow`) linked to a function creating a port object and launching in a thread the treatment of its stream.

```
functor
import
   QTk at 'x-oz://system/wp/QTk.ozf' % functor for graphic interface element
   Input
export
   portWindow:StartWindow
define
    StartWindow
    TreatStream
in
    fun{StartWindow}
        Stream
        Port
    in
        {NewPort Stream Port}
        thread
        {TreatStream Stream <p1> <p2> ...}
        end
        Port
    end
    proc{TreatStream Stream <p1> <p2> ...} % has as many parameters as you want
        ...
    end
end
```

The port object has to handle the following messages :

- `buildWindow` : Create and launch the window (no player on it)
- `initPlayer(ID Position)` : ID is of type $< id >$ and position of type $< position >$. If the ID isn't null, the window should show a submarine with its associated color (border around a picture, the whole square in color, a flag with the color,...) at the initial position given.
- `movePlayer(ID Position)` : Move the submarine from its previous position to the new one. The previous place of the submarine should contain an indication that it passed by this location (small colored dot,...). If two submarines are at the same position, one can disappear under the other. Except for this case, hide and seek is prohibited by the marine laws, submarines can't hide under the mines. Two submarine on the same square don't make them take damages. Following their marine registration certificate, every submarine is only allowed to travel at a given depth. In this way, if two of them are in the same square, they don't collide. Also, as they don't have windows on the submarines, they can only detect the presence of another submarine when they use sonar or drone. Moreover, they can deduce the place of the others by computation on what they hear on the radio.
- `lifeUpdate(ID Life)` : Update the life indication for the player depicted by the ID. Life is a strict positive number.
- `putMine(ID Position)` : Draw a mine, visually identifiable by the color of the associated player.
- `removeMine(ID Position)` : Remove the mine of the player identified by the ID that was at the given position. If there are two mines at the same position (from the same player or other one), only one explodes (these are special mines shielded with diamond allowing them not to blow when another mine or a missile explode at the same position).

- `surface(ID)` : indicate to the GUI that a submarine has made surface. The indication of path where the submarine went should thus disappear.
- `explosion(ID Position)` (*not mandatory*) : message to notify an explosion should be displayed (text on the side, animation at explosion point,... feel free to be inventive!), "The explosion was triggered by the player with `ID`". If the explosion is fancy enough, you can get bonus :)
- `drone(ID Drone)` (*not mandatory*): message to notify drone inspection of a specific zone. Drone is of type $< drone >$.
- `sonar(ID)` (*not mandatory*) : message to notify sonar inspection of a specific zone.
- `removePlayer(ID)` : The submarine and any element (mines, path,...) of the player depicted by ID should be removed from the board. If you made a separate life board, the life should go to 0.

In every message, if the ID is *null*, the message should be ignored (defensive programming allowing to handle dead submarine message). For a given ID, every message received concerning this ID after the reception of a `removePlayer(ID)` message should be ignored.

Some messages are not essential for the right behaviour of the game (flagged with the *not mandatory* remark). It is better to implement them, but do it only if you have time :)

The source of a working interface is given to you as you hardly never played with GUI. You are allowed to start from it but in this case we expect you to modify it (some examples : with pictures, another arrangement, size, texture, improving the function, change the state storage, ASCII board if you want,...).

Guidelines for modularity : Don't add any export, don't import other files from the project, implement the right behaviours for every message, don't invent new messages, don't modify the arguments of the messages, program in a defensive way (ignore not stored ID, ignore other messages,...). The GUI is only receiving information, it is just used to display the state of the game.

Consult the following resources for designing GUI:

- Book chapter 10 (page 679-705)
- `https://mozart.github.io/mozart-v1/doc-1.4.0/mozart-stdlib/wp/qtk/html/index.html`

2.6 Players (`PlayerXXXMyCustomName.oz`)

Each player is, as the GUI, summarised by a port object. The basic structure will thus be close to the one of the GUI.

```
functor
import
   Input
export
   portPlayer:StartPlayer
define
    StartPlayer
    TreatStream
in
    fun{StartPlayer Color ID}
        Stream
        Port
    in
        {NewPort Stream Port}
        thread
        {TreatStream Stream <p1> <p2> ...}
        end
        Port
    end
```

```
    proc{TreatStream Stream <p1> <p2> ...} % has as many parameters as you want
        ...
    end
end
```

The messages handled are :

- `initPosition(?ID[1] ?Position)` : player has to choose its initial position. He answers by binding ID to its own $< id >$ and Position to the chosen position. Initial position can only be a water square on the map.
- `move(?ID ?Position ?Direction)` : player has to choose where to move, he answers by binding its ID, new position and giving the direction as type $< direction >$. Going east means increasing the y part by one, going south meaning increasing x part by one and so on. If the direction is surface, the position remains unchanged. The submarine can only go on a square it didn't visit since last surface phase. The submarine also can't go on square with island.
- `dive` : player is granted the permission to dive again.
- `chargeItem(?ID ?KindItem)` : Player increases the load (by 1) on one of its loaded items (mine, missile, drone or sonar). If a new item is created (the loader reaches the right number of loads given in the input file), the player should say it. It is done by binding its ID and giving the item load of type $< item >$. If no item is created, the ID is still bound but the item has null value.
- `fireItem(?ID ?KindFire)` : Player may choose to use one of his item (only possible if at least one item is ready). He answers the message by binding its ID and the item fired (of type $< fireitem >$). The item fired contains parameters specific to the item (the mine will have the position of setup, the missile will have the position of explosion, the drone will have the kind of coordinate (row or column) he is looking and the coordinate of it, and the sonar doesn't have any parameters). If no item is fired, the ID is still bounded, but the item has value null.
- `fireMine(?ID ?Mine)` : If a mine was already placed before, the player may decide to make one explode. In this case he binds its ID and Mine (of type $< mine >$) is bound to the position of the mine chosen. If no mine is selected, ID is still bounded and Mine has value null.
- `isSurface(?ID ?Answer)` : player binds its id and puts in the answer field true if it is at surface and false otherwise.
- `sayMove(ID Direction)` : as all the messages beginning with say, this message is an indication from the others. As already mentioned, the neutral referee (game controller) doesn't want to complicate itself and may just broadcast message to everyone. It is thus possible that the ID in the message corresponds to the ID of the player (possible for every indication messages). This message indicates that the player identified by ID has moved in the direction (as a $< carddirection >$) given.
- `saySurface(ID)` : Indicate that player identified has made surface.
- `sayCharge(ID KindItem)` : Indicate that player identified has charged the item.
- `sayMinePlaced(ID)` : Indicate a mine has been planted by the player identified.
- `sayMissileExplode(ID Position ?Message)` : Indicate that the player identified has made a missile explode at the given position. The receiving player has to answer its damage situation by binding the Message variable to a message indicating its damage status (message that will be broadcast by the game controller). The damages are computed as follows : if the distance between the submarine and the explosion is ≥ 2, the submarine doesn't get damages, if the distance is $= 1$, the submarine gets 1 damage and if the distance is $= 0$ (same square), the submarine gets 2 damages. If no damage had

[1] Question marks are only there to ease the reading, meaning the parameter concerned is initially unbound and the initialisation is done during the treatment of the message

been taken, null is sent. If the appends lead to death, the message `sayDeath(Id)` is put as Message value, Id having the identification of the dead submarine. If damages were taken but the submarine is still alive, the message `sayDamageTaken(Id Damage LifeLeft)` indicates the Id of the damaged submarine, the number of damage points taken and the life left of the submarine.

- `sayMineExplode(ID Position ?Message)` : This one is the same as the previous one except that it concerns mine explosion (detonators of mines and missiles emits different type of electromagnetic waves allowing them to be distinguished).
- `sayPassingDrone(Drone ?ID ?Answer)` : A drone detection is occurring, the submarine has to answer the question given in the drone (is the submarine on row/column of the number given?). To answer, the ID is bound and Answer is put to true or false
- `sayAnswerDrone(Drone ID Answer)` : These are the messages you get back after sending a drone. They indicate for every player (may be yourself too) the answer they gave to the drone query
- `sayPassingSonar(?ID ?Answer)` : A sonar detection is occurring, the submarine has to answer (it can be its own sonar) the question by binding its id and by answering a position with one coordinate right and the other wrong (answer $pt(x:3y:2)$ when the real position is $pt(x:3y:7)$ for example).
- `sayAnswerSonar(ID Answer)` : These are the message you get back after sending asking for a sonar. They contain the ID of the player answering (may be yourself), and the position they return (one coordinate true, one false)
- `sayDeath(ID)` : Informative message of the death of the identified player
- `sayDamageTaken(ID Damage LifeLeft)` : Informative message of the damage received and the state of life of the identified player.

If a dead submarine receive a message asking for an answer, the one asking for ID should just bind the ID to the null value. The message noticing explosions just bind the value of the message to null. Game controller should handle the case where ID is null.

Guidelines for modularity : We ask you to name your player following this rule : The file is in form PlayerXXXMyCustomName.oz, XXX being replaced by your group number (004, 056, 103,...) and MyCustomName by any descriptor you can chose to differentiate your different players. The name (used in the input file) associated with your player file will be the name of your file, in lowercase. For example, Player042BasicAI.oz contains the player named player042basicai. This will ensure every player made by each group has a different name.

2.7 Selection of the right player (`PlayerManager.oz`)

This file is responsible for facilitating the selection of the player following the type of the players given in the input file.

```
functor
import
   Player042BasicAI
   Player023AdvancedAI
   Player005Custom
   Player053Human
   ....
export
   playerGenerator:PlayerGenerator
define
   PlayerGenerator
in
   fun{PlayerGenerator Kind Color ID}
```

```
      case Kind
      of player042basicai then {Player042BasicAI.portPlayer Color ID}
      [] player005custom then {Player005Custom.portPlayer Color ID}
      ...
      end
   end
end
```

This file shall be the only one (with the input one to change the names) to modify to make different player playing together. To add a new player, the name of the file should be added to the import section and a pattern matching line should be added to create the corresponding port object for a player.

Guidelines for modularity : Nothing should be done here except adding players to the list.

3 Functors and compilation

For Syntax and use of functors, consult book pages 220–230

Compiling functors : `ozc -c myfunctor.oz`

Executing functors : `ozengine myfunctor.ozf`

To make the whole project working, first compile the `Input.oz` file, `PlayerManager.oz` file, players files, `GUI.oz` and `Main.oz`. Then execute the created functor file `Main.ozf`.

4 What do you have to do (summary)

This work has to be done by groups of maximum 2 persons. The deadline is Friday, the 28 of April (S10). Normally your code should not contain cells but only declarative code, with threads and instructions to create ports and send messages to it.

4.1 Mandatory

- Create a `GUI.oz` file for the GUI or modify the one given
- Create a `Main.oz` file for the game controller that should be able to handle both turn by turn and simultaneous game modes (it should adapt according to the variable specified in the Input file)
- Create at least two different players (the first two should have strictly different behaviours for everything). Beginning with a random one is a good idea to make you used to the rules.
- Update the `PlayerManager.oz` for your own players
- Write a `Report.oz` explaining your implementation and detailing the two strategies for the two first players

4.2 Extension Part (how to get more points)

This is a list of ideas to improve the project; you don't have to do them all and you don't have to do them in order. Feel free to follow your own ideas (you can post them on the forum, so we can discuss about them)!

- Create a human player file, creating its own window making it possible for a human player to answer the messages sent by the game controller.
- Create a generator of maps in the Input file, creating the map used.
- Create other players files, with other behaviours and strategies. There can be overlap of strategies in these files (same displacement but different firing approach,...)

- Manage to implement a GUI allowing us to display only one submarine at a time (button to select either every submarine or one in particular)
- Add awesome sounds to the GUI, making submarine war great again! ;)

5 Q & A

5.1 Modularity

- **Can we put utility functions in another file for the methods used by all the players?** I would rather not, this will keep player separated, in one file each of them, making it easier to exchange the binaries (no dependances,...).

5.2 Positions

- **How do we chose the starting position?** Random, always in the corner, always at a fix position,... It's a strategic decision, you can chose how you want. But keep in mind the starting point should be a water square and not an island one.