Vidéo

https://drive.google.com/file/d/1Pt2whWpquAvWg-phHrST04JQmk0qLQJ8/view?usp=drive_link

Fonctions ajoutées

Une fois avoir terminé toutes les fonctionnalités de mon programme, j'ai amélioré mon code à l'aide d'une couche d'abstraction. Voici les fonctions élaborées durant cette phase :

calculateCoords est une fonction qui prend une direction, le nombre de 'blocks' qui reste et les coordonnées x et y du devant d'un bateau. Elle me permet de calculer la coordonné d'un bloc du bateau dépendamment de la direction dont le bateau fait face, le nombre de bloc du bateau qui reste à calculer et les coordonnées du devant du bateau. J'utilise cette fonction lors de l'ajout d'un nouveau bateau.

```
//Calculate coords from the end to the front
let calculateCoords (facing: Direction) (blocksLeft: int) (x: int, y: int): Coord =
    //Return the correct position based on the direction of the ship and the number of blocks left to calculate
    match facing with
    | North -> (x + blocksLeft, y)
    | South -> (x - blocksLeft, y)
    | East -> (x, y - blocksLeft)
    | West -> (x, y + blocksLeft)
```

generateCoordsList est une fonction qui prend une size, un centre et une direction en paramètre. Cette fonction me permet de générer une liste des coordonnées d'un bateau à partir de sa taille et son centre. Chaque coordonnée est calculée grâce à la fonction précédente, **calculateCoords** . Évidemment, cette fonction est aussi utilisée durant l'ajout d'un bateau.

```
//Function to generate the list of coords of a new ship
let generateCoordsList (size : int) (center: Coord) (facing: Direction): Coord List =
    //Get the x and y of the center
    let (center_x, center_y) = center
    //Get offset based on size of the size
    let offset =
        if size % 2 = 0 then
            (size / 2) - 1
        else
            size / 2
    //Determine the coords in front of the ship
    let (frontx, fronty) =
        match facing with
        | North -> (center_x - offset, center_y)
        | South -> (center_x + offset, center_y)
        | East -> (center_x, center_y + offset)
        | West -> (center_x, center_y - offset)

    //Recursively generate the list of coords, starting from the last position,
    //so the front of the ship is the first elem of the list (blocks = each coord of the ship)
    let rec generateShipCoords (currentBlock: int): Coord List =
        if currentBlock = size then
            []
        else
            (calculateCoords facing (currentBlock) (frontx,fronty))::generateShipCoords (currentBlock + 1)

    generateShipCoords 0
```

calculateCoordsByShip est une fonction qui prend un centre, un nom et une direction. Cette fonction me permet de générer la liste des coordonnées d'un bateau en prenant son nom et en appelant **generateCoordsList** avec la taille associée au bateau. Cette fonction est appelée dans **createShip** et aide aussi à la l'ajout des bateaux.

```
//Calculate coords for each different ship
let calculateCoordsByShip (center: Coord) (name: Name) (facing: Direction): Coord List =
    //Calls generateCoordsList with the size according to the ship name
    match name with
    | Spy -> (generateCoordsList 2 center facing)
    | PatrolBoat -> (generateCoordsList 2 center facing)
    | Destroyer -> (generateCoordsList 3 center facing)
    | Submarine -> (generateCoordsList 3 center facing)
    | Cruiser -> (generateCoordsList 4 center facing)
    | AircraftCarrier -> (generateCoordsList 5 center facing)
```

surroundCoordWithPerimeter est une fonction qui prend une coordonné en paramètre, et retourne toutes les coordonnées voisines (8). Cette fonction est utile pour générer une liste des coordonnées périmètre à un bateau.

```
//Generate a list of coords around the passed coord
let surroundCoordWithPerimeter (coord: Coord) : Coord List =
    //Get the x and y of coord
    let (x, y) = coord
    //Add every position around the coord
    [(x - 1, y);(x + 1, y);(x, y - 1);(x, y + 1);(x - 1, y - 1);(x - 1, y + 1);(x + 1, y - 1);(x + 1, y + 1)]
```

getPerimeterCoords est une fonction qui prend une liste de coordonnées et une dimension en périmètre et retourne une liste de coordonnées périmètre. J'utilise la fonction précédente **surroundCoordWithPerimeter** pour trouver toutes les coordonnées autour de chaque coordonnée dans la liste passée en paramètre. Ensuite je filtre les coordonnées périmètres qui sont hors des dimensions, les coordonnées passées en paramètre (car ce sont les coordonnées d'un bateau) et ensuite j'enlève les coordonnées qui sont dupliquées.

```
let getPerimeterCoords (coordsList: Coord List) (dims: Dims): Coord List =
    //Get the dimension of the grid
    let (dimx, dimy) = dims
    //Recursively add every coord surrounding coord to a list
    let rec generatePerimeterCoords coords =
        match coords with
        | [] -> []
        | (x, y)::r -> (surroundCoordWithPerimeter(x,y))@(generatePerimeterCoords r)
    let perimeterCoords =
        //Call the recursive fun to generate all the coords
        generatePerimeterCoords coordsList
        //Then remove coords that are out of bounds from the list
        |> List.filter (fun (x,y) -> x >= 0 && x < dimx && y >= 0 && y < dimy)
        //Then remove the ship coords from the list
        |> List.filter (fun coord -> not (List.contains coord coordsList))
        //Then remove duplicates
        |> List.fold (fun acc coord -> if (List.contains coord acc) then acc else coord::acc) []
    //Return the list
    perimeterCoords
```

getSector est une fonction d'ordre supérieure qui prend un grid et une coordonnée. Elle itère récursivement à travers la grille et les rangées de la grille. Une fois arrivé à la coordonnée envoyée en paramètre, elle exécute la fonction f qui prend le sector en paramètre. En principe, cette fonction permettre de me rendre a un sector spécifique et de faire ce que je veux avec.

```
//Go to specific coord and do something
let getSector (grid: Sector Grid) (coord: Coord) f =
    //Get x and y of coord
    let (x, y) = coord

    let checkSectorStatus sector =
        //Call passed fun
        f sector

    //Go to the right sector using recursion and index
    let rec checkSectorList sectorList sectorIndex =
        match sectorList with
        //Return none if reached the end
        | [] -> None
        //If reached the index, checkSectorStatus else recursively advance through sectorList
        | sector::restSectorList -> if sectorIndex = y then (checkSectorStatus sector) else (checkSectorList restSectorList (sectorIndex + 1))

    //Go to the right row using recursion and index
    let rec checkGrid grid' rowIndex =
        match grid' with
        //Return none if reached the end
        | Empty -> None
        //If reached the index, checkSectorList else recursively advance through grid
        | Row (sectorList, restGrid) -> if rowIndex = x then (checkSectorList sectorList 0) else (checkGrid restGrid (rowIndex + 1))

    //Call the recursive fun
    checkGrid grid 0
```

setSector est une fonction qui prend un grid, une coordonnée et un sector en paramètre. Cette fonction me permet d'aller à une coordonnée et de changer le sector pour le nouveau sector envoyé.

```
let setSector (grid: Sector Grid) (coord: Coord) (newSector: Sector): Sector Grid =
   //Get x and y of coord
    let (x, y) = coord

   //Goes to y coord with sectorIndex and updates the data with newSector on the passed row
    let updateSector (sectorIndex: int) (newSector: Sector) (row: Sector List): Sector List =
        //Go through list. If i = sector index then return newSector, else return oldSector. Creates new list with the returns
        List.mapi (fun i oldSector -> if i = sectorIndex then newSector else oldSector) row

    //Recursively go through the grid
    let rec updateRow rowIndex grid =
        match grid with
        | Empty -> Empty
        | Row (sectorList, restGrid) ->
            //If we have reached the x coord
            if rowIndex = x then
                //Update row with new sector
                let updatedRow = (updateSector y newSector sectorList)
                //Return the new row with updated sector list and the rest of the grid
                Row(updatedRow, restGrid)
            else
                //If we haven't reached the x coord, return a new return with the current sector list, and recursively go to the next row
                Row(sectorList, updateRow(rowIndex + 1) restGrid)

    //Call the recursive fun
    updateRow 0 grid
```

iterGrid est une fonction d'ordre supérieure qui prend un grid et une fonction f en paramètre. Comme son nom le mentionne, elle permet d'itérer un grid et d'appliquer une fonction f sur chaque sector présent dans le grid. Similaire a un List.map.

```
//Iterate through the grid and apply f to each element
let iterGrid (grid: Sector Grid) f: Sector Grid =
    //Apply f to each sector
    let updateSector sector x y =
        f sector x y

    //Iterate through the sector list
    let rec iterSectorList sectorList rowIndex sectorIndex: Sector List =
        match sectorList with
        | [] -> []
        | sector::restSectorList -> (updateSector sector rowIndex sectorIndex)::(iterSectorList restSectorList rowIndex (sectorIndex + 1))

    //Iterate through the grid
    let rec iterRows grid' rowIndex =
        match grid' with
        | Empty -> Empty
        | Row (sectorList, restGrid) ->
            let updatedRow = (iterSectorList sectorList rowIndex 0)
            Row(updatedRow, (iterRows restGrid (rowIndex + 1)))

    //Call the recursive function
    iterRows grid 0
```

removeShip est une fonction qui prend un nom et un grid en paramètre. Elle permet de mettre tous les sectors qui contient le nom passé en paramètre à Clear. Elle utilise **iterGrid** pour pouvoir itérer à travers le grid.

```
let removeShip (shipName: Name) (grid: Sector Grid) =
    //Function passed to iterGrid
    let extractDataFromGrid sector x y =
        match sector with
        //If clear, stay clear
        | Clear -> Clear
        //If active and name = shipName, clear
        | Active (name, _) when name = shipName -> Clear
        //If active and name != shipName, copy sector
        | Active (name, pos) -> Active (name, pos)
    //Call recursive function
    iterGrid grid extractDataFromGrid
```

getAllSector est une fonction d'ordre supérieure qui prend un grid et une fonction en paramètre. Cette fonction itère à travers le grid et applique la fonction f sur chaque rangée. Cela me permet de faire des applications différentes sur les rangées.

```
//Function to get a list of all sectors in a grid and apply a function to each sector
let getAllSector grid f =
    //Recursive function to iterate through the grid. Store all coords in a list
    let rec checkGrid grid rowIndex =
        match grid with
        //If empty, return empty list
        | Empty -> []
        //If row, check individual sector with getAllActiveSectorRow (returns a list of coords) and concatenate with recursive call on checkGrid
        | Row (sectorList, restGrid) -> (f sectorList rowIndex)@(checkGrid restGrid (rowIndex + 1))
    //Call recursive function
    checkGrid grid 0
```

getAllSectorRow est une fonction qui prends une liste de sector et un index. Elle est passée en paramètre à la fonction précédente, **getAllSector**. Cette fonction me permet de prendre les données nécessaires, dans le format désiré lorsque du parcours du grid. Formatter l'information comme désiré m'a été très utile pour l'extraction des données du grid.

```
 //Function to get a list of all Active sectors in a row
let getAllSectorRow (sectorList: Sector List) (rowIndex: int) =
    //Recursive function to iterate through the row. Store all coords in a list
    let rec checkRow sectorList sectorIndex =
        match sectorList with
        //If reached the end of the list, return empty list
        | [] -> []
        //If there is a sector, match
        | sector::restSectors ->
            match sector with
            //If the sector is clear, check the rest of the row
            | Clear -> (checkRow restSectors (sectorIndex + 1))
            //If the sector is active, add the current coord to the list and check the rest of the row recursively
            | Active(name,pos) -> (name, pos, rowIndex, sectorIndex)::checkRow restSectors (sectorIndex + 1)
    //Call recursive function
    checkRow sectorList 0
```

getDimsGrid est une petite fonction utilitaire qui prends un grid en paramètre et qui retourne la dimension de celle-ci.

```
//Function to get dimensions of the grid
let getDimsGrid (grid: Sector Grid) : Dims =

    //Recursively go to the last row while incrementing rowIndex
    let rec countRows grid' rowIndex =
        match grid' with
        | Empty -> rowIndex
        | Row (sectorList, restGrid) -> (countRows restGrid (rowIndex + 1))
    let nbRows = countRows grid 0

    //Get the length of the first sector list
    let countColumns grid' =
        match grid' with
        | Empty -> 0
        | Row (sectorList, _) -> List.length sectorList
    let nbColumns = countColumns grid
    (nbRows, nbColumns)
```

getAllSectorRow est une autre fonction qui est passée en paramètre à la fonction **getAllSector**. Elle me permet de formatter les données des secteurs comme je le désire. Cette fonction est utile pour avoir une liste des coordonnées de toutes les sectors qui sont Active.

```
//Function to get a list of all Active sectors in a row
let getAllSectorRow (sectorList: Sector List) (rowIndex: int): Coord List =
    //Recursive function to iterate through the row. Store all coords in a list
    let rec checkRow sectorList sectorIndex =
        match sectorList with
        //If reached the end of the list, return empty list
        | [] -> []
        //If there is a sector, match
        | sector::restSectors ->
            match sector with
            //If the sector is clear, check the rest of the row
            | Clear -> (checkRow restSectors (sectorIndex + 1))
            //If the sector is active, add the current coord to the list and check the rest of the row recursively
            | Active(_,_) -> (rowIndex, sectorIndex)::(checkRow restSectors (sectorIndex + 1))
    //Call recursive function
    checkRow sectorList 0
```

canPlaceCoords est une fonction qui prend deux listes de coordonnées, un grid et un bool. Comme le nom dit, cette fonction était utile pour voir si le mouvement d'une certaine liste de coordonnées est possible sur un grid.

```
//Function to verify if the new coords can move
let canPlaceCoords (coords: Coord List) (grid: Sector Grid) (coordsToExclude: Coord List) (checkPerimeter: bool): bool =
    //Get dimensions of the grid
    let (dimx, dimy) = getDimsGrid grid

    //Get a list of coords of all active sectors
    let allActiveCoords = Grid.getAllSector grid getAllSectorRow

    //Remove coords from coordsToExclude list from the allActiveCoords list. Useful for canMove (exclude old ship coords)
    let allActiveCoords = List.filter (fun coord -> not (List.contains coord coordsToExclude)) allActiveCoords

    //Get a list of all perimeter coords if checkPerimeter is true. Useful for difference between Play and Fleet Deployment
    let allPerimeter =
        if checkPerimeter then
            Ship.getPerimeterCoords allActiveCoords (dimx, dimy)
        else
            []

    //Verifies if newShipCoords has coords that are out of bounds
    let outOfBounds = List.exists (fun (x, y) -> x < 0 || x > (dimx - 1) || y < 0 || y > (dimy - 1)) coords
    //Verifies if newShipCoords has coords that are on an active sector
    let onActiveSector = List.exists (fun elem -> List.contains elem allActiveCoords) coords
    //Verifies if newShipCoords has coords that are on a perimeter
    let onPerimeter = List.exists (fun elem -> List.contains elem allPerimeter) coords

    //All verifications
    not outOfBounds && not onActiveSector && not onPerimeter
```

getMoveCoords est une fonction qui prend une liste de coordonnées et une direction. Elle retourne la liste des coordonnées après une translation.

```
//Get a list of the moved coord based on direction
let getMoveCoords (coords: Coord List) (direction: Direction): Coord List =
    //Mapped list of ship coords. Translate coord according to direction.
    let movedShipCoords = (List.map (fun (x, y) ->
        match direction with
        | North -> (x - 1, y)
        | South -> (x + 1, y)
        | East -> (x, y + 1)
        | West -> (x, y - 1)
    ) coords)

    movedShipCoords
```

getCenterBlockPos est une petite fonction utilitaire qui prend une size et retourne le milieu de se size. Utile pour trouver le centre d'un bateau.

```
//Find center by size
let getCenterBlockPos (size: int) =
    if size % 2 = 0 then
        (size / 2) - 1
    else
        size / 2
```

getNewCenter est une fonction qui prend une liste de coordonnées en paramètres. Elle permet de trouver le nouveau centre d'un bateau à la suite d'un mouvement.

```
//Get a new center of a moved ship
let getNewCenter (movedShipCoords: Coord List) : Coord =

    //Get the pos of the center block of the ship
    let centerBlockPos = getCenterBlockPos (List.length movedShipCoords)

    //Get the center block coords based on the moved ship coords and the center block position
    let rec getCenterBlockCoords shipCoords' pos =
        match shipCoords' with
        //If we reach the end of the list, return (0,0) (shouldn't happen)
        | [] -> (0, 0)
        //If we reach the center block, return the coord
        | coord::_ when pos = centerBlockPos -> coord
        //If we haven't reached the center block, continue to the next coord
        | coord::restCoord -> getCenterBlockCoords restCoord (pos + 1)

    //Get the center block coords based on the moved ship coords and the center block position
    getCenterBlockCoords movedShipCoords 0
```

getRotateCoords est une fonction qui prend une liste de coordonnées et deux directions en paramètres. Elle me permet de calculer les coordonnées d'un bateau à la suite d'une rotation.

```
let getRotateCoords (coords: Coord List) (oldDirection: Direction) (newDirection: Direction): Coord List =
    let centerBlockPos = getCenterBlockPos (List.length coords)

    let rotatedCoords = (List.mapi (fun currentPos (x, y) ->
        let distanceFromCenter = centerBlockPos - currentPos
        match oldDirection, newDirection with
        | North, East ->    (x + distanceFromCenter, y + distanceFromCenter)    //From north to east
        | North, West ->    (x + distanceFromCenter, y - distanceFromCenter)    //From north to west
        | North, South ->   (x + (distanceFromCenter * 2), y)                   //From north to south

        | South, East ->    (x - distanceFromCenter, y + distanceFromCenter)    //From south to east
        | South, West ->    (x - distanceFromCenter, y - distanceFromCenter)    //From south to west
        | South, North ->   (x - (distanceFromCenter * 2), y)                   //From south to north

        | East, North ->    (x - distanceFromCenter, y - distanceFromCenter)    //From east to north
        | East, South ->    (x + distanceFromCenter, y - distanceFromCenter)    //From east to south
        | East, West ->     (x, y  - (distanceFromCenter * 2))                  //From east to west

        | West, North ->    (x - distanceFromCenter, y + distanceFromCenter)    //From west to north
        | West, South ->    (x + distanceFromCenter, y + distanceFromCenter)    //From west to south
        | West, East ->     (x, y  + (distanceFromCenter * 2))                  //From west to east
        | _, _ -> (0,0)

    ) coords)

    rotatedCoords
```

Fonctions d'ordres supérieur :

- getSector
- iterGrid
- getAllSector

Difficultés

Bien que j'aie fait des fonctions d'ordres supérieures, je trouve qu'ils ne sont pas optimaux. Les deux fonctions **getAllSectorRow** qui sont passées à **getAllSector** sont très semblables et aurait pu être plus efficaces. La seule différence entre les deux fonctions est un des cas du match. Or, je ne sais pas comment passer un cas de match en paramètre. J'ai essayé, mais parce que le module Grid n'a pas accès au type Sector, je ne pouvais pas utiliser les cas 'Active' et 'Clear'.

Ma fonction iterGrid ressemble aussi beaucoup à ma fonction setSector. Je suis confiant qu'il y ait une façon de rendre iterGrid plus modulaire et efficace mais je n'arrivais pas à le déchiffrer.

Le manque de temps à jouer un gros rôle dans mes fonctions non-optimisées.