# Projet de 3I008 (MPIL) - Partie 1
# ORush

Zixuan FENG
Kewei XU

11/04/2019

# Porte.ml :

## Type :

***orientation:***

Qui est l'orientation d'un bateau qui doit être ***H*** comme horizontal et ***V*** comme vertical.

***boat :***

Qui est l'état d'un bateau : son identifiant ***i*** (type char), sa longueur ***l*** (type int), son orientation ***o*** (type orientation), sa position ***x*** (type int) et sa position ***y*** (type int).

***state :***

Qui représente le status du port. C'est une liste des bateaux.

## Exception:

***End :***

Elle est utilisée dans la fonction ***input_state*** pour marquer la fin du fichier input.

## Fonction :

***boat_of_string :***

Pour un string qui est donné, on le divise en char et les transmet en bons types. Après on renvoie le bateau selon ces informations.

***string_of_boat :***

Pour un bateau qui est donné, on le représente sous la forme ***ILOXY*** (la même façon dans le sujet) avec les valeurs correspondantes dans boat.

***boat_exist :***

Pour un identifiant qui est donné, on teste s'il existe déjà un bateau qui a le même identifiant avec lui dans un state donné. Si oui, on retourne l'indice de ce bateau ; sinon, on retourne -1.
Pour faire cette méthode, on crée une liste des identifiants de tous les bateaux dans le state. Par la fonction ***List.mem*** on peut savoir s'il est dedans ou pas. Si il retourne vrai, on utilise la fonction interne « ***search_index*** » pour chercher son indice.

***add_boat :***

Pour ajouter un boat dans un state, on doit s'assurer que cela n'existe pas un d'autre bateau qui a le même l'identifiant avec lui dans le state donné (en utilisant la fonction ***boat_exist*** qu'on a défini avant ) et que la position qu'il veut rester doit être libre.
Pour tester si la position est libre, on crée une fonction interne qui s'appelle « ***verifier*** ». Elle teste selon les orientations des bateaux s'il y a des cases libéré.

***grid_of_state :***

Cette fonction d'aboard initialiser une matrice de taille 6*6 avec le char '~'. Ensuite, pour chaque bateau de state, on remplit ses cases avec son identifiant.

***input_state :***

Cette fonction prend un argument de type ***in_channel***. Pour chaque ligne, on la lit par cet argument, on la transmet en type bateau avec la fonction ***boat_of_string*** et l'ajoute en utilisant la fonction ***add_boat*** dans un state initialisé au début de la fonction.

Cette fonction peut lancer une exception ***End*** quand il arrive à la fin de ***in_channel***.

***output_state :***

Cette fonction prend un state et une sortie de type ***out_channel***. Pour l'envoyer vers ***out_channel***, elle transmet le state dans une matrice de char en utilisant la fonction ***grid_of_state***. Ensuite pour chaque case de la matrice, on utilise ***output_char*** pour afficher.

# Moves.ml :

## Type :

***direction:***

C'est la direction d'un movement qui doit être ***A*** pour avancer et ***R*** pout reculer.

***move :***

C'est la description d'un movement qui contient l'identifiant du bateau (type char) qui va appliquer ce movement et la direction (type direction).

## Exception :

***Cannot_move:***

Elle est utilisée dans la fonction ***apply_move*** pour dire qu'un movement ne peut pas être appliqué. Cela peut être lancée quand le bateau n'existe pas ou la case qu'on veut occuper n'est pas libre.

## Fonction :

***string_of_move :***

Pour un move donné, on fait la concaténation de l'identifiant du movement et sa direction ('>' pour A et '<' pour R).

***move_of_string :***

Pour un string donné, on le divise en deux parties: le premier char est l'identifiant du movement et le deuxième correpond à la direction du movement (La direction est A si le char est '>' et sinon c'est R).

***copy_boat :***

Retourne une copie du bateau donné.

***copy_state :***

Retourne une copie du state donné. On crée une nouvelle liste vide et pour chaque bateau dans le state, on enregistre sa copie dans la nouvelle liste.

***apply_move :***

D'abord on teste dans le port s'il existe un bateau qui a le même identifiant avec celui du movement. Sinon on lance l'exception ***Cannot_move***.

Une fois qu'on est sûr que le bateau existe, on regarde si la case qu'on va occuper est libre ou pas. Selon l'orientation du bateau et la direction du movement, il y a 4 cas possibles. Pour chaque cas, on teste si le bateau est encore dans la région du port et si la case à prendre est libre en regardant si la case correspondante dans la matrice est bien '~'. Si oui, on la récupère et libérer la case d'où on part ; sinon on lance l'exception ***Cannot_move***.

A la fin, on retourne un nouveau state qui correspond à l'état après appliquer ce movement.

***win :***

On trouve le bateau qui correspond à l'état du bateau A et retourne si ses coordonnées sont (4,2).

***check_solution :***

On définit une fonction interne qui s'appelle « ***apply*** ». Elle prend un state « ***s*** » et un string « ***str*** » comme les arguments. D'abord on teste si « ***s*** » est déjà gagné. Si c'est vrai, on retourne true ; sinon, on regarde s'il existe encore des movements à appliquer, sinon on retourne false. Si c'est vrai, on prend les deux premières caractères et les transmet au type move en utilisant la fonction ***move_of_string***. Après on l'applique avec la fonction ***apply_move***. Et on appelle récursivement la fonction ***apply*** avec le nouveau state obtenu et le reste des movements.

# Solver.ml :

## Type :

***state_solving:***

Qui est utilisé dans les fonctions ***solve_state*** et ***solve***. Ce type permet pour chaque state, on enregistre lui-même et les movements appliqués pour arriver ici.

## Exception :

***Solution_trouve of string :***

Qui est utilisé dans la function ***solve*** pour arrêter le programme une fois qu'on trouve la solution. Le string est la solution trouvée.

***Unequal :***

Qui est utilisé dans la fonction ***campare_state*** pour marquer qu'on a trouvé la différence entre deux state donné.

## Fonction :

***all_possible_moves :***

Pour chaque bateau dans la liste, on teste s'il peut avancer ou reculer. Si c'est vrai, on l'enregistre dans la liste ***res***, sinon on continue à tester jusqu'à la fin de la liste.

***all_reachable_states :***

On utilise la foncton ***List.map*** sur la list des moves pour appliquer le move sur le state donné.

***compare_state :***

Cette fonction compare deux states donnés en arguments et retourne true s'ils sont identiques. On d'aboard transmet les states en matrices et les compares pour chaque case s'ils sont identiques. Sinon, on lance l'exception ***Unequal*** et le programme se termine.

***check_state_occured :***

Cette fonction prend un state et une liste de states. Et elle renvoie true si la liste contient ce state.

***solve :***

C'est une fonction récursive. Elle prend comme argument une Queue « ***q*** » qui contient les states à tester et une liste de state « ***all*** » qui contient tous les states sont déjà présents avant. On prend la tête de la queue (c'est possible d'avoir une exception ***Queue.Empty***) et teste s'il a gagné :

- Si oui, on lance l'exception ***Solution_trouve*** avec les moves effectués.
- Sinon, on calcule pour l'état actuel les moves possibles et les states correspondants (on les appelle « ***states fils*** » ici. ). Pour chaque state fils, on regarde s'il est déjà présent dans les étapes précédentes, c'est-à-dire on teste s'il est présenté dans la liste « ***all*** » :
  - Si c'est vrai, on l'ignore pour éviter de traiter les cas qu'on a déjà traité.
  - Sinon, on l'ajoute dans la queue « ***q*** » et la liste « ***all*** » et on continue la recherche en appelant récursivement cette fonction.

Si on reçoit l'exception ***Queue.Empty***, c'est-à-dire il y a aucun movement qu'on peut tester. Donc, on retourne « Problème insolvable ! »
Si on reçoit l'exception ***Solution_trouve***, on retourne la solution (ici, c'est le message d'erreur).

***solve_state :***

On crée la queue et ajoute l'état initial dedans. Après on appelle la fonction ***solve_state***.

***solve_input :***

Pour le ***in_channel*** donné en argument, d'abord on fait créer le state correpondant en appelant la fonction ***input_state***. Après, on appelle ***solve_state*** avec ce state. On retourne la solution trouvée. Dans le cas insolvable, on retourne 'Probleme insolvable'.

# Complexité :

Au pire cas le bateau A est à la position(0,2), et on suppose qu'il y a ***m*** bateaux dans la port. Pour aller au (4,2). Il faut minimum 4 moves. Donc il y aura $4^m$ solutions. Donc on suppose que $n = 4^m$, la complexité est $O(n^2)$.

# Annexe :

Ce qu'on a écrit dans le fichier solver.ml est le programme pour tester les fichiers donnés.(voir figure1)

```
let ()=
  let j=(open_in "./tests-orush/tests/pos1.txt") in
  let res=solve_input j in
  print_string res;
  print_newline ()
```

figure1

Les résultats sont comme suit : (voir figure2)

resultat.txt

| name | time | nb_state_test | solution |
|---|---|---|---|
| pos1 | 0m0,017s | 5 | A>A>A> |
| pos2 | 0m0,033s | 69 | A>B>C<A>A> |
| pos3 | 0m0,021s | 22 | A>A>B<A> |
| pos4 | 0m0,028s | 52 | A>A>A>C<B>B>B>A> |
| pos5 | 0m26,317s | 3867 | B>A<B>C<D<D<D<F<F<E>E>F<D>D>D>A>G>E>A>A>D<D<D<F>B<B<B<F<D>C>B<D>D>A<A<A<D<E<G<G<G<E>G<D>A>A>A>H>H>A> |
| pos6 | 7m42,847s | 11013 | E>E>F<G>A<D>C>A<A<E<E<E<A>I>N<M<N<L<L<N<I<F>C>F>F>I<D>D>N<A>A>A> |
| pos7 | 0m29,984s | 2713 | B>C<E<F>H<H<G>H<F>I<I<G>I<F>A>A>K>A> |
| pos8 | 0m0,445s | 430 | B>B>B>A<C<C<C<A>B<B<B<D<D<F<F<F<E>E>E>A>A>A> |
| pos9 | 0m58,069s | 3583 | E>D>C>F<I<J<K>H>L<L<K>M>H>I<A>G<J<K>A>M>M>A> |
| pos10 | 23m16,531s | 20166 | A<I>B>C<A<G<I>E>J>B>K<K<J>E>G<D>M>B>G<A>C>C>C>A<C>D<G>B<B<G>E<E<E<G<D>C<I<D>F<H<H<J<J<D>G>E>K>G>A>C< |
| | | | C<C<G>K>M<D>I>E>I>A>F<F<I>D<D<D<D<G<I<J>B>F<A<A<G<I<K<K<J>K<I>K<G>A>A>L<L<J>A>M>M>M>A> |
| pos11 | 0m18,420s | 2367 | G>C>C>D<D<E<E<H<H<G>H<E>A>A>A>I<J>A> |
| pos12 | 0m2,954s | 775 | A>B<B<B<D<G>G>A>C>E<H<I<J>A>K<K<A> |

figure2