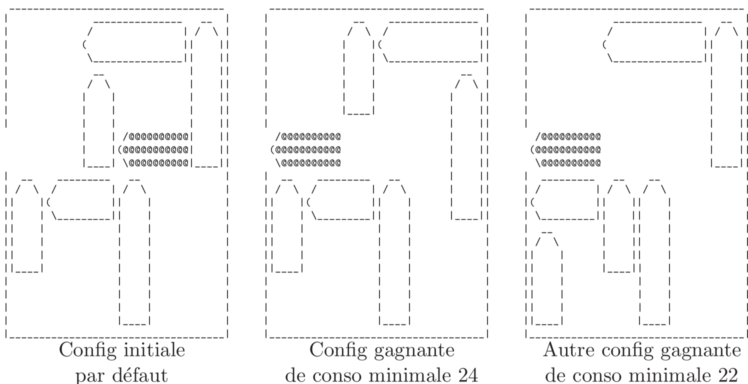
TPL0 : Introduction à “*Embouteillage*”

Un jeu de taquin à réaliser dans les TPs en temps libre d'Algo1

“Embouteillage” est un jeu de taquin de 6 lignes par 6 colonnes, dont les pièces représentent des voitures (pièces de taille 2) ou des camions (pièces de taille 3) prises dans un embouteillage sur un parking. Comme dans un taquin classique, on ne peut déplacer les pièces que par des glissements horizontaux ou verticaux dans un espace libre, à ceci près qu'un rail empêche les pièces de se déplacer latéralement : elles ne peuvent se déplacer que suivant l'axe principal du “véhicule” (en avançant ou en reculant). Le but du jeu est de placer la voiture horizontale remplie ici avec des arrobases (caractère `@`) en face de la sortie : le trou à gauche du plateau sur la troisième ligne.



Config initiale par défaut — Config gagnante de conso minimale 24 — Autre config gagnante de conso minimale 22

Fig. 1 – Exemples de configurations

Pour pimenter le jeu, on associe par ailleurs un calcul de consommation d'essence pour chaque déplacement de véhicule. Ce calcul est donné par une fonction $c(t,d)$ où t est la taille du véhicule (2 ou 3) et d est la distance parcourue lors du déplacement (un entier entre 0 et “$6-t$”). Par défaut,

$$c(t,d) = t \times (1+d)$$

Ci-dessus, la constante 1 ajoutée à d représente ici le coût de démarrage du véhicule. Par exemple, à partir de la configuration initiale par défaut, la configuration au milieu ci-dessus peut être obtenue en 4 déplacements avec une consommation totale minimale de 24 :

$$c(3,1) + c(3,1) + c(2,1) + c(2,3) = 24$$

On peut trouver une autre configuration gagnante (à droite ci-dessus) en 4 déplacements avec une consommation totale de seulement 22 :

$$c(2,1) + c(2,1) + c(2,2) + c(2,3) = 22$$

Essayer de minimiser la consommation permet donc de continuer à s'amuser même pour des configurations dont on a déjà trouvé une solution.

`?`	affiche l'aide	`d`	défait le dernier coup
`q`	quitte le jeu	`e`	enregistre l'état actuel du jeu
`a`	affiche l'état du plateau	`r`	restaure l'état enregistré
`n`	nouvelle partie	`j...`	joue le coup considéré
`l`	liste des coups actuellement possibles		

Fig. 2 – Commandes du programme `semba`

Ce jeu va servir de thème aux TP en temps libre du semestre. Le but du TPL1 est de construire un jeu interactif de simulation du taquin en ligne de commande. Auparavant, dans les exos P12 et P13 des TPs encadrés, on va construire un paquetage appelé `Analyseur` pour initialiser le plateau de jeu dans une configuration écrite dans un fichier texte. Enfin, le TPL2 consiste à construire un programme qui calcule s'il existe une solution à ce casse-tête, et qui dans ce cas, affiche une solution de consommation minimale.

Le but de ce document est de vous présenter le jeu à réaliser au TPL1 et le paquetage `Analyseur` à réaliser dans les TPs encadrés. Pour réaliser ce paquetage `Analyseur`, il suffit d'avoir bien compris la section 2 : on peut se contenter d'un rapide survol du reste du document.

1 Jouer à “Embouteillage” en ligne de commande

Le programme de jeu en ligne de commande s'appelle `semba`, acronyme de “*Simulation (du taquin) EMBouteillage en Ascii*”. Il commence par demander à l'utilisateur le nom du fichier de configuration initiale du plateau. La syntaxe de ce fichier de configuration est décrite en section 2.

```
Fichier de "Configs/" contenant la config initiale [n04.emb]:
```

Si l'utilisateur se contente ici de taper “*entrée*” (un nom de fichier vide), alors le programme va initialiser le plateau à partir du fichier `n04.emb` situé dans le répertoire `Configs/`. Alternativement, l'utilisateur peut donner le nom d'un autre fichier de ce répertoire en utilisant par exemple une des nombreuses *configurations initiales fournies* [sur le kiosk] par les enseignants. En changeant la configuration initiale, ces fichiers permettent d'avoir différents niveaux de jeu : `n04.emb` correspond à la configuration d'initiation au jeu présentée en introduction. Généralement, les enseignants ont donné des noms comme “`n`X`.emb`” où X est un numéro qui représente le nombre minimal de coups vers une configuration gagnante (attention, ce nombre minimal de coups n'est pas forcément synonyme de consommation minimale). Le but du jeu est toujours le même : placer la voiture horizontale de la 3-ième ligne en face de la sortie. Précisons ici que **toutes les configurations initiales font en sorte qu'il n'y ait aucun autre véhicule horizontal sur cette ligne.**

Une fois que le plateau est initialisé, le programme affiche la configuration courante du plateau sur la sortie standard, puis la liste de déplacements possibles à partir de cette configuration. Il attend ensuite une commande de l'utilisateur sur l'entrée standard (voir l'ensemble des commandes possibles figure 2). En fonction de cette commande, il effectue une certaine action : par exemple, il déplace une pièce. Il affiche ensuite la nouvelle configuration, etc. Le jeu s'arrête lorsque l'utilisateur tape la commande `q`. Voir l'exemple de partie figure 3 page 4.

Lorsque dans la configuration courante, la voiture “pleine” se trouve en face de la sortie, l'utilisateur a gagné et le programme affiche :

```
Bravo, coup gagnant en 1871 coups !
```

où “1871” est ici le nombre de coups joués par l'utilisateur pour arriver dans cette configuration.

1.1 Les coups, une notation pour les déplacements

On donne un nom à chaque case du plateau sous la forme d'un nom de ligne compris entre `A` et `F` suivi d'un numéro de colonne entre `0` et `5`. On définit alors la *position* d'une pièce comme le nom de la case située dans le coin supérieure de la pièce. Pour rendre la lecture plus facile, `semba` affiche le nom de cette case entre 2 délimiteurs : "`>`" à gauche et "`<`" à droite.

```
 ------------------------------------
|                                    |
|              _______________   __  |
|             /               | /  \ |
|  A0    A1  (>A2<   A3    A4 ||>A5<||
|             \_______________||    ||
|              __              |    ||
|             /  \             |    ||
|  B0    B1  |>B2<|  B3    B4  | B5 ||
|            |    |            |    ||
|            |    |            |    ||
             |    | /@@@@@@@@@@|    ||
   C0    C1  | C2 |(>C3<@@@C4@@| C5 ||
             |____| \@@@@@@@@@@|____||
|  __    _________   __              |
| /  \  /         | /  \             |
||>D0<|(>D1<   D2 ||>D3<|  D4    D5  |
||    | \_________||    |            |
||    |            |    |            |
||    |            |    |            |
|| E0 |  E1    E2  | E3 |  E4    E5  |
||____|            |    |            |
|                  |    |            |
|                  |    |            |
|  F0    F1    F2  | F3 |  F4    F5  |
|                  |____|            |
|____________________________________|
```

Dans la suite, on désigne par "*coup*" le nom d'un déplacement d'une pièce. Un coup est formé de la position initiale de la pièce suivi du nom de la coordonnée qui change dans ce déplacement. Par exemple, pour la voiture de la configuration initiale située en $D0$ les coups possibles sont : $D0A$, $D0B$, $D0C$ et $D0E$. Pour le camion situé en $A2$, les coups possibles sont : $A20$ et $A21$. Voir les exemples de la figure 3.

1.2 Défaire le dernier coup

Lorsqu'on vient de jouer un coup, la commande `d` permet d'annuler le coup qui vient d'être joué. On revient alors dans l'avant-dernière configuration (en décrémentant le nombre de coups joués et la consommation totale). La commande `d` n'a pas de sens lors du commencement du jeu (avant le premier coup) ou lorsqu'on vient de faire une des commandes `n`, `r` ou `d`.

1.3 Enregistrer/Restaurer l'état courant

La commande `e` permet d'enregistrer la configuration courante du plateau ainsi que la consommation totale et le nombre des coups qui ont été joués pour arriver à cette configuration. Cette configuration peut être restaurée à n'importe quel moment du jeu via la commande `r`. Le plateau se retrouve ainsi dans la configuration enregistrée, avec la consommation totale et le nombre de coups correspond aux valeurs enregistrées.

Par ailleurs, lorsque la configuration courante correspond à la configuration enregistrée, le message suivant s'affiche au-dessus de la liste des coups valides :

```
Info: l'état actuel du plateau est l'état enregistré
```

Ce message indique en particulier à l'utilisateur qu'il est déjà passé par là ! Au commencement d'une partie, l'état enregistré est positionné sur la configuration initiale, de sorte que tant qu'on n'a pas fait la commande `e`, alors la commande `r` permet de revenir à la configuration initiale.

Lorsqu'il lance "`semba`" dans la configuration initiale `n04.emb`, l'utilisateur voit apparaître :

```
Fichier de "Configs/" contenant la config initiale [n04.emb]:

 ------------------------------------
|                                    |
|              _______________   __  |
|             /               | /  \ |
|  A0    A1  (>A2<   A3    A4 ||>A5<||
|             \_______________||    ||
|              __              |    ||
|             /  \             |    ||
|  B0    B1  |>B2<|  B3    B4  | B5 ||
|            |    |            |    ||
|            |    |            |    ||
             |    | /@@@@@@@@@@|    ||
   C0    C1  | C2 |(>C3<@@@C4@@| C5 ||
             |____| \@@@@@@@@@@|____||
|  __    _________   __              |
| /  \  /         | /  \             |
||>D0<|(>D1<   D2 ||>D3<|  D4    D5  |
||    | \_________||    |            |
||    |            |    |            |
||    |            |    |            |
|| E0 |  E1    E2  | E3 |  E4    E5  |
||____|            |    |            |
|                  |    |            |
|                  |    |            |
|  F0    F1    F2  | F3 |  F4    F5  |
|                  |____|            |
|____________________________________|
Nom de la config initiale:Configs/n04.emb
Consommation totale: 0
Info: l'état actuel du plateau est l'état enregistré
 1 :
```

La dernière ligne "`1 :`" est l'invite de la ligne de commande indiquant à l'utilisateur qu'il doit saisir une commande. Le nombre avant les deux-points indique le numéro du coup (depuis la configuration initiale) que l'utilisateur s'apprête à jouer. Si l'utilisateur veut jouer le coup `A20`, il tape `jA20` et obtient :

```
j(oue) A20
 ------------------------------------
|  _______________               __  |
| /               |            /  \ |
|(>A0<   A1    A2 |  A3    A4  |>A5<||
| \_______________|            |    ||
|              __              |    ||
|             /  \             |    ||
|  B0    B1  |>B2<|  B3    B4  | B5 ||
|            |    |            |    ||
|            |    |            |    ||
             |    | /@@@@@@@@@@|    ||
   C0    C1  | C2 |(>C3<@@@C4@@| C5 ||
             |____| \@@@@@@@@@@|____||
|  __    _________   __              |
| /  \  /         | /  \             |
||>D0<|(>D1<   D2 ||>D3<|  D4    D5  |
||    | \_________||    |            |
||    |            |    |            |
||    |            |    |            |
|| E0 |  E1    E2  | E3 |  E4    E5  |
||____|            |    |            |
|                  |    |            |
|                  |    |            |
|  F0    F1    F2  | F3 |  F4    F5  |
|                  |____|            |
|____________________________________|
Nom de la config initiale:Configs/n04.emb
Consommation totale: 9
 2 :
```

En tapant `jA03`, il obtient

```
ERREUR: Coup impossible
 2 :
```

En tapant `l`, il obtient

```
"jA01" "jA02" "jA5B" "jA5C" "jA5D" "jD0C" "jD0B"
"jD0E"
 2:
```

En tapant `jA5D`, il obtient

```
j(oue) A5D
 ------------------------------------
|  _______________                   |
| /               |                  |
|(>A0<   A1    A2 |  A3    A4    A5  |
| \_______________|                  |
|              __                    |
|             /  \                   |
|  B0    B1  |>B2<|  B3    B4    B5  |
|            |    |                  |
|            |    |                  |
             |    | /@@@@@@@@@@      |
   C0    C1  | C2 |(>C3<@@@C4@@  C5  |
             |____| \@@@@@@@@@@      |
|  __    _________   __          __  |
| /  \  /         | /  \        /  \ |
||>D0<|(>D1<   D2 ||>D3<|  D4  |>D5<||
||    | \_________||    |      |    ||
||    |            |    |      |    ||
||    |            |    |      |    ||
|| E0 |  E1    E2  | E3 |  E4  | E5 ||
||____|            |    |      |    ||
|                  |    |      |    ||
|                  |    |      |    ||
|  F0    F1    F2  | F3 |  F4  | F5 ||
|                  |____|      |____||
|____________________________________|
Nom de la config initiale:Configs/n04.emb
Consommation totale: 21
 3 :
```

En tapant `jA03`, il obtient

```
j(oue) A03
 ------------------------------------
|                    _______________ |
|                   /               ||
|  A0    A1    A2  (>A3<   A4    A5 ||
|                   \_______________||
|              __                    |
|             /  \                   |
|  B0    B1  |>B2<|  B3    B4    B5  |
|            |    |                  |
|            |    |                  |
             |    | /@@@@@@@@@@      |
   C0    C1  | C2 |(>C3<@@@C4@@  C5  |
             |____| \@@@@@@@@@@      |
|  __    _________   __          __  |
| /  \  /         | /  \        /  \ |
||>D0<|(>D1<   D2 ||>D3<|  D4  |>D5<||
||    | \_________||    |      |    ||
||    |            |    |      |    ||
||    |            |    |      |    ||
|| E0 |  E1    E2  | E3 |  E4  | E5 ||
||____|            |    |      |    ||
|                  |    |      |    ||
|                  |    |      |    ||
|  F0    F1    F2  | F3 |  F4  | F5 ||
|                  |____|      |____||
|____________________________________|
Nom de la config initiale:Configs/n04.emb
Consommation totale: 33
 4 :
```

En tapant `d`, il revient à la configuration antérieure avec une consommation totale de 21 (et un nombre de coups valant 3). En tapant `n`, il peut recommencer une nouvelle partie avec éventuellement une autre configuration. En tapant `q`, il quitte le jeu.

Remarquons qu'il est possible d'enregistrer dans un fichier texte les commandes "`semba`" tapées au clavier lors de cette partie. Par exemple, le fichier "`demo.cse`" permet de rejouer cette partie via la ligne de commande "`./semba < demo.cse`".

FIG. 3 – Extrait d'une partie

2 Initialiser le plateau depuis un fichier

À son démarrage ou lors de la commande `n`, le programme `semba` lit la configuration initiale depuis un certain fichier texte. Cela permet en particulier d'avoir différents niveaux de jeu : voir les *configurations initiales fournies* par les enseignants. On décrit ici ce format de fichier. Par convention, le nom du fichier se termine en "`.emb`" (pour "*embouteillage*").

2.1 Format du fichier de configuration

Dans le fichier de configuration, les différentes formes de pièces sont identifiées par une des 4 lettres suivantes :

- la lettre minuscule `v` correspond à une voiture horizontale.
- la lettre majuscule `V` correspond à une voiture verticale.
- la lettre minuscule `c` correspond à un camion horizontal.
- la lettre majuscule `C` correspond à un camion vertical.

Chaque pièce est ainsi désignée par la lettre correspondant à sa forme et par sa position (c'est-à-dire celle de son coin supérieur gauche). La voiture à sortir du plateau n'est pas désignée de façon spéciale : elle est en principe le seul véhicule horizontal qui figure en ligne de nom `C`. Par exemple, le fichier de configuration "`n04.emb`" est donné ci-dessus :

```
            c A2              C A5
            V B2
                  v C3
V D0  v D1        C D3
```

Les règles concernant le format de fichier sont les suivantes :

- Le caractère d'espacement et le caractère de passage à la ligne sont considérés comme des séparateurs.
- Il peut y avoir un nombre quelconque de séparateurs en début ou en fin de fichier (ce nombre pouvant être 0).
- Chaque pièce est donnée par le nom de sa forme, suivi d'un nombre quelconque de séparateurs, suivi du nom de ligne, suivi d'un nombre quelconque de séparateurs, suivi du numéro de colonne.
- Deux pièces apparaissant successivement dans le fichier doivent être séparées par au moins un séparateur (mais il peut y en avoir plus).
- Les caractères autres que les séparateurs et ceux apparaissant dans la composition des pièces ne sont pas admis.

L'ordre dans lequel sont donnés les pièces n'a pas d'importance. Par convention, on profite de la possibilité d'insérer des espaces blancs et sauter des lignes pour donner une forme au fichier de configuration où chaque pièce est dans une position correspondant à celle du plateau. Ça rend la visualisation plus facile (voir les fichiers fournis par les enseignants).

2.2 Paquetage `Analyseur` : une machine séquentielle de modèle 1

Le paquetage chargé de la lecture du fichier de configuration s'appelle `Analyseur`. C'est une machine séquentielle avec sentinelle de fin (modèle 1) qui énumère la suite des pièces écrites dans le fichier. Il sera donc utilisé lors du TPL1 pour initialiser le plateau conformément au fichier de configuration.

Ce paquetage `Analyseur` utilise un autre paquetage appelé `ParamJeu`, qui définit des types de données aussi utilisés dans le TPL1. Ce paquetage `ParamJeu` est fourni par les enseignants : son

interface est `paramjeu.ads` [sur le kiosk] et son implémentation est `paramjeu.adb` [sur le kiosk]. Il donne un type `Piece` pour représenter les pièces :

```
type Vehicule is (Voiture, Camion) ;

type Direction is (Horizontal, Vertical) ;

type Piece is record
   Nat: Vehicule ; -- nature du vehicule
   Dir: Direction ;
end record ;
```

La position d'une pièce est une valeur de type `Position` :

```
subtype Numero is Integer range 0..5 ;

type Position is record
   Lig, Col: Numero ;
end record ;
```

Le numéro associé à un nom de ligne ou de colonne et les numéros est donnée par les fonctions `NomLigNum` et `NomColNum` ci-dessous :

```
subtype NomLig is Character range 'A'..'F' ;
subtype NomCol is Character range '0'..'5' ;

function NomLigNum(C: NomLig) return Numero is
begin
   return Character'Pos(C)-Character'Pos(NomLig'First) ;
end ;

function NomColNum(C: NomCol) return Numero is
begin
   return Character'Pos(C)-Character'Pos(NomCol'First) ;
end ;
```

Voilà maintenant la spécification du paquetage `Analyseur` à construire :

```
-- Machine Séquentielle de modèle 1, pour lire la configuration
-- du taquin depuis un fichier.

-- L'implémentation de ce paquetage ferme les fichiers qui ne sont
-- plus utilisés automatiquement.  Par contre, on ne peut analyser
-- qu'un seul fichier à la fois.

with ParamJeu ; use ParamJeu ;

package Analyseur is

   ErreurLecture: exception ;

   procedure Demarrer(NomFichier: String) ;
   -- ouvre le fichier NomFichier
   --   et positionne la tête de lecture sur
   --   le 1er élément à énumérer.
   -- ErreurLecture levée garantit une pièce incorrecte.
   -- Name_Error ou Use_Error levée si impossible d'accéder
```

```
   --   au fichier en lecture.
   --

   function Fin return Boolean ;
   -- retourne False ssi il y a encore une position à lire.

   procedure Avancer ;
   -- requiert not Fin
   -- positionne la tête de lecture sur le prochain élément
   --    à énumérer.
   -- ErreurLecture levée garantit une pièce incorrecte.

   function PieceCourante return Piece ;
   -- requiert not Fin
   -- retourne la pièce sous la tête de lecture

   function PositionCourante return Position ;
   -- requiert not Fin
   -- retourne la position sous la tête de lecture.

   function LigErreurLecture return Natural ;
   -- num de ligne du caractère provoquant l'erreur de lecture

   function ColErreurLecture return Natural ;
   -- num de colonne du caractère provoquant l'erreur de lecture

end Analyseur ;
```

Quand l'analyseur tombe sur un fichier qui ne respecte pas le format décrit section 2.1, il lève une exception appelée `ErreurLecture` (qui peut donc survenir soit lors d'un `Demarrer`, soit lors d'un `Avancer`). `Demarrer` peut aussi lever d'autres erreurs si le fichier est inaccessible en lecture. Lors des cas d'erreurs, on peut savoir la position du caractère provoquant l'erreur dans le fichier.

Précisons ici que l'analyseur doit se contenter de vérifier le respect du format décrit section 2.1. En particulier, il vérifie que :

- le nom des pièces appartient bien au type `'c'`, `'C'`, `'v'`, `'V'`.
- les noms des lignes sont dans `NomLig`, et les colonnes dans `NomCol`.

Mais, il ne vérifie pas que :

- les pièces rentrent effectivement sur le plateau (par exemple `cD5` ne sera pas rejeté au niveau de l'analyseur).
- les pièces ne se chevauchent pas (par exemple, "`cA0 cA1`" ne sera pas rejeté).
- il y a un unique véhicule horizontale en ligne `C` qui est une voiture.

Ce type de vérification sera pris en compte lors du TPL1, au moment où l'on tente de poser effectivement les pièces sur le plateau.

Une utilisation typique de l'analyseur est donnée par le programme ci-dessous. Ce programme tente de lire un fichier de configuration (dont le nom est lu sur la ligne de commande) et d'afficher les pièces reconnues sur la sortie standard. Si le fichier ne respecte pas le format des fichiers de configuration, le programme s'arrête en affichant un message indiquant la position de l'erreur dans le fichier.

```
begin
   Demarrer(Argument(1)) ;
   while not Fin loop
```



```
      AfficheNomPiece(PieceCourante) ;
      AffichePosition(PositionCourante) ;
      New_Line ;
      Avancer ;
   end loop ;
exception
   when ErreurLecture =>
      Put("Configuration non reconnue ") ;
      Put_Line("ligne " & Positive'Image(LigErreurLecture)
               & ", colonne " & Positive'Image(ColErreurLecture)) ;
end ;
```

Il utilise aussi les procédures d'affichage suivante qui sont définies dans le paquetage `ParamJeu` :

```
procedure AfficheNomPiece(Obj: Piece) ;
procedure AffichePosition(Pos: Position) ;
```

2.3 Tester `Analyseur`

On vous demande d'écrire des fichiers de configuration pour tester le paquetage `Analyseur`. Il faut écrire un ensemble de tests qui permette de détecter si un aspect de la spécification du paquetage n'a pas été correctement pris en compte. Il s'agit ici de tester si le programme lit correctement les fichiers de configuration qu'il est censé lire, et s'il rejette correctement les fichiers incorrects avec le bon numéro de ligne et de colonne. Vous devez donc écrire deux types de fichiers de configuration.

2.3.1 Fichiers corrects

Ces fichiers de configuration doivent être acceptés par l'analyseur. Il faut vérifier :

- la bonne gestion des séparateurs entre pièces et du nombre de pièces : tester relativement exhaustivement l'ensemble des cas possibles avec un petit nombre de pièces entre 0 et 3 par exemple (il faut ici en particulier tester les *cas limites*, c-à-d. ici avec un minimum de séparateurs, car il y a souvent des erreurs sur les cas limites) ;
- la bonne reconnaissance des pièces : écrire un fichier correct qui permet de vérifier la bonne reconnaissance de l'ensemble de noms de pièces, l'ensemble des numéros de lignes, l'ensemble des numéros de colonne ;
- la bonne gestion des séparateurs à l'intérieur des pièces.

On adoptera une convention de nommage des fichiers du genre "`ok_bidule_n.emb`" où "`ok`" indique que le fichier doit être accepté par l'analyseur, "*bidule*" indique ce qu'on vérifie avec ce fichier, et "*n*" indique le nombre de pièces dans le fichier. Cela permettra d'avoir rapidement une idée du comportement attendu de l'analyseur sur ce fichier via le *pilote de tests* (voir sous-section 2.3.3).

2.3.2 Fichiers incorrects

Ces fichiers de configuration doivent être rejetés par l'analyseur. Il faut partir de quelques fichiers corrects bien choisis, et injecter différentes erreurs en différents endroits pour couvrir l'ensemble des cas possibles d'erreurs : numéro de colonne trop grand, mauvais nom de pièce, etc.

On doit évidemment faire un fichier distinct pour chaque erreur. On adoptera une convention de nommage des fichiers du genre "`ko_n_m.emb`" où "`ko`" indique que le fichier doit être rejeté par l'analyseur, "`n`" le numéro de ligne où se trouve l'erreur et "`m`" le numéro de colonne.

2.3.3 Pilote de tests

Il faut aussi écrire un pilote de tests appelé `testconfig` permettant de savoir comment se comporte le paquetage `Analyseur` pour les fichiers de configuration. On pourra prévoir trois formes d'utilisations :

avec un seul fichier "`testconfig fich.emb`" affiche ligne à ligne les pièces reconnues par l'analyseur, tout en respectant le format des fichiers de configurations. Si le fichier en paramètre de `testconfig` est incorrect, alors on doit obtenir un message d'erreur avec le numéro de ligne et le numéro de colonne. Par exemple, si le fichier `ok_pieces_6.emb` est un fichier correct qui a lui-même été écrit dans le style affiché par `testconfig`, alors pour vérifier que `ok_pieces_6.emb` est bien reconnu par l'analyseur, il suffit de vérifier que l'affichage produit par "`testconfig ok_pieces_6.emb`" est identique au contenu de `ok_pieces_6.emb`. Ceci est vérifié automatiquement par le programme `diff` dans la ligne de commande suivante :

```
./testconfig ok_pieces_6.emb | diff - ok_pieces_6.emb
```

Si `diff` ne détecte aucune différence, il n'affiche rien. Sinon, il affiche une liste de lignes qui diffèrent.

avec plusieurs fichiers "`testconfig fich1.emb fich2.emb ...`" affiche un résumé de l'analyse pour chaque fichier de la ligne de commande. Sur les fichiers corrects, il affiche "OK" suivi du nombre de pièces détectées. Sur les fichiers incorrects, il affiche "KO" suivi du numéro de ligne et de colonne de l'erreur. Par exemple, pour la ligne de commande :

```
./testconfig ok*_?.emb ko*4*.emb
```

on obtient quelque chose du genre :

```
**** Fichier ok_pieces_6.emb
  => OK: 6 pieces trouvees
**** Fichier ok_sepinterne_5.emb
  => OK: 5 pieces trouvees
**** Fichier ko_1_4.emb
  => KO: Configuration non reconnue ligne 1, colonne 4
**** Fichier ko_2_4.emb
  => KO: Configuration non reconnue ligne 2, colonne 4
**** Fichier ko_4_3.emb
  => KO: Configuration non reconnue ligne 4, colonne 3
```

Un résultat différent signifierait ici que l'analyseur a un bogue ou que le nom des fichiers ne respecte pas les conventions de nommage décrites ci-dessus.

avec plusieurs fichiers en mode debug "`testconfig -d fich1.emb fich2.emb ...`" fait comme la commande précédente, mais en affichant aussi les pièces reconnues pour chacun des fichiers.

TPL1 : simulation du taquin "Embouteillage"

La description du jeu de taquin en ligne de commandes qu'on cherche à réaliser ce TP est donnée dans le document TPL0 des TPs en temps libre. On complète ici le document TPL0 en donnant des instructions sur la façon d'implémenter le programme attendu.

1 Travail à faire

Pour ce TP, on vous fournit essentiellement trois fichiers : `taquin.ads` [sur le kiosk] contient la spécification des procédures de manipulation du plateau de jeu, `taquin.adb` [sur le kiosk] qui doit contenir l'implémentation de ces procédures, et `semba.adb` [sur le kiosk] qui contient l'implémentation de l'interface pour jouer au taquin. Le programme `semba.adb` utilise en particulier les procédures définies dans le paquetage `Taquin`. Les fichiers `semba.adb` et `taquin.ads` sont achevés, mais vous devez compléter `taquin.adb`. En fait, il y a aussi un quatrième fichier fourni `taquin-affiche.adb` [sur le kiosk] qui contient l'implémentation d'une procédure du paquetage `Taquin` qui est elle aussi achevée.

Par ailleurs, il faut aussi utiliser le paquetage `ParamJeu` du TPL0 et le paquetage `Analyseur` que vous avez normalement débogué à l'exo P12. Lorsque celui-ci fonctionne encore imparfaitement, il est possible s'en passer : on initialisera le taquin dans une configuration par défaut directement codée dans le programme (voir section 3). Mais il est plutôt conseillé de passer du temps à déboguer complètement `Analyseur` : vous gagnerez ensuite du temps pour tester et déboguer votre paquetage `Taquin` !

Lors du prochain TP, les procédures de `Taquin` sont destinées à être utilisées pour le calcul d'une solution. Le programme `semba` pourra aussi être utilisé pour tester que le résultat obtenu est bien une solution correcte.

Comme il est important de tester et corriger vos fonctions et procédures **au fur et à mesure** que vous les écrivez, on vous fournit un exemple de programme `testtaquin.adb` [sur le kiosk] qui permet de tester l'initialisation du taquin. Vous **devez** copier ce fichier et l'adapter pour tester et corriger vos autres sous-programmes. Les tests à réaliser avec le programme `semba` sont à réserver quand vos procédures et vos fonctions fonctionnent bien pour des tests simples. On donne des indication en section 2 sur la façon de réaliser les tests avec `semba`.

Pour réaliser le TP, traitez-les sous-tâches de l'énoncé dans l'ordre. Ne passez à la sous-tâche suivante que quand la réalisation de la sous-tâche courante est achevée. En particulier, les procédures et fonctions demandées doivent avoir été testées et corrigées. Les correcteurs privilégieront très largement "*le code de qualité*" à "*la quantité de code*". Il n'est en particulier pas nécessaire d'avoir réalisé toutes les sous-tâches pour avoir une note honorable.

Barème indicatif

Correction du code (respect des specs)	4 pts
Lisibilité du code (qualité des commentaires, choix des identificateurs, etc)	3 pts
Élégance du code (simplicité, concision)	3 pts
Validation du code (démarche de tests)	6 pts
Rédaction du compte-rendu	4 pts

Chaque équipe doit réaliser un *compte-rendu de TP* (ou CR). Ce doit être un document *synthétique* de **5 pages au maximum** qui doit *aider* les enseignants à évaluer *rapidement* le travail effectué.[1] Il doit contenir :

[1] Au cours de votre carrière future, vous aurez souvent à rédiger ce type de document devant faciliter la réutilisation ou l'évaluation de vos réalisations techniques.

1. une description (3 pages maximum) de l'implémentation explicitant :
   - les points réalisés par rapport au sujet, points non réalisés, bogues ou problèmes connus (non corrigés), précisions par rapport aux ambiguïtés ou aux incohérences du sujet.
   - les choix de conception *additionnels* par rapport au sujet (procédures auxiliaires, adaptation des structures de données, invariants sur les données, etc).

   il ne faut donc pas inclure de code source dans le CR sauf exception.
2. une description (3 pages au maximum) de la démarche de validation :
   - description des pilotes de tests éventuels.
   - classification et description des tests avec leur objectif (c-a-d. comportement attendu du programme & description de ce qui est testé) et éventuellement une description du comportement obtenu du programme lorsque celui-ci diffère du comportement attendu (cas d'un bogue non corrigé).

Les étudiants devront imprimer ce CR ainsi que le code source des programmes à réaliser, et déposer le tout dans le casier du groupe de TD d'un membre de l'équipe (près de la salle E001 au rez-de-chaussée de l'Ensimag) **avant le vendredi 19 novembre à 14h**. Si tous les membres de l'équipe n'appartiennent pas au même groupe de TD, ils déposent leurs documents dans **un seul** des casiers. Merci d'indiquer au début de chaque document, le numéro d'équipe sur `Teide`, puis le nom et le numéro de groupe de chacun des membres de l'équipe. Par ailleurs, chaque équipe doit aussi déposer sur `Teide` (avant cette même date) ses sources Ada et ses fichiers de tests.

L'évaluation des enseignants se basera principalement sur les documents papiers. Les fichiers électroniques déposés sur `Teide` pourront être utilisés pour contrôler la réalité du travail présenté, lancer des programmes de contrôle anti-plagiat, etc. Les incohérences entre la version papier et la version électronique seront sanctionnées. Les CR mal rédigés seront sanctionnés au titre de *la double peine* : d'une part sur la note de rédaction proprement dite, et d'autre part à cause du fait que les enseignants (qui n'ont pas beaucoup de temps à consacrer aux corrections) seront conduits à sous-noter le travail réalisé pour les autres critères du barème. La double peine s'appliquera aussi au code source peu lisible (absence de commentaires pertinents, mauvaise indentation, etc).

2 Tests avec `semba`

Les tests à réaliser avec le programme `semba` sont à effectuer quand vos procédures et vos fonctions fonctionnent bien pour des tests simples. On vous demande de rendre (sur `Teide`) les tests effectués avec `semba`. Il faut pour cela copier-coller dans un fichier, les commandes que vous tapez interactivement dans `semba` (mettre une commande par ligne dans le fichier). Si vous appelez un des ces fichier "`toto.cse`", on pourra rejouer facilement ce test en tapant :

```
./semba < toto.cse
```

Par ailleurs, il faut préférer faire plusieurs tests courts à partir de fichiers de configurations bien choisies, qu'un seul gros test.

Un test pour `semba` sera donc typiquement constitué d'un fichier de configuration "`toto.emb`" et d'un fichier "`toto.cse`". La première ligne du fichier "`toto.cse`" sera donc typiquement "`toto.emb`" pour initialiser le taquin avec ce fichier de configuration, et la dernière ligne sera "`q`" pour quitter proprement le programme. Il faudra donc **rendre** à la fois "`toto.cse`" mais aussi "`toto.emb`". Il faut faciliter la visualisation de vos fichiers "`.emb`" en respectant les conventions données en section TPL0.2.

Au cours du développement, c'est une bonne idée de faire des tests de non-régression (vérifier que les tests qui passaient, continuent de le faire). Pour cela, quand vous êtes contents du résultat pour le test `toto.cse`, vous sauvergardez le résultat dans un fichier `toto.out` :

```
./semba < toto.cse > toto.out
```

Lorsque vous rejouez le test, vous pouvez facilement détecter les différences entre le résultat attendu et le résultat obtenu via le programme `diff` :

```
./semba < toto.cse | diff toto.out -
```

Il est aussi bienvenu de rendre `toto.out` dans l'archive `Teide`.

Pensez à décrire dans le compte-rendu ce que vous cherchez à mettre en évidence par le couple "`toto.emb/toto.cse`".

3 Initialisation du plateau

Le fichier `taquin.adb` commence par définir la structure de données utilisée pour représenter l'état du taquin à un instant du jeu. Elle utilise les types `Piece` et `Numero` définis dans le paquetage `ParamJeu` (voir TPL0.2).

3.1 Structure de données imposée

L'état du taquin est simplement défini comme un tableau `Plateau` qui associe à chaque position du plateau une valeur de type `Cellule` qui représente le contenu de cette case. Une telle valeur a un champ `Typ` qui décrit si la case est vide (valeur `Vide`) ou si c'est un morceau de véhicule (autres valeurs). Dans le cas où `Typ` n'est pas la valeur `Vide`, le champ `Obj` indique la forme du véhicule (voiture ou camion, horizontal ou vertical). Si `Typ=Vide` alors `Obj` n'a aucune signification est peut-être quelconque. Notons ici, que tous les véhicules ont une tête (valeur `Tete`) et une queue (valeur `Queue`), mais quel seul les camions ont un milieu (valeur `Milieu`). Autrement dit, si `Typ=Milieu` alors `Obj=Camion`.

```
type Contenu is (Vide, Tete, Milieu, Queue) ;

type Cellule is record
   Typ: Contenu ;
   Obj: Piece ;
end record ;

Plateau: array(Numero,Numero) of Cellule ;
```

Par exemple, la figure 1 détaille la valeur de `Plateau` dans la configuration correspondant à `n04.emb`. Remarquons que la tête d'un véhicule horizontal (resp. vertical) est toujours sur une colonne (resp. ligne) de numéro inférieur à celle de sa queue.

3.2 À faire dans cette sous-tâche

1. Écrire la procédure `InitPlateauDefaut` qui initialise le plateau dans la configuration correspondant à `n04.emb`. Il est fortement conseillé de prévoir des procédures intermédiaires de façon à faciliter le changement de configuration par défaut : en particulier, si l'analyseur écrit au TPL0 ne fonctionne pas suffisament bien, on pourra écrire chacune des différentes configurations de test du paquetage directement comme une procédure du paquetage à la manière de `InitPlateauDefaut`. Tester que `InitPlateauDefaut` fonctionne à l'aide du programme fourni `testtaquin`.

	0	1	2	3	4	5
0	Vide,_	Vide,_	Tete, (Camion, Horizontal)	Milieu, (Camion, Horizontal)	Queue, (Camion, Horizontal)	Tete, (Camion, Vertical)
1	Vide,_	Vide,_	Tete, (Voiture, Vertical)	Vide,_	Vide,_	Milieu, (Camion, Vertical)
2	Vide,_	Vide,_	Queue, (Voiture, Vertical)	Tete, (Voiture, Horizontal)	Queue, (Voiture, Horizontal)	Queue, (Camion, Vertical)
3	Tete, (Voiture, Vertical)	Tete, (Voiture, Horizontal)	Queue, (Voiture, Horizontal)	Tete, (Camion, Vertical)	Vide,_	Vide,_
4	Queue, (Voiture, Vertical)	Vide,_	Vide,_	Milieu, (Camion, Vertical)	Vide,_	Vide,_
5	Vide,_	Vide,_	Vide,_	Queue, (Camion, Vertical)	Vide,_	Vide,_

Fig. 1 – Valeur de `Plateau` dans la configuration correspondant à `n04.emb`

2. En utilisant l'analyseur du TPL0, implémenter la fonction `InitPlateau` qui lit la configuration dans le fichier désigné par `NomFichierConfig`.[2] Cette procédure lève `ErreurConfig` si certaines pièces ne rentrent pas sur le plateau où se chevauchent, et s'il n'y a pas un unique véhicule horizontal en ligne `C` (numéro 2) ou si ce véhicule n'est pas une voiture. Tester que les configurations incorrectes au sens du paquetage `InitPlateau`, mais correctes au sens de `Analyseur` lèvent bien l'exception `ErreurConfig`.
3. Écrire et tester la fonction `PlateauGagnant` qui retourne `True` ssi la voiture horizontale en ligne `C` est en colonne 0.

4 Coups sur le plateau

4.1 Structure de données imposée

La structure de données pour représenter les coups est très proche de la syntaxe des coups dans `semba` voir TPL0.1.1. Une valeur de type `Coup` est ainsi constituée du champ `Pos` qui désigne la position initiale de la tête de la pièce à déplacer, du champ `Dir` qui représente la direction de déplacement de la tête et du champ `Dest` qui représente le numéro de la coordonnée qui change lors du déplacement (donc un numéro de ligne ou de colonne en fonction de `Dir`).

```
type Coup is record
   Pos: Position ;
   Dir: Direction ;
   Dest: Numero ;
end record ;
```

Ainsi, `((0,1),Vertical,2)` représente le coup noté `A12` alors que `((0,1),Horizontal,2)` représente le coup noté `A1C`.

Bien entendu, pour qu'un coup soit valide, il faut que la pièce à deplacer (dont la tête est initialement en position `Pos`) soit de même direction que `Dir`.

[2] C'est une bonne idée ici de réutiliser les procédures intermédiaires de `InitPlateauDefaut`.

L'implémentation de la fonction Conso calculant la consommation d'un coup en fonction de l'état du plateau est donnée dans taquin.adb.

4.2 À faire dans cette sous-tâche

1. Écrire et tester les procédures Put et Get.
2. Écrire la fonction CoupValide qui teste si un coup est valide dans la configuration courante du taquin. On rappelle (cf. section TPL0.1.1) qu'un coup est valide s'il correspond à un déplacement possible sur le jeu physique.
3. Écrire la procédure Effectue qui joue le coup Cp. Cette fonction requiert que Cp est un coup valide (elle ne le vérifie pas). Elle change la configuration du taquin.

5 Enregistrement et restauration de la configuration

On s'intéresse ici au mécanisme d'enregistrement/restauration d'une configuration décrit en section TPL0.1.3. Il s'agit de pouvoir stocker des états du plateau dans des variables, et aussi de pouvoir comparer deux états du plateau pour savoir s'ils sont égaux. Cette possibilité est utilisée par le programme semba pour savoir si la configuration courante correspond à la configuration enregistrée.

Concrètement, on va ici construire une fonction qui code chaque configuration du taquin comme un couple d'entiers 32 bits. Le codage des configurations est ainsi beaucoup plus économique en mémoire que la recopie complète du plateau de jeu. On va de plus choisir un codage injectif, de sorte que pour savoir si deux configurations sont égales, il suffit de tester si leur codage sont égaux.

Plus précisément, Etat est le type des codages des configurations. La fonction ExporteEtat retourne le code associé à la configuration courante du plateau. La procédure MetPlateau décode son argument et met le plateau dans la configuration correspondante. Cette procédure suppose que l'argument fourni correspond bien à une configuration (il a été obtenu par un précédent appel à ExporteEtat). On peut utiliser l'égalité entre éléments de type Etat pour comparer les configurations correspondantes.

```
type Etat is private ;

function ExporteEtat return Etat ;

procedure MetPlateau(S: Etat) ;
```

5.1 Structure de données imposée

Le type Etat est en fait défini comme un tableau de 2 entiers. Chacun de ces entiers code les positions des pièces d'une direction donnée. Ainsi si Dir:Direction et S:Etat, alors S(Dir) contient uniquement les informations de position des pièces de direction Dir. La position des pièces de direction perpendiculaire à Dir n'a aucune influence sur S(Dir).

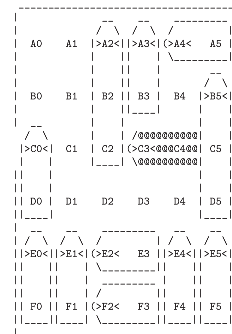
```
type Etat is array(Direction) of Natural ;
```

Grosso-modo, l'entier S(Dir) représente un tableau unidimensionnel de 6 rangées, où chaque *rangée* représente une ligne si Dir est Horizontal ou une colonne si Dir est Vertical. Dans la suite, on appelle coordonnée de la rangée son numéro de ligne si c'est une ligne, ou son numéro de colonne si c'est une colonne. Chaque rangée représente donc elle-même une suite de véhicules

de direction Dir dans cette rangée. Comme dans la syntaxe des fichiers de configurations, on ne conserve dans les rangées que l'information sur les têtes de pièce.

Concrètement, la nature Nat:Vehicule d'une pièce dans la rangée est codée par un chiffre Vehicule'Pos(Nat)+1 (qui vaut donc 1 dans le cas d'une voiture et 2 dans le cas d'un camion). Le chiffre 0 représente l'absence d'une tête de véhicule de direction Dir (autrement dit, soit l'absence de véhicule, soit un morceau de véhicule dans la direction opposée à Dir, soit un milieu ou une queue de véhicule de direction Dir).

Chaque rangée correspond donc à un entier ternaire (en base 3), où la position du chiffre représente la position de la tête du véhicule dans sa rangée : cette position est un numéro de colonne si Dir est Horizontal (la rangée étant une ligne) et un numéro de ligne sinon. Autrement dit, une rangée est un nombre $\sum_{i=0}^{4} d_i 3^i$ où d_i signifie qu'il y a, en position i, soit une tête de voiture (si d_i vaut 1), soit une tête de camion (s'il vaut 2), soit autre chose (s'il vaut 0). Remarquons qu'une tête de véhicule ne peut pas avoir une position supérieure à 4 sans que le véhicule ne déborde. Une rangée est donc toujours codable sur 5 chiffres.



Considérons la configuration donnée par nhl.emb ci-contre. La rangée verticale de coordonnée 0 correspond ainsi au nombre $0.3^0 + 0.3^1 + 1.3^2 + 0.3^3 + 1.3^4 = 90$ (tête de voiture en ligne C et E). Les rangées verticales de coordonnée 1 et 4 sont identiques et valent $3^4 = 81$. Les rangées verticales 2 et 3 valent respectivement 2 et 1. La rangée verticale 5 vaut $2.3 + 3^4 = 87$.

La rangée horizontale 0 (ligne A) vaut $3^4 = 81$. Les rangées horizontales 1 et 3 (lignes B et D) valent 0 (elles sont vides). La rangée horizontale 2 vaut $3^3 = 27$. Les rangées 4 et 5 valent $3^2 = 9$.

L'implémentation définit une rangée comme une valeur du type Rangee ci-dessous. La valeur 91 est en effet la rangée maximale possible : elle correspond à une rangée contenant 3 voitures (par exemple, une rangée de 2 camions a la valeur 56).

```
subtype Rangee is Integer range 0..91 ;
```

Maintenant qu'on a détaillé le calcul des rangées, revenons au calcul de S(Dir). Pour coder l'état de 6 rangées quelconques, il faut donc un nombre compris entre 0 et $92^6 - 1$ soit environ 2^{39}, ce qui ne tient pas sur 32 bits. Mais si on prend en compte les impossibilités de chevauchement des véhicules, il y a seulement 24 rangées "possibles", et 24^6 est de l'ordre de 2^{28} donc ça rentre sur 32 bits. On associe donc à chaque rangée *possible* un code entre 0 et 23 de type RangeeCode. On a fait un précalcul des 24 rangées *possibles*. Deux tableaux définis dans taquin.adb permettent de passer des rangées *possibles* à leur code et réciproquement.

```
subtype RangeeCode is Integer range 0..23 ;
RangeeVal: constant array(RangeeCode) of Rangee := -- voir "taquin.adb"
RangeePos: constant array(Rangee) of -1..23 := -- voir "taquin.adb"
```

Ainsi, si r_i est la valeur de la rangée de direction Dir et de coordonnée i, alors le calcul de S(Dir) est donné par $\sum_{i=0}^{5} \texttt{RangeePos}(r_i).24^i$.

5.2 À faire dans cette sous-tâche

1. Écrire la procédure `ExporteEtat` (ne pas oublier le schéma de Horner vu dans l'exo (5)1).
2. Écrire la procédure `MetPlateau`.
3. Après avoir testé les procédures `ExporteEtat` et `MetPlateau`, écrire la fonction `Effectue` qui sera utile au TPL2. L'idée est bien sûr de calculer directement le nouvel état en remarquant qu'une seule rangée est modifiée par le coup joué. On pourra ici utiliser l'opérateur Ada infixe d'élévation à la puissance "`**`".

6 Énumération de l'ensemble des coups valides

Pour énumérer tous les coups valides de la configuration courante, on utilise une sorte de machine séquentielle avec sentinelle de fin (modèle 1). Par rapport à la notion de machine séquentielle vue en cours, l'état de la machine est ici explicite sous la forme d'une valeur de type `EnumCoup`. Cela permet d'utiliser en parallèle plusieurs *instances* de la machine : on a en fait la même chose avec le type `File` de `Ada.Text_Io`.

```
type EnumCoup is private ;
procedure Demarre(E: out EnumCoup) ;
function FinCoup(E: EnumCoup) return Boolean ;
function CoupCourant(E: EnumCoup) return Coup ;
procedure Avance(E:in out EnumCoup) ;
```

Typiquement, le programme `semba` utilise la machine d'énumération pour l'affichage de la liste des coups valides.

```
procedure Affiche_Liste_Coup is
   E: EnumCoup ;
begin
   Demarre(E) ;
   while FinCoup(E) loop
      Put("""j") ;
      Put(CoupCourant(E)) ;
      Put(""" ") ;
      Avance(E) ;
   end loop ;
end ;
```

6.1 Structure de données suggérée

On suggère d'implémenter cette machine séquentielle de manière à ce que chaque case du plateau ne soit visitée qu'au plus deux fois au cours d'un parcours de l'ensemble des coups valides : une première fois lorsqu'on recherche une tête de véhicule à déplacer, et une seconde fois (éventuelle) pour rechercher les destinations possibles de cette tête de véhicule. Pour cela, quand on a trouvé un véhicule à déplacer, on va rechercher, dans sa rangée, les cases vides contiguës en s'éloignant du véhicule. On partira de la tête pour rechercher les cases vides de coordonnée inférieure (dans la rangée) et on partira de la queue pour celles de coordonnée supérieure.

Par exemple, le type `EnumCoup` pourra être implémenté comme un enregistrement formé d'un coup `Cp`, d'un booléen `Fini`, d'une position `Dest` et d'un entier `Pas` valant `-1` ou `1`.

```
type EnumCoup is record
   Cp: Coup ;
   Fini: Boolean := True ;
   Dest: Position ;
   Pas: Integer ;   -- vaut -1 ou 1 (sens de déplacement pour "Dest") ;
end record ;
```

Lorsque `Fini` vaut `True`, cela signifie que l'énumération est finie : les autres champs n'ont aucune signification. Dans le cas contraire, l'entier `Pas` indique le sens d'exploration des cases voisines du véhicule : `-1` pour les cases de coordonnée inférieure et `1` pour les cases de coordonnée supérieure. Le champ `Cp` correspond au dernier coup valide trouvé et le champ `Dest` correspond à la dernière case vide trouvée pour l'extremité du véhicule qui est soit la tête lorsque `Pas=-1` ou soit la queue lorsque `Pas=1`. Autrement dit lorsque `Fini` vaut `False`, si `Pas=-1` alors `Dest=Cp.Dest`, sinon si T est la taille du véhicule à déplacer, alors `Cp.Dest` vaut "`Dest`$+1-T$". Et finalement, le principe d'exploration des cases voisines correspond à l'invariant suivant :

> *Si* **not** *`Fini` alors tous les coups de direction `Cp.Dir`, de position initiale `Cp.Pos` et de destination comprise entre `Cp.Pos` et `Cp.Dest` sont valides.*

6.2 À faire dans cette sous-tâche

Écrire les procédures et fonctions de la machine séquentielle. Il est recommandé d'écrire des procédures intermédiaires. Par exemple, une pour chercher une tête de véhicule à déplacer et une autre pour chercher une destination possible pour cette tête de véhicule.