Céline AUSSOURD
Adrien JOLY
4IF2 - H4202

PROJET PROLOG

PACMAN

Mardi 9 Novembre 2004

TABLE DES MATIÈRES

A. Résolution détaillée d'un exercice : Concat

A.1) Spécifications

• Liste

- la liste vide notée [ ] est une liste,
- un objet constitué d'un premier élément, appelé tête, suivi d'une liste appelée reste, est une liste.

• Le prédicat concat\3

- La concaténation d'une première liste vide [ ] avec une seconde liste L donne pour résultat une liste identique à L,
- de même, concaténation d'une première liste L avec une seconde liste vide [ ] donne pour résultat une liste identique L,
- la liste résultat est la concaténation de la première liste avec la deuxième liste,
- la tête de la liste résultat est identique à la tête de la première liste, et le reste du résultat est la concaténation du reste de la première liste avec la deuxième liste.

- ***LES MODES D'APPEL DU PRÉDICAT (UTILISATION DU PRÉDICAT AVEC DES VARIABLES)***

Remarque : liste1, liste2 et liste3 sont des listes. X, Y et Z sont des variables.

1. concat(liste1, liste2 ,liste3) : vraie quand 'liste3' est la concaténation de 'liste1' et 'liste2' et faux dans le cas contraire.
2. concat(X, liste2, liste3) :
    a. Si 'liste2' est une sous liste de 'liste3' et qu'il existe une liste 'liste1' telle que la concaténation 'liste1' avec 'liste2' donne 'liste3', alors X=liste1.
    b. Si liste2=liste3, alors X=[] : la liste vide.
    c. Dans tous les autres cas, le prédicat est faux.
3. concat(liste1, Y, liste3) :
    a. Si 'liste'1 est une sous liste de 'liste3' et qu'il existe une liste 'liste2' telle que la concaténation de 'liste1' avec 'liste2' donne 'liste3', alors Y=liste2.
    b. Si liste1=liste3, alors Y=[] : la liste vide.
    c. Dans tous les autres cas, le prédicat est faux.
4. concat(liste1, liste2, Z) : Z=liste1|liste2 : la concaténation des deux listes.
5. concat(X, Y, liste3) :
    a. X et Y prennent différentes valeurs telles que la concaténation de X avec Y donne 'liste3'.
    b. Si 'liste3' est la liste vide, X et Y sont également des listes vides.

Appels non valides :

On n'autorise pas la présence de variables dans la valeur du troisième argument de concat, c'est pour cela que les appels suivants sont traités particulièrement. Ainsi, l'utilisateur n'aura pas de réponses contenant des valeurs sans signification (valeurs attribuées par Prolog) et n'obtiendra pas non plus une infinité de solutions.

6. concat(X, liste2, Z) : La solution trouvée sera Z=liste2 et X=[]. Ce sera la seule bien que d'autres valeurs de X (et donc de Z) sont envisageables.
7. concat(liste1, Y, Z) : La solution trouvée sera Z=liste1|Y, aucune valeur ne sera trouvée pour Y. Ce sera la seule solution bien que d'autres valeurs de Y (et donc de Z) sont envisageables.
8. concat(X, Y, Z) : La solution trouvée sera Y=Z et X=[]. Ce sera la seule solution, aucune autre valeur ne sera envisagée pour les variables.

A.2) RÉALISATION

```
concat([],L,L).
concat([X|Reste],Y,[X|G]) :- nonvar(X), concat(Reste,Y,G).
```

A.3) TESTS

- **ÉTUDE EN DÉTAILS D'UN APPEL DE CONCAT\3**

- 5. a

```
| ?- concat(X,Y,[a,b,c,d]).
Call: concat(_570830,_570860,[a,b,c,d]) ?          % appel de concat([],L,L)->X=[] et
Exit: concat([],[a,b,c,d],[a,b,c,d]) ?             % Y=L=[a,b,c,d]
Y = [a,b,c,d]                                      %
X = []?;                                           % échec.
Redo: concat([],[a,b,c,d],[a,b,c,d]) ?             % backtrack.
  Call: nonvar(a) ?                                % appel de nonvar(X) pour la tête du
  Exit: nonvar(a) ?                                % troisième argument -> vrai.
  Call: concat(_5709dc,_570860,[b,c,d]) ?          % appel de concat(Reste,Y,G)
  Exit: concat([],[b,c,d],[b,c,d]) ?               % appel de concat([],L,L)->Y=[b,c,d]
Exit: concat([a],[b,c,d],[a,b,c,d]) ?              % reconsidération de
Y = [b,c,d]                                        % concat([X|Reste],Y,[X|G])->X=[a]
X = [a]?;                                          % échec.
Redo: concat([a],[b,c,d],[a,b,c,d]) ?              % backtrack
  Redo: concat([],[b,c,d],[b,c,d]) ?               %
    Call: nonvar(b) ?                              % appel de nonvar(X) -> vrai
    Exit: nonvar(b) ?                              %
    Call: concat(_570a98,_570860,[c,d]) ?          % appel de concat(Reste,Y,G)
    Exit: concat([],[c,d],[c,d]) ?                 % appel de concat([],L,L)-> Y=[c,d]
  Exit: concat([b],[c,d],[b,c,d]) ?                % reconsidération de
Exit: concat([a,b],[c,d],[a,b,c,d]) ?              % concat([X|Reste],Y,[X|G])->X=[a,b]
Y = [c,d]                                          %
X = [a,b]?;                                        % échec.
Redo: concat([a,b],[c,d],[a,b,c,d]) ?              % backtrack.
  Redo: concat([b],[c,d],[b,c,d]) ?                %
    Redo: concat([],[c,d],[c,d]) ?                 %
      Call: nonvar(c) ?                            % appel de nonvar(X) -> vrai
      Exit: nonvar(c) ?                            %
      Call: concat(_570b50,_570860,[d]) ?          % appel de concat(Reste,Y,G)
      Exit: concat([],[d],[d]) ?                   % appel de concat([],L,L)->Y=[d]
    Exit: concat([c],[d],[c,d]) ?                  % reconsidération de
  Exit: concat([b,c],[d],[b,c,d]) ?                % concat([X|Reste],Y,[X|G])
Exit: concat([a,b,c],[d],[a,b,c,d]) ?              % -> X=[a,b,c]
Y = [d]                                            %
X = [a,b,c]?;                                      % échec.
Redo: concat([a,b,c],[d],[a,b,c,d]) ?              % backtrack.
  Redo: concat([b,c],[d],[b,c,d]) ?                %
    Redo: concat([c],[d],[c,d]) ?                  %
      Redo: concat([],[d],[d]) ?                   %
        Call: nonvar(d) ?                          % appel de nonvar(X) -> vrai
        Exit: nonvar(d) ?                          %
        Call: concat(_570c04,_570860,[]) ?         % appel de concat(Reste,Y,G)
        Exit: concat([],[],[]) ?                   % appel de concat([],L,L)->Y=[]
      Exit: concat([d],[],[d]) ?                   % reconsidération de
    Exit: concat([c,d],[],[c,d]) ?                 % concat([X|Reste],Y,[X|G])
  Exit: concat([b,c,d],[],[b,c,d]) ?               % -> X=[a,b,c,d]
Exit: concat([a,b,c,d],[],[a,b,c,d]) ?             %
Y = []                                             %
X = [a,b,c,d]?;                                    % échec.
Redo: concat([a,b,c,d],[],[a,b,c,d]) ?             % backtrack
  Redo: concat([b,c,d],[],[b,c,d]) ?               %
    Redo: concat([c,d],[],[c,d]) ?                 %
      Redo: concat([d],[],[d]) ?                   %
        Redo: concat([],[],[]) ?                   %
        Fail: concat(_570c04,_570860,[]) ?         %
        Fail: nonvar(d) ?                          %
      Fail: concat(_570b50,_570860,[d]) ?          %
      Fail: nonvar(c) ?                            %
    Fail: concat(_570a98,_570860,[c,d]) ?          %
    Fail: nonvar(b) ?                              %
  Fail: concat(_5709dc,_570860,[b,c,d]) ?          %
  Fail: nonvar(a) ?                                %
Fail: concat(_570830,_570860,[a,b,c,d]) ?          %
No                                                 % échec, fin des solutions.
```

- ***AUTRES TESTS***

- 1.

```
| ?- concat([a,b],[c,d],[a,b,c,d]).
Yes
| ?- concat([a,b],[c],[a,b,c,d]).
no
```

- 2.a

```
| ?- concat(X,[c,d],[a,b,c,d]).
X = [a,b]?;
no
```

- 2.c

```
| ?- concat(X,[c],[a,b,c,d]).
no
```

- 3.a

```
| ?- concat([a,b],Y,[a,b,c,d]).
Y = [c,d]?;
no
```

- 3.c

```
| ?- concat([e],Y,[a,b,c,d]).
no
```

- 4.

```
| ?- concat([e],[f,g],Z).
Z = [e,f,g]?;
no
```

- ***APPELS NON VALIDES:***

- 6.

```
| ?- concat(X,[c],Z).
Z = [c]
X = []?;
no
```

- 7.

```
| ?- concat([a,b],Y,Z).
Z = [a,b|Y]?;
no
```

- 8.

```
| ?- concat(X,Y,Z).
Y = Z
X = []?;
no
```

B. Projet libre : Pacman

B.1) Introduction

Le langage Prolog permet non pas de programmer à l'aide d'algorithmes, mais d'une façon plus mathématique. Ainsi pour résoudre un problème, on n'implémente pas la manière de procéder (méthode) mais plutôt en fixant des règles et des contraintes. La seule méthode connue de Prolog est la déduction. C'est pourquoi ce langage est particulièrement adapté à l'intelligence artificielle.

Notre souhait pour ce projet était donc de tirer parti de cet atout pour implémenter un programme intelligent. Nous avons alors pensé au « pathfinding » (recherche de chemins), et une de ses applications était le mouvement des fantômes dans le célèbre jeu culte « Pacman ».

B.2) CAHIER DES CHARGES

Les points d'intérêts principaux de ce projets étant :

- une structuration « intelligente » du labyrinthe, c'est à dire en évitant d'utiliser une matrice, ce qui est courant dans les implémentations algorithmiques
- et la faculté des fantômes à se déplacer efficacement dans le but d'atteindre leur cible (pacman) au plus vite

Nous n'avons accordé que peu d'intérêt à l'interface homme-machine (affichage et contrôle clavier). De toute façon Prolog n'est pas fait pour développer des applications multimédias !

Le jeu devait donc permettre à l'utilisateur de déplacer pacman dans le labyrinthe, son principal but étant de passer par toutes les cases circulables pour manger des pastilles, et d'éviter les fantômes pour rester en vie. Nous avons donc opté pour un mode de jeu par tours, chaque déplacement d'une case constituant un tour. Chaque tour serait déclenché par un déplacement de pacman, quand l'utilisateur presse une touche de direction du clavier. Les fantômes se déplaceraient alors aussi d'une case chacun lors de chaque tour.

• PACMAN

- Gagne en mangeant toutes les pastilles
- perd quand il est « touché » (à la même position) par un fantôme, sauf quand il est sous l'effet d'une potion
- devient invulnérable pendant 30 tours lorsqu'il mange une potion

• LA SITUATION AU DÉPART

- Sur chaque case circulable il y a une pastille à manger
- les fantômes sont « enfermés » dans une « salle » possédant deux ouvertures seulement, au centre du labyrinthe
- pacman est placé vers le bas du labyrinthe
- le score est nul

• LES CARACTÉRISTIQUES DU LABYRINTHE

- symétrique horizontalement
- un couloir permet de passer d'un côté à l'autre de l'écran
- il n'y a pas de « cul de sac », chaque case circulable est liée à au moins deux autres cases circulables
- il n'y a pas de circuits fermés au sein du labyrinthe, toutes les cases circulables sont directement ou indirectement liées entre elles

- ***LES FANTÔMES***

- sont au nombre de 4
- possèdent tous la même intelligence
- se déplacent indépendamment les uns des autres
- sont des obstacles les uns pour les autres
- ne peuvent pas changer de direction dans les couloirs (cases circulables liées à seulement deux autres cases circulables alignées), sauf en cas de collision
- ne peuvent pas faire demi-tour, sauf en cas de collision
- ont des cases « cible » dynamiques
- s'enfuient lorsque Pacman est sous l'effet d'une potion
- se déplacent à la même vitesse que pacman : à raison d'une case par tour

- ***LES POTIONS***

- Sont réparties au nombre de 4 dans le labyrinthe
- Mangée par Pacman, une potion fait effet durant 30 tours
- Fait fuir les fantômes car ils deviennent vulnérables à Pacman
- Permet à Pacman de remporter 10 points en touchant un fantôme

B.3) ARCHITECTURE

• PRÉDICATS DYNAMIQUES

- posPacman(X,Y) : Coordonnées courantes de Pacman
- dirPacman(Dir) : Direction courante de Pacman

- posFantome(N,X,Y) : Coordonnées courantes du fantôme N
- dirFantome(N,Dir) : Direction courante du fantôme N
- cibleFantome(N,X,Y) : Coordonnées courantes de la cible du fantôme N

- decomptePotion(T) : Nombre de tours avant la fin d'effet de la dernière potion bue
- fin(Booleen) : Le jeu se termine quand Booleen est VRAI
- score(Points) : Nombre de points en cours acquis par Pacman
- visite(X,Y) : Coordonnées des cases déjà visitées (pastille mangée par pacman)

- ***STRUCTURE DU LABYRINTHE***

Le labyrinthe a une structure symétrique horizontale. Pour décrire la structure du labyrinthe, nous avons positionné les murs (et pas les cases circulables). Nous avons indiqué la position des murs seulement pour une moitié du labyrinthe, l'autre moitié étant complétée par symétrie. Le prédicat `symetrieHoriz\2` permet de réaliser cette symétrie.

Nous avons introduit les prédicats `largeur\1`, `hauteur\1` et `dims\2` pour pouvoir récupérer les dimensions du labyrinthe dans d'autres prédicats.

En effet, le labyrinthe est caractérisé par sa largeur et sa hauteur (qui constituent ses dimensions). Il est constitué de cases. Pour repérer une case, on utilise ses coordonnées qui correspondent au numéro de colonne et de ligne.

Le prédicat `case\2` fait référence à cette notion.

Les prédicats `ligne\1` et `colonne\1` permettent d'indiquer quelles sont les valeurs possibles des arguments de `case\2`, c'est à dire quelles peuvent être les valeurs prises par les coordonnées d'une case.

- ***CIRCULATION DANS LE LABYRINTHE***

Le labyrinthe est constitué de cases, définies par leurs coordonnées. Les prédicats définissant le nombre de cases en hauteur et en largeur ont été cités plus haut. Chaque `case` peut être un obstacle (`obstacle/2`) ou circulable (`caseCirculable/2`). Les murs et les entités (pacman et les fantômes) sont des obstacles. Les entités ne peuvent se déplacer que sur des cases circulables.

Les cases circulables voisines (`voisin/4`), liées entre elles, constituent des chemins qui peuvent être caractérisés :

- « Couloir » : trois cases circulables alignées, la case du milieu n'a que deux voisines circulables
- « Angle » (`angle/2`) : lorsqu'une case circulable a au moins deux voisines circulables non alignées
- « Carrefour » (`carrefour/2`) : lorsqu'une case circulable a au moins trois voisines circulables

Les prédicats `gauche/4`, `droite/4`, `dessus/4` et `dessous/4` permettent de situer les cases entre elles.

- ***NOTION DE DIRECTION D'ENTITÉS***

Pour modéliser les directions de déplacements d'entités, nous avons associé des valeurs aux quatre points cardinaux : Nord=0, Est=1, Sud=2, Ouest=3.

Le prédicat `dirIncrement(Dir,Xincr,Yincr)` lie ces valeurs de directions aux incréments horizontal (`Xincr`) et vertical (`Yincr`). Ces incréments valent `-1`, `0` ou `+1`.

Le prédicat `positionSuivante(X3,Y3,X,Y,Xincr,Yincr)` permet d'ajouter les incréments `Xincr` et `Yincr` aux coordonnées (`X,Y`) puis de renvoyer le résultat dans les coordonnées (`X3,Y3`).

Le prédicat `positionSuivanteVersCible(X1,Y1,X,Y,Xc,Yc)` permet de connaître les coordonnées (`X1,Y1`) de la première case circulable voisine des coordonnées (`X,Y`) en direction (si possible) d'une cible de coordonnées (`Xc,Yc`). Dans le cas où la case trouvée n'est pas circulable, le prédicat choisira les coordonnées (`X1,Y1`) de la case circulables voisine la moins éloignée possible de la case cible.

- ***DÉPLACEMENT ADAPTIF VERS UNE CIBLE, DANS LE LABYRINTHE***

Nous allons décrire le mécanisme de décision adaptif utilisé dans les prédicats `positionSuivanteVersCible et trouverItineraire`. On parle d'adaptivité car le déplacement d'une case à une autre case voisine dans une direction donnée n'est pas forcément possible à cause des murs et autres obstacles. Le but est alors de faire des compromis, de manière à s'éloigner le moins possible de la cible choisie.

Pour cela, imaginons un segment reliant la case d'origine à la case cible, puis décomposons-le en un segment horizontal et un segment vertical. Le plus long des deux constituera une « direction majeure » parmi nos quatre points cardinaux ; le plus court, une « direction mineure ». Voici les choix parmi lesquels devra s'opérer la décision, du cas optimal au cas le pire :

- Déplacement dans la direction majeure
- Déplacement dans la direction mineure
- Déplacement dans la direction mineure opposée
- Déplacement dans la direction majeure opposée

Prenons comme exemple la situation suivante (Fig. 1) :

Fig. 1 : Situation

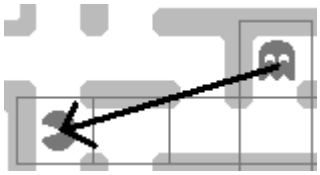

Fig. 2 : Vecteur vers la cible

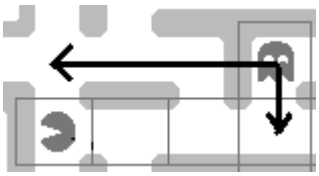

Fig. 3 : Vecteur majeur et mineur

Disons que l'entité source est le fantôme, et la cible Pacman. Nous voyons que le vecteur vers la cible (Fig. 2) se décompose en un vecteur horizontal et un vertical (Fig. 3). Le plus long est l'horizontal, appelé « majeur », en opposition au plus court, appelé « mineur ». On voit bien que la cible (Pacman) se trouve plus à gauche (incrément horizontal négatif) que vers le bas (incrément vertical positif). La case voisine optimale en direction de la cible, suivant le vecteur majeur, serait alors à gauche. Or cette case n'est pas circulable à cause du mur. Nous évaluons alors la possibilité de suivre le vecteur minoritaire pour élire la case du dessous, qui elle est circulable. Ce sera donc la solution retenue.

- ***DÉPLACEMENT DES FANTÔMES***

Le déplacement des fantôme lors d'un tour de jeu est assuré par le prédicat `deplaceFantomes/0`. Celui-ci se contente d'appeler le prédicat `deplaceFantome(N)` pour chacun d'eux, `N` étant le numéro du fantôme à déplacer.

Chaque fantôme a une direction en cours parmi les quatre points cardinaux, et une position, tous deux déterminés par les prédicats dynamiques `posFantome(N,X,Y)` et `dirFantome(N,Dir)`. A chaque tour, les fantômes se déplacent d'une case dans leur direction courante tant qu'ils se trouvent dans un couloir. Dès qu'une alternative de direction (angle, carrefour) s'offre à l'un deux, celui-ci évalue la direction possible (case voisine circulable) qui l'approchera le plus possible de sa cible, dont les coordonnées sont déterminées par le prédicat `cibleFantome(N,X,Y)` ; ou du moins, s'éloignera le moins possible de sa cible, sachant que des obstacles peuvent se trouver sur son chemin optimal.

Voici les grandes lignes du prédicat `deplaceFantome(N)` :

- Calcul de la case circulable suivante, en utilisant la direction en cours du fantôme
- Si le fantôme est dans un couloir, le choix se porte sur cette case
- Si le fantôme n'est pas dans un couloir, choix d'une case circulable suivante optimale en direction de Pacman dans la mesure du possible, et mise à jour de la direction courante
- Le fantôme se déplace dans la case suivante choisie

Le calcul de la case circulable suivante dans la direction courante du fantôme est assuré par le prédicat `positionSuivanteVersCible` vu plus haut.

Le choix d'une case circulable suivante optimale, dans le cas où le fantôme peut changer de direction (angle, carrefour), est fait dans le prédicat `redirigerFantome(N,X,Y,X2,Y2)`. Celui-ci commence par identifier les coordonnées de la cible visée par le fantôme, trouve un première étape d'itinéraire optimal avec `trouverItineraire/4`, puis déplace le fantôme dans la première case voisine menant à la première étape de l'itinéraire, avec `positionSuivanteVersCible/6`.

Nous avons vu plus haut que le prédicat `positionSuivanteVersCible/6` possédait une certaine adaptivité, permettant de choisir des cases alternatives avec la notion de vecteurs majeur/mineur. Le prédicat `trouverItineraire([X2,Y2],[X,Y],[Xc,Yc],ODir)` permet, quant à lui, de trouver les coordonnées `[X2,Y2]` d'un angle dans les directions possibles du fantôme (de coordonnées `[X,Y]`), dans la mesure du possible vers la cible `[Xc,Yc]`. Il garantit aussi que la direction choisie ne sera pas l'opposé de la direction précédente `ODir`, de manière à interdire les demis-tours aux fantômes. Bien sûr cela reste possible pour eux dans le cas où il y a collision entre deux fantômes dans un couloir par exemple, mais `trouverItineraire/4` ne trouvera pas de solution, c'est `positionSuivanteVersCible/6` qui prendra la décision de faire demi-tour, à défaut d'alternatives.

Mais pourquoi déterminer les coordonnées d'un angle dans `trouverItineraire/4` au lieu de réutiliser simplement `positionSuivanteVersCible/6` pour trouver directement une case voisine ? Pour améliorer la stratégie ! Dans la situation représentée à gauche, les deux fantômes vont vers la droite. Le fantôme 2 suit sa route vers Pacman en descendant à l'angle 2, car il n'a pas le droit de faire demi-tour. Le fantôme 1 quant à lui, lorsqu'il se trouve à l'angle 1 (qui est un carrefour) a le choix entre continuer vers la droite ou monter. Mais comme il n'y a pas d'accès direct (sans obstacles) à un angle vers la droite à cause du fantôme 2, `trouverItineraire/4` va décider de diriger le fantôme 1 vers le haut. Ainsi les fantômes se séparent pour mieux encercler Pacman ! En effet deux fantômes se suivant n'ont aucun intérêt car seul le premier pourrait entrer en collision avec Pacman pour le tuer.

Pour trouver les coordonnées `[X2,Y2]` d'un angle optimal à partir d'une case `[X,Y]` et vers une cible `[Xc,Yc]`, `trouverItineraire/4` fait appel à `chercherAngle(XA,YA,X,Y,Dir)`. Ce prédicat récursif « avance » case par case dans la direction `Dir` à partir de la case `[X,Y]` jusqu'à trouver un angle (ou Pacman, car c'est la cible première des fantômes) `[XA,YA]` qui consiste alors en la solution. Si un obstacle est trouvé sur le chemin, il n'y a pas de solution, du moins dans cette direction.

Le prédicat `cibleFantome(N,X,Y)` permet de déterminer les coordonnées `[X,Y]` de la cible que suit le fantôme `N`. Il est utile pour ajouter de la stratégie aux fantômes, par exemple : le fantôme 1 suit Pacman, le fantôme 2 vise derrière Pacman pour le prendre à revers, le fantôme 3 protège la potion pour dissuader Pacman de trop s'en approcher, etc... Mais il permet surtout faire « fuir » les fantômes lorsque Pacman vient de boire une Potion les rendant vulnérables, en donnant comme nouvelle cible leur position initiale au centre du labyrinthe.

- ***GESTION DE L'AFFICHAGE***

Pour l'affichage, voici comment nous procédons :

- On appelle le prédicat `affichage\0` qui va appelé le prédicat `afficherScore\0` (pour l'affichage du score actuel) puis le prédicat `ttDessiner\1` pour l'affichage du labyrinthe.
- Le prédicat `ttDessiner\1`, permet de lancer l'affichage ligne par ligne. `ttDessiner\1` est récursif : une ligne est dessinée à chacun de ses appels. Pour dessiner la ligne, il fait appel à `dessinerLigne\1`.
- `dessinerLigne\1` va permettre l'affichage d'une ligne. Pour cela, elle affiche la ligne case par case en faisant appel à `dessinercarac\2`
- `dessinercarac\2` va afficher une case dont les coordonnées sont mentionnées dans les arguments. Pour savoir quel caractère affiché, il fait appel à `schema\3`.

Une case a 20 affichages différents possibles. Voici les représentations des objets du jeu :

- `###` : un mur
- `0` : une potion
- `.` : une pastille
- ` ` : un chemin où Pacman est passé (la pastille est mangée)
- `v` : Pacman se déplaçant vers le haut
- `^` : Pacman se déplaçant vers le bas
- `<` : Pacman se déplaçant vers la droite
- `>` : Pacman se déplaçant vers la gauche
- `$` : Le fantôme 1 cherchant à manger Pacman
- `%` : Le fantôme 2 cherchant à manger Pacman
- `§` : Le fantôme 3 cherchant à manger Pacman
- `@` : Le fantôme 4 cherchant à manger Pacman
- `1` : Le fantôme 1 fuyant Pacman
- `2` : Le fantôme 2 fuyant Pacman
- `3` : Le fantôme 3 fuyant Pacman
- `4` : Le fantôme 4 fuyant Pacman
- `RIP` : Pacman et un fantôme sur la même case (fin du jeu)

- `schema\3` permet donc de récupérer le caractère à afficher (contenu dans le troisième argument) de la case dont les coordonnées correspondent aux deux premiers arguments.

- ***DÉPLACEMENT DE PACMAN, CONTRÔLE CLAVIER***

Pacman se déplace d'une case dans sa direction courante, à chaque tour, tant qu'il n'est pas face à un obstacle. Sa direction courante est modifiée par l'utilisateur lorsqu'il presse une touche de direction reconnue.

A chaque tour, le prédicat `deplacePacman/0` est appelé. Celui-ci attend que l'utilisateur appuie sur une touche K par appel à `attenteTouche(K)` puis `commandeClavier(K)` interprète la touche pressée. A ce stade il est possible que la direction de Pacman ait changé. Le prédicat `changerPosPacman/4` change sa position actuelle pour le déplacer dans sa direction courante. Ce dernier évalue la case suivante et n'y place pacman que si ce n'est pas un mur. Ensuite l'appel au prédicat `ajoutPoint/2` permet de mettre à jour le score si une pastille a été mangée sur la case en cours.

Le prédicat `commandeClavier(K)` est défini autant de fois qu'il y a de touches reconnues dans le jeu. Chaque instance de ce prédicat est associé à un code de caractère saisi au clavier, et se charge alors de changer la direction de Pacman (prédicat `changerDirPacman/1`) ou de quitter le jeu.

- ***FONCTIONNEMENT DES POTIONS***

La position des potions dans le labyrinthe est déterminée par le prédicat `posPotion(X,Y)`. L'existence d'une potion est liée au fait que Pacman n'a pas visité la case sur laquelle elle se trouve. D'où le prédicat `trouvePotion(X,Y)`. Lorsqu'une potion est « bue » par Pacman, `trouvePotion/2` affecte 30 au prédicat dynamique `decomptePotion(T)`. Cette valeur décroît dans la troisième définition du prédicat `testPotion(X,Y)`, et le prédicat `decomptePotion(T)` est détruit lorsque Pacman n'est plus sous l'effet de la potion (valeur nulle). Ce prédicat `testPotion/2` teste la présence d'un fantôme sur la case de Pacman pour ajouter 10 points au score dans ce cas, puis teste si une potion se trouve sur cette case, avant d'assurer la mise à jour de `decomptePotion/1`.

Lorsque Pacman est « sous l'effet » d'une potion (prédicat `decomptePotion/1` défini), la cible des fantômes (prédicat `cibleFantome(N,X,Y)`) est modifiée pour viser la « maison des fantômes » au centre de l'écran, plutôt que Pacman.

- ***FONCTIONNEMENT DU JEU***

Comme nous l'avons mentionné plus tôt, le jeu fonctionne par tours. On commence par déplacer les fantômes puis on lance l'affichage. Et ensuite on laisse la main à l'utilisateur pour déplacer Pacman. De cette manière, on choisit d'avantager le joueur. Il connaît les positions des fantômes avant de jouer.

- A chaque déplacement (de Pacman ou des Fantômes), on vérifie que Pacman n'est pas mort ; c'est à dire qu'il n'a pas la même position qu'un fantôme (`mort\0`).
- A chaque tour, on vérifie également que l'utilisateur n'ait pas appuyer sur la touche 𝔸 (`fin\1`).
- Pour finir, on vérifie que le joueur n'a pas gagné ; c'est à dire que toutes les pastilles ne sont pas mangées (`gagne\0`).

Si toutes ces conditions sont vérifiées, on lance un nouveau tour. Sinon, le jeu se termine.

C'est le prédicat `vie\0` qui se charge de tout ce processus.

La gestion du score se fait après chaque déplacement de Pacman. S'il est positionné sur une case contenant une pastille, cette case est ajoutée aux cases « visitées » ; c'est à dire aux cases ne contenant plus de pastille. Et le score est incrémenté d'un point. Les prédicats `ajoutPoint\2`, `point\2` et `score\1` gèrent le score.

On utilise une variable dynamique nommée `visite\2` pour indiquer si une case contient une pastille ou non. Comme toutes les variables dynamiques, elle doit être remise à zéro (ou réinitialisée) à chaque fois que le joueur veut recommencer une partie. C'est pour cela que nous utilisons les prédicats `destruction\0` et `initialisation\0` pour « purger » le contenu des variables et les réinitialiser. C'est le prédicat `jeu\0` qui se charge d'appeler ces prédicats puis de lancer le jeu (en appelant `vie\0`).

B.4) Comment jouer ?

Lancez le jeu en saisissant « jeu. », après compilation dans SWI-Prolog.

Pour le lancer dans un autre interpréteur, décommentez la section « Compatibilité avec bProlog ».

- ***Commandes clavier***

Voici les touches reconnues pendant le jeu :

- ℚ pour déplacer Pacman vers la gauche
- 𝔻 pour déplacer Pacman vers la droite
- ℤ pour déplacer Pacman vers le haut
- 𝕊 pour déplacer Pacman vers le bas
- 𝔸 pour quitter

- ***Règles du jeu***

Vous dirigez Pacman (affiché sous forme de <, >, v, ou ^ selon la direction) à l'aide du clavier. Vous ne pouvez pas passer à travers les murs (###).

Pour gagner vous devez manger toutes les pastilles du labyrinthe, elles sont matérialisées à l'écran par des points (.).

Quatre fantômes ($, %, §, et @) vous suivent, vous perdez si l'un d'eux se trouve sur la même case que vous. Évitez-les donc à tout prix !

En buvant une potion (0) vous rendez les fantômes vulnérables à Pacman pendant 30 tours, ce qui vous permet alors de gagner des points en les mangeant !

B.5) CODE SOURCE

```
% Projet Prolog : Pacman
% ======================
%  - Celine AUSSOURD
%  - Adrien JOLY


% ____________________
% Predicats dynamiques

:- dynamic posPacman/2.         % posPacman(X,Y) : Coordonnées de pacman
:- dynamic dirPacman/1.         % dirPacman(Dir) : Direction de pacman
:- dynamic posFantome/3.        % posFantome(N,X,Y) : Coordonnées du fantome N
:- dynamic dirFantome/2.        % dirFantome(N,Dir) : Direction du fantome N
:- dynamic cibleFantome/3.      % cibleFantome(N,X,Y) : Coordonnées de la cible du fantome N
:- dynamic decomptePotion/1.    % decomptePotion(T) : Nombre de tours avant la fin d'effet de potion
:- dynamic fin/1.               % fin(B)
:- dynamic score/1.             % score(S)
:- dynamic visite/2.            % pour la gestion du score ' . ' devient '   '

% ____________________________
% Définition des constantes

largeur(19). % Largeur du niveau : nombre de cases colonnes
hauteur(21). % Hauteur du niveau : nombre de cases lignes
dims(Xmax,Ymax) :- largeur(Xmax), hauteur(Ymax).
symetrieHoriz(X,X2) :- largeur(Xmax), Xmax2 is Xmax-1, X2 is Xmax2-X.

debugMode :- fail.
%debugMode.  % décommenter pour passer en mode débogage (traces, etc...)

% ____________________________
% Compatibilité avec bProlog

%between(Min,Max,V) :- V >= Min, V =< Max.
%get_single_char(X) :- get0(X).

% ____________________
% Validation de position

ligne(Y) :- hauteur(Haut), Ymax is Haut-1, between(0,Ymax,Y). % Y >= 0, Y < Haut.
colonne(X) :- largeur(Larg), Xmax is Larg-1, between(0,Xmax,X). % X >= 0 ,X < Larg.
case(X,Y) :- ligne(Y),colonne(X).
caseCirculable(X,Y) :- case(X,Y), not(obstacle(X,Y)).
obstacle(X,Y) :- mur(X,Y) ; posFantome(_N,X,Y).

% angle : case circulable connectée perpendiculairement à au moins 2 autres cases circulables

angle(X,Y) :-
    caseCirculable(X,Y),
    ( ( gauche(X1,Y1,X,Y), voisinCirculable(X,Y,X1,Y1) ) ;
      ( droite(X2,Y2,X,Y), voisinCirculable(X,Y,X2,Y2) ) ),
    ( ( dessus(X3,Y3,X,Y), voisinCirculable(X,Y,X3,Y3) ) ;
      ( dessous(X4,Y4,X,Y), voisinCirculable(X,Y,X4,Y4) ) ).

% carrefour : case circulable connectée à au moins 3 autres cases circulables

carrefour(X,Y) :-
    caseCirculable(X,Y),
    gauche(X1,Y1,X,Y), voisinCirculable(X,Y,X1,Y1),
    droite(X2,Y2,X,Y), voisinCirculable(X,Y,X2,Y2),
    ( ( dessus(X3,Y3,X,Y), voisinCirculable(X,Y,X3,Y3) ) ;
      ( dessous(X4,Y4,X,Y), voisinCirculable(X,Y,X4,Y4) ) ).

carrefour(X,Y) :-
    caseCirculable(X,Y),
    dessus(X1,Y1,X,Y), voisinCirculable(X,Y,X1,Y1),
    dessous(X2,Y2,X,Y), voisinCirculable(X,Y,X2,Y2),
    ( ( gauche(X3,Y3,X,Y), voisinCirculable(X,Y,X3,Y3) ) ;
      ( droite(X4,Y4,X,Y), voisinCirculable(X,Y,X4,Y4) ) ).

% ____________________________
% Points cardinaux (directions)
% Nord=0, Est=1, Sud=2, Ouest=3

dirIncrement(0,0,-1).
dirIncrement(1,1,0).
dirIncrement(2,0,1).
dirIncrement(3,-1,0).
```

```
positionSuivante(X3,Y3,X,Y,Xincr,Yincr) :-
    dims(Larg,Haut),                        % recuperation des dimensions du labyrinthe
    X2 is X+Xincr+Larg,                     % incrémentation horizontale
    X3 is X2 mod Larg,                      % modulo la largeur du labyrinthe (wrapping)
    Y2 is Y+Yincr+Haut,                     % incrémentation verticale
    Y3 is Y2 mod Haut.                      % modulo la hauteur


positionSuivanteVersCible(X1,Y1,X,Y,Xc,Yc) :-
    Xdiff is Xc-X, ( ( Xincr is sign(Xdiff), Xincr \== 0) ; Xincr is 1 ),
    Ydiff is Yc-Y, ( ( Yincr is sign(Ydiff), Yincr \== 0) ; Yincr is 1 ),
    ( ( abs(Xdiff) > abs(Ydiff), % si pacman est plus loin horizontalement que verticalement
        ( ( positionSuivante(X1,Y1,X,Y,Xincr,0), caseCirculable(X1,Y1)) ;  % increment majeur
          ( positionSuivante(X1,Y1,X,Y,0,Yincr), caseCirculable(X1,Y1)) ;  % increment mineur
          ( positionSuivante(X1,Y1,X,Y,0,-Yincr), caseCirculable(X1,Y1)) ; % increment mineur opposé
          ( positionSuivante(X1,Y1,X,Y,-Xincr,0), caseCirculable(X1,Y1)))  % increment majeur opposé
      ) ;                           % si pacman est plus loin verticalement que horizontalement
      ( ( positionSuivante(X1,Y1,X,Y,0,Yincr), caseCirculable(X1,Y1)) ;    % increment majeur
        ( positionSuivante(X1,Y1,X,Y,Xincr,0), caseCirculable(X1,Y1)) ;    % increment mineur
        ( positionSuivante(X1,Y1,X,Y,-Xincr,0), caseCirculable(X1,Y1)) ;   % increment mineur opposé
        ( positionSuivante(X1,Y1,X,Y,0,-Yincr), caseCirculable(X1,Y1))     % increment majeur opposé
      ) ;
      ( X1 is X, Y1 is Y, afficher('[IMMOBILISME positionSuivanteVersCible]') ) ).

% ____________________
% Voisinnage de cases

% X1,Y1 est en dessus/dessous/gauche/droite de X2,Y2
gauche(X1,Y1,X2,Y2) :-  case(X2,Y2), Y1 is Y2,   X1 is X2-1, case(X1,Y1).
droite(X1,Y1,X2,Y2) :-  case(X2,Y2), Y1 is Y2,   X1 is X2+1, case(X1,Y1).
dessus(X1,Y1,X2,Y2) :-  case(X2,Y2), Y1 is Y2-1, X1 is X2,   case(X1,Y1).
dessous(X1,Y1,X2,Y2) :- case(X2,Y2), Y1 is Y2+1, X1 is X2,   case(X1,Y1).

% Voisin : la case [X2,Y2] est voisine directe (sans wrapping) de [X1,Y1]
% => a sa gauche, a sa droite, en dessus ou en dessous

voisin(X1,Y1,X2,Y2) :-
    %case(X1,Y1),
    ( ( Y2 is Y1, ( X2 is X1-1 ; X2 is X1+1 ), case(X2,Y2) ) ;
      ( X2 is X1, ( Y2 is Y1-1 ; Y2 is Y1+1 ), case(X2,Y2) ) ).

% la case [X2,Y2] est directement accessible depuis la case [X1,Y1]

voisinCirculable(X1,Y1,X2,Y2) :-
    %case(X1,Y1),
    ( ( Y2 is Y1, ( X2 is X1-1 ; X2 is X1+1 ), caseCirculable(X2,Y2) ) ;
      ( X2 is X1, ( Y2 is Y1-1 ; Y2 is Y1+1 ), caseCirculable(X2,Y2) ) ).


% ____________________________________________________
% DEFINITION DU NIVEAU ET POSITIONNEMENT DES ENTITES

mur(X,0) :- colonne(X). % murs haut
mur(X,Y) :- Y \= 9, ligne(Y), largeur(Xmax), X is Xmax-1. % murs droite
mur(X,Y) :- hauteur(Ymax), Y is Ymax-1, colonne(X). % murs bas
mur(0,Y) :- Y \= 9, ligne(Y). % murs gauche

mur(X,1) :- X=9.
mur(X,2) :- X=2;X=3;X=5;X=6;X=7;X=9.
mur(X,4) :- X=2;X=3;X=5;X=7;X=8;X=9.
mur(X,5) :- X=5;X=9.
mur(X,6) :- X=1;X=2;X=3;X=5;X=6;X=7;X=9.
mur(X,7) :- X=1;X=2;X=3;X=5.
mur(X,8) :- X=1;X=2;X=3;X=5;X=7;X=9.
mur(X,9) :- X=7.
mur(X,10) :- X=1;X=2;X=3;X=5;X=7;X=8;X=9.
mur(X,11) :- X=1;X=2;X=3;X=5.
mur(X,12) :- X=1;X=2;X=3;X=5;X=7;X=8;X=9.
mur(X,13) :- X=9.
mur(X,14) :- X=2;X=3;X=5;X=6;X=7;X=9.
mur(X,15) :- X=3.
mur(X,16) :- X=1;X=3;X=5;X=7;X=8;X=9.
mur(X,17) :- X=5;X=9.
mur(X,18) :- X=2;X=3;X=4;X=5;X=6;X=7;X=9.

mur(R,Y) :- mur(X,Y), symetrieHoriz(X,R), !.

% posPotion(X,Y) : potion aux coordonnees X,Y
posPotion(1,15).
posPotion(1,2).
posPotion(17,15).
posPotion(17,2).
```

```
% initPosFantome(N,X,Y) : coordonnees initiales du fantome N : [X,Y]
initPosFantome(0,8,8).
initPosFantome(1,10,8).
initPosFantome(2,10,9).
initPosFantome(3,8,9).

initialisation :-
    assert(posPacman(9,15)),
    assert(dirPacman(1)),

    initPosFantome(0,X0,Y0), assert(posFantome(0,X0,Y0)),
    initPosFantome(1,X1,Y1), assert(posFantome(1,X1,Y1)),
    initPosFantome(2,X2,Y2), assert(posFantome(2,X2,Y2)),
    initPosFantome(3,X3,Y3), assert(posFantome(3,X3,Y3)),

    assert(dirFantome(0,0)),
    assert(dirFantome(1,0)),
    assert(dirFantome(2,0)),
    assert(dirFantome(3,0)),

    assert(score(0)),

    posPacman(X,Y), % la case où se trouve Pacman ne contient pas de pastille
    assert(visite(X,Y)),

    assert(visite(8,8)),   % pas de pastille dans la "maison" des fantomes
    assert(visite(10,8)),
    assert(visite(8,9)),
    assert(visite(9,9)),
    assert(visite(10,9)).


destruction :-
    retractall(fin(_X)),           % pour pouvoir relancer le jeu
    retractall(score(_X)),
    retractall(visite(_X,_Y)),
    retractall(posPacman(_X,_Y)),
    retractall(dirPacman(_Dir)),
    retractall(posFantome(_N,_X,_Y)),
    retractall(dirFantome(_N,_Dir)),
    retractall(decomptePotion(_T)).


% ____________________________________
% Symboles textuels associés aux entités

schema(X,Y,'RiP') :- posFantome(_,X,Y), posPacman(X,Y), !.
schema(X,Y,' V ') :- dirPacman(0), posPacman(X,Y), !.
schema(X,Y,' < ') :- dirPacman(1), posPacman(X,Y), !.
schema(X,Y,' ^ ') :- dirPacman(2), posPacman(X,Y), !.
schema(X,Y,' > ') :- dirPacman(3), posPacman(X,Y), !.
schema(X,Y,' 1 ') :- decomptePotion(_T), posFantome(0,X,Y),!.
schema(X,Y,' 2 ') :- decomptePotion(_T), posFantome(1,X,Y),!.
schema(X,Y,' 3 ') :- decomptePotion(_T), posFantome(2,X,Y),!.
schema(X,Y,' 4 ') :- decomptePotion(_T), posFantome(3,X,Y),!.
schema(X,Y,' $ ') :- posFantome(0,X,Y),!.
schema(X,Y,' % ') :- posFantome(1,X,Y),!.
schema(X,Y,' § ') :- posFantome(2,X,Y),!.
schema(X,Y,' @ ') :- posFantome(3,X,Y),!.
schema(X,Y,'[X]') :- debugMode, cibleFantome(_,X,Y),!.
schema(X,Y,' + ') :- debugMode, carrefour(X,Y),!.
schema(X,Y,' L ') :- debugMode, angle(X,Y),!.
schema(X,Y,'###') :- mur(X,Y),!.
schema(X,Y,'   ') :- visite(X,Y),!.  % pastille mangée
schema(X,Y,' O ') :- posPotion(X,Y),!.
schema(_X,_Y,' . ').                  % pastille

% ____________________
% GESTION DE L'AFFICHAGE

clearScreen.                                % pour vider l'ecran
/*
clearScreen :- clearScreen(0).
clearScreen(Y) :- Y >= 20, !.
clearScreen(Y) :- nl, Y2 is Y+1, clearScreen(Y2).
*/

afficher(M) :- debugMode, write(M).       % alias de write utilisé pour afficher des traces
afficher(_).

nl2 :- debugMode, nl.                     % alias de nl utilisé pour afficher des traces
nl2.
```

```
afficherCoord(Nom1,Coord1,Nom2,Coord2) :-      % affiche des coordonnées (traces)
    afficher(Nom1),afficher('='),afficher(Coord1),afficher(' '),
    afficher(Nom2),afficher('='),afficher(Coord2),afficher(' ').

dessinercarac(X,Y) :- schema(X,Y,Signe), write(Signe).

dessinerligne(Y) :- dessinerligne(0,Y).
dessinerligne(X,_Y) :- largeur(Xmax), X >= Xmax, !.
dessinerligne(X,Y) :- dessinercarac(X,Y), X2 is X+1, dessinerligne(X2,Y).

ttDessiner(Y) :- hauteur(Ymax), Y >= Ymax, !.
ttDessiner(Y) :- dessinerligne(Y), nl, Y2 is Y+1, ttDessiner(Y2).

affichage :- clearScreen, nl, afficherScore, ttDessiner(0).   % dessine le jeu


% _____________
% GESTION DU JEU

jeu :-          % jeu = construction du niveau puis passage a la vie
    write('** jeu : initialisation'), nl,
    destruction,
    initialisation,
    vie.

jeu :- write('Fin de jeu'), nl.

vie :-          % vie = plusieurs tours jusqu'à la mort
    write('** vie'), nl,
    deplaceFantomes,
    affichage,!,
    not(mort),
    not(fin),
    deplacePacman,!,
    not(mort),
    not(fin),
    not(gagne),
    vie,
    !.

vie :- write('Fin de vie'), nl.

fin :-        %dans le cas où l'utilisateur appuie sur a.
    X is 0,
    fin(X),
    write('Fin du jeu').

restePastille :- caseCirculable(X,Y), not(visite(X,Y)).

gagne :-
    not(restePastille),
    write('Vous avez gagné !'), nl.

mort :-          % collision entre pacman et un fantome
    not(decomptePotion(_T)),
    posPacman(XP,YP),
    posFantome(_N,XP,YP),
    write('LE FANTOME VOUS A ATTRAPE'), nl.

% _________________
% GESTION DU SCORE

point(X,Y) :- posPacman(X,Y), not(visite(X,Y)).

ajoutPoint(X,Y) :-
    point(X,Y),
    score(S1),
    S is S1+1,
    retractall(score(_S)),
    assert(score(S)),
    assert(visite(X,Y)).

afficherScore :-
    write('votre score est de : '),
    score(X),
    write(X),
    write(' points'),
    nl.

afficheDureePotion(T) :-
    write('Duree d''effet de la potion : '),
    write(T), nl.
```

```
trouvePotion(X,Y) :-
    posPotion(X,Y),
    not(visite(X,Y)),
    write('Potion trouvee ! Les fantomes sont vulnerables !'), nl,
    T is 30,
    assert(decomptePotion(T)),
    afficheDureePotion(T).

testPotion(X,Y) :-
    decomptePotion(_T),
    posFantome(N,X,Y),
    write('Vous avez mange un fantome ! Score + 10'), nl,
    score(S0),
    S is S0+10,
    score(S),
    initPosFantome(N,XF,YF),
    changerPosFantome(N,XF,YF),
    fail, !.

testPotion(X,Y) :- trouvePotion(X,Y).

testPotion(_X,_Y) :-
    decomptePotion(T),
    ( ( T is 0,
        write('Effet de la potion termine'),
        retractall(decomptePotion(_T)) ) ;
      ( retractall(decomptePotion(_T)),
        T2 is T-1,
        assert(decomptePotion(T2)),
        afficheDureePotion(T2) ) ).

testPotion(_X,_Y).



% ____________________
% GESTION DE PACMAN

deplacePacman :-
    write('** deplacePacman : attenteTouche...'), nl,
    attenteTouche(K),              % attend l'appui sur une touche
    nl, write('** commandeClavier : '), write(K), nl,
    commandeClavier(K),            % reagit a la touche pressee
    posPacman(X,Y),
    dirPacman(Dir),
    dirIncrement(Dir,Xincr,Yincr),
    changerPosPacman(X,Y,Xincr,Yincr),

    afficher('Position pacman : '),
    afficher(X), afficher('+'), afficher(Xincr), afficher(', '),
    afficher(Y), afficher('+'), afficher(Yincr), nl2.


changerPosPacman(X,Y,Xi,Yi) :-
    positionSuivante(X2,Y2,X,Y,Xi,Yi),    % calcul de la nouvelle position
    ( mur(X2,Y2) ;                        % si il y a un mur on ne fait rien, sinon :
      ( retractall(posPacman(_X,_Y)),     % - suppression de la position en cours
        assert(posPacman(X2,Y2)),         % - enregistrement de la nouvelle position
        testPotion(X2,Y2),                % - gestion des potions
        ( ajoutPoint(X2,Y2) ; true ) ) ). % - gestion du score

changerDirPacman(Dir) :-
    retractall(dirPacman(_Dir)),   % supprimer la direction en cours
    assert(dirPacman(Dir)).        % enregistre la nouvelle direction


% ____________________________
% GESTION DES COMMANDES CLAVIER

commandeClavier(K) :- K == 113, changerDirPacman(3), /*write('gauche'),*/ !.
commandeClavier(K) :- K == 100, changerDirPacman(1), /*write('droite'),*/ !.
commandeClavier(K) :- K == 122, changerDirPacman(0), /*write('haut'),*/ !.
commandeClavier(K) :- K == 115, changerDirPacman(2), /*write('bas'),*/ !.
commandeClavier(K) :- K ==  97, retractall(fin(_X)), assert(fin(0)), !.    % touche a
commandeClavier(_K).

% ____________________
% GESTION DU CLAVIER

attenteTouche(K) :- get_single_char(K).  % commande SWI Prolog
```

```
% ____________________
% GESTION DES FANTOMES

deplaceFantomes :-
    deplaceFantome(0), deplaceFantome(1), deplaceFantome(2), deplaceFantome(3).

deplaceFantome(N) :-
    write('** deplaceFantome : '), write(N), nl,
    posFantome(N,X,Y),                  % récupération de la position du fantome
    retractall(posFantome(N,_X,_Y)),    % supprimer ce Fantome pour eviter qu'il ne soit un obstacle
    dirFantome(N,Dir),                  % récupération de la direction du fantome
    dirIncrement(Dir,Xi,Yi),            % récupération de l'increment a partir de la direction
    positionSuivante(X1,Y1,X,Y,Xi,Yi),  % calcul de la position prévue pour ce tour
    ( ( caseCirculable(X1,Y1), X2 is X1, Y2 is Y1 ) ; % si elle est circulable, OK
      positionSuivanteVersCible(X2,Y2,X,Y,X1,Y1) ),   % sinon, trouver une autre case
    ( ( angle(X,Y),                     % si on peut changer de direction (angle)
        redirigerFantome(N,X,Y,XF,YF), ! ) ;          % alors on cherche une autre direction
      ( XF is X2, YF is Y2 ) ),         % sinon on continue d'avancer dans la meme direction
    assert(posFantome(N,XF,YF)).        % dans tous les cas on enregistre la nouvelle position

redirigerFantome(N,X,Y,X2,Y2) :-
    write('** redirigerFantome'), nl,
    cibleFantome(N,Xc,Yc),                  % on identifie la cible de ce fantome
    dirFantome(N,ODir),                     % on recupere la direction du fantome
    ( ( trouverItineraire([X3,Y3],[X,Y],[Xc,Yc],ODir),  % [X3,Y3] est l'angle cible
        positionSuivanteVersCible(X2,Y2,X,Y,X3,Y3),     % on se déplace vers cet angle => [X2,Y2]
        Xi is X2-X, Yi is Y2-Y,             % on déduit l'incrément de ce mouvement
        dirIncrement(Dir,Xi,Yi),            % on déduit la direction de cet incrément
        changerDirFantome(N,Dir) ) /*;      % on modifie la direction du fantome
      ( X2 is X, Y2 is Y )*/ ).             % MAIS si il n'y a pas d'itinéraire, on ne bouge pas

changerPosFantome(N,X,Y) :-
    retractall(posFantome(N,_X,_Y)),        % supprimer la position en cours
    assert(posFantome(N,X,Y)).              % enregistre la nouvelle position

changerDirFantome(N,Dir) :-
    retractall(dirFantome(N,_Dir)), % supprimer la direction en cours
    assert(dirFantome(N,Dir)).      % enregistre la nouvelle direction

cibleFantome(_N,X,Y) :-         % cible des fantomes,
    decomptePotion(_T),         % dans le cas ou une potion a ete bue par pacman
    posPacman(Xp,Yp),
    dims(Xm,Ym),
    X is Xm-Xp,                 % = la direction opposee de pacman
    Y is Ym-Yp.
%   initPosFantome(N,X,Y).      % = position initiale du fantome, dans sa maison

cibleFantome(_N,X,Y) :-         % cible par défaut des fantomes => pacman
    posPacman(X,Y).
/*  posPacman(Xp,Yp), dirPacman(Dir), ( chercherAngle(X,Y,Xp,Yp,Dir) ;
      ( Dir2 is Dir+1 mod 4, chercherAngle(X,Y,Xp,Yp,Dir2) ) ;
      ( Dir2 is Dir+2 mod 4, chercherAngle(X,Y,Xp,Yp,Dir2) ) ;
      ( Dir2 is Dir+3 mod 4, chercherAngle(X,Y,Xp,Yp,Dir2) ) ). */

trouverItineraire([X2,Y2],[X1,Y1],[XA,YA], OldDir) :-
    Xdiff is XA-X1,                                     % vecteur horizontal
    Ydiff is YA-Y1,                                     % vecteur vertical
    ( ( Xincr is sign(Xdiff), Xincr \== 0 ) ; Xincr is 1 ), % increment horizontal non nul
    ( ( Yincr is sign(Ydiff), Yincr \== 0 ) ; Yincr is 1 ), % increment vertical non nul

    ( ( abs(Xdiff) > abs(Ydiff), % si pacman est plus loin horizontalement que verticalement
        dirIncrement(Dir,Xincr,0),          % alors increment majeur = horizontal
        dirIncrement(DirMin,0,Yincr) ) ;    %    et increment mineur = vertical
      ( dirIncrement(Dir,0,Yincr),          % sinon increment majeur = vertical
        dirIncrement(DirMin,Xincr,0) ) ),   %    et increment mineur = horizontal

    OppDir is (Dir + 2) mod 4,              % direction opposée de l'increment majeur
    OppDirMin is (DirMin + 2) mod 4,        % direction opposée de l'increment mineur

    % on cherche un angle dans la direction ...
    ( ( OppDir    \== OldDir, chercherAngle(X2,Y2,X1,Y1,Dir) ) ;       % majeure
      ( OppDirMin \== OldDir, chercherAngle(X2,Y2,X1,Y1,DirMin) ) ;    % mineure
      ( DirMin    \== OldDir, chercherAngle(X2,Y2,X1,Y1,OppDirMin) ) ; % mineure opposée
      ( Dir       \== OldDir, chercherAngle(X2,Y2,X1,Y1,OppDir) ) ;    % majeure opposée
    afficher('[IMMOBILISME trouverItineraire]'), fail ). % cas où rien n'est possible

chercherAngle(XA,YA,X,Y,Dir) :-
    dirIncrement(Dir,Xi,Yi),                % on recupère les increments de direction
    positionSuivante(X2,Y2,X,Y,Xi,Yi),      % on cherche la position suivante
    caseCirculable(X2,Y2),                  % si ce n'est pas circulable on s'arrete
    ( ( angle(X2,Y2), XA is X2, YA is Y2 ) ;  % si c'est un angle, on a notre solution
      ( posPacman(X2,Y2), XA is X2, YA is Y2 ) ;  % encore mieux si c'est pacman !
      ( chercherAngle(XA,YA,X2,Y2,Dir) ) ).   % sinon on continue de chercher
```

B.6) CONCLUSION

Nous étions au départ très enthousiastes concernant le sujet que nous avions choisi pour ce projet, car les atouts du langage Prolog auraient été mis en oeuvre dans plusieurs mécanismes intelligents : déplacement des fantômes et génération de niveaux (labyrinthe du jeu). Or le temps nous étant imparti pour ce projet était trop faible pour nous investir sérieusement dans ces deux mécanismes. Nous avons donc retenu le plus important : le déplacement des fantômes ; et abandonné avec regret la génération de niveaux malgré y avoir accordé de la réflexion. Dans le cadre d'un projet plus long, nous aurions voulu découvrir l'interfaçage de Prolog avec d'autres langages, tels que Java, de manière à donner du graphisme et améliorer le contrôle clavier de notre jeu.

Nous avons le regret de ne pas avoir pu nous permettre d'y passer plus de temps, néanmoins nous sommes ravis du résultat obtenu et des connaissances que nous a apporté le travail sur ce projet. Cela nous a permit de programmer (et même de penser) autrement qu'en termes d'algorithmes.