Luet Nicolas
Couvidou Laurent

IA41

Responsable de L'UV : **M. César**

Printemps 2004

SOMMAIRE

Introduction

L'objectif de l'UV IA41 est de nous initier aux concepts de l'intelligence artificielle principalement à travers deux langages : le Lisp et le Prolog.

Ainsi le projet à mener va nous permettre d'approfondir nos connaissances dans le langage Lisp. Le sujet de ce projet porte sur deux parties. La première partie, l'Architecte, porte sur la construction d'un édifice à partir de cubes avec la possibilité d'échanger deux cubes entre eux. La seconde partie, le Minotaure, porte, à partir de ce même édifice, sur la construction d'un labyrinthe en ouvrant des portes sur les différentes faces des cubes et enfin, à rechercher quel est le plus court chemin pour aller d'un cube à un autre.

Je vais donc tout d'abord expliquer les fonctions nécessaires à la partie Architecte, puis nous verrons celles du Minotaure pour enfin terminer en annexe avec celles nécessaires à l'affichage OpenGL.

I) Architecte

1) Fonctions de construction

Avant toute chose, voilà comment nous avons défini un cube :

```
(initg 'cube1 '((nom cube1) (r 1) (v 0) (b 0) (x 1) (y 1) (z 1)
                (nord cube6) (sud vide) (ouest vide) (est vide) (haut cube3) (bas vide)
                (fnord 0) (fsud 1) (fouest 0) (fest 0) (fhaut 1) (fbas 0)))
```

La première ligne nous donne son nom (ici cube1), sa couleur en RVB (ici, le cube est rouge) et ses coordonnées.
La seconde ligne nous donne ses voisins, ainsi ici, le cube1 a pour voisins le cube6 sur sa face nord et le cube3 sur sa face du haut.
Enfin, la dernière ligne est pour les portes, une valeur de 0 sur une face indique qu'il s'agit d'un mur, une valeur de 1 une porte.

Et voilà comment est définit la liste de tous les cubes :

```
; clé = nom du cube
; valeur = sur le terrain (t) ou en stock (nil)
(initg 'cubes '((cube1 t) (cube2 t) (cube3 t) (cube4 t) (cube5 t) (cube6 t)))
```

Les nouveaux cubes sont ajouté à la fin de cette liste ce qui fait de cette liste un historique avec le premier cube posé en tête et le dernier en queue.

- Posercube

```
; Poser un cube (sur le sol ou sur un autre cube le cas échéant)
; ATTENTION : les coordonnées passées doivent être impaires
; Exemple : (posercube 'cube6 '1 '3)

(defun posercube (cube abs ord)
  (if (accesg 'cubes cube)
      (print "'Erreur: ce cube existe déjà")
      (initg cube ())
      (majg cube 'nom cube)
      (majg cube 'x abs)
      (majg cube 'y ord)
      (majg cube 'z (+ (maxz (listerz abs ord) -1) 2))
      (majptcards cube)
      (if (or (trophaut cube) (troplarge (pont cube)) (not (stable (edifice cube))))
          ; si pas stable
          (print "Erreur : impossible de placer le cube ici, l'édifice s'écroulerait")
          ; si stable
          (majg 'cubes cube 't)
          (majvoisins cube)
```

t)))

La fonction posercube est la première fonction permettant de construire un édifice. On lui passe en paramètre le nom d'un nouveau cube (si le cube existe déjà, un message d'erreur nous en avertit) et les coordonnées x y où l'on souhaite le poser. La coordonnée z se calcule automatiquement grâce à maxz de listerz. On teste si le cube n'est pas placé trop haut (trohaut), s'il ne forme pas un pont trop large (troplarge de pont) et enfin si l'édifice ainsi construit ne risque pas de s'effondrer. Si toutes ces conditions sont réunies, on met à jour la liste cubes (liste de tous les cubes) et on met à jour les voisins (majvoisins).

- Collercube

```
; Coller un cube à un autre
; Exemple : (collercube 'cube7 'est 'cube5)

(defun collercube (cube face cubecible)
  (if (accesg 'cubes cube)     ;test si le cube existe déjà
     (print "'Erreur: ce cube existe déjà")
     (if (not (accesg 'cubes cubecible))    ;test si le cube cible existe
           (print "'Erreur: le cube cible n'existe pas")
           (if (not (eq (accesg cubecible face) 'vide))      ;test s'il n'y a pas déjà un cube sur la face du cube cible
                (print "'Erreur: il y a déjà un cube ici")
                (initg cube ())
                (majg cube 'nom cube)
                ; Création des coordonnées du nouveau cube selon le cube cible et sa face
                (cond ((eq face 'ouest)
                          (majg cube 'x (- (accesg cubecible 'x) 2))
                          (majg cube 'y (accesg cubecible 'y))
                          (majg cube 'z (accesg cubecible 'z)))
                         ((eq face 'est)
                          (majg cube 'x (+ (accesg cubecible 'x) 2))
                          (majg cube 'y (accesg cubecible 'y))
                          (majg cube 'z (accesg cubecible 'z)))
                         ((eq face 'sud)
                          (majg cube 'x (accesg cubecible 'x))
                          (majg cube 'y (- (accesg cubecible 'y) 2))
                          (majg cube 'z (accesg cubecible 'z)))
                         ((eq face 'nord)
                          (majg cube 'x (accesg cubecible 'x))
                          (majg cube 'y (+ (accesg cubecible 'y) 2))
                          (majg cube 'z (accesg cubecible 'z)))
                         ((eq face 'haut)
                          (majg cube 'x (accesg cubecible 'x))
                          (majg cube 'y (accesg cubecible 'y))
                          (majg cube 'z (+ (accesg cubecible 'z) 2)))
                         ((eq face 'bas)
                          (majg cube 'x (accesg cubecible 'x))
                          (majg cube 'y (accesg cubecible 'y))
                          (majg cube 'z (- (accesg cubecible 'z) 2))))
                (majptcards cube)
                (if (or (trophaut cube) (troplarge (pont cube)) (not (stable (edifice cube))))
```

```
    ; si pas stable
    (print "Erreur : impossible de placer le cube ici, l'édifice s'écroulerait")
    ; si stable
    (majg 'cubes cube 't)
    (majvoisins cube)
    t)))))
```

La fonction collercube est l'autre fonction dont nous disposons pour construire un édifice. Elle consiste à coller un nouveau cube sur la face d'un autre cube, c'est ainsi que l'on peut créer des arches ou des ponts. Les fonctions collercube et posercube sont donc parfaitement complémentaires.
Après avoir tester si le nouveau cube n'existe pas déjà, si le cube cible existe bien et si la place sur la face du cube cible est bien libre, on crée un nouveau cube (initg cube ()) auquel on donne le nom passer en paramètre. La partie qui suit est la plus longue de la fonction puisqu'il s'agit de calculer les coordonnées du nouveau cube à partir de celle du cube cible et de la face choisie. Ainsi, selon la face choisi, on recopiera les coordonnées du cube cible dans le nouveau à l'exception d'un qui variera de 2.
Enfin, on refait les même teste que dans posercube pour vérifier que l'édifice tient debout.

2) Fonctions limites

Ces fonctions sont utilisées dans posercube et collercube.

```
; Tester la hauteur d'une tour (plafond : 9)
(defun trophaut (cube)
  (if (<= (accesg cube 'z) 9)
     nil
     (print "Erreur : le cube est placé trop haut")
     t))
```

Simple test qui vérifie que la coordonnée z d'un cube ne dépasse pas 9 (plafond).

```
; Lister les cubes d'un pont
; LR : liste des cubes restant à traiter, initialisé au cube que l'on veut ajouter
; LP : liste des cubes appartenant au pont, initialisée à nil
(defun pon (LR LP)
  (cond ((null LR)              LP)
        ((app (car LR) LP)      (pon (cdr LR) LP))
        (t                      (pon (nettoiedouble (append (memeniv (car LR)) (cdr LR)) nil)
(cons (car LR) LP)))))
```

Fonction récursive qui renvoie la liste des cubes situés au même niveau que le cube donné en paramètre. Cas d'arrêt : LR null (quand il n'y a plus de cube à traiter).

```
; Appeler pon de manière simple
(defun pont (cube)
  (pon (list cube) nil))

; Tester la largeur d'un pont (maximum : 12)
(defun troplarge (LP)
  (if (<= (length LP) 12)
```

```
    nil
    (print "Erreur : le pont est trop large")
    t))
```

Fonction qui teste qu'il n'y a pas de pont de plus de 12 cubes.

3) Fonction Edifice

```
;Lister l'édifice lié à un cube
;LR : Liste des cubes restant à traiter, initialisée au nouveau cube
;LE : Liste de l'édifice déjà construit, initialisée à nil
(defun edif (LR LE)
  (cond ((null LR)              LE)     ; Cas d'arrêt -> on renvoie l'édifice
        ((app (car LR) LE)      (edif (cdr LR) LE))     ;si le cube apparaît déjà dans l'édifice, on ne le traite pas
        (t                      (edif (nettoiedouble (append (voisins (car LR)) (cdr LR)) nil) (cons (car LR) LE)))))
```

Un édifice est un cube ou un ensemble de cubes qui sont tous rattaché par au moins une face à au moins un autre cube de l'édifice.
Cette fonction permet de déterminer l'édifice d'un cube que l'on passe en paramètre. Elle s'arrête quand la liste des cubes restant à traiter est vide et elle renvoie la liste LE qui est l'édifice ainsi calculé.
(nettoiedouble (append (voisins (car LR)) (cdr LR)) nil) renvoie la liste des voisins de la tete de LR au reste de LR nettoyé de ses répétitions (dans le cas où un cube apparaît plusieurs fois).
Edif effectue cette opération de façon récursive sur les voisins du cubes puis sur les voisins des voisins et ainsi de suite jusqu'à avoir traité tous les cubes.

```
; Pour appeler edif de manière simple (juste un cube en paramètre)
(defun edifice (cube)
  (edif (list cube) nil))
```

4) Fonctions Centre de Gravité et Stabilité

```
; Liste les cubes à la base d'un édifice
; LE : édifice
; LB : liste résultat, initialisée à nil
(defun listerb (LE LB)
  (cond ((null LE)  LB)     ; Cas d'arrêt -> on renvoie la liste résultat LB
        ((= (accesg (car LE) 'z) 1) (listerb (cdr LE) (cons (car LE) LB)))
        (t                  (listerb (cdr LE) LB))))
```

Cette fonction teste si un cebe appartenant à un édifice est situé à la coordonnée z=1, auquel cas elle l'ajoute à la liste résultat LB, sinon elle passe au suivant.

```
; Pour appeler listeb de manière simple (juste l'édifice en paramètre)
(defun listerbase (LE)
  (listerb LE '()))
```

```
; Lister les coordonnées x, y d'une liste de cubes à la base d'un édifice
(defun listerxy (LB)
  (mapcar '(lambda (cube) (list (accesg cube 'x) (accesg cube 'y))) LB))
```
Cette fonction prend en paramètre le résultat de listerbase et renvoie pour chaque cube ses coordonnées x et y.

```
; Calcul du centre de gravité d'un édifice
(defun cdg (LE)
  (list
   (/ (float (apply '+ (mapcar '(lambda (cube) (accesg cube 'x)) LE))) (float (length LE)))
   (/ (float (apply '+ (mapcar '(lambda (cube) (accesg cube 'y)) LE))) (float (length LE)))))
```
La fonction cdg renvoie les coordonnées x y du centre de gravité d'une liste de cube situé au niveau du sol (z=1). Il s'agit d'une simple formule mathématique répétée pour chaque coordonnées : Xcdg=moyenne des x des cubes=(x1+x2+…+xn)/n
Ycdg=moyenne des y des cubes=(y1+y2+…+yn)/n

```
; Construire une équation avec deux points
(defun equation (pt1 pt2)
  (list '- (list '* (list '- 'x (car pt1)) (- (cadr pt2) (cadr pt1))) (list '* (list '- 'y (cadr pt1)) (- (car pt2) (car pt1)))))
```
Cette fonction prend une paire de coordonnées x y et renvoie l'équation affine associée. L'équation s'obtient par la formule mathématique suivant : (X-x1)*(y2-y1)-(Y-y1)*(x2-x1)=0

```
; Construire toutes les équations de droites possible avec une liste de coordonnées x, y
(defun equations (LPT LQ)
  (cond ((null (cdr LPT))       LQ)     ; Cas d'arrêt -> on renvoie la liste résultat LQ
        (t                      (equations (cdr LPT) (append (mapcar '(lambda (pt) (equation pt (car LPT))) (cdr LPT)) LQ))  )))
```
Cette fonction applique equation vu au-dessus à une liste de coordonnées x y (LPT) et s'arrête quand la tête du reste de LPT est null, c'est-à-dire lorsqu'il n'y a plus qu'un seul point dans la liste des points. A noté que cette fonction calcule toutes les équations possibles en prenant le premier point et en appliquant equation avec tous les autres points puis, en passant au point suivant et en recommençant avec les autres.

```
; Evaluer une équation avec les coordonnées d'un point
(defun evalequa (e pt)
  (cond ((eq (aieme-terme '(1 1 1) e) 'x)
         (eval (chaieme '(1 1 1) (car pt) (chaieme '(2 1 1) (cadr pt) e))))
        ((eq (aieme-terme '(1 1 1) e) 'y)
         (eval (chaieme '(1 1 1) (cadr pt) (chaieme '(2 1 1) (car pt) e))))))
```
Cette fonction remplace x et y d'une équation e par les x et y d'un point pt donné en paramètre. Elle vérifie tout d'abord que le aieme-terme '(1 1 1) de e est un x sinon, elle vérifie que c'est un y. A noter que e s'écrit sous la forme : (- (* (- x x1) (- y2 y1)) (* (- ***y*** y1) (- x2 x1))) et que aieme-terme '(1 1 1) et chaieme '(1 1 1) de e correspondent au terme souligné et que chaieme '(2 1 1) de e correspond au terme en italique gras.

```
; Fonction qui évalue une équation pour une liste de points
(defun evalequalpt (e LPT)
  (mapcar '(lambda (CPT) (evalequa e CPT)) LPT))
```
Cette fonction applique la fonction précédente evalequa à une liste de points. (LPT)

```
; Fonction qui évalue une liste d'équations avec un point
```

```
(defun evalequaspt (LQ pt)
  (mapcar '(lambda (e) (evalequa e pt)) LQ))
```
Cette fonction applique la fonction précédente evalequa à une liste d'équations (LQ).

```
; Fonction qui test si une liste de nombres est entièrement positif ou nul
(defun positif (Lnb)
  (every '(lambda (nb) (>= nb 0)) Lnb))

; Fonction qui test si une liste de nombres est entièrement négatif ou nul
(defun negatif (Lnb)
  (every '(lambda (nb) (<= nb 0)) Lnb))

; Fonction qui rends l'opposé d'une équation
(defun oppequa (e)
  (cons '- (list (nth '2 e) (nth '1 e))))
```
Fonction qui rend l'opposé d'une équation e, soit :
(- (* (- x x1) (- y2 y1)) (* (- y y1) (- x2 x1))) → (-(* (- y y1) (- x2 x1)) (* (- x x1) (- y2 y1)))

```
; IL FAUT SELECTIONNER LES BONNES DROITES
; Sélectionner les équations "extérieures" (le polygone de sustentation est convexe)
(defun equationsext (LPT LQ LR)
  (cond ((null LQ)                                LR)     ; Cas d'arrêt -> on renvoie la liste résultat LR
        ((positif (evalequalpt (car LQ) LPT))     (equationsext LPT (cdr LQ) (cons (car LQ) LR)))
        ((negatif (evalequalpt (car LQ) LPT))     (equationsext LPT (cdr LQ) (cons (oppequa (car LQ)) LR)))
        (t                                        (equationsext LPT (cdr LQ) LR))))
```
Cette fonction prend en paramètre une liste de points (LPT) une liste d'équations (LQ) et une liste résultat (LR) initialisé à nil et qui se verra ajouter les équations dites « extérieures » (celles qui déterminent le polynôme de sustentation). Elle s'arrête lorsque l'on a traité toutes les équations (LQ null). Si tous les résultats de evalequalpt (car LQ) LPT sont du même signe, alors il faut garder l'équation, en effet, cela signifie que tous les points sont du même côté de la droite donc celle-ci représente une droite du polynôme de sustentation. On fait en plus un test pour savoir si tous les résultat sont positif ou négatif et dans le cas où les résultats sont tous négatifs, on inverse l équation avant de l'ajouter à la liste résultat LR grâce à oppequa (car LQ) (voir ci-dessus).

```
; Fonction qui prends un point et renvoie les quatre points aux "coins"
(defun coins (pt)
  (mapcar '(lambda (cpl) (list (+ (car cpl) (car pt)) (+ (cadr cpl) (cadr pt)))) '((1 1) (-1 1) (1 -1) (-1 -1))))
```
Cette fonction prend un couple x y de coordonnées, ajoute à ce couple (1 1), (-1 1), (1 -1) et (-1 -1) et renvoie les quatre couple ainsi calculé. Ces quatre couples de coordonnées correspondent aux coordonnées x y du cube.

```
; Dans les cas particulier où on a seulement un ou deux cubes au sol,
; on élargit légèrement le polygone de sustentation en ajoutant les points des
; coins des cubes à la liste
(defun cas1ou2 (LPT)
  (cond ((= (length LPT) 1)    (coins (car LPT)))
        ((= (length LPT) 2)    (append (coins (car LPT)) (coins (cadr LPT))))
```

```
        (t                      LPT)))
```
Cette fonction prend en paramètre la liste des coordonnées x y des cubes d'un édifice situés au niveau du sol, c'est-à-dire listerxy(listerbase (edifice)). Si il n'y a qu'un point (= (length LPT) 1) alors on applique coins à ce point (et on crée ainsi 4 points pour le polynôme de sustentation). Si il n'y en a que 2 ((= (length LPT) 2), alors on applique coins à ces deux points (et on crée ainsi 8 points pour le polynôme de sustentation). Sinon, on renvoie la liste des points inchangée.

```
; Combiner equationsext et equations (pour avoir 1 seul appel à edifice)
(defun polysust (LPT)
  (equationsext LPT (equations LPT 'nil) 'nil))

; Tester la stabilité d'un edifice
(defun stable (LE)
  (positif (evalequaspt (polysust (cas1ou2 (listerxy (listerbase LE)))) (cdg LE))))
```
Enfin, la fonction stable reprend toutes les fonctions précédentes pour déterminer si un édifice est stable ou non. Son exécution est simple, elle prend les équations du polynôme de sustentation qu'elle fait tester au centre de gravité de l'édifice et elle vérifie que tous les résultats sont positif (dans ce cas là, l'édifice est stable, sinon il ne l'est pas).

5) Fonctions d'échange de deux cubes

```
; Pour un cube, renvoyer la liste des cubes de son édifice posés après lui (dans l'ordre)
(defun hist (cube edifice cubesinv LC)
  (cond ((eq (car cubesinv) cube)          (cons cube LC))     Cas d'arrêt -> on renvoie la liste résultat LC
        ((app (car cubesinv) edifice)      (hist cube edifice (cdr cubesinv) (cons (car cubesinv) LC)))
        (t                                  (hist cube edifice (cdr cubesinv) LC))))
```
Cette fonction prend un cube, son édifice, cubes inversé (cubes est la liste de tous les cubes) et une liste résultat LC initialisée à nil. Si la tête de cubesinv appartient à l'édifice de cube (app (car cubesinv) edifice) alors on ajoute ce cube à LC et on relance hist sur le reste de cubesinv (hist cube edifice (cdr cubesinv) (cons (car cubesinv) LC))). Si ce n'est pas le cas, on relance hist sur le reste de cubesinv sans ajouter de cube à LC (hist cube edifice (cdr cubesinv) LC)). Enfin, si la tete de cubesinv est égal au cube que l'on traite (eq (car cubesinv) cube), alors on s'arrête en ajoutant le cube à LC et en renvoyant LC.

```
; Rajouter les hauteurs des cubes dans l'historique (en a-List)
(defun sauvez (histo)
  (mapcar '(lambda (cube) (list cube (accesg cube 'z))) histo))
```
Cette fonction prend en paramètre la fonction hist précédente en ajoutant à chaque cube de LC sa coordonnée z. Ceci dans le but de remettre les cubes à la même hauteur qu'avant.

```
; Appeler hist de manière simple (avec sauvegarde des hauteurs)
(defun histo (cube)
  (sauvez (hist cube (edifice cube) (presents cubes nil) nil)))

; Mettre un cube dans la zone de stockage
(defun stocker (cube)
  (majg cube 'z (+ (maxz (listerz (accesg cube 'x) (+ (accesg cube 'y) 20)) -1) 2))
```

```
  (majg cube 'y (+ (accesg cube 'y) 10))
  (majg 'cubes cube nil))
```
Nous avons choisi que les cubes qui n'était pas à échanger mais qui devaient être bouger iraient dans une zone de stockage. Pour faire plus simple, nous avons décidé qu'il s'agissait seulement de déplacer les cubes selon y en leur ajoutant 10 (majg cube 'y (+ (accesg cube 'y) 10)). Leur coordonnées z, sauvée grâce à histo serait temporairement remplacé par (majg cube 'z (+ (maxz (listerz (accesg cube 'x) (+ (accesg cube 'y) 20)) -1) 2)), qui équivaut au z que posercube calcule. Enfin, on met la valeur du cube dans la liste des cubes à nil pour absent : (majg 'cubes cube nil).

```
; Mettre tous les cubes d'un historique dans la zone de stockage
(defun stockertout (histo)
  (mapcar '(lambda (cube) (stocker cube)) (mapcar 'car (reverse (cdr histo)))))
```
Cette fonction applique la fonction précédente à tous les cubes d'un historique (en fait, tous les cubes que l'on doit enlever pour accéder au cube à échanger).

```
; Enlever un cube dans la zone de stockage
(defun destocker (cube svg)
  (majg cube 'z svg)
  (majg cube 'y (- (accesg cube 'y) 10))
  (majg 'cubes cube t))
```
Cette fonction est l'opposé de stocker, à la différence près qu'elle utilise un paramètre en plus, svg, qui sert à remettre le cube à la bonne coordonnée z.

```
; Enlever tous les cubes d'un historique dans la zone de stockage
(defun destockertout (histo)
  (mapcar '(lambda (cpl) (destocker (car cpl) (cadr cpl))) (cdr histo)))
```
Cette fonction applique la fonction précédente à tous les cubes stockés. Rappel : histo est une A-list avec le nom d'un cube et sa coordonnée z.

```
; Enlever un cube et ceux qui le bloquent
(defun enlevercubes (histo cube)
  (stockertout histo)
  (majg cube 'z (+ (maxz (listerz (accesg cube 'x) (- (accesg cube 'y) 10)) -1) 2))
  (majg cube 'y (- (accesg cube 'y) 10))
  (majg 'cubes cube nil))
```
Cette fonction, en plus de lancer stockertout, déplace le cube passé en paramètre en zone de réparation. En effet, nous avons décidé que les cubes à échanger passeraient d'abord par une zone de réparation. Pour faire plus simple, nous avons décidé qu'il s'agissait seulement de déplacer le cube selon y en leur soustrayant 10 (majg cube 'y (- (accesg cube 'y) 10)).

```
; Remettre un cube et ceux qui vont le bloquer
(defun remettrecubes (cube nvx nvy nvz histo)
  (majg cube 'x nvx)
  (majg cube 'y (+ nvy 10))
  (majg cube 'z nvz)
  (majg 'cubes cube t)
  (destockertout histo))
```
Cette fonction est l'opposée de la précédente puisqu'elle replace un cube de la zone de réparation dans la zone de construction avec comme nouvelles coordonnées les anciennes du cube qu'il remplace. Enfin, elle remet les autres cubes par-dessus à leurs anciennes places.

```
; Mettre à jour l'historique "cubes" après échange des cubes
(defun majhisto (cubea cubeb)
  (initg 'cubes (mapcar '(lambda (cpl)
                           (cond ((eq (car cpl) cubea)          (cons cubeb (cdr cpl)))
                                 ((eq (car cpl) cubeb)  (cons cubea (cdr cpl)))
                                 (t                             cpl)))
                        cubes)))
```

Cette fonction prend deux cubes en paramètre et échange leurs places dans la liste cubes.

```
; Echange de deux cubes
(defun echangercubes (cubea cubeb)
  (echangercubes2 cubea (histo cubea) cubeb))

(defun echangercubes2 (cubea histoa cubeb)
  (enlevercubes histoa cubea)
  (echangercubes3 cubea histoa cubeb (histo cubeb)))

(defun echangercubes3 (cubea histoa cubeb histob)
  (enlevercubes histob cubeb)
  (echangercubes4 cubea histoa cubeb histob (accesg cubea 'x) (accesg cubea 'y)))

(defun echangercubes4 (cubea histoa cubeb histob xa ya)
  (remettrecubes cubea (accesg cubeb 'x) (accesg cubeb 'y) (cadr (car histob)) histob)
  (remettrecubes cubeb xa ya (cadr (car histoa)) histoa)
  (majhisto cubea cubeb)
  (majptcards cubea)
  (majvoisins cubea)
  (majptcards cubeb)
  (majvoisins cubeb))
```

Ces fonctions s'appelle une par une, c'est-à-dire que echangercubes appelle echangercubes2, que echangercubes2 appelle echangercubes3 … Le but de cette manœuvre est de ne pas avoir plusieurs fois à calculer un histo (fonction assez coûteuse qui utilise edifice) et d'avoir seulement comme paramètre à donner le nom des deux cubes que l'on veux échanger.
Dans un premier temps, echangercubes2 enlève les cubes bloquant le cube a et le cube a, ensuite echangercubes3 enlève les cubes bloquant le cube b et le cube b. Puis, dans echangercubes4, on remet le cube a et les cubes qui bloquaient le cube b. On remet le cube b et les cubes qui bloquaient le cube a. Pour terminer, on met à jour l'historique des cubes (l'ordre dans lequel les cubes ont été posé), on met à jour le cube a vis-à-vis de ses nouveau voisins, on met à jour ses nouveaux voisins vis-à-vis de lui et l'on fait de même avec le cube b.

6) Fonctions de recherche standard

```
; Renvoyer les cubes présents sur le terrain de construction
 (defun presents (LC LP)
  (cond ((null LC)                    LP)    Cas d'arrêt -> on renvoie la liste résultat LP
        ((cadr (car LC))        (presents (cdr LC) (cons (caar LC) LP)))
        (t                      (presents (cdr LC) LP))))
```

Cette fonction teste les valeurs de la A-List cubes, Si elle est t, on ajoute le cube à LP sinon non.

```
; Renvoyer les cubes absents du terrain de construction (en zone de stockage)
 (defun absents (LC LP)
  (cond ((null LC)              LP)    Cas d'arrêt -> on renvoie la liste résultat LP
        ((cadr (car LC))        (absents (cdr LC) LP))
        (t                      (absents (cdr LC) (cons (caar LC) LP)))))
```

Fonction inverse de la précédente.

```
; Donner le nom du cube placé en x,y,z
(defun donnercube (abs ord hau)
  (car (nettoievide (mapcar '(lambda
                               (cube)
                                (if (and (= (accesg cube 'x) abs) (= (accesg cube 'y) ord) (= (accesg
cube 'z) hau))
                                    (accesg cube 'nom)
                                   'vide))
                             (mapcar 'car cubes)))))
```

Cette fonction, qui prend en paramètre trois coordonnées x y et z, teste dans la liste cubes si les coordonnées correspondent avec un cube, si c'est le cas, on renvoie le nom du cube sinon, on renvoie vide. Une fois le mapcar terminé, nettoievide efface chaque vide de la liste retournée et on fini par prendre la tête du résultat. Ainsi, on peux avoir le nom de cube ou alors nil comme résultat.

```
; Lister les hauteurs des cubes placés à la même abscisse et la même ordonnée
(defun listerz (abs ord)
  (nettoievide (mapcar '(lambda
                          (cube)
                          (if (and (= (accesg cube 'x) abs) (= (accesg cube 'y) ord))
                             (accesg cube 'z)
                            'vide))
                       (mapcar 'car cubes))))
```

Cette fonction, qui ressemble beaucoup dans sa structure à la précédente renvoie tous les cubes ayant les mêmes abscisses et ordonnées que celles passées en paramètre.

```
; Rechercher la hauteur la plus grande d'une A-Liste avec les cubes et leur hauteur
; ATTENTION, initialiser max à -1
(defun maxz (LZ max)
  (cond ((null LZ)              max)    Cas d'arrêt -> on renvoie max
        ((>= (car LZ) max)      (maxz (cdr LZ) (car LZ)))
        (t                      (maxz (cdr LZ) max))))
```

Cette fonction prend une liste de nombre et renvoie son max.

```
; Lister les voisins d'un cube donné
(defun voisins (cube)
  (nettoievide (mapcar '(lambda (x) (accesg cube x)) '(nord sud ouest est haut bas))))

; Lister les cubes au même niveau qu'un cube donné
(defun memeniv (cube)
  (nettoievide (mapcar '(lambda (x) (accesg cube x)) '(nord sud ouest est))))
```

7) Fonctions de mise à jour des cubes

Ces fonctions servent à mettre à jour les relations de « voisins » entre cubes, majptcards met un cube à jour vis-à-vis de ses voisins alors que majvoisins met à jour les voisins vis-à-vis de ce cube. Elles font toutes deux appel à une sous-fonction qui ne traite qu'une face à la fois.

```
; Mettre à jour les clés nord, sud, etc. d'un nouveau cube
(defun majptcards (cube)
  (majunptcard cube (donnercube (+ (accesg cube 'x) 2) (accesg cube 'y) (accesg cube 'z)) 'est)
   (majunptcard cube (donnercube (- (accesg cube 'x) 2) (accesg cube 'y) (accesg cube 'z)) 'ouest)
   (majunptcard cube (donnercube (accesg cube 'x) (+ (accesg cube 'y) 2) (accesg cube 'z)) 'nord)
  (majunptcard cube (donnercube (accesg cube 'x) (- (accesg cube 'y) 2) (accesg cube 'z)) 'sud)
   (majunptcard cube (donnercube (accesg cube 'x) (accesg cube 'y) (+ (accesg cube 'z) 2)) 'haut)
  (majunptcard cube (donnercube (accesg cube 'x) (accesg cube 'y) (- (accesg cube 'z) 2)) 'bas))

; Mettre à jour une clé du nouveau cube
 (defun majunptcard (cube cubevois ptcard)
  (if (null cubevois)
     (majg cube ptcard 'vide)
     (majg cube ptcard cubevois)))

; Mettre à jour les clés des voisins du nouveau cube
(defun majvoisins (cube)
   (majunvoisin cube (donnercube (+ (accesg cube 'x) 2) (accesg cube 'y) (accesg cube 'z)) 'ouest)
  (majunvoisin cube (donnercube (- (accesg cube 'x) 2) (accesg cube 'y) (accesg cube 'z)) 'est)
  (majunvoisin cube (donnercube (accesg cube 'x) (+ (accesg cube 'y) 2) (accesg cube 'z)) 'sud)
  (majunvoisin cube (donnercube (accesg cube 'x) (- (accesg cube 'y) 2) (accesg cube 'z)) 'nord)
  (majunvoisin cube (donnercube (accesg cube 'x) (accesg cube 'y) (+ (accesg cube 'z) 2)) 'bas)
   (majunvoisin cube (donnercube (accesg cube 'x) (accesg cube 'y) (- (accesg cube 'z) 2)) 'haut))

; Mettre à jour une clé d'un voisin du nouveau cube
(defun majunvoisin (cube cubevois ptcardvois)
  (if (not (null cubevois))
     (majg cubevois ptcardvois cube)))
```

8) Fonctions de mise à jour des cubes

```
; Nettoyer la liste des "vide"
(defun nettoievide (L)
  (cond ((null (car L))          nil)     Cas d'arrêt -> on renvoie nil
        ((eq (car L) 'vide)      (nettoievide (cdr L)))      Atome à supprimer : récursivité sur le reste
        (t                       (cons (car L) (nettoievide (cdr L))))))     Atome à conserver

; Nettoyer la liste L des doublons, résultat dans LN
```

```
(defun nettoiedouble (L LN)
  (cond ((null L)                    LN)     ;Cas d'arrêt -> on renvoie la liste nettoyée LN
        ((app (car L) LN)      (nettoiedouble (cdr L) LN))     ;l'atome appartient déjà à LN, on le supprime
        (t                     (nettoiedouble  (cdr  L)  (cons  (car  L)  LN)))))         ;Atome  à conserver
```

II) Le Minotaure

1) Fonctions du labyrinthe

Les quatre fonctions suivantes sont similaires. Elles prennent comme paramètres un cube et une face et associe à cette face une valeur
0 → c'est un mur
1 → c'est une porte ouverte
2 → c'est une porte fermée à clé
3 → c'est une baie vitrée (c'est joli)
Il s'agit à chaque fois d'une simple mise à jour global d'un élément du cube.

```
; Construction d'un mur ou condamnation d'une porte
(defun mur (cube face)
  (majg cube (cadr (assoc face '((nord fnord) (sud fsud) (ouest fouest) (est fest) (haut fhaut) (bas fbas)))) 0)
  t)

; Construction d'une ouverture dans une face d'un cube
(defun porte (cube face)
  (majg cube (cadr (assoc face '((nord fnord) (sud fsud) (ouest fouest) (est fest) (haut fhaut) (bas fbas)))) 1)
  t)

; Construction d'une porte à clef dans une face d'un cube
(defun porteclef (cube face)
  (majg cube (cadr (assoc face '((nord fnord) (sud fsud) (ouest fouest) (est fest) (haut fhaut) (bas fbas)))) 2)
  t)

; Construction d'une baie vitrée dans une face cube
(defun baie (cube face)
  (majg cube (cadr (assoc face '((nord fnord) (sud fsud) (ouest fouest) (est fest) (haut fhaut) (bas fbas)))) 3)
  t)

; Construction d'un labyrinthe simple
; (Deux cubes collés sont obligatoirement ouverts sur les faces qui correspondent)
(defun labsimple (edifice)
  (mapcar '(lambda (cube) (majportes cube)) edifice)
  t)
```

Cette fonction à pour but de construire rapidement un labyrinthe en posant une porte sur les faces des cubes possédant des voisins.

```
; Mise à jour des portes cube par cube
(defun majportes (cube)
  (mapcar '(lambda (face)
             (if (not (eq (accesg cube face) 'vide))
                 (porte cube face)
               (mur cube face)))
          '(nord sud ouest est haut bas))
  t)
```

Fonction qui, pour un cube donné, teste chacune de ses face en regardant si il y a un voisin (if (not (eq (accesg cube face) 'vide)), dans ce cas, on met une porte (porte cube face) et si ce n'est pas le cas, on met un mur (mur cube face).

2) Fonctions de recherche d'un chemin

```
; Recherche d'un chemin
; cubea : cube de départ
; cubeb : cube d'arrivée
(defun chemin (cubea cubeb)
  (algoa cubea (majouvert (listeouv cubea) nil cubeb 0) (list (list nil cubea)) cubeb))
```

Cette fonction prend deux cubes, un de départ et un d'arrivé, et rend le plus court chemin pour relier les deux.

```
; Algorithme A*
; depart : le cube de départ (qui change récursivement)
; LO : la liste "ouvert" → les voies non développés
; LA : l'"arbre" du chemin parcouru
; but : le cube d'arrivée (qui ne change jamais)
(defun algoa (depart LO LA but)
  (cond ((eq depart but) (append (cdr (reconstituer LA but nil)) (list but)))
        (t                (algoa (caar LO)
                                 (majouvert (retire2 depart (listeouv (caar LO))) (cdr LO) but (cadr (car LO)))
                                 (cons (cons depart (listeouv depart)) LA)
                                 but))))
```

Help :

Arbre est récursive, j'ai rajouté « depart » qui est le cube à partir duquel tu cherches ton chemin. Il change récursivement c'est à dire qu'à chaque nouvel appel de arbre du places le cube en tête de « ouvert » dans départ >> c'est logique, il s'agit du en quelque sorte du nouveau cube de départ.

- le cas d'arrêt (2° ligne) : tu as atteint ton but, puisuqe le cube que tu traites (« départ ») est le cube d'arrivée (« but »)
- le cas récursif (3° ligne et suivantes) : tu rappelles arbre avec les nouvelles valeurs suivantes :
  - « départ » = la tete de « ouvert »
  - « ouvert » = sa mise à jour avec les nouvelles issues possibles calculées à partir du premier élément de « ouvert » (qui sera « départ » au prochain coup). Le retire2 (que tu trouveras dans fct.primitives.lisp) enlève « départ » de ces issues possibles pour éviter que l'algo ne revienne sur ses pas. C'est ce qui pouvait dans certains cas bloquer l'algo, je t'en avais parlé.
  - « arbre » = on y ajoute le cube *actuellement* traité (« départ ») et ses fils. C'est pourquoi on a besoin de « départ », c'était pas possible autrement (je sais pas si tu te souviens c'est là qu'on était bloqué mardi soir…)
  - « but » changes pas, forcément…

nb : Un élément de "ouvert" : (cube cpt heuristique)
cpt = (cadr touv)
heuristique = (cadr (cdr touv))

Fonction heuristique : estimation du chemin restant
On prend ici la distance entre deux cubes

```
(defun heuristique (cubea cubeb)
  (sqrt (+ (carre (- (accesg cubeb 'x) (accesg cubea 'x)))
           (carre (- (accesg cubeb 'y) (accesg cubea 'y)))
           (carre (- (accesg cubeb 'z) (accesg cubea 'z))))))
```

Cette fonction qui prend comme paramètre deux cubes, renvoie la distance directe qu'il y a entre ces deux cubes par la formule : $((x1-x2)^2+(y1-y2)^2+(z1-z2)^2)^{(1/2)}$

Renvoyer toutes les "issues" possibles d'un cube (liste de cubes)

```
(defun listeouv (cube)
  (nettoievide (mapcar '(lambda (face)
                          (cond ((eq (accesg cube face) 1)    (if (voisouv cube face) (accesg
cube (fasansf face)) 'vide))
                                (t                            'vide)))
                       '(fnord fsud fouest fest fhaut fbas))))
```

Cette fonction regarde dans un premier temps chacune des faces du cube donné en paramètre et pour chacune qui correspond à une porte ouverte (eq (accesg cube face) 1) on teste si le voisin à une porte qui coïncide (if (voisouv cube face)), dans ce cas, on renvoie la face. Dans tous les autres cas, on renvoie vide. Pour terminer, on efface tous les « vide » de la liste résultat.

Teste si le voisin du côté "face" du cube a une ouverture

```
(defun voisouv (cube face)
  (if (eq (accesg cube (fasansf face)) 'vide)
      nil
      (if (eq (accesg
                (accesg cube (fasansf face))
                 (cadr (assoc face '((fnord fsud) (fsud fnord) (fouest fest) (fest fouest) (fhaut fbas)
(fbas fhaut)))))
              1)
          t
          nil)))
```

Cette fonction prend un cube et une face en paramètre. Si ce cube n'a pas de voisin sur cette face (if (eq (accesg cube (fasansf face)) 'vide), la fonction rend nil, sinon, on teste si le voisin à sa face opposé à la face donné en paramètre (nord pour sud, haut pour bas…) est ouverte, c'est-à-dire égale à 1 (if (eq (accesg (accesg cube (fasansf face)) (cadr (assoc face '((fnord fsud) (fsud fnord) (fouest fest) (fest fouest) (fhaut fbas) (fbas fhaut))))) 1), alors la fonction rend t sinon, elle rend nil.

Ajout d'une liste de cubes à la liste "ouvert" et mise à jour
- Niveau dans l'arbre du nouveau cube (cpt)
- Calcul de l'heuristique pour chaque nouveau cube
- Tri de la liste créée par ordre de cpt + heuristique croissant

```
(defun majouvert (LC LO but cpt)
  (cond ((null LC)      LO)
        (t                    (majouvert     (cdr LC)
                                    (trier (list (car LC) (1+ cpt) (heuristique (car LC) but))
LO nil)
                                    but
```

```
                                        cpt))))
```

Trier un nouvel élément de "ouvert" (selon le chemin espéré croissant)

```
(defun trier (elm LO LOt)
  (cond ((null LO)
                (append LOt (list elm)))
        ((<= (+ (cadr (cdr elm)) (cadr elm)) (+ (cadr (cdr (car LO))) (cadr (car LO))))
                (append LOt (cons elm LO)))
        (t
                (trier elm (cdr LO) (append LOt (list (car LO)))))))
```

Reconstituer le chemin parcouru (en effaçant les branches mortes)

Remplacer la deuxième condition par (app svg (cdr (car arbre))) tout court pour éviter certaines erreurs de reconstituer (et en créer d'autres)

→ la vraie solution : un ID pour chaque étape

```
(defun reconstituer (arbre svg L)
  (cond ((null arbre)                                                    L)
        ((and (app svg (cdr (car arbre))) (not (app (caar arbre) L)))    (reconstituer    (cdr
arbre) (caar arbre) (cons (caar arbre) L)))
        (t                                                               (reconstituer (cdr arbre) svg
L))))
```

Fonctions utiles

fface -> face

```
(defun fasansf (face)
  (cadr (assoc face '((fnord nord) (fsud sud) (fouest ouest) (fest est) (fhaut haut) (fbas bas)))))
```

face -> fface

```
(defun sansfaf (face)
  (cadr (assoc face '((nord fnord) (sud fsud) (ouest fouest) (est fest) (haut fhaut) (bas fbas)))))
```

Dernier élément d'une liste

```
(defun dernier (L)
  (cond ((null (cdr L)) (car L))
        (t          (dernier (cdr L)))))
```

Annexe : interface 3D avec OpenGL

Bien qu'optionnel, l'affichage 3D s'avère presque indispensable pour appréhender le fonctionnement de notre projet. C'est pourquoi nous nous sommes attaché à créer une interface efficace pour réaliser cette affichage.

Nous utilisons pour cela la bibliothèque 3D la plus accessible : *OpenGL*. Pour faciliter le travail, nous nous servons aussi de la bibliothèque *glut*, qui fournit des fonctions très utiles pour la création de fenêtre ou encore de primitives. *Glut* se charge même d'inclure les fichiers d'en-tête nécessaire.

Les fichiers prenant part à l'interfaçage sont les suivants :

- **prj.interface.lisp** : il gère l'exportation de la liste de cubes dans un fichier texte (fonction *interface*), ainsi que l'interfaçage éventuel de n'importe quelle fonction du projet (fonction *interfacer*, qui rajoute un appel à *interface* à la fin de la fonction visée, en utilisant le mode de fonctionnement de *fibo* du TP1).

- **prj.interface.txt** : c'est le fichier texte en question. La première ligne contient le nombre de cubes total, et les suivantes les coordonnées spatiales et de couleur de chaque cube.

- **prj.interface.c** : c'est le fichier C qui fait appel aux bibliothèques mentionnées ci-dessus. Mentionnons le fait qu'il gère la mise à jour de l'affichage en cas de modification du fichier texte. En clair, un appel à interface sous emacs-lisp modifiera immédiatement l'affichage 3D. Techniquement, on réalise ceci en se servant de la fonction d'attente d'*OpenGL* (*glutIdleFunc*), qui observe en permanence la date de dernière modification du fichier (appel à *stat*, test sur *st_mtime*) et met à jour l'affichage s'il y a changement. Le reste du programme fonctionne comme n'importe quel programme utilisant *OpenGL* et ne présente donc pas d'orignalité particulière (et d'intérêt pour ce projet).

- **Makefile** : le classique fichier auquel fait appel le programme *make*. Il prends soin d'appeler *gcc* avec les bonnes options (inclusions de toutes les bibliothèques). À noter que nous ne sommes pour l'instant pas parvenu à compiler ni sur les stations Sun, ni sur les stations Debian. Seule une compilation sous une Red Hat personnelle a fonctionné pour l'instant.

- **prj.interface** : l'éxécutable rendu. Lui fonctionne par contre sans problème sous les stations Debian.

Conclusion

Ce projet nous a permis de nous familiariser avec le langage lisp : sa syntaxe et son fonctionnement. Nous avons pu aussi nous familiariser avec les notions de récursivité qui sont facilement compréhensible sous Lisp.

Il nous a aussi permis de nous initier à OpenGL, bien que, par manque de temps, nous n'ayons pu afficher le tavail effectué pour la partie Minotaure.

Améliorations possibles.