Université Savoie Mont Blanc, UFR SceM
L3 S&T TRI/ESET

11 janvier 2018
Durée: 1h30

INFO501 – EXAMEN

Documents autorisés : une feuille au format A4.

Directives :
- Les exercices sont indépendants et l'ordre n'est pas important.
- Tout le code demandé doit être rédigé dans le langage pseudo-code vu en cours.
- Vous pouvez utiliser sans les redéfinir tous les types de données abstraits vu en cours et en TD. Les fonctions pour les manipuler sont données aux dernières pages de cet énoncé.
- Vous pouvez toujours supposer les questions précédentes résolues, et donc utiliser les procédures et fonctions précédentes comme si vous les aviez écrites correctement.
- Le barème est donné à titre indicatif.

Question I. Trace d'un programme (/3)

Considérons le programme suivant :

```
Fonction mystère( x : E Entier ) : Entier
Variables : y : Entier initialisé à 0
Début
    Tant que x != 0 faire
        y := 10 * y
        y := y + (x%10)
        x := x/10
    Fin tant que
    retourne y
Fin
Variables : x, y, z : Entier
Début
    x := 127
    y := mystère( x )
    Affiche ( y )
    z := mystère( y )
    Affiche ( z )
Fin
```

1) Donnez une trace du programme et déterminez ce qu'il affiche.
2) Que fait la fonction mystère ?

Question II. Utilisation d'un type de données abstrait (/9)

On souhaite développer une intelligence artificielle pour jouer à divers jeux à deux joueurs (échecs, dames, morpion...). Pour ce faire, on dispose des types abstraits

Type : Jeu, qui représente un jeu (avec toutes ses règles, etc.),
Type : Position qui représente une position du jeu,
Type : Coup qui représente un coup dans le jeu,

ainsi que des fonctions

Fonction jeu_posInit(j : **E** Jeu) : Position, qui donne la position initiale du jeu,
Fonction pos_gagnée(j : **E** Jeu, p : **E** Position) : Booléen, qui répond **vrai** si, arrivé à la position p du jeu j, la partie est finie et l'on a gagné,
Fonction pos_perdue(j : **E** Jeu, p : **E** Position) : Booléen, qui répond **vrai** si, arrivé à la position p du jeu j, la partie est finie et l'on a perdu,
Fonction pos_nulle(j : **?** Jeu, p : **?** Position) : Booléen, qui répond **vrai** si, arrivé à la position p du jeu j, la partie est finie et aucun des deux joueurs n'a gagné,
Fonction pos_joueur(j : **E** Jeu, p : **E** Position) : Booléen, qui répond **vrai** si c'est à nous de jouer à la position p du jeu j,
Fonction pos_coupsPossibles(j : **E** Jeu, p : **E** Position) : Ensemble de Coups, qui donne l'ensemble des coups possibles à partir de la position p du jeu j,
Fonction coup_arrivée(j : **E** Jeu, c : **E** Coup) : Position, qui donne la position d'arrivée du coup c dans le jeu j.

Par exemple, si on considère le jeu du morpion, le type des positions pourrait être **Type :** Position : Tableau[0..2][0..2] d'Entiers, où 0 signifie que la case est vide, 1 qu'on a joué dedans et 2 que l'adversaire a joué dedans, un coup pourrait être un couple (i, j) qui dit dans quelle case jouer, et le type **Type :** Jeu contiendrait toutes les règles du jeu de morpion, comme le fait qu'on ne peut pas jouer dans une case déjà occupée, etc.

On suppose que le jeu est *fini*, c'est-à-dire qu'il est impossible de jouer des coups à l'infini (on tombe forcément sur une position où il est impossible de jouer à un moment ou à un autre).

1) Implémentez la fonction

   Fonction pos_posSuivantes(j : **?** Jeu, p : **?** Position) : Ensemble de Positions,

   qui donne l'ensemble des positions que l'on peut atteindre en un coup depuis la position p du jeu j, en remplaçant **?** par le symbole le plus approprié parmi **E**, **S** et **ES**.

2) On cherche à savoir quelles positions sont *gagnantes*, c'est-à-dire les positions à partir desquelles on est sûr de gagner si l'on joue parfaitement. Pour ce faire, on teste la position p à laquelle on est arrivée :
   — si la partie est finie, la position est gagnante si et seulement si on a gagné, sinon, il y a au moins un coup à jouer,
   — si c'est à nous de jouer, il suffit qu'il existe un coup de p vers une position gagnante pour que p soit gagnante,
   — si c'est à l'adversaire de jouer, il faut que toutes les positions vers auxquelles il peut arriver soient gagnantes.

   Le concept équivalent pour l'adversaire est celui de positions *perdantes*, à partir desquelles on est sûr de perdre si l'adversaire joue parfaitement (même si on joue aussi parfaitement). Implémentez les fonctions

   Fonction pos_gagnante(j : **?** Jeu, p : **?** Position) : Booléen, qui retourne **vrai** quand la position p du jeu j est gagnante,
   Fonction pos_perdante(j : **?** Jeu, p : **?** Position) : Booléen, qui retourne **vrai** quand la position p du jeu j est perdante.

3) Implémentez les fonctions

   Fonction jeu_gagnant(j : **?** Jeu) : Booléen, qui retourne **vrai** si, en jouant parfaitement au jeu, on est sûr de gagner,
   Fonction jeu_perdant(j : **?** Jeu) : Booléen, qui retourne **vrai** si, peu importe comment on joue, on est sûr de perdre si l'adversaire joue parfaitement.

Question III. Implémentation d'un type abstrait (/9)

Une *table de hachage* est une structure de donnée qui permet de stocker un nombre d'éléments inconnu à l'avance, sans gaspiller trop de mémoire et en ayant quand même des performances comparables à celles d'un tableau. Ici, on veut ranger des éléments de type **Elt** dans notre table de hachage.

Pour cela, on dispose d'une fonction

Fonction hachage(x : E Elt, n : E Entier) : Entier

appelée *fonction de hachage* qui, si elle est appelée sur un entier n, renvoie toujours un entier entre 0 et $n-1$. L'idée est donc, si l'on a une table de hachage de taille n, d'insérer x dans la case renvoyée par la fonction de hachage appliquée à x. Cependant, il peut y avoir des *collisions* : comme le nombre d'éléments n'est pas fixé mais la table est de taille fixe, il se peut qu'on veuille insérer un x et un y dans la même case. Du coup, au lieu de mettre juste un élément dans la case, on met un ensemble d'éléments dans chaque case de la table de hachage.

Par exemple, si on veut stocker des chaînes de caractères dans une table de hachage, on définit le type **Elt** comme étant **Chaîne** et une fonction de hachage correspondant. Ensuite si on a une table de hachage t de taille 1000 et hachage("toto", 1000) vaut 42, on stockera "toto" dans la 42ème case de t. Mais si hachage("titi", 1000) vaut aussi 42, il faudra aussi stocker "titi" dans la 42ème case de t. Une case de t contient donc un ensemble d'éléments (plutôt qu'un unique élément, comme les tableaux).

Le type **Tbl** des tables de hachage sera implémenté de la façon suivante :

Type : Tbl :
tab : Tableau d'Ensembles d'Elt
taille : Entier

Le tableau tab contient dans sa $i$ème case l'ensemble de tous les éléments x qu'on veut stocker dans la case i de la table et la variable taille contient la taille du tableau tab.

1) Implémentez la fonction

   Fonction tbl_vide(t : ? Tbl, n : ? Entier),

   qui fait de t une table de hachage vide de taille n, en remplaçant ? par le symbole le plus approprié parmi E, S et ES.

   Attention : pour que t soit vide, il faut que chacune des cases de t le soit...

2) Implémentez les fonctions

   Fonction tbl_contient(t : ? Tbl, x : ? Elt) : Booléen
   Fonction tbl_insère(t : ? Tbl, x : ? Elt)
   Fonction tbl_retire(t : ? Tbl, x : ? Elt).

   La première renvoie **vrai** si x est dans t et **faux** sinon, la deuxième insère x dans t et la troisième le retire.

3) Le but d'une table de hachage étant de garder les mêmes performances qu'un tableau, il ne faut pas qu'il y ait trop de collisions. Hors, comme nos tables de hachage ont des tailles fixes, il y aura forcément beaucoup collisions si on met beaucoup d'éléments dedans. Il faut donc être en mesure de changer la taille d'une table de hachage.

   Implémentez la fonction

   Fonction tbl_agrandir(t : ? Tbl, n : ? Entier),

   qui change t en une table de taille qui contient les mêmes éléments.

   Attention : les éléments doivent être placés dans la bonne case dans la nouvelle table.

4) On ne veut pas forcer l'utilisateur à agrandir sa table de hachage, donc on décide de le faire pour lui : dès qu'on insère plus de $2n$ éléments dans une table t de taille n, on double la taille de t.

   Décrivez les changements qui doivent être faits pour rendre cela possible de façon efficace (on ne veut pas parcourir toutes les cases de t à chaque fois qu'on insère un élément). Faites les changement nécessaires à l'implémentation.

ANNEXE : ENTÊTES DE FONCTIONS

Les fonctions suivantes ont été vues en cours ou en TD. Vous pouvez les utiliser sans les redéfinir. **Attention :** ne faites aucune hypothèse sur la manière donc sont implémentés ces types abstraits.

Ensemble

Fonction ens_init(e : **S** Ensemble)

Fonction ens_estVide(e : **E** Ensemble) : Booléen

Fonction ens_contient(e : **E** Ensemble, x : **E** Élément) : Booléen

Fonction ens_insère(e : **ES** Ensemble, x : **E** Élément)

Fonction ens_retire(e : **ES** Ensemble, x : **E** Élément)

Fonction ens_union($e1, e2$: **E** Ensemble) : Ensemble

Fonction ens_inter($e1, e2$: **E** Ensemble) : Ensemble

Fonction ens_diff($e1, e2$: **E** Ensemble) : Ensemble

Itérateur

On considère ici un itérateur défini sur un type `Conteneur` quelconque.

Fonction cnt_it_init(e : **E** Conteneur, it : **S** ConteneurIterateur)

Fonction cnt_it_suivant(e : **E** Conteneur, it : **ES** ConteneurIterateur) : Element

Fonction cnt_it_termine(e : **E** Conteneur, it : **E** ConteneurIterateur) : Booléen