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

11 janvier 2017
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 (/5)

Considérons le programme suivant :

Variables : x, y, i : Entiers, k : Entier initialisé à 1, z : Entier initialisé à 0
r : Booléen initislisé à Faux
t, u : Tableaux [0..5] de Booléens

Début
  Pour i de 0 à 5 faire **faire**
    $t[i] := x \bmod 2 == 1$
    $u[i] := y \bmod 2 == 1$
    $x := x/2$
    $y := y/2$
  Fin pour
  Pour i de 0 à 5 faire **faire**
    Si $(t[i]$ **xor** $u[i])$ **xor** r **alors**
      $z := z + k$
    Fin si
    $r := (t[i]$ **et** $u[i])$ **ou** $(t[i]$ **et** $r)$ **ou** $(u[i]$ **et** $r)$
    $k := 2 * k$
  Fin pour
  Si r **alors**
    $z := z + k$
  Fin si
  Affiche (z)
Fin

Rappel : *xor* signifie "ou exclusif". a xor b est vrai quand a est vrai ou b est vrai, sauf s'ils sont tous les deux vrais (auquel cas a xor b est faux).

1) Donnez une trace grossière du programme (inutile de donner chaque étape, mais par exemple seulement après chaque boucle) et déterminez ce qu'il affiche
   - quand x est initialisé à 17 et y à 25,
   - quand x est initialisé à 21 et y à 46.

2) Que fait ce programme ?

Question II. Utilisation d'un type de données complexes (/10)

Un *graphe* est une structure composée de "points" appelés *sommets* reliés par des "flèches" appelées *arêtes*. Il s'agit d'une structure de donnée extrêmement répandue qui modélise beaucoup de problèmes (les infrastructures routières ou ferroviaires, les réseaux câblés ou les relations d'amitiés sur les réseaux sociaux, pour n'en citer qu'une infime partie).

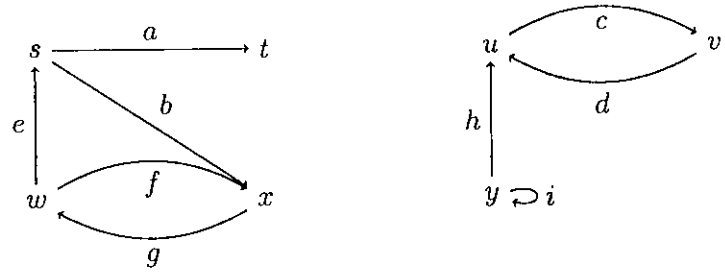

FIGURE 1 – Un graphe G.

Dans ce problème, on se pose le problème des *composantes fortement connexes* d'un graphe. Une partie P des sommets d'un graphe G est fortement connexe si, depuis tout sommet s de C, on peut atteindre n'importe quel autre sommet t de C et revenir à s. Une composante fortement connexe est une partie fortement connexe maximale (on ne peut pas rajouter de sommets de telle façon que la partie reste fortement connexe).

Par exemple, dans le graphe G de la Figure 1, les composantes connexes sont : $\{s, w, x\}$, $\{t\}$, $\{u, v\}$ et $\{y\}$. En effet, à partir de s, on peut bien atteindre s (sans suivre aucune flèche), w (en suivant b puis f) et x (en suivant b), et on peut revenir à s depuis chacun de ses sommets, c'est donc une partie fortement connexe. C'est une composante fortement connexe car, si on rajoute des sommets, soit on ne peut pas les atteindre depuis s, w et x (c'est le cas de u, v et y), soit on ne peut pas revenir sur s, w ou x une fois qu'on les a atteints (c'est le cas de t). On peut vérifier de la même manière que les autres ensembles donnés sont aussi composantes fortement connexes.

Pour représenter un graphe, on dispose :
- d'un type abstrait **Type** : Gph qui représente les graphes,
- d'un type abstrait **Type** : Smt qui représente les sommets,
- d'un type abstrait **Type** : Art qui représente les arêtes,
- d'une fonction **Fonction** gph_sommets(g : **E** Gph) : Ensemble de Smt, qui retourne l'ensemble des sommets du graphe g,
- d'une fonction **Fonction** gph_arêtes(g : **E** Gph) : Ensemble d'Art, qui retourne l'ensemble des arêtes du graphe g,
- d'une fonction **Fonction** art_source(g : **E** Gph, a : **E** Art) : Smt, qui renvoie le sommet dont l'arête a part,
- d'une fonction **Fonction** art_but(g : **E** Gph, a : **E** Art) : Smt, qui renvoie le sommet sur lequel l'arête a pointe.

Sur le graphe G de la Figure 1, on a par exemple :
- gph_sommets(G) retourne $\{s, t, u, v, w, x, y\}$,
- gph_arêtes(G) retourne $\{a, b, c, d, e, f, g, h, i\}$,
- art_source(G, e) retourne w,
- art_but(G, e) retourne s.

1) Écrivez la fonction

Fonction smt_smtSuivants(g : ? Gph, s : ? Smt) : Ensemble de Smt,

qui renvoie tous les sommets qu'on peut atteindre depuis s en suivant une seule arête, en remplaçant ? par le symbole le plus approprié parmi **E**, **S** et **ES**. Par exemple, en ne suivant qu'une seule arête, dans le graphe G de la Figure 1 :
— depuis s, on peut atteindre t et x,
— depuis x, on peut atteindre w,
— depuis t, on ne peut atteindre aucun sommet...
Rappel : pour parcourir un ensemble, vous aurez besoin d'un itérateur sur cet ensemble. Pour définir un itérateur i sur un ensembles d'éléments de type X, vous pouvez écrire i : Itérateur sur Ensemble de X.

2) Écrivez la fonction

Fonction smt_smtAtteignables(g : ? Gph, s : ? Smt) : Ensemble de Smt,

qui renvoie l'ensemble des sommets atteignables depuis s (en suivant autant d'arêtes que nécessaire). Par exemple, dans le graphe G, on peut atteindre :
— s, t, x et w depuis s,
— u et v depuis u...
Indice : maintenez un ensemble de sommets que vous savez que vous allez devoir visiter ainsi qu'un ensemble de ceux que vous avez déjà visités. Faites attention à ne pas revisiter le même sommet plus d'une fois, ou votre algorithme risque de boucler.

3) Écrivez la fonction

Fonction smt_fortementConnectés(g : ? Gph, s, t : ? Smt) : Booléen,

qui retourne **vrai** si on peut aller de s à t et de t à s.

4) Écrivez la fonction

Fonction smt_ajouterPartie(g : ? Gph, s : ? Smt, c : ? Ensemble d'Ensembles de Smt)

où c est un ensemble de parties fortement connexes, et la fonction doit ajouter s à la bonne partie.

Par exemple, dans le graphe G de la Figure 1 :
— si on appelle la fonction avec s alors que $c = \{ \{w\}, \{t\}, \{y\} \}$, elle doit modifier c pour qu'il vaille $\{ \{s, w\}, \{t\}, \{y\} \}$,
— si ensuite on la rappelle avec u, elle doit modifier c pour y ajouter $\{u\}$ car u ne se trouve dans aucune des composantes déjà définites dans c.
Indice : il est conseillé d'écrire une fonction pour savoir si un sommet doit être dans une partie fortement connexe donnée.

5) Écrivez la fonction

Fonction gph_CFC(g : ? Gph) : Ensemble d'Ensembles de Smt,

qui retourne l'ensemble des composantes fortement connexes du graphe G. Par exemple, sur le graphe G de la Figure 1, l'algorithme doit renvoyer $\{ \{s, x, w\}, \{t\}, \{u, v\}, \{y\} \}$.

Question III. Implémentation d'un type abstrait (/8)

Dans la majorité des systèmes d'exploitation modernes, pour donner l'illusion que plusieurs processus s'exécutent en même temps, il existe un processus spécial, appelé l'*ordonnanceur*, qui répartit le temps CPU entre tous les processus qui en demandent. C'est utile pour que l'utilisateur puisse avoir un lecteur MP3 qui tourne en même temps qu'il navigue sur internet, ou tout simplement pour que les tâches de fond nécessaires au bon fonctionnement du système puissent s'exécuter en même temps que les programmes de l'utilisateur.

On souhaite donc implémenter un ordonnanceur de processus très basique inspiré de celui de Linux. Il fonctionne de la manière suivante :
— un certain nombre de processus (les processus *actifs*) demandent au système d'exploitation d'avoir accès au CPU pour faire des calculs et l'ordonnanceur doit choisir le processus qui va avoir accès au CPU ;

— chacun de ces processus a une *priorité dynamique* (un entier entre -20 et 19) qui donne son degré de priorité : l'ordonnanceur choisira toujours un processus de priorité minimale ; les processus avec une priorité dynamique de -20 sont donc très prioritaires, alors que ceux de priorité 19 ne sont pas prioritaires du tout ;
— pour cela, il maintient une *file de priorité*, qui est un type de conteneur qui contient tous les processus actifs, rangés par priorité croissante, et qui fonctionne comme une **File** pour les processus d'une même priorité ;
— quand l'ordonnanceur choisit un processus à exécuter, il donne accès au CPU au processus qui est en tête de la file la plus prioritaire.

Pour l'implémenter, on dispose :
— d'un type abstrait **Type :** Processus, qui représente les processus,
— des fonctions
Fonction proc_prio(p :E Processus) : Entier, qui retourne la *priorité statique* de p, c'est-à-dire la priorité initiale de p et qui ne change jamais,
Fonction proc_estFini(p : E Processus) : Booléen, qui retourne **vrai** quand p est terminé,
Fonction proc_exécuter(p : ES Processus, n : E Entier), qui donne à p l'accès au CPU pendant n millisecondes (ce qui modifie p, par exemple, il est peut-être maintenant terminé),
Fonction proc_nouveaux(f : S File de Proc), qui met dans la file f tous les processus qui ont été créés depuis le dernier appel à proc_nouveaux.

Voici un exemple d'utilisation d'un ordonnanceur :

```
Variables : p : Processus
            o : Ordonnanceur
            f : File de Proc
Début
   ord_init( o ) // On crée un ordonnanceur qui ne connait aucun processus
   Tant que vrai faire // L'ordonnanceur ne doit jamais s'arrêter
      proc_nouveaux( f ) // On récupère les nouveaux processus créés...
      Tant que non file_estVide( f ) faire
         p := file_premier( f )
         file_défile( f )
         ord_ajouter( o, p ) // ... et on les ajoute à l'ordonnanceur
      Fin tant que
      ord_exécuter( o, 50 ) // On exécute le prochain processus
   Fin tant que
Fin
```

Le type **Ordonnanceur** sera implémenté de la façon suivante :

Type : Ordonnanceur : Tableau[$-20..19$] de Files de Proc

Les processus de priorité i sont rangés dans la file numéro i. Le processus le plus prioritaire pour l'ordonnanceur o se trouve donc dans la file $o[-20]$, sauf si elle est vide, auquel cas il se trouve dans la file $o[-19]$, etc. On parle de *priorité dynamique*, car l'ordonnanceur peut très bien choisir de changer la file dans laquelle se trouve un processus.

1) Dans un premier temps, notre ordonnanceur ne modifiera pas les priorités des processus : un processus de priorité i restera toujours dans la $i$ème file.

   a) Implémentez les fonctions

   Fonction ord_init(o : ? Ordonnanceur)
   Fonction ord_ajouter(o : ? Ordonnanceur, p : ? Processus),

   en remplaçant ? par le symbole le plus approprié parmi E, S et ES.

b) Implémentez la fonction

Fonction ord_exécuter(o : ? Ordonnanceur, n : ? Entier),

qui doit trouver le prochain processus à exécuter, l'exécuter et le remettre dans la file correspondant à sa priorité, sauf s'il est terminé.

2) Cet ordonnanceur n'est pas très bon, car si un processus très prioritaire dure longtemps, les processus moins prioritaires ne s'exécutent pas du tout pendant ce temps. On souhaite donc rafiner notre ordonnanceur pour qu'il soit performant dans ce cas. Pour ceci, on décide d'augmenter la *priorité dynamique* du processus qui vient de s'exécuter. Ainsi, tous les processus pourront s'exécuter, car les processus prioritaires deviendront moins prioritaires au fur et à mesure que le temps passe.

Réécrivez la fonction

Fonction ord_exécuter(o : ? Ordonnanceur, n : ? Entier),

en incrémentant la priorité *dynamique* du processus qui est exécuté.

3) Cet ordonnanceur n'est toujours pas très bon, car les processus qui durent longtemps finissent par être très peu prioritaires.

Écrivez la fonction

Fonction ord_reinitPrio(o : ? Ordonnanceur),

qui remet la priorité dynamique de tous les processus à leur priorité statique.

Attention : si p et p' ont la même priorité statique, mais que p devait s'exécuter avant p' (soit parce que p avait une priorité dynamique plus faible, soit parce qu'ils avaient la même priorité dynamique, mais que p était avant p' dans la file correspondante), il faut que p s'exécute encore avant p' une fois qu'on a lancé ord_reinitPrio.

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.

Chaines de caractères et caractères

Fonction getChar(c : E Chaine, *pos* : E Entier) : Caractère

Fonction setChar(c : ES Chaine, *pos* : E Entier, *valeur* : E Caractère)

Fonction taille(c : E Chaine) : Entier

Fonction init(c : S Chaine, n : E Entier, a : E Caractère)

Fonction sousChaine(c : E Chaine, *pos* : E Entier, *lng* : E Entier) : Chaine

Fonction concat($c1$: E Chaine, $c2$: E Chaine) : Chaine

Fonction asciiValue(x : E Caractère) : Entier

Fonction charValue(e : E Entier) : Caractère

Fonction carVersEntier(x : E Caractère) : Entier

Fonction LireEntier(c : E Chaine, *pos* : E Entier, *lng* : E Entier) : Entier

Files et piles

Fonction file_init(f : **S** File)

Fonction file_estVide(f : **E** File) : Booléen

Fonction file_enfile(f : **ES** File, e : **E** Élément)

Fonction file_premier(f : **E** File) : Élément

Fonction file_défile(f : **ES** File)

Fonction pile_init(p : **S** Pile)

Fonction pile_estVide(p : **E** Pile) : Booléen

Fonction pile_empile(p : **ES** Pile, e : **E** Élément)

Fonction pile_dessus(p : **E** Pile) : Élément

Fonction pile_dépile(p : **ES** Pile)

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