UPMC SORBONNE UNIVERSITÉS

UFR d'Ingénierie

8 septembre 2013

Licence d'Informatique (2013-2014)

Programmation Concurrente (LI330)

TD/TME

Thèmes 1 à 11

Thème 1

Objectifs

- environnement de travail
- révision de notions de programmation de base (en Ada)
  - structures de contrôles de base (conditions, boucles)
  - fonctions et procédures
  - récursivité
  - tableaux

Exercices

Cette semaine ne contient que des TME.

Lors de chaque TME, vous devrez créer un répertoire pour chaque exercice, y mettre les fichiers s'y rapportant et soumettre ce répertoire à la fin du TME.

Si vous n'avez pas le temps de finir ces exercices pendant les deux séances qui y sont consacrées, il est impératif que vous les ayez finis avant la séance suivante. Ils n'introduisent aucune nouvelle notion de programmation mais vous permettent de vous familiariser avec Ada sur des notions connues.

Exercice 1 – Prise en main de l'environnement

Cet exercice va vous permettre d'apprendre à compiler un programme Ada (et donc de corriger les erreurs), à l'exécuter et à le soumettre. Vous devez récupérer les fichiers :

- `exemple.adb`
- `premier_prog.adb`
- `donnees_test_1`
- `donnees_test_2`

Question 1

Compilez le programme `exemple.adb` avec la commande `gnatmake exemple.adb` (ou `gnatmake exemple`) et exécutez le programme obtenu (l'exécutable s'appelle `exemple`).

Question 2

Compilez le programme `premier_prog.adb`. Corrigez les avertissements et les erreurs (les messages du compilateur Ada sont assez explicites pour que vous puissiez corriger les erreurs seul). Une fois tous les avertissements et erreurs corrigés, exécutez le programme obtenu.

Question 3

Si vous ne l'avez pas déja fait, exécutez le programme en saisissant une valeur négative. Modifiez le programme pour que l'erreur constatée ne se produise plus.

Question 4

Modifiez le programme `premier_prog.adb` pour pouvoir calculer plusieurs factorielles. Votre programme doit s'arrêter lorsque le nombre saisi est négatif. Faites attention au type des variables pour ne pas vous retrouver avec le problème vu dans la question précédente.

Pour tester votre programme vous pouvez l'exécuter en redirigeant l'entrée standard sur les fichiers de test fournis. La commande `premier_prog < donnees_test_1` doit produire l'affichage suivant :

```
Saisissez un entier : La factorielle de  1 est         1
Saisissez un entier : La factorielle de  2 est         2
Saisissez un entier : La factorielle de  3 est         6
Saisissez un entier : La factorielle de  4 est        24
Saisissez un entier : La factorielle de  5 est       120
Saisissez un entier : La factorielle de  6 est       720
Saisissez un entier : La factorielle de  7 est      5040
Saisissez un entier : La factorielle de  8 est     40320
Saisissez un entier : La factorielle de  9 est    362880
Saisissez un entier : La factorielle de 10 est   3628800
Saisissez un entier : La factorielle de 11 est  39916800
Saisissez un entier : La factorielle de 12 est 479001600
Saisissez un entier : La factorielle de  0 est         1
Saisissez un entier :
```

La commande `premier_prog < donnees_test_2` doit produire l'affichage suivant :

```
Saisissez un entier : La factorielle de  8 est     40320
Saisissez un entier : La factorielle de  5 est       120
Saisissez un entier : La factorielle de  7 est      5040
Saisissez un entier : La factorielle de  6 est       720
Saisissez un entier :
```

Question 5

Modifiez `premier_prog.adb` en remplaçant la fonction `factorielle` par une fonction récursive.

Question 6

Supprimez les fichiers produits par la compilation (`.o`, `.ali` et l'exécutable) et soumettez le répertoire contenant le fichier `premier_prog.adb`. Pour cela, vous devez aller dans la rubrique `soumission` du site de l'ue (`http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2013/ue/LI330-2013oct/`). Vous visualisez alors le formulaire de la Figure 1, à partir duquel vous pouvez soumettre un répertoire (en donnant son nom absolu).

FIGURE 1 – Formulaire de soumission

Exercice 2 – Matrices

Cet exercice va vous permettre de manipuler des tableaux en Ada. Vous devez récupérer les fichiers :
- `matrice.ads`
- `matrice.adb`
- les huit programmes de test `test_XXXX.adb`

– les huit fichiers de données de test associés aux programmes `donnees_test_XXXX`
– `resultats_tests`

Les tableaux que vous allez manipuler sont non-contraints. Un type tableau non-contraint ne définit pas la taille des tableaux. Deux tableaux de ce même type peuvent donc avoir des tailles différentes et des indices différents. Les indices d'un tableau en Ada ne commencent pas obligatoirement à 0 (ni 1). Des attributs existent pour pouvoir manipuler ces tableaux sans connaissance de leur taille :

– déclaration d'un type tableau d'entiers, non-contraint, à deux dimensions
`type tableau is array (integer range<>, integer range<>) of integer;`
– déclaration d'une variable de type `tableau`, les contraintes doivent être définies lors de la déclaration
`monTableau : tableau(1..13,-4..9);`
– indices du premier et du dernier élément d'une dimension :
`monTableau'first(1)` et `monTableau'last(1)` pour la première dimension, qui correspondent respectivement à `1` et `13` dans l'exemple ;
– nombre d'éléments d'une dimension :
`monTableau'length(2)` pour la la deuxième dimension, qui correspond à `14` dans l'exemple ;
– intervalle des indices d'une dimension :
`monTableau'range(1)` pour la première dimension, qui correspond à l'intervalle `1..13` dans l'exemple.

Question 1

Les fichiers `matrice.ads` et `matrice.adb` contiennent les définitions de types, fonctions et procédures permettant de manipuler des matrices. Vous ne devez pas modifier le fichier `matrice.ads`. Vous testerez ce que vous avez écrit en utilisant les programmes `test_XXXX.adb`, ce sont eux qui doivent être compilés (le compilateur Ada se débrouille tout seul pour calculer les dépendances). Pour compiler ce que vous avez écrit, vous devez exécuter la commande `gnatmake test_XXXX.adb`. Les programmes dont le nom contient le mot `dim` testent que vos fonctions et procédures sont bien indépendantes des intervalles d'indices de vos matrices.

Complétez le fichier `matrice.adb` et testez vos fonctions et procédures en exécutant les programmes de test. Pour comparer votre exécution à notre exécution de référence, vous devez rediriger l'entrée de votre programme de la façon suivante : `./test_XXXX < donnees_test_XXXX`. Le fichier `resultats_tests` contient les résultats que vous devez obtenir.

Question 2

Supprimez les fichiers produits par la compilation (`.o`, `.ali` et l'exécutable) et soumettez le répertoire contenant le fichier `matrice.adb`.

Exercice 3 – Polynômes

En plus de continuer à vous familiariser avec les tableaux non contraints en Ada, cet exercice va vous permettre de manipuler des tranches de tableaux.

Vous récupérez les fichiers :
– `polynomes.ads`,
– `jeu_de_test.ads`,
– `polynomes.adb`,
– `test_polynome.adb`,
– `resultat_test`.

Dans cet exercice, nous manipulons des polynômes à coefficients entiers. Un polynôme est vu comme une somme de monômes, chacun étant composé d'un coefficient et d'un degré. Plusieurs représentations des polynômes sont possibles. Celle qui nous intéresse consiste à représenter un polynôme sous forme d'un tableau, indexé sur entier et à éléments entiers. La position de l'index du tableau par rapport à la valeur initiale de l'index représente la valeur du degré du monôme et le contenu de la case indexée son coefficient.

Ainsi, le polynôme $1 + 3x + 8x^3$ est représenté par le tableau (1,3,0,8), quel que soit l'intervalle de progression de son index.

Le fichier `polynomes.ads` contient la définition du type `T_PP` utilisé pour représenter les polynômes :

```
type T_PP is array (Natural range <>) of Integer;
```

ainsi que les prototypes des fonctions et procédures de manipulation des polynômes.

Le fichier `polynomes.adb` contient le corps des fonctions et procédures. La fonction d'affichage d'un polynôme vous est donnée. Les autres fonctions sont à compléter (elles retournent toutes une valeur `0` dans le fichier que vous récupérez, ceci vous permet de le compiler même si vous n'avez pas complété toutes les fonctions).

Les déclarations des variables de type polynôme qui ont été utilisées pour produire le fichier `resultat_test` se trouvent dans le fichier `jeu_de_test.ads`. Il s'agit des variables `PP1`, `PP2` et `PP3` représentant respectivement les polynômes $1 + x + x^2 + x^3 + x^4$, $1 + x + x^3$ et $4x^2 + 5x^3 + 2x^4 + x^5$:

```
PP1 : T_PP(1..5) := (1,1,1,1,1);
PP2 : T_PP(2..5) := (1,1,0,1);
PP3 : T_PP(1..6) := (0,0,4,5,2,1);
```

Dans chaque polynôme `PPi`, le monôme d'index `I` (si `I` appartient à `PPi'range`) est de degré `I - PPi'first`.

Le fichier `test_polynome.adb` contient le programme de test qui a été utilisé pour produire les résultats fournis dans le fichier `resultat_test`. C'est donc ce fichier qui est à compiler pour tester vos fonctions et procédures. Attention, l'exécution du programme doit être lancée avec un paramètre indiquant le numéro du test à effectuer. Vous pouvez donc commencer les tests avant d'avoir écrit toutes les fonctions :

- `./test_polynome 1` teste la procédure d'affichage,
- `./test_polynome 2` teste la fonction d'évaluation d'un polynôme,
- `./test_polynome 3` teste la fonction de calcul de la dérivée d'un polynôme,
- `./test_polynome 4` teste la fonction d'addition de deux polynômes.

Le seul fichier que vous devez modifier est le fichier `polynomes.adb`. Vous pouvez aussi enrichir les fichiers `jeu_de_test.ads` et `test_polynome.adb`.

Question 1

Le langage Ada dispose d'un opérateur "puissance" : en Ada, x^p s'écrie `x**p`.

Écrivez le corps de la fonction `Evalue_PP` qui évalue le polynôme pour une valeur donnée de sa variable. Cette fonction ne devra pas faire d'hypothèse quant à l'intervalle de progression des indices du tableau représentant le polynôme à évaluer. Sa spécification est donnée ci-après :

```
function Evalue_PP(PP : T_PP; X : Integer) return Integer;
```

Question 2

Écrivez le corps de la fonction `Derive_PP` qui dérive le polynôme (selon son unique variable, implicite). Cette fonction ne devra pas faire d'hypothèse quant à l'intervalle de progression des indices du tableau représentant le polynôme à évaluer, le polynôme résultant aura le même intervalle d'indices. Sa spécification est donnée ci-après :

```
function Derive_PP(PP : T_PP) return T_PP;
```

Question 3

Écrivez le corps de la fonction `Somme_PP` qui retourne le polynôme résultant de la somme de deux polynômes passés en argument. Il n'y a pas d'hypothèse sur l'intervalle de progression des indices, ni sur la taille des polynômes à sommer. Vous pourrez avantageusement construire une fonction de sommation auxiliaire `Somme_Aux` qui suppose que le premier polynôme est le plus court (possède le moins de monômes) et le second en possède le plus, cela vous réduira le nombre de cas à analyser . La spécification de `Somme_PP` est la suivante :

```
function Somme_PP(PP1,PP2 : T_PP) return T_PP ;
```

Question 4

Soumettez votre travail.

Thème 2

Objectifs

- découpage d'un programme en tâches ;
- partage du processeur et représentation par chronogrammes ;
- prise en main de la bibliothèque graphique.

Exercices

Le premier est à faire en TD, les deux derniers en TME.

La bibliothèque graphique

Les applications graphiques permettent de mettre en évidence l'exécution concurrente de tâches. Pour ces applications, nous utiliserons un paquetage `graphique` contenant une interface minimale avec la librairie `SDL`. La spécification de ce paquetage est donnée ci-dessous :

```
-- Encapsulation d'une bibliotheque graphique BASIQUE (voire elementaire;-))
package Graphique is

    -- Creation d'une fenetre associee au programme
    procedure Creer_Fenetre (Width  : in Natural;
                             Height : in Natural;
                             Title : in String) ;

    -- Remplissage de la fenetre avec la couleur en parametre
    procedure Remplir_Fenetre(Color : in String);

    -- Le programme attend la fermeture de la fenetre pour se terminer
    procedure Attendre_Fermeture_Fenetre;

    -- Affichage d'un point de couleur Color
    procedure Afficher_Point (X, Y : in Natural; Color : in String);

    -- Affichage d'une ligne de couleur Color
    procedure Afficher_Ligne (X1, Y1, X2, Y2: in Natural; Color : in String);

end Graphique;
```

La compilation se fera en utilisant le fichier `Makefile` fourni en TME, convenablement paramétré avec le nom de votre programme. La variable `GRAPHLOC` doit être initialisée avec le chemin d'accès aux fichiers `graphique.ad*`.

Avant de dessiner, il faut créer une fenêtre. L'appel à la fonction `Attendre_Fermeture_Fenetre` en fin de programme est nécessaire pour que la fenêtre ne se ferme pas automatiquement lorsque les opérations de calcul et de dessin effectuées par le programme sont terminées. C'est la fermeture de la fenêtre qui provoquera la terminaison du programme.

Exercice 4 – Tracé d'un triangle

Dans cet exercice, nous voulons expérimenter différentes manières d'utiliser les tâches pour tracer un triangle. Nous utiliserons pour cela le paquetage `graphique`.

Question 1

Écrivez une tâche réalisant ce tracé.

On souhaite maintenant utiliser deux tâches : la tâche principale s'occupe des opérations de gestion de la fenêtre graphique, et une deuxième tâche s'occupe des opérations de dessin.

Question 2
Écrivez le code de la tâche de dessin.

Question 3
Comment écrire le code de la tâche principale pour garantir que la fenêtre est créée avant le démarrage de la tâche de dessin et qu'elle n'est fermée qu'une fois le dessin terminé ?

Question 4
On suppose qu'une commutation n'intervient que lorsque la tâche en cours d'exécution est bloquée. Représentez l'exécution de ce programme sous forme d'un chronogramme.

On veut maintenant introduire du parallélisme en découpant l'opération de dessin en trois tâches, chacune étant chargée de tracer un côté du triangle.

Question 5
Les tâches réalisant le tracé exécutent toutes le même code, avec simplement des coordonnées de lignes différentes. Donnez une définition de type permettant ensuite la déclaration des tâches.

Question 6
Peut-on utiliser un tableau pour déclarer les trois tâches ? Pourquoi ?

Question 7
Écrivez le corps d'une tâche qui trace un côté et le corps de la tâche principale, en respectant les mêmes contraintes de synchronisation que dans la version à deux tâches. Donnez un chronogramme correspondant à l'exécution de ce programme.

Exercice 5 – Affichages en parallèle

Le but de cet exercice est de vous faire constater l'exécution en parallèle des tâches et l'aspect non déterministe des commutations.

Question 1
Complétez le programme suivant de façon à ce que chaque tâche `boucle` affiche les `fin` premiers entiers et signale sa terminaison par un message. Chaque fois qu'elle fait un affichage, une tâche `boucle` s'identifie (son identifiant a du lui être affecté lors de sa création).

```
with ada.text_io;
use ada.text_io;

procedure affichage is
fin : constant natural := 10;

-- declaration d'un type tache avec une entier naturel comme discriminant
task type boucle -- A completer

-- Corps de la tache qui affiche les "fin" premiers entiers
-- (en s'identifiant a chaque affichage) et qui signale a la fin qu'elle a termine
task body boucle
    -- A completer
end boucle;

-- Declaration de 6 taches avec des discriminants differents

begin
  put_line(" --Debut tache principale--");

  put_line(" --Fin tache principale--");
end affichage;
```

Question 2
Exécutez plusieurs fois le programme précédent, que constatez-vous ? Que pouvez-vous dire sur la terminaison de la tâche principale ?

Question 3
Modifiez votre programme pour que les tâches `boucle` soient déclarées dans la partie instructions de la tâche principale, à l'intérieur d'un bloc et avant la dernière instruction d'affichage. Que constatez-vous lors de l'exécution du programme ?

Exercice 6 – Variations sur le tracé d'un triangle

Pour commencer à visualiser le parallélisme, nous allons repartir du tracé du triangle tel que nous l'avons vu en TD, avec une tâche différente pour tracer chaque côté du triangle. Afin que chaque tâche n'exécute pas une unique opération, nous allons calculer les coordonnées des points qui se trouvent au milieu de chacun des côtés, et afficher un côté en traçant séparément les deux segments qui le composent.

Question 1
Reprenez la procédure de tracé d'un triangle, en modifiant le corps d'une tâche de dessin afin d'effectuer le tracé d'un côté en deux opérations. Pour mettre en évidence le parallélisme entre les tâches, insérez des instructions **`delay`** dans votre programme.

On veut maintenant tester l'exécution lorsque les tâches de dessin ne sont plus déclarées dans un bloc mais directement dans la partie déclarative de la tâche principale.

Question 2
Modifiez votre programme en conséquence, et ajoutez une instruction **`delay`** au début de la tâche principale. Que constatez-vous ?

Pour mieux visualiser le parallélisme, nous souhaitons maintenant tracer chaque côté du triangle point par point. Le problème est que les points de la fenêtre de dessin ont des coordonnées entières, ce qui n'est pas le cas de tous les points d'une droite. Prenons par exemple le segment qui joint le point de coordonnées $(10, 10)$ au point de coordonnées $(60, 20)$. L'équation correspondante est $y = 0.2x + 8$. Donc lorsque $x = 11$, on devrait avoir $y = 10.2$, ce qui n'est pas représentable en coordonnées entières.

L'algorithme de Bresenham trace une ligne en choisissant toujours le point de coordonnées entières le plus proche du point réel de la droite. Dans notre exemple, le point de coordonnées $(11, 10.2)$ est entre les points $(11, 10)$ et $(11, 11)$, mais plus proche du point $(11, 10)$: c'est donc celui-ci qui est choisi pour poursuivre le tracé.

L'algorithme manipule les quantités $dx =\mid x_1 - x_0 \mid$ et $dy =\mid y_1 - y_0 \mid$, où (x_0, y_0) et (x_1, y_1) sont les points aux extrémités de la droite. La pente de la droite est donnée par le rapport dy/dx (0.2 dans notre exemple). Elle correspond à la modification qu'il faut effectuer sur la coordonnée y chaque fois que la coordonnée x augmente de 1. Si la pente est ≤ 1, le choix de la valeur de y se fait toujours entre deux entiers consécutifs (10 et 11 dans notre exemple). Pour se ramener à ce cas simple quand la condition n'est pas vérifiée, il suffit d'inverser les coordonnées : on utilise y comme variable de parcours, et on cherche la meilleure valeur pour x.

Revenons maintenant au choix du point à tracer. En affichant le point $(11, 10)$, on a introduit une erreur $e = 0.2 = dy/dx$ sur la valeur de y. Pour $x = 12$, on a $y = 10.4$, on trace donc le point $(12, 10)$, et l'erreur e est maintenant $0.2 + 0.2 = 2 * dy/dx$. mais lorsque $x = 13$, $y = 10.6$, on choisit donc le point $(13, 11)$: lorsque l'erreur cumulée e (la différence entre la vraie valeur de y et la valeur choisie) devient ≥ 0.5, on augmente donc la valeur de y. Il faut alors tenir compte du fait que le point affiché se trouve maintenant au-dessus de la droite réelle.

Chaque fois qu'on incrémente la valeur de y, on va donc décrémenter de 1 la valeur de e qui, dans l'exemple, passe de 0.6 à -0.4.

Il reste le problème du typage. L'opération `e := e + dy/dx` suivie de la comparaison `e >= 0.5` impose de manipuler des nombres réels, ce que nous voulons justement éviter ici.

Pour contourner le problème, nous allons multiplier par `2*dx` la valeur ajoutée à l'erreur à chaque nouveau point (ce qui revient de fait à multiplier l'erreur par `2*dx`) et comparer la valeur de e à `dx`. Attention dans ce cas à la correction de l'erreur cumulée lorsqu'on modifie la valeur de y : puisqu'on multiplie par `2*dx` la valeur de l'erreur, il faut aussi multiplier la correction par cette valeur.

Question 3

On se place dans le cas où $x_0 < x_1$, $y_0 < y_1$ et $dy/dx \leq 1$. L'algorithme de tracé de la ligne s'écrit alors :

```
x = x0, y = y0
repeter
   afficher (x, y)
   incrementer x
   e = e + 2.dy
   si e >= dx
      incrementer y
      e = e - 2.dx
   fsi
jusqu'a x = x1
```

Transformez cet algorithme en une procédure qui permet de tracer point par point une ligne du point (x_0, y_0) vers le point (x_1, y_1),

Testez votre programme en créant une fenêtre de dimensions 600×600 pixels et en y traçant une ligne horizontale de longueur 300, puis une ligne du point $(10, 320)$ au point $(410, 400)$.

Insérez une instruction **`delay`** `0.1;` après chaque affichage de point. Que constatez-vous ?

Question 4

Complétez la procédure pour qu'elle puisse tracer une ligne quelles que soient les coordonnées des points aux extrémités. Insérez-la dans le programme de tracé du triangle en ajoutant des instructions **`delay`** pour visualiser le parallélisme.

Thème 3

Objectifs

- communication par point d'entrée : rendez-vous
- attente sur conditions multiples (**`select`**)
- tableaux de tâches
- cohérence d'une ressource partagée

Exercices

Le premier exercice sera fait en TD et les deux suivants en TME

Exercice 7 – Un système producteur/consommateur

Nous souhaitons écrire un programme dans lequel une tâche `Gestionnaire` contrôle un stock qui peut être alimenté par une tâche `Produc` et dans lequel une tâche `Conso` peut venir retirer des éléments.

Seul le gestionnaire peut accéder au stock. Lorsque le producteur (resp. le consommateur) veut effectuer un dépôt (resp. un retrait), il doit contacter le gestionnaire et celui-ci effectue l'opération si elle est possible.

Question 1

Le stock est défini par son contenu (un tableau dans lequel sont stockés les éléments déposés) et le nombre d'éléments qu'il contient effectivement. Un élément du stock est ici simplement un entier naturel.

Donnez les définitions de type à partir desquelles on pourra par la suite créer des stocks de différentes tailles.

Nous nous intéressons dans un premier temps uniquement au gestionnaire et au producteur. Lorsqu'il souhaite faire un dépôt, le producteur transmet au gestionnaire l'élément à déposer.

Question 2

Donnez la spécification de la tâche `Gestionnaire` et de la tâche `Produc`.

Question 3

On définit dans la tâche principale une constante représentant la taille du stock :

```
Dimension : constant Positive := 5;
```

Écrivez le corps des tâches `Gestionnaire`, `Produc`, et de la tâche principale, dans le cas où le producteur effectue un nombre de dépôts correspondant à la taille du stock. Chacun de ces dépôts doit être traité par le gestionnaire.

Question 4

On suppose (arbitrairement) que la tâche `Produc` est activée avant la tâche `Gestionnaire` et on fait l'hypothèse que les commutations n'ont lieu que lorsqu'une tâche est bloquée.

Représentez, sous forme d'un chronogramme, l'exécution de ce système de tâches jusqu'à la fin de la deuxième itération de la tâche `Produc`.

On complète maintenant le système de tâches en ajoutant une tâche `Conso` qui peut, par l'intermédiaire du gestionnaire, retirer des éléments du stock. Lors d'un retrait, la tâche `Conso` récupère le numéro de l'élément retiré.

Question 5

Donnez la spécification de la tâche `Conso` et la nouvelle spécification de la tâche `Gestionnaire`.

Question 6

On suppose que le consommateur effectue un nombre de retraits égal à la taille du stock (ce qui correspond au nombre

de dépôts). Le gestionnaire est susceptible, à tout instant, de traiter une requête en provenance du producteur ou du consommateur.

Écrivez le corps de la tâche `Conso` et modifiez le corps de la tâche `Gestionnaire` pour prendre en compte ces nouvelles contraintes.

Question 7
Comment faire pour éviter un dépôt dans un stock plein ou un retrait dans un stock vide ?

Question 8
Comment peut-on gérer proprement la terminaison du système de tâches si le gestionnaire ne connaît pas a priori le nombre de dépôts ou de retraits qui vont être effectués ?

Exercice 8 – Un système multi-producteurs/multi-consommateurs
Nous souhaitons dans cet exercice construire un système multi-producteurs/multi-consommateurs.

Le service de livraison que nous avons vu dans l'exercice précédent reposait sur une gestion centralisée : une seule tâche, le gestionnaire, accédait au stock. Pour diminuer la centralisation, nous choisissons dans un premier temps de supprimer la tâche `Gestionnaire` : le stock devient une ressource directement partagée par les producteurs et les consommateurs, qui peuvent y accéder de manière concurrente. Cette ressource est implémentée sous forme de variable dans la tâche principale qui sera partagée par toutes les tâches ce qui va provoquer des problèmes de cohérence.

Pour pouvoir tracer les opérations effectuées, nous affectons aux tâches `Produc` un discriminant de type entier. Chaque élément stocké est maintenant étiqueté à la fois par un numéro et par l'identité du producteur qui l'a déposé.

Question 1
Donnez la nouvelle définition des types permettant de déclarer le stock.

Le système comporte un ensemble de producteurs, identiques à la valeur du discriminant près, et un ensemble de consommateurs qui ont tous le même comportement. Chaque producteur (resp. consommateur) effectue un nombre de dépôts (resp. retraits) égal à la taille du stock.

Question 2
Donnez la spécification et le corps des tâches `Produc` et `Conso`.

Question 3
Écrivez la partie déclarative et le corps de la tâche principale, pour un système avec 3 producteurs et 3 consommateurs.

Question 4
Introduisez des instructions **`delay`** et des affichages dans le corps des tâches `Produc` et `Conso` pour mettre en évidence les problèmes de cohérence sur le stock (vous verrez la semaine prochaine en TD comment assurer la cohérence du stock sans tâche `Gestionnaire`).

Exercice 9 – Tracé d'un triangle : le retour
Pour illustrer la synchronisation de tâches par points d'entrée, nous reprenons le tracé du triangle (toujours en utilisant le paquetage `graphique`), en supprimant les discriminants des tâches qui tracent les lignes.

Une tâche de dessin reçoit maintenant les coordonnées de la ligne à tracer par l'intermédiaire d'un *point d'entrée*.

Question 1
Écrivez la nouvelle déclaration de type, à partir de laquelle les tâches de dessin seront créées.

Question 2
On s'interdit ici d'utiliser un bloc pour déclarer les tâches de dessin dans le corps de la tâche principale. Écrivez la

partie déclarative et le corps de la tâche principale, de manière à respecter la contrainte qu'une tâche ne peut pas commencer son tracé tant que la fenêtre n'a pas été créée.

Question 3

Écrivez le corps d'une tâche de dessin, en assurant que les tracés se font en parallèle. Pour mettre en évidence ce parallélisme, insérez des instructions **`delay`** dans votre programme.

Question 4

Vérifiez qu'on n'a plus de parallélisme si le traitement de l'affichage est effectué à l'intérieur du point d'entrée.

Thème 4

Objectifs

- cohérence de variables en cas d'accès concurrents
- types protégés

Exercices

Le premier exercice sera fait en TD et le second en TME.

Exercice 10 – Producteurs/consommateurs et objets protégés

Nous avons vu que l'accès direct à une même donnée (ressource partagée) par plusieurs processus posait des problèmes de cohérence. Nous reprenons le problème de la gestion de stock en partant de l'approche centralisée avec une tâche `Gestionnaire`, plusieurs producteurs et plusieurs consommateurs pour voir comment l'adapter à une gestion distribuée. Nous allons pour cela nous intéresser à la nature des tâches impliquées dans ce système.

Question 1

Comment proposez-vous de définir une "tâche active" ? Même question pour une "tâche passive".

Question 2

Quel est le comportement type d'une "tâche passive" ? Pouvez-vous donner le code "type" d'une telle tâche ?

Question 3

Qu'apporte l'utilisation d'un objet protégé par rapport à une tâche passive ?

Revenons au problème de la gestion de stock avec une tâche `Gestionnaire` dont la spécification est donnée ci dessous :

```
task Gestionnaire is
  -- Depot d'un element d'idendite Id par le Produc d'identite Id_P
  entry Depot(Id : in Natural; Id_P : in Natural);
  -- Retrait d'un element d'idendite Id
  entry Retrait(Id : out Natural);
end Gestionnaire;
```

Les types nécessaire à la définition du stock sont donnés ci-dessous :

```
Taille_Buffer : constant Natural := 10;
type T_Stock_elem is record
  No_Element : Natural;
  Id_Prod : Natural;
end record;
type Tab_Buffer is array(1..Taille_Buffer) of T_Stock_elem;

-- type du stock
type T_Stock is record
  Nb_Elements : Natural := 0;
  Tab_Stock : Tab_Buffer;
end record;
```

Question 4

Écrivez la déclaration de l'objet protégé destiné à remplacer la tâche `Gestionnaire`, sachant qu'un dépôt ne peut avoir lieu que si le stock n'est pas plein et un retrait que si le stock n'est pas vide.

Question 5
Écrivez le corps de l'objet protégé.

Question 6
Nous souhaitons avoir la possibilité de supprimer du stock un élément n'ayant pas encore été retiré. Nous ajoutons donc à l'objet protégé une entrée (ou fonction ou procédure, à vous de choisir et de justifier votre choix) prenant en paramètre les identifiants de l'élément et du producteur. Contrairement au retrait, on souhaite pouvoir choisir l'élément à supprimer.

Modifiez la déclaration et le corps du type protégé `Gestionnaire`.

Question 7
Nous souhaitons qu'un producteur ne dépose un nouvel élément que si le précédent a été retiré. Nous supposons qu'un élément a été retiré s'il n'est pas dans le stock (ce qui veut dire qu'on considère comme retiré un élément qui n'a jamais été déposé). Nous ajoutons donc une entrée (ou fonction ou procédure, à vous de choisir et de justifier votre choix) au type protégé prenant en paramètre les identifiants de l'élément et du producteur et indiquant par un booléen si l'élément a été retiré ou non. Le booléen doit avoir la valeur `true` si l'élément a été retiré et `false` sinon.

Modifiez la déclaration et le corps de l'objet protégé `Gestionnaire`. Ecrivez les instructions de la tâche `Produc` garantissant qu'un nouveau dépôt d'élément n'a lieu que si le précédent a été retiré.

Question 8
Proposez une solution permettant de résoudre le problème mis en évidence dans le corrigé de la question précédente.

Exercice 11 – Lecteurs-Écrivains

L'objectif de cet exercice est de réaliser une synchronisation de type lecteurs-écrivains en utilisant les attributs `'count` associés aux entrées d'une tâche, et la notion de type protégé.

Le problème des lecteurs-écrivains est un exemple de synchronisation extrêmement courante dans le contexte des systèmes répartis. On considère deux classes de processus - des lecteurs et des écrivains - souhaitant intervenir sur une même zone mémoire partagée. Les lecteurs *lisent* la zone mémoire ; cette action n'entraîne aucune modification de la zone mémoire. Les écrivains au contraire *écrivent* dans la zone mémoire et en modifient le contenu.

Les règles de synchronisation sont les suivantes :
– plusieurs lecteurs peuvent lire simultanément la zone mémoire partagée,
– un écrivain accède à la zone mémoire partagée à l'exclusion de tout autre lecteur ou écrivain,

Cette synchronisation est réalisée par une tâche ordonnanceur qui autorise les lecteurs et les écrivains à accéder à la zone mémoire partagée. Cette tâche possède six entrées : `dem_L, dem_E, Lecture, Ecriture, fin_L, fin_E` représentant respectivement une demande d'accès en lecture à la zone mémoire, une demande d'accès en écriture à la zone mémoire, une lecture, une écriture, une fin de lecture et une fin d'écriture. Ce découpage en trois étapes de chaque action est classique est correspond à ce qu'on trouve avec les sections critiques : demande d'accès à la section critique, accès à la section critique, libération de la section critique.

La zone mémoire partagée sera représentée par une variable entière gérée par l'ordonnanceur. Les écritures devront modifier la valeur de cette variable et les lectures devront récupérer cette valeur. Pour assurer que les données lues aient un sens, la première opération sur la mémoire partagée doit être une écriture.

Pour pouvoir suivre l'évolution du système vous donnerez à vos écrivains et lecteurs un identifiant de type entier naturel que chacun d'entre eux récupèrera en utilisant une entrée.

Question 1
Un lecteur est repésenté par une tâche effectuant de façon répétitive les trois actions suivantes : demande l'accès à la zone mémoire partagée, lecture, relâche l'accès à la zone mémoire partagée. De façon similaire, un écrivain est modélisé par une tâche effectuant de façon répétitive les trois actions suivantes : demande l'accès à la zone mémoire, écriture, relâche l'accès. Ces actions seront effectuées un nombre de fois égal à l'identifiant du lecteur (ou écrivain).

Les demandes, relâchements d'accès et lectures (resp. écritures) sont réalisés par le biais de rendez-vous entre la tâche lecteur (resp. écrivain) et la tâche ordonnanceur.

Ecrivez le code d'une tâche lecteur et d'une tâche écrivain. Pour pouvoir suivre l'évolution du système, lors de chaque demande d'accès à la mémoire (ou de libération de cette dernière) le demandeur devra afficher son identité, le type de la demande, et le numéro de la demande. Lors de la lecture, les lecteurs devront afficher la valeur lue ainsi que leur identifiant. Lors de l'écriture, les écrivains devront afficher la valeur écrite. L'exécution suivante a été obtenue avec 2 lecteurs et 2 écrivains où la valeur écrite est égale à l'identité de l'écrivain multipliée par le "numéro" de l'écriture.

```
ecrivain  1 demande acces  1e ecriture
ecrivain  1 ecriture :  1
ecrivain  1 fin acces  1e ecriture
lecteur  1 demande acces  1e lecture
lecteur  2 demande acces  1e lecture
lecteur  1 lecture :  1
lecteur  2 lecture :  1
lecteur  1 fin acces  1e lecture
lecteur  2 fin acces  1e lecture
ecrivain  2 demande acces  1e ecriture
ecrivain  2 ecriture :  2
ecrivain  2 fin acces  1e ecriture
ecrivain  2 demande acces  2e ecriture
ecrivain  2 ecriture :  4
ecrivain  2 fin acces  2e ecriture
lecteur  2 demande acces  2e lecture
lecteur  2 lecture :  4
lecteur  2 fin acces  2e lecture
```

Question 2

Ecrivez le code de l'ordonnanceur, de telle sorte qu'une demande de lecture est acceptée s'il n'y a pas d'écriture en cours et une demande d'écriture est acceptée s'il n'y a ni lecture en cours ou demandée (vous utiliserez l'attribut `'count`), ni écriture en cours . Cette stratégie donne une priorité aux lecteurs : lorsque des lecteurs sont en train de lire et qu'il y a des écrivains en attente, une nouvelle demande de lecture est autorisée, passant avant les demandes d'écriture en attente.

Pour tester l'exécution de vos tâches vous créerez 5 lecteurs et 3 écrivains. Vous ajouterez des instructions **`delay`** pour mettre en évidence l'exécution en parallèle des différentes tâches.

Question 3

Modifiez l'ordonnanceur pour qu'une lecture ne puisse débuter que s'il n'y a ni écriture en cours, ni demande d'accès en écriture (priorité aux écrivains).

Question 4

Nous souhaitons réaliser la synchronisation lecteurs-écrivains avec priorité aux écrivains à l'aide d'un objet protégé. Expliquez pourquoi transformer la tâche ordonnanceur en un objet protégé n'ayant que deux points de synchronisation : une fonction `lecture` et une procédure `ecriture` ne permet pas de répondre à la question.

Question 5

Ecrivez la version des lecteurs-écrivains avec un objet protégé.

Thème 5

Objectifs

– Respiration : on revient sur les notions présentées dans les semaines précédentes.

Exercices

Le premier exercice est à faire en TD, le deuxième en TME.

Exercice 12 – Gestion d'un pont à voie unique

Nous considérons le système représenté dans la Figure 2. Ce système comporte les composants suivants :
– un **pont** à voie unique, gardé par des **feux**,
– des **voitures** classées en deux catégories (A ou B) selon leur sens de circulation,
– un **contrôleur** qui gère les feux.

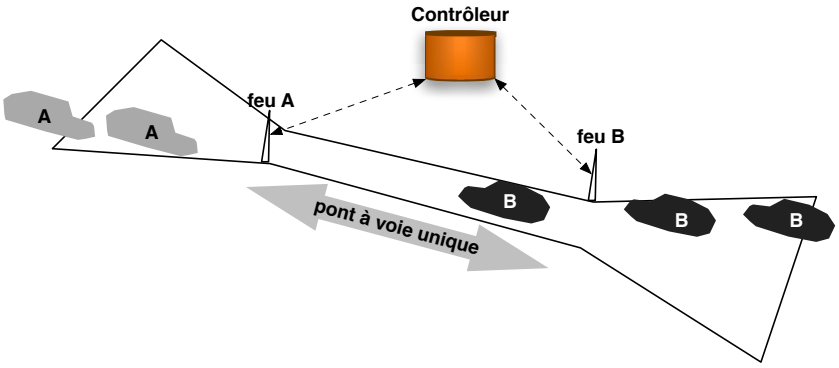


FIGURE 2 – Le pont à voie unique et le système de gestion associé

Nous considèrerons que :
– lorsqu'une voiture arrive devant un feu rouge, elle attend que ce dernier passe au vert ; on considèrera que tout se passe comme si la voiture "demande" au feu le droit de passer.
– la capacité du pont est limitée. De manière similaire à ce qui se passe avec les feux, la voiture "demande" le droit de passer sur le pont et ne peut le faire que si ce dernier n'est pas saturé (on utilisera la constante `Capacite` pour représenter le nombre maximum de voitures simultanément sur le pont).
– le contrôleur n'a aucune interaction directe avec les voitures. Il gère de manière alternative les changements de sens sur le pont sur la base d'une temporisation définie par la constante `Attente_Alt` (en secondes).

Question 1

Quelles vérifications doit effectuer le contrôleur lorsqu'il décide d'inverser le sens de passage sur le pont pour éviter les collisions ?

Question 2

Identifiez les entités qui doivent être des tâches et les entités qui seraient mieux sous la forme de types ou objets protégés.

Considérons que l'on dispose des types et constantes indiquées ci-après :

```
--| Gestion de durees aleatoires...
package Alea_Int is new Ada.Numerics.Discrete_Random (Integer);

Graine : Alea_Int.Generator;

function Duree_Attente return Duration is
begin
   return 0.1 * Duration (Alea_Int.Random (Graine) mod 10);
end Duree_Attente;

--|-------------------------------------------------------\\
--| Types et constantes pour gerer le systeme.

type Couleur is (Vert, Rouge); -- Pas de feu orange ;-)
type T_Vehicule is (Va, Vb);

Capacite    : constant Natural  := 2;
Nb_Voitures : constant Natural  := 5;
Attente_Alt : constant Duration := 0.5;
```

Question 3
Déclarez le `Pont`.

Question 4
Programmez le corps du `Pont`.

Question 5
Déclarez un `Feu`.

Question 6
Programmez le corps d'un `Feu`.

Les voitures sont toutes identiques mais typées par une catégorie (pour savoir si elles sont A ou B). Chaque voiture dispose également d'un identifiant (unique à l'intérieur de sa catégorie) qui devra lui être transmis avant toute autre opération de sa part.

Question 7
Déclarez une `Voiture`.

Question 8
Déclarez le `Controleur`.

On s'appuie sur les variables globales suivantes :

```
--|-------------------------------------------------------
--| Variables

Les_Feux   : array (T_Vehicule) of Feu;
Voitures_A : array (1 .. Nb_Voitures) of Voiture (Va);
Voitures_B : array (1 .. Nb_Voitures) of Voiture (Vb);
```

Le comportement d'une voiture est d'arriver au feu, de passer le feu, d'entrer sur le pont, d'en sortir et de se terminer.

Question 9
Programmez le corps d'une `Voiture`.

Question 10
Programmez le corps du `Controleur`.

Question 11
Écrivez le corps de la tâche principale.

Question 12
Que se passe-t-il si une voiture s'arrête après avoir passé le feu et avant d'entrer sur le pont ? Peut-on mettre cette situation en évidence ?

Question 13
Proposez un mécanisme simple pour résoudre ce problème.

Exercice 13 – Gestion d'un tunnel à une voie
Nous considérons un tunnel ferroviaire à une seule voie. Ce tunnel doit être partagé entre les trains circulant dans les deux sens. Pour éviter tout accident dans le tunnel, le sens de circulation de ce dernier alterne tous les `NB_MAX` passages de trains. Un train ne peut passer dans le tunnel que si son sens de circulation correspond au sens du tunnel.

Les trains sont tous de même type, un discriminant identifie leur sens de circulation. Nous avons alors deux tableaux de `NB_TRAINS` trains. Chaque tableau représente l'ensemble des trains circulant dans le même sens.

Question 1
Une tâche `tunnel` assure la gestion du tunnel. Chaque train demande à la tâche `tunnel` la permission d'entrer dans le tunnel et l'informe de sa sortie du tunnel. La tâche tunnel doit assurer qu'une fois que `NB_MAX` trains sont passés dans le tunnel dans le même sens, le sens de circulation change. Faites attention à ne pas changer le sens de circulation trop tôt.

Complétez le fichier `gest_tunnel.adb` pour répondre à ce problème. Votre programme devra contenir les affichages nécessaires au suivi de l'évolution du système.

Question 2
Faites un deuxième programme dans lequel le tunnel sera représenté par un objet protégé.

Thème 6

Objectifs

– Mécanismes d'exceptions dans une tâche

Exercices

Les trois premiers exercices seront faits en TD et les trois derniers en TME.

Exercice 14 – Division par zéro

Nous allons voir comment utiliser les exceptions pour éviter d'effectuer une division par zéro.

Question 1

Que se passe-t-il si une division par zéro est effectuée par un programme Ada ? Quelle utilité pourrait-il y avoir à lever un exception particulière (`DIV_ZERO` par exemple) dans ce cas ?

Question 2

Pour effectuer la division, nous disposons d'une fonction `division` prenant deux entiers en paramètres et retournant le résultat de la division entière. Cette fonction est appelée par une procédure principale ou toute autre fonction ou procédure. Qui doit lever l'exception et qui doit en assurer le traitement ?

Exercice 15 – Saisie d'un numéro de sécurité sociale

Le but de cet exercice est d'écrire un programme qui permet la saisie d'un numéro de sécurité sociale et qui traite les erreurs de saisie.

Question 1

En considérant le programme suivant, expliquez ce qui se passe si l'utilisateur saisit un nombre :

1. inférieur à 1000000000000 ?
2. supérieur à 2999999999999 ?
3. comportant un caractère n'appartenant pas à l'alphabet des entiers (par exemple une lettre) ?

```
with Ada.Text_Io;
use Ada.Text_Io;

procedure no_sec_sociale_sur_sujet is

  type Un_Numero is range 1000000000000..2999999999999;
  package Un_Numero_IO is new Integer_IO(Un_Numero);

  Le_Numero : Un_numero;

begin
  Put("Quel numero INSEE ? ") ;
  Un_Numero_IO.Get(Le_Numero);

  if Le_Numero/1000000000000 = 1 then
    Put_Line("Bonjour Monsieur");
  else
    Put_Line("Bonjour Madame");
  end if ;
end no_sec_sociale_sur_sujet ;
```

Question 2
Même question que précédemment mais en considérant le programme suivant :

```
with Ada.Text_Io;
use Ada.Text_Io;

procedure no_sec_sociale_sur_sujet2 is

type Un_Numero is range 1000000000000..2999999999999;
package Un_Numero_IO is new Integer_IO(Un_Numero);

task saisie;

task body saisie is
  Le_Numero : Un_numero;
begin
  Put("Quel numero INSEE ? ") ;
  Un_Numero_IO.Get(Le_Numero);

  if Le_Numero/1000000000000 = 1 then
    Put_Line("Bonjour Monsieur");
  else
    Put_Line("Bonjour Madame");
  end if ;
end saisie;

begin
    null;
end no_sec_sociale_sur_sujet2 ;
```

On veut intégrer des contrôles dans ce programme pour éviter une terminaison brutale en cas d'erreur de saisie.

Question 3
Modifiez le code du programme ci-dessus pour qu'en cas d'erreur de saisie, un message d'erreur soit affiché avant la terminaison du programme.

Tout le monde peut faire une faute de frappe... On veut donc modifier le programme pour permettre à l'utilisateur de recommencer la saisie jusqu'à ce que le numéro fourni soit un numéro de sécurité sociale.

Question 4
Modifiez le code du programme ci-dessus en conséquence.

On veut maintenant utiliser ce programme pour implanter un mécanisme d'authentification. L'utilisateur a la possibilité de saisir 3 fois son numéro. S'il n'a pas réussi après 3 tentatives, le programme doit se terminer avec un message lui expliquant que l'authentification a échoué.

Question 5
Modifiez le code du programme ci-dessus en conséquence.

Exercice 16 – Produit d'un ensemble d'entiers
Considérons un ensemble d'entiers, dont il s'agit de calculer le produit. On appliquera une démarche récursive, mais on souhaite interrompre la récursivité (sans remonter dans les appels) dès que le résultat est connu.

Les déclarations de type suivantes permettent de manipuler un ensemble d'entiers :

```
type Tab_Entiers is array(positive range <>) of integer;
type Ensemble_Entiers(Max : Positive) is record
   Taille  : Natural := 0;         -- nb d'elements significatifs
   Contenu : Tab_Entiers(1..Max);
end record;
```

On suppose dans un premier temps que l'ensemble que l'on manipule est non vide.

Le prototype de la fonction qui calcule le produit est le suivant :

```
   -- renvoie le produit des elements de la liste L_Ensemble.
   function Produit(L_Ensemble : Ensemble_Entiers) return Integer is
```

Question 1
Déterminez la relation de récurrence et le cas d'arrêt permettant de faire un produit récursif.

Question 2
Dans quel cas le résultat du calcul est-il connu avant la fin de la fonction ? Quel mécanisme peut-on utiliser pour gérer efficacement ce cas ?

On s'intéresse maintenant au cas où l'ensemble est vide. Dans ce cas, il n'est pas correct que la fonction renvoie une valeur, par contre, on souhaite que l'utilisateur du programme puisse récupérer un message d'erreur.

Question 3
Que se passe-t-il si la fonction se contente d'afficher un message d'erreur, sans renvoyer de valeur ? Que peut-on faire pour éviter ce problème ?

Question 4
Complétez le code Ada suivant pour répondre aux besoins précédemment cités. Pour mettre en évidence le comportement particulier induit par l'utilisation d'une exception dans le cas du calcul d'un produit nul, nous allons calculer le résultat de la fonction `Produit_Aux` dans une variable locale. L'insertion d'une commande d'affichage à un emplacement judicieusement choisi devra permettre de visualiser le comportement particulier induit par l'exception.

Écrivez un programme permettant de traiter les différents cas d'appel de la fonction.

```
function Produit(L_Ensemble : Ensemble_Entiers) return Integer is
  -- a completer

   function Produit_Aux(Tableau : Tab_Entiers) return Integer is
   -- renvoie la valeur du produit des valeurs contenues dans TOUTES les cases du tableau
      res : integer;
   begin
      -- a completer pour calculer la valeur de res
      return res;
   end Produit_Aux ;

begin
  -- a completer
end Produit;
```

Exercice 17 – Division par zéro - TME

Soit le programme suivant :

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_Io, Ada.Integer_Text_IO;

procedure division is

  function division(dividende : integer; diviseur : integer) return integer is
  begin
    return dividende/diviseur;
  end division;

  procedure calcul1 is
  a,b : integer ;
  begin
    put_line("Calcul1");
    put_line("Saisie du dividende");
    get(a);
    put_line("Saisie du diviseur");
    get(b);
    put_line(integer'image(a)&"/"&integer'image(b)&"="&integer'image(division(a,b)));
```

```
    new_line;
  end calcul1;

  procedure calcul2 is
  a,b,c : integer ;
  begin
    put_line("Calcul2");
    put_line("Saisie du dividende 1");
    get(a);
    put_line("Saisie du dividende 2");
    get(b);
    put_line("Saisie du diviseur 2");
    get(c);
    put_line(integer'image(a) & "/(" & integer'image(b) & "/" & integer'image(c)
                              & ") =" & integer'image(division(a,division(b,c))));
    new_line;
  end calcul2;

begin
    calcul1;
    new_line;
    calcul2;
end division ;
```

Question 1
Compilez et exécutez le programme précédent. Vérifiez ce qui se passe lorsque le premier (le deuxième) calcul fait une division par zéro.

Question 2
Modifiez le programme précédent pour que l'exception `DIV_ZERO` soit levée dans le cas d'une division par zéro.

Dans vos réponses aux questions suivantes, les traitements d'exceptions doivent être effectués dans la procédure principale.

Question 3
Modifiez le programme précédent pour que, lors d'une division par zéro, le programme soit interrompu et le message `PB : division par zero` soit affiché.

Question 4
Modifiez le programme précédent pour qu'une division par zéro dans `calcul1` n'empêche pas la réalisation de `calcul2`.

Dans vos réponses aux questions suivantes, les traitements d'exceptions doivent être effectués dans les procédures `calcul1` et `calcul2`.

Question 5
Répondez à la question précédente en traitant les exceptions dans les procédures `calcul1` et `calcul2`.

Question 6
Modifiez le comportement de `calcul2` de façon à ce que `calcul2` redemande des données tant que celles fournies conduisent à une division par zéro.

Exercice 18 – Recherche d'un élément dans un tableau

Soit le programme suivant :

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_Io, Ada.Integer_Text_IO;

procedure recherche is
```

```
   type tableau is array (natural range <>) of integer;

   function trouve(tab : tableau; element : integer) return boolean is
      OK : boolean;
      i : natural;
   begin
      OK := false;
      i := tab'first;
      while (not OK and then i <= tab'last)
      loop
         OK := tab(i) = element;
         i := i+1;
      end loop;
      return OK;
   end trouve;

   Mon_tab : tableau(1..10);
begin
   Mon_tab := (3, -2, 5, 7, -7, 1, -5, 0, 9, 4);
   for elem in -10 .. 10 loop
      if trouve(Mon_tab, elem) then
         put_line("L'element " & natural'image(elem) & " se trouve dans le tableau");
      else
         put_line("L'element " & natural'image(elem) & " ne se trouve pas dans le tableau");
      end if;
   end loop;
end recherche_rec;
```

Question 1

Modifiez la fonction `trouve` pour quelle soit récursive. Votre programme devra permettre de tracer les appels récursifs effectués.

Question 2

Modifiez le programme pour interrompre la récursivité dès que le résultat de la fonction `trouve` est connu. Vous devrez mettre en évidence la différence de comportement entre ce nouveau programme et la solution précédente.

Exercice 19 – Matrices encapsulées sûres

Nous souhaitons rendre sûr le paquetage `Matrice` dont la spécification suit. C'est-à-dire que nous voulons traiter les exceptions levées lors de l'exécution lorsque les deux matrices devant être additionnées ou multipliées n'ont pas des dimensions compatibles.

```
package Matrice is

  --declaration du type matrice
  type Une_Matrice_Entiere is array (Positive range <>, Positive range <>) of Integer;

  -- procedure d'initialisation d'une matrice par saisie des valeurs au clavier
  -- l'initialisation doit se faire ligne par ligne
  procedure Initialiser_Une_Matrice(Matrice : in out Une_Matrice_Entiere);

  -- procedure d'affichage d'une matrice
  procedure Afficher_Une_Matrice(Matrice : in Une_Matrice_Entiere);

  -- fonction d'addition de 2 matrices
  function "+"(Matrice1, Matrice2 : Une_Matrice_Entiere) return Une_Matrice_Entiere;

  -- fonction de multiplication d'une matrice par un scalaire
  function "*"(Matrice  : Une_Matrice_Entiere; Scalaire : Integer) return Une_Matrice_Entiere;

  -- fonction de multiplication d'un scalaire par une matrice
  function "*"(Scalaire : Integer; Matrice  : Une_Matrice_Entiere) return Une_Matrice_Entiere;

  -- fonction de multiplication d'une matrice par une matrice
  function "*"(Matrice1, Matrice2 : Une_Matrice_Entiere) return Une_Matrice_Entiere;
```

```
  -- exception qui doit etre levee si les parametres d'une fonction ne permettent pas l'operation
  -- souhaitee
  Tailles_Non_Compatibles : Exception;

end Matrice;
```

Question 1
Modifiez le fichier `matrice.adb` pour que l'exception `Tailles_Non_Compatibles` soit levée à bon escient.

Question 2
Compilez et exécutez le programme `test_matrice_sure.adb` qui vous est fourni. Modifiez le pour constater la levée des exceptions.

Question 3
Écrivez un programme `test_matrice_sure1.adb` qui, lors de la levée d'une exception, affiche le message `La taille des matrices ne permet pas l'operation souhaitee` **et se termine**.

Question 4
Écrivez un programme `test_matrice_sure2.adb` qui lors de la levée d'une exception affiche le message `La taille des matrice ne permet pas l'addition` ou le message `La taille des matrice ne permet pas la multiplication` **et se termine**. Le choix du message à afficher dépend bien sûr de l'opération qui n'a pas pu s'exécuter.

Question 5
Écrivez un programme `test_matrice_sure3.adb` qui, lors de la levée d'une exception, affiche le message `La taille des matrices ne permet pas l'addition` ou le message `La taille des matrices ne permet pas la multiplication` **et poursuit ensuite son exécution**. Le choix du message à afficher dépend bien sûr de l'opération qui n'a pas pu s'exécuter.

Question 6
Peut-on gérer un problème de dimensions qui survient lors de l'affectation du résultat de l'opération dans une matrice de dimensions incompatibles ?

Thème 7

Objectifs

- pointeurs
- passage de paramètres en mode **out** dans les procédures

Exercices

Le premier exercice sera fait en TD et le second en TME.

Exercice 20 – Pointeurs en Ada

Le but de cet exercice est de gérer de manière **itérative** et **récursive** une liste d'entiers.

Question 1

Définissez les types nécessaires à la déclaration du type `Une_Liste_Entiers` suivant :

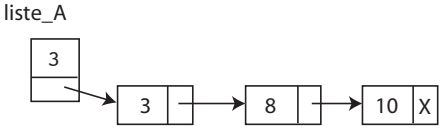
```
type Une_Liste_Entiers is
record
    Tete : Ptr_Element_De_Liste;
    Nb_Elements : Natural := 0;
end record;
```

On souhaite écrire le corps des procédures dont la spécification est donnée ci-dessous :

```
-- Ajoute l'entier Valeur en tete de liste
procedure Ajout_Tete(Liste : in out Une_Liste_Entiers; Valeur : Natural);

-- Ajoute iteratif de l'entier Valeur en fin de liste
procedure Ajout_Fin(Liste : in out Une_Liste_Entiers; Valeur : Natural);

-- Ajout recursif de l'entier Valeur en fin de liste
procedure Ajout_Rec_Fin(Liste : in out Une_Liste_Entiers; Valeur : Natural);

-- Ajoute iteratif de l'entier Valeur a sa place dans une liste deja triee
-- en ordre croissant
procedure Ajout_Trie(Liste : in out Une_Liste_Entiers; Valeur : Natural);

-- Ajout recursif de l'entier Valeur a sa place dans une liste deja triee
procedure Ajout_Rec_Trie(Liste : in out Une_Liste_Entiers; Valeur : Natural);

-- Enleve iterativement la permiere apparition de l'entier Valeur de la liste
procedure Enleve(Liste : in out Une_Liste_Entiers; Valeur : Natural);

 -- Enleve recursivement la permiere apparition de l'entier Valeur de la liste
procedure Enleve_Rec(Liste : in out Une_Liste_Entiers; Valeur : Natural);
```

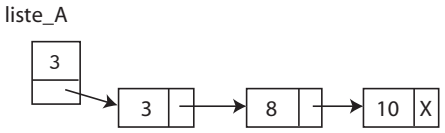
Question 2

Représentez ce qui se passe si on insère l'entier 1 en tête de la liste suivante. Que se passe-t-il si l'entier est ajouté dans une liste vide ?



Question 3
Ecrivez le corps de la procédure `Ajout_Tete`.

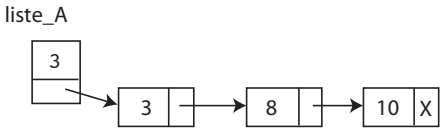
Question 4
Représentez ce qui se passe si on insère l'entier 15 en fin de la liste suivante. Que se passe-t-il si l'entier est ajouté dans une liste vide ?



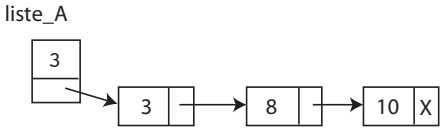
Question 5
Ecrivez le corps de la procédure itérative `Ajout_Fin`.

Question 6
Ecrivez le corps de la procédure récursive `Ajout_Rec_Fin`.

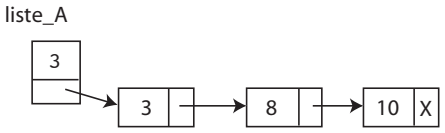
Question 7
Représentez ce qui se passe si on insère l'entier 1 dans la liste suivante triée en ordre croissant.



Question 8
Représentez ce qui se passe si on insère l'entier 15 dans la liste suivante triée en ordre croissant.



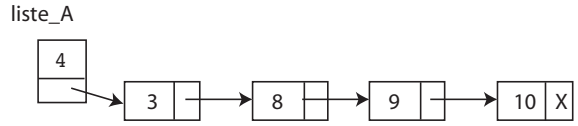
Question 9
Représentez ce qui se passe si on insère l'entier 9 dans la liste suivante triée en ordre croissant.



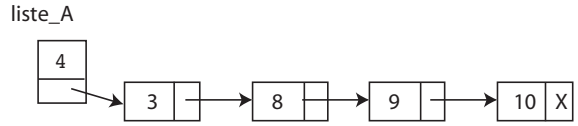
Question 10
Ecrivez le corps de la procédure itérative `Ajout_Trie`.

Question 11
Ecrivez le corps de la procédure récursive `Ajout_Rec_Trie`.

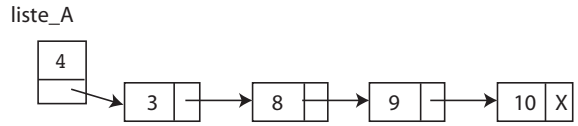
Question 12
Représentez ce qui se passe si on enlève l'entier 8 dans la liste suivante.



Question 13
Représentez ce qui se passe si on enlève l'entier 10 dans la liste suivante.



Question 14
Représentez ce qui se passe si on enlève l'entier 3 dans la liste suivante.



Question 15
Lequel des cas précédents permet de traiter la suppression d'un élément dans une liste ne contenant qu'un seul élément ?

Question 16
Ecrivez le corps de la procédure itérative `Enleve`. Cette procédure doit désallouer l'espace mémoire occupé par l'élément supprimé.

Question 17
Ecrivez le corps de la procédure récursive `Enleve_Rec`. Cette procédure doit désallouer l'espace mémoire occupé par l'élément supprimé.

Exercice 21 – Gestion de matrices creuses

Les *matrices creuses* sont des matrices dans lesquelles un grand nombre d'éléments prend la valeur zéro. Une optimisation classique pour le stockage de telles matrices consiste à ne pas représenter les éléments nuls. Les tableaux à deux dimensions ne sont donc pas adaptés, il faut une structure dynamique qui s'adapte aux valeurs contenues dans le tableau.

Nous avons choisi de représenter chaque ligne de la matrice par une liste dont chaque cellule contient comme informations : la valeur de l'élément (non nulle), la colonne à laquelle se trouve l'élément et l'adresse de l'élément non nul suivant sur la même ligne. L'ensemble des éléments de la matrice est représenté par un tableau dont chaque élément est un pointeur vers le premier élément non nul de la ligne. Pour pouvoir gérer des matrices de différentes tailles, nous représentons la matrice par un enregistrement à discriminants (le nombre de lignes et le nombre de colonnes). Voici les déclarations des types associés à cette représentation.

```
-- type associe a un element de la matrice :
-- Prochain est l'adresse du prochain eleement sur la meme ligne,
-- Colonne est la colonne a laquelle se trouve l'element,
-- Donnee est la valeur de l'element.
type Un_Element_De_Liste is
record
  Prochain : Ptr_Element_De_Liste;
  Colonne  : Positive;
```

```
  Donnee   : Integer;
end record;

-- tableau permettant de stocker l'adresse du premier element de chaque ligne
type Tab_Ptr is array (Positive range <>) of Ptr_Element_De_Liste;

-- type Une_Matrice_entiere avec deux discriminants : le nombre de lignes (qui determine la taille du
-- tableau) et le nombre de colonnes (necessaire pour connaitre la taille de la matrice et savoir ou se
-- trouvent les 0)
type Une_Matrice_Entiere(Lignes, Colonnes : Positive) is
record
  Tete : Tab_Ptr(1..Lignes);
end record;
```

L'accès à la matrice qui se faisait jusqu'à présent grâce à `Matrice(i, j)` doit maintenant se faire par un accès à notre structure de données.

On peut remarquer dès à présent que l'accès à un élément de la liste ne peut se faire qu'en partant de la tête de la liste indexée par la ligne et en passant d'élément en élément, en vérifiant pour chacun d'entre eux s'il est à la colonne demandée. Si on a de la chance, on recherche le premier élément de la liste, mais si l'on n'a pas de chance, on recherche le dernier.

Pour changer la valeur d'une donnée d'une case, il faut parcourir la liste jusqu'à ce que la colonne de l'élément parcouru soit supérieure ou égale à celle de l'élément demandé. En moyenne, on parcourera $\frac{n}{2}$ éléments sur les n contenus dans la liste. Comme il faut de toute façon effectuer ce parcours, on peut en profiter pour insérer l'élément trié dans la liste (c'est le principe du tri par insertion ou *insertion sort*).

Les questions suivantes doivent vous guider pour compléter le fichier `matrice.adb`.

Question 1

Ecrivez la fonction `Valeur` qui retourne la valeur d'une case de la matrice en utilisant maintenant la structure précédente.

Question 2

Ecrivez la procédure `Valeur` qui positionne la valeur d'une case de la matrice. On rappelle que lorsque la valeur est zéro, il n'y a pas d'élément inséré dans la liste (positionner une valeur peut amener à supprimer un élément de la liste).

Question 3

Testez vos fonction et procédure `Valeur` avec les programmes `test_saisie.adb`, `test_fct_valeur.adb`, `test_ajout_0.adb` et `test_ajout_1.adb`. Pour comparer votre exécution à notre exécution de référence, vous devez rediriger l'entrée de votre programme de la façon suivante : `./test_XXXX < donnees_test_XXXX`. Le fichier `resultats_tests` contient les résultats que vous devez obtenir.

Question 4

Réecrivez les procédures d'addition et de multiplication des matrices en utilisant les fonctions et procédures `valeur`.

Question 5

Testez vos opérations sur les matrices avec les programmes `test_addition.adb`, `test_scalaire.adb`, `test_multiplication.adb`. Pour comparer votre exécution à notre exécution de référence, vous devez rediriger l'entrée de votre programme de la façon suivante : `./test_XXXX < donnees_test_XXXX`. Le fichier `resultats_tests` contient les résultats que vous devez obtenir.

Thème 8

Objectifs

– création dynamique de tâches ;

Exercices

Le premier exercice sera fait en TD et les deux suivants en TME. Le dernier exercice est facultatif.

Exercice 22 – Serveur multi-tâches

Dans cet exercice, nous nous intéressons à un système client/serveur très simplifié : les clients émettent vers un serveur des requêtes identifiées par un numéro de client émetteur et un type de requête, lui aussi représenté par un numéro. Le traitement d'une requête est matérialisé par un affichage et ne donne pas lieu à l'envoi de résultats par le serveur au client.

Question 1

Le serveur ne connaît pas a priori le nombre de requêtes qu'il aura à traiter. Quelle est la structure adaptée pour le corps de la tâche serveur ?

Question 2

Programmez un système avec deux clients identiques, identifiés par un numéro (respectivement 1 et 2), et envoyant chacun au serveur cinq requêtes de type 1. Le traitement de ces requêtes est un simple affichage. Le corps d'une tâche client est donné ci-dessous :

```
task body client is
begin
    for i in 1..5 loop
        serveur.requete(id, 1);
        delay 0.5;
    end loop;
end client;
```

On ajoute au système une nouvelle tâche client d'identificateur 3 qui envoie une unique requête de type 2. Le traitement associé à cette requête est une boucle infinie. Le nouveau corps d'une tâche client est donné ci-dessous :

```
task body client is
begin
    if id <= 2 then
        for i in 1..5 loop
            serveur.requete(id, 1);
            delay 0.5;
        end loop;
    else
        serveur.requete(id, 2);
    end if;
end client;
```

Question 3

Modifiez en conséquence le code de la tâche serveur. Que se passe-t-il lorsqu'on exécute ce système de tâches ?

On veut maintenant donner la possibilité au serveur de lancer des traitements de requêtes en parallèle. Ainsi, la requête bloquante d'un client ne devrait plus bloquer les autres clients. Donc, lorsque le serveur reçoit une requête, il

n'effectue pas le traitement associé à cette requête, mais crée une tâche à laquelle il transmet l'identifiant du client et le type de requête à traiter.

Question 4
Donnez la déclaration des nouveaux types nécessaires à la mise en place de cette stratégie.

Une première solution pour que le serveur puisse créer des tâches de manière répétitive est de déclarer une variable de type tâche dans un bloc interne à la boucle du serveur.

Question 5
Écrivez le nouveau corps de la tâche serveur, et le corps des tâches appelées.

Question 6
Cette solution résout-elle le problème posé par les requêtes bloquantes ?

On se propose maintenant d'utiliser des tâches dynamiques pour traiter les requêtes.

Question 7
Donnez la déclaration du type permettant la création dynamique de tâches de traitement.

Question 8
Écrivez le nouveau corps de la tâche serveur. Le problème précédent est-il résolu ?

Question 9
Le serveur utilise ici un unique pointeur qui est mis à jour au fur et à mesure de la création de nouvelles tâches. Quelles conséquences cela a-t-il sur l'exécution du système de tâches ?

Question 10
Combien y a-t-il de tâches créées au total ? Combien d'entre elles se terminent-elles ?

Exercice 23 – Crible d'Eratosthène avec création dynamique de tâches

Le crible d'Eratosthène est un algorithme permettant de construire la suite des n premiers nombres premiers.
Il est bâti sur la propriété suivante :
$(\forall q < p$ et q premier, q ne divise pas $p) \implies p$ est premier.
Lorsqu'il traite le nombre p, l'algorithme a déjà construit la liste `Liste_nb_premiers` des nombres premiers inférieurs à p, et il teste pour chacun des nombres q de cette liste si q divise p. S'il existe au moins un q qui divise p, alors p n'est pas premier. Sinon, p est premier et est ajouté à la liste `Liste_nb_premiers`.
Vous devez récupérer les fichiers :
- `erato_statique.adb`,
- `erato.adb`

Question 1
Compilez le fichier `erato_statique.ad` et exécutez le programme obtenu.

On souhaite réaliser maintenant une implantation dynamique de cet algorithme, afin de paralléliser le filtrage des nombres dont on cherche s'ils sont premiers.

Pour ce faire, la liste des nombres premiers est remplacée par une liste de tâches. Les tâches possèdent un discriminant dont la valeur est un nombre premier. Le rôle d'une tâche est de filtrer par son discriminant toute valeur qui lui est transmise. Chaque tâche possède un pointeur sur la tâche dont le discriminant est le nombre premier directement supérieur à son propre discriminant, que l'on appellera tâche suivante.

La tâche de discriminant q qui reçoit le nombre p détermine si p est divisible par q. Si c'est le cas, p n'est pas premier. Sinon, p n'est pas divisible par q et p est transmis à la tâche suivante. Si une telle tâche n'existe pas (car p a franchi tous les filtres), alors p est premier, et une nouvelle tâche d'identité p est ajoutée à la liste des tâches).

Vous devez compléter le fichier `erato.adb` qui vous est fourni.
Le principe de fonctionnement de ce programme est le suivant :
- initialement la tâche `filtre(2)` est créée,
- lorsque `filtre(2)` reçoit le nombre `3` à filtrer, `filtre(2)` laisse passer `3` (non divisible par `2`) et doit transmettre `3` au filtre suivant. Mais `filtre(2)` est le dernier de la liste, donc c'est à lui de créer `filtre(3)` (et de stocker la réference à cette nouvelle tâche). Il y a donc une nouvelle tâche créée après l'exécution de `filtre(2)` sur `3`,
- lorsque `filtre(2)` doit filtrer `4`, il ne propage pas `4` à son successeur (car `4` est filtré par `filtre(2)`, il est divisible par `2`).
- lorsque `filtre(2)` doit filtrer `5`, il le laisse passer (non divisible par `2`) et doit le transmettre à son successeur `filtre(3)`, qui existe maintenant, puisqu'il a été créé lors du filtrage de `3` par `filtre(2)`. Donc `filtre(3)` doit maintenant filtrer `5` (et pendant ce temps `filtre(2)` peut travailler sur le filtrage de `6`...). `5` passe `filtre(3)`, donc `filtre(3)` doit le transmettre à son successeur ; il n'y en a pas, donc `5` est premier, et `filtre(3)` crée une nouvelle tâche `filtre(5)` (et ainsi de suite ...).

Question 2
Attention, le mécanisme de création des tâches en Ada ne permet pas à une tâche de créer directement une tâche de même type qu'elle. Dans notre exemple, une tâche de type `Filtre` ne peut donc pas créer directement un nouveau filtre. Elle peut par contre appeler une fonction qui va effectuer pour elle cette opération.

Complétez la fonction `Cree_Nouveau_Filtre` qui crée une nouvelle tâche de type `Filtre` en attribuant à son discriminant la valeur entière passée en paramètre. La valeur de retour de cette fonction est une référence sur cette nouvelle tâche.

Question 3
Ecrivez le corps de la tâche `Filtre`. N'oubliez pas que tout nouveau filtre doit afficher son discriminant pour obtenir l'affichage des nombres premiers.

Exercice 24 – Arbre binaire
Le but de cet exercice est de construire un arbre binaire de recherche : chaque nœud de l'arbre est une tâche `noeud` qui possède un pointeur sur son sous-arbre gauche et un pointeur sur son sous-arbre droit.

L'arbre binaire a les caractéristiques suivantes :
- chaque nœud de l'arbre est créé avec un discriminant dont la valeur est un entier naturel,
- les nœuds qui appartiennent au sous-arbre gauche d'un nœud `n` possèdent un discriminant inférieur strictement à celui du nœud `n`,
- les nœuds qui appartiennent au sous-arbre droit d'un nœud `n` possèdent un discriminant supérieur strictement à celui du nœud `n`.

Tous les nœuds de l'arbre ont donc des discriminants de valeurs différentes.

Chaque nœud de l'arbre est une tâche `noeud` qui a les caractéristiques suivantes :
- elle possède un pointeur sur son sous-arbre gauche et un autre sur son sous-arbre droit,
- elle possède une variable qui comptabilise le nombre de requêtes dont la valeur correspond à celle de son discriminant,
- elle peut être appelé sur les entrées `recoit` et `affiche`,
- lorsque la tâche accepte une entrée `recoit`,
  - elle met à jour son compteur si le paramètre de la requête correspond à la valeur de son discriminant,
  - si la valeur est inférieure à celle de son discriminant, elle la transmet à son sous-arbre gauche si celui-ci existe, sinon elle crée celui-ci avec un discriminant dont la valeur est celle du paramètre,

  - si la valeur est supérieure à celle de son discriminant, elle la transmet à son sous-arbre droit si celui-ci existe, sinon elle crée celui-ci avec un discriminant dont la valeur est celle du paramètre.
- lorsque la tâche accepte une entrée `affiche`, elle doit afficher son discriminant et son compteur, demander l'affichage de ses sous-arbres droit et gauche. Ces trois actions doivent être faites dans un ordre qui permet l'affichage en ordre croissant des discriminants.
- la tâche doit se terminer dès qu'elle n'a (ou n'aura) plus rien à faire.

La tâche principale doit implanter l'algorithme suivant :
- créer le sommet de l'arbre en tirant aléatoirement la valeur du discriminant dans l'intervalle `0..NB_VALEURS-1`.
- tirer aléatoirement une suite de valeurs et transmettre chacune d'elles au sommet de l'arbre pour que l'arbre soit mis à jour,
- une fois l'arbre créé, demander son affichage.

Question 1

Complétez le fichier `arbreBinaire.adb` (donné ci-dessous) pour réaliser le programme demandé.

```
with Ada.Text_IO, Ada.Numerics.Discrete_Random;
use Ada.Text_IO;

procedure arbreBinaire is

package Alea_Int is new Ada.Numerics.Discrete_Random (natural);
use Alea_Int;

Graine : Generator;

-- nombre de valeur aleatoires a tirer
NBVALEURS : constant positive := 20;

task type noeud(valeur : natural) is
  entry recoit(val : in natural);
  entry affiche;
end noeud;

type acc_noeud is access noeud;

function nouveau_noeud(val : in natural) return acc_noeud is
-- fonction de creation d'une nouvelle tache
-- une tache ne peut pas directement creer une tache de meme
-- type, mais elle peut appeler une fonction qui le fait
begin
  -- A COMPLETER
end nouveau_noeud;

task body noeud is
 filsGauche, filsDroit : acc_noeud := NULL;

 -- A COMPLETER
end noeud;

-- Declaration des variables de la tache principale

begin
  Alea_Int.Reset(Graine);

  -- A COMPLETER
end arbreBinaire;
```

Exercice 25 – La courbe de Von Koch

La courbe de Von Koch est une fractale dont la construction repose sur le principe récursif suivant : partant d'un segment limité par deux points M et N, on divise ce segment en trois parties égales. Soient A et C les points créés par cette division, on construit entre ces deux points un point B tel que A, B, C est un triangle équilatéral. Puis on répète l'opération sur chacun des segments ainsi créés. Le principe est illustré par la Figure 3.

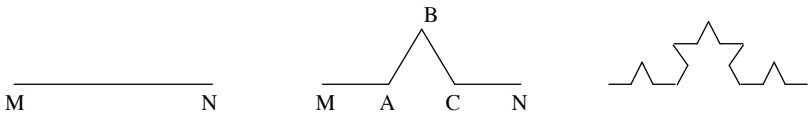


FIGURE 3 – Découpage d'un segment

La condition d'arrêt de la récursivité est liée à la taille des segments créés : lorsque la longueur de ceux-ci devient inférieure à une taille déterminée, on arrête le découpage. Le programme suivant propose une implémentation de ce tracé. La figure initiale est un triangle, et chaque côté du triangle subit une transformation.

```
with graphique, Ada.Numerics.Generic_Elementary_Functions;
use graphique;

procedure Von_Koch_Seq is

package Math_FLoat is new Ada.Numerics.Generic_Elementary_Functions(Float);
use Math_FLoat;

procedure tracer_segment(xm, ym, xn, yn : natural) is
    xa, xb, xc, ya, yb, yc, longueur : float;

begin
    longueur := sqrt(float (( (ym -yn) * (ym -yn) + (xm -xn) * (xm -xn) )));
    if (longueur > 8.0) then
        xa := float(2 * xm + xn) / 3.0;
        xc := float(xm + 2 * xn) / 3.0;
        ya := float(2 * ym + yn) / 3.0;
        yc := float(ym + 2 * yn) / 3.0;

        xb := xa + (xc - xa - (sqrt(3.0) * float(yc - ya))) / 2.0;
        yb := ya + (yc - ya + (sqrt(3.0) * float(xc - xa))) / 2.0;

        tracer_segment(natural(xm), natural(ym), natural(xa), natural(ya));
        tracer_segment(natural(xa), natural(ya), natural(xb), natural(yb));
        tracer_segment(natural(xb), natural(yb), natural(xc), natural(yc));
        tracer_segment(natural(xc), natural(yc), natural(xn), natural(yn));

    else
        Afficher_Ligne(xm, ym, xn, yn, "black");
    end if;
end tracer_segment;

    XU, YU, XV, YV, XW, YW : natural;
    XMAX : constant natural := 400;
    YMAX : constant natural := 400;


begin

    creer_fenetre(XMAX, YMAX, "von Koch");
    remplir_fenetre("white");

    YV := YMAX/8;
    XV := XMAX/2;
    YU := YV + natural(0.5196 * float(XMAX)) ;
    XU := XV - natural(3.0 * float(XMAX)/10.0);
    YW := YU;
    XW := XV + natural(3.0 * float(XMAX)/10.0);

    tracer_segment(XV, YV, XU, YU);
    tracer_segment(XW, YW, XV, YV);
    tracer_segment(XU, YU, XW, YW);

    Attendre_Fermeture_Fenetre;

end Von_Koch_Seq;
```

Question 1

Compilez le programme précédent et exécutez le.

Le programme proposé est récursif et nous voulons en construire une version multi-tâches où chaque appel récursif engendrera la création d'une nouvelle tâche. Ainsi, la récursivité ne s'exprimera plus en termes d'appels de sous-programmes mais en terme de créations dynamiques de tâches.

Attention, vous êtes limité par le nombre de tâches pouvant s'exécuter en parallèle. Si vous souhaitez diminuer la taille déterminant l'arrêt de la création de nouvelles tâches, vous risquez de ne pas obtenir le résultat attendu.

Question 2

Écrivez le source de la procédure `Von_Koch_Dyn` en suivant le principe énoncé ci-dessus.

Thème 9

Objectifs

- Propagation d'erreurs entre tâches
- Notions de flux d'exécution entre tâches

Exercices

Les deux premiers exercices seront faits en TD et les deux derniers en TME.

Exercice 26 – Propagation d'exceptions entre tâches

Soit le programme suivant dans lequel une tâche `calculateur` effectue les calculs demandés par des tâches `client`.

```
with Ada.Text_IO, Ada.Numerics.Discrete_Random, Ada.Exceptions;
use Ada.Text_IO, Ada.Exceptions;

procedure calcul is

   task calculateur is
      entry addition(a:integer; b: integer; res : out integer);
      entry division(a:integer; b: integer; res : out integer);
   end calculateur;

   task body calculateur is
   begin
      loop
         select
            accept addition(a:integer; b: integer; res : out integer) do
               res:=a+b;
            end addition;
         or
            accept division(a:integer; b: integer; res : out integer) do
               res:=a/b;
            end division;
         or
            terminate;
         end select;
      end loop;
   exception
      when E : others => Put_line("Dans calculateur " & Exception_Name(E)
                                                      & " " & Exception_Message(E));
   end calculateur;

   task type client is
      entry init(i : natural);
   end client;

   task body client is
      package Random_Natural is new Ada.Numerics.Discrete_Random (Natural);
      use Random_Natural;

      G : Generator;
      identite : natural;
      op : natural;
      opA, opB, resultat : integer;
   begin
      accept init(i:natural) do
         identite := i;
      end init;
```

```
      put_line("Debut client " & Natural'image(identite));
      Reset(G);

      for i in 1..(random(g)*identite) mod 10 loop
         op  := (random(g)*identite) mod 2;
         opA := (random(g)*identite) mod 10;
         opB := (random(g)*identite) mod 10;
         case op is
            when  0 => calculateur.addition(opA,opB,resultat);
                       put_line("Client" & Natural'Image(identite) &":" & Integer'Image(opA)
                                 & " +" & Integer'Image(opB) & " = " & Integer'Image(resultat));
            when  1 => calculateur.division(opA,opB,resultat);
                       put_line("Client" & Natural'Image(identite) &":" & Integer'Image(opA)
                                 & " /" & Integer'Image(opB) & " = " & Integer'Image(resultat));
            when others => put_line("Ne devrait jamais arriver !");
         end case;
      end loop;
      put_line("Fin client " & Natural'image(identite));
   exception
      when E : others => Put_line("Client " & Natural'Image(identite) & " " & Exception_Name(E)
                                                                      & " " & Exception_Message(E));
   end client;

   tableau_client : array(1..5) of client;

begin
   for i in 1..5 loop
      tableau_client(i).init(i);
   end loop;
end calcul;
```

Question 1

Combien y a-t-il de tâches créées lors de l'exécution du programme `calcul` ? Pour chacune d'elles, donnez ses conditions de terminaison lors d'un fonctionnement "normal".

Question 2

Que se passe-t-il si un `client` demande une division par zéro ?

Question 3

Modifiez le programme pour que lors d'une division par zéro le `client` concerné se termine en affichant un message et le `calculateur` continue son exécution pour répondre aux requêtes des autres `clients`.

Exercice 27 – Détection de la fin d'une tâche

Soit le programme suivant dans lequel une tâche `grand_calcul` distribue des sous-calculs à plusieurs tâches `calcul_local` et attend de recevoir le résultat de chacune d'elles pour calculer le résultat global et se terminer.

```
with Ada.Text_IO, Ada.Numerics.Discrete_Random, Ada.Exceptions;
use Ada.Text_IO;

procedure calcul is
   type calcul_local;
   type acc_calcul is access calcul_local;

   task type calcul_local(opA: integer; opB:integer);

   task grand_calcul is
      entry resultat(res : in integer);
   end grand_calcul;

   task body calcul_local is
      res : integer;
   begin
      put("Parametres :" & integer'Image(opA) & ", "  & integer'Image(opB));
      res := opA / opB;
```

```
      put_line(". Resultat partiel : " & integer'Image(res));
      grand_calcul.resultat(res);
   end calcul_local;

   task body grand_calcul is
      package Random_Natural is new Ada.Numerics.Discrete_Random (Natural);
      use Random_Natural;

      G : Generator;
      resultat_total : integer := 0;
      ptr : acc_calcul;
      nb_taches : Natural;

   begin
      Reset(G);
      nb_taches := random(g) mod 20 + 1;
      put_Line("nb taches " & natural'image(nb_taches));
      for i in 1 .. nb_taches loop
         ptr := new calcul_local(random(g) mod 10, random(g) mod 10);
      end loop;

      for i in 1 ..  nb_taches loop
         accept resultat(res : in integer) do
            resultat_total := resultat_total + res;
         end resultat;
      end loop;
      put_line("Resultat total : " & Integer'Image(resultat_total));
   end grand_calcul;

begin
   null;
end calcul;
```

Question 1

Que se passe-t-il si une tâche `calcul_local` effectue une division par zéro ?

Question 2

Modifiez le programme pour que la tâche `grand_calcul` se termine dans tous les cas. Elle ne doit afficher le résultat total que si elle a récupéré les résultats de *tous* les calculs intermédiaires.

Question 3

Plutôt que d'attendre la fin de tous les calculs intermédiaires, on veut maintenant que la tâche `grand_calcul` puisse se terminer immédiatement avec un message d'erreur dès qu'elle sait que l'un des calculs intermédiaires n'a pas pu s'exécuter correctement.

Modifiez le programme en introduisant un mécanisme d'exceptions pour que les affichages effectués par chacune des tâches permettent d'identifier leur condition de terminaison :

- pour la tâche `grand_calcul` : affichage du résultat global ou message indiquant une terminaison prématurée suite à une erreur dans un calcul intermédiaire ;
- pour les calculs locaux : affichage du résultat si le calcul s'est déroulé normalement ou message d'erreur signalant une division par zéro. Si la communication avec la tâche `grand_calcul` n'a pas pu s'effectuer, un message doit l'indiquer.

Voici un exemple de trace d'exécution : 5 tâches `calcul_local` ont été créées. 3 ont pu calculer leur résultat et 2 ont effectué une division par zéro, ce qui a provoqué la fin prématurée de la tâche `grand_calcul`. L'un des calculs locaux a tenté de contacter `grand_calcul` après sa terminaison.

```
nb taches  5
Resultat partiel :  3
Erreur : division par zero
Resultat partiel :  1
Erreur : division par zero
Resultat partiel :  6
grand_calcul : detection de pb dans un calcul intermediaire
```

```
Echec de la communication avec la tache grand_calcul
```

Exercice 28 – Composants et capteurs

Nous nous intéressons à des composants qui, avant d'exécuter leur traitement, doivent effectuer une phase de vérification. Cette dernière consiste à interroger les 3 capteurs associés à chaque composant. Chaque capteur retourne la valeur 0 ou 1. Si les trois capteurs d'un composant ont retourné la valeur 1, les conditions d'exécution du composant sont optimales. Si deux des trois capteurs ont retourné la valeur 1, les conditions d'exécution du composant sont bonnes. Si seulement un des trois capteurs a retourné la valeur 1, les conditions d'exécution du composant sont acceptables. Sinon, les conditions d'exécution du composant sont risquées.

Nous représenterons la réponse du capteur par un tirage aléatoire.

Question 1

Complétez le fichier `composants.adb` de façon à ce que chaque composant affiche les conditions dans lesquelles il va s'exécuter.

Question 2

Pour simuler les pannes des capteurs, nous allons ajouter à une tâche `capteur` la possibilité de se terminer avant de répondre aux demandes des composants. Un capteur sera aléatoirement choisi comme étant en panne ou non. Modifiez votre programme pour prendre en compte cette possibilité. Que constatez-vous lors de l'exécution ? Toutes les tâches se terminent-elles normalement ?

Question 3

Modifiez votre programme pour que, si un capteur ne répond pas, le composant l'ayant appelé se termine après avoir affiché un message signalant qu'il y a un problème avec un capteur.

Question 4

Écrivez un nouveau programme dans lequel l'absence de réponse d'un capteur est considérée par le composant comme le fait de récupérer la valeur 0.

Exercice 29 – Choix aléatoire d'un serveur

Nous considérons un ensemble de clients et de serveurs tels que chaque serveur peut répondre à un nombre de requêtes égal au nombre de clients et chaque client envoie un nombre de requêtes égal au nombre de serveurs. Dans ce cas, il doit y avoir autant de requêtes traitées que de requêtes envoyées. Chaque client choisit aléatoirement le serveur auquel il s'adresse. Pour que ce soit possible, la variable de type tableau regroupant l'ensemble des serveurs est déclarée avant le corps de la tâche client.

Question 1

Complétez le fichier `choix_serveur.adb`. Pour pouvoir suivre l'évolution de vos tâches vous ajouterez des instructions d'affichage. Vous afficherez en particulier le nom des exceptions qui pourraient se produire.

Question 2

Vous avez dû constater que votre solution n'est pas satisfaisante. Modifiez votre programme pour qu'un client s'adresse à un nouveau serveur lorsque qu'il constate que celui qu'il souhaite contacter ne peut plus répondre à sa requête. Le choix du nouveau serveur à contacter se fait aussi aléatoirement (il est donc possible que le hasard choisisse le même serveur deux fois de suite, ce n'est pas un problème). Votre solution devra utiliser l'attribut de tâche `callable`.

Question 3

Vous avez dû constater que votre solution n'est toujours pas satisfaisante. Modifiez votre programme en utilisant cette fois les exceptions à la place des attributs.

Thème 10

Objectifs

- tâches dynamiques
- anneaux de tâches

Exercices

Les deux premiers exercices seront faits en TD et les deux suivants en TME.

Exercice 30 – Accès aux informations locales d'une tâche

Soit le programme suivant :

```
with Ada.Text_IO, Ada.Numerics.Discrete_Random;
use Ada.Text_IO;

procedure acces is

package Alea_Int is new Ada.Numerics.Discrete_Random (natural);
use Alea_Int;

Graine : Generator;

NBSITES : constant positive := 7;

task type tache(Identifiant : natural) is
   entry fin;
end tache;

type acc_tache is access tache;

type liste_taches;

type acc_liste_taches is access liste_taches;

type liste_taches is record
   la_tache : acc_tache;
   suivant : acc_liste_taches;
end record;


task body tache is
   valeur_locale : natural;
begin
   valeur_locale := random(Graine) mod 100;
   put_line("Tache creee. Identifiant =" & natural'image(identifiant) &
            " - valeur locale =" & natural'image(valeur_locale));
   accept fin;
end tache;
-----------------------------------------------------------------
les_taches : acc_liste_taches;
ptr : acc_liste_taches;

begin
   Alea_Int.Reset(Graine);
   les_taches := new liste_taches;
    ptr := les_taches;
    for i in 1 .. NBSITES-1 loop
      ptr.all.la_tache := new tache(random(Graine) mod NBSITES);
      ptr.all.suivant := new liste_taches;
      ptr := ptr.all.suivant;
```

```
    end loop;
    ptr.all.la_tache := new tache(random(Graine) mod NBSITES);
    ptr.all.suivant := NULL;

   put_line("Fin des taches");
   ptr := les_taches;
    while ptr /= NULL loop
      ptr.all.la_tache.all.fin;
      ptr := ptr.all.suivant;
    end loop;

end acces;
```

Question 1
Modifiez le programme pour que la tâche principale affiche l'identifiant de toutes les autres tâches.

Question 2
Modifiez le programme pour que la tâche principale affiche la valeur locale de chacune des autres tâches.

Exercice 31 – Calcul du minimum sur un anneau

Le calcul de minimum est un problème classique en algorithmique répartie qui permet de tester différents schémas de communication entre tâches. Dans cet exercice, nous nous proposons de créer un ensemble de tâches organisées en anneau, chacune possédant une variable stockant un minimum local. Ces tâches devront ensuite communiquer pour que l'une d'elles (au moins) puisse apprendre la valeur du minimum global sur l'anneau.

Nous nous intéressons dans un premier temps uniquement à la construction de l'anneau. Toutes les tâches exécutent le même code, mais à chaque tâche de l'anneau est associé un entier naturel qui représente la valeur initiale qu'elle va donner à son minimum local.

Question 1
Nous considérons des tâches avec un discriminant qui sera utilisé pour identifier l'entier naturel associé à chaque tâche. Donnez la déclaration des tâches correspondant à cette spécification.

Question 2
Chaque tâche gère une variable locale `suivant` par l'intermédiaire de laquelle elle pourra accéder aux informations (identité, points d'entrée, ...) sur son successeur. Quel est le type de la variable `suivant` ?

Question 3
Pour connaître l'identité de son successeur, une tâche de l'anneau doit invoquer le point d'entrée `Init` d'une tâche `Creer_Anneau` extérieure à l'anneau. Cette tâche n'a pas d'autre entrée. Donnez la spécification de la tâche `Creer_Anneau`.

Question 4
On limite pour l'instant le rôle d'une tâche de l'anneau à l'affichage de sa valeur et de celle de son successeur. Écrivez le corps d'une tâche de l'anneau.

Question 5
La tâche `Creer_Anneau` crée les tâches de l'anneau selon le principe suivant : chaque fois qu'une tâche invoque le point d'entrée `Init`, `Creer_Anneau` crée dynamiquement une nouvelle tâche qui sera affectée comme successeur à la tâche appelante. Lorsque le nombre de tâches créées correspond au nombre de processus coopérant dans l'application de calcul du minimum, l'invocation d'`Init` affectera comme successeur à la tâche appelante la première tâche créée. Le discriminant affecté à chaque tâche sera généré aléatoirement.

`Creer_Anneau` initialise la construction en créant une première tâche. Soit `NBSITE` le nombre de tâches participant au calcul du minimum. Donnez le code de la tâche `Creer_Anneau`.

Question 6
Nous voulons maintenant faire coopérer les tâches pour que l'une d'elles apprenne le minimum des valeurs locales détenues par chacune. Le principe est le suivant : une tâche initiatrice lance le calcul en transmettant sa valeur locale à son successeur. On choisit d'effectuer le traitement de la réception d'une valeur de la manière suivante :
- si la tâche invoquée est l'initiatrice, elle affiche la valeur reçue qui est le minimum des valeurs locales sur l'anneau,
- sinon, elle compare la valeur reçue à son minimum et transmet la plus petite des deux à son successeur.

La tâche initiatrice est choisie par la tâche `Creer_Anneau`. Donnez une nouvelle spécification du type tâche permettant de réaliser cette communication.

Question 7
Modifiez le code de la tâche `Creer_Anneau` pour qu'elle appelle la tâche devant débuter le calcul du minimum. Que faudrait-il modifier dans le code de `Creer_Anneau` pour qu'elle puisse réellement choisir la tâche qui va débuter le calcul ?

Question 8
On suppose que la tâche `Creer_Anneau` peut choisir la tâche qui démarre le calcul et que l'identité de celle-ci n'est donc pas connue a priori. Écrivez le nouveau code exécuté par une tâche.

Question 9
Le choix que l'on a fait pour le traitement de la réception provoque en fait un interblocage. Expliquez pourquoi, et proposez une modification simple du programme permettant de résoudre ce problème.

Question 10
Maintenant que le problème de l'interblocage a été résolu, on considère que la tâche `Creer_Anneau` peut lancer plusieurs demandes de calcul à la suite l'une de l'autre, avec une tâche initiatrice qui peut être différente à chaque fois. Expliquez pourquoi, si on demande deux fois un calcul à partir de la même tâche, on peut avoir un programme qui boucle. Proposez une solution pour résoudre ce problème.

Exercice 32 – Nouvelle version du calcul du minimum sur un anneau

Nous souhaitons programmer le calcul du minimum sur un anneau comme vu en TD. A la différence de l'exercice fait en TD, dans cet exercice, ce ne sont pas les tâches qui demandent l'identité de leur successeur à la tâche de création de l'anneau. La tâche de création de l'anneau crée l'ensemble des tâches de l'anneau et transmet le pointeur sur toute nouvelle tâche créée à celle créée précédemment pour les relier. N'oubliez pas de fermer l'anneau (la tâche qui suit la dernière créée est la première).

Question 1
Complétez le fichier `anneau_min.adb` pour permettre le calcul du minimum des valeurs associées à chacune des tâches de l'anneau.

Exercice 33 – Anneau doublement chaîné

Nous nous intéressons ici à la construction d'un anneau de tâches doublement chaîné. Chaque tâche possède un identifiant unique (son numéro) et une valeur entière qui lui est affectée par tirage aléatoire (qu'on pourrait interpréter comme le résultat d'un calcul effectué par la tâche).

Question 1
Complétez le fichier `anneau_double.adb` pour construire l'anneau doublement chaîné et afficher les informations sur les tâches le composant. Pour faciliter la création de l'anneau, vous pourrez déclarer un tableau de tâches `tache` dans la tache `Creer_Anneau`. Voici un affichage possible pour un anneau à 5 tâches :

```
Tache: 2 (val:  39) --- Succ: 3 (val: 6) --- Pred: 1 (valeur: 13)
Tache: 3 (val:  6) --- Succ: 4 (val: 68) --- Pred: 2 (valeur: 39)
Tache: 4 (val:  68) --- Succ: 5 (val: 45) --- Pred: 3 (valeur: 6)
Tache: 5 (val:  45) --- Succ: 1 (val: 13) --- Pred: 4 (valeur: 68)
Tache: 1 (val:  13) --- Succ: 2 (val: 39) --- Pred: 5 (valeur: 45)
```

Question 2

Nous introduisons maintenant une notion de faute dans l'application répartie : toutes les tâches exécutent le même calcul et doivent donc obtenir le même résultat. Une tâche qui a une valeur différente des autres est donc fautive. Nous nous plaçons dans le cas où il y a au plus une tâche fautive. Modifiez votre programme pour que la tâche `Creer_Anneau` :

- détermine de manière aléatoire si l'exécution est fautive ;
- si c'est le cas, choisisse aléatoirement l'identité de la tâche fautive et attribue à son résultat une valeur différente des résultats des autres (qui sont tous identiques). Si l'exécution n'est pas fautive, toutes les tâches ont le même résultat.

Voici une exécution possible en présence d'une faute :

```
Tache: 2 (val:  68) --- Succ: 3 (val: 68) --- Pred: 1 (valeur: 68)
Tache: 3 (val:  68) --- Succ: 4 (val: 136) --- Pred: 2 (valeur: 68)
Tache: 4 (val:  136) --- Succ: 5 (val: 68) --- Pred: 3 (valeur: 68)
Tache: 5 (val:  68) --- Succ: 1 (val: 68) --- Pred: 4 (valeur: 136)
Tache: 1 (val:  68) --- Succ: 2 (val: 68) --- Pred: 5 (valeur: 68)
```

Voici une exécution en absence de faute :

```
Tache: 2 (val:  13) --- Succ: 3 (val: 13) --- Pred: 1 (valeur: 13)
Tache: 3 (val:  13) --- Succ: 4 (val: 13) --- Pred: 2 (valeur: 13)
Tache: 4 (val:  13) --- Succ: 5 (val: 13) --- Pred: 3 (valeur: 13)
Tache: 5 (val:  13) --- Succ: 1 (val: 13) --- Pred: 4 (valeur: 13)
Tache: 1 (val:  13) --- Succ: 2 (val: 13) --- Pred: 5 (valeur: 13)
```

Le programme doit maintenant permettre d'identifier la tâche fautive (si elle existe) et la supprimer de l'anneau en réorganisant le chaînage. Pour cela, chaque tâche va comparer son résultat avec celui de ses voisins. Si la tâche a une valeur différente de celle de ses voisins, elle est fautive et doit se supprimer de l'anneau en indiquant les nouveaux chaînages à ses voisins.

Question 3

Modifiez votre programme pour que chaque tâche, après affichage de ses informations, afffiche si elle est fautive. Voici une exécution possible :

```
Tache: 2 (val:  30) --- Succ: 3 (val: 30) --- Pred: 1 (valeur: 30)
Tache: 3 (val:  30) --- Succ: 4 (val: 60) --- Pred: 2 (valeur: 30)
Tache: 4 (val:  60) --- Succ: 5 (val: 30) --- Pred: 3 (valeur: 30)
 4 est fautive
Tache: 5 (val:  30) --- Succ: 1 (val: 30) --- Pred: 4 (valeur: 60)
Tache: 1 (val:  30) --- Succ: 2 (val: 30) --- Pred: 5 (valeur: 30)
```

Question 4

Nous souhaitons maintenant que la tâche fautive se détache de l'anneau. Pour cela, elle doit transmettre, à la tâche qui la suit et à celle qui la précède, l'identité de la tâche qui la remplace dans chacun des cas. Vous utiliserez deux entrées (une pour la modification de la tâche suivante et une pour la tâche précédente). Chaque tâche modifiant ses données devra afficher les informations la concernant.

ATTENTION, toute tâche est susceptible d'être appelée sur chacune des deux entrées mais pour chaque tâche, une entrée au plus sera effectivement activée. Vous devez faire en sorte que vos tâches se terminent dans tous les cas. Voici une exécution possible :

```
Tache: 2 (val:  132) --- Succ: 3 (val: 66) --- Pred: 1 (valeur: 66)
Tache: 3 (val:  66) --- Succ: 4 (val: 66) --- Pred: 2 (valeur: 132)
 2 est fautive
Tache: 4 (val:  66) --- Succ: 5 (val: 66) --- Pred: 3 (valeur: 66)
Tache: 5 (val:  66) --- Succ: 1 (val: 66) --- Pred: 4 (valeur: 66)
Tache: 1 (val:  66) --- Succ: 2 (val: 132) --- Pred: 5 (valeur: 66)
Modif: Tache: 1 (val:  66) --- Succ: 3 (val: 66) --- Pred: 5 (valeur: 66)
Modif: Tache: 3 (val:  66) --- Succ: 4 (val: 66) --- Pred: 1 (valeur: 66)
```

Thème 11

Objectifs

- vers un environnement complètement réparti : le problème de la terminaison
- interblocage

Exercices

Le premier exercice sera fait en TD et les deux suivants en TME.

Exercice 34 – Terminaison sur un anneau

Nous souhaitons écrire un programme construisant un anneau de tâches. Comme dans l'exercice de TME de la semaine dernière, ici ce n'est pas la tâche créée qui attend l'attribution d'un successeur. Toute nouvelle tâche créée se voit affecter la dernière tâche créée avant elle comme prédécesseur.

Question 1

Complétez le code ci-dessous en suivant les consignes indiquées par les commentaires. Le type `Id_Tache` permet de numéroter les tâches dans l'anneau. Ainsi, `Id_Tache'First` correspond au numéro de la première tâche de l'anneau et `Id_Tache'Last` correspond au numéro de la dernière tâche de l'anneau (qui a pour successeur la première tâche).

```
with Ada.Text_IO, Ada.Numerics.Discrete_Random;
use Ada.Text_IO;

procedure taches_dyn is

   type id_tache is range 1 .. 10;
   type noeud;
   type a_noeud is access noeud;

   task type noeud (id : id_tache) is
      entry init (succ : in a_noeud);
      -- Recupere le successeur de la tache
      entry recoit (msg : in Natural; dest : in id_tache);
      -- Reception d'un message.
      -- Si la tache qui recoit le message est la tache destinataire (dest),
      --    elle affiche le message au format suivant: <id> -> j'ai recu le message <msg>.
      -- Sinon, elle transmet le message a son successeur sur l'anneau.
      entry afficher(succ : out a_noeud);
      -- Affiche l'identite de la tache. Le parametre de sortie est un pointeur sur la tache suivante.
   end noeud;

   task body noeud is
      -- A COMPLETER
   end noeud;

   procedure Afficher_anneau (tache_dep : a_noeud) is
      -- A COMPLETER : la tache principale appelle successivement toutes les taches de l'anneau
   end;

   -- tache_1 est la premiere tache creee, celle a partir de laquelle on peut "entrer" dans l'anneau
   -- VOUS POUVEZ AJOUTER D'AUTRES VARIABLES SI VOUS EN AVEZ BESOIN
   tache_1: a_noeud;

begin
   Reset (G);
```

```
   -- AJOUTER LES INSTRUCTIONS DE CREATION DE L'ANNEAU

   Afficher_anneau(tache_1);
end taches_dyn;
```

Maintenant que l'anneau est construit, nous souhaitons faire communiquer les tâches par échange de messages.

Question 2
A quel endroit du corps d'une tâche `Noeud` doit on inclure l'émission d'un message ? Que se passe-t-il si chaque tâche envoie un message à son successeur ?

Question 3
Une tâche (sauf la première) qui n'a pas eu de demande de synchronisation après un délai de 0.3 secondes décide aléatoirement d'envoyer ou non un nouveau message à un destinatire choisi aléatoirement.

Complétez votre code pour effectuer ces communications.

Nous voulons donc gérer la terminaison des tâches sans utiliser de clause **`terminate`**. Nous allons pour cela utiliser le fait que notre mécanisme de communication est FIFO : si un site p envoie un message m_1 puis un message m_2 à un site r, r reçoit dans l'ordre m_1 puis m_2.

L'idée de l'algorithme de terminaison est la suivante : après un délai aléatoire, une tâche initiateur envoie sur l'anneau un message particulier appelé jeton. Une tâche qui reçoit le jeton n'a plus le droit de créer de nouveau message (elle doit par contre continuer à relayer les messages qui sont déjà en circulation). Lorsque le jeton revient à un site, il ne peut plus y avoir de message en circulation devant transiter par ce site : en effet, tous les messages ayant été émis avant le premier passage du jeton, ils sont forcément reçus avant le retour de celui-ci (FIFO). Le site peut donc transmettre à nouveau le jeton et se terminer. Lorsque le jeton revient pour la deuxième fois au site initiateur, il n'est plus transmis et la tâche se termine.

Question 4
Modifiez votre code pour prendre en compte ce nouveau mécanisme de gestion de la terminaison.

Exercice 35 – Accès à une piscine

Nous nous intéressons à l'accès à une piscine par un ensemble de baigneurs. La piscine dispose d'un certain nombre de cabines et de paniers. Pour pouvoir nager, un baigneur doit d'abord prendre un cabine, puis un panier dans lequel il met ses affaires. Il libère ensuite la cabine avant d'aller se baigner. Lorsqu'il veut quitter la piscine, le baigneur doit prendre une cabine dans laquelle il vide son panier avant de le rendre. En quittant la piscine, il libère la cabine.

Question 1
Complétez le fichier `piscine.adb`. Chaque baigneur sera représenté par une tâche. Un même type protégé sera utilisé pour représenter l'ensemble des paniers et l'ensemble des cabines. Pour favoriser le parallélisme, chaque baigneur effectuera une attente de durée aléatoire après chaque accès à une ressource.

Attention, lors de l'exécution de votre programme vous devez constater qu'il y a un interblocage.

Question 2
Nous souhaitons maintenant supprimer l'interblocage. Nous envisageons deux solutions :

1. modifier le nombre de ressources,
2. changer l'ordre de prise et/ou de libération des ressources

Vous devez programmer les deux solutions :

1. Donnez, sous forme d'un commentaire dans votre programme, une condition sur le nombre de ressources qui garantit qu'il n'y aura pas de blocage, sans qu'il soit nécessaire de modifier l'ordre de prise et/ou de libération des ressources.

2. Modifiez votre programme (en changeant l'ordre de prise et/ou de libération des ressources) pour que, indépendamment du nombre de ressources, il ne puisse plus y avoir de blocage.

Exercice 36 – Dîner des philosophes

Cet exercice a pour but de vous faire programmer un problème classique de programmation concurrente : "le dîner des philosophes" dans lequel `N` philosophes sont autour d'une table ronde, `N` fourchettes alternent avec les philosophes. Il y a autant de fourchettes que de philisophes. Pour 5 philosophes nous avons la configuration donnée par la figure 4 :

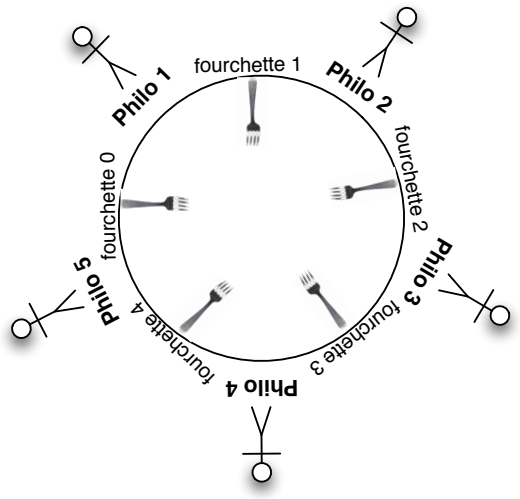


FIGURE 4 – Tablée de 5 philosophes

Pour pouvoir manger chaque philosophe doit disposer des fourchettes de trouvant à sa droite et à sa gauche. Chaque philosophe prend sa fourchette de gauche puis sa fourchette de droite, mange et repose sa fourchette de gauche puis sa fourchette de droite. Les philosophes mangent plusieurs fois.

Question 1

Complétez le fichier `philo.adb` suivant. Les philosophes sont du type tâche `philosophe` et les fourchettes du type protégé `fourchette`. Vous ajouterez des affichages pour pouvoir suivre l'évolution de l'exécution, en particulier il serait intéressant de voir quel philosophe est en attente de quelle fourchette, quel philosophe prend quelle fourchette, quel philosophe rend quelle fourchette et quel philosophe mange. Le nombre de philosophes est défini par la constante `nb_philos` et le nombre de fois où il devra manger est défini par la constante `nb_repas`.

```
with Ada.Text_IO, Ada.Numerics.Discrete_Random;
use Ada.Text_IO;

procedure philo is

   ------------------- Gestion de durees aleatoires
   package Alea_Int is new Ada.Numerics.Discrete_Random (Natural);
   use Alea_Int;

   Graine : Generator;

   function Duree_Attente return Duration is
   begin
      return 0.1 * Duration (Random (Graine) mod 10);
   end Duree_Attente;
   -----------------------------------------------------

-- nombre de philosophes
nb_philos : constant Natural := 6;
-- nombre de repas
```

```
nb_repas : constant Natural := 4;

task type philosophe is
  entry init(id : Natural);
end philosophe;

protected type fourchette is
  entry prendre(id_philo : natural; id_fourchette : natural);
  procedure rendre(id_philo : natural; id_fourchette : natural);
  private
    disponible : boolean := true;
end fourchette;

tab_philo : array (1..nb_philos) of philosophe;
tab_fourchette : array(0..nb_philos-1) of fourchette;

-- A completer

end philo;
```

Question 2

Vous avez dû constater que la solution précédente présente un inter-blocage. Modifiez votre programme pour garantir qu'un philosophe prend ses deux fourchettes de façon atomique. **Attention**, la prise des fourchettes doit, comme dans la question précédente, s'effectuer en deux actions distinctes. Vous devez donc garantir que lorsqu'un philosophe a commencé à prendre sa fourchette de gauche, aucun autre philosophe ne pourra prendre de fourchette tant que le premier n'aura pas les deux siennes. Vérifiez qu'il n'y a plus d'inter-blocage.