LINFO1104 TP10: Message Passing

Cette session se concentre sur l'utilisation des ports, nottament sur l'utilisation de constructions fonctionnelles comme modèle concurrentiel.

1. **Ports**. L'abstraction la plus basique pour permettre le Message Passing est l'utilisation des ports. Les ports sont des canaux de communication où l'on peut envoyer et lire des messages. Chaque port est associé à un stream dans lequel on envoie les messages. Pour rappel, un stream est une liste infinie. Pour créer un port, il faut utiliser la procédure `{NewPort S P}` où `P` est le port crée et `S` le stream qui y est associé. Pour envoyer un message à un port, on utilise la procédure `{Send P Msg}`, qui envoie le message `Msg` à `P`. Le code suivant est un exemple sur l'utilisation des ports:

   ```
   declare
   P S
   {NewPort S P}

   {Send P foo}
   {Send P bar}
   ```

   - Qu'observe-t-on lorsque qu'on execute la ligne suivante , `{Browse S}`?
   - Implémentez une procédure permettant d'afficher chaque message envoyé à `P`.

2. **Calculette concurrente**. Mettez en place un serveur recevant le type de messages suivants:

   - `add(X Y R)` : Ajoute le résultat de l'addition de `X` et `Y` à `R`.
   - `pow(X Y R)` : Calcule `X` à la puissance de `Y` et lie le résultat à `R`.
   - `'div'(X Y R)` : Divise `X` par `Y` et lie le résultat à `R`.

   Vous pouvez considérer toutes les valeurs d'entrée comme des entiers. Si le message reçu par le serveur ne correspond à aucun des 3 messages décrits ci-dessus, il doit afficher `je ne comprends pas ton message` dans la sortie de l'émulateur. Le code suivant peut être utilisé pour tester le serveur. Notez que la fonction que vous devez implémenter est `LaunchServer`. Vous devrez peut-être ajouter quelques instructions pour vous assurer que les appels à `Show` ou `Browse` are affichent toujours des informations valides. Quel type de protocole utilisez-vous ?

   ```
   declare A B N S Res1 Res2 Res3 Res4 Res5 Res6

   S = {LaunchServer}
   {Send S add(321 345 Res1)}
   {Browse Res1}

   {Send S pow(2 N Res2)}
   N = 8
   {Browse Res2}

   {Send S add(A B Res3)}
   {Send S add(10 20 Res4)}
   {Send S foo}
   {Browse Res4}
   A = 3
   B = 0-A
   {Send S 'div'(90 Res3 Res5)}
   {Send S 'div'(90 Res4 Res6)}
   {Browse Res3}
   {Browse Res5}
   {Browse Res6}
   ```

3. **Sacrée Charlotte**. Une étudiante en marketing, Charlotte, réalise une étude sur la consommation de bière pendant les 24 heures vélo. Elle dispose d'une liste de ports à contacter pour chaque étudiant de l'UCLouvain, qu'elle a obtenue grâce à une API secrète qui peut être obtenue sur le site uclouvain.be, depuis le répertoire des étudiants Pour son étude, elle envoie un message à chaque étudiant pour savoir combien de bières chacun d'entre eux a consommé. Une fois qu'elle aura rassemblé toutes les

informations, elle saura :

- combien de bières ont été consommées au total,
- quelle était la consommation moyenne,
- quel était le minimum et
- le maximum de consommation.

Choisissez l'une des métriques suivantes et rédigez un petit programme pour aider Charlotte à accomplir sa tâche.

Implémentez Charlotte comme serveur pouvant supporter deux types d'étudiants (clients): le premier type travaille comme le RMI. Le second répond en envoyant un message au port de Charlotte. Les exemples d'étudiants sont donnés dans les codes du dessous.

```
fun {StudentRMI}
  S in
  thread
    for ask(howmany:Beers) in S do
      Beers={OS.rand} mod 24
    end
  end
  {NewPort S}
end
```

```
fun {StudentCallBack}
  S in
    thread
      for ask(howmany:P) in S do
        {Send P {OS.rand} mod 24}
      end
    end
  {NewPort S}
end
```

Le code suivant peut vous aider à créer une liste d'élèves prêts à répondre aux questions de Charlotte.

```
fun {CreateUniversity Size}
  fun {CreateLoop I}
    if I =< Size then
      % pour {Student} choisissez soit StudentRMI ou StudentCallBack,
      % défini plus haut, selon l'humeur de Charlotte
      {Student}|{CreateLoop I+1}
    else nil end
  end
in
  {CreateLoop 1}
end
```

4. **Port Objects**. C'est un concept très important. Comme vous le savez peut-être, certaines des propriétés importantes des objets sont l'état et l'encapsulation. Ces deux propriétés sont également présentes dans les `Port Objects`. Un `Port Object` commence comme tout objet, avec un état initial. Chaque fois qu'il reçoit un message, il utilise le message reçu et son état pour appeler une fonction qui met en œuvre le comportement encapsulé de l'objet. La fonction renvoie le nouvel état de l'objet, et celui-ci est prêt à recevoir le message suivant. C'est exactement comme une méthode qui appelle d'autres langages orientés objet. Le code suivant est une implémentation d'un `Port Object`.

```
fun {NewPortObject Behaviour Init}
  proc {MsgLoop S1 State}
    case S1 of Msg|S2 then
      {MsgLoop S2 {Behaviour Msg State}}
    [] nil then skip
    end
  end
  Sin
in
  thread {MsgLoop Sin Init} end
  {NewPort Sin}
end
```

La fonction renvoie un port. Elle reçoit comme arguments une fonction que nous appelons `Behaviour`,

et l'état initial `Init`.

Créez une fonction `Portier` qui compte le nombre de personnes entrant dans un concert. `Portier` utilise un objet port qui reçoit trois types de messages : `getIn(N)`, `getOut(N)` et `getCount(N)`. Le message `getIn(N)` incrémente le compteur interne de `N` personnes. Le message `getOut(N)` décrémente le compteur de `N` personnes. Maintenant, on peut à tout moment demander combien de personnes se trouvent à l'intérieur avec le message `getCount(N)`, ce qui lie la variable `N` du message au nombre actuel de personnes.

5. **Ma Stack bien aimée**. Implémentez la fonction `NewStack` sans arguments, qui renvoie un `Port Object` représentant une pile. Utilisez la fonction `NewPortObject` donnée ci-dessus (ou utilisez la vôtre si vous le souhaitez). Ensuite, implémentez les trois opérations principales qui peuvent être effectuées sur une pile. `{Push S X}`, qui pousse l'élément `X` dans la pile `S`. `{Pop S}` qui renvoie l'élément en haut de la pile `S`, et `{IsEmpty S}`, qui renvoie une valeur booléenne indiquant si la pile `S` est vide ou non. Chacune de ces procédures doit envoyer le message correspondant à l'objet port créé par `NewStack`.

6. **File concurrente**. Maintenant que vous savez comment mettre en place une pile, la mise en place d'une file d'attente ne sera pas du tout difficile. Implémentez la fonction `NewQueue` sans argument, qui renvoie un objet port représentant une file d'attente FIFO. Les opérations suivantes doivent être implémentées : `{Enqueue Q X}` qui met l'élément `X` dans la file d'attente `Q`. `{Dequeue Q}` qui renvoie le premier élément de la file d'attente. `{IsEmpty Q}` qui renvoie si `Q` est vide ou non, et `{GetElements Q}` qui renvoie tous les éléments de la file d'attente.

7. **Mineurs**. Écrivez une autre version du programme pour compter les mineurs sauvés (voir TP8, exercice 5). Dans ce programme, implémentez le serveur de comptage en utilisant `NewPortObject`. Notez qu'en dehors de l'état interne, le serveur de comptage doit également produire un flux de listes, avec la quantité de caractères comptés jusqu'à présent. Écrivez une fonction `{Counter Output}` qui renvoie un `Port Object`. La variable `Output` correspond au flux avec l'historique des mineurs comptés.

   Astuce: Pensez à la sortie comme le flux d'un autre port. Ensuite, la fonction `Behaviour`, qui modifie l'état du serveur, enverra d'abord l'état au port de sortie, puis renverra le nouvel état.

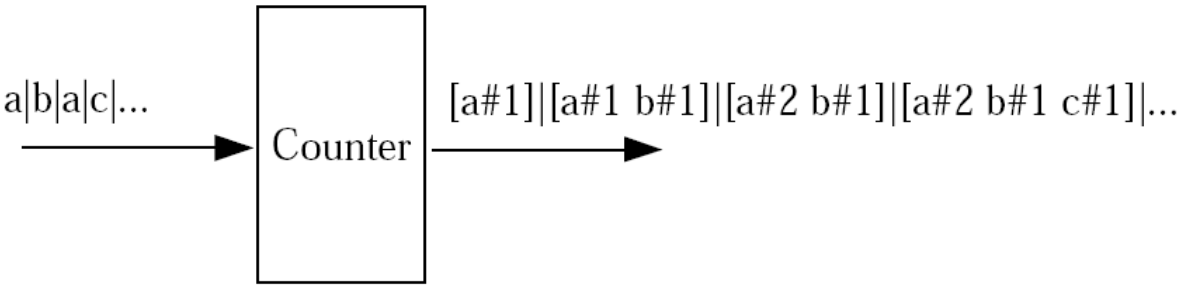

Figure 1: Programme des mineurs

Pour rappel, le programme doit permettre de suivre l'apparition de caractères dans un flot. Le programme doit générer un flot de sortie où chaque position correspond auxcaractères vu à ce moment. Chaque position du flot de sortie est une liste de tuples `C#N`, où `N` est le nombre de fois que `C` apparaît dans le flot d'entrée