LINFO1104 TP6: Suite lambda calcul et Programmation OO

Pour cette séance, finissez d'abord le TP 5 avant de commencer la partie orienté objet.

Lambda calcul, ordre d'évaluation

11. **Stratégie d'évaluation**.
    - Réduisez l'expression suivante selon l'ordre applicatif et l'ordre normal:

    $(\lambda x \lambda y \lambda u.y)((\lambda z.((\lambda x.z)\lambda z.v))z)$

    - L'affirmation suivante est-elle vraie? La première β-réduction effectué pour l'expression suivante sera la même peu importe l'ordre utilisé.

    $((\lambda v \lambda u(vv))\lambda z.x)v$

12. **Ordre normal et applicatif**.
    - Réduisez l'expression suivante selon l'odre normal et l'ordre applicatif:

$(\lambda x.m)((\lambda x(xx))\lambda x(xx))$

- Que remarquez vous? L'ordre utilisé a-t-il un impact sur le résultat final? Si oui pourquoi? Sinon pourquoi?

États explicites, programmation orientée objet et abstraction de données

Dans cette séance, nous allons utiliser des *cellules* et des *tableaux* pour implémenter des algorithmes. Nous allons ainsi pouvoir comparer des versions déclaratives et non déclaratives de plusieurs algorithmes.

Dans ces exercices, nous utiliserons souvent des boucles `for`, sur des listes et des intervalles d'entiers. La syntaxe des deux types de boucle est schématisée ci-dessous. La boucle à gauche exécute l'instruction donnée pour chaque élément `X` de la liste `Xs`. La boucle à droite exécute l'instruction donnée pour chaque entier `I` entre `A` et `B` (dans l'ordre de `A` à `B`).

```
for X in Xs do instruction end
```

```
for I in A..B do instruction end
```

Chacune est en fait une notation pour un appel d'une procédure récursive, à savoir:

```
local
   proc {Loop L}
      case L of X|T then
         instruction
         {Loop T}
      else skip end
   end
in
   {Loop Xs}
end
```

```
local
   proc {Loop I}
      if I=<B then
         instruction
         {Loop I+1}
      else skip end
   end
in
   {Loop A}
end
```

Les identificateurs `Loop`, `L` et `T` sont choisis de telle sorte qu'ils n'apparaissent pas dans `instruction`.

1. **Accumulateurs et état.** Pour implémenter des fonctions efficaces dans le paradigme déclaratif, nous avons utilisé des accumulateurs dans les paramètres des fonctions. Un accumulateur est une forme d'état implicite. Un exemple typique est la fonction `Reverse`, qui renvoie la liste passée en argument avec ses éléments dans l'ordre inverse.

```
fun {Reverse Xs}
   fun {ReverseAux Xs Ys}
      case Xs of nil then Ys
      [] X|Xr then {ReverseAux Xr X|Ys}
      end
   end
```

```
in
   {ReverseAux Xs nil}
end
```

Réécrivez cette fonction pour rendre explicite l'état passé dans l'argument `Ys` de `ReverseAux`. En d'autres mots, construisez une implémentation de la fonction qui utilise une cellule. *Conseil:* Utilisez une boucle `for`.

- A votre avis, votre fonction est-elle efficace, comparée à la définition avec accumulateur ci-dessus? Calculez le nombre d'appels récursifs.
- L'état explicite que vous avez introduit est-il visible ou encapsulé? S'il est encapsulé, à quel endroit? Combien de fois créez-vous une cellule ?
- Pouvez-vous également créer un état explicite pour l'argument `Xs`? Si c'est le cas, implémentez-le. Sinon, expliquez pourquoi.

2. **Une calculatrice.** Dans cet exercice, nous allons implémenter une calculatrice utilisant la notation *postfixe* (ou *notation polonaise inversée*). Dans cette notation, l'opérateur est écrit après ses deux opérandes. Par exemple, l'expression `(13+45)*(89-17)` se note "`13 45 + 89 17 - *`". L'avantage de cette notation est qu'elle ne nécessite pas l'utilisation de parenthèses.

   L'évaluation d'une expression postfixe est très simple. On utilise une *pile* pour stocker les résultats intermédiaires. Il suffit de parcourir l'expression de gauche à droite, et de traiter les éléments (valeurs et opérateurs) un à un. Lorsqu'on lit une valeur, on la met sur la pile. Lorsqu'on lit un opérateur, on retire les deux valeurs au sommet de la pile, on leur applique l'opérateur, puis on met le résultat sur la pile. A la fin, la pile doit contenir une seule valeur: le résultat du calcul. Ci-dessous, de gauche à droite, les étapes du calcul de l'expression postfixe "`13 45 + 89 17 - *`". La ligne supérieure contient le reste de l'expression, la ligne inférieure la pile des valeurs intermédiaires.

```
13 45 + 89 17 - *
45 + 89 17 - *      13
+ 89 17 - *         45 13
89 17 - *           58
17 - *              89 58
- *                 17 89 58
*                   72 58
                    4176
```

   Ecrivez une abstraction de données *pile*. Cette abstraction est définie par quatre opérations: `NewStack`, `IsEmpty`, `Push` et `Pop`. `Push` est une procédure, les autres sont des fonctions.

   - `{NewStack}` renvoie une nouvelle pile.
   - `{IsEmpty S}` renvoie `true` si la pile `S` passée en argument est vide, `false` sinon.
   - `{Push S X}` empile l'élément `X` sur la pile `S`.
   - `{Pop S}` enlève le sommet de la pile `S` et le renvoie.

   Une pile est représentée par une cellule contenant une liste. La liste contient les éléments de la pile, du sommet à la base. Le premier élément de la liste est donc le sommet de la pile.

   Ensuite, écrivez une fonction `Eval` qui prend en paramètre une expression postfixe et renvoie son évaluation. L'expression est représentée par une liste dont chaque élément est soit un entier, soit un des atomes `'+'`, `'-'`, `'*'` et `'/'`, cette dernière devant être interprétée comme la division entière. L'expression de l'exemple ci-dessus sera représentée par la liste `[13 45 '+' 89 17 '-' '*']`. La fonction `Eval` parcourt cette liste et met à jour le contenu d'une pile, en suivant l'algorithme présenté informellement ci-dessus.

```
{Browse {Eval [13 45 '+' 89 17 '-' '*']}}   % affiche 4176 = (13+45)*(89-17)
{Browse {Eval [13 45 '+' 89 '*' 17 '-']}}   % affiche 5145 = ((13+45)*89)-17
{Browse {Eval [13 45 89 17 '-' '+' '*']}}   % affiche 1521 = 13*(45+(89-17))
```

3. **Encapsulation de l'état de la pile.** Nous allons maintenant implémenter une variante de l'abstraction *pile* présentée dans l'exercice précédent. Cette variante rend l'état de la pile invisible à l'utilisateur. Il s'agit donc de ''cacher" la cellule contenant la liste par portée lexicale. La fonction `NewStack` ne peut donc pas renvoyer directement la cellule. Elle va plutôt renvoyer un enregistrement de fonctions et procédures:

```
{NewStack} -> stack(isEmpty:IsEmpty push:Push pop:Pop)
```

Les fonctions `IsEmpty`, `Pop` et la procédure `Push` sont *spécifiques* à la pile qui vient d'être créée. Voici un exemple d'utilisation, avec deux piles.

```
declare
Stack1={NewStack}              % pile 1
Stack2={NewStack}              % pile 2

{Browse {Stack1.isEmpty}}      % affiche true; la pile 1 est vide
{Stack1.push 13}               % empile 13 sur la pile 1
{Browse {Stack1.isEmpty}}      % affiche false; la pile 1 n'est pas vide
{Browse {Stack2.isEmpty}}      % affiche true; la pile 2 est toujours vide
{Stack1.push 45}               % empile 45 sur la pile 1
{Stack2.push {Stack1.pop}}     % enlève 45 de la pile 1 et l'empile sur la pile 2
{Browse {Stack2.isEmpty}}      % affiche false; la pile 2 n'est pas vide
{Browse {Stack1.pop}}          % enlève 13 de la pile 1 et l'affiche
```

Voici un squelette d'implémentation pour `NewStack`. Complétez-le.

```
fun {NewStack}
   ...
   fun {IsEmpty} ... end
   proc {Push X} ... end
   fun {Pop} ... end
in
   stack(isEmpty:IsEmpty push:Push pop:Pop)
end
```

- Adaptez la fonction `Eval` que vous avez écrite dans l'exercice précédent afin qu'elle utilise cette implémentation de l'abstraction de données *pile*.
- Quel type d'abstraction de données avez-vous réalisé pour la pile ? Existe-t-il d'autres types d'abstraction de données possibles ?

4. **Mélanger une liste.** Dans cet exercice, nous allons utiliser un tableau pour "mélanger" les éléments d'une liste. Vous devez définir une fonction `Shuffle` qui prend en argument une liste `Xs` et renvoie une liste, contenant les mêmes éléments que `Xs`, mais dans un ordre aléatoire.

```
{Browse {Shuffle [a b c d e]}}   % peut afficher [d c a b e]
```

Vous utiliserez la fonction `Random` définie ci-dessous pour effectuer des tirages aléatoires d'entiers entre 1 et l'argument `N`.

```
%% renvoie un nombre aleatoire entre 1 et N
fun {Random N} {OS.rand} mod N + 1 end
```

Voici un rappel des principales opérations sur les tableaux en Oz.

`{NewArray I J X}` renvoie un tableau indicé de `I` à `J` dont les éléments sont initialisés avec la valeur `X`. Sa complexité est de Θ(`J`-`I`).

`A.I` renvoie le `I`ème élément du tableau `A` et s'exécute en $\Theta(1)$.

`A.I:=X` affecte la valeur `X` au `I`ème élément de `A`, aussi en $\Theta(1)$.

L'algorithme que vous devez utiliser est le suivant. Tout d'abord, mettez les éléments de la liste dans un tableau a, indicé de 1 à n, où n est le nombre d'éléments de la liste. Ensuite, tirez un nombre aléatoire i entre 1 et n. Le $i$ème élément de a est la tête de la liste résultat. Copiez l'élément $a(n)$ à l'indice i. Les éléments $a(1)$, ..., $a(n\text{-}1)$ sont les éléments restants. Faites un nouveau tirage en remplaçant n par n-1 pour choisir le deuxième élément de la liste résultat, et ainsi de suite jusqu'au dernier.

Voici une représentation de l'exécution de l'exemple ci-dessus. La colonne de gauche représente les états successifs du tableau, la seconde colonne est l'indice i choisi, et la colonne à droite montre le résultat "en construction". Les `*` dans le tableau montrent les éléments ignorés dans le tableau.

tableau	i	résultat
a b c d e	4	d|_
a b c e *	3	d|c|_
a b e * *	1	d|c|a|_

tableau	i	résultat
e b * * *	2	d|c|a|b|_
e * * * *	1	d|c|a|b|e|_
* * * * *		d|c|a|b|e|nil

Pourriez-vous trouver un algorithme *déclaratif* simple qui résoud le même problème ?

5. **Cellules et listes**

```
L1={NewCell 0}|{NewCell 1}|{NewCell 2}|nil
L2=0|{NewCell 1|{NewCell 2|{NewCell nil}}}
```

- L1 est-elle une liste? et L2?
- Ecrivez des fonctions pour ajouter un élément au début de chacune de ces deux structures.
- Ecrivez des fonctions pour ajouter un élément à la fin de chacune de ces deux structures.
- Ecrivez des fonctions pour inverser les deux premiers éléments dans chacune de ces deux structures.
- Quelles sont les complexité temporelles et spatiales de ces six fonctions?

Programmation oritentée objet

Nous allons définir des objets à partir de *classes*, et mettre en évidence le concept de *polymorphisme*. L'exemple ci-dessous rappelle la syntaxe d'une classe, la création d'un objet et son utilisation.

```
declare
class Counter
   attr value
   meth init   % (re)initialise le compteur
      value:=0
   end
   meth inc   % incremente le compteur
      value:=@value+1
   end
   meth get(X)   % renvoie la valeur courante du compteur dans X
      X=@value
   end
end

MonCompteur={New Counter init}
for X in [65 81 92 34 70] do {MonCompteur inc} end

{Browse {MonCompteur get($)}}   % affiche 5
```

Notez la présence du signe `$`: l'expression `{MonCompteur get($)}` est équivalente à

```
local X in {MonCompteur get(X)} X end
```

6. **Collections.** Une *collection* regroupe des valeurs. Voici une classe qui implémente des collections. Trois méthodes sont définies: `put(X)`, `get(X)` et `isEmpty(B)`.

```
class Collection
   attr elements
   meth init   % initialise la collection
      elements:=nil
   end
   meth put(X)   % insere X
      elements:=X|@elements
   end
   meth get($)   % extrait un element et le renvoie
      case @elements of X|Xr then elements:=Xr X end
   end
   meth isEmpty($)   % renvoie true ssi la collection est vide
      @elements==nil
   end
end
```

Notez que par défaut, le corps d'une méthode est une instruction. Mais la notation `$` permet de définir des méthodes comme des fonctions. Leur contenu doit être une expression qui décrit la valeur renvoyée. Dans l'exemple, la méthode `put(X)` est définie par une instruction, tandis que la méthode `isEmpty($)` l'est par une expression.

- Ajoutez une méthode `union(C)` à cette classe. L'appel `{C1 union(C2)}` fait l'union des collections `C1` et `C2`. Après l'appel, `C1` contient cette union et `C2` est vide. Votre méthode doit être *polymorphe*, c'est-à-dire qu'elle ne doit pas dépendre des implémentations des objets concernés.
- Définissez maintenant une classe `SortedCollection` pour des *collections triées*. L'interface d'une collection triée est la même qu'une collection, à la différence près que la méthode `get(X)` renvoie les éléments dans l'ordre du tri. En d'autres mots, la méthode renvoie toujours l'élément le plus petit de la collection. Votre classe doit hériter de la classe `Collection`.
- Considérez l'appel `{C1 union(C2)}`. Quelle est sa complexité temporelle si `C1` et `C2` sont de la classe `Collection`? Que devient cette complexité si `C1` et `C2` sont de votre classe `SortedCollection`?
- Utilisez votre classe `SortedCollection` pour trier une liste. Quelle est sa complexité temporelle?
- Pouvez-vous facilement convertir une collection quelconque en une collection triée?

7. **Des objets pour représenter des expressions.** Dans cet exercice, l'idée est de représenter des expressions par des objets. Chaque sous-expression d'une expression est représentée par un objet, y compris les symboles de variables. On définit pour cela différents types d'expressions, et chaque type est implémenté par une classe propre. Par exemple, on peut représenter la formule $3x^2 - xy + y^3$ par

```
declare
VarX={New Variable init(0)}
VarY={New Variable init(0)}
local
   ExprX2={New Puissance init(VarX 2)}
   Expr3={New Constante init(3)}
   Expr3X2={New Produit init(Expr3 ExprX2)}
   ExprXY={New Produit init(VarX VarY)}
   Expr3X2mXY={New Difference init(Expr3X2 ExprXY)}
   ExprY3={New Puissance init(VarY 3)}
in
   Formule={New Somme init(Expr3X2mXY ExprY3)}
end
```

Notez qu'on utilise le même objet `VarX` pour toutes les occurrences de x dans la formule.

Ces objets vont vous permettre de réaliser deux choses : (1) **évaluer** une expression pour une valeur donnée des variables, et (2) produire la **dérivée** d'une expression par rapport à une variable donnée. Cette dérivée est elle-même une expression.

Pour évaluer la formule de l'exemple, on affecte des valeurs aux variables x et y en utilisant la méthode `set(N)` des objets `VarX` et `VarY`. Ensuite, on appelle la méthode `evalue(X)` de l'objet `Formule`. On doit pouvoir réévaluer la formule avec d'autres valeurs pour les variables. L'exemple suivant évalue la formule avec x=*7* et y=*23*, puis avec x=*5* et y=*8*.

```
{VarX set(7)}
{VarY set(23)}
{Browse {Formule evalue($)}}   % affiche 12153

{VarX set(5)}
{VarY set(8)}
{Browse {Formule evalue($)}}   % affiche 547
```

Pour dériver la formule, on appelle la méthode `derive(V E)` sur l'objet `Formule`. Le paramètre `V` est l'objet représentant la variable par rapport à laquelle on fait la dérivée. Par exemple, pour dériver par rapport à x, on utilise `V=VarX`. La méthode affecte alors à `E` un objet représentant l'expression résultat. L'exemple ci-dessous dérive la formule par rapport à x et l'évalue avec x=*7* et y=*23*.

```
declare
Derivee={Formule derive(VarX $)}   % represente 6x - y
```

```
{VarX set(7)}
{VarY set(23)}
{Browse {Derivee evalue($)}}   % affiche 19
```

Implémentez les classes `Variable`, `Constante`, `Somme`, `Difference`, `Produit` et `Puissance` pour qu'elles représentent des formules.

- Ces classes doivent toutes implémenter les méthodes `evalue(X)` et `derive(V E)`. Ces méthodes sont polymorphes, car on peut les appeler quelle que soit la classe de l'objet formule. Par exemple, l'objet `ExprXY` évalue son résultat en appelant les objets `VarX` et `VarY` avec la méthode `evalue`.
- Les classes ont également des méthodes propres. Par exemple, leur méthode d'initialisation, pour laquelle les arguments ont des significations différentes suivant les classes. Un autre exemple est la méthode `set(N)` de la classe `Variable`, qui permet d'affecter les variables avant d'évaluer la formule. Cette méthode est absente des autres classes.

Quels sont les avantages et inconvénients d'utiliser des objets et des méthodes plutôt que des enregistrements et des fonctions? Est-il difficile d'ajouter un nouveau type d'expression? Pouvez-vous facilement implémenter une méthode de simplification d'expression?

8. **Elu par cette crapule.** Un *palindrome* est un texte qui est indépendant du sens de la lecture. Les mots *"radar"*, *"ici"* et *"elu par cette crapule"* (sans les espaces) en sont des exemples. Le but de l'exercice est d'écrire un programme qui détermine si une chaîne de caractères est un palindrome.

   L'algorithme devra utiliser une classe `Sequence`. Un objet de cette classe représente une liste modifiable d'éléments. Nous utilisons le mot *séquence* pour éviter toute confusion avec les listes. L'objet permet d'accéder aux premier et dernier éléments de la séquence, ainsi que d'ajouter et retirer des éléments. Les méthodes de la classe `Sequence` sont

   - `isEmpty($)` renvoie `true` si la séquence est vide, `false` sinon.
   - `first($)` renvoie le premier élément.
   - `last($)` renvoie le dernier élément.
   - `insertFirst(X)` ajoute l'élément `X` au début de la séquence.
   - `insertLast(X)` ajoute l'élément `X` à la fin de la séquence.
   - `removeFirst` retire le premier élément.
   - `removeLast` retire le dernier élément.

   Voici l'algorithme, décrit dans les commentaires de la fonction `Palindrome`.

```
fun {Palindrome Xs}
   S={New Sequence init}
   fun {Check}
      %% si S est vide, alors Xs est un palindrome
      %% sinon, on compare les premier et dernier elements de S:
      %%  - s'ils sont egaux, on les retire de S et on continue
      %%  - sinon, Xs n'est pas un palindrome
   end
in
   %% mettre tous les elements de Xs dans S (dans l'ordre)
   {Check}
end
```

   Implémentez la classe `Sequence` et complétez la fonction `Palindrome`.