LINFO1104: TP 2

L'objectif de cette séance est de:

- Utiliser la récursivité sur les listes et les arbres;
- Manipuler les enregistrements (structure de données)

1. **Le minimum syndical.** Rappelons qu'une liste est soit la liste vide `nil`, soit une paire `H|T`, où `H` est l'élément de tête et `T` est la queue, c'est-à-dire la liste des éléments restants. Formellement, cela se définit par la grammaire (notation EBNF)

   ```
   <List T> ::= nil | T '|' <List T>
   ```

   On utilise aussi la notation `[a b c]`, qui est simplement un raccourci de notation pour la liste `a|b|c|nil`, qui se lit comme `a|(b|(c|nil))`.

   - Ecrivez les définitions des listes suivantes dans la notation de base, c'est-à-dire avec `nil` et "`|`". Affichez-les pour vérifier votre traduction.

     ```
     L1=[a]
     L2=[a [b c] d]
     L3=[proc {$} {Browse oui} end proc {$} {Browse non} end]
     L4=[est une liste]
     L5=[[a p]]
     ```

   - Creez une nouvelle liste en ajoutant `ceci` en tête de la liste `L4`.
   - La liste `L3` a pour éléments des procédures. Appelez la première procédure (en utilisant la varaiable `L3`). Vérifiez qu'elle affiche `oui`.
   - A partir de la liste `L2`, obtenez la liste `[[b c] d]`.
   - Ecrivez deux fonctions `Head` et `Tail` qui retournent respectivement la tête et la queue de la liste passée en argument.

     ```
     {Browse {Head [a b c]}}   % affiche a
     {Browse {Tail [a b c]}}   % affiche [b c]
     ```

2. **Quelle est la plus longue?** Construisez la fonction récursive `Length`, qui renvoie la longueur de la liste passée en argument. La liste vide a une longueur nulle.

   ```
   {Browse {Length [r a p h]}}       % affiche 4
   {Browse {Length [[b o r] i s]}}   % affiche 3
   {Browse {Length [[l u i s]]}}     % affiche 1
   ```

   Votre définition est-elle récursive terminale? Si ce n'est pas le cas, rendez-la récursive terminale à l'aide d'un accumulateur. N'oubliez pas de spécifier l'invariant que vous utilisez.

3. **Concaténer deux listes.** Ecrivez une fonction récursive `Append`, qui prend deux listes en argument et renvoie leur *concaténation*, c'est-à-dire les deux listes mises bout à bout.

   ```
   {Browse {Append [r a] [p h]}}       % affiche [r a p h]
   {Browse {Append [b [o r]] [i s]}}   % affiche [b [o r] i s]
   {Browse {Append nil [l u i s]}      % affiche [l u i s]
   ```

4. **Pattern matching.** Le *pattern matching* permet de comparer une valeur avec un *pattern*, c'est-à-dire un patron ou une forme. Cette technique permet de décomposer un traitement par cas de façon très concise et lisible. Par exemple, l'instruction

   ```
   case L
   of nil then {Browse rien}
   [] H|T then {Browse H} {Browse T}
   end
   ```

   teste d'abord si `L` correspond à la liste vide (`nil`), et affiche `rien` si c'est le cas. Sinon, elle teste si `L` est une liste composée d'une tête et d'une queue, et affiche ces parties. Si aucun test ne réussit, et qu'il n'y a pas de clause `else` (comme dans l'exemple), une erreur est signalée.

   - Ecrivez une fonction qui prend un argument, et renvoie `empty` si c'est la liste vide, `nonEmpty` si c'est une liste non-vide, et `other` si ce n'est pas une liste.
   - Réécrivez les fonctions `Head`, `Tail`, `Length` et `Append` (voir exercices précédents) en utilisant le pattern matching.

5. **Sous-séquences dans des listes.** Implémentez les fonctions suivantes. Tâchez de raisonner de manière inductive sur la structure des listes.

   - `{Take Xs N}` renvoie une liste contenant les `N` premiers éléments de la liste `Xs`.

     ```
     {Browse {Take [r a p h] 2}}     % affiche [r a]
     {Browse {Take [r a p h] 7}}     % affiche [r a p h]
     {Browse {Take [r [a p] h] 2}}   % quel est le resultat?
     ```

   - `{Drop Xs N}` renvoie la liste `Xs` sans ses `N` premiers éléments.

     ```
     {Browse {Drop [r a p h] 2}}     % affiche [p h]
     {Browse {Drop [r a p h] 7}}     % affiche nil
     {Browse {Drop [r [a p] h] 2}}   % quel est le resultat?
     ```

6. **Multiplions.** Écrivez une fonction `MultList` telle que `{MultList L}` renvoie le résultat de la multiplication des éléments de la liste `L`. Par exemple,

   ```
   {Browse {MultList [1 2 3 4]}}   % affiche 24
   ```

7. **Ecriture des listes.** Une liste vide est représentée par `nil`, une liste non-vide formée d'un élément H suivi d'une liste T est représentée par `H|T`. Nous pouvons donc représenter la même liste de plusieurs façons.

   ```
   declare
   {Browse [5 6 7 8]}
   {Browse 5|[6 7 8]}
   {Browse 5|6|7|8|nil}
   A=5
   B=[6 7 8]
   {Browse A|B}
   ```

   En partant de l'exemple, créez une liste de deux éléments dont le premier élément est la liste des entiers de 1 à 3 et le second élément la liste des entiers de 4 à 6. Ecrivez votre réponse une première fois en utilisant uniquement les crochets, puis une seconde fois en utilisant uniquement la barre verticale et les parenthèses.

8. **Occurrences d'une sous-liste.** Écrivez une fonction `FindString` telle que `{FindString S T}` renvoie la liste des occurrences de la succession de caractères `S` dans le texte `T`. Chaque occurrence est identifiée par la position de son premier caractère dans `T`. Par exemple,

   ```
   {Browse {FindString [a b a b] [a b a b a b]}}   % affiche [1 3]
   {Browse {FindString [a] [a b a b a b]}}         % affiche [1 3 5]
   {Browse {FindString [c] [a b a b a b]}}         % affiche nil
   ```

   Conseil: commencez par écrire une fonction `Prefix` telle que `{Prefix L1 L2}` renvoie `true` si `L1` est un préfixe de `L2`, `false` sinon. Par exemple,

   ```
   {Browse {Prefix [1 2 1] [1 2 3 4]}}   % affiche false
   {Browse {Prefix [1 2 3] [1 2 3 4]}}   % affiche true
   ```

9. **Enregistrements.** Considérez l'enregistrement suivant. Il représente la carte des menus d'un restaurant.

   ```
   Carte = carte(menu(entree: 'salade verte aux lardons'
                      plat: 'steak frites'
                      prix: 10)
                 menu(entree: 'salade de crevettes grises'
                      plat: 'saumon fume et pommes de terre'
                      prix: 12)
                 menu(plat: 'choucroute garnie'
                      prix: 9))
   ```

   En utilisant la variable `Carte`, répondez aux questions suivantes.

   - Quel est le second menu ? A quel champ de `Carte` correspond-il ?
   - Quel est le plat du second menu ? Quelle est l'entrée du premier menu ? De quel type sont ces valeurs ?
   - Supposons qu'un groupe de personnes se rendent dans ce restaurant, que `N1` personnes commandent le premier menu, `N2` personnes le second et `N3` personnes le troisième. Quel prix vont-ils payer au total ?
   - Quelle est l'arité de la carte ? Quelle est celle de chacun des menus ?

10. **Promenade arboricole.** Considérez la grammaire suivante. Elle définit un type pour des arbres binaires dont les nœuds contiennent une valeur de type `T`. Un arbre binaire est soit un arbre vide, soit un nœud contenant une valeur et deux sous-arbres.

```
<btree T> ::= empty | btree(T left:<btree T> right:<btree T>)
```

- Définissez une fonction `Promenade` qui réalise un parcours *en profondeur d'abord* d'un arbre. Dans ce type de parcours, le sous-arbre gauche d'un noeud est parcouru entièrement avant son sous-arbre droit. L'appel `{Promenade BT}` renvoie la liste des valeurs des noeuds visités dans l'arbre `BT`. Le noeud racine d'un arbre est toujours visité avant ses sous-arbres.

```
%% affiche [42 26 54 18 37 11]
{Browse
 {Promenade
     btree(42
           left: btree(26
                       left: btree(54
                                   left: empty
                                   right: btree(18
                                                left: empty
                                                right: empty))
                       right: empty)
           right: btree(37
                        left: btree(11
                                    left: empty
                                    right: empty)
                        right: empty))}}
```

- Utilisez cette fonction pour calculer la somme des valeurs d'un arbre binaire contenant des entiers. Utilisez *la fonction* ***FoldL*** (http://mozart2.org/mozart-v1/doc-1.4.0/base/list.html) pour écrire cette fonction.
- Définissez une fonction qui calcule la somme des valeurs d'un arbre en effectuant elle-même un parcours de l'arbre.

11. **Dictionnaires** Nous allons implémenter un dictionnaire en utilisant des arbres binaires ordonnés.

    Un dictionnaire vide est représenté avec l'atome `leaf`.

    Un dictionnaire non vide ressemble à : `dict(key:Key info:Info left:Left right:Right)`

    Écrivez une fonction `{DictionaryFilter D F}` qui, selon un dictionnaire D et une fonction booléenne F, retourne une liste de tuple `Key#Info` tel que `{F Info}` vaut `True`. Le code ci-dessous montre un exemple d'utilisation de cette fonction.

```
local Old Class Val in
  Class = dict(key:10
               info:person('Christian' 19)
               left:dict(key:7
                         info:person('Denys' 25)
                         left:leaf
                         right:dict(key:9
                                    info:person('David' 7)
                                    left:leaf
                                    right:leaf))
               right:dict(key:18
                          info:person('Rose' 12)
                          left:dict(key:14
                                    info:person('Ann' 27)
                                    left:leaf
                                    right:leaf)
                          right:leaf))
  fun {Old Info}
    Info.2 > 20
  end

  Val = {DictionaryFilter Class Info Old}
  % Val --> [7#person('Denys' 25) 14#person('Ann' 27)]
end
```

12. **Listes, tuples, enregistrements.** Voici un ensemble d'enregistrements. Pour chacun, indiquez s'il s'agit d'une liste, d'un tuple ou d'un enregistrement. Étant donné qu'une liste est un cas particulier de tuple, et qu'un tuple est un cas particulier d'enregistrement, donnez le type le plus précis.

'|'(a b)	'|'(a '|'(b nil))	'|'(2:nil a)
state(1 a 2)	state(1 3:2 2:a)	tree(v:a T1 T2)
a#b#c	[a b c]	m|n|o

Vous pouvez vérifier vos réponses en utilisant les fonctions `IsList` et `IsTuple`. `{IsList X}` renvoie `true` si `X` est une liste, `false` sinon. `IsTuple` fonctionne de manière similaire avec les tuples.

13. **Il faut s'appliquer.**

    - Écrivez une fonction `{Applique L F}` qui applique la fonction `F` sur chaque élément de la liste `L` et retourne la liste des résultats.

      ```
      declare
      fun {Lol X} lol(X) end
      {Browse {Applique [1 2 3] Lol}} % Affiche [lol(1) lol(2) lol(3)]
      ```

    - Écrivez une fonction `MakeAdder` qui prend en argument un entier `N` et renvoie une fonction qui ajoute `N` à tous les entiers qu'elle reçoit en argument.

      ```
      declare
      Add5 = {MakeAdder 5}
      {Browse {Add5 13}}      % Affiche 18
      ```

    - Utilisez `Applique` et `MakeAdder` pour créer une fonction `{AddAll L N}` qui ajoute `N` à tous les éléments d'une liste `L`.

    - *Note:* La fonction `Applique` existe déjà dans l'environnement de base de Mozart et s'appelle `Map` ou `List.map`

14. **Un tuple étrange.** L'opérateur `#` est un peu particulier. Si son proche cousin `|` permet de créer des listes, l'opérateur `#` permet de créer des tuples.

    `m#i#6 = '#'(m i 6) = '#'(1:m 2:i 3:6)`

    - Déterminez ce qu'affiche le code suivant

      ```
      {Browse {Label a#b#c}}
      {Browse {Width un#tres#long#tuple#tres#tres#long}}
      {Browse {Arity 1#4#16}}
      ```

    - La fonction suivante vérifie que deux listes ont bien la même taille. Maintenant que vous connaissez l'opérateur `#`, expliquez son fonctionnement.

      ```
      fun {SameLength Xs Ys}
          case Xs#Ys
          of nil#nil then true
          [] (X|Xr)#(Y|Yr) then {SameLength Xr Yr}
          else false
          end
      end
      ```