LINFO1104 Séance 2 *Extra*

Programmation symbolique et enregistrements

Programmation avec les listes.

1. **Mettons les listes à plat.** Écrivez une fonction `Flatten` prenant une liste `L` en paramètre. Les éléments de `L` peuvent eux-même être des listes, dont les éléments peuvent aussi être des listes, et ainsi de suite. L'appel `{Flatten L}` renvoie la liste "aplatie", c'est-à-dire les éléments de `L` et des listes contenues dans `L` qui ne sont pas des listes. Exemple:

   ```
   {Browse {Flatten [a [b [c d]] e [[[f]]]]}}   % affiche [a b c d e f]
   ```

2. **Représentation binaire d'entiers avec des listes.**Écrivez une fonction `Add` telle que `{Add B1 B2}` renvoie le résultat de l'addition des nombres binaires `B1` et `B2`. Un nombre binaire est représenté par une liste de chiffres binaires (`0` ou `1`). Le chiffre le plus significatif est le premier élément de la liste. Par exemple,

   ```
   {Browse {Add [1 1 0 1 1 0] [0 1 0 1 1 1]}}   % affiche [1 0 0 1 1 0 1]
   ```

   Quelques remarques:

   - Vous pouvez supposer que les deux listes passées en argument ont la même taille.
   - Vous devriez commencer par écrire un additionneur de trois chiffres `{AddDigits D1 D2 CI}`, qui prend trois chiffres binaires `D1`, `D2` et `CI`, et renvoie une liste de deux chiffres binaires représentant leur somme. Le bit `CI` et le premier chiffre du résultat est un chiffre de report. Exemples:

     ```
     {Browse {AddDigits 1 0 0}}   % affiche [0 1]
     {Browse {AddDigits 1 1 0}}   % affiche [1 0]
     {Browse {AddDigits 1 0 1}}   % affiche [1 0]
     {Browse {AddDigits 1 1 1}}   % affiche [1 1]
     ```

3. **Traiter une liste.** On peut calculer la moyenne des éléments d'une liste en calculant simultanément la somme et la longueur puis en divisant.

   ```
   declare
   fun{Average L}
      fun{AverageAcc L S N}
         case L
         of nil then S / N
         [] H|T then {AverageAcc T H+S N+1.}
         end
      end
   in
      {AverageAcc L 0. 0.}
   end

   {Browse {Average [42. 17. 25. 61. 9.]}}
   ```

   En partant de l'exemple, créez une fonction qui calcule la variance (σ^2) des valeurs d'une liste (de plus de deux éléments) Utilisez la formule: $\sigma^2 = (\sum(x_i^2))/n - ((\sum x_i)/n)^2$

4. **Nombres factoriels.** Écrivez une fonction `Fact` telle que `{Fact N}` renvoie la liste des `N` premiers nombres factoriels dans l'ordre croissant. Par exemple,

   ```
   {Browse {Fact 4}}   % affiche [1 2 6 24]
   ```

Programmation sur les enregistrements.

Rappelons qu'un enregistrement est composé d'une *étiquette* (`label`) (atome) et d'un certain nombre de *champs* (`X1`, `Xn`). Chaque champ est identifié par un nom (atome ou nombre) (`F1`, `Fn`).

Opération	Description	$\Theta(\ )$
`label(F1:X1 ... Fn:Xn)`	Crée un enregistrement	$\Theta(n)$
`R.N`	Renvoie le champ `N` de l'enregistrement `R`	$\Theta(1)$
`{Label R}`	Renvoie l'étiquette de l'enregistrement `R`	$\Theta(1)$
`{Width T}`	Renvoie le nombre de champs de l'enregistrement `R`	$\Theta(1)$
`{Arity R}`	Renvoie la liste des noms de champs de l'enregistrement `R`	$\Theta(1)$

1. **Recoller les morceaux** Un virus malicieux s'est introduit dans votre ordinateur et a décomposé tous les nombre en une liste de chiffres.

   - Écrivez une fonction `Recompose` qui reconstruit un nombre à partir de la liste de ses chiffres.

     ```
     {Browse {Recompose [1 3 3]}} % Affiche 133
     ```

   - Écrivez une fonction `{AppendDigit Num Digit}` qui ajoute un chiffre à la fin du nombre `Num`.

     ```
     {Browse {AppendDigit 1234 5}} % Affiche 12345
     ```

   - Mozart fournit de nombreuses fonctions pour manipuler les listes. Toutes ces fonctions sont documentées en ligne, sur https://mozart.github.io/mozart-v1/doc-1.4.0/base/list.html. Par exemple, la fonction `FoldL` est définie comme suit :

     FoldL:

     ```
     {List.foldL +Xs +P X ?Y}
     ```

     Used for folding the elements of Xs by applying a ternary procedure P.

     Application of the left folding procedure `{FoldL [X1 ... Xn] P Z Out}` reduces to `{P ... {P {P Z X1} X2} ... Xn Out}`

     The first actual argument of P is the accumulator in which the result of the previous application or the start value Z is passed. The second actual argument is an element of Xs.

     For example, `{FoldL [b c d] fun {$ X Y} f(X Y) end a}` returns `f(f(f(a b) c) d)`.

     Ré-écrivez la fonction `Recompose` à l'aide de `FoldL` et de `AppendDigit`.

   - Admirez le travail !

2. **Fibonacci generator.** Ecrivez une fonction `FiFib` qui renvoie un tuple `fib(1 F1)` tel que

   - `{F1}` renvoie un tuple `fib(1 F2)` tel que ...
   - `{F2}` renvoie un tuple `fib(2 F3)` tel que ...
   - `{F3}` renvoie un tuple `fib(3 F4)` tel que ...
   - `{F4}` renvoie un tuple `fib(5 F5)` tel que ...
   - `{F5}` renvoie un tuple `fib(8 F6)` tel que ...
   - etc.

   Autrement dit, les premiers éléments des tuples forment la suite de fibonacci, tandis que le second élément d'un tuple est une fonction qui permet de générer le tuple suivant de la séquence.

3. **Guinness, the record maker.**

   Comme vous le savez surement, la fonction `{Guinness Label Features Values}` permet de créer des enregistrements dont le label est `Label`, les noms de champs sont dans la liste `Features` et les valeurs associées sont stockées dans la liste `Values`.

   - Renseignez vous sur la fonction `List.toRecord`, qui crée des record à partir d'un label et d'une liste.
   - Renseignez vous sur la fonction de niveau supérieur `Zip` qui combine les éléments de deux liste en leur appliquant deux à deux une une fonction binaire passée en argument.
   - Vous avez maintenant tous les outils nécessaires pour écrire la fonction `Guinness` de manière concise et sans utiliser `MakeRecord`

Programmation sur les arbres.

1. **Trier une liste avec des arbres.** Dans cet exercice, vous allez implémenter une méthode de tri qui utilise des *arbres ordonnés*. Nous avons vu au cours une implémentation d'arbres ordonnés. Pour trier une liste, on construit un arbre ordonné qui contient toutes les valeurs de la liste, puis les valeurs sont lues dans l'arbre dans l'ordre croissant. On fait l'hypothèse que les éléments de la liste sont distincts.

   - Pour construire l'arbre ordonné, définissez une fonction `InsertElements` qui prend en argument une liste de valeurs et un arbre ordonné, et renvoie l'arbre dans lequel on a inséré les éléments de la liste. Chaque élément de la liste est à la fois clé et valeur dans l'arbre. L'appel

     ```
     {InsertElements [42 17 23] tree(key:31 value:31 leaf leaf)}
     ```

     renvoie l'arbre

     ```
     tree(key:31 value:31
          tree(key:17 value:17
               leaf
               tree(key:23 value:23
                    leaf
                    leaf))
          tree(key:42 value:42
               leaf
               leaf))
     ```

   - Donnez la complexité temporelle de la fonction `InsertElements`. Décrivez ensuite les meilleur et pire cas, et donnes leur complexité.

   - Écrivez une fonction `Keys` qui prend un arbre ordonné et renvoie la liste de ses clés en ordre croissant. Si `T` est l'arbre renvoyé ci-dessus, alors `{Keys T}` renvoie la liste `[17 23 31 42]`.

   - Définissez maintenant la fonction `OrderedTreeSort`, qui prend une liste en argument et renvoie la liste triée. Cette fonction utilise bien sûr les deux précédentes.