# LANGAGES DE PROGRAMMATION
# IFT-3000
## HIVER 2018

**Travail pratique 1 - TRAVAIL INDIVIDUEL**
Pondération de la note finale : 15%

À remettre, par le biais de Pixel, et avant 23h59, le dimanche 25 février 2018

## 1 Énoncé

Ce travail pratique consiste à compléter un module permettant de manipuler des arbres de preuves génériques et de les afficher correctement à l'écran ; 2 types d'affichage sont considérés dans le code qui vous est fourni : (i) l'affichage texte qui considère un arbre comme plusieurs sous-arbres, et les affiche un à la suite de l'autre en utilisant des références pour les lier les uns aux autres ; (ii) l'affichage en format TEX, dans un navigateur, grâce au module Javascript *MathJax*. Notez qu'il est possible de viser son propre format d'affichage (en format *MathML*, par exemple), le code étant assez générique pour supporter cette option. Aussi, en plus d'arbres de preuves, les structures de données utilisées sont assez génériques pour pouvoir représenter d'autres types d'arbres.

Voici un exemple d'arbre de preuve, affichés en utilisant une macro LATEX :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\square}{x_1 = x_2 \wedge x_1 = x_3}(u)}{x_1 = x_2}(et)}{x_2 = x_1}(sym) \quad \dfrac{\dfrac{\square}{x_1 = x_2 \wedge x_1 = x_3}(u)}{x_1 = x_3}(et)}{x_2 = x_3}(tr)}{x_1 = x_2 \wedge x_1 = x_3 \Rightarrow x_2 = x_3}(imp)$$

Cette arbre de preuve pourrait être affiché en utilisant un ou plusieurs sous-arbres ; dans l'exemple qui suit, on décide d'y aller avec un seul sous-arbre ($A_2$) :

$$A_1 : \quad \dfrac{A_2}{x_1 = x_2 \wedge x_1 = x_3 \Rightarrow x_2 = x_3}(imp)$$

$$A_2 : \quad \dfrac{\dfrac{\dfrac{\dfrac{\square}{x_1 = x_2 \wedge x_1 = x_3}(u)}{x_1 = x_2}(et)}{x_2 = x_1}(sym) \quad \dfrac{\dfrac{\square}{x_1 = x_2 \wedge x_1 = x_3}(u)}{x_1 = x_3}(et)}{x_2 = x_3}(tr)$$

Un des objectifs du travail pratique est de pouvoir générer un tel type d'affichage en offrant à l'utilisateur la possibilité de préciser à partir de quel hauteur d'arbre l'algorithme devra y aller par un affichage en sous-arbres en les numérotant automatiquement pour qu'ils puissent être liés les uns aux autres et ainsi représenter un seul grand arbre (en me référant à l'exemple précédent, les identifiants des arbres $A_1$ et $A_2$, ont été automatiquement générés par l'algorithme).

Évidemment, en y allant avec un affichage en format texte, dans un terminal, il est difficile d'afficher un arbre au complet ; dans ce travail, pour l'affichage en format texte, on ira donc avec un affichage en sous-arbres de hauteur 1 ; l'exemple précédent serait alors affiché comme suit :

```
# gen a;;
A1:
                A2
        ------------------------------- (imp)
        x1 = x2 ET x1 = x3 => x2 = x3

A2:
           A3   A6
        ------------- (tr)
           x2 = x3

A3:
           A4
        --------- (sym)
        x2 = x1

A4:
           A5
        --------- (et)
        x1 = x2

A5:

        ---------------------- (u)
        x1 = x2 ET x1 = x3

A6:
           A7
        --------- (et)
        x1 = x3

A7:

        -------------------- (u)
        x1 = x2 ET x1 = x3
```

# 2 Signatures

La signature des deux principales structures de données et des principales fonctions du module PTree sont :

```
module type PTREE =
  sig
    exception Non_Implante of string

    type ('formula, 'rule) pTree = PF of 'formula
                                 | PT of 'rule * 'formula * ('formula, 'rule) pTree list
    type strTree = St of int | Leaf of string | Tree of string * string * strTree list

    val includeSep : string -> string list -> string
    val height : strTree -> int
    val ptree2stree : ('formula -> string) -> ('rule -> string) ->
                      ('formula, 'rule) pTree -> strTree
    val tree2mtree : ?l:int -> strTree -> (int * strTree) list
    val mtree2pretty : (strTree -> string) -> (int -> string) -> string ->
                       (int * strTree) list -> string list

    val id2str : int -> string
    val tree2str : strTree -> string
    val id2latex : int -> string
    val tree2latex : strTree -> string

    val genstree : ?tree2str:(strTree -> string) -> ?id2str:(int -> string) -> strTree -> unit
    val genltree : ?l:int -> ?tree2latex:(strTree -> string) -> ?id2latex:(int -> string)
                   -> strTree -> int

end
```

Elle précise que deux structures de données sont à considérer :

1. le type *pTree*, paramétré par deux variables de type [1] (*'formula* et *'rule*) ; ce type permet de considérer des arbres n-aires quelconques qui comprennent au niveau des feuilles des valeurs d'un certain type (*'formula*), et au niveau des nœuds deux valeurs de types respectifs *'rule* et *'formula*, ainsi qu'une liste de sous-arbres de type *pTree* ; ainsi, avec ce type de données, il sera possible de représenter n'importe quel type d'arbres de preuves ayant, à chaque niveau, une conclusion et un ensemble de prémisses, ainsi qu'un numéro de règle associé ;
2. le type *strTree* s'apparente au type *pTree* mais comprend un constructeur de plus, *St*, qui permet d'avoir, au niveau des feuilles, des entiers référençant d'autres (sous-)arbres ; c'est à travers ce type de structures de données qu'il sera possible de générer, pour un arbre donné, un ensemble de sous-arbres le représentant.

Étant donnés ces deux types, sont précisées aussi les signatures des 5 fonctions que vous devez compléter :

1. «includeSep» : fonction assez basique qui consiste, à partir d'une liste de chaines de caractères, et d'une chaine de caractères représentant un séparateur, de retourner une unique chaine de caractères comprenant les éléments de la liste séparés par le séparateur ;
2. «height» : comme son type l'indique, cette fonction retourne la taille d'un arbre de type *strTree* ;
3. «ptree2stree» : cette fonction transforme un arbre de type *pTree* en un arbre de type *strTree* ; deux fonctions sont en paramètres pour respectivement préciser comment un élément de type *'formula* et un élément de type *'rule* doivent être traduits en éléments de type *string* ;
4. «tree2mtree» : cette fonction transforme un arbre de type *strTree* en une liste de sous-arbres de type *strTree* aussi ; chaque sous-arbre est accompagné par un entier qui permet de l'identifier (dans les autres sous-arbres) ; aussi, la fonction comprend un paramètre optionnel, fixé par défaut à zéro, permettant de préciser à partir de quel hauteur d'un sous-arbre considéré dans l'arbre de départ, il sera nécessaire de le détacher pour en faire un sous-arbre à part (lié, tout de même, à l'arbre dont il était rattaché par le numéro qui l'identifie) ?
5. «mtree2pretty» : cette fonction prend typiquement le résultat de la fonction précédente, soit «tree2mtree», et retourne une chaine de caractères qui comprend l'information nécessaire pour afficher tous les sous-arbres, et leur identifiant ; cette fonction prend deux fonctions en argument permettant de préciser comment un sous-arbre doit être affiché (représenté en chaine de caractères) et comment le nom d'un sous-arbre devrait être affiché.

De plus, les signatures d'autres fonctions utilitaires dont le code est fourni, sont précisées :

— «genstree» : fonction qui prend un arbre de type *strTree*, le transforme en un ensemble de sous-arbres, et les affiche en format texte à l'écran (dans le terminal) ; cette fonction prend deux autres fonctions optionnels en argument, qui permettent respectivement de préciser comment un sous-arbre de type *strTree* doit être affiché, ainsi que le numéro l'identifiant ; ces deux arguments-fonctions sont par défaut fixés à deux fonctions qui vous sont fournies : «tree2str» et «id2str» ;
— «genltree» : cette fonction est semblable à la précédente, à la différence qu'elle vise l'affichage d'un arbre dans un navigateur, en profitant du module Javascript *MathJax*, et qu'elle comprend un argument optionnel permettant à son utilisateur de préciser la hauteur à partir de laquelle il faudra considérer des sous-arbres.

# 3 Implantation

Dans ce TP, il s'agit de programmer des fonctions du module PTree en utilisant le paradigme fonctionnel. Voici quelques indications au niveau du corrigé :

— Toutes les fonctions à compléter dans ce travail sont indépendantes les unes des autres, hormis «height» utilisée par la fonction «tree2mtree».
— La fonction prédéfinie «fold_left» est pratiquement utilisée dans toutes les fonctions que vous devez compléter.
— Toutes les fonctions requièrent entre 1 et 5 lignes de code, hormis «tree2mtree» qui en requiert un peu plus de 10.
— Aucune fonction n'a de cas d'exception à traiter ; les seules exceptions soulevées, présentes dans le code, se trouvent au niveau de la fonction utilitaire «read_lines_file» pour souligner qu'un fichier est absent ; et au niveau de la fonction «tree2str» pour préciser qu'on ne peut traiter, pour un affichage en mode texte, un arbre de taille plus grande que 1, ou autre chose qu'un arbre.

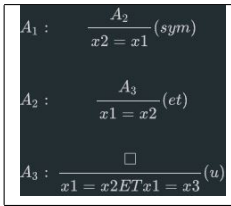
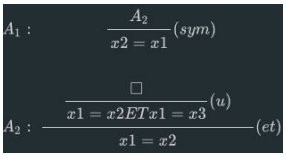
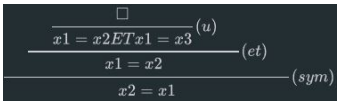
Une fois les fonctions complétées, et le module chargé dans l'interpréteur, il pourra être utilisé comme suit :

1. Il aurait été possible de nommer ces variables de type *'a* et *'b* ; l'idée est simplement de préciser davantage à quoi correspond chacune de ces deux variables de types.

```
# open PTree;;

# let a = PT("(sym)","x2 = x1",[PT("(et)","x1 = x2",[PT("(u)","x1 = x2 ET x1 = x3",[])])]);;
val a : (string, string) PTree.pTree =
PT ("(sym)", "x2 = x1", [PT ("(et)", "x1 = x2", [PT ("(u)", "x1 = x2 ET x1 = x3", [])])])

# let t = ptree2stree (fun x -> x) (fun x -> x) a ;;
val t : PTree.strTree = Tree ("(sym)", "x2 = x1",
[Tree ("(et)", "x1 = x2", [Tree ("(u)", "x1 = x2 ET x1 = x3", [])])])

# let lt0 = tree2mtree t ;;
val lt0 : (int * PTree.strTree) list = [(1, Tree ("(sym)", "x2 = x1", [St 2]));
                                        (2, Tree ("(et)", "x1 = x2", [St 3]));
                                        (3, Tree ("(u)", "x1 = x2 ET x1 = x3", []))]

# let lt1 = tree2mtree ~l:1 t ;;
val lt1 : (int * PTree.strTree) list =
[(1, Tree ("(sym)", "x2 = x1", [St 2]));
 (2, Tree ("(et)", "x1 = x2", [Tree ("(u)", "x1 = x2 ET x1 = x3", [])]))]

# let lt2 = tree2mtree ~l:2 t;;
val lt2 : (int * PTree.strTree) list =
[(1,Tree ("(sym)", "x2 = x1",
          [Tree ("(et)", "x1 = x2", [Tree ("(u)", "x1 = x2 ET x1 = x3", [])])]))]

# mtree2pretty tree2str id2str "\n" lt0 ;;
- : string list =
["A1:"; "\t   A2   \n\t--------- (sym)\n\tx2 = x1\n"; "A2:";
 "\t   A3   \n\t--------- (et)\n\tx1 = x2\n"; "A3:";
 "\t   \n\t------------------- (u)\n\tx1 = x2 ET x1 = x3\n"]

# List.iter print_endline (mtree2pretty tree2str id2str "\n" lt0);;
A1:
          A2
        --------- (sym)
        x2 = x1

A2:
          A3
        --------- (et)
        x1 = x2

A3:

        ------------------- (u)
        x1 = x2 ET x1 = x3

- : unit = ()
# mtree2pretty tree2str id2str "\n" lt1;;
Exception: Failure "Impossible de convertir!".

# mtree2pretty tree2str id2str "\n" lt2;;
Exception: Failure "Impossible de convertir!".

# mtree2pretty tree2latex id2latex "\\\\\\\\" lt1;;
- : string list =
["A_{1}: \\ & "; "\\cfrac{~~~~~A_{2}~~~~~}{x2 = x1}(sym)\\\\\\\\";
 "A_{2}: \\ & ";
 "\\cfrac{~~~~~\\cfrac{\\Box}{x1 = x2 ET x1 = x3}(u)~~~~~}{x1 = x2}(et)\\\\\\\\"]

# mtree2pretty tree2latex id2latex "\\\\\\\\" lt2;;
- : string list =
["\\cfrac{~~~~~\\cfrac{~~~~~\\cfrac{\\Box}{x1 = x2 ET x1 = x3}(u)~~~~~}
                                    {x1 = x2}(et)~~~~~}{x2 = x1}(sym)"]

# genltree t;;
>>>- : int = 0  (* Voir résultat dans copie d'écran 1 qui suit *)
# genltree ~l:1 t;;
>>>- : int = 0  (* Voir résultat dans copie d'écran 2 qui suit *)
# genltree ~l:2 t;;
>>>- : int = 0  (* Voir résultat dans copie d'écran 3 qui suit *)
```

```
# height t ;;
- : int = 3
# height (Leaf "a");;
- : int = 0
# height (St 1);;
- : int = 0
# height (Tree("a", "b", []));;
- : int = 1

# includeSep "-" [];;
- : string = "-"
# includeSep "-" ["a"];;
- : string = "-a-"
# includeSep "-" ["a"; "b"; "c"];;
- : string = "-a-b-c-"
```

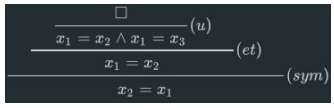
Voici les 3 copies d'écran :

| Copie d'écran 1 | Copie d'écran 2 | Copie d'écran 3 |
|---|---|---|
|  |  |  |

L'affichage n'est pas optimal puisque le texte saisi, pour l'exemple d'arbre de preuve, avait été prévu pour un affichage texte ; en visant un affichage en LaTeX, on aurait alors considérer les choses comme suit :

```
# let a'=PT("(sym)","x_2=x_1",[PT("(et)","x_1=x_2",[PT("(u)","x_1=x_2␣\\wedge␣x_1=x_3",[])])]);;
val a' : (string, string) PTree.pTree =
PT ("(sym)", "x_2=x_1",[PT ("(et)", "x_1=x_2",[PT ("(u)", "x_1=x_2 \\wedge x_1=x_3",[])])])

# genltree ~l:2 (ptree2stree (fun x -> x) (fun x -> x) a');;
- : int = 0
```

Ce qui aurait produit un affichage bien plus adapté (Le symbole _ permettant de mettre en indice les chiffres associés aux lettres ; la commande \wedge permettant d'afficher un beau symbole de conjonction : $\wedge$) :



Quelques précisons au niveau des exemples présentés dans la page précédente :

- Au niveau de l'arbre créé «a», comme tous ses éléments sont de type *string*, les 2 fonctions passées en argument de «ptree2stree» correspondent à la fonction identité ; aucun traitement n'est nécessaire, pour cet exemple d'arbre, pour transformer ses éléments en *string*.
- Au niveau de la fonction «tree2mtree», on aperçoit des résultats différents en fonction de la valeur de son argument optionnel «l».
- La fonction «mtree2pretty» peut s'appliquer sur la liste de sous arbres «lt0» ; par contre, lorsqu'on applique cette fonction aux listes «lt1» ou «lt2», due à la fonction «tree2str» qui lui est passée en argument, une exception est soulevée (par «tree2str») car cette fonction ne peut afficher des arbres de hauteur supérieur à 1.
  Par contre, cette fonction, lorsqu'utilisée avec la fonction «tree2latex», peut traiter n'importe quel type d'arbres (de n'importe quel hauteur).

**Exploitation du TP** Le fichier «test.ml» comprend la définition d'un module permettant d'exploiter les principales fonctionnalités du Tp. Pour utiliser ce module, une fois que votre Tp sera complété, il faut simplement charger le fichier, et faire appel aux 2 principales fonctions qui y sont définies ; à la fin du fichier, en commentaires, figurent quelques exemples d'utilisation des 2 fonctions (il suffit d'y aller par un copier-coller).

# 4 Démarche à suivre

Tel que le fichier "ptree.ml" a été conçu (utilisation d'exceptions pour les fonctions non implantées), vous pouvez charger le fichier :

```
# #use "ptree.ml";;
   ...
module PTree : PTREE
```

Il faut donc compléter une à une les fonctions à implanter et les tester progressivement ; évidemment, lorsque vous complétez une fonction, vous devez mettre en commentaire (ou supprimer la ligne) la ligne «raise (Non_Implante "«nom_fonction» à compléter")». Vous pouvez à tout moment tester les fonctions de votre Tp à l'aide du fichier testeur.ml :

```
# #use "testeur.ml";; (* on suppose que Ptree est chargé *)
module L = List
...

# test1();; (* pour tester la première fonction *)
- : float * string list * bool = (8., [], false)
(* ça précise qu'on a 8 pts (sur 8); false indique qu'il n'y a pas eu d'exceptions soulevées;
   la liste, qui est vide pour le corrigé, indique éventuellement là où votre code n'a pas
   retourné les valeurs prévues *)

# test();;  (* pour tester toutes les fonctions *)
- : (string * float * float * string list * bool) list * bool =
([("includeSep", 8., 8., [], false); ("height", 12., 12., [], false);
("ptree2stree", 20., 20., [], false); ("tree2mtree", 40., 40., [], false);
("mtree2pretty", 20., 20., [], false)],
false)
```

# 5 À remettre

Il faut remettre un fichier .zip contenant uniquement le fichier "ptree.ml" complété. Le code doit être clair et bien structuré [2].

# 6 Remarques importantes

**Plagiat :** Tel que décrit dans le plan de cours, le plagiat est interdit. Une politique stricte de tolérance zéro est appliquée en tout temps et sous toutes circonstances. Tous les cas seront référés à la direction de la Faculté.

**Travail remis en retard :** Tout travail remis en retard se verra pénalisé de 25% par jour de retard. Chaque journée de retard débute dès la limite de remise dépassée (dès la première minute). Un retard excédant 2 jours (48h) provoquera le rejet du travail pour la correction et la note de 0 pour ce travail. La remise doit se faire par la boite de dépôt du TP1 dans la section «Évaluation et résultats».

**Barème :** Au niveau des fonctions à implanter, le barème est précisé dans le fichier "ptree.ml". Notez cependant que :

— (-10pts), si votre code ne compile pas (provoque une erreur/exception lors du chargement du fichier "ptree.ml" dans l'interpréteur, i.e. «**#use** "ptree.ml" ; ;»).

Bon travail.

2. Suggestion : http://ocaml.org/learn/tutorials/guidelines.html