Travaux Pratiques ML No.4

Cours d'Informatique de Deuxième Année

—Deug SSM M4—

Aujourd'hui...

L'organisation de ce TP est la suivante:

— La définition de types.

— Un exemple de type polymorphe : le type `'a liste`.

— Fonctionnelles d'itération.

Définition de types

Il est possible de définir un certain nombre de types que l'on peut ranger en deux catégories (les types *sommes* dits types *énumérés* et les types *produits* dits types *enregistrements*) :

Les enregistrements. Un type enregistrement est un type dont la structure est analogue à celle d'un `record` en *Pascal*. Par exemple, on peut définir un type enregistrement de renseignements concernant une personne de la façon suivante:

```
#type person = {
#        name :  string ;
#        mutable age :  int ;
#        mutable activity :  string ;
#        mutable Where :  string
#        };;
```

Les champs précédés de `mutable` sont modifiables, alors que les autres ne le sont pas (une personne possède à priori un nom de façon durable!). On peut alors déclarer par exemple `Tarzan` de type `person` de la manière suivante:

```
#let Tarzan = {
#        name="Tarzan";
#        age=20;
#        activity="savage";
#        Where="jungle"
#        };;
```

puis augmenter son âge:

```
#Tarzan.age <- Tarzan.age + 1 ;;
```

ou encore;

```
#let get_older ({age=n; _} as pers) = pers.age <- n+1 ;;
#get_older Tarzan ;;
```

▸ **Exercice 2**. Exprimez le fait que `Tarzan` passe de l'état sauvage à celui de civilisé en arrivant un an après à Greystoke...

. . . . . .

. . . . . .

Les types énumérés. Un type énuméré est, comme son nom l'indique, une énumération de différents constructeurs de types. Exemple:

```
#type entier =
#          Z (* Zero :  constructeur d'arite nulle *)
#        | S of entier (* Successeur :  constructeur d'arite 1 *) ;;
```

Un tel type est dit *récursif* et on verra qu'il est intéressant de définir une fonctionnelle d'itération sur tous les types récursifs – c'est-à-dire une fonction qui prend en arguments une fonction f qui traite la partie récursive du type et une fonction g (souvent constante) qui traite le cas de la partie constante du type. Par exemple, la fonction `entier_it` suivante permet de traiter les `entiers` de façon simplifiée[1].

```
# let entier_it f base = ent_rec where rec ent_rec = function
#          Z -> base
#       | S x -> f (ent_rec x) ;;
```

▸ **Exercice 3**. Définir les fonctions `addition`, `soustraction`, `multiplication` et `division` travaillant sur le type `entier`.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

▸ **Exercice 4**. Définir les fonctions `addition`, `soustraction`, `multiplication` et `puissance` travaillant sur le type `entier` en utilisant `entier_it`.

. . . . . .

. . . . . .

. . . . . .

On assimile les booléens à des entiers de la manière suivante: `True=S Z` et `False=Z` (en fait, tout résultat supérieur à `Z` pourra être assimilé à `True`). Ceci va nous permettre de réaliser un certain nombre de tests intéressants.

▸ **Exercice 5**. Définir les fonctions `isnull`, `Not`, *`iseven` et `If` à partir de `entier_it` de telle sorte que `isnull Z` = `True` et `isnull _` = `False`; `Not True` = `False` et `Not False` = `True`; `iseven` teste les entiers pairs; `If cond x y` retourne `x` si `cond` vaut `True`, et `y` sinon.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

[1] On peut faire l'analogie avec la fonctionnelle `list_it`, les listes étant des objets qu'on a l'habitude de traiter récursivement.

▸ **Exercice 6**. Définir à partir de ce qui a été construit jusque là les fonctions booléennes suivantes: `isgreater-than` qui teste si un entier est strictement plus grand qu'un autre; `And` qui teste si les deux arguments sont positionnés à `True`; `Or` qui teste si au moins un des deux vaut `True`; `equals` qui teste si les deux arguments ont la même valeur, en tant qu'entiers.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

▸ **Exercice 7**. A votre avis, quelles sont les modifications qu'il faut apporter à `entier_it` pour pouvoir écrire la fonction factorielle de manière simple?

. . . . . .

. . . . . .

. . . . . .

Un exemple de type polymorphe

▸ **Exercice 8**. Réécrire le type `'a liste` polymorphe au moyen de constructeurs de types.

. . . . . .

▸ **Exercice 9**. Ecrire une fonction `List_it` sur le type `'a liste`. En déduire la fonction `Map` correspondante.

. . . . . .

. . . . . .

On se propose de programmer le tri par insertion, sur les listes de type `'a liste`.

▸ **Exercice 10**. Ecrire une fonction d'insertion d'un élément dans une liste[(2)].

. . . . . .

. . . . . .

▸ **Exercice 11**. En déduire la fonction de tri par insertion.

. . . . . .

Exercice supplémentaire

▸ **Exercice 12**. Refaire le tri par fusion sur les listes de type `'a liste`.

(2) de type `'a liste`.