TP de programmation n° 2

Julien Reichert

Lundi 21 octobre 2013

Ce TP, qui s'étalera potentiellement sur deux séances, est consacré aux structures de données. Il est fondamental de maîtriser chaînes de caractères, listes et tableaux ; créer des types impressionnera sans doute le jury et utiliser des objets est un avantage bien supérieur [1].

1 Les chaînes de caractères

Après la brève introduction au TP1, traitons les chaînes de caractères plus en détail.

Comme on l'a dit, `str.[i]` est de type `char`. Par défaut, les caractères sont au nombre de 256 (ASCII étendu), leur conversion en entier s'effectue via la fonction `int_of_char` (ou son équivalent `Char.code`), la réciproque étant `char_of_int` (ou `Char.chr`). Le module Char ne contient pas beaucoup d'autres fonctions : `escaped` qui ajoute des \ aux caractères échappés, `uppercase` et `lowercase` qui gère les capitales et `compare` qui retourne la différence entre le code de deux caractères.

Notez que pour ouvrir un module et donc accéder à toutes les fonctions qu'il contient sans les préfixer par les nom dudit module, on utilise `open` suivi du nom du module. Lorsque ce module n'est pas dans la librairie standard, il faut aussi le charger. Par exemple, pour les graphismes, que nous verrons plus tard :

```
#load "graphics.cma";;
open Graphics;;
```

Pour autant, évitez d'ouvrir des modules tels que String, List et Array qui contiennent des fonctions homonymes [2].

1. sous réserve que contexte et temps s'y prêtent
2. dans ce cas, ouvrir un module écrasera les fonctions homonymes des précédents

Le module String est un peu plus fourni. Il contient des fonctions très utiles comme `length`, `make` (longueur en premier argument, caractère de base en second), `copy`, `sub` (chaîne à extraire en premier argument, indice de départ en deuxième et longueur en troisième), `compare` (ordre lexicographique, retourne une valeur entre -1 et 1 alors que les opérateurs tels que $<$ retournent un booléen), ...

Pour modifier un élément d'une chaîne de caractère, il existe une fonction, mais on préfèrera le raccourci suivant :

```
str.[i] <- 'x';;
```

2 Les listes

La liste est peut-être la structure de données dont vous vous servirez le plus en OCaml[3]. Les références de listes ont l'avantage de ne pas avoir de taille fixée, contrairement aux tableaux (qui sont en quelque sorte des références par nature). En fait, il est très facile d'ajouter ou de retirer des éléments à des listes.

La liste vide est notée `[]`, les crochets sont les délimiteurs des listes définies en donnant leurs éléments, séparés par des **points-virgules**[4]. La liste dont le premier élément est `a` et les autres forment la liste `q` est `a::q`. On a en outre `List.hd (a::q) = a` et `List.tl (a::q) = q`.

Notez que tous les éléments d'une liste doivent avoir le même type. C'est cruel, je sais. Le type d'une liste est alors `(type des éléments) list`.

Les opérations basiques sur les listes sont dans le module List, nommons entre autres `length`, `nth`, `rev` pour obtenir le miroir, `append` qui peut être remplacé par l'opérateur infixe `@`, `concat` qui fusionne une liste de listes (toutes de même type), `mem` qui répond si le premier argument est un élément du second et `sort` pour trier suivant la fonction `'a` $\rightarrow$ `'a` $\rightarrow$ `int` en premier argument.

On trouve aussi des itérateurs dont en particulier `iter` qui applique la fonction `'a` $\rightarrow$ `unit` en premier argument à tous les éléments de la `'a list` en second argument, `map` qui applique la fonction `'a` $\rightarrow$ `'b` en premier argument à tous les éléments de la `'a list` en second argument pour former une `'b list`, et `for_all` (resp. `exists`) qui applique la fonction `'a` $\rightarrow$ `bool` en premier argument à tous les éléments de la `'a list` en second argument et retourne la conjonction (resp. disjonction) des résultats.

3. en tout cas, c'est vrai pour moi
4. sinon la liste contient un seul élément, qui est un n-uplet, et lolsplit fait n'importe quoi...

Pour le filtrage, on utilise `find : ('a → bool) -> 'a list -> 'a` (premier élément satisfaisant le prédicat, s'il en existe un, exception sinon), `find_all : ('a → bool) -> 'a list -> 'a list` (liste des éléments qui le satisfont) et `partition : ('a → bool) -> 'a list -> 'a list * 'a list`.

3 Les tableaux

Il n'est pas étonnant que les fonctions sur les tableaux soient plus ou moins similaires aux fonctions sur les chaînes de caractères. Il y a donc `length`, `make`, `copy` et `sub` dans le module Array. Concernant les liens avec les listes, on dispose de `to_list` et `of_list` et on retrouve `iter`, `map` et `sort` qui a cette fois pour type de retour `unit`.

Un tableau peut aussi s'écrire en donnant la liste de ses éléments, avec la syntaxe `[|elt`$_1$`;elt`$_2$`;...;elt`$_n$`|]`.

Pour modifier en place un tableau, on fait par exemple

```
tab.(i) <- x;;
```

mais l'ajout de nouveaux éléments nécessite la création d'un nouveau tableau, via des copies (`copy` ou `blit`), des concaténations de deux tableaux (`append`) ou des fusions de listes de tableaux (`concat`).

4 Les types construits

Les types construits peuvent être de deux genres, appelés habituellement « type somme » et « type produit ».

4.1 Type somme

Un type somme est défini par disjonction, avec de possibles récursions, par exemple un arbre est une feuille ou un nœud dont un ou plusieurs arbres sont des descendants [5].

```
type arbre = Feuille | Noeud of arbre list;;
```

5. avec la définition qui suit, rien n'exclut qu'il n'y ait pas de descendants, certes

Et les types peuvent dépendre d'autres types déjà existants, éventuellement au choix.

```
type 'a arbre = Feuille of 'a | Noeud of 'a * 'a arbre list;;
```

OCaml reconnaîtra alors l'expression suivante comme un `int arbre` :

```
Noeud(3,[Feuille(5);Noeud(6,[Feuille(2);Feuille(1)])]);;
```

Notez que les booléens sont définis comme un type somme, quoique les constructeurs ne prennent pas de majuscule.

4.2 Type produit

Un type produit est défini par une liste d'attributs. Il s'agit en quelque sorte d'une fiche d'identité. En parlant de fiche d'identité, voici un exemple :

```
type humain =
{prenom: string;
mutable nom: string;
annee_naissance: int;
mutable [6] sexe: string;}
;;
```

Les variables d'un type produit se créent ainsi :

```
let moi =
{prenom = "Julien Daniel";
nom = "REICHERT";
annee_naissance = 1988;
sexe = "M";}
;;
```

Et pour changer une valeur mutable :

```
moi.nom <- "Reichert";;
```

Si les attributs sont statiques, on peut aussi y accéder par `variable.attribut`.

6. !!!

5 Les objets

[Cette section est un copier-coller du TP 2 de l'an dernier.]

On peut voir les objets comme des types produits évolués.

Pour définir une classe d'objet, on utilise la syntaxe suivante :

```
class nom_class arg_1 =
object
val mutable x = ...
val y = ...
method get_x = ...
method f arg = ...
end;;
```

Pour créer un object de la classe `nom_class`, on utilise le mot-clef `new`.

```
let a = new nom_class
```

Contrairement aux types produits, les valeurs d'un objet ne sont pas directement accessibles. Il est nécessaire d'implémenter des fonctions donnant l'accès à ces valeurs. Pour appeler ces fonctions, on utilise l'opérateur `#`.

```
let value_x = a#get_x
```

Il est possible pour une classe d'objet d'hériter d'autres classes :

```
class nom_class =
object (self)
inherit autre_class as nom_reference
val mutable x = ...
val y = ...
end;;
```

6 Exercices

Exercice 1 :

Réécrivez vous-mêmes des fonctions des modules List et Array, comme `iter`, `to_list`, etc.

Exercice 2 :

Codez une fonction qui décide si un arbre est valide (pas de liste vide dans les fils d'un nœud), puis s'il s'agit d'un arbre binaire de recherche.

Exercice 3 :

Implémentez l'insertion, la suppression et la modification d'une entrée dans un arbre binaire de recherche, à l'aide des fonctions de l'exercice précédent.

Exercice 4 :

Définissez les nombres complexes comme un type produit et écrivez quelques fonctions dessus, par exemple le calcul du module, de l'argument, etc.