Travaux Pratiques de programmation fonctionnelle n°9

Licence d'informatique

—*Caml*—

▸ Exercice 1. *Filtrage*

Il est possible d'utiliser la contruction `match <...> with <...>` *avec plusieurs expressions simultanément.*

1. *Écrire une fonction* `meme_longueur` *qui teste si deux listes ont la même longueur.*

On cherche à écrire une fonction qui teste si les deux membres d'un couple sont égaux. La solution usuelle consiste à faire un filtrage puis à utiliser une expression conditionnelle `if then else`*. En fait, on peut demander à Caml de ne filtrer que sous certaines conditions. On écrit alors*

<*filtre*> `when` <*condition*> `->` <*resultat*>

2. *Écrire une fonction* `egal_liste` *qui teste l'égalité de deux listes ;*
3. *Écrire une fonction* `member` *qui teste si une valeur apparaît dans une liste ;*
4. *Que signifient les filtres suivants :*
   - `(_::r1, _::r2)`
   - `(a,d as m)::r`

▸ Exercice 2. *Exceptions*

Quand un problème intervient dans un calcul Caml, pour signaler que le calcul ne peut pas se faire, il est possible de renvoyer un résultat particulier : une ***exception****. On déclare une exception avec le mot clé* `exception`*. Le type exception s'appelle* `exn`*. Par exemple*

```
#Division_by_zero;;
- : exn = Division_by_zero
#Failure;;
- : string -> exn = <fun>
```

On lève une exception quelconque avec `raise` *et une* `Failure` *avec* `failwith`*. Le fait qu'une exception soit apparue ne signifie pas que le programme doit être arrêté. Il est ainsi possible de rattraper les exceptions. On utilise la contruction*

`try` <*expression*> `with` <*filtrage des exceptions possibles*>

On peut aussi utiliser les exceptions pour optimiser le code. Lorsqu'on connait le résultat d'un calcul avant qu'il soit terminé, on le stoppe en lançant une exception.

1. *Écrire une fonction qui calcule le produit des éléments d'une liste d'entiers. Lorsque la liste contient l'élément 0, on sait que le produit vaut 0 et on arrête le calcul en lançant une exception.*

▸ Exercice 3. ***Flux syntaxique***

Si elle permet de représenter en interne n'importe quelle expression, la syntaxe Caml n'est pas des plus pratiques pour l'utilisateur : on préfère taper «`A et B`*» que* `Bin(Et,Var "A",Var "B")` *! La syntaxe de la première expression est dite* **concrète** *alors que la deuxième est dite* **abstraite**. *On a donc besoin d'une fonction qui traduit la syntaxe concrète en syntaxe abstraite (analyse syntaxique). Plutôt que des listes on utilise des flux. Les différences entre les flux et les listes sont*

- *l'accès dans les flux est destructif. Cela se traduit par l'utilisation d'une syntaxe de filtrage propre aux flux :* `match ... with parser ...`
- *les éléments d'un flux sont évalués au fur et à mesure (évaluation paresseuse).*

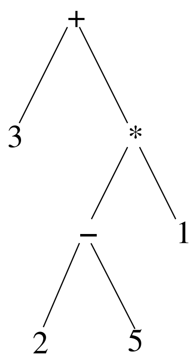
On déclare un flux par `[< 'elem1; 'elem2... >]`*, le flux vide est noté* `[<>]`*. On ne peut pas afficher un flux. Le premier caractère d'un flux est lu avec la fonction* `Stream.next`*, mais le plus souvent on fera appel au filtrage.*

1. *Écrire une fonction* `saute_blancs` *qui supprime les blancs en tête d'un flux ;*
2. *Écrire une fonction* `lire_code_ascii` *qui lit un caractère dans un flux et qui renvoie son code ASCII (on utilisera la fonction* `int_of_char` *qui convertit un caractère en son code ASCII).*

On crée un flux à l'aide des fonctions `stream_of_string` *qui prend une chaîne de caractères et* `stream_of_channel` *qui prend en entrée un* `channel` *tel que* `stdin`*.*

▸ Exercice 4. ***Flux et arbres***

On représente une expression en notation préfixe par un arbre. Par exemple $+3*-251$ *correspond à l'arbre :*



1. *Écrire la déclaration du type* `arbre` *;*
2. *Écrire une fonction* `analyse` *qui prend un flux et qui retourne l'arbre correspondant. On générera une exception en cas d'erreur (par exemple l'exception prédéfinie* `Parsing.Parse_error`*).*