Expression Logique et Fonctionnelle . . . Évidemment

TP n° 4 : Réalisation d'un interpréteur du langage miniCaml

Objectifs du TP
- Déclarations de types.
- Types sommes, enregistrements.
- Définition de fonctions par filtrage.
- Un interpréteur de Mini Caml.

1 Déclaration de types

Comme beaucoup d'autres langages de programmation, OCaml offre au programmeur la possibilité de définir ses propres types de données à partir des types de base. Pour cela, il faut utiliser le mot clé `type` suivi du nom du type à définir et de la définition de ce type.

```
# type identificateur = string ;;
type identificateur = string
```

Il est même possible de déclarer simultanément plusieurs types, leurs définitions faisant référence à d'autres types en cours de déclaration.

```
# type identificateur = string
  and valeur = int
  and variable = identificateur * valeur
  and environnement = variable list ;;
type identificateur = string
and valeur = int
and variable = identificateur * valeur
and environnement = variable list
```

Supposons que nous voulions définir une fonction qui à partir d'un identificateur, d'une valeur et d'un environnement construit un nouvel environnement dans lequel une variable a été ajoutée, c'est–à–dire une fonction dont le type est

`identificateur` $->$ `valeur` $->$ `environnement` $->$ `environnement`.

Nous pourrions écrire :

```
let ajoute x v e = (x,v)::e ;;
```

Malheureusement, cette définition donne une fonction dont le type est plus général que celui désiré :

```
# let ajoute x v e = (x,v)::e ;;
val ajoute : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

Cela provient du mécanisme de type qui ne peut pas déduire de la définition de la fonction que `x` doit être de type `identificateur`, etc... même si c'est pourtant la volonté du programmeur.

Une solution consiste à laisser le soin au programmeur de typer la fonction qu'il définit :

```
# let ajoute (x : identificateur)
             (v : valeur)
             (e : environnement) : environnement =
                (x,v)::e ;;
val ajoute : identificateur -> valeur -> environnement -> environnement = <fun>
```

ou plus simplement encore

```
# let ajoute x v e : environnement = (x,v)::e ;;
val ajoute : identificateur -> valeur -> environnement -> environnement = <fun>
```

Il existe d'autres solutions pour permettre une reconnaissance syntaxique du type des expressions lorsque ce type est définit par le programmeur, sans avoir recours au typage explicite : les types sommes et les enregistrements.

2 Types sommes

Un *type somme* est un type déclaré par énumération des différentes *formes* que peuvent prendre les valeurs de ce type. Les formes sont déclarées au moyen de *constructeurs* dont les identificateurs en CAML doivent nécessairement débuter par une lettre majuscule.

Ainsi, le programmeur peut (s'il en éprouve le besoin) déclarer un type `booleen` n'ayant que deux valeurs `Vrai` et `Faux`.

```
# type booleen = Vrai | Faux ;;
type booleen = Vrai | Faux
# Vrai ;;
- : booleen = Vrai
# Faux ;;
- : booleen = Faux
```

Une telle déclaration de type correspond à ce qu'on appelle dans d'autres langages de programmation à un type énuméré. Le nombre de valeurs de types ainsi déclarés est fini.

En CAML, les types sommes ne se limitent pas à des types finis. Les constructeurs ne sont pas nécéssairement constants, mais peuvent être paramétrés par d'autres types. Par exemple, le type `int_etendu` étend le type `int` avec un constructeur non paramétré `Non_defini`, et un constructeur, `Int`, paramétré par les `int`.

```
# type int_etendu = Non_defini | Int of int ;;
type int_etendu = Non_defini | Int of int
# Non_defini ;;
- : int_etendu = Non_defini
# Int 3 ;;
- : int_etendu = Int 3
# Int (3+2) ;;
- : int_etendu = Int 5
```

Question 1 Programmez la fonction `quotient` de type `int->int->int_etendu` qui donne le quotient entier des deux entiers passés en paramètres lorsque le second n'est pas nul, et la valeur `Non_defini` dans le cas contraire. On doit avoir

```
# quotient 17 3 ;;
- : int_etendu = Int 5
# quotient 17 0 ;;
- : int_etendu = Non_defini
```

Les types sommes permettent même de définir des structures de données récursives. Par exemple, pour définir un type pour les listes d'entiers, si on se souvient qu'une liste d'entiers est soit la liste vide, soit un couple dont le premier élément est un entier et le second une liste d'entiers, on peut écrire la déclaration :

```
# type liste_entiers = Vide | Cons of (int*liste_entiers) ;;
type liste_entiers = Vide | Cons of (int * liste_entiers)
# Vide ;;
- : liste_entiers = Vide
# Cons (1,Vide) ;;
- : liste_entiers = Cons (1, Vide)
```

Question 2 Construisez en CAML une valeur de type `liste_entiers` représentant la liste $(1, 2, 3, 1)$.

3 Filtrage

Comment peut-on définir des fonctions dont les paramètres sont d'un type somme ? Par exemple, comment définir les opérateurs logiques usuels sur notre type `booleen` ? Comment définir l'arithmétique usuelle sur notre type `int_etendu` ? Comment calculer la longueur d'une liste ?

Lorsque le type somme ne contient que des constructeurs d'arité 0 (ou encore non paramétrés), une simple énumération des cas peut suffire. Par exemple, on peut programmer l'opérateur `et` en écrivant

```
# let et a b =
    if a = Vrai then
```

```
      b
    else
      Faux ;;
val et : booleen -> booleen -> booleen = <fun>
# et Vrai Faux ;;
- : booleen = Faux
# et Vrai Vrai ;;
- : booleen = Vrai
# et Faux Vrai ;;
- : booleen = Faux
# et Faux Faux ;;
- : booleen = Faux
```

Mais ce style de programmation ne convient plus dès qu'un des constructeurs du type a une arité au moins égale à 1. En effet comment extraire la valeur du paramètre d'un tel constructeur ?

La réponse réside dans le *filtrage de motifs* (*pattern matching* en anglais). En CAML, c'est l'expression `match ... with ...` qui permet d'effectuer du filtrage de motifs. Sa syntaxe est

```
match expression with
  | motif_1 -> expr_1
  | motif_2 -> expr_2
  ...
  | motif_n -> expr_n
```

Voici comme premier exemple, une définition de notre opérateur logique `et` à l'aide du filtrage des quatre motifs possibles pour les deux booléens.

```
let et a b =
   match a,b with
    | Vrai, Vrai -> Vrai
    | Vrai, Faux -> Faux
    | Faux, Vrai -> Faux
    | Faux, Faux -> Faux ;;
```

En utilisant le motif universel _, satisfait par toutes les instances, on peut écrire de manière plus concise :

```
let et a b =
   match a,b with
    | Vrai, Vrai -> Vrai
    | _ -> Faux ;;
```

ce qui signifie : lorsque les deux paramètres sont `Vrai` alors on renvoie `Vrai` et dans tous les autres cas on renvoie `Faux`.

Remarque 1 : La première version de la fonction `et` énumère quatre motifs deux à deux incompatibles, tandis que la deuxième version ne donne que deux motifs non incompatibles : le second (motif universel) contient le premier. L'incompatibilité ou non des motifs dans une énumération d'un filtrage a une conséquence sur l'ordre d'écriture des motifs. Dans le premier cas (incompatibilité des motifs), l'ordre importe peu. Dans le second cas (non incompatibilité des motifs), l'ordre est important. Par exemple, si on écrit

```
# let et a b =
     match a,b with
      | _ -> Faux
      | Vrai, Vrai -> Vrai ;;
Warning U: this match case is unused.
# et Vrai Vrai =
- : booleen = Faux
```

nous pouvons remarquer un message d'avertissement de l'interprète qui prévient qu'un motif est inutile. De plus la fonction ainsi définie est fausse.

Remarque 2 : Si l'énumération des différents cas d'un filtrage n'est pas exhaustif, l'interprète avertit le programmeur en donnant un exemple de motif non couvert par l'énumération.

```
# let et a b =
    match a,b with
     | Vrai, Vrai -> Vrai
     | Vrai, Faux -> Faux
     | Faux, Vrai -> Faux ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(Faux, Faux)
val et : booleen -> booleen -> booleen = <fun>
```

Remarque 3 : L'expression `match` peut être utilisée dans tous contexte. Dans les exemples qui précèdent, elle est utilisée dans le contexte d'une définition de fonctions à deux paramètres, le filtrage portant sur les deux paramètres. Dans le contexte d'une définition d'une fonction à un seul paramètre, une autre écriture du filtrage de motif est possible qui n'utilise pas l'expression `match` :

```
let une_fonction = function
 | motif_1 -> expr_1
 | motif_2 -> expr_2
 ...
 | motif_n -> expr_n
```

En fait toutes les fonctions que l'on définit en OCaml le sont par filtrage de motifs.

Question 3 Programmez la négation avec du filtrage de motifs sans utiliser l'expression `match`.

Voyons maintenant comment par le filtrage de motifs, on peut extraire des parties d'une valeur d'un type somme. Nous allons l'illustrer sur la fonction calculant la somme de deux `int_etendu`. Si les deux valeurs sont de la forme `Int n` et `Int p` alors la somme est de la forme `Int (n+p)`. Dans le cas contraire, la somme est `Non_definie`.

```
let plus n1 n2 =
   match n1,n2 with
    | Int n, Int p -> Int (n+p)
    | _ -> Non_defini
```

Question 4 Reprenez la fonction `quotient` pour l'étendre aux `int_etendu`. Le type de la fonction doit maintenant être

`int_etendu->int_etendu->int_etendu`.

Question 5 Définissez une fonction calculant la longueur d'une liste d'entiers. Son type est

`liste_entiers->int`.

```
# longueur Vide ;;
- : int = 0
# longueur (Cons (1,Vide)) ;;
- : int = 1
# longueur (Cons (2, Cons (1,Vide))) ;;
- : int = 2
```

Remarque 4 : Dans un motif, on ne peut pas utiliser plusieurs occurences de la même variable de motif. Par exemple, pour définir l'égalité de deux `int_etendu` on pourrait être tenté d'écrire

```
let egal n1 n2 =
  match n1,n2 with
   | Int n, Int n -> Vrai
   | Non_defini, Non_defini -> Vrai
   | _ -> Faux ;;
```

mais ce code produit le message d'erreur `Variable n is bound several times in this matching`.

Si on tient malgré tout à suivre ce schéma de motif, on peut écrire

```
let egal n1 n2 =
  match n1,n2 with
    | Int n, Int p -> if n=p then Vrai else Faux
    | Non_defini, Non_defini -> Vrai
    | _ -> Faux ;;
```

ou bien utiliser la notion de *garde*

```
let egal n1 n2 =
  match n1,n2 with
    | Int n, Int p when n=p  -> Vrai
    | Non_defini, Non_defini -> Vrai
    | _ -> Faux ;;
```

Question 6 Définissez une fonction de type `int->liste_entiers->int_etendu` qui donne le rang de l'entier passé en premier paramètre dans la liste passée en second paramètre si cet entier y figure, qui donne la valeur `Non_defini` dans le cas contraire.

```
# rang 1 Vide ;;
- : int_etendu = Non_defini
# rang 1 (Cons (1,Vide)) ;;
- : int_etendu = Int 0
# rang 1 (Cons (2, Cons (1,Vide))) ;;
- : int_etendu = Int 1
```

4 Mini Caml

Le but de cette partie est de réaliser un évaluateur de phrases du langage Mini Caml étudié en cours.

4.1 Grammaire du langage

On rappelle la grammaire définissant les phrases (ou programmes) de Mini Caml.

$\langle Phrase \rangle$	::=	$\langle Declaration \rangle \mid \langle Expression \rangle$	
$\langle Declaration \rangle$	::=	**let** *variable* = $\langle Expression \rangle$	
$\langle Expression \rangle$	::=	*entier* | *booleen* | *variable*	(Expr. atomiques)
		$\mid \langle Expr\_unaire \rangle \mid \langle Expr\_binaire \rangle$	(Expr composees)
		$\mid \langle Expr\_cond \rangle \mid \langle Expr\_fonc \rangle$	
		$\mid \langle Expr\_appl \rangle$	
$\langle Expr\_unaire \rangle$	::=	$(\langle Op\_unaire \rangle \ \langle Expression \rangle)$	(Expr unaires)
$\langle Expr\_binaire \rangle$	::=	$(\langle Expression \rangle \ \langle Op\_binaire \rangle \ \langle Expression \rangle)$	(Expr binaires)
$\langle Expr\_cond \rangle$	::=	(**if** $\langle Expression \rangle$ **then** $\langle Expression \rangle$ **else** $\langle Expression \rangle$)	(Expr conditionnelles)
$\langle Expr\_fonc \rangle$	::=	(**fun** *variable* -> $\langle Expression \rangle$)	(Expr fonctionnelles)
$\langle Expr\_appl \rangle$	::=	$(\langle Expression \rangle \ \langle Expression \rangle)$	(Application)
$\langle Op\_unaire \rangle$	::=	- | **not**	(Op unaires)
$\langle Op\_binaire \rangle$	::=	+ | - | * | / | && | || | = | <=	(Op binaires)

4.2 Le matériel fourni

Pour vous aider, vous trouverez dans le fichier `minicaml.zip` des fichiers sources de certains modules.

Question 7 Récupérez ce fichier et décompressez-le. Vous devez obtenir

1. les deux fichiers sources du module `Lexical`, `lexical.mli` et `lexical.ml`, donnant une fonction d'analyse lexicale utilisée par le module `Syntaxe` ;
2. les deux fichiers sources du module `Syntaxe`, `syntaxe.mli` et `syntaxe.ml`, donnant une fonction d'analyse syntaxique utilisée par le module `Evaluation` et une fonction de conversion d'expressions en chaîne utilisée par le module `Minicaml` ;
3. les deux fichiers sources du module `Evaluation`, `evaluation.mli` et `evaluation.ml`, qui ne définit qu'une seule fonction : `eval`. À noter que l'implémentation du module (fichier `evaluation.ml`) est très incomplète ;
4. les deux fichiers sources du module `Minicaml` qui déclenche une boucle d'interaction avec un interpréteur du langage Mini Caml ;

5. un répertoire `Doc/` contenant la documentation de ces différents modules ;
6. un `Makefile` pour compiler.

4.3 Objectif

Il s'agit de réaliser l'implémentation du module `Evaluation`, seule pièce manquante pour pouvoir compiler et utiliser notre interpréteur de Mini CAML.

En attendant vous allez compiler les modules `Lexical` et `Syntaxe`.

Question 8

Commencez par compiler l'interface du module `Lexical`

```
ocamlc -c lexical.mli
```

puis l'implémentation

```
ocamlc -c lexical.ml
```

Vous devez avoir obtenu deux nouveaux fichiers `lexical.cmi` et `lexical.cmo`.

Question 9 Faîtes de même avec pour le module `Syntaxe`.

4.4 Familiarisation avec le module `Syntaxe`

Dans cette section vous allez vous familiariser avec le module `Syntaxe`. Vous allez utiliser un interpréteur CAML avec ce module. Pour cela, il faut lancer l'interpréteur en précisant ce module et le module `Lexical` dont il dépend sur la ligne de commande [1]

```
ocaml lexical.cmo syntaxe.cmo
```

Question 10 Étudiez soigneusement la documentation du module `Syntaxe`. En particulier, comparez les types définis et la grammaire du langage rappelée ci-dessus (cf 4.1).

Question 11 Utilisez la fonction `Syntaxe.analyse` pour construire une représentation interne des phrases suivantes :

1. `12;`
2. `true;`
3. `x;`
4. `(x + 1);`
5. `(y && (not z));`
6. `(0 <= x);`
7. `(- x);`
8. `(not y);`
9. `(if (0 <= x) then x else (- x));`
10. `(f 3);`
11. `let f = (fun x -> (if (0 <= x) then x else (- x))).`

Étudiez soigneusement la structure de chacune de ces phrases.

Question 12 Quelques phrases syntaxiquement correctes, mais sémantiquement incorrectes.

1. `(not 3);`
2. `(12 && 15);`
3. `(if 0 then 1 else 2);`
4. `(3 4);`
5. `(12 -3);`

Attention, notez bien le résultat de l'analyse syntaxique de la dernière expression. Est-ce bien ce que vous attendiez ?

Question 13 Enfin quelques phrases qui semblent syntaxiquement correctes mais qui ne le sont pas.

1. `12 + 3;`
2. `(if x = 0 then 1 else 2);`
3. `(fun x -> x + 1).`

1. Ajoutez `ledit` au début de cette commande pour plus de confort de travail.

4.5 Implémentation du module Evaluation

Dans cette partie il s'agit de compléter petit à petit le fichier d'implémentation du module : `evaluation.ml`.

Question 14 Commencez par y porter votre nom dans l'en-tête.

L'évaluation qu'il s'agit de mettre en œuvre ici est essentiellement celle de CAML à une différence près :

- dans une expression composées, les différentes sous-expressions sont évaluées de droite à gauche ;
- dans une application d'une fonction à un argument, l'argument est évalué d'abord puis la fonction et enfin la réduction peut s'opérer (stratégie par valeur).

La seule différence réside dans l'évaluation des variables situées dans le corps d'une fonction qui se fera dans l'environnement courant d'évaluation (liaison dynamique).

4.5.1 La fonction eval

La fonction principale à réaliser est la fonction `eval` qui est de type

`environnement` → `Syntaxe.phrase` → `environnement` ∗ `valeur`.

Cette fonction prend un environnement d'évaluation comme premier argument ((`string` ∗ `valeur`) `list`), et une phrase en second argument. Elle renvoie un couple constitué d'un environnement et d'une valeur. L'environnement est augmenté d'une liaison si la phrase correspond à une déclaration de variable

Voici le code de cette fonction

```
let eval = fun env p ->
  match p with
    | Decl (Decla (x,e)) ->
      let v = eval_expr env e in
      ((x, v)::env, Liaison (x, v))
    | Expr e -> (env, Valeur (Expr_int 0))
    | Vide -> (env, Aucune)
```

Question 15 Chargez le fichier `evaluation.ml` dans un interpréteur. Observez le type de la fonction `eval`. Est-il conforme à ce que demande l'interface (cd documentation) ?

Question 16 Observez la réponse fournie par la fonction `eval` pour les deux phrases

- `12`
- et `let x = 1`

évaluées dans un environnement vide.

4.5.2 La fonction eval_expr

Comme on a pu le voir tout repose sur la fonction `eval_expr` chargée d'évaluer des expressions (phrases du type `Syntaxe.Expression`).

Cette fonction n'est pas dans la partie interface du module. Ce n'est pas nécessaire. Cette fonction doit avoir le type

`environnement` → `Syntaxe.Expression` → `valeur`.

Pour l'instant la réalisation de cette fonction ne comprend que deux cas :

1. celui des expressions entières qui donne une valeur correspondante immédiate ;
2. et tous les autres expressions, pour obtenir un filtrage exhaustif, qui donne la valeur `Aucune`.

Question 17 Dans cette réalisation actuelle, quel est le type de la fonction `eval_expr` ? Est-ce normal ?

Dans la suite, nous allons compléter cette fonction cas par cas.

4.5.3 Cas des expressions atomiques booléennes

Question 18 Complétez la fonction pour les expressions atomiques booléennes.

4.5.4 Cas des variables

C'est ici que l'environnement entre en scène !

Question 19 Faîtes quelques essais avec la fonction `List.assoc` pour bien en comprendre le fonctionnement.

Question 20 Complétez la fonction `eval_expr` pour traiter le cas de variables. La fonction doit déclencher une exception `Variable_non_liee` si la variable n'est liée à aucune valeur dans l'environnement.

Question 21 Observez maintenant le type de la fonction.

4.5.5 Cas des fonctions

Les fonctions étant des valeurs, leur évaluation est aussi immédiate.

Question 22 Complétez la fonction pour ce cas.

Question 23 Vérifiez pour les phrases

- `(fun x -> (x + 1))`
- et `let f = (fun x -> (x + 1))`.

4.5.6 Cas des expressions unaires

Il n'y a que deux opérateurs unaires, l'un sur les entiers, l'autre sur les booléens. Cependant, le langage Mini CAML n'étant pas statiquement typé, il se peut qu'à l'exécution le type sur lequel porte un de ces deux opérateurs ne soit pas correct.

D'autre part une expression unaire peut avoir un second membre non réduit. Par exemple :

`(- (3 + 1))`.

Conformément à ce qui a été annoncé plus haut, la stratégie d'évaluation impose de calculer d'abord la sous-expression. C'est notre premier cas récursif.

Question 24 Complétez la fonction pour évaluer les expressions unaires. Elle déclenche l'exception `Typage_incorrect` en cas de valeur de type incompatible avec l'opérateur.

4.5.7 Cas des expressions binaires

La stratégie d'évaluation d'une expression binaire

$$(e_1 \text{ op } e_2)$$

que nous avons adoptée impose l'évaluation complète de e_2 puis celle de e_1 et enfin celle de l'expression complète. Encore un cas récursif.

Question 25 Complétez la fonction pour évaluer les expressions binaires. Elle déclenche l'exception `Typage_incorrect` en cas de valeur de type incompatible avec l'opérateur.

Question 26 Testez l'évaluation de plusieurs expressions arithmétiques et booléennes de votre choix. Pensez aussi à tester des types incompatibles.

4.5.8 Cas des conditionnelles

Les expressions conditionnelles de la forme

$$(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)$$

nécessitent l'évaluation d'abord de l'expression e_1 (la condition), puis selon la valeur de cette expression celle de e_2 ou de e_3.

Question 27 Complétez la fonction `eval_expr` afin de prendre en compte les expressions conditionnelles. La fonction déclenche une exception `Condition_incorrecte` si la valeur de e_1 n'est pas un booléen.

4.5.9 Cas des application

Vient maintenant le cas essentiel ! Celui de l'application d'une expression à une autre

$$(e_1 \;\; e_2).$$

Il s'agit

1. d'évaluer d'abord e_2 afin d'obtenir sa valeur v_2 ;
2. puis d'évaluer e_1 pour obtenir sa valeur v_1 ;
3. et enfin d'évaluer l'expression $e[x \leftarrow v_2]$ si la valeur v_1 est bien de la forme

$$v_1 = \text{Valeur } (\text{Expr\_fonc}(x, e)).$$

Si v_1 n'est pas de cette forme, l'application est impossible.

L'évaluation de l'expression $e[x \leftarrow v_2]$ se fait dans l'environnement courant, celui passé en paramètre à la fonction `eval_expr`. Si e contient des variables libres autres que x, ces variables sont évaluées dans cet environnement à l'exécution : c'est la *liaison dynamique* des variables.

Question 28 Réalisez d'abord une fonction nommée `subs` de type

`string` → `Syntaxe.Expression` → `Syntaxe.Expression` → `Syntaxe.Expression`,

qui renvoie l'expression $e_1[x \leftarrow e_2]$ si on lui passe en paramètre la variable x et les expressions e_1 et e_2. Cette fonction n'est pas déclarée dans l'interface du module `Evaluation`, ce n'est pas nécessaire.

Question 29 Achevez la fonction `eval_expr` en traitant le cas des expressions application. La fonction déclenche l'exception `Application_impossible` dans le cas où la valeur de la première expression n'est pas une fonction.
Question 30 Testez l'évaluation de l'expression `((fun x -> (x + y)) (3 + 1))`

1. dans l'environnement vide ;
2. puis dans l'environnement `[("y", Valeur (Expr_int 2))]`.

4.6 Utilisation de l'interpréteur `minicaml`

Il ne reste plus qu'à effectuer les dernières compilations.

```
make minicaml
```

On obtient un exécutable nommé `minicaml`.
Pour l'utiliser

```
ledit ./minicaml
?-
```

Souvenez-vous que toutes les phrases doivent être rédigées sur une seule ligne.

```
?- 12
-> 12
?- true
-> true
?- x
Variable x non liee.
?- let x = 1
val x = 1
?- (x + 1)
-> 2
```

Question 31 Définissez une fonction signe qui vaut 0 pour l'entier nul, 1 pour un entier positif, et -1 pour un entier négatif.
Question 32 Pouvez-vous écrire la fonction factorielle ?