Programmation 1 – TP n° 4
Devoir 1 : OrlandosML & SuperML

Ce TP constitue le Devoir 1 qui est à rendre pour le Mardi 21 Octobre.

L'objectif est d'écrire en OCaml un évaluateur pour un langage de programmation fonctionnel, intelligemment baptisé *OrlandosML*.

On commence par définir un mini-langage contenant les expressions arithmétiques, l'opérateur "Let in" et les fonctions. On s'intéresse ensuite à 3 extensions de ce langage :

1. Fonctions récursives.
2. Exceptions.
3. Aspect impératifs.

Vous rendrez donc 3 fichiers `.ml` différents, un pour chacune des extensions, chaque fichier contenant le code du mini-langage.

Quelques conseils

- Faites nous parvenir avant la ligne-morte vos fichiers `.ml` ainsi qu'un rapport (en .pdf par exemple) explicitant vos implémentations (et contenant les réponses aux questions avec un " ?").
- Ne cherchez pas à faire des fonctions optimisées (récursives terminales, par exemple, ou par continuations) si cela doit vous prendre plus de temps.
- Rendez un code aéré, clair et commenté (et qui compile, bien entendu).
- Illustrez abondamment votre rapport d'exemples adéquats.
- Utilisez ce que vous avez vu en cours.

1 Un mini-langage de programmation fonctionnel

Dans cette section, OrlandosML est construit de manière incrémentale.

1.1 Expressions arithmétiques

Le langage de base est décrit par le type de données suivant :

```
type expr =
    | TInt of int              (* i *)
    | TAdd of expr * expr      (* e1 + e2 *)
    | TMul of expr * expr      (* e1 * e2 *)
```

Q 1.1 Écrivez une fonction `eval : expr -> int` qui évalue un terme de OrlandosML en une valeur de OCaml.

1.2 Variables

On ajoute une construction "Let in" à OrlandosML permettant d'associer une valeur à une variable dans l'environnement.

Les variables seront représentées par des chaînes de caractères :

```
type var = string
```

On définit le type d'un environnement, ainsi que les primitives nécessaires à son utilisation, par la signature suivante :

```
module type Env = sig
    type 'a env
    exception Unbound_variable
    val empty : 'a env
    val get : 'a env -> var -> 'a
    val add : 'a env -> var -> 'a -> 'a env
end
```

Ici, `'a` sera le type des valeurs associées aux variables de l'environnement.

Remarque. Les environnements sont *fonctionnels* car la fonction `add` ne modifie pas l'environnement mais en renvoie un nouveau.

Q 1.2 Implémentez un module de type `Env`. Une solution simple consiste à prendre pour le type `'a env` le type `(var * 'a) list`.

On ajoute à notre langage la construction "Let in" et les variables :

```
| TLet of var * expr * expr     (* let x = e1 in e2 *)
| TVar of var                   (* x *)
```

Q 1.3 Écrivez la nouvelle fonction d'évaluation `eval : int env -> expr -> int`.

Q 1.4 Vérifiez son comportement sur des exemples. Que se passe-t-il en particulier pour l'expression suivante : `let x = 5 in (let x = 3 in x) * x` ?

1.3 Fonctions et Applications

L'objectif est d'ajouter encore deux constructions : les fonctions et l'application.

```
| TFun of var * expr            (* fun x -> e *)
| TApp of expr * expr           (* e1 (e2) *)
```

Valeurs

Désormais, la fonction d'évaluation ne renvoie pas nécessairement un entier : l'évaluation peut renvoyer une valeur fonctionnelle comme `TFun("x",TAdd (TVar "x",TInt 3))`.

Q 1.5 Proposez un type `value` pour le résultat de la fonction d'évaluation. Ce type suffit-il pour évaluer les termes suivants ?

`let x = 3 in fun y -> y + x` et

`(fun x -> fun y -> x * y) 7` ?

Evaluation

Q 1.6 Modifiez la fonction d'évaluation en conséquence ; le nouveau type de `eval` est donc `expr -> value env -> value`.

Q 1.7 Que doit renvoyer l'exécution OCaml du terme suivant :

```
let x = 5 in
let f = fun y -> x + (let x = y + 2 in x * 3) in
let y = 8 in
f y
```

et que renvoie votre fonction `eval` ?

1.4 Constructeurs et destructeurs

Pour se faciliter le travail dans les traductions futures, on ajoute à OrlandosML le couple constructeur/destructeur des paires.

```
| TPair of expr * expr                (* (e1,e2) *)
| TMatch of var * var * expr * expr   (* match e1 with (x,y) -> e2 *)
```

Q 1.8 Modifier `value` et `eval` pour prendre en compte ces opérations.

On dispose maintenant d'un langage fonctionnel sur les entiers, OrlandosML.
La suite du projet consiste à créer trois extensions du langage : ajoutant la gestion des fonctions récursives, des exceptions et des opérations impératives, pour obtenir un super langage qu'on a intelligemment appelé *SuperML*.

2 Fonctions récursives

On ajoute dans la première extension les fonctions récursives, codées à l'aide d'un "Let rec". Pour pouvoir utiliser facilement dans des exemples cette construction on ajoute aussi `TZero` qui encode une structure conditionnelle.

```
| TZero of expr * expr * expr         (* if e1=0 then e2 else e3 *)
| TLetrec of var * expr * expr        (* let rec x = e1 in e2 *)
```

Q 2.1 Ecrivez `rvalue` et `reval` pour inclure `TZero` et `TLetrec` (car l'opérateur `TLetrec` impose d'avoir un nouveau type de valeur).

Q 2.2 Programmez la fonction calculant la suite de Fibonacci ($u_0 = 1, u_1 = 1, u_{n+2} = u_{n+1} + u_n$) en OrlandosML.

3 Exceptions

On définit d'abord le type des expressions de cette extension :

```
type sexpr =
    | SInt of int                    (* i *)
    | SAdd of sexpr * sexpr          (* e1 + e2 *)
    | SMul of sexpr * sexpr          (* e2 * e2 *)
    | SVar of var                    (* x *)
    | SLet of var * sexpr * sexpr    (* let x = e1 in e2 *)
    | SApp of sexpr * sexpr          (* e1 (e2) *)
    | SFun of var * sexpr            (* fun x -> e *)
    | SRaise of sexpr                (* raise (Exce e1) *)
    | STry of sexpr * var * sexpr    (* try e1 with Exce x -> e2 *)
```

Q 3.1 Définissez une fonction d'évaluation `seval : sexpr -> svalue` pour cette extension.

Q 3.2 Que renvoie en OCaml l'évaluation du code suivant :

```
exception WTF of int
let _ = try
          try raise (WTF (raise (WTF 0)))
          with
            | WTF n -> n
        with
          | WTF n -> 1
```

Pourquoi ? Vérifiez que votre évaluateur se comporte de la même façon.

Q 3.3 Programmez une fonction `trans : sexpr -> expr` de cette extension dans OrlandosML telle que l'évaluation d'un terme et de sa traduction soient les mêmes.

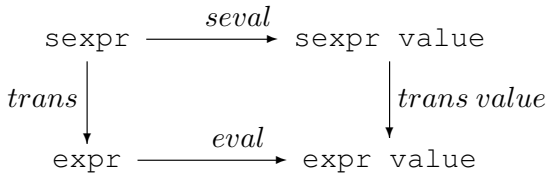

FIG. 1 – un exemple de question subsidiaire.

4 Aspects impératifs

On ajoute dans la dernière extension d'OrlandosML trois nouvelles opérations impératives :

- `SRef e` qui renvoie une adresse contenant la valeur de `e`,
- `SDeref a` qui renvoie la valeur contenue à l'adresse `a` et
- `SAffect a e` qui affecte la valeur `e` à l'adresse `a` et renvoie cette valeur (comme en C, et pas comme en OCaml).

Le type `iexpr` contient (entre autres) :

```
    | SRef of iexpr                  (* ref e *)
    | SDeref of iexpr                (* !e *)
```

```
    | SAffect of iexpr * iexpr          (* e1 := e2 *)
```

Remarque. Si jamais vous vous posez des questions à propos du type de `SAffect`, essayez le code qui suit avec OCaml.

```
let a = ref 0 and b = ref 1 in
let f x = if x = 0 then a else b in
f 4 := 3
```

Q 4.1 Choisissez une implémentation de la mémoire.

Q 4.2 Définissez `ivalue` et `ieval` pour cette extension.

Q 4.3 *(Long et Difficile)* Ajoutez les constructions nécessaires à OrlandosML afin de définir une traduction de cette extension dans OrlandosML.