Programmation fonctionnelle
TP noté - 2 heures

Introduction

Le TP noté est à effectuer seul. Tout manquement à cette règle provoquera une pénalité qui peut aller jusque 20 points.

Le TP noté est constitué de quatre exercices.

— L'exercice 1

— Les exercices A,B et C.

Il suffit de faire correctement l'exercice 1, et un autre exercice en entier pour avoir une note *au minimum* de 12, quelque soit le deuxième exercice choisi.

Les exercices ne sont pas du même ordre de difficulté.

Rendu

Vous devez rendre un fichier zip contenant deux répertoires et un fichier `reponses.txt`

— Le premier répertoire s'appelle `eval` et contiendra l'interpréteur modifié en tenant compte des exercices 1, A et B.

— Le deuxième répertoire s'appelle `eval-mini` et contiendra l'interpréteur modifié pour la deuxième partie de l'exercice C.

Chaque question faite doit être mentionnée dans le fichier `reponses.txt` : seules les questions mentionnées dans ce fichier seront prises en compte dans la notation.

— Pour chaque question, expliquer dans quel fichier, et à quelle ligne (approximativement) se trouve la réponse à la question. **Ne pas recopier le code dans le fichier `reponses.txt` !**

1 Exercice 1 (exercice commun) : chaînes de caractères

Dans ce premier exercice, on ajoute les chaînes de caractères.
La chaîne de caractères `"toto"` sera représentée par l'expression `String "toto"`.
On dispose de 3 opérations sur les chaînes de caractères :

— Concaténer deux chaînes. Cet opérateur s'écrit `^` dans le langage et `Concat` dans l'arbre syntaxique. Par exemple l'expression `"to" ^ x` sera représentée par `Concat(String "to", Var x)`

— Obtenir la lettre numéro i. La `i`-ème lettre de la chaîne `x` s'écrit `x.[i]` dans le langage et `Cell` dans l'arbre syntaxique. Par exemple l'expression `"to".[1]` sera représentée par `Cell(String "to", Num 1)`. Pour vous aider à faire cette fonction, vous pouvez utiliser la fonction `string_get : str -> int -> str` qui prend en argument une chaîne, un entier, et qui renvoie la $i$ème lettre de la chaîne (sous forme d'une chaîne).

— Récupérer la longueur de la chaîne. Cette fonction est notée `len` dans le langage et se transforme en `Builtin "len"` dans l'arbre syntaxique. Pour vous aider à faire cette fonction, vous pouvez utiliser la fonction `string_length : str -> int` qui prend en argument une chaîne et renvoie sa longueur.

```
type expr = ...
| String of string
| Concat of  expr *expr
| Cell of expr * expr
```

Q 1) Modifier l'évaluateur pour traiter le cas de `String`.

Q 2) Modifier l'évaluateur pour traiter le cas de `Concat`.

Q 3) Modifier l'évaluateur pour traiter le cas de `Cell`.

Q 4) Modifier l'évaluateur (la fonction `eval_builtins`) pour traiter le cas de `len`.

2 Exercice A : Séquences

Dans cet exercice, on ajoute :

— `()` qui correspond à `Unit` dans l'arbre syntaxe

— `;` qui permet d'évaluer deux expressions et de renvoyer le résultat de la deuxième. Il correspond dans l'arbre syntaxique à `Then`. Ainsi l'expression `x;3` sera représentée par `Then(Var "x", Num 3)`

— `print_int` qui permet d'afficher un entier et qui renvoie `()`.

— `print_expr` qui permet d'afficher une expression et qui renvoie `()`.

— `ignore`, la fonction qui ne fait rien de son argument et renvoie `()`.

```
type expr = ...
| Then of expr * expr
| Unit
```

Q 5) Modifier l'évaluateur pour traiter le cas de `Unit`.

Q 6) Modifier l'évaluateur pour traiter le cas de `Then`. Attention : Dans le cas où le premier argument n'est pas de type unit, il faut afficher un warning "this expression should have type unit."

Q 7) Modifier l'évaluateur (la fonction `eval_builtins`) pour traiter le cas de `print_int` et `print_expr`

Q 8) Modifier l'évaluateur (la fonction `eval_builtins`) pour traiter le cas de `ignore`.

Q 9) (Cette question doit être traitée uniquement dans le fichier `reponses.txt`). Expliquer comment on peut remplacer n'importe quel code qui utilise `tee` et `";"` en un code au comportement équivalent qui n'utilise aucun de ses deux constructions.

— Expliquer par quelle expression il faut remplacer `tee`. (Rappel : la fonction `tee` affiche et renvoie son argument)

— Expliquer comment remplacer `a ; b` par un code (dépendant de `a` et `b`) qui donne le même résultat.

3 Exercice B : tuples

Dans cet exercice, on généralise les paires aux tuples, grâce à l'élément `Tuple`, qui prend une liste en argument.

Ainsi, l'expression `(true,2,x)` sera représentée par `Tuple [Bool true, Num 2, Var "x"]`.

Comme une paire n'est rien d'autre qu'un tuple de deux éléments, les paires ont disparu du langage.

```
type expr = ...
| Tuple of expr list
```

Dans la suite, il est conseillé (mais pas obligatoire) d'utiliser les fonctions suivantes

— `List.map` (voir cours)

— `List.fold_left` (voir cours)

Q 10) Modifier l'évaluateur pour gérer le cas de `Tuple`.

Q 11) Modifier l'évaluateur (la fonction `eval_builtins`) pour traiter le cas de `fst` et `snd`. On vérifiera que le tuple est bien une paire.

Pour faciliter l'utilisation des tuples, on introduit la possibilité d'avoir un argument qui est un tuple.

Par exemple, on peut écrire :

```
let (x,y) = (1,2) in x + y
```

ou encore

```
(fun (x,y,z) -> x+y) (1,2,4)
```

On utilisera pour cela les deux constructions suivantes dans l'arbre syntaxique

```
type expr = ...
| LetTuple of string list * expr * expr
| FunTuple of string list * expr
```

Les deux exemples s'écrivent donc :

```
LetTuple (["x"; "y"], Tuple [Num 1; Num 2],
          Plus (Var "x", Var "y"))
App (FunTuple (["x"; "y"; "z"], Plus (Var "x", Var "y")),
  Tuple [Num 1; Num 2; Num 4])
```

Q 12) Modifier l'évaluateur pour gérer le cas de `LetTuple` et `FunTuple` (Aide : pour gérer `LetTuple`, inspirez-vous du code de `Let`).

4 Exercice C : Valeurs

4.1 Partie 1

Attention : cette première partie est à réaliser dans le répertoire `eval`.

Une valeur est une expression qui ne peut pas être évaluée (ou plus exactement pour laquelle l'évaluation ne va rien faire). Par exemple, les quatre exemples suivants sont des valeurs :

```
2
true
(3,4)
(fun x -> 1 + 1)
```

Les quatre exemples suivants ne sont pas des valeurs :

```
2 + 2
if true then false else true
(3,2+2)
(fun x -> x + 1) 4
```

Q 13) Rajoutez dans le fichier `test.ml` une fonction `is_value: expr -> bool` qui détermine si une expression est une valeur. Ajoutez cette fonction au fichier `test.ml` pour qu'elle vérifie que le résultat de l'évaluation est bien une valeur.

4.2 Partie 2

Attention : la deuxième partie change substantiellement le code. Elle est à réaliser dans le répertoire `eval-mini`.

Le répertoire `eval-mini` contient un mini-évaluateur, plus simple que celui à utiliser dans les questions précédentes.

Q 14) Modifier l'évaluateur pour qu'il soit en appel par nom et pas en appel par valeur (voir le cours pour trouver ce qu'il faut changer. Il y a maximum deux lignes à changer).

Le but de cette partie est de définir un type value, et de changer l'évaluateur pour que la fonction eval ait comme type : `expr -> value`, c'est à dire que le type retourné par la fonction eval soit différent du type d'entrée.

Le type `value` est dans le fichier `eval.ml` et défini de la façon suivante :

```
type value =
  | VNum of int
  | VFun of string * expr
  | VBool of bool
  | VPair of value * value
  | VBuiltin of string
;;
```

Les 5 cas correspondent aux 5 cas avec un nom similaire du type `expr`. Dans le deuxième cas, le deuxième argument est bien `expr` et non pas `value`.

Q 15) Modifier le fichier `eval.ml` et le fichier `main.ml` pour que la fonction `eval` soit de type `expr -> value`. Il faudra en particulier définir une fonction `string_from_value : value -> string`.

Q 16) (Répondre à cette question dans le fichier `reponses.txt`). Expliquer pourquoi il est difficile d'avoir une fonction `eval` de type `expr -> value` en appel par valeur (commencez par changer l'évaluateur pour qu'il soit en appel par valeur, regardez l'erreur produite, puis proposez une explication).