DM PFITA - listes, types utilisateur et modules

(*UPS - L3 Info - 2021-2022*)

Ce Devoir Maison a pour but :

1. d'exercer vos compétences acquises en TP PFITA (sur les mêmes thèmes qui seront au programme du CT PFITA)
2. d'obtenir une note individuelle de DM (comptant 20% de l'UE)
3. de vous permettre de vous frotter à la résolution d'un problème réaliste, auquel le langage OCaml est typiquement bien adapté (l'encodage d'un langage de programmation tiers en OCaml, puis l'implémentation de fonctions de typage, d'exécution, et d'optimisation du code de programmes de ce langage tiers).

Pour toute question, vous privilégierez l'utilisation du forum Moodle commun à toute la promotion (ou à défaut pour une question "critique" nécessitant d'envoyer un fragment de code : un e-mail à votre encadrant de TD).

La section 1 précise la date et les consignes de rendu, et la section 2 donne d'autres informations importantes sur la préparation de ce devoir maison.

1° Consignes de rendu

Ce devoir Devoir Maison 1 de PFITA peut être préparé directement sur la plateforme PFITAXEL, ou en dehors avec un autre éditeur adapté à OCaml, mais dans tous les cas il doit être soumis à la fin sur PFITAXEL avec votre compte personnel (TOKEN).

Votre rendu doit être effectué (en cliquant une dernière fois sur le bouton `Noter`) avant le **dimanche 12 décembre 2021 à 23:55**.

Votre code doit comporter :

- votre nom et votre prenom (sans accent) définis globalement de la façon suivante :

  ```
  let nom = "DUPONT" and prenom = "John";;
  ```

- le code des questions que vous avez implantées.

2° Informations importantes

Ce devoir maison est un devoir personnel et par conséquent **tout échange de code vous est formellement interdit et serait sanctionné**. Ne vous laissez donc pas tenter par un plagiat. Lors de la correction, des outils de mesure de similarité de code seront utilisés.

Pour évaluer votre travail, nous utiliserons à nouveau une évaluation automatisée à base de tests fonctionnels et de tests de qualité de code.

Mais contrairement au fonctionnement des TP, tous les tests fonctionnels ne sont pas fournis à l'avance : mais lorsque vous cliquerez sur `Noter` dans l'application PFITAXEL, le rapport de tests donnera essentiellement une vérification du type de vos fonctions.

D'où les remarques suivantes :

1. Vous devez prendre en compte les **exigences de qualité et de lisibilité du code** (cf. le document *PFITA et qualité de code* disponible sur Moodle). Comme en TP, des malus seront appliqués en cas de motif de code indésirable.
2. Après avoir développé vos fonctions, vous devez **tester votre code** vous-même, en utilisant par exemple le *toplevel* de l'application PFITAXEL.
3. Vous devez **respecter le nom et le type des fonctions** spécifiées dans l'énoncé (ainsi que l'ordre de leurs arguments, si la fonction en comporte plusieurs).
4. Et après avoir terminé votre devoir, **vous devez vous assurer que votre code ne comporte aucune erreur de syntaxe ou de typage** (c'est la moindre des choses pour un tel rendu, si vous ne vérifiez pas cela vous risquez d'obtenir la note 0/20).

Pour vérifier les points 3. et 4. ci-dessus, vous pouvez utiliser les boutons `Compiler` et `Noter!` de PFITAXEL.

Rappel : dans les fichiers source OCaml (fichiers ayant l'extension `.ml`), il convient de mettre tout le texte non exécutable entre commentaires `(* comme ceci *)`

Attention : Vous devez **utiliser uniquement le fragment fonctionnel d'OCaml** (pas de boucle `while` ou `for`, de séquence d'instructions "`;`" ou autres structures de contrôle impératives, ni de références ou autres structures de données mutables).

L'utilisation des fonctions de la bibliothèque standard des listes (`List.`) est autorisée. Vous pouvez notamment utiliser l'opérateur de concaténation `( @ )`, et définir vos propres fonctions auxiliaires.

3° Fonctions sur les listes

1) Définir la fonction `doublons : 'a list -> 'a list` prenant en argument une liste `l` et renvoyant la liste sans répétition des éléments apparaissant plusieurs fois dans `l`. L'ordre des éléments dans la liste résultat est indifférent.

Exemple : `doublons [4; 5; 5] = [5]`

Indication : vous pourrez utiliser la fonction `List.mem : 'a -> 'a list -> bool` testant si un élément est présent dans une liste

2) Définir la fonction `occs : 'a list -> ('a * int) list` prenant en argument une liste `l` et renvoyant une liste `res` contenant des couples `(e, n)` où chaque élément `e` est unique dans la liste `res` et `n` est le nombre d'occurrences de `e` dans la liste `l`. L'ordre des éléments dans la liste `res` est indifférent.

Exemple : `occs [4; 5; 5] = [(4, 1); (5, 2)]`

4° Modules et Types Abstraits de Données

On considère la signature de module suivante, représentant un "environnement" (correspondant à la notion de "dictionnaire" en Python ou encore de "Map" en Java) :

```
module type tENV = sig
  type ('x, 'v) env
  val empty : ('x, 'v) env
```

```
  val get : 'x -> ('x, 'v) env -> 'v
  val put : 'x -> 'v -> ('x, 'v) env -> ('x, 'v) env
end
```

Intuitivement, un module `(E : tENV)` fournit un type abstrait OCaml `('x, 'v) E.env` et plusieurs définitions :

- la constante `E.empty` représente l'environnement vide,
- la fonction `E.put` permet d'ajouter à un environnement existant une valeur (de type `'v`) en l'associant à un identificateur (de type `'x`)
- la fonction `E.get` permet de récupérer la valeur associée à un identificateur, en levant une exception avec `failwith` si cela n'est pas possible.

Ce type abstrait de données peut donc être désigné par plusieurs noms (environnement, dictionnaire, Map…) mais on privilégiera la terminologie "environnement" étant donné l'utilisation que l'on en fera dans les section 5.3° et 5.4° (utilisation pour modéliser le "stockage en mémoire des variables d'un programme" ; dans ce cas, la variable de type `'x` sera juste instanciée par `string`, pour représenter des "vrais identificateurs de variables").

Afin de montrer l'interchangeabilité des implémentations pour un même type abstrait de données, on introduit deux signatures "concrètes" alternatives à `tENV` :

```
module type tENV_LIST = sig
  type ('x, 'v) env = ('x * 'v) list
  val empty : ('x, 'v) env
  val get : 'x -> ('x, 'v) env -> 'v
  val put : 'x -> 'v -> ('x, 'v) env -> ('x, 'v) env
end
module type tENV_FUN = sig
  type ('x, 'v) env = 'x -> 'v
  val empty : ('x, 'v) env
  val get : 'x -> ('x, 'v) env -> 'v
  val put : 'x -> 'v -> ('x, 'v) env -> ('x, 'v) env
end
```

Noter la différence concernant le type `env`.

3) Implémenter un `module ENV_LIST : tENV_LIST` (utilisant donc comme définition `type ('x, 'v) env = ('x * 'v) list`), avec comme contrainte supplémentaire que :

> la fonction `put` doit être "idempotente" → c'est-à-dire que si on l'appelle successivement deux fois avec les mêmes arguments `"x"` et `42`, on doit obtenir le même environnement qu'avec un seul appel :

```
E.(put "x" 42 (put "x" 42 empty)) = E.(put "x" 42 empty)
```

4) Implémenter un `module ENV_FUN : tENV_FUN` (utilisant donc comme définition `type ('x, 'v) env = 'x -> 'v`).

Remarques :

- Vous aurez bien sûr besoin de la commande `failwith`…
- Deux implémentations `ENV_LIST` et `ENV_FUN` sont demandées mais une seule vous suffira pour tester la suite du Devoir Maison.

5° Type utilisateurs et encodage d'un mini-langage-fonctionnel (MLF)

Le but de cette section est de manipuler plusieurs types utilisateurs représentant un sous-ensemble des expressions de OCaml, correspondant à un mini-langage-fonctionnel (noté MLF dans la suite).

```
type binop =
  | And  (* conjonction booléenne (&&) *)
  | Or   (* disjonction booléenne (||) *)
  | Add  (* addition des nombres entiers (+) *)
  | Leq  (* comparaison (<=) sur les entiers *)

type typ = Int | Bool

type expr =
  | IConst of int   (* constante entière *)
  | BConst of bool  (* constante booléenne *)
  | Var of string   (* variable définie par son identificateur *)
  | If of expr * expr * expr  (* If(b,e1,e2) <=> "if b then e1 else e2" *)
  | Let of string * expr * expr  (* Let(x,e1,e2) <=> "let x = e1 in e2" *)
  | Call of binop * expr * expr  (* Call(Add,e1,e2) <=> "e1 + e2", etc. *)
```

- les constructeurs du type `binop` représentent chacun une **opération binaire** sur les entiers ou les booléens,
- et les constructeurs du type `expr` représentent des **expressions** (constantes, variable de type entier ou booléen, if-then-else, définition locale, et appel d'opérateur).

Exemple :

Considérons l'expression suivante du langage MLF.

```
Let("x", IConst(5),
  Let("a0", Call(Leq, IConst(0), Var("x")),
    Let("b0", Call(Leq, Var("x"), IConst(10)),
      If(Call(And, Var("a0"), Var("b0")),
        IConst(2),
        IConst(0)))))
```

Sémantiquement parlant, il s'agit d'une `expr`ession représentant le programme fonctionnel suivant :

```
let x = 5 in
let a0 = 0 <= x in
let b0 = x <= 10 in
if a0 && b0 then 2 else 0
```

Dans les quatres sous-sections ci-dessous, on vise à implémenter quatre fonctionnalités différentes et essentiellement indépendantes :

1. Implémenter une fonctionnalité de simplification pour détecter et éliminer les "anti-patterns" du style `"if c1 then true else c2"`.
2. Implémenter une fonctionnalité pour "éliminer tous les `let`" et les réintroduire si nécessaire pour éviter les "recalculs".
3. Implémenter un interprète permettant d'exécuter un programme MLF.
4. Implémenter un *type-checker* permettant de vérifier le typage d'un programme MLF.

5.1° Élimination des "if booléens"

5) Définir la fonction *non récursive* `regle_if : expr -> expr` qui effectue les 3 simplifications suivantes sur une expression MLF si elle est de la forme `If(_, _, _)` et la laisse inchangée sinon :

- `if c then true else false` → `c`
- `if c1 then true else c2` → `c1 || c2`
- `if c1 then c2 else false` → `c1 && c2`

(ces simplifications étant ici exprimées "en pseudo-code" pour plus de lisibilité)

6) Définir la fonction `apply : (expr -> expr) -> expr -> expr` qui, étant données une règle de transformation `regle : expr -> expr` et une expression `e`, applique cette `regle` à toutes les sous-expressions de `e`, en commençant par les expressions les plus profondes et en terminant par l'expression de plus haut niveau.

Indication : noter que l'application partielle `apply regle_if` permet de résoudre le problème visé dans cette section.

5.2° Optimisation pour éviter les "recalculs"

7) Définir la fonction `forall : (expr -> bool) -> expr -> bool` qui, étant donné un prédicat sur les expressions puis une expression, vérifie que le prédicat est vrai sur l'expression elle-même et sur chacune de ses sous-expressions.

8) Définir le prédicat `no_let : expr -> bool` qui vérifie qu'une expression MLF ne contient aucun `Let`.

9) Définir la fonction `subst : expr -> expr -> expr -> expr` prenant en argument trois expressions `e0`, `e1`, `e` (en supposant que `e` ne contient pas de `Let`), et utilisant `apply` pour :

- parcourir toutes les sous-expressions de `e`,
- remplacer `e0` par `e1`,
- mais lever une exception avec `failwith` si un `Let` est rencontré.

10) Définir la fonction `subst_list : expr -> expr -> expr list -> expr list` prenant en argument deux expressions `e0`, `e1` et une liste d'expressions `l_e`, et substituant récursivement `e0` par `e1` dans chaque expression de la liste `l_e`.

Indication : La liste résultat devra préserver (le nombre et) l'ordre des éléments de la liste `l_e` passée en argument à `subst_list`.

11) Définir la fonction *non récursive* `regle_let : expr -> expr` qui effectue une simplification d'une epression MLF si elle est de la forme `Let(_, _, _)` et la laisse inchangée sinon :

- `let v = e1 in e` → remplacer la variable `v` par `e1` dans `e`.

(cette simplification étant ici exprimée "en pseudo-code" pour plus de lisibilité)

En déduire la fonction `elim_let : expr -> expr` qui applique la règle précédente sur toutes les sous-expressions, récursivement :

```
let elim_let = apply regle_let
```

12) Définir la fonction `ssexpr : expr -> expr list` qui, étant donnée une expression `e`, renvoie la liste de toutes ses sous-expressions, y compris `e`. L'ordre des éléments dans la liste résultat n'importe pas, en revanche il est important de renvoyer **toutes** les occurrences.

On suppose définie la fonction `gensym : unit -> string` générant un "symbole" (identificateur de type `string`) différent à chaque appel, et commençant par `_x`. Cette fonction, donnée dans le prélude, est par définition impérative (tandis que toutes les fonctions que vous coderez dans ce DM devront être pures).

13) Définir la fonction récursive `gen_lets : expr -> expr list -> expr` prenant en argument une expression `e` ne contenant pas de `Let` et une liste d'expressions `[e_1; e_2; ...]`, et renvoyant **une expression équivalente à `e`** où chaque expression individuelle `e_1` (etc.) a été remplacée dans `e` **et dans le reste de la liste** `[e_2; ...]` par des `Var(x_1)`, où `x_1` est un identificateur de variable généré par `gensym ()`.

Indication : vous aurez besoin de `subst` et `subst_list`.

14) Définir la fonction `optim_lets : expr -> expr` prenant en argument une expression quelconque `e` du langage MLF, et renvoyant **une expression équivalente à `e`** où les sous-expressions répétées sont factorisées.

Indications :

- la transformation de `e` sera effectuée après élimination des `Let` et énumération des sous-expressions. On ne prendra pas en compte les répétitions des `Var(_)`, `BConst(_)` et `IConst(_)` (puisqu'on ne gagnerait rien à les substituer par des `Let`…)
- Un exemple complet d'utilisation de la fonction `optim_lets` est donné en section 5.5°.

5.3° Implémentation d'un interprète MLF

On considère la signature de module suivante, paramétrée par un environnement `E : tENV` qui pourra être en pratique, soit le module `ENV_LIST`, soit le module `ENV_FUN` (tous deux compatibles avec la signature `tENV`) :

```
module type tENV = sig
  type ('x, 'v) env
  val empty : ('x, 'v) env
  val get : 'x -> ('x, 'v) env -> 'v
  val put : 'x -> 'v -> ('x, 'v) env -> ('x, 'v) env
end
type typ = Int | Bool  (* rappel *)

module type tEVAL = functor (E : tENV) -> sig
  type value = Vint of int | Vbool of bool
  val eval_op : binop -> value -> value -> value
  val eval : (string, value) E.env -> expr -> value
end
```

15) Définir un module paramétré `EVAL : tEVAL` implémentant un interprète du langage MLF, c'est-à-dire, fournissant une fonction `eval` qui calcule la valeur d'une expression de type `expr` "en allant chercher dans l'environnement" les valeurs des `Var(x)`.

Remarques :

- la fonction `eval_op` prend en argument un opérateur binaire de type `binop` et deux valeurs (entières ou booléennes),
- calcule le résultat si les valeurs sont compatibles en termes de typage,
- ou lève une exception avec `failwith` en cas d'impossibilité.

5.4° Implémentation d'un *type-checker* MLF

On considère la signature de module suivante, paramétrée par un environnement `E : tENV` :

```
module type tTYPECHECK = functor (E : tENV) -> sig
  val typeof_op : binop -> typ * typ * typ
  val typeof : (string, typ) E.env -> expr -> typ
end
```

16) Implémenter un module paramétré `TYPECHECK : tTYPECHECK` implémentant un vérificateur de type pour le langage MLF, c'est-à-dire, fournissant une fonction `typeof` qui renvoie `Int` (resp. `Bool`) si l'expression donnée en argument est bien typée conformément au type de ses variables renseigné dans l'environnement. Une exception sera levée avec `failwith` si l'expression est mal typée.

Remarques :

- la fonction `typeof_op` renvoie un triplet `(type_arg1,type_arg2,type_resultat)` indiquant le "profil" de l'opérateur binaire considéré,
- par exemple, `typeof_op Leq = (Int, Int, Bool)`.

5.5° Instanciation et test (non noté)

Si vous avez implémenté les sections 4°, 5.3° et 5.4°, vous pourrez faire :

```
module EL = EVAL(ENV_LIST)
module TCL = TYPECHECK(ENV_LIST) ;;
```

puis tester le résultat de `EL.eval` et `TCL.typeof` sur :

- `BConst(true)`
- `Let("x0", IConst(2), Var("0"))`
  et
- `Call(Add, BConst(true), BConst(false))`…

Enfin, si vous avez implémenté la section 5.2°, vous pouvez tester votre fonction `optim_lets` sur le cas de test suivant :

```
optim_lets
  ( Call(Add, Call(Add, Var("x"), Var("x")),
      Let("y", IConst(1),
```

```
        Call(Add, Call(Add, Var("x"), Var("x")), Var("y")))) )
=
Let ("_x1", Call (Add, Var "x", Var "x"),
  Call (Add, Var "_x1", Call (Add, Var "_x1", IConst 1)))
```

qui est donc un programe équivalent au programme en argument, mais optimisé vis-à-vis des `Let` (pour éviter de dupliquer les calculs).

6° Remarque sur le typage

Dans ce Devoir Maison, vous devez implémenter des modules et fonctions ayant exactement le type suivant :

```
val doublons : 'a list -> 'a list
val occs : 'a list -> ('a * int) list
module ENV_LIST : tENV_LIST
module ENV_FUN : tENV_FUN
val regle_if : expr -> expr
val apply : (expr -> expr) -> expr -> expr
val forall : (expr -> bool) -> expr -> bool
val no_let : expr -> bool
val subst : expr -> expr -> expr -> expr
val subst_list : expr -> expr -> expr list -> expr list
val regle_let : expr -> expr
val elim_let : expr -> expr
val ssexpr : expr -> expr list
val gen_lets : expr -> expr list -> expr
val optim_lets : expr -> expr
module EVAL : tEVAL
module TYPECHECK : tTYPECHECK
```

L'implémentation des tests unitaires PFITAXEL associés au Devoir Maison prend en compte ce typage. Autrement dit, n'hésitez pas à utiliser la fonctionnalité `Noter!` de PFITAXEL pour vérifier ces types.

7° Rappels

Concernant le contenu de votre devoir :

- Vous devez commencer par définir votre nom et votre prenom (sans accent) de la façon suivante :

  ```
  let nom = "DUPONT" and prenom = "John";;
  ```

- Vous devez **utiliser uniquement le fragment fonctionnel d'OCaml** (pas de boucle `while` ou `for`, de séquence d'instructions "`;`" ou d'autres structures de contrôle impératives).
- L'utilisation des fonctions de la bibliothèque standard des listes (`List.`) est autorisée. Vous pouvez notamment utiliser l'opérateur de concaténation `( @ )`, et définir vos propres fonctions auxiliaires.

Concernant les modalités de rendu :

- Ce devoir maison est un devoir personnel et par conséquent **tout échange de code entre vous est formellement interdit et serait sanctionné**.

- Après avoir terminé votre devoir, **vous devez vous assurer que votre code ne comporte aucune erreur de syntaxe ou de typage** (c'est la moindre des choses pour un tel rendu, si vous ne vérifiez pas cela vous risquez d'obtenir la note 0/20).
- Le rendu de votre devoir maison doit être effectué **sur PFITAXEL**.