TP 1 : Le langage Ouf!

30 mai 2019 - IFT 2035

Vous devez faire le TP en équipe de deux.
Vous devez remettre votre travail pour le dimanche 23 juin 23h59 (2019).

1 Ouf! un langage qui rend fou

Le langage Ouf! est un langage qui permet aux programmeurs de changer d'idée en tout temps. Après tout, il n'y a que les fous[1] qui ne changent pas d'idée.

Le langage permet donc de changer *lors de l'évaluation du programme* le type de portée des variables et l'immuabilité ou non des variables.

Votre travail consiste à écrire un interpréteur pour le langage Ouf!. La donnée de ce TP1 va d'abord vous présenter plus en détail le langage Ouf! et ensuite définir le travail que vous aurez à faire.

1.1 Syntaxe de Ouf!

La syntaxe de Ouf! est définie par la grammaire à la page 2. Elle ressemble beaucoup à la syntaxe de LISP et Scheme puisqu'elle est composée de S-expressions. Il s'agit d'une syntaxe *préfixe* avec parenthèses. Voici des exemples de S-expression :

```
(+ 1 2)

(lambda (x) x)

(let ((x 5)
      (y 8))
 (+ x y))
```

Attention les espaces sont importants, c'est-à-dire que `(+1 1)` n'est pas égal à `(+ 1 1)`. Dans le premier cas, il s'agit d'une S-expression avec deux éléments dont le premier est la variable qui a pour identifiant `+1`, identifiant qui serait invalide en Ouf!.

Globalement la syntaxe est assez simple. Un fichier `.ouf` contient soit des expressions, des définitions ou des déclarations de types. Les types commencent par une majuscule

1. fou se dit ouf en verlan

Fig. 1 : La syntaxe du langage Ouf!.

$$
\begin{aligned}
\text{toplevel} &::= (\text{expr} \mid \text{datatype} \mid \text{definition})^{+} \\
\text{definition} &::= \texttt{(define } \text{variable expr}\texttt{)} \\
\text{datatype} &::= \texttt{(data } \text{type constructeur}^{+}\texttt{)} \\
\text{constructeur} &::= \texttt{(}\text{type}^{+}\texttt{)} \\
\text{type} &::= [A-Z]([a-z]|[A-Z]|[0-9])^{+} \\
\text{variable} &::= [a-z]([a-z]|[A-Z]|[0-9])^{+} \\
\text{entier} &::= [0-9]^{+} \\
\text{expr} &::= \text{variable} \\
&\quad \mid \text{type} \\
&\quad \mid \text{entier} \\
&\quad \mid \texttt{((}\text{expr} \mid \text{type}\texttt{) expr}^{*}\texttt{)} \\
&\quad \mid \texttt{(lambda (}\text{variable}^{+}\texttt{) expr)} \\
&\quad \mid \texttt{(set } \text{variable expr}\texttt{)} \\
&\quad \mid \texttt{(let ((}\text{variable expr}\texttt{)}^{+} \text{ expr}\texttt{)} \\
&\quad \mid \texttt{(case } \text{expr case-branch}^{+}\texttt{)} \\
&\quad \mid \texttt{(ouf scope } \text{variable } (\texttt{lexical}|\texttt{dynamic})\texttt{)} \\
&\quad \mid \texttt{(ouf mutability } \text{variable } (\texttt{constant}|\texttt{variable}) \\
\text{case-branch} &::= \texttt{(}\text{constructeur expr}\texttt{)}
\end{aligned}
$$

et les variables par une minuscule. Ouf! possède des fonctions anonymes (`lambda`), des déclarations locales (`let`) et un filtrage par motif rudimentaire (`case`).

La particularité du langage Ouf! réside dans les expressions `ouf` qui modifient la portée et l'immuabilité des variables.

Notez que les mots clés de la syntaxe sont réservés par le langage et ne peuvent pas être utilisés comme identificateur. Aussi, il est possible d'inclure des commentaires. Tous les caractères après `--` sont ignorés jusqu'à la fin de la ligne.

1.2 Sémantique de Ouf!

Ouf! est un langage typé dynamiquement avec une évaluation stricte. C'est à dire que chaque déclaration ou expression est immédiatement calculée. Ceci est important notamment pour les expressions `ouf` prennent effets immédiatement. Le typage dynamique est abordée plus en détail dans la section sur le `case`.

Le fichier unittests.ouf fournit beaucoup d'exemples de la sémantique de ouf. Vous pouvez y faire référence en lisant cette section. Le contenu du fichier est également disponible à la fin de ce document (page 5).

1.2.1 Toplevel

Il y a trois possibilité au niveau global.

Définition Une définition sert uniquement à associer la valeur d'une expression avec une variable. *Par défaut la portée est statique et la variable est immuable.*

Datatype Il est possible de déclarer de nouvelles structures de données au niveau global uniquement.

Expression Il est possible qu'une expression se situe au niveau global.

1.3 Datatype

Les datatypes sont comme en Haskell. Le premier identificateur de la définition donne le nom du type. Chaque déclaration comprend possiblement plusieurs constructeurs. Pour chaque constructeur il faut spécifier le type de ces arguments. Les datatypes peuvent être récursifs.

1.4 Let

Les définitions locales d'un `let` sont *mutuellement* récursives. *Par défaut la portée est statique et la variable est immuable.* Ces définitions ne doivent pas être visible à l'extérieur du `let`. Les définitions s'évaluent dans l'ordre, de la première à la dernière. Puisque Ouf! est un langage strict, la récursion mutuelle sert principalement à écrire des fonctions mutuellement récursives.

Une expression `ouf` à l'intérieur d'un `let` n'a pas d'effet à l'extérieur du let.

1.5 Case

Le case évalue l'expression qui doit être filtrée par motif. Si cela ne correspond pas à un objet créé par `data` il s'agit d'une erreur. Donc il est impossible de faire un `case` sur un nombre ou une fonction. Ensuite le case essaie la première branche. Si les constructeurs sont les mêmes alors l'expression associée à la première branche est exécutée.

Si les constructeurs appartiennent à des types différents cela est une erreur. Aussi, s'il n'y a pas le bon nombre d'arguments ceci est aussi une erreur.

Si le constructeur de la première branche n'est pas le bon, on passe alors à la deuxième branche et ainsi de suite. Si aucune branche ne fonctionne, il faut signaler une erreur.

Noter que votre évaluateur vérifie uniquement le type à l'exécution et donc que les branches non exécutées ne génère pas des erreurs de types.

1.6 Ouf

Les expressions `ouf` permettent de changer les propriétés identificateurs et des variables.

Il faut clarifier que l'instruction `ouf` pour la portée s'applique en fait à tous les identificateurs rencontrés dans le code après cette instruction. Ainsi toutes les variables ayant ce même identificateur sont affectées.

Par contre, l'instruction `ouf` pour l'immuabilité s'applique à une variable. Ainsi, même si plusieurs variables ont le même identificateur, seulement une seule verra sa propriété modifiée. Bien sûr, ce sont les règles de portée en vigueur pour l'identificateur qui décidera de quelle variable il s'agit.

La subtilité est que les expressions `ouf` qui sont exécutées au niveau global on un effet global. Par contre, exécutée comme définition dans un `let` ou dans le corps du `let` elles n'ont effet que dans ce `let`. Les propriétés antérieurs (portée et mutabilité) doivent êtres restorées. Oui c'est Ouf!.

Pour ce qui est des expressions `ouf` qui sont dans le corps d'une fonction, cela dépend de si la fonction est exécuté au niveau global (appel directement fait au niveau global) ou si elle est exécutée dans un `let`.

1.7 Set

Les expressions `set` change la valeur d'une variable. Cela affecte aussi les fermetures.

2 Votre travail

La boucle principale de l'évaluateur (Main.hs) ainsi que le parseur (Parseur.hs) vous sont fournis. Vous n'avez pas à modifier ces fichiers.

Concrètement votre travail consiste à complèter les fonctions du fichier Eval.hs. Principalement `sexp2Exp`, `evalGlobal` et `eval` ainsi que les fonctions de support utilisées par les trois mentionnées ci-haut. Le fichier Eval.hs contient assez de commentaires pour vous guider.

Le code fournit dans Eval.hs est bien adapté pour un évaluateur avec la portée lexicale et des variables immuables. Toutefois, vous allez devoir adapter certaines choses (comme le type de l'environnement) au fur et à mesure que vous progressez dans votre travail.

Je vous conseille fortement d'implanter les fonctionnalités de votre évaluateurs dans l'ordre des tests unitaires.

3 Évaluation

L'évaluation du TP 1 se base sur les tests unitaires et votre rapport. Le rapport vaut 3 points et les tests unitaires 17 points. Le fichier unittests.ouf (page 5) indique la répartition des points en fonctions des tests réussis.

Pour éviter un évaluateur fait sur mesure pour les tests unitaires, d'autres tests unitaires seront ajoutés aux sections existantes un peu avant la date limite du TP. Cela n'affectera pas le nombre de points pour chaque section de tests unitaires.

Vous devez remettre votre fichier Eval.hs et le rapport en PDF. Votre fichier Eval.hs doit compiler avec les fichiers Main.hs et Parseur.hs fournis. Si votre fichier ne compile pas, le correcteur ne peut pas faire les tests unitaires, vous perdez donc potentiellement 17 points.

4 Rapport

Vous devez fournir un rapport d'au moins 3 pages (excluant la page titre). Votre rapport doit expliquer votre expérience de développement avec le TP 1 et la structure globale de votre évaluateur. Cette partie vaut 3 points.

Si votre évaluateur réussi tous les tests unitaires, votre rapport peut possiblement comporter 3 pages. Par contre, si vous échouez certains tests vous avez la chance dans votre rapport d'expliquer selon vous l'algorithme qui devrait être implanté dans votre évaluateur. Si la description de votre algorithme est juste, vous pouvez récupérer jusqu'à la moitié des points accordés pour ces tests.

5 Fichier unittests.ouf

```
-- Fichier pour les tests unitaires

-- Les définitions et expressions sont lues et exécutées dans l'ordre.
-- L'environnement global est possiblement modifié
-- à chaque définition / expression

-- La forme spéciale :check-equal : prend deux Sexp, les évalue et les compare
--   Chaque Sexp est exécuté avec le même environnement global

-- La forme spéciale :check-error : prend une Sexp et s'attend à recevoir une
-- erreur
```

```
-- Les formes spéciales ne modifient *PAS* l'environnement global

-- ********** Quelques définitions utiles **********
(define x 4)
(define y 2)

-- ********** Test les variables et primitives *********
(:check-equal : x 4)
(:check-equal : y 2)
(:check-equal : (+ x y) 6)
(:check-equal : (* x y) 8)

-- ********* Test les fonctions *********
(define foo (lambda (y) y))
(define bar (lambda (a b c) a))

(:check-equal : (foo 1) 1)
(:check-equal : (bar 4 5 6) 4)
(:check-equal : (((bar 4) 5) 6) 4)
(:check-error : (lambda () 2))

-- ********* Test le Let (non mutuellement récursif) *********
(:check-equal : (let ((a 1)) a) 1)
(:check-equal : (let () 1) 1)
(:check-error : (let 1))

-- Les définitions du let ne doivent pas être visible
-- hors du let
(let ((a 1)) a)
(:check-error : a)

-- test une définition locale de x et y
(:check-equal : (let ((x 3) (y 7)) (+ x y)) 10)

-- test qu'on voit toujours la définition de x
(:check-equal : (let ((a x) (y 7)) (+ x y)) 11)

-- Les clauses d'un let sont évaluées dans l'ordre
(:check-error : (let ((a b) (b 4)) a))

-- Les clauses d'un let sont évaluées dans l'ordre, sauf que ici
-- le x doit cacher le x global et donc ceci retourne une erreur
(:check-error : (let ((a x) (x 0)) a))
```

```
-- ********** Fermeture **********
-- Le v dans le let doit être mémorisé par la fonction
-- Mais il n'est plus accessible
(define y (let ((v 1)) (lambda (x) v)))
(:check-equal : (y 0) 1)
(y 0)
(:check-error : v)

-- ********** Test les définitions Data **********
(data Bool (True) (False))
(data Maybe (Nothing) (Just Int))
(data List (Nil) (Cons Int List))

-- Deux objets fait des mêmes constructeurs avec les mêmes arguments sont
-- identiques
(:check-equal : True True)
(:check-equal : False False)
(:check-equal : (Cons 3 Nil) (Cons 3 Nil))
(:check-equal : Nothing Nothing)
(:check-equal : (Cons 3 (Cons 2 Nil)) (let ((a (Cons 2 Nil)) (b (Cons 3 a))) b))

-- Fournir des arguments en trop à un construteur est une erreur
(:check-error : (True 2))
(:check-error : (Cons 2 4 Nil))

-- Il est possible de faire du currying avec un constructeur
(:check-equal : (let ((a (Cons 2))) (a Nil))
                (Cons 2 Nil))

-- ********** Test sur case **********
(define l (Cons 2 (Cons 3 Nil)))
(define if (lambda (test iftrue iffalse) (case test (((True) iftrue) ((False) iffalse)))))

(:check-equal : (case l (((Cons a b) a))) 2)
(:check-equal : (if True 1 0) 1)

-- Le nombre de variables du case doit être en lien avec l'objet
(:check-error : (case (Cons 3 Nil) (((Cons a b c) 0))))
(:check-error : (case (Cons 3 Nil) (((Cons a) 0))))

-- Case ne peut pas être fait sur un nombre ou une fonction
(:check-error : (case + (((Nil) 0))))
(:check-error : (case 24 (((24) 0))))
```

```
-- ********** Test portée dynamique *********
(define myX (lambda (y) x)) -- Fait référence au x du tout début

(ouf scope x dynamic)

-- Pour l'instant les deux portées donnent le même résultat
(:check-equal : x 4)

-- Test de portée dynamique
(:check-equal : (let ((x 0)) (myX 0)) 0)

-- Ici zz est capté en portée lexicale,
-- mais en porté dynamique il n'y aura plus de zz de défini
-- après le let
(ouf scope zz dynamic)
(define closure (let ((zz 42)) (lambda (u) zz)))

(:check-error : (closure 0))

(ouf scope zz lexical)
(:check-equal : (closure 0) 42)


-- ********** Test sur mutabilité (ouf et set) **********

-- Par défaut tout est immuable
(:check-error : (set x 0))
(:check-equal : (myX 0) 4)

-- Test ouf mutable
(ouf mutability x variable)
(set x 0)
(:check-equal : x 0)

-- ********** Test mutabilité pour les fermetures **********
(:check-equal : (myX 0) 0)
```