Rapport du TP 1 :
Le langage Ouf

Par
Steven Han, 20129462

Département d'informatique et de recherche opérationnelle

Travail présenté à Érick Raelijohn
Dans le cadre du cours IFT-2035
section A Concepts des langages de programmation

23 Juin 2019

0. Conseil :

On vous conseille fortement de suivre ce document avec le code des tests unitaires et le fichier eval.hs à côté de celui-ci. Les explications seront dans l'ordre des tests unitaires et non l'ordre du code. Aussi, on ne parlera pas des fonctions ajoutées à l'exception de *exp2Val*, car ils ne sont pas pertinents et il y a des commentaires pour vous guider.

1. Fonctionnement du code :

L'ordre d'appel des méthodes est comme suit : le fichier de tests appel soit la fonction *sexp2Exp* pour les formes simples d'expressions (comme les opérateurs mathématiques et la vérification de variable) ou soit la fonction *specialForm2Exp.* Il gérera entre autres les « *lamba, let, case, data, define, set, ouf »*. Peu importe la fonction appelé (*sexp2Exp* ou *specialForm2Exp*), le retour devra être un *Either Error Exp* (soit une erreur soit une expression) où l'expression devra contenir les bons arguments avec le bon datatype. Lorsque c'est fait, la fonction evalGlobal (pour les *define* et *data*) ou la eval (pour le reste) est appelé. Si le bon datatype d'expression avec les bons arguments est retourné, alors il ne devrait pas avoir d'erreur lorsque les tests unitaires se feront.

1.1 Déroulement de la fonction sexp2Exp :

La partie qui manquait était celui où l'argument de la fonction est une liste de simple expression. Chaque ligne est coupée selon les espaces. Le paramètre de la fonction à coder (*SList (func :args))* faisait en sorte qu'on pouvait accéder au premier argument de la ligne d'un fichier *ouf* et le reste de la ligne. Par exemple si on a la ligne *(:check-equal: x 4)*, *func* = x et *args* = [4]. N'oubliez pas que cette fonction ne gère que les cas de bases (vérification de variables et opérateurs mathématiques). Pour se faciliter la tâche, on fera un case sur *func* découper les cas.

1.2 Déroulement de la fonction specialForm2Exp :

Nous avons remarqué qu'au départ, l'interprète ne reconnaît absolument aucun mot. Donc, on devait implémenter quelque part dans le code où lorsque l'interprète lit la séquence de mot *define,* par exemple, il devait agir d'une certaine façon. Le paramètre de la fonction ([Sexp]) est en fait une coupure de la ligne en question selon les espaces et les parenthèses. Pour tous les cas, le premier élément sera un symbole de type SSym identifiant l'expression (lambda, let, case, data, define, set et ouf). Le reste sera les arguments à traiter selon son datatype. Il est aussi à noter qu'il se fera aussi appeler lors des tests unitaires. Par exemple, si on a *(:check-equal: (let ((x 3) (y 7)) (+ x y)) 10)*, *func* = let et *args* = [(x 3) (y 7), (+ x y)]. Il n'appellera pas la fonction *sexp2Exp*. Il est à noter que certaine fonction se répète à l'exception d'un [] au lieu d'une parenthèse. Voir le code pour les détails.

1.3 Déroulement de la fonction exp2Val (ajouté) :

On a ajouté cette fonction, car on trouvait cela plus simple de convertir seulement une expression en sa valeur, sans jouer avec l'environnement. Le retour de la fonction *eval* (Either Error (Env, Value) causait problème à certains endroits et rendait le codage plus difficile.

Par exemple, si on avait *define x 4* et qu'on voulait transformer l'expression EInt 4 en VInt 4. On pourrait utiliser la fonction *eval* qui nous retournerait le même type de retour que *evalGlobal*. Il ne reste qu'à l'insérer dans l'environnement, mais on a besoin de sa valeur. Et sa valeur est dans le retour de la fonction *evalGlobal* (Environnement, Valeur). Cependant, extraire la variable du couple (Environnement, Valeur) est une tâche de plus et n'est pas évidente.

Cette fonction prend en argument une expression et un environnement et le converti en datatype *Value*. Par exemple EInt = VInt, ELam = VLam, etc.

2. Section Variables, fonctions et primitives :

2.1 Explication du define:

Le *define* est assez simple : chaque ligne aura la forme *define variable* (*expression*). Donc, il fallait juste intégrer à la fonction *specialForm2Exp* un cas où il reconnaît le mot « define ». Ensuite, le premier argument du reste de la liste est en fait la variable et ce qui suit est un tableau de simple expression. On appellera la fonction *var2Symbol*, pour convertir notre variable (premier argument) de type Sexp en SSym. Il ne reste qu'à traduire le tableau de simple expression. On réappellera la fonction *specialForm2Exp* pour traiter les cas de lambda et let. Il y a aussi un cas où on a seulement une simple expression comme *define x 4*.

Une fois le bon retour envoyé, la fonction *evalGlobal* sera appelé. Il suffit de l'insérer dans notre environnement avec la fonction *insertVar* et de retourner le nouvel environnement (par la fonction *insertVar*) et la valeur de la variable.

2.2 Explication des opérateurs mathématiques :

Remarque : Notre code supporte seulement les opérateurs avec deux arguments.

Pour les opérateurs mathématiques, c'est la fonction *sexp2Exp* qui se fera appeler. Il sera de type SList et coïncide avec la fonction que nous devions coder. Le truc est que l'argument *func* sera le premier élément de la liste, soit les opérateurs (+, - *). Et les arguments seront les variables où nous devrons faire les opérations. Donc, il suffit de mettre dans le *case* de *func* les opérateurs (+, -, *). Ensuite, on transformera nos deux premiers arguments de *args* en symbole avec la fonction *var2Symbol*. Lorsque fait, on retournera une expression de type

EApp (*Evar symbolOpérateur*)(*EApp* (*EVar variable*1)(*EVar variable*2))

Ensuite, la fonction *eval* se fera appeler. Il suffit de faire un *case* pour le premier argument de EApp sur les opérateurs mathématiques. Aussi, arg sera en fait une expression de type EApp et on appellera une fonction spéciale (*eApp2Var)* pour traduire chacun des arguments en symbole pour pouvoir chercher la valeur de la variable. Pour cela, on utilisera la fonction *lookupVar* et on additionne, soustrait ou multiplie les deux variables selon le cas et dans l'ordre de parution.

2.3 Explication du lambda :

Remarque : On assume que le lambda peut seulement retourner une des valeurs de ces arguments et ne peut pas faire des opérations sur les arguments.

Le lambda est une fonction anonyme qui prend des arguments (comme une fonction normale) et une expression pour déduire son comportement. Par exemple la fonction anonyme \x -> x+1 dit qu'il prend un Int en argument et retourne cette valeur incrémentée de 1. Dans notre cas, si on fait *define foo (lambda (y) y)*, cela veut dire que si on passe un argument à *foo* (foo 1, par exemple), alors foo aura comme valeur l'argument (1). On peut seulement utiliser le lambda lors des déclarations de variables, car on ne peut pas utiliser directement une fonction anonyme puisqu'elle n'a pas de nom.

Le cas du lambda dans la fonction *specialForm2Exp* a déjà été codé. Cependant, lorsque utilisé, il retourne une série de ELam imbriqué. Par exemple, si on a *lambda (a b c) a*, l'expression sera *ELam a (ELam b (ELam c (EVar a)))*. Le *EVar* indique quel symbol donnera la réponse de la fonction annonyme. Donc, pour l'implémenter, il faut retourner un *VLam symbol exp env* où le symbol est le symbol de la valeur (dans l'exemple, ce serait a) et exp est l'expression de tous les lambda imbriqués. La fonction *checkArgEqual* retourne la valeur des arguments passé par le programmeur. Les arguments du programmeur sont des EApp imbriqués. Par exemple si on a *bar 4 5 6* alors l'expression sera (EApp (EInt 4) (EApp (EInt 5) (EInt 6))). L'idée est que le premier symbol du ELam imbriqué correspond au premier argument du programmeur (qui est sous forme de EInt). Si le symbol correspond à celui recherché, alors le premier argument du EApp est celui qu'on cherche. Sinon, on accède au deuxième argument du lambda et le deuxième argument du EApp (qui est un EApp).

3. Section Let :

Lorsque l'interprète voit le mot *let*, la fonction *specialForm2Exp* est appelé, comme mentionné dans la partie *fonctionnement du code*. On fera un cas où il n'y a pas de déclaration de variable, mais une expression (par exemple *let () 1*). Dans ce cas-là on appellera la fonction *sexp2Exp* pour retourner la valeur de l'expression.

Si c'est un let avec plusieurs variables déclarées, il faudra prendre tous les arguments (sous forme de SList, dans notre code c'est *params*) et retourner un (ELet ([déclarationVariables]) exp). Les arguments sont de type [Sexp] et il fallait les convertir en [(symbol, exp)]. Prenons cet exemple : *let* : *let ((x 3) (y 7)) 2*. [Sexp] sera égale à [(x 3), (y 7)]. Pour convertir nos variables, on utilisera la fonction *joinSymEtExp* (joindre les symbols et les expressions) qui prend le premier argument de la liste et le converti. Puis, on utilisera la récursion pour faire les autres déclarations de variables.

Lorsque toutes les variables sont de la bonne forme, on appellera la fonction *insertVars* qui insérera nos variables dans l'environnement et retournera un nouvel environnement.

Dans nos tests, la 15^{e} n'est pas bonne (les définitions du let ne doivent pas être visible hors du let). Je dois vous avouer que c'est un mystère, car nos variables sont bien locales (le test de la définition locale le prouve et la section fermeture aussi).

4. Section fermetures :

Nos variables sont bien locales, mais il y a une erreur au test 18 à cause du lambda. Notre lambda est codé de façon à ce que s'il y a un argument, alors la valeur du lambda sera cet argument. Or, la valeur du lambda est v et non x, mais le code pense que c'est x.

5. Section Data et Case :

5.1 Data

Le data était un vrai casse-tête. Commençons dans la fonction *specialForm2Exp*. Il fallait couper notre liste en sous liste : le premier argument est en fait le mot « data » et le reste contient le nom du data ainsi que ses types. Donc, il fallait encore séparer le nom du data de ses déclarations. Par exemple, voici la structure si on a comme ligne *(data Bool (True) (False))* : [(SSym « data ») : [Bool : [True, False]]]. Puis, il fallait convertir les déclarations en une liste de type « DataConstructor ».

L'idée est que le « Constructor » sera toujours le nom du data (dans l'exemple en haut ce serait 'Bool »). Ensuite, il fallait transformer notre liste de Sexp en liste de types. Mais comme « Constructor », « Type » sont des symbols, il fallait les convertir en symbol. Pour ce faire, on prend chaque élément de notre liste de Sexp et on le converti en symbol avec la fonction *tabSexp2TabType*. Cette fonction utilisera la récursion pour transformer tous les éléments du tableau. Le reste n'est que de l'assemblage de symbol, la grosse partie est terminé.

Dans notre code, les déclarations des constructeurs sont bel et bien en mémoire (les tests que deux objets fait des mêmes constructeurs avec les mêmes arguments son identique le prouvent).

5.2 Case

Il y a une erreur lors de la comparaison entre une expression et un « CasePattern ». Le fait est que le « CasePattern » est un plus gros casse-tête que le data et on ne savait pas trop comment l'implémenter. Certains sont bon, certains sont faux. Tous ce qu'on a fait est de retourner la valeur de l'expression sans tenir compte du « CasePattern »

Ce qu'il fallait faire c'est de transformer le premier argument du *case* en EData. Ensuite, le deuxième argument sera le « CasePattern ». On mettra comme symbol le constructeur, le champ du tableau des symbols sera les types et l'expression sera un EData. Par exemple, si on a *case (Cons 3 Nil) (((Cons a b c) 0)).* Alors il faudra retourner un
ECase (*EData* (*List* [(*Nil*, *Cons Int List*)]) [(*List*, [*Cons* 3 *Nil*, *Cons a b c* 0], *EData*)]

6. Section Set :

La section set était assez simple. Il fallait juste regarder si la variable est dans l'environnement. Si elle l'est, on la met à jour avec la nouvelle valeur et on retourne le nouvel environnement. Sinon, on retourne une erreur. Il y a quelques erreurs dû au fait que nous avons très peu touché à la partie ouf.

Conclusion :

Notre programme marche très bien avant la section Case à l'exception de quelques erreurs (1 dans le let et 1 dans la fermeture). Pour ce qui est de Data, notre code les met bel et bien en mémoire (le test des mêmes constructeurs le prouve). Pour le set, il marche, mais il y a des erreurs, car les tests mélangent la partie ouf et set. Pour ce qui est de la partie ouf et case, nous n'avons pas réussi, car c'est trop ouf !