**Structure du programme**

Après avoir regardé rapidement les tests unitaires, nous avons déterminé que nous allions suivre l’ordre des tests pour construire l’évaluateur *Ouf*! La première étape était donc les entrées *define* utilisées pour stocker des variables et des fonctions dans l’environnement. Pour la traduction du langage *Ouf*! vers une forme évaluable, nous avons ajouté trois cas dans la fonction *specialForm2Exp*. Si *define* n’est pas suivi d’argument, la fonction renvoie une erreur. Si la fonction reçoie un nom de variable et un nombre, on transforme la variable en symbole et le nombre en expression évaluable. Enfin, si la fonction reçoit en paramètre une autre fonction, tel qu’une fonction lambda par exemple, un appel récursif se passe jusqu’à ce que la dernière fonction soit traduite pour l’évaluation. Pour la partie d’évaluation des fonctions *define*, nous utilisons tout simplement la fonction *insertVar* pour ajouter la variable à l’environnement.

La deuxième étape de l’évaluateur consiste à lire des entrées de primitives (+, -, \*) en préfixe et de les évaluer. Pour la traduction, nous utilisons la fonction *sexp2Exp* qui vérifie que le nombre d’arguments est suffisant et transforme l’équation en *EApp* en fonction du nombre et du type d’arguments passés. Pour l’évaluation des *EApp*, nous séparons l’évaluation en deux cas, un pour les primitives et l’autre pour les fonctions lambda.

Ensuite, pour les expressions lambda, nous effectuons la traduction en *ELam* grâce à un appel récursif de *specialForm2Exp* implanté dans le cas du *define* afin de passer au cas du *lambda*. Pour l’évaluation, nous transformons le corps de la fonction lambda en *VLam* et nous retournons sa valeur et son environnement.

Quant aux expressions *let*, nous appliquons une fonction anonyme dans un *map* sur toutes les variables passées en argument, ce qui nous permet de séparer les éléments importants pour les transformer en *Symbol* et en *Exp* qui seront utilisés pour l’évaluation. On utilise ensuite la fonction *sequence* pour regrouper tout ceux-ci dans une même liste. On construit donc le *ELet* avec la liste de *Symbol* et d’*Exp* avec la dernière expression du *let*, qui est la fonction qu’on cherche à faire. Pour l’évaluation, nous séparons encore une fois les éléments du *ELet* avec une fonction lambda dans un *map* afin de faire des nouveaux tuples de *Symbol* et d’*Exp*. Nous mettons le tout dans un environnement fermé afin de ne pas pouvoir accéder aux variables depuis l’extérieur du let.

La fermeture fonctionne comme prévu pour les *let*, les variables sont mémorisées au sein de la fonction mais ne sont ni accessible par les autres fonctions, ni accessible au niveau global.

Pour la traduction des *data*, nous séparons les arguments en 3 parties, le nom du type de *data*, et les deux constructeurs. Nous avons ajouté une fonction *dataConst*, qui nous permet de renvoyer des *DataConstructor* à partir d’une *Sexp*. Nous utilisons cette fonction sur les deux constructeurs et nous les concaténons pour les renvoyer avec le nom du type de *data* dans un *EData* pour l’évaluation. Pour l’évaluation d’un *data*, nous avons ajouté une fonction *addTypeToEnv*, qui ajoute les types de *data* dans l’environnement. Ainsi, nous n’avons qu’à utiliser *map* sur les éléments du *EData* avec cette fonction et nous renvoyons ainsi un environnement contenant tous les types et les *data.*

**Expérience de développement**

***Difficultés***

Nous avons eu de nombreux problèmes lors de la conception de cet évaluateur.

***Fonctions manquantes***

Nous n’avons pas eu le temps de compléter les fonctions *case* et la portée dynamique.

Pour la fonction *case*, notre algorithme ressemblerait à :

Pour la portée dynamique, nous ferions :

***Globale***

Dans l’ensemble, nous sommes satisfaits du résultat obtenu. Nous avons passé beaucoup de temps à essayer de comprendre l’ensemble des fonctions du programme et nous avons fini par résoudre la plupart de nos problèmes. Nous n’avons pas eu le temps de terminer l’ensemble du programme car nous avons passé beaucoup de temps sur la fonction *data.*