Livrable 4 : Projet informatique

Rimbaud Jaoua

Introduction :

Dans ce quatrième livrable , on a travaillé sur les agrégats , les formes begin , lambda et let.
On a écrit les jeux de tests qui couvrent les fonctionnalités attendues et qui
permettent de tester les formes mentionnées précédemment.
En outre , on a implémenté le prédicat procedure ?.

Implémentation :

1) La forme begin :

Cette forme permet d'évaluer toutes les instructions et retourne le résultat d'évaluation de la dernière S-expression.
Son implémentation revient alors à évaluer la liste de ses arguments et à retourner au final le résultat de la dernière évaluation.
Basiquement cette fonction est déjà implémentée, elle correspond à la fonction permettant d'évaluer tous les arguments d'une primitive : on recycle.

2) La forme lambda :

Cette forme est équivalente à une fonction anonyme à laquelle on associe un ou plusieurs paramètres et un corps.
Son implémentation consiste à créer un environnement et à y exécuter le corps de la fonction anonyme.
C'est pourquoi,on utilise un agrégat afin de décrire cette forme.
On a implémenté pour l'occasion les agrégats. Ces nouveaux types possédent leur propre fonction d'évaluation : sfs_eval_compound. Dans cette fonction on créer un nouvel environnement dans lequel on crée les variables voulues puis on évalue le corps de la fonction.
Nous avons du modifier sfs_eval pour qu'il support les agrégats et l'évaluation d'une fonction anonyme.

3) La forme let :

La forme let a un fonctionnement proche de lambda mais son implémentation dans notre code est très différente.
Dans on premier temps on vérifie que l'entrée est correcte.
Sur une entrée type (let ((x 4) (y 5)) (+ x y)) on modifie la structure interne de la liste pour obtenir la chose suivante :
(let ((define x 4) (define y 5) (begin (+ x y))).
A partir d'ici on évalue nos define et notre begin pour faire fonctionner le let.
Cet évaluation se fait dans un nouvel environnement.

4) La forme map :

Nous n'avons pas implémenté map, cependant celle-ci est implémentable en scheme dans notre interpréteur et elle est fonctionnelle. De ce fait nous considérons qu'elle est implémentée et fonctionnelle.

5) Autres :

Nous avons réussi dans notre interpréteur à implémenter avec succés des fonctions récursives comme la factorielle. On a à l'occasion constaté que nos opérateurs arithmétiques ne controlent pas l'overflow.
Nous avons également pu définir la fonction count qui est elle aussi opérationnelle.

Autres changement :

Nous avons corrigé de nombreux bug, car cdr set-car ! Set-cdr ! Supporte désormais des atomes et ne provoquent plus de segfault.

Define et set ! Ne renvoient rien. Nous avons utilisé le type SFS_UNKNOWN qui est retourné par define et set !. Un atome de ce type n'affiche rien dans le shell. Dans une liste un atome de ce type apparaît sous la forme : #undef.
Notre if était également faux, nous l'avons corrigé, il faisait indifféremment l'évaluation de tous les arguments. Désormais il évalue l'alternative ou la conséquence selon le résultat de l'évaluation du prédicat.

Nous avons corrigé notre fonction or. Elle fonctionnait pour de mauvaises raisons nous l'avons donc mise à jour.

Nous avons corrigé les prédicats qui désormais ne supporte plus qu'un seul argument. Auparavant ils interprétaient par défaut le premier argument et ce quelque soit leur nombre, aujourd'hui ils renvoient un warning message et retourne NULL si jamais il y a trop d'arguments.

La fonction sfs_eval ne supporte plus une liste qui ne possède pas de symbole comme premier élément. Elle affiche un warning et renvoie NULL. (Il était possible d'écrire : (1 2) maintenant il faut écrire (list 1 2) ou '(1 2)). Nous sommes ainsi plus fidèle au langage scheme.

Grosse modification sur la libération de la mémoire. Elle a été intégralement supprimée. Aujourd'hui il n'y a plus de libération de mémoire. Define ne copie plus l'entrée de l'utilisateur de ce fait il n'y a plus de problème de free. On a donc aisin des variables qui peuvent valoir le même objet. Ainsi :
(define x '(4 5)))
(define y x)
(set-car ! X 34)
y
⇒ (34 5)
Fonctionne correctement on a bien x et y qui ont le même objet. En revanche la fonction set ! Elle effectue bien un copie ce qui donne des objets distincts.

Amélioration :

Nous pouvons désormais modifier l'interpréteur pour qu'il exécute des librairies lors de son démarrage pour définir des formes et fonctions (map ou autres). Actuellement cette fonctionnalité n'existe pas mais l'implémenter permettrai d'ajouter rapidement de nombreuses fonctionnalités.

Nous pouvons améliorer la gestion des erreurs.

Nous n'avons pas implémenté let* cependant il est implémentable rapidement avec LET_check et LET_eval. Il suffit juste de ne pas créer de nouvel environnement et de l'ajouter à l'ensemble des formes lisibles par notre interpréteur.

Nous n'avons pas eu le temps de passer notre interpréteur à la moulinette de nos fichiers tests. Nous devons le faire. Les fonctionnalitées ont été testées à la main seulement ce qui ne constitue pas un ensemble de tests rigoureux.

Nous pouvons améliorer define pour qu'il support la definition réduite d'un lambda ce qui n'est pas le cas actuellement.

Nos fichiers repl.c et read.c ne sont pas à jour nous n'avons donc le readline et nous ne pouvons pas obtenir la dernière instruction en appuyant sur la flèche du haut.

Points délicats, points faciles :

Nous avons eu quelques difficultés à implémenter lambda, ce fut l'occasion de débugger et modifier tous nos environnements et également de modifier sfs_eval pour qu'il supporte un environnement.
Le plus difficile est de comprendre en scheme le fonctionnement des environnements qui différe sensiblement de celui du C. Les fonctions ont comme environnement parent celui dans lequel elles ont été définies et nous celui dans lequel elles ont été appelées, d'ou le fonctionnement de count.

Begin était facile.

Let était également facile à implémenter même si elle demande du travail.