Projet d’informatique Livrable 3 rapports :

Rimbaud Sébastien, Donia Jaoua

Dans ce livrable nous devions implémenter les primitives. Pour implémenter une primitive nous avons choisi de créer un nouveau type d’objet : SFS_PRIMITIVE. Ces objets contiennent un pointeur de fonction et un pointeur sur un autre objet. Ce second pointeur sert à pointer le nom de la primitive en mémoire si jamais on a besoin d’afficher celle-ci. Les primitives sont enregistrées comme des variables dans l’environnement top-level. Elles possèdent un nom et une valeur qui correspond au pointeur de fonction.

Les primitives sont initialisées au démarrage de l’interpréteur. Celui-ci crée toutes les primitives et affecte le pointeur de fonction qui leur correspond.

Nous avons également modifié ENV_add_var afin qu’il ne puisse pas supprimer une primitive.
Toutes les primitives que nous avons implémenté sont visibles dans la documentation réalisée avec doxygen, il faut consulter la fonction init_primitive. On peut aussi les voir dans le code source dans primitive.c dans init_primitive.

Toutes nos primitives vérifient le nombre d’arguments reçus ainsi que leurs types. Si le type ou le nombre ne correspondent pas à ce qui est requis un warning apparaît et la fonction renvoie NULL ce qui correspond à aucun retour dans le shell.

La liste des arguments d’une primitive est évaluée récursivement dans sfs_eval.

I/ Les prédicats :

Les prédicats ont été simples à implémenter. Leur rôle est de tester le type d’un objet il suffit donc juste de regarder la variable ‘type’ de la structure.
Les prédicats vérifient le nombre d’arguments reçus. S’ils ont reçu trop ou pas assez d’arguments un warning le signale et ils renvoient NULL.

II/ Les conversions de types :

La conversion de type est très simple pour un symbole vers un string et inversement. L’union peut être vue comme string ou symbole qui sont tous des char[256] autrement dit il faut seulement changer le type de l’objet. Cette méthode fonctionne uniquement parce que l’on a pas d’allocation dynamique. Si ça avait le cas l’explication aurait été différente. ‘Symbol ‘et string sont des pointeurs sur char ils font donc 4 octets et ont le même type. Qu’on regarde ‘symbol ‘ou ‘string’ on voit la même chose et ces pointeurs sont égaux, cela est du à l’union.

Pour les integer vers string on utilise notre snprintf qui permet d’afficher un nombre dans une chaine de caractère.
La conversion inverse se fait avec notre fonction de lecture des entiers que nous avons implémenté pour sfs_read. La logique est la même.

On convertit les chars vers entiers et entiers vers chars avec un simple cast. Cette conversion va donc convertir un caractère dans son code ASCII et un entier vers le caractère correspond dans la table ASCII.

Ces fonctions sont fonctionnelles.

II/ Manipulation de listes :

Nous avons implémenté car, cdr avec la fonction OBJECT_get_cxr. Nous contrôlons le nombre d'arguments. Set-car ! Et set-cdr ! Ont été implémentés à l'aide de OBJECT_set_cxr qui a reçu pour l'occasion une session de débogage.
Eq ? A été codé avec la fonction OBJECT_isEqual déjà présente au livrable 2. Elle compare le type et la valeur de chaque élément, s'ils ne correspondent pas ils sont considérés comme différents.
List est simple, elle renvoie les arguments qu'on lui a passé en paramètre.
Cons n'accepte que 2 arguments. Cons provoque un affichage avec un point ce qui a entrainé une modification de OBJECT_print_pair. Désormais si le cdr n'est pas de type SFS_NIL ou SFS_PAIR elle ajoute le fameux point.

III/Arithmétique :

Nous avons implémenté toute l'arithmétique demandée, entière, infinie et réelle. Elle est fonctionnelle sauf pour la division réelle qui a des problèmes avec les infinis.
L'arithmétique a été implémentée avec une forte logique orientée objet. On a défini pour chaque structure, num et object les opérateurs de somme, division, différence, produit, division euclidienne, reste de la division euclidienne. De cette même façon nous avons implémenté les comparateurs avec > ≥ < ≤ =. Les comparateurs supportent l'arithmétique infini.
Pour les supérieurs ou inférieurs on regarde surtout le signe de l'infini. Deux infinis de même signe ne peuvent être comparés, le = ne peut comparer les infinis, il ne les supporte donc pas.

A chaque fois on définit l'opérateur sur la structure num qui va être chargé de gérer tous les cas avec des opérations entre réels, entiers, infinis, réel/infini, réel/entier ect… . Nous avons fait attention à ce que ces opérateurs soient bien symétriques : réel/entier fonctionne aussi bien que entier/réel.
Nous définissons ensuite l'opérateur pour object. Il appelera celui de num et effectuera surtout des vérifications sur les objets passés en paramètre (notamment pour le type).
Enfin nous utilisons l'opérateur défini sur les objets pour effectuer l'opération. On prend le premier terme auquel on va ajouter, diviser, soustraire, multiplier tous les autres. De ce fait on est obligé de copier le premier élément afin de ne pas l'altérer. Cette copie, si elle n'est pas stockée dans une variable est perdue dans la mémoire nous n'avons trouvé aucun moyen (sans garbage collector) de libérer simplement cet objet.

IV/ Tests :

Nous n'avons pas eu le temps de finir d'écrire les fichiers tests. Certains fichiers de tests sont faux et ne passent donc pas. Les seuls tests que nous avons fait ont été fait à la main pour vérifier si tous les cas étaient bien implémenté. Évidemment cela ne suffit pas à prouver la robustesse du programme.

V/ Correction de bug :
Nous avions un bug sur les entrées de type non reconnue en effet (define x \#c) faisait un boucle infinie. Cela provenait du fait que pour une entrée non reconnu on ne faisait pas bouger le curseur here. Maintenant une entrée non reconnue est lue comme une chaine et l'objet construit est de type SFS_UNKNOWN (comme avant).

Un autre bug était le suivant : (define a 'b) (define b 'a)
Cela provoquait une boucle infinie qui était due à une évaluation en trop dans éval. On évaluait 'a' à 'b' qui était encore évalué à 'a' ect….

Nous avions d'autre segfault notamment vis à vis du type SFS_UNKNOWN qui était mal géré dans sfs_eval. Nous avons corrigé cela.

Nous avons également corrigé l’implémentation douteuse des booléens. Désormais un booléen est unique et il correspond aux variables globales vrai ou faux il n’y a plus de nouvelle allocation dynamique d’un objet ou special pointe vrai ou faux. Nous avons corrigé la fonction OBJECT_destroy en conséquence.

Il semblerait que la primitive de division réelle : / ait du mal à gérer les infinis et divisons pas zéros. Elle retourne nan et inf au lieu de +inf ou -inf. Nous pensons que cela provient de printf. Nan et inf sont présents de base dans le langage C. Nous n’avons chercher à corriger ce bug pour l’instant l’arithmétique infinie n’étant pas la plus couramment utilisée dans un langage de programmation.

VI/ Ajout de fonctionnalité :

Le code supporte maintenant les infinis positifs et négatif ainsi que les réels. Toutes les primitives ont été implémentées en tenant compte de ce paramètre et supportent toutes tous ces nombres. L’arithmétique infinie est problématique. Nous avons raisonné comme on le fait en terme de limite en mathématique ce qui est la méthode de raisonnement de repl.it.