|  |  |
| --- | --- |
|  | Rapport informatique |
|  |  |
| 03/09/2016 | Programmation fonctionnelle et interprêteur Scheme : Livrable n°I |
|  |  |

# TABLE DES MATIERES

I. Introduction 3

II. Spécifications 4

II.1 Données : description des structures de données 4

II.2 Fonctions : Entête et rôle des fonctions essentielles 4

II.3 Tests : quels sont les tests prévus 6

III. Implémentation 7

III.1 Etat du logiciel : ce qui fonctionne, ce qui ne fonctionne pas 7

III.2 Tests effectués 7

III.3 Les optimisations et les extensions réalisées 7

IV. Suivi 8

IV.1 Problèmes rencontrés 8

IV.2 Planning effectif 8

V. Conclusion 9

# Introduction

Le but de cette première partie est de mettre en place, sur les bases des ressources fournis au départ, une analyse lexicale et l’affichage du résultat de celle-ci.

Dans ce document, nous allons essayer de décrire au mieux notre livrable 1 : ce qui fonctionne, ne fonctionne pas et pourquoi. Les jeux de tests fournis permettront de valider ou non certaines fonctionnalités de notre interpréteur.

# Spécifications

### II.1 Données : description des structures de données

Les structures crées sont présentes dans le fichier objet.h du répertoire principal.

On retrouve pour notre projet une structure principale qui est essentielle aux fonctions du programme :

* object qui est une structure d'object.

typedef struct object\_t {

uint type;

union {

num number;

char character;

string string;

string symbol;

struct pair\_t {

struct object\_t \*car;

struct object\_t \*cdr;

} pair;

struct object\_t \*special;

} this;

} \*object;

Elle est composée d'un unsigned int qui définit le type de l'objet lu (object special, pair, number character, string, symbol).

Les booléens ici sont traités comme des objets externes (nil est construit de la même manière) :

extern object nil;

extern object true;

extern object false;

Grâce à cette structure, nous pouvons initialiser dans le fichier principal repl.c ces objets de manière à créer des booléens dans le langage Scheme.

### II.2 Fonctions : Entête et rôle des fonctions essentielles

Notre premier livrable est composé de deux fonctions essentielles : sfs\_read et sfs\_print réparties dans les deux fichiers respectifs read.c et print.c.

* sfs\_read est une fonction qui renvoie l'arbre sous la structure object décrite précédemment. Son entête est la suivante : object sfs\_read( char \*input, uint \*here ).

Cette fonction lit au clavier ou en script l'entrée donnée. Elle parcourt alors toute l'entrée clavier pour construire l'arbre object de la manière décrite dans le sujet.

Selon ce qu'elle lit, elle va réagir de trois façons : si c'est une liste vide, elle va renvoyer nil ; si c'est une paire elle va aller lire une paire grâce à la fonction sfs\_read\_pair ; et si c'est un atome (élément de plus basse constitution), elle va aller le lire grâce à sfs\_read\_atom.

* + object sfs\_read\_atom( char \*input, uint \*here ) renvoie l'objet atom et prend en entrée la chaîne input et la position \*here dans la chaîne.

Cette fonction va d'abord chercher le type de l'atome lu grâce à uint typeInput(char \*input, uint \*here) présent dans aux\_read.c. Ce dernier va lire les premiers caractères de la chaîne et identifier s'il s'agit d'un type connu. Sinon il renvoie le type uint NO\_TYPE qui est interprêté comme un atome NULL. Cela a pour conséquence et comme dans tout le reste du sujet de passer à la suite sans prendre en compte la chaîne effective en entrée.

La fonction sfs\_read\_atom va ensuite selon le cas, affecter selon le type, une des fonctions du fichier read\_atom.c qui sont les suivantes :

/\*Les différents types d'atome\*/

object read\_atom\_number (char \*input, uint \*here ) /\*read\_atom.c\*/

object make\_integer (int valeur ) /\*object.c\*/

Très simple, elle utilise strtol pour convertir une chaîne en entier. On pourra envisager pour la suite une lecture de réel et non pas simplement d'int. make\_integer va récupérer la valeur converti et la transmettre à l'atome.

object read\_atom\_boolean (char \*input, uint \*here ) /\*read\_atom.c\*/

object make\_boolean (int b ) /\*object.c\*/

On lit au clavier #f ou #t. Si c'est TRUE (1) alors on renvoie true qui sera interprêté comme un object special par la lecture. De même si c'est FALSE (0) : renvoie false. Le make crée l'object special si besoin est.

object read\_atom\_character (char \*input, uint \*here ) /\*read\_atom.c\*/

object make\_character (char character ) /\*object.c\*/

C'est ici la plus compliquée à traiter mais dans le fonctionnement elle est très simple car c'est juste de la lecture pure et simple char par char. Si on détecte newline ou space on crée les caractères '\n' et ' '. Sinon on renvoit le caractère lu dans un char.

object read\_atom\_chaine (char \*input, uint \*here ) /\*read\_atom.c\*/

object make\_string (char\* chaine ) /\*object.c\*/

On lit la chaîne de manière normale en la stockant dans char\* chaine Il faut juste faire attention aux caractères tels que '\"'.

object read\_atom\_symbol (char \*input, uint \*here ) /\*read\_atom.c\*/

object make\_symbol (char\* symbol , int i ) /\*object.c\*/

Même combat que pour chaîne mais traité avec les special\_initial. (Fonction dans aux\_read.c d'entête int is\_special\_initial( char input ) pour les identifier.

* + object sfs\_read\_pair( char \*input, uint \*here ) renvoie l'objet atom et prend en entrée la chaîne input et la position \*here dans la chaîne.

Elle fonctionne de telle manière à créer l'arbre selon le modèle étudié dans le sujet. Elle retourne l'object complet formée par la paire. Le car créé refait intervenir sfs\_read pour vérifier si c'est un atome, la liste vide ou une nouvelle paire. Le cdr quant à lui vérifie que nous ne sommes pas à la fin de la liste. Si tel est le cas, on crée nil sinon on relit une paire.

* sfs\_print est la fonction qui va lire l'object formé par sfs\_read (plus tard, nous aurons la fonction d'évaluation entre les deux). Son entête est la suivante :

void sfs\_print( object o , uint\* root )

La variable root est là uniquement pour savoir si on se trouve à la racine de l'object ou non. Si oui, on affiche une parenthèse supplémentaire de départ qui ne s'affiche pas sinon (j'ai essayé pleins de comprendre parfaitement la structure de l'arbre objet pour l'afficher mais après du temps perdu, j'ai changé de stratégie de programmation de cette parenthèse). Cette variable root est instauré dès le début de la boucle infinie de repl.c pour retraiter chaque fois l'objet dans son intégralité. C'est une variable globale (même si c'est le Mal …) qui n'a pas d'influence sur le reste du programme à part l'affichage d'une parenthèse et de la liste vide si elle se trouve seule sur une ligne.

Si on lit un object de type SFS\_PAIR, on appelle alors sfs\_print\_pair( object o , uint \*root) qui traite les parenthèses selon les conditions que l'on peut retrouver dans print.c et rappelle sfs\_print pour le car et le cdr de la paire.

Lorsque l'on tombe sur un atome, on utilise void sfs\_print\_atom( object o ) qui selon le cas, affichera de la bonne façon les atomes prévus. (Avec gestion des booléens, des newline et space. De plus, il gère l'object nil pour afficher une liste vide si tel est le cas.

### II.3 Tests : quels sont les tests prévus

Les tests prévus ont été gardés intacts dans le répertoire test\_step1/ face à un manque de temps, le but était de pouvoir faire fonctionner le programme dans un premier temps le plus simplement possible. Dans les prochains livrables, les tests seront bien sûr adaptés à nos besoins puisque nous devons les écrire nous-mêmes de toute façon.

# Implémentation

### III.1 Etat du logiciel : ce qui fonctionne, ce qui ne fonctionne pas

Tous les tests\_step1/simple/ fonctionnent de manière correctes et 18/26 des ../evolved/. Ce qui ne fonctionne pas relève de la poussée à l'extrême du programme (très grande string non gérée encore ou entier de type int alors qu'on lui envoie un long long int). Le but pour ses prochaines semaines est de réparer toutes ces erreurs et d'optimiser le programme déjà écrit.

### III.2 Tests effectués

Les tests sont ceux fournis par le site.

### III.3 Les optimisations et les extensions réalisées

Par manque de temps pour de diverses raisons, aucune optimisation n’a été mise en place.

# Suivi

### IV.1 Problèmes rencontrés

Aucun réel problème n’a été rencontré lors de l’écriture des programmes (seulement des segmentation fault ou autres erreurs classiques qu’on aurait pu éviter en réfléchissant plus en amont). La difficulté était surtout de traiter tous les cas sans en oublier.

### IV.2 Planning effectif

Le planning a été peu respecté et le travail très mal réparti. Pour ce qui est des détails, la correction des erreurs nous a parfois pris plus de temps que prévu.

# Conclusion

Il reste à faire marcher tous les tests evolved puis se mettre au boulot pour l'implément 2 sans prendre de retard.