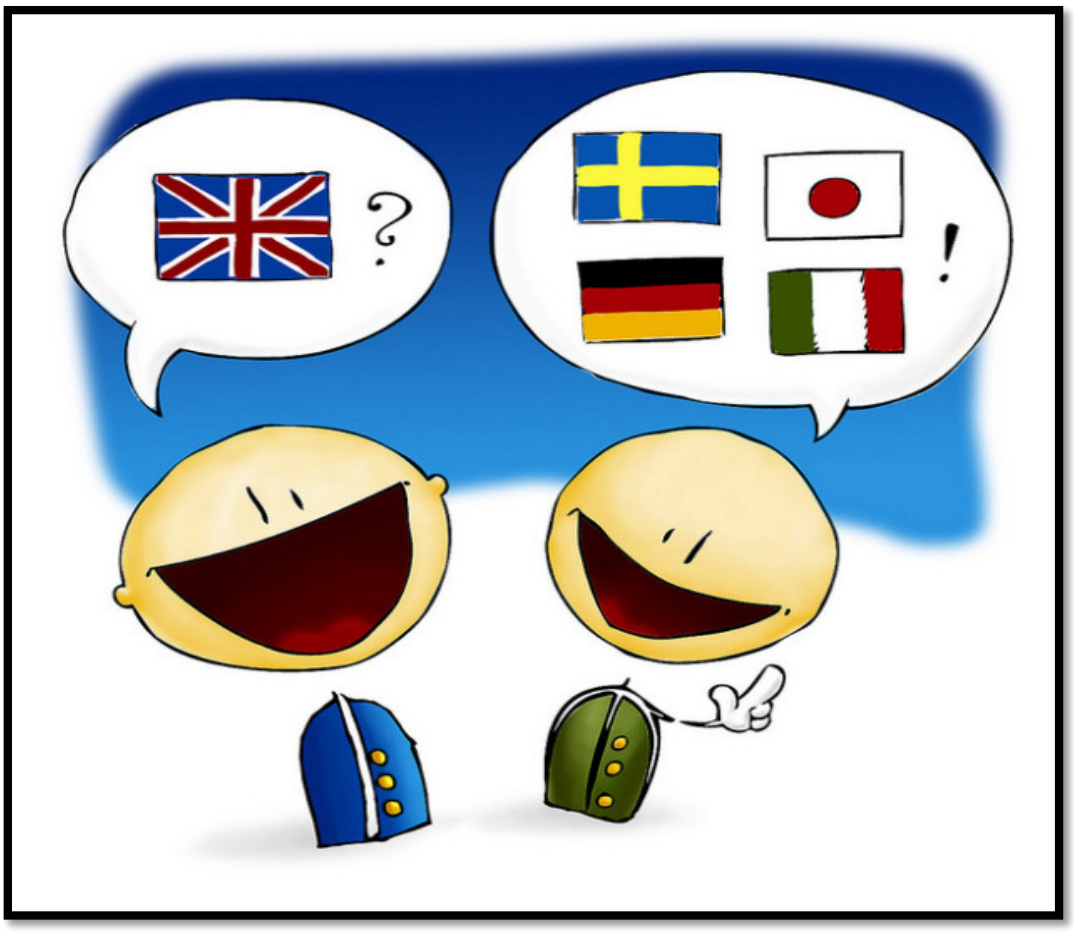
RAPPORT INFORMATIQUE



25/10/2016

Programmation fonctionnelle et interprêteur
Scheme : Livrable n°2

HRAMAN Jérémy & WALCH-GUINEBERT Romain

TABLE DES MATIERES

I. INTRODUCTION

L'objectif de cette seconde partie est de mettre en place, sur ce qui est déjà acquit, la gestion des environnements, l'ajout de l'évaluation des formes et des jeux de tests. Les améliorations du livrable I n'ont pas toutes été effectuées mais 23/26 tests évolués fonctionnent (les 3 restants concernent les infinis, les verylongchar et les lignes très longues).

Dans ce document, nous allons essayer de décrire au mieux notre livrable 2 : ce qui fonctionne, ne fonctionne pas et pourquoi. Nous verrons également brièvement les problèmes rencontrés, les solutions envisagées.

II. SPECIFICATIONS

Notre deuxième livrable est composé de trois fichiers principaux : `eval.c`, `aux_eval.c` et `is_functions.c`. Les fonctions du dernier fichier n'ont pas besoin d'être expliquées. Ce sont juste des vérifications de symboles, de types, ou de présence dans un environnement qui allègent les notations dans les autres fichiers.

II.1 Fonction `sfs_eval` et dérivées de `aux_eval.c`

La fonction `sfs_eval` récupère la sortie de `sfs_read`, on a alors un arbre correctement formé. `sfs_eval` gère les formes `define`, `set!`, `if`, `and`, `or` mais aussi les opérateurs de calcul et de comparaison simples. Grâce à la récursivité croisée, on évalue chaque élément de manière dépendante en effectuant une gestion d'erreur bien plus poussée que dans le livrable 1. Rien n'est encore parfait et certains cas ne sont pas traités. Dans `aux_eval.c`, on retrouve toutes les fonctions nécessaires à la gestion des différents symboles du langage `Scheme` et certains atomes qui demandent une gestion particulière. On renvoie alors un `object` évalué dans le programme qui sera lu par `sfs_print`.

II.2 Fonctions environnementales et formes associées

Les formes `define` et `set` appellent des fonctions environnementales d'ajout de variable et de changement de valeur dans un environnement. L'environnement principal est créé comme un `extern object` et initialisé dans `repl.c` comme une paire avec un `nil` sur chaque branche. Cet objet est le `toplevel` et stocke toutes les variables du niveau d'initialisation.

La fonction d'ajout de variable stocke une nouvelle variable en tête de `toplevel` avec sa valeur. Celle qui change la valeur, recherche dans l'environnement la dite variable.

L'objet symbol n'est pas auto-évaluant donc on recherche la variable dans l'environnement et on renvoie sa "valeur" grâce à `object eval_symbol`.

Une fonction non utilisée de création d'environnement par dessus le toplevel a été faite et sera utile dans l'incrément III.

II.3 Plus profondément dans l'évaluation : les fonctions récursives croisées

a) `eval_and` et `eval_or`

Construites sur le même modèle, elles utilisent la forme itérative `while` pour traiter tous les arguments présents dans l'`object`.

b) `eval_if`

Fonctionne sur le même modèle qu'une vraie boucle `if` traduite pour gérer les cas avec un prédicat et une ou deux expressions.

c) Autres fonctions

Les autres fonctions ont déjà été expliquées précédemment.

III. IMPLEMENTATION

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

Pour une raison encore inexpliquée, tous les tests `*.scm` qui contiennent define ne fonctionnent pas (/!\ A voir d'urgence, mail envoyé pendant les vacances). Ce qui est étrange, c'est qu'en mode interactif, il n'y a aucun soucis.

En mode interactif, toutes les formes implémentées, et l'évaluation demandée fonctionne très bien et gestionne de manière assez bien les erreurs (message d'erreur et `return NULL`).

Seule reste l'apostrophe pour `quote` n'est pas implanté dans le `read` de notre programme.

Il manque aussi des tests plus poussés pour tester les limites du programme ou juste pour voir s'il réagit de manière correcte.

```
MacBook-Pro-de-Jeremy:scheme President$ ./scheme
SFS:0 > (define bob 4)
SFS:0 > bob
==> 4
SFS:0 > (set! bob "#JeSuisScheme")
SFS:0 > bob
==> "#JeSuisScheme"
SFS:0 > (define x 5)
SFS:0 > (if (< x 6) bob)
==> "#JeSuisScheme"
SFS:0 > (if (> x 6) 4 bob)
==> "#JeSuisScheme"
SFS:0 > (if (and #t #t #t #f) 4 bob)
==> "#JeSuisScheme"
SFS:0 > (if (or #f #f #f #t) bob 4)
==> "#JeSuisScheme"
SFS:0 >
```

IV. SUIVI

IV.1 Problèmes rencontrés

Des problèmes de compréhension au début sur les environnements mais finalement, tout s'est implémenté de manière correcte pour cette incrément. On vérifiera bien que pour la suite, les fonctions déjà mises en place conviennent pour la gestion de multiples environnements.

IV.2 Répartition du travail

1ère répartition :

Romain : Jeux de tests de toutes les fonctions (permettant de mieux comment fonctionne le programme par (re)lecture du sujet) et forme `quote` avec apostrophe.

Jérémy : Jeux de tests de base pour les formes `define`, `set!`, `if`, `and` et `or`. Gestion des environnements et fonction d'évaluation `sfs_eval`.

Selon l'avancement, on devait se répartir les formes `define`, `set!`, `if`, `and` et `or`.

2ème répartition :

Terminé : Gestion d'environnement, fonction d'évaluation `sfs_eval`.

Romain : Forme `quote` avec apostrophe + fin des jeux de tests.

Jérémy : Formes `define`, `set!`, `if`, `and`, `or` et toutes les `is_functions`, opérateurs de calcul et de comparaison.

V. CONCLUSION

Un peu moins de temps qu'au premier incrément mais meilleure qualité d'écriture (peu de fautes de syntaxes, programmation plus rapide dû à la maîtrise du sujet).