

# Programmation fonctionnelle et interpréteur Scheme

Lucien Dos Santos  
Mohamed Hage Hassan

October 17, 2016

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Implémentation du mode d'évaluation</b>	<b>2</b>
<b>3</b>	<b>Environnements</b>	<b>3</b>
<b>4</b>	<b>Les formes</b>	<b>3</b>
4.1	Formes auto-génères . . . . .	3
<b>5</b>	<b>Notes</b>	<b>4</b>

# 1 Introduction

Ce rapport porte sur les travaux effectués pour la mise en place de la deuxième partie de l'interpréteur scheme. Cette partie consiste à programmer l'ensemble des fonctions qui vont remplacer la représentation tout simple par une évaluation, ainsi que des différents environnements (locaux et global) qui les différents variables, et surtout les formes essentielles du langage comme *quote*, *define*, *set!* et *if*.

On rappelle de nouveau que le code qu'on programme **ne dépend pas** de l'embryon du code fourni dans la partie 1. Notre démarche de travail va être de conséquence bien différente de celle qui est recommandée.

Il faut aussi noter que le code est programmé d'une façon incrémentale, par effet, le code fourni dans le livrable 2 va être rattaché à celui de la première partie. Par contre, on a implémenté des arguments pour l'interpreteur de façon qu'il soit possible de commuter entre le code de la partie 1 et les additions qui sont ajoutées de suite.

## 2 Implémentation du mode d'évaluation

On va parler tout d'abord du process pour le remplacement de la représentation par une partie d'évaluation des fonction.

Comme on l'a indiqué précédement, le code est très modulaire et réparti en **parseurs**, chaqu'un sert pour une fonction donnée (parseur de chaîne de caractères, des entiers, des charatères...), qui sont ainsi groupés dans des fichiers séparés. En premier temps, le code était dépendant de la fonction `represent()` (présente dans tous les parseurs).

Les fonctions `evaluate()` et `evaluate_on()` étaient déjà existantes dans le code de la partie 1, mais non implémentées. Prenons par exemple le parseur **MgIdentifier.c**

```
1 static MgStatus* not_implemented_too_late(void) {  
    return Mg_error_not_implemented;  
3 }
```

Le travail de cette partie était réparti pour la mise en place correcte de ces 2 fonctions **pour tout les parseurs**, reprenons l'exemple du parseur de symbols (**MgIdentifier**) :

```
1 static MgStatus* evaluate(MgIdentifier* self, MgObject** output, MgInterpreter*  
    interpreter, MgEnv* env){  
    // Code  
3 }  
  
5 static MgStatus* evaluate_on(MgIdentifier* self, MgIdentifier** output){  
    // Code  
7 }
```

Les structures de données associées sont aussi modifiées (*MgObjectType type = {}*).

### 3 Environnements

La méthodologie utilisée globalement dans la production du code est d'implémenter toutes les fonctionnalités possibles d'une manière modulaire (et en fichier séparé) . Cela sert principalement à prendre en compte facilement des futures implémentation ou expansion de nouvel code.

Prenons par exemple le cas de l'implémentation des environnements : celle-ci se présente par le fichier `MgEnvironment.c` présent dans le répertoire `src/core/`. Cette procédure crée les fonctions `MgEnv_create()`, `MgEnv_destroy()`, `MgEnv_add_identifieur()`.. qui ne sont autre que des wrappeurs pour les **manipulations de listes**.

La technique de programmation consiste à programmer le même fichier (ou même fonctionnalité) simultanément, de façon que si un des binôme bloque sur un problème, on peut voir tous les deux les moyens pour le résoudre (ou pour la proposition d'idées).

L'idée des environnements nestés ce fait principalement avec la fonction `MgEnv_create()` :

```
1 MgStatus* MgEnv_create(MgEnv** env, MgEnv* parent_env)
```

On rappelle que le type de structure `MgEnv` n'est que **MgList** (manipulation de listes). Les autres fonctions de l'environnement peuvent accéder à cette fonctionnalité de nesting des environnements avec `MgEnv** env` et `int scope_limited`. Le fichier `.h` est associé pour les définitions de prototypes.

La manipulation directe des environnements se fait notamment dans le parseur **MgIdentifier**, surtout la fonction `evaluate()` qui fait appelle vers des fonctions de manipulation d'environnement.

Pour une prise en compte correcte de l'addition des environnements, on a modifié les fichiers `MgObject.c`, `MgIdentifier.c`, `MgList.c`, `MgString.c`, `Mgpair.c`.

Il faut aussi noté que pour améliorer le code de façon générale, on a ajouté le module **MgInterpreter.c**, ce qui nettoie la fonctionnalité de **Interactive.c** (On déplace la plupart des fonctions inclus dans ce fichier dans **MgInterpreter.c** en récrivant quelque fonctions).

### 4 Les formes

Après une implémentation correcte des environnements, on procède à l'introduction des fonctions Built-in (*define, set!..*). Ces procédures sont créés dans un répertoire séparé `src/buildin/`.

Par rapport aux manipulation des procédures (les fonctions `evaluate()`, `evaluate_on()`, `represent()`, `destroy()`) ainsi que les fonctions qui gèrent la mémoire (`MgBuildInProcedure_create()`, `MgBuildInProcedure_destroy()`), sont définies dans le fichier **MgBuildinProcedure.c** et son fichier `.h` associé dans `src/core`.

#### 4.1 Formes auto-générées

Revenons maintenant vers les formes (dans `src/buildin`), ce répertoire contient :

- `buildin_procedure_macro.h` : Des MACROs qui vont faciliter la création des structures de données de type `Mg_buildin_procedure` (décrit les formes demandées dans la partie 2 de l'interpréteur Scheme). Cette norme de programmation est pratiquée par Lucien généralement.
- `buildin_procedures.h` : Après la définition des macros, on commence à définir ces formes comme suit (pareil pour les autres formes) :  
`MG_BUILDIN_PROCEDURE(define, ‘‘define’’);`
- `buildin_procedures.c` : Définit la liste (`Mg_buildin_procedure_array[]`) des différentes formes à utiliser.
- Chaque une des formes (*if*, *set!*, *print*, *define*, *quote*) est alors dans un fichier séparé.
- Lors de la compilation, des instructions présentes dans le fichier `Makefile` permet de voir que le compilateur remplace les noms des procédures automatiquement.

## 5 Notes

En résumé, on peut mettre l'accent sur certains points :

- Après plusieurs essais sur Valgrind, on trouve qu'il n'y a pas de possibilité de Leak : Le programme a réussi à nettoyer la mémoire (par exemple quand on n'a plus besoin des objets, ils sont tous nettoyés).
- L'exécutable a des flags de compatibilité, afin de seulement parser sans évaluer comme pour le code de la partie 1.
- Les environnements fonctionnent, même en mode du "nesting".
- Toutes les formes implémentées fonctionnent parfaitement.
- Le programme est très modulaire, on peut ajouter plein de parseurs en plus.
- Il reste la partie du "Tail Call Optimisation", qu'on ne trouve pas de moyens pour l'implémenter.
- Il en reste qu'en même des tests à faire.