

Programmation fonctionnelle et interpréteur Scheme

Binôme :

Lucien Dos Santos

Mohamed Hage Hassan

Table des matières

Analyse lexicale et affichage	2
1 Introduction	2
2 Architecture de l'interpréteur	2
2.1. Explication du code source	2
2.2. Bibliothèques séparées	4
3 Gestion de mémoire	4

Programmation fonctionnelle et interpréteur Scheme

Analyse lexicale et affichage

1. Introduction

Ce document porte sur les travaux effectués pour la mise en place de la première partie de l'interpréteur scheme. Cette partie consiste à programmer un interpréteur minimal scheme en langage C, qui a pour rôle de lire l'arbre syntaxique fournie à travers de la méthode générale (en parsing) et l'affichage de cet arbre.

De ce fait, un embryon initial était fourni pour faciliter la mise en place de tel interpréteur, avec la programmation des fonctionnalités préliminaires comme `print()`, `read()`, `read_atom()`, `read_pair()` et ainsi de suite.

Il est **important** de noter que cet embryon **n'était pas utilisé, ainsi que la structure fondamentale `object_t` et toute autre structure/définitions dépendantes de cette structure**. Celle-ci est remplacée par une **strucutre similaire** nommée `MgObject` avec une autre structure définie par `MgobjectType`. Ces structures sont expliquées dans la partie architecture.

2. Architecture de l'interpréteur

Le programme comporte un makefile générique placé dans le répertoire `src/` et qui pointe vers des autres makefile dans le répertoire `mk/`. On peut trouver le source code de l'interpréteur dans le répertoire `src/core/`. Une fois compiler **l'exécutable** se trouve dans `src/build/maniganc/debug/` nommé **maniganc**.

2.1. Explication du code source

Le programme de la fonction `main()` est `interactive.c` qui comporte essentiellement le tableau des parseurs `const MgObjectParser* object_parsers[]`, définissant ainsi les différents parseurs à utiliser. Cette architecture comporte une méthodologie **différente** de celle qui est utilisée dans l'embryon fournie, sachant que chaque **parseur** est séparé dans des fichiers source `.c`, et il sont appelé dans le programme principale (`interactive.c`).

Le programme comporte plusieurs parseurs définies d'une manière modulaire, ce qui permet d'étendre arbitrairement la fonctionnalité de l'interpreteur.

On commence tout d'abord à définir une structure `MgObject` dans **`Mgobject.h`** dans `src/core/include` et une autre qui définit les opérations qu'on peut effectuer sur les

Programmation fonctionnelle et interpréteur Scheme

fonctions `MgObjectType`. Le fichier header définit aussi des MACROs qui facilitent la manipulation des différents objets dans les structures définies.

MgStatus.h définit une structure qui contient un pointeur vers un message, qu'on va utiliser tout au long du programme.

On définit après les prototypes des fonctions (opérations) qu'on peut mener sur chaque type de parseurs dans **MgParser.h**.

On peut toujours trouver les définitions de chaque prototype de fonctions utilisées dans chaque parseurs dans les fichiers : **MgBool.h**, **MgChar.h**, **MgInteger.h**, **MgString.h**, **MgQuote.h**, **MgList.h** (sert au traitement des listes), **MgIdentifier.h**.

Chaque **fonctionnalité** des **parseurs** est alors implémentée proprement dans des les fichiers .c d'une manière totalement séparée des autres parseurs.

Pour expliquer le fonctionnement des parseurs, prenons par exemple le cas du parseur `MgList` :

La première fonction dans celui-là (et l'essentielle) est `MgList_parser_func()` qui va gérer les appels d'autres fonctions :

- On peut manipuler des listes (création/suppression des listes) par `MgList_create()` et `MgList_destroy()`.
- La vérification et la manipulation des listes et des objets par `MgList_parse_object()` et `MgList_push_back()/MgList_push_front()`

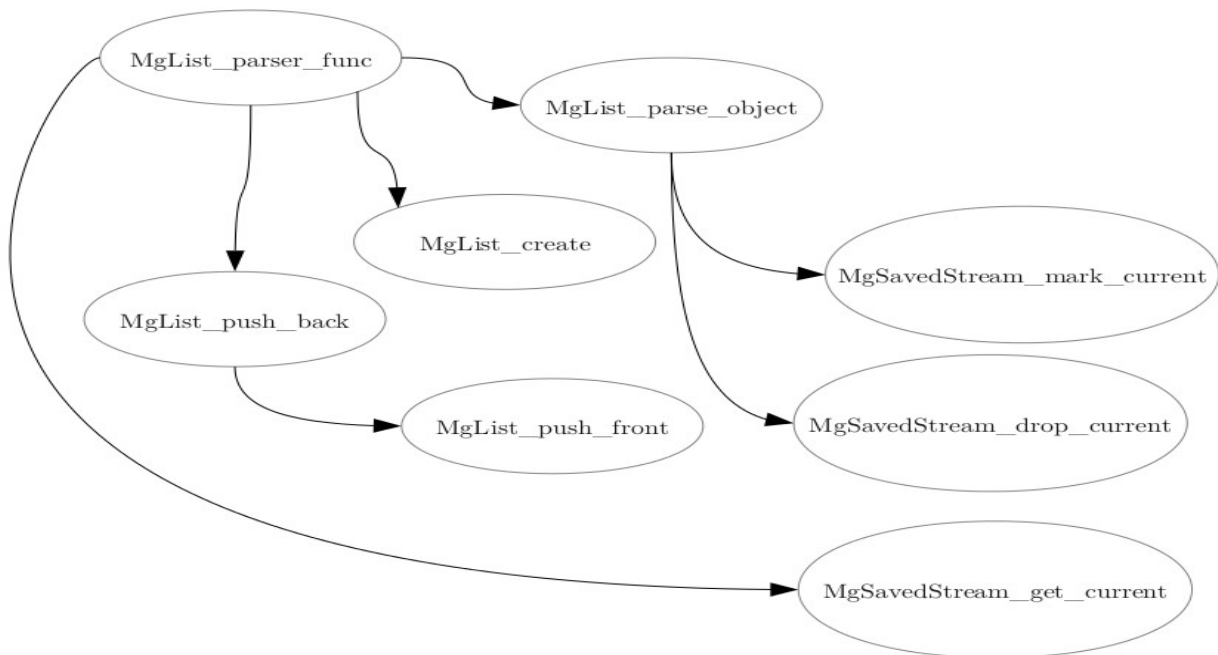


Figure 1: Appels de fonctions de `MgList`

Programmation fonctionnelle et interpréteur Scheme

2.2. Bibliothèques séparées

La figure 1 explique ce type de fonctionnement qui est similaire pour les autres parseurs. On peut alors noter l'existence des fonctions `MgSavedstream_get_current` qui est très utilisée dans le code.

Cette fonction est implémenter en utilisant des bibliothèques de manipulation de chaînes de caractères définie dans **MgSavedstream.h** (prototypes) et leur implémentation dans **MgSavedstream.c**.

Ces fonctions facilitent cette manipulation, qui permet de mettre des break-points (marks) dans une chaîne, et de parser les caractères un à un.

Toutes sortes de parseurs sont dépendants de `MgSavedstream` pour la manipulation des caractères.

3. Gestion de mémoire

Les fonctions de `MgSavedstream` appellent les fonctions `vector_char_get_idx()` et `vector_char_push()` et aussi d'autres.

Ces vecteurs sont responsables de la gestion de mémoire dynamique dans l'interpréteur, ils sont définis dans `src/std/include` et `src/core/vector_instanced.c`.

Ils sont implémentés en utilisant des MACROS (**template.h**, **class__template.h** et **vector__template.h**) ainsi que des définitions en utilisant `malloc()`, `memcpy()` puis `free()` dans **vector__template_code.h**.