

# Interprète Lisp en C++

Aurèle Barrière & Jérémy Thibault

24 avril 2016

## Table des matières

<b>1</b>	<b>Partie obligatoire : interprète à liaison dynamique</b>	<b>2</b>
1.1	Exceptions . . . . .	2
1.2	Organisation du toplevel . . . . .	2
1.3	Subroutines . . . . .	2
1.4	La directive setq . . . . .	2
1.5	Mode verbeux et affichage d'environnement . . . . .	2
<b>2</b>	<b>Partie optionnelle : gestion de la mémoire</b>	<b>3</b>
2.1	Allocation de cellules . . . . .	3
2.2	Garbage collector . . . . .	3
2.3	Lancer le Garbage Collector . . . . .	4
<b>3</b>	<b>Approche défensive</b>	<b>4</b>
<b>4</b>	<b>Graphe de dépendances</b>	<b>4</b>
<b>5</b>	<b>Mise en oeuvre et tests</b>	<b>4</b>
<b>6</b>	<b>Conclusion</b>	<b>6</b>

## Résumé

L'objectif de ce projet est de créer un interpréteur dynamique Lisp en C++. Nous nous intéresserons à la gestion de la mémoire et au comportement défensif de notre programme. Nous commençons par améliorer le code à notre disposition pour inclure de nouvelles fonctionnalités. Ensuite, nous changeons la gestion de la mémoire pour n'utiliser qu'un vecteur totalement contrôlé, avec implémentation de **Garbage Collector**. Enfin, nous considérons une approche plus défensive pour notre programme.

# 1 Partie obligatoire : interprète à liaison dynamique

## 1.1 Exceptions

La première amélioration de notre toplevel fut de rattraper toutes les exceptions lancées par le programme. En effet, il ne fallait pas que l'interprète s'arrête si l'utilisateur fait une erreur : nous voulions qu'il lui indique le type d'erreur, l'objet concerné et qu'il continue à s'exécuter.

Ainsi, nous avons créé un module `exceptions` contenant tous les types d'exceptions (des classes différentes) : exceptions liées à l'absence d'arguments, à un problème de typage, à un manque de liaisons dans l'environnement etc...

Ensuite, dans l'exécution du toplevel, on rattrape ces exceptions et on affiche les messages d'erreurs correspondant.

## 1.2 Organisation du toplevel

Dans le code initial, le fichier `main` contenait le toplevel et l'appel au parseur mélangés. Nous avons séparés ça.

Nous avons ainsi créé un module `toplevel` contenant une fonction `toplevel()` qui sera appelée dans le `main`. Ce module fait appel aux fonctions de bison pour parser un fichier donné.

## 1.3 Subroutines

Nous avons également eu besoin de rajouter des subroutines : au moins `-` et `=` étaient nécessaires pour implémenter des fonctions récursives.

Nous en avons profité pour séparer toutes les subroutines de l'évaluation dans un module distinct.

Nous avons également rajouté une subroutine `read`. Lorsqu'elle est évaluée, elle demande à l'utilisateur un objet qui va la remplacer.

## 1.4 La directive `setq`

Pour ajouter des liaisons dans l'environnement courant, nous avons du ajouter un cas particulier à l'évaluation : `setq`.

Lorsque l'utilisateur utilise cette commande, on évalue le deuxième argument, et on crée la liaison entre le premier argument et l'objet évalué.

## 1.5 Mode verbeux et affichage d'environnement

Nous avons également laissé à l'utilisateur la possibilité d'activer le mode verbeux (qui indique chaque appel d'évaluation ou d'application de fonction) avec un argument optionnel.

Nous avons également ajouté un mode verbeux pour la mémoire, qui indique à l'utilisateur toute création de cellule et libération de mémoire (cf. partie Garbage Collector).

Ces deux modes s'activent avec les options `-v` ou `-m`.

Une commande est également disponible pour afficher l'environnement courant.

Enfin, nous avons ajouté une commande `loadfile` pour lire un fichier Lips existant (notons qu'on peut également lancer l'interpréteur avec l'option `-f <nomdufichier>`).

## 2 Partie optionnelle : gestion de la mémoire

Nous avons choisi d'étudier la gestion mémoire de notre interpréteur.

Ainsi, on ne se permet que d'utiliser une certaine mémoire : un vecteur de cellules.

Nous avons donc créé un module `memory` contenant une classe du même nom. Cette classe comporte le vecteur de mémoire et les méthodes pour le manipuler.

Nous avons alors choisi de modifier le type `Object` : il ne s'agit plus d'un pointeur vers une cellule (donc son adresse dans la mémoire qu'on ne contrôle pas), mais d'un entier (son indice dans le vecteur représentant la mémoire qu'on contrôle).

### 2.1 Allocation de cellules

Pour chaque création d'objet, on veut entièrement se débarrasser des `new Cell()`, puisqu'on voudrait l'ensemble des cellules dans notre vecteur. On dispose d'un vecteur en parallèle, le vecteur `flags` qui indique pour chaque cellule du tableau si elle est utilisée ou non.

Ainsi, on crée une méthode `allocate()` dans la classe `memory` qui regarde l'indice de la première cellule non occupée du vecteur et le renvoie (tout en modifiant l'autre vecteur pour préciser que cette cellule est désormais occupée).

Alors, chaque fois qu'on a besoin d'un nouvel objet, on utilise la méthode `allocate`.

Lorsqu'on a atteint la taille maximale de notre vecteur et qu'on a appelé la méthode `allocate`, on augmente la taille du vecteur mémoire.

### 2.2 Garbage collector

Il est important de noter que la plupart des objets créés pendant une évaluation ne sont plus utilisables après. En effet, à chaque évaluation, pour chaque appel à `cons` ou chaque conversion d'objet, on demande une case de notre vecteur mémoire.

Cependant, une fois l'évaluation terminée, très peu de ces cellules sont encore atteignables depuis le toplevel.

Pour déterminer celles qui sont encore utiles, on parcourt l'environnement courant. Pour chacune des liaisons de cet environnement, on regarde récursivement les objets de cette liaison. Ces objets sont encore atteignables, et dans un vecteur on précise donc qu'ils le sont. Enfin, on supprime toutes les cellules qu'on n'a pas rencontré de la sorte. Supprimer

une cellule consiste seulement à dire dans le vecteur annexe de la mémoire qu'elle n'est pas utilisée.

Ainsi, quand on appellera de nouveau la méthode `allocate`, on pourra réutiliser ces cellules.

## 2.3 Lancer le Garbage Collector

Une fois le Garbage Collector implémenté, il faut savoir quand l'exécuter. En effet, son exécution est coûteuse en temps de calcul. Nous allons donc éviter de l'appeler à chaque fin de calcul.

Dans un premier temps, le Garbage Collector ne sera lancé qu'en dehors des calculs (sinon, on pourrait perturber les mesures de temps d'exécution d'un calcul).

Au début, nous l'appelions tous les 3 calculs. Cependant, on peut avoir une meilleure estimation de l'utilité du Garbage Collector en calculant le nombre d'allocations. Plus il y a d'allocations, plus il y a d'objets susceptibles d'être inutilisables par la suite.

On se fixe donc une borne (une constante). Lorsqu'on dépasse ce nombre d'allocations, on exécute le Garbage Collector.

## 3 Approche défensive

Nous avons ensuite choisi d'organiser la gestion de la mémoire de manière plus défensive. Il était important que les méthodes de traitement de la mémoire ne soient pas accessibles partout.

Ainsi, toutes les méthodes de la classe `memory` ont été mises en `private`. Ensuite, nous avons déclarées toutes les méthodes ou classes qui étaient autorisées à les appeler comme des classes ou fonctions *amies*.

Cela consistait à lister leur prototypes avec le mot-clé `friend` devant dans la classe.

Ainsi, on se garantit que l'accès à la mémoire ne peut se faire qu'en des endroits contrôlés.

## 4 Graphe de dépendances

On peut voir FIGURE 4 le graphe de dépendances de notre interpréteur.

On constate que beaucoup de modules utilisent le module mémoire : que ce soit pour créer une mémoire dans `main`, le `toplevel` qui peut lancer le Garbage collector, ou les autres modules qui la manipulent.

De même, de nombreux modules sont susceptibles de lancer des exceptions.

## 5 Mise en oeuvre et tests

Pour vérifier le fonctionnement de la gestion mémoire, on rajoute une option `(-p)` qui affiche le statut du vecteur mémoire.



