

Interprète Lisp en C++

Antonin Garret

Rémy Sun

21 avril 2016

1 Objectif

L'objectif que nous nous sommes fixés pour ce projet était de construire un interprète Lisp gérant les liaisons à la manière de Scheme. Plus particulièrement, nous étions intéressés par la gestion de liste chaînées de frames, et par la façon dont on pouvait représenter ces frames pour permettre la création d'un tel interpréteur.

Au delà des problèmes de définitions circulaires posés, le grand défi a été de gérer correctement la création des copies (implicites dès qu'on passe en argument de fonction) et la bonne détection des clôtures.

Nous nous sommes fixés les objectifs intermédiaires suivant :

- Compléter le squelette fourni pour obtenir un interprète dynamique, ce qui constitue une première étape facilement vérifiable.
- Passer de la gestion d'environnement proposée à une gestion par listes chaînées, ce qui permettrait eu passage de commencer à réfléchir à la façon de gérer les frames.
- Créer un premier interprète à liaison statique identique à celui vu en TD.
- Implémenter une gestion des clôtures dans cet interprète. Le but original était d'explorer la possibilité de stocker tout l'environnement courant (donc en le dédoublant), possibilité non développée en TD. Après constatation des nombreux problèmes que cela pose (mais aussi de ce qu'il est possible de faire) nous avons créé une autre clôture utilisant des pointeurs d'environnement.
- Implémenter la gestion de frames, adapter la clôture Scheme.

2 Travail réalisé

2.1 Interprète dynamique

Dans un premier temps, nous avons implémenté, à partir du code fourni, un interprète qui possède les principales fonctions Lisp, avec une gestion dynamique de l'environnement.

2.1.1 Toplevel et environnement

Nous avons encapsulé la gestion des entrées dans un `toplevel`, appelé par la fonction `main`. Ce `toplevel` gère la directive `setq`, qui permet de modifier l'environnement. Elle fait appel à une fonction `add_new_binding` que nous avons implémenté, qui permet de créer un nouvel objet `Binding` (qui comporte deux éléments : un symbole et la valeur qui lui est associée) que

l'on ajoute à l'environnement courant. Dans le code original, les environnement sont traités comme des vecteurs de pointeurs vers des `Binding`. Cette version dynamique de l'interprète ajoute un `Binding` à chaque fois que `setq` est appelée, sans vérifier si un `Binding` correspondant au symbole utilisé existe déjà. Comme la fonction de recherche dans l'environnement renvoie le premier `Binding` trouvé, cette nouvelle liaison masque de façon effective celles déjà enregistrées.

D'autre part nous avons aussi ré-encapsulé la gestion des entrées de l'utilisateur dans le `toplevel` dans une classe `read` qui gère les appels au parseur.

2.1.2 Subroutines

Dans le code fourni, quelques subroutines étaient traitées dans la fonction `eval` de l'interprète. Nous avons encapsulé les subroutines pour qu'elles soient traitées de manière séparée, afin de pouvoir facilement en ajouter de nouvelles. Les opérations de bases sur les nombres, ainsi que les fonctions `car`, `cdr`, `cons`, `concat`, `newline` et `read` ont été implémentées.

Pour la subroutine `=`, nous nous sommes inspirés de l'interprète `Caml` vu en TD, en comparant le type de chaque objet, avant de comparer leur valeur si le type correspond. Nous avons aussi implémenté la gestion des booléens dans la classe `Object`, afin de clarifier et de simplifier leur utilisation, en particulier dans cette fonction.

Nous avons d'autre part pu remarquer l'utilité de l'encapsulation déjà effectuée, car il nous a suffi de réutiliser les fonctions de la classe `read` pour implémenter la subroutine `read`.

2.1.3 Eval

La principale modification apportée à la classe `eval` fournie est l'utilisation de l'encapsulation précédemment mentionnée pour mieux compartimenter les différentes parties de l'évaluation. Nous avons utilisé la programmation par exception afin de s'assurer que chacune des ces parties se déroulent correctement. Par exemple, lors de l'évaluation d'un symbole, on fait appel à une fonction qui parcourt l'environnement courant afin de trouver une liaison correspondante, et renvoie une exception en cas d'échec. Cela nous permet de repérer lorsque l'utilisateur cherche à évaluer un symbole qui n'est pas attribué.

Nous avons aussi implémenté un dispositif de trace via la commande `debug` et d'impression de l'environnement via `printenv`, pour permettre à l'utilisateur de suivre l'évolution de l'environnement courant.

2.2 Environnement par listes chaînées

Si l'implémentation des environnement comme vecteurs de pointeurs vers des `Binding` a le mérite de marcher, elle n'en est pas moins problématique au sens où, pour copier un environnement, il est nécessaire de copier l'intégralité de la liste des pointeurs, ce qui présente un coût linéaire en la taille de l'environnement considéré !

La première idée pour remédier à cela est de faire de l'environnement un unique pointeur vers le vecteur de pointeurs. Cela ne fait cependant que déplacer le problème dans le cas où il faut copier aussi ce vecteur.

Aussi, il semble judicieux de transformer l'environnement en liste chaînée : chaque bloc de la liste contient un unique `Binding` et un pointeur vers le prochain bloc (éventuellement

NULL). Il faut noter que nous ne perdons rien en efficacité, puisqu'il fallait de toute façon parcourir tout le vecteur de pointeurs pour trouver le Binding correspondant à une chaîne donnée.

2.2.1 Blocs d'environnement

Nous avons ainsi défini une structure EnvBlock qui constitue un maillon de la liste chaînée. Puisque ces maillons sont eux mêmes repérés par leur pointeur, et pour une raison que nous verrons plus tard, il n'est pas nécessaire d'utiliser des pointeurs de Binding.

2.2.2 Environnement = Bloc ?

La principale question qui s'est posée est la suivante : considère-t-on qu'un environnement est un bloc (le bloc de « tête ») ou considère-t-on que l'environnement est quelque chose d'autre que la somme individuelle de ses parties ?

Nous avons privilégié la deuxième approche. En effet, elle permet de gérer plus proprement l'allocation dynamique en mémoire de blocs d'environnement. De plus, cela rendait beaucoup plus facile la destruction des blocs d'environnements alloués en mémoire ainsi que la création de copies indépendantes (créant aussi de nouveaux blocs). Cette gestion des copies s'est révélée par la suite sans grand intérêt, mais le code utilisé demeure dans le dossier *Copie*.

C'est cette macro-structure qui s'occupe de la gestion des ajouts et consultations de Binding : c'est elle qui fournit l'interface publique de l'environnement.

⚡ Une difficulté rencontrée avec l'utilisation d'instances de Binding au lieu de pointeurs dans les blocs d'environnement a été la création implicites de copies que nous avons du tracer à l'aide d'impressions de pointeurs. L'utilisation d'un setter a permis de remédier à ces problèmes.

2.3 Première liaison statique

2.4 Première clôture

2.5 Passage en frames

2.6 Clotures scheme

3 Vérification

3.1 Interprète dynamique

Les fonctions de l'interprète dynamique fonctionnent correctement et permettent une utilisation classique de Lisp. Le seul problème est la gestion des erreurs de l'utilisateur via les exceptions. La programmation défensive utilisée dans le code original permet de s'assurer que tout fonctionne comme prévu, mais en cas d'erreur, la fonction assert qui a été utilisée stoppe le processus de l'interprète et empêche de rattraper les exception comme nous voulions le faire. Il serait possible de surmonter ce problème en modifiant la manière dont ces tests sont effectués afin de générer des exceptions spécifiques, mais nous avons choisi de consacrer notre temps aux autres aspects de l'interprète.

3.2 Clotûre Scheme

4 Contribution

- <http://stackoverflow.com>
- <https://openclassrooms.com>
- <http://www.cs.rpi.edu>