Interprète LISP en C++
Projet 3 de programmation

Nicolas Bailluet
nicolas.bailluet@ens-rennes.fr

Tom Bordin
tom.bordin@ens-rennes.fr

Samuel Bouaziz
samuel.bouaziz@ens-rennes.fr

Rémi Piau
remi.piau@ens-rennes.fr

ENS Rennes, ISTIC, Université Rennes 1
5 mars 2019

Résumé

Dans ce rapport, nous décrivons les méthodes utilisées dans le développement d'un interprète LISP en C++. Nous y ajoutons un *Garbage Collector* pour limiter la saturation de la mémoire.

Introduction

Le but de ce projet était de réaliser un interprète LISP en C++. Rappelons tout d'abord ce qu'est LISP (`http://lisp.org`). LISP est un langage de programmation fondé sur les listes dont voici un exemple de code : `(if (= 0 1) (+ 1 2) (* 3 12))`. Nous avons réalisé cet interprète en C++14 avec l'aide de l'implémentation moderne de *lex* et *yacc* (*flex* et *bison*).

1 L'implémentation de l'API

1.1 Les objets LISP : la classe `Cell`

Afin d'interpréter les entrées de l'utilisateur, il est nécessaire de pouvoir représenter les objets LISP (symboles, nombres, listes, etc.) en C++ dans notre API. Le cœur de notre implémentation repose sur le type `Cell` ainsi que les types dérivés `Cell_simple` et `Cell_generic_pair`.

```
enum class sort { UNDEFINED, NUMBER, STRING, SYMBOL, PAIR, SUBR, CLOSURE };
class Cell {
 private:
  ...
  uint64_t magic;
  sort s;
  void clean();
  ...
 protected:
  Cell(sort s_);
  virtual ~Cell();
  ...
  void check() const;
  ...
};
```

La vérification de l'intégrité des cellules se fait grâce au membre `magic` initialisé comme étant égal à `this` durant l'instanciation. Il est donc ensuite possible de vérifier l'égalité `this == magic` en utilisant la fonction `check`. On s'assure par ailleurs de nettoyer la valeur de `magic` (et des autres attributs) lors de la destruction des cellules afin d'éviter de garder des valeurs valides en mémoire. L'attribut `s` permet quant à lui la distinction des différents types de cellules et donc des différents types d'objets LISP.

Le type `Cell` n'est cependant pas directement utilisé, les types `Cell_simple` et `Cell_generic_pair` permettent respectivement la représentation d'objets simples (chaînes de caractères, nombres, etc.) et de paires (et listes).

```
template <typename T, sort S>
class Cell_simple : public Cell {
 private:
```

```
  T contents;
 private:
  Cell_simple(T t);
  T get_contents() const;
  ...
};

template <sort S>
class Cell_generic_pair : public Cell {
 private:
  Cell *car;
  Cell *cdr;
 private:
  Cell_generic_pair(Cell *_car, Cell *_cdr);
  Cell *get_car() const;
  Cell *get_cdr() const;
  ...
};
...
using Cell_number = Cell_simple<int, sort::NUMBER>;
using Cell_pair = Cell_generic_pair<sort::PAIR>;
...
```

L'utilisation de `template` permet la définition de différents types de cellules partageant une structure commune tout en ayant un contenu de type variable. On évite ainsi une définition lourde et redondante des différentes sortes de cellules.

On représente alors un objet LISP par un pointeur vers une cellule :

```
using Object = Cell*;
```

1.2 Environnement et liaisons

Pour avoir un interprète fonctionnel, il est important de pouvoir stocker et conserver les objets. On utilise pour cela un environnement implémenté comme une liste chaînée (objet LISP) contenant des liaisons.

```
using Env = Object;
```

Les liaisons sont des objets LISP, des paires avec d'une part un objet *string* représentant le symbole et d'autre part l'objet auquel on doit lier le symbole.

Nous avons ensuite implémenté des clôtures. Pour cela, nous avons simplement ajouté un pointeur vers l'environnement courant lors de la création d'une lambda expression. Cet environnement est alors utilisé à chaque évaluation de la lambda expression associée.

1.3 Les sous-programmes et mots-clés

L'interprète LISP traditionnel contient des sous-programmes aussi appelés *subroutines*. Ceux-ci permettent de faire les opérations les plus basiques sur les objets. Nous avons donc décidé de les implémenter de la manière suivante :

```
using subr = std::function<Object(Object)>;
```

Le type `subr` permet de représenter des fonctions prenant un objet LISP en paramètre, par exemple une liste d'arguments, et renvoyant la valeur désirée sous la forme d'un objet LISP. Les opérateurs sur les nombres ou les listes comme `+` et `car` sont donc des sous-programmes de notre interprète. Une liste exhaustive de l'ensemble des sous-programmes est disponible en annexe (cf Fig. 1).

Afin de pouvoir redéfinir les sous-programmes ou de les modifier, nous avons fait le choix de les placer dans l'environnement initial de notre interprète. L'instruction `(printenv)` permet de les afficher (cf Fig. 1). Pour ajouter un sous-programme, il suffit d'implémenter (en C++) sa fonction puis de la lier à son symbole dans l'environnement initial (géré par le *toplevel*).

D'autres instructions sont disponibles à l'initialisation, celles-ci sont appelées mots-clés. Par exemple, `if` ou `begin` sont des mots-clés. Contrairement aux sous-programmes qui ne nécessitent pas la donnée de l'environnement pour effectuer leurs opérations, certains mots-clés peuvent en avoir besoin. Dans le but d'obtenir un seul type pour les mots-clés, nous avons choisi de les représenter grâce au type suivant :

```
using keyword = std::function<Object(Object, Env)>;
```

L'environnement n'est pas toujours utilisé même si il est passé en paramètre. Cette uniformisation permet de regrouper les mots-clés que l'on enregistre dans une table de hachage. On fait une liaison entre la chaîne désignant le mot-clé et le couple message d'aide/fonction. Ceci permet une évaluation plus propre (pas de `if` / `else if` excessifs) et l'ajout de nouveaux mots-clés est alors relativement simple (pas besoin de modifier la fonction d'évaluation).

Des messages d'aide peuvent être affichés grâce à au mot-clé `help`. Celui-ci peut prendre un mot-clé en paramètre optionnel, si c'est le cas, il n'affichera que l'aide concernant ce dernier. Dans le cas contraire, il affiche les informations de toutes les fonctions.

2 L'interprète

2.1 *Toplevel*

Interfacer l'entrée clavier et l'évaluateur est le rôle primordial que joue le *toplevel* dans notre programme. Ce *toplevel* est composé de deux boucles infinies (`while(true)`) imbriquées. La première boucle infinie se lance au début du programme par l'appel à la fonction `go` et sert à afficher les erreurs de l'utilisateur sans sortir de l'interprète et permet la sortie du programme lors de l'exception `EndToplevelException`. Cette fonction permet aussi d'enregistrer les *subroutines* dans l'environnement avant le démarrage de l'interprète lors de l'appel à la fonction

`loop`.

L'autre boucle se trouve dans cette même fonction (`loop`), elle permet la lecture et l'analyse lexicale des caractères entrés par l'utilisateur pour récupérer l'objet d'entrée `iob`. Cet objet une fois récupéré est évalué par la fonction `eval` qui fournit en retour l'objet de sortie `oob`, résultat de l'évaluation de `iob`. `oob` est ensuite affiché.

```
void Toplevel::go() {
  print_criminalisp(cout);
  register_keywords();
  while (true) {
    try {
      loop();
    } catch (EndToplevelException& e) {
      clog << e.what() << endl;
      break;
    } catch (runtime_error& e) {
      clog << e.what() << endl;
    }
  }
}
```

```
void Toplevel::loop() {
  // on omet dans cette fonction la gestion d'erreur pour plus de lisibilité
  Object iob = NULL;
  Object oob = NULL;
  while (true) {
    iob = read_object();
    if (Toplevel::check_directive(iob)) continue;
    oob = eval_direct(iob, global_env);
    cout << "Got: " << oob << endl;
  }
}
```

2.2 La directive `load`

Nous avons ajouté une directive permettant de charger des fichiers pour faciliter l'utilisation de l'interprète par l'utilisateur (voir Figure 4). L'évaluation des directives étant récursive il est tout à fait possible d'imbriquer des directives `load` à travers plusieurs fichiers.

Il faut toutefois noter que la totalité du fichier est passée à l'analyseur à son ouverture, ainsi le contenu du fichier est traité de la même manière qu'une longue expression écrite par l'utilisateur. Par conséquent des `load` imbriqués entraîneront l'évaluation des fichiers demandés seulement les uns après les autres, c'est-à-dire que chaque fichier est évalué entièrement avant qu'un autre le soit à son tour.

2.3 *Eval* et *apply*

Les fonctions `eval` et `apply` sont mutuellement récursives. Lors de l'évaluation d'un objet, le *toplevel* appelle `eval` avec en argument l'objet et l'environnement associé à l'objet (l'environnement global au *toplevel*).

La fonction `eval` retourne ensuite le résultat de l'évaluation de l'objet mais cet objet peut contenir des appels de fonctions et plusieurs sous-expressions imbriquées, dans ce cas la fonction `apply` est appelée.

La fonction `apply` se charge donc de récupérer les *closures* pour les fonctions et d'appliquer ces dernières (en rappelant `apply` avec l'environnement de la *closure*).

L'évaluation étant un procédé récursif, les objets les plus profonds sont évalués en premier. Enfin, l'évaluation se fait de gauche à droite.

3 Le ramasse-miettes

L'évaluation et l'application des formules sont coûteuses en terme de nombre de cellules allouées. On remarque par exemple l'allocation exponentielle de cellules (cf. Figure 2 et 3) par la fonction de Fibonacci :

```
(definerec fib
  (lambda (n) (if (= n 0) 1 (if (= n 1) 1 (+ (fib (- n 1)) (fib (- n 2)))))))
```

Un important nombre de ces cellules ne sont plus utiles en sortie d'évaluation, d'où la nécessité de recycler la mémoire grâce à un ramasse-miettes.

3.1 Principe du *Mark & Sweep*

Afin de libérer les cellules inutiles, un ramasse-miettes de type *Mark & Sweep* [2] a été implémenté. L'algorithme s'effectue en deux parties :

1. **Marquage :** lors de cette phase l'ensemble de cellules utilisées et utilisables sont marquées. Pour cela on effectue un parcours en profondeur (*DFS*) de l'environnement.
2. **Libération :** lors de cette phase on libère toutes les cellules allouées qui n'ont pas été marquées précédemment. On enlève ensuite le marquage des cellules restantes.

3.2 Implémentation

On étend tout d'abord la classe `Cell` afin de stocker le marquage des cellules et la liste des objets alloués (les cellules s'y enregistrent lors de l'instanciation et s'en retirent lors de la destruction).

```
enum class mark { MARKED, NOT_MARKED };
class Cell {
```

```
 private:
  static std::set<Cell *> allocated;
  mark m;
  ..
};
```

Remarque : il est ici préférable d'utiliser le type `std::set<Cell*>` afin d'éviter la présence de doublons qui pourraient poser problème lors de la libération des objets.

On introduit ensuite les fonctions `API::GC::mark_rec` et `API::GC::sweep` d'un nouveau module `API::GC` qui effectuent les deux phases de l'algorithme.

3.3 Difficultés rencontrées

Même si l'algorithme est simple, son implémentation n'en reste pas moins délicate. En effet, le moindre oubli ou la moindre erreur est susceptible d'engendrer l'utilisation de *dangling pointers* et des erreurs de type *Use-after-Free*.

Il est notamment nécessaire d'exclure les cellules globales *#t*, *#f* et *nil* de l'algorithme car elles ne sont pas toujours présentes dans l'environnement et les libérer produirait inévitablement des erreurs.

De plus, libérer un objet engendre la modification de la structure de `Class::allocated`, il en résulte donc un comportement imprédictible si cette libération est effectuée lors du parcours de l'ensemble. Pour éviter toute erreur, on effectue une copie de l'ensemble et on parcourt la copie à la place de l'original dans `API::GC::sweep`.

En cas de problème l'utilisation de *gdb* et de AddressSanitizer [3] (*ASAN*) est recommandée (voir Section 4).

3.4 Résultats et discussion

Afin d'évaluer l'efficacité de notre implémentation nous avons réalisé une analyse de la consommation mémoire de notre interprète grâce à l'outil *mprof* [1]. L'analyse de dix évaluations successives de `(fib 12)` est ainsi présentée à la Figure 5. On observe alors bien dans un cas la libération de la mémoire grâce au ramasse-miettes et l'augmentation de la mémoire consommée dans l'autre cas.

Cependant le ramasse-miettes implémenté ne prend pas en compte les allocations mémoire réalisées par le *parser*, il ne gère en effet que les cellules.

De plus, l'algorithme ne peut être exécuté en cours d'évaluation d'une expression sans causer la libération de cellules utilisées par l'évaluateur. On remarque ensuite que certaines cellules de l'environnement pourraient aussi être libérées comme dans l'exemple suivant :

```
(define x 1)

(define x 2)
```

Ici, après la seconde définition de `x`, la première cellule associée à cette variable pourrait être libérée.

4 Gestion des erreurs et mode *debug*

Nous avons ajouté un mode *debug* pour faciliter le développement, il est activé en compilant avec la règle `make debug` qui ajoute les symboles avec l'option `-g` et la définition d'une macro-expression `DEBUG` avec l'option `-DDEBUG`. Ce mode a été particulièrement utile pour implémenter le mot-clé `let` et le ramasse-miettes (voir Section 3).

La gestion des erreurs se fait au niveau du fichier `error.cc`. Deux macros ont été créées : `ERROR(message)` et `ERROR_CHAIN(e, message)`. La première sert à envoyer une erreur avec le message `message`, et la seconde permet de gérer les chaînes d'erreurs, c'est-à-dire lorsqu'on attrape une exception `e` qui a été lancée à un niveau inférieur du programme. En mode *debug*, l'intégralité des chaînes d'erreurs, les noms de fichiers, les noms de fonctions et les numéros de ligne sont affichés lorsqu'une qu'une exception est levée afin de trouver précisément la source du problème. Pour l'utilisateur, le message d'erreur doit être clair et c'est pour cela que seul le message d'erreur le plus haut est affiché (hors mode *debug*).

Nous avons également implémenté une règle `make sanitizer`. Cette règle ajoute l'option de compilation `-fsanitize=address` pour compiler avec ASAN [3] qui permet de tracer les corruptions mémoire de manière détaillée. Ce mode a été particulièrement utile lors du développement du ramasse-miettes.

Conclusion

Ce projet nous a guidé dans la réalisation d'un interprète LISP complet et modulaire bénéficiant de fonctionnalités avancées comme la gestion des continuations, la coloration syntaxique et la gestion fine des erreurs. Il bénéficie aussi d'un ramasse-miettes permettant d'éviter certains problèmes de saturation de mémoire. Cet interprète, bien que complet, bénéficierait de l'implémentation de quelques fonctionnalités supplémentaires comme :

— le recyclage dynamique de mémoire au cours de l'évaluation ;
— l'auto-complétion et la coloration en temps réel ;
— l'ajout d'une limite de récursion pour éviter les *stack-overflow*.

Références

[1] Fabian Pedregosa Philippe Gervais. Memory profiler. `https://github.com/pythonprofilers/memory_profiler`.

[2] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4) :184–195, April 1960.

[3] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer : A fast address sanity checker. In *USENIX ATC 2012*, 2012.

Annexe

```
% (printenv)
=> (printenv)
newline <- <subr>
display <- <subr>
list? <- <subr>
symbol? <- <subr>
number? <- <subr>
string? <- <subr>
null? <- <subr>
cdr <- <subr>
car <- <subr>
cons <- <subr>
= <- <subr>
* <- <subr>
- <- <subr>
+ <- <subr>
```

FIGURE 1 – Ensemble des sous-programmes contenus dans l'environnement.

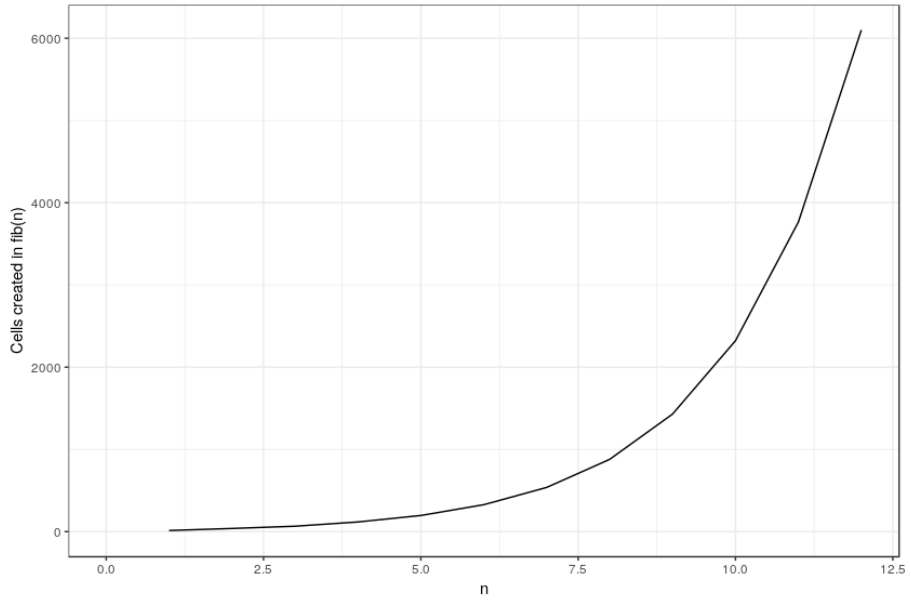

FIGURE 2 – Nombre de cellules créées lors de l'appel à `fib` en fonction de n.

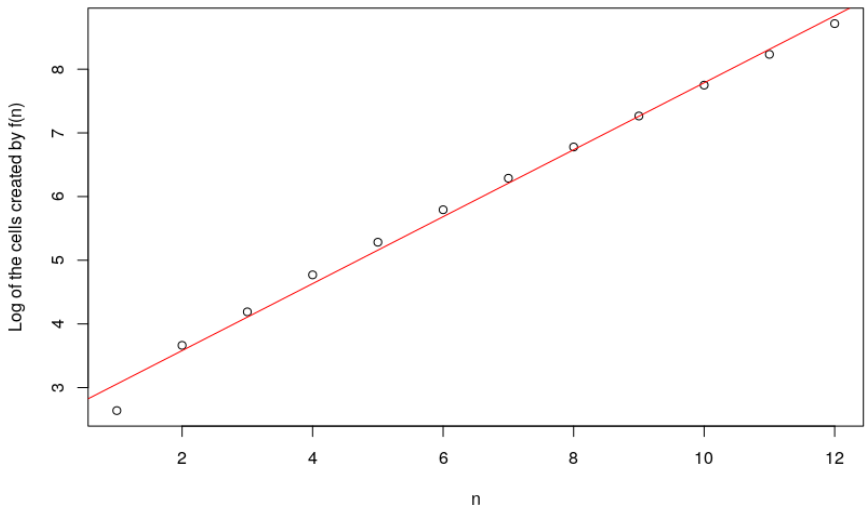

FIGURE 3 – Régression linéaire du logarithme du nombre de cellules créées lors de l'appel à `fib` en fonction de n ($R = 0.9934$).

```
% (load "init.lsp")
=> (load "init.lsp")
Loading: init.lsp

% => (define a 1)
Define : a = 1

% => (definerec fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
Define : fact = [closure | (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)
)))) | <env>]

% => (fact 10)
Got: 3628800
```

FIGURE 4 – Un exemple avec la directive `load` .

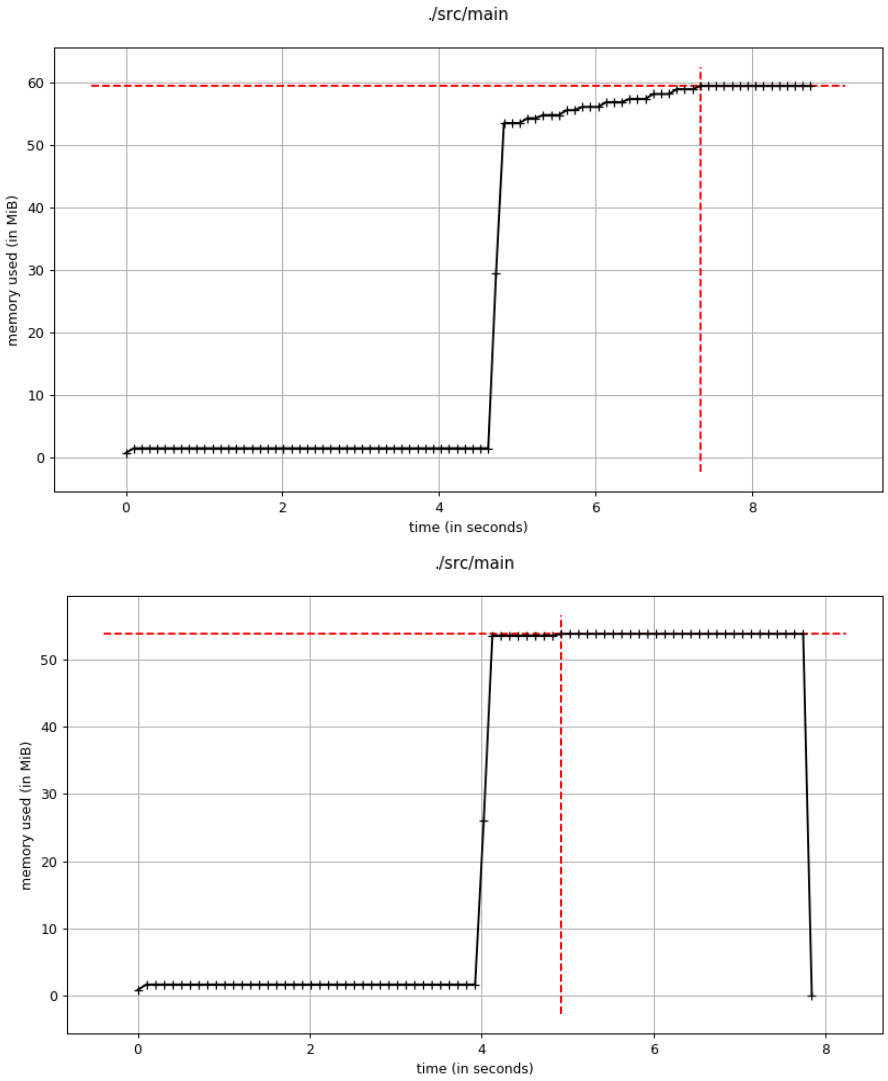

FIGURE 5 – Analyse mémoire de dix évaluations successives de `(fib 12)` avec ramasse-miettes (en bas) et sans ramasse-miettes (en haut).