Rapport de projet : Interprète Lisp

Maxime CAUTÉ, Julien DURON, Mathieu POIRIER

ENS Rennes

5 mars 2019

Table des matières

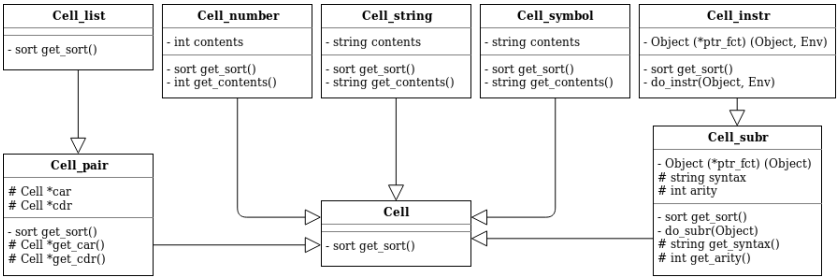


FIGURE 1 – Héritages entre cellules.

Introduction

Lisp regroupe une famille de langages, qui fut pensée en premier lieu par John McCarthy. Lisp est caractérisé par une importance centrale des listes, donnant une sémantique forte aux parenthèses. Ces langages suivent de plus un paradigme fonctionnel, car les lambda-expressions font partie du langage et peuvent être manipulées.

Le but de ce projet est d'implémenter un interprète Lisp simple en C++. Un objectif concret est de fournir un interprète permettant la définition de la fonction *fibonacci*.

1 Structure

1.1 Définition et architecture des cellules

Les classes de type `Cell` sont un moyen de représenter les objets Lisp. Plus exactement, chaque objet Lisp est un pointeur vers une cellule dans notre interprète. Les différents types existants en Lisp sont implémentés grâce à différentes classes de cellules liées par héritage.

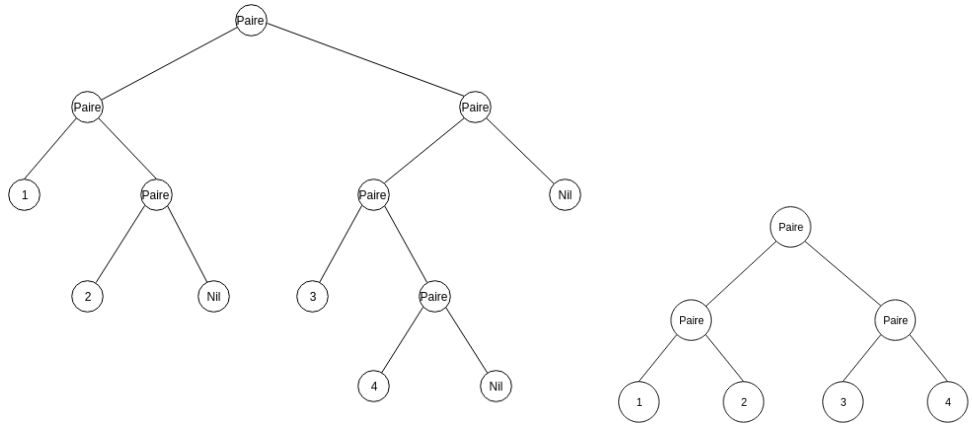
La figure 1 illustre les différentes classes de cellules existantes. La plupart héritent directement de la classe `Cell`, notamment pour les objets simples comme les nombres, les chaînes de caractères et les symboles. Seules les instructions et les listes héritent de classes filles de `Cell`. Les instructions sont des *subroutines* particulières, dont l'implémentation et l'utilisation sont détaillés en partie 3.3. Quant à la dissociation faite entre listes et paires, celle-ci est détaillée dans la partie 1.2.

1.2 Choix de la différence entre listes et arbres quelconques

Notre interprète se veut général. Il faut donc pouvoir créer des paires d'objets quelconques, puisque `Cell_pair` ne possède que deux pointeurs vers des cellules quelconques. Une idée est d'utiliser cette structure de paire aussi bien pour façonner des listes que des arbres. Nous avons donc choisi de différencier les deux dès le module `Core` en faisant hériter les paires créant les listes des paires quelconques. Le principe d'une liste est que le fils de droite est soit une autre liste soit la liste vide définie par `nil`. Il peut toutefois être très utile de garder la structure d'arbre originale pour avoir moins de cellules à allouer, comme l'illustre la figure 2 montrant la sémantique du code suivant :

```
Command? (cons (cons 1 (cons 2 ())) (cons (cons 3 (cons 4 ())) ()))
Read: (cons (cons 1 (cons 2 ())) (cons (cons 3 (cons 4 ())) ()))
((1 2) (3 4))

Command? (cons (cons 1 2) (cons 3 4))
```



(a) Avec des paires. (b) Avec des listes.

FIGURE 2 – Comparatif du nombres de `Cell` utilisée par les paires et par les listes.

```
Read: (cons (cons 1 2) (cons 3 4))
((1 : 2) : (3 : 4))
```

Néanmoins l'affichage sous forme de liste rend plus lisible des listes comme : `(+ 1 2 3 4 5)`.

1.3 Rôle de l'API et le la bibliothèque

Une fois les cellules construites, il faut pouvoir les utiliser. L'API est implémentée dans ce but. C'est elle qui gère la séparation entre les opérations de bas niveau, telles que la création de paire et commandes de haut niveau implémentées dans la bibliothèque.

Les fonctions de l'API ne prennent jamais en entrée des pointeurs vers un type de cellule particulier. Il faut donc que ces fonctions fassent en interne la transition entre un pointeur vers une cellule quelconque et un pointeur vers une cellule spécialisée. Cela est géré par des `dynamic_cast`, qui permettent de s'assurer du type d'arrivée. Ceux-ci renvoyant un pointeur nul si les types ne sont pas compatibles, il faut et il suffit de s'assurer que les pointeurs fournis sont non-nuls pour savoir que nous utilisons les bons types d'objets.

L'API rend donc les cellules abstraites aux yeux du monde extérieur, comme l'illustre la figure 3.

Au-delà de l'API, la bibliothèque propose l'ensemble des fonctions utiles aux fichiers de haut niveau, telles que `cdr` ou `number_to_object`.

1.4 Vue globale

La figure 4 représente l'ensemble de la structure de notre interprète. Comme nous l'avons précisé plus tôt, le `Core` contenant les cellules est totalement séparé du reste de l'interprète par l'API, et la bibliothèque est elle-même la seule à utiliser l'API. La bibliothèque est d'ailleurs le point de convergence global de notre code, car tous les fichiers de plus haut niveau l'utilisent, sans exception. Nous pouvons observer dans les utilitaires le module gérant l'environnement et le module `Hints`, deux sujets que nous aborderons respectivement en partie 3.1 et 5.1. Enfin nous arrivons dans le module `Eval` qui se chargera de calculer la valeur des expressions considérés, puis le `Toplevel` qui s'assure tout simplement du passage des commandes de l'utilisateur aux niveaux inférieurs.

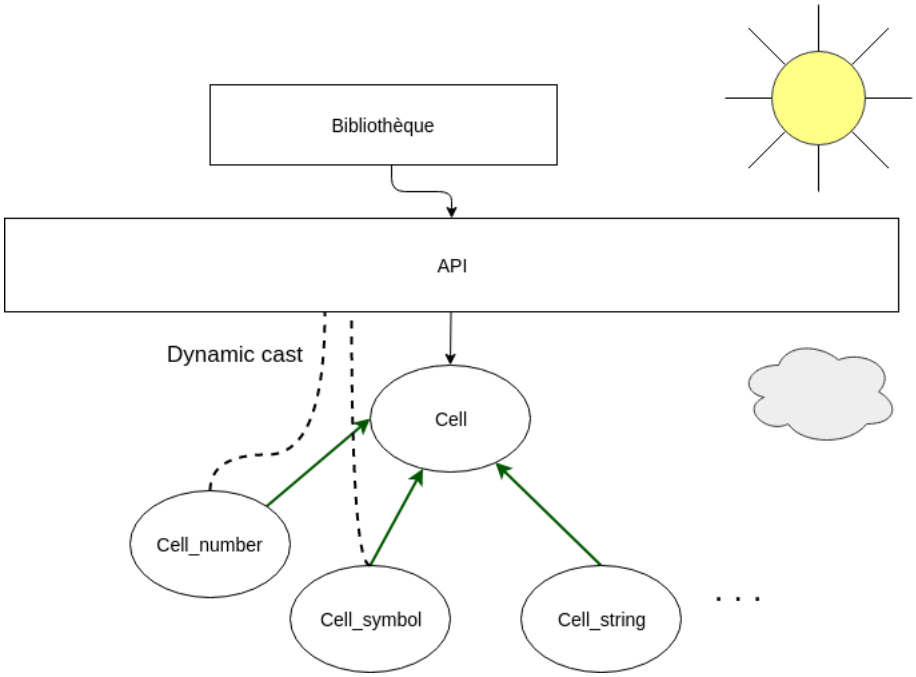


FIGURE 3 – Place de l'API dans l'interprète.

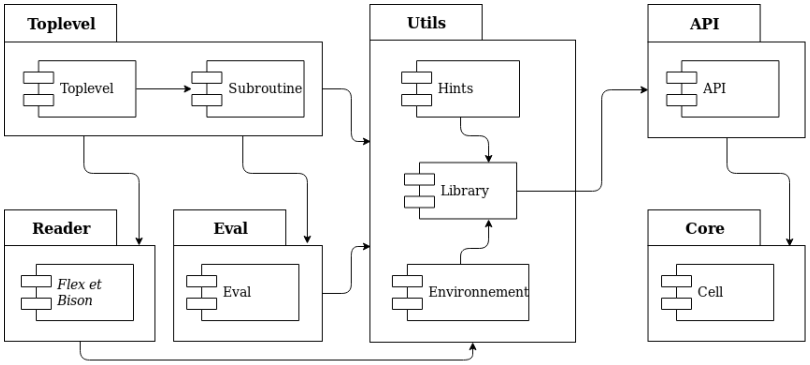


FIGURE 4 – Diagramme de classes complet.

Nous pouvons remarquer que le module `Reader` est assez indépendant du reste du code de l'interprète, cela est dû au fait que sa tâche n'a pas besoin de reposer sur autre chose que la bibliothèque.

2 Lecture et analyse

2.1 Principe

Un analyseur lexical nous était fourni par le concepteur du sujet. Cet analyseur emploie les outils *Yacc* et *Lex*.

Les composants des entrées clavier peuvent être regroupés en catégories :

— les briques élémentaires du lexique : nombres (suites de chiffres), mots, opérateurs, espaces, lignes ;

— les futurs objets Lisp : nombres (en tant qu'objets Lisp, potentiellement précédés de "-"), symboles, booléens, etc. ;

— les éléments syntaxiques liés au langage : parenthèses, fin de fichier.

Ces catégories peuvent servir de fondations pour les suivantes : c'est notamment le rôle de la première.

Ainsi un symbole est défini comme :
— une suite d'opérateurs ;
— une suite de caractères alphanumériques commençant par un caractère alphabétique ;
— le booléen vrai ;
— le booléen faux.

```
SYMBOL ( {OPERATION}+ | ({ALPHA}({ALPHA}|{NUM})*) | {TRUE} | {FALSE})
```

Lors de l'analyse, les éléments liés au langage même (objets, éléments syntaxiques) sont associés à un *token* permettant de remonter l'information vers le niveau supérieur. Cette remontée a lieu lorsque l'analyseur observe une syntaxe correcte dans son tampon de travail, commençant au début d'icelui. Il envoie alors le *token* associé.

Une syntaxe correcte, appelée expression, est définie récursivement comme :
— un objet Lisp (autre qu'une liste) ;
— une expression entre parenthèses (associée aux listes) ;
— une *quote* (') suivie d'une expression.

Il est à noter que les parenthèses fermantes en entrée de tampon sont ignorées. De plus, la dernière option correspond à une macro-expansion de la commande lisp `(quote obj)`. Le *token* renvoyé est alors un *token* d'expression correspondant à la commande ci-avant.

3 Traitement

3.1 Environnement

Le pré-traitement effectué par l'analyseur est très limité : il traduit simplement en objet Lisp l'entrée fournie par l'utilisateur. Cependant, pour enrichir le langage au-delà de la reconnaissance des nombres et des chaînes de caractères, il faut fournir à l'interprète un dictionnaire symbolique, afin qu'il puisse justement interpréter les symboles. C'est là le rôle de l'environnement.

Formellement , l'interprétation d'un symbole se fait à l'aide d'un couple symbole/valeur, appelé liaison. Ce couple est représenté en machine par une paire. Un environnement est donc une liste de telles paires, et fonctionne comme une simple liste associative.

Il existe dans l'interprète une instance globale de l'environnement. Il est également possible de créer des environnements locaux, c'est-à-dire ajouter en tête de liste des liaisons temporaires à l'environnement global. Ceux-ci priment alors sur l'environnement global. En effet, rien n'empêche l'environnement de contenir plusieurs liaison dont le symbole est identique. Nous considérons alors la première rencontrée.

3.2 Les fonctions `eval` et `apply`

Les fonctions `eval` et `apply` sont deux fonctions mutuellement récursives qui sont au coeur de l'interprète. Ces fonctions permettent respectivement d'évaluer une expression et d'appliquer à des arguments une fonction qui peut prendre la forme d'une lambda-expression ou encore d'une *subroutine*.

En pratique, l'évaluation d'un objet passe par une disjonction de cas sur les types que peut prendre l'objet. Pour les types simples tels que les nombres, ou encore chaînes de caractères, l'évaluation est l'objet lui-même. De même pour une *subroutine* ou une lambda-expression. En revanche, lorsqu'il s'agit d'évaluer une liste non-vide qui n'est pas une lambda-expression, le premier élément sera traité comme une fonction à laquelle nous appliquons comme arguments le reste des éléments de la liste, éventuellement aucun. C'est alors que la fonction `apply` est appelée.

L'application d'une fonction à des arguments échoue si l'élément considéré comme fonction se révèle ne pas pouvoir en être une, notamment si c'est un nombre ou une chaîne de caractères par exemple. Dans le cas d'un symbole, il est nécessaire de commencer par évaluer ce symbole, car il peut recéler une fonction après recherche dans l'environnement de sa valeur. Ainsi le cas du symbole se ramène aux deux suivants s'il est évalué en une fonction. Si une lambda-expression est est trouvée, un environnement local est créé, dans lequel chaque paramètre de la lambda-expression est associé aux arguments apposés à la fonction, puis le corps de la lambda-expression est évalué dans cet environnement local. Enfin pour une *subroutine*, la *subroutine* est appliquée aux arguments fournis. Ces arguments sont pré-évalués ou non, selon si la *subroutine* n'est pas une instruction ou en est une, notion détaillée dans la partie qui suit. Nous pouvons constater qu'en de multiples points, la fonction `apply` a besoin d'évaluer tout ou partie des arguments qui lui sont transmis.

3.3 *Subroutines* et instructions

Les *subroutines* sont les fonctions de base de l'interprète. Elles sont chargées dans l'environnement dès le lancement de l'interprète. Les *subroutines* sont des objets `Cell_subr` uniques et constants, qui disposent d'un pointeur vers une fonction associée de type `Object` vers `Object`. Les *subroutines* ne considèrent que des arguments déjà évalués, indépendemment du fait que l'expression ait un sens ou non. Par exemple :

```
Command? (- (display 1) (display 2) (display 3))
Read: (- (display 1) (display 2) (display 3))
123An error occured in a subroutine: -. Error: Number type needed.

Command? (- (display 3))
Read: (- (display 3))
3An error occured in a subroutine: -. Error: Not enough arguments.
```

Où `display` est une *subroutine* qui affiche son argument et renvoie nil, qui a une valeur de vérité fausse. Ici, les arguments ont été évalués avant l'entrée dans la *subroutine* `-`, c'est pourquoi on observe l'action de tous les `display` avant l'arrivée de l'erreur.

Les instructions sont un autre genre de fonctions de base, qui sont dites *lazy*. Cela signifie qu'elle n'évaluent pas toujours leur arguments. Elles permettent à l'utilisateur l'accès à des actions conditionnelles ou à des opérateurs logiques *lazy*. À l'instar des *subroutines*, les instructions possèdent un pointeur sur fonction, sauf que cette fois-ci la fonction est de type `(Object, Env)` vers `Object`. La donnée de l'environnement est nécessaire car c'est l'instruction qui décide quand appeler `eval`, ce qui nécessite de fournir un environnement.

On peut considérer l'exemple suivant, qui illustre que les arguments ne sont pas tous évalués :

```
Command? (if (display 1) (display 2) (display 3))
Read: (if (display 1) (display 2) (display 3))
13()
```

```
Command? (if (display 1))
Read: (if (display 1))
An error occured in a subroutine: if. Error: Not enough arguments.
```

Nous avons par ailleurs décidé de simplifier l'écriture de certaines *subroutines* ou instructions, en permettant de les appeler sur un nombre quelconque d'arguments.

```
Command? (+ 1 2 3 4 5)
Read: (+ 1 2 3 4 5)
15
```

4 *Toplevel*

Le *toplevel* est la boucle d'interaction principale. Il s'agit d'une boucle conditionnelle dans laquelle se suivent les opérations de lecture de l'entrée utilisateur, évaluation de l'entrée, et enfin retour de la valeur évaluée.

4.1 Directives

Les directives sont des commandes particulières qui permettent une interaction spéciale avec l'interprète. Leur détection est faite en amont de l'évaluation de l'entrée de l'utilisateur à l'aide d'une fonction `is_directive`. Dans un tel cas l'entrée n'est pas évaluée au sens de Lisp mais la directive est traitée. Un retour a toujours lieu pour l'utilisateur.

Les deux directives que nous avons implémentées sont `define` et `exit`.

La première a pour syntaxe : `(define symbol expr)`. Le traitement de cette directive modifiera alors l'environnement global pour y ajouter l'association entre le symbole `symbol` et la valeur correspondant à l'évaluation de l'expression `expr`.

La directive `exit` quant à elle est beaucoup plus simple : elle a pour effet de quitter l'interprète. Pour le faire, elle passe à faux le booléen `continue_lisp` qui est dans la condition de la boucle principale du *toplevel*. [1]

4.2 Rattrapage d'erreurs

Lors de l'utilisation de notre interprète, l'utilisateur, même bienveillant, peut tenter d'effectuer des opérations illicites telles que : `(+ 1 "test")` ou encore `(define 1)`. C'est l'une des raisons pour lesquelles il est nécessaire d'adopter une approche défensive en tant que programmeurs.

Dans la manipulation des objets Lisp que nous faisons en C++, aucune différence n'est faite a priori entre les différents types Lisp. Ainsi il est tout à fait possible d'appeler `object_to_number` sur un symbole par exemple. Avant chaque appel à cette fonction, une vérification du type est effectuée avec un prédicat tel que `numberp`, et dans le cas où ce prédicat n'est pas respecté, une exception est levée.

Différents types d'exceptions ont été écrits, chacun héritant de la classe C++ `runtime_error`. Ces différents types permettent de connaître la source de l'erreur ainsi que de contenir en attribut les informations nécessaires à son traitement. C'est au niveau du *toplevel* que les exceptions

1. Remarque : l'interprète rendu présente pour défaut de ne pas considérer le cas de `exit` pour la détection de directives. Ainsi bien que le coeur de la directive soit implémenté, elle n'est pas utilisable.

sont rattrapées, les différents types permettent alors d'écrire un message d'erreur le plus précis possible, permettant à l'utilisateur de comprendre le problème.

Remarquons qu'avec une telle démarche, il nous appartient d'imaginer tous les cas possibles d'erreur qu'un utilisateur peut générer. Or nous ne pouvons pas nous assurer avec une certitude absolue d'avoir recouvert tous ces cas. C'est pourquoi en plus des tests menant à l'appel de fonctions de la bibliothèque, les fonctions de l'API elles-même se prémunissent des cas indésirables grâce à des instructions `assert` dans leur code.

Par exemple nous nous sommes rendu compte après le rendu du code que l'exécution de l'entrée `(quote)` fait planter notre interprète. En effet nous avons oublié dans ce cas précis de nous assurer de l'existence d'un premier argument à côté du mot-clé avant de le récupérer. Toutefois la présence des `assert` nous a permis d'identifier précisément l'erreur, et ce assez rapidement, ici, un `car` appelé sans prise de précaution.

5 Extension

5.1 Correction assistée

Nous avons implémenté une extension de correction assistée pour ce projet. Le principe est de suggérer des corrections à l'utilisateur quand un symbole passé en entrée ne correspond à aucune liaison connue.

Cette extension se manifeste dans le programme lorsqu'une exception `No_binding_found` est rattrapée par le `Toplevel`. L'algorithme récursif suivant est alors effectué, renvoyant les chaînes les plus proches et leur distance :

1. on marque la chaîne de caractère `c` du symbole recherché ;
2. si l'environnement est vide, on renvoie l'objet `nil` et la distance entre la chaîne vide et `c` ; sinon, on parcourt l'environnement ;
3. à chaque liaison on transforme le symbole en une chaîne de caractère ;
4. on évalue la distance entre cette chaîne et celle recherchée ;
5. on observe le résultat de la recherche des chaînes les plus proches dans la suite de l'environnement ;
6. si la distance de la suite est plus petite que celle de la liaison considérée, on renvoie le résultat reçu ; si les distances sont égales, on ajoute l'objet considéré aux objets renvoyés ; sinon on renvoie l'objet considéré avec sa distance.

Il reste à définir une distance entre deux chaînes de caractères. Nous avons pris le parti d'utiliser la distance de la plus grande sous-séquence commune (PGSC). Celle-ci consiste à rechercher la plus grande sous-séquence communes aux deux chaînes, et retrancher sa taille à celle de la plus grande chaîne.

Définition. *La distance PGSC de c_1 à c_2 est définie comme $max(|c_1|, |c_2|) - d$,*
où d est la taille de la plus grande sous chaîne commune à c_1 et c_2

La distance employée est relativement satisfaisante, car elle considère de distance 1 les fautes de frappe de type insertion (augmente la longueur de la chaîne complète), omission (diminue la longueur de la PGSC), substitution (diminue la longueur de la PGSC), transposition (diminue de 1 la longueur de la PGSC, un seul des caractères transposés est considéré).

Cependant, elle reste perfectible car elle peut considérer de distance 1 la combinaison d'une faute de transposition et de substitution : par exemple, *car* et *anr* sont tous les deux à une distance PGSC de 1 de *var*.

Une distance permettant de parfaitement considérer les quatres fautes susmentionnées serait la distance de Damerau-Levhenstein [2]. Cela serait parfaitement implémentable en modifiant la fonction distance de notre code.

6 Conclusion

Nous avons implémenté un interprète Lisp en C++, et pour cela nous nous sommes servis d'une structure reposant sur un objet universel, la cellule. Les cellules spécialisées qui héritent de ses propriétés nous ont permis de centraliser toute l'information du programme de l'utilisateur dans des listes, que nous avons pu évaluer ensuite grâce aux méthodes `eval` et `apply` et à l'environnement dynamique. Nous nous sommes ensuite concentrés sur la création des *subroutines* et des instructions pour permettre à l'utilisateur de créer ses programmes en utilisant les fonctions usuelles. Nous avons mis en place un dispositif de rattrapage d'erreurs permettant de fournir à l'utilisateur les moyens de corriger son programme. Enfin nous avons créé un algorithme de recherche de mots voisins pour aider l'utilisateur à corriger d'éventuelles fautes de frappe.

Cependant, il s'est avéré que cet algorithme aurait pu être amélioré en modifiant la distance choisie. De plus, nous aimerions renforcer notre code vis-à-vis des appels à `(quote)` et à `(exit)`. Enfin une réflexion sérieuse est à fournir sur les performances de l'interprète : d'une part nous avons exhibé une méthode plus efficace de stockage de données via la possibilité de remplacer les listes par des arbres, mais à part pour créer l'environnement, nous ne l'avons pas utilisée. D'autre part, nous pourrions réfléchir à un moyen de recycler, voire même recycler dynamiquement, la mémoire utilisée par l'interprète.

2. Fred J. Damerau. A technique for computer detection and correction of spelling errors, *Communications of the ACM*, Mars 1964