[Prog2] Projet : un interprète Lisp en C++

Simon Coumes, Théo Losekoot, Odile Radet, Quentin Zanini

5 mars 2019

Résumé

Dans ce projet, nous avons réalisé un interprète Lisp écrit en C++. Ce rapport revient sur les choix effectués et sur certains détails de notre implémentations.

Introduction

Lisp est une famille de langages fonctionnels, qui dérivent directement de la notion de λ-calcul. L'objet fondamental manipulé par un dialecte Lisp est la liste (d'où le nom LISt Processing). Les listes sont délimitées par des parenthèses, à l'intérieur desquelles les éléments sont séparés par des espaces. De plus, la notation utilisée est préfixe.

On cherche ici à implémenter un interprète Lisp en C++, qui est capable d'évaluer des expressions Lisp reçues en ligne de commande.

Dans ce rapport, nous allons tout d'abord exposer, dans la partie 1, le fonctionnement global de l'interprète. Puis, nous détaillerons l'implémentation dans la partie 2, ainsi que les extensions dans la partie 3. Finalement, dans la partie 4, nous discuterons les décisions que nous avons prises au cours de ce projet.

1 Fonctionnement global de l'interprète

Dans cette section nous allons exposer les spécifications sémantiques de notre interprète. Nous n'y présenterons pas les raisons de nos choix, qui seront discutées en partie 4.

1.1 Eléments de sémantique donnés à l'avance

La base de sémantique à suivre dans le cadre de ce projet a pour origine la sémantique partagée par tous les dialectes Lisp et l'énoncé.

Nous évaluons des expressions Lisp une par une. Chacune de ces expression peut être un entier, une chaîne de caractères, un symbole, ou une liste. Si une liste est évaluée et qu'elle est non vide, son premier élément est vu comme une fonction ou une directive à appliquer aux autres éléments. Le but déclaré du projet était d'implémenter suffisament de fonctionnalités Lisp pour pouvoir coder dans notre interprète la fonction Fibonacci. On impose également que les opérateurs arithmétiques de base (+,-,*,/,mod) soient «eager», ce qui signifie que tous leurs arguments sont évalués avant d'être passés à l'opérateur.

1.2 Evaluation des arguments en position fonctionelle

Dans l'interprète Lisp Ocaml que nous avons étudié en TP au cours du semestre, l'élément en position fonctionnelle d'une liste n'est pas évalué mais seulement identifié. Nous avons fait le choix d'évaluer cet

élément avant de l'identifier. Cela fait par exemple une différence dans le cas suivant :

Exemple 1
L'évaluation de l'expression suivante renverra `+2` *dans notre interprète et renverra une erreur dans l'interprète présenté en TP.*

```
((display +) 1 1)
```

Du fait de ce choix de toujours évaluer l'élément fonctionnel d'une liste, il est nécéssaire que l'évaluation d'une fonction renvoie toujours cette même fonction.

1.3 Choix concernant les opérateurs

Au total, nous avons choisi d'implémenter les sous-routines, directives et structures de contrôle suivantes : +, -, *, /, <, >, mod, =, let, begin, display, cons, cond, if, quote, lambda, print_env, define.

Nous avons imposé que les sous-routines soient toutes *n*-aires, c'est-à-dire qu'elles prennent un nombre quelconque d'arguments. Ainsi, on définit récursivement les sous-routines. Par exemple, pour définir un opérateur qui canoniquement prend deux arguments :

- s'il n'y a pas d'argument, on renvoie le neutre pour l'opération que l'on cherche à implémenter ;
- s'il y a un argument, on renvoie cet argument (l'opérateur se comporte alors comme l'identité) ;
- s'il y a deux arguments, on renvoie l'opérateur appliqué aux deux arguments ;
- s'il y a strictement plus, on applique l'opérateur aux deux premiers arguments puit on procède à un appel recursif en prenant le résultat obtenu comme nouveau premier argument.

Exemple 2
L'expression suivante s'évalue en passant par les étapes suivantes :

```
(+ 1 1 1 1)
(+ 2 1 1)
(+ 3 1)
4
```

2 Détails de notre implémentation

Notre interprète est en C++. Nous allons commencer par présenter une vision globale de notre interprète, puis ensuite ses briques importantes.

2.1 Le parcours d'une entrée clavier

Le paragraphe suivant explique le parcours d'une évaluation, de l'entrée du texte au clavier jusqu'à la réponse de l'interprète. Il est une explication du schéma 1.

L'interprète est géré dans une boucle infinie, appelée toplevel. Ce toplevel, quand une commande est tapée dans l'interprète, appelle le fichier qui s'occupe de transformer le texte en objet Lisp. Cette action est réalisée grâce au fameux couple bison et flex. Ensuite, l'objectif est d'évaluer cet objet. L'objet est passé à l'évaluateur. L'évaluateur analyse le type d'objet qui lui est envoyé. Pour ce faire, les fonctions de librairie

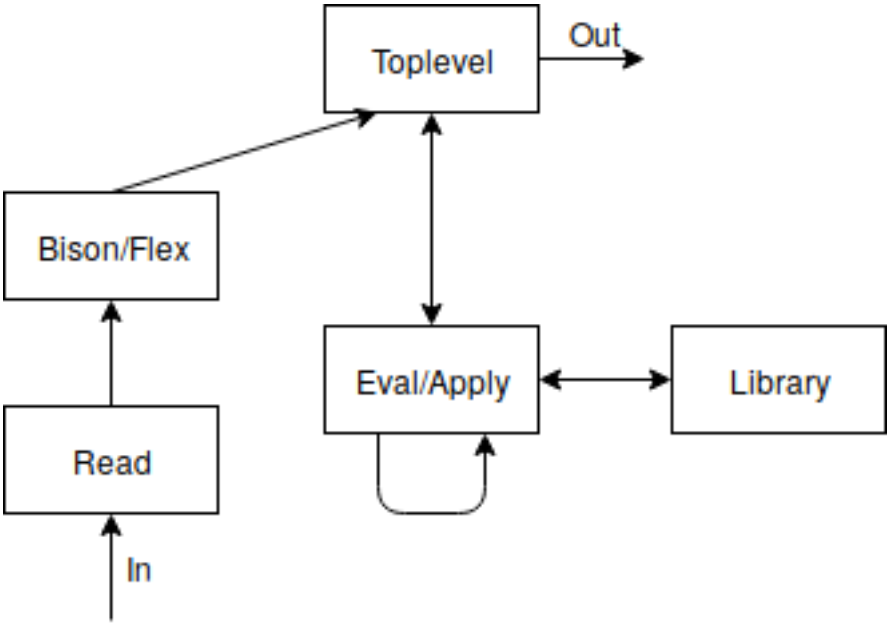

FIGURE 1 – Parcours de l'évaluation d'une expression Lisp, de l'entrée dans le terminal jusqu'à son retour.

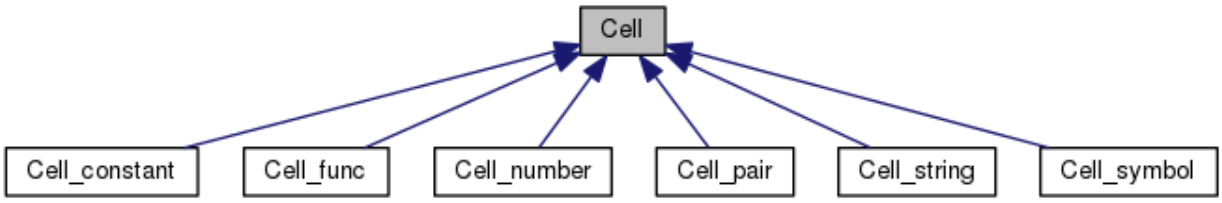

FIGURE 2 – Les classes filles de la classe Cell, représentant les types primitifs de Lisp.

sont utilisées. En fonction du type de l'objet (les différents types sont discutés dans la partie suivante), soit *eval* le renvoie tel quel, soit il effectue une des deux actions suivantes. Si c'est un *symbol*, *eval* cherche dans l'environnement. Si c'est une liste, alors il appelle *apply* après avoir ou non évalué les arguments (en fonction du type de l'objet en position fonctionnelle de la liste). *Apply* applique la sémantique de l'objet en position fonctionnelle de la liste aux autres éléments de la liste, puis renvoie le résultat. Dans certains cas, *apply* doit évaluer certains de ses arguments et, dans ce cas là, ré-appelle *eval*. L'implémentation de *eval* et *apply* est discutée en partie 2.3. La gestion des lambdas est discutée en partie 2.4.

2.2 Les cellules de base de Lisp

Dans notre interprète, les objets Lisp sont représentés par des classes C++ appelées *Cell*. Dans la suite de la partie, le terme *Objet* (sans précision supplémentaire) désignera un pointeur vers une cellule.

Il est possible de représenter les objets Lisp en 6 types C++ différents. Pour ce faire, nous avons implémenté une classe *Cell*, et 6 classes en héritent, cf 2

Ces types sont :
— *constante*, qui représente les valeurs suivantes : *nil*, *true* et *false* ;
— *func*, qui représente toutes les fonctions, qu'elles soient lazy (structures de contrôle), ou eager (sous-routines) ;
— *nombre*, qui représente les entiers naturels ;
— *pair*, qui représente une paire d'objets ;
— *string*, qui représente une chaine de caractère ;
— *symbol*, qui représente le reste des mots.

Ces objets sont ensuite abstraits dans une classe appelée API. Après ce niveau, nous ne manipulerons plus chacune des cellules individuellement, mais simplement un objet. Des fonction et prédicats nous permettrons d'intéragir avec cet objet sans en entrer dans les détails. Les paires seront également abstraites en listes.

Nous avons également défini un fichier *library* pour abstraire encore certaines fonctions et être certains de ne plus être en contact des détails de l'implémentation bas niveau.

2.3 Eval/Apply

2.3.1 Eval

La fonction *eval* est la fonction qui est appelée lorsqu'un objet doit être évalué. Elle est découpée en quatre parties comme suit : *eval*, *eval_core*, *eval_fun* et *eval_list*.

eval ne fait, chez nous, qu'appeler la fonction *eval_core* car un mode de débogage n'a pas été implémenté.

L'action d'*eval_core* est déterminée par le type de l'objet à évaluer. Voici la table de disjonction des actions en fonction du type de l'objet.

1. Type *liste* : si la liste est vide, renvoie *nil*, sinon appelle *eval_fun*.
2. Type *symbole* : recherche ce symbole dans l'environnement. S'il y est, renvoie sa valeur associée. Sinon, lance une erreur.
3. Sinon : renvoie l'objet tel quel.

L'action d'*eval_fun* est déterminée par le type du premier objet de la liste reçue en argument. Voici la table de disjonction des actions en fonction du type de l'objet.

1. Type *func*, et cette *func* est *eager* : évalue tout le reste de la liste grâce à la fonction *eval_list*, puis appelle *apply_eager* en lui passant en arguments les valeurs.
2. Type *func*, et cette *func* est *lazy* : appelle *apply_lazy* en lui passant en argument le reste de la liste sans les évaluer, et également l'environnement.
3. Une liste ayant lambda comme premier élément lambda : appelle *apply_lambda* avec comme paramètre l'évaluation de la liste (celle reçue en paramètre de *eval_fun*).
4. Sinon : lance une erreur expliquant qu'on ne peut pas appliquer cet objet (car ce n'est pas une *func*)

eval_list se contente d'évaluer la liste de ses arguments et de renvoyer l'objet Lisp liste correspondant.

2.3.2 Apply

Le rôle de *apply* est divisé en trois fonctions, *apply_eager*, *apply_lazy* et *apply_lambda*. *apply_eager* est une fonction appliquant le premier élément de la liste qu'il reçoit en paramètre aux autres éléments de cette liste.

apply_lazy est une fonction appliquant le premier élément de la liste qu'il reçoit en paramètre à l'évaluation de certains autres éléments de la liste qu'il reçoit en paramètre. Les éléments évalués dépendent de la *func* qui est en train d'être appliquée, donc le premier élément de la liste.

apply_lambda sera discutée dans la partie suivante.

2.4 Les lambdas

Dans cette partie, considérons les termes suivants :

— Une *fonction* lambda est un objet de type *func* qui correspond à la transformation du *symbole* lambda en *func* lambda.
— Une liste lambda symbolique est une liste dont le premier élément est le *symbole* lambda.
— Une liste lambda fonctionnelle est une liste dont le premier élément est une *func* lambda.
— Une application lambda est une liste dont le premier élément est une liste lambda fonctionnelle.

Notons que l'évaluation du symbole lambda renvoie la *func* lambda. Ce choix est discuté dans la partie Discussion.

Une *func* lambda étant une *func* comme les autres (du moins dans ce cas), si une liste lambda fonctionnelle est évaluée, alors elle rentre dans la condition pour être évaluée par *apply_lazy*.

Lorsqu'une liste lambda fonctionnelle est évaluée, alors *eval_lazy* la renvoie telle quelle.

Pour appliquer une liste lambda fonctionnelle à des arguments, il faut que l'objet à évaluer soit une application lambda. C'est précisement ce cas que capture la fonction *apply_lambda*. Dans ce cas, les arguments sont évalués dans *eval_fun* avec d'être envoyés à *apply_lambda*.

Dans *apply_lambda*, l'environnement est augmenté en ajoutant les liaisons $(v_1 : p_1; ...; v_n : p_n)$, les V étant les variables définies dans la liste lambda et les V les arguments de l'application lambda. Ensuite le corps de la liste lambda est simplement évalué puis renvoyé.

Notons que l'évaluation d'une liste dont le premier élément est une liste lambda symbolique conduit à une erreur disant qu'il est impossible d'appliquer une liste à un objet. Ce cas peut être atteint si l'on écrit, par exemple, cette commande : `( (quote (lambda (n) (+ n 1))) 5)` .

3 Extensions proposées

Nous avons implémenté trois extensions pour notre interprète Lisp.

3.1 Gestion hiérarchique des erreurs

La première de nos extensions consiste en une gestion hiérarchique des erreurs. Autrement dit, les erreurs suivent le fil d'exécution ayant conduit à ces dernières (voir exemple 3).

Exemple 3

```
1 > (if 4)
=> Lisp error : evaluation error : application error : if error : if with one argument.

2 > (define v (+ 1 a))
=> Lisp error : error in define : evaluation error : application error : evaluation error : bad symbol : unknown sy
mbol : a.

3 > v
=> Lisp error : evaluation error : bad symbol : unknown symbol : v.
```

n	Cellules crées lors de l'appel de la fonction	Ratio i/(i-1)
5	158	NONE
10	1 859	11,77
15	20 717	11,14
20	229 856	11,10
25	2 549 243	11,09
30	28 271 639	11,09

TABLE 1 – Nombre de cellules créées lors de l'appel à la fonction de Fibonacci.

Pour cela, toutes les fonctions en mesure de vérifier les données entrée, comme les fonctions de typages, sont en mesure d'envoyer une chaîne de caractères lors d'une erreur à l'aide de la commande C++ *throw*. Ensuite, toutes les fonctions ou instruction prenant en argument des objets Lisp sont en mesure de rattraper une chaîne avant de la renvoyer précédée d'un message indiquant ladite fonction. Enfin, lors de l'arrivée au *toplevel*, ce dernier convertit la chaîne en objet Lisp avant de l'afficher. De plus, lorsque le message d'erreur possèdent plus de trois cent,caractères, seuls les cent premier et derniers caractères sont conservés afin d'éviter une erreur trop longue.

3.2 Décompte des cellules instanciées

Afin de pouvoir visualiser le nombre de cellules instanciées au cours d'une exécutions, et donc de pouvoir observer les fuites mémoire, nous avons également implémenté un décompte des cellules instanciées. Pour cela, nous avons ajouté à la classe *cell* un paramètre `static int counter_cells;` qui s'incrémente à chaque création de cellule.

Nous pouvons ainsi nous intéresser aux fuites mémoires induites par la fonction de Fibonacci naïve que nous avons implémentée. Pour cela, nous nous sommes intéressés à l'augmentation de la quantité de cellules créées lorsque les données en entrée augmentent linéairement. Les résultats nous montrent ainsi que le nombre de cellules crées augmente énormément avec la taille des données en entrée.

L'initialisation de Lisp et la définition de la fonction nécessitent la création de 160 cellules. Le nombre de cellules crée après l'utilisation est donné par la table 1. *n* correspond à l'entier pris en argument par la fonction et la colonne *Ratio* contient le quotient de la ligne considérée par la ligne précédente.

On observe ainsi une croissance géométrique du nombre de cellules créées lorsque l'argument croît linéairement. De fait, la fuite mémoire croît exponentiellement en fonction des données entrées.

Plus généralement, cette extension peut s'avérer utile pour l'utilisateur qui pourra ainsi être conscient de la consommation mémoire de ses fonctions.

3.3 Macro-expansion

Pour faciliter l'expérience utilisateur, nous avons implémenté deux macro-expansions. La première étant pour l'instruction *quote* et la seconde est liée à la définition des λ-expressions.

La première macro-expansion se définit en remplaçant *('expr)* par *quote expr* où *expr* est une expression Lisp quelconque (voir exemple 4). Cette dernière est implémentée directement dans le lexer.

Exemple 4

```
1 > (define l (quote (May Lisp be with you)))
=> New definition : (l -> (May Lisp be with you))

2 > (define l '(May Lisp be with you))
=> New definition : (l -> (May Lisp be with you))
```

La seconde permet de définir une λ-expression en se passant du terme *lambda* (voir exemple 5). Cette dernière est implémentée directement dans la gestion du *define* au *toplevel* car elle nécessite un accès plus fin aux éléments manipulés, ce qui est plus simple au *toplevel* que dans le lexer.

Exemple 5

```
1 > (define f (lambda (n) (+ 1 n)))
=> New definition : (f -> (lambda (n) (+ 1 n)))

2 > (define (f n) (+ 1 n))
=> New definition : (f -> (lambda (n) (+ 1 n)))
```

4 Discussion

Le premier choix que nous avons fait a été d'imposer que les sous-routines sont toutes n-aires, c'est-à-dire qu'elles prennent un nombre quelconque d'arguments.

Exemple 6
Par exemple, `(+ 1 2 3)` *renverra* $1 + (2 + 3)$ *c'est-à-dire 6.*

Exemple 7
Par exemple, `(- 1 2 3)` *renverra* -4.

Exemple 8
Par exemple, `(< 1 2 3)` *renverra* `#t`.

Ce choix peut sembler arbitraire. En réalité, la motivation principale était la suivante : puisque les sous-routines que nous avons implémentées sont des fonctions de type *eager*, elles évaluent tous leurs arguments. Ainsi, pour éviter d'évaluer tous les arguments pour ensuite ne renvoyer qu'une erreur (signalant que le nombre d'arguments étaient incorrect), nous avons préféré étendre les sous-routines afin de pouvoir traiter ces cas.
Cependant, si ce choix peut paraître pertinent pour l'opérateur $+$ par exemple (on peut supposer que la sémantique de `(+ 1 2 3)` semble au premier abord relativement claire), il souffre deux inconvénients majeurs :

— en premier lieu, le sens que l'on donne à la soustraction appliquée à un nombre quelconque d'arguments n'est pas évident (car non canonique) et entame la qualité de l'expérience utilisateur. De même, les autres opérateurs n-aires peuvent poser des problèmes de compréhension, comme la division euclidienne avec un nombre quelconque d'arguments, ou encore la comparaison en chaîne. De plus, déterminer un élément neutre pour l'opération *modulo* n'est pas un choix canonique.

— D'autre part, cela empêche l'utilisateur de définir des sous-routines non *n*-aires. En effet, il est certes possible, lorsque l'opérateur à définir est par exemple binaire, et renvoie un objet de même type que ses arguments, d'étendre récursivement cet opérateur. Cependant, si cet opérateur ne renvoie pas le même type, alors il ne sera pas possible, dans notre implémentation, d'ajouter cette sous-routine.

Une solution à laquelle nous n'avons pas pensé durant le projet aurait été de vérifier l'arité et le type des paramètres avant de tous les évaluer dans une fonction *eager*.

Nous avons également décidé que les mots clefs tels que *if*, $+$, *lambda*, ou toutes les *func* ne seront pas gérés en symbolique, mais grâce à un objet particulier (la classe *Cell_func*). Dans ce procédé, nousa vons défini que pour pouvoir appliquer un fonction aux autres arguments de la liste, il est nécessaire que cet argument ait été évalué, et soit donc dans une forme de *func* et non pas de *symbole*. Il y a donc une disctinction de faite en le *symbole* et la *fonction*. Cette distinction a pour effet, entre autres, de pouvoir évaluer `(+ 1 2)` mais pas `('+ 1 2)`, et que `(= + +)` rende vraie mais pas `(= + '+)`.

Enfin, l'implémentation de la coloration dans le terminal suppose que le terminal respecte la norme VT100, ce qui n'est pas nécessairement le cas. Il faudrait donc envisager de laisser à l'utilisateur la possibilité de désactiver la coloration.

5 Conclusion

En conclusion, nous proposons ici un interprète Lisp fonctionnel, avec une gestion hiérarchisée des erreurs. Si nous avions disposé de plus de temps, nous aurions pu envisager de gérer les fuites mémoire à l'aide d'un ramasse-miettes (*garbage collector*). De plus, la gestion des continuations ou de `call/cc` aurait pu être une extension intéressante à mettre en oeuvre.