Projet 3 : Interprète Lisp en C++

Alexandre Drewery, Matthieu Gillet, Maé Mavromatis, Kerian Thuillier
ENS Rennes – ISTIC Université Rennes 1
L3 Informatique – parcours SIF

5 Mars 2019

Résumé

Compte rendu du projet 3 qui consiste en la réalisation d'un interprète Lisp en C++. Ce rapport présente notre approche du problème et retrace les différentes étapes de conception.

Mots-clefs : sémantique, interprète Lisp, C++
Classification ACM : F.3.2

Table des matières

Introduction

Lisp est un langage de programmation fonctionnel développé par McCarthy en 1958. L'objectif de ce projet est la réalisation d'un interprète Lisp dynamique en C++ permettant d'exécuter les fonctions factorielle et de Fibonacci.

Nous allons présenter notre travail en suivant les différentes étapes que nous avons réalisées. Nous commencerons donc par l'implémentation des objets de base, les briques élémentaires du projet, puis nous présenterons un premier évaluateur arithmétique. Nous nous intéresserons ensuite à l'amélioration de cet évaluateur en lui ajoutant la notion d'environnement et de λ-expressions. Enfin, nous présenterons quelques extensions que nous avons réalisées.

1 Implémentation des objets de base

Notre premier objectif est de définir les objets Lisp et d'implémenter l'API (*application programming interface*) pour pouvoir lire et écrire des objets Lisp. Ici, afin de partir sur des bases sûres, nous avons employé des techniques de **programmation défensive** (utilisation des `assert` en particulier).

1.1 La classe `Cell` et ses filles

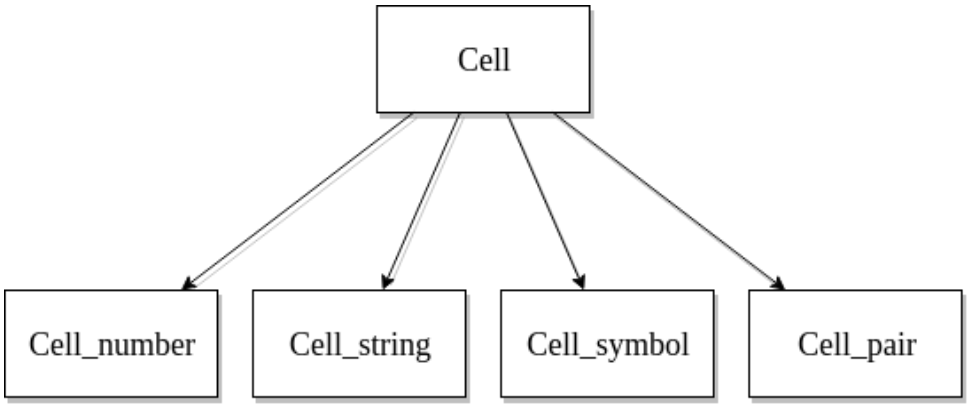

FIGURE 1 – Héritage de la classe `Cell`.

Dans notre interprète, les objets sont des pointeurs vers des cellules `Cell`. Nous avons donc implémenté (cf. figure 1) une classe virtuelle `Cell`, dont les filles sont les classes `Cell_number`, `Cell_string`, `Cell_symbol`, et `Cell_pair`. Une attention toute particulière a été portée au suivi des pointeurs : de nombreuses vérifications viennent assurer que les objets manipulés sont bien des cellules valides.

Pour simplifier le code, nous avons défini `Object` comme un alias de `Cell*`.

1.2 API

Notre API sert, comme son nom l'indique, à faire interface entre entre le programmeur et l'utilisateur. Le diagramme de classes en figure 2 présente bien cette jonction.

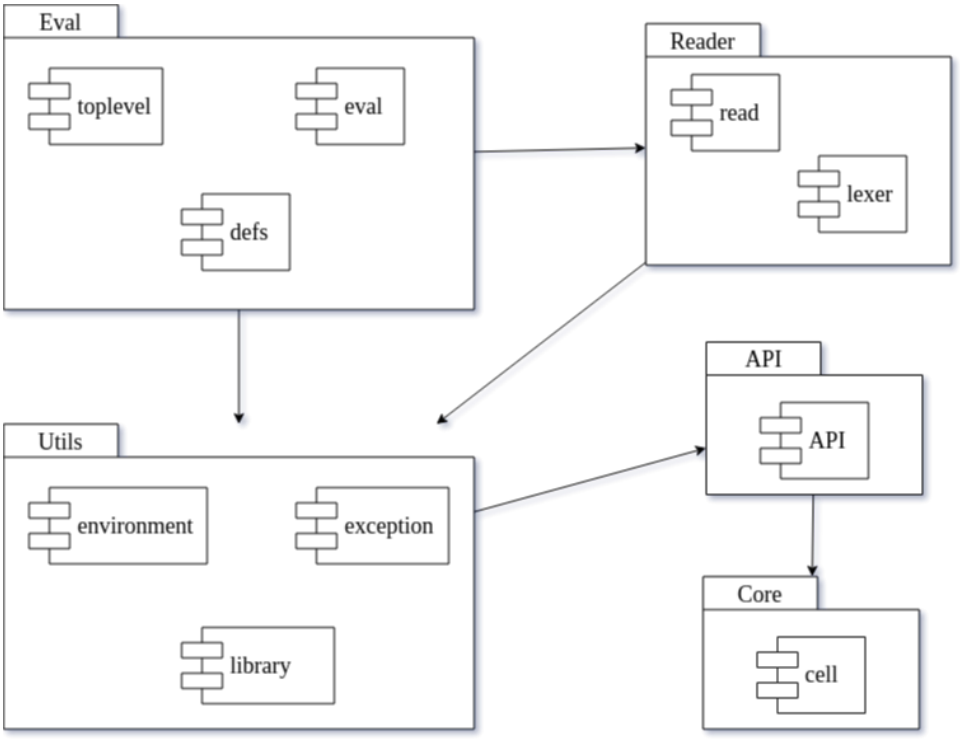

FIGURE 2 – Diagramme de classes.

L'API permet :
— la déclaration des objets `nil` , `#t` (*true*) et `#f` (*false*) ;
— la vérification du type des objets ;
— la conversion d'un objet Lisp vers un objet C++, et réciproquement.
Ces fonctions seront reprises dans la bibliothèque, afin qu'elles puissent être utilisées par des modules plus haut niveau, de manière opaque.

2 Bibliothèque standard

Notre nouvel objectif est la mise en place d'une bibliothèque standard élémentaire permettant de réaliser des opérations arithmétiques.

2.1 Fonctions `eval` et `apply`

Les fonctions `eval` et `apply` sont les fonctions majeures de l'interpréteur. La première permet l'évaluation des données tandis que la seconde sert à appliquer des structures de contrôle ou des fonctions à ces données.

Ainsi, la fonction `eval` est utilisée pour évaluer l'expression fournie. C'est elle qui distingue les structures de contrôle comme `if` ou `cond` . L'ensemble des arguments d'une fonction est évalué avant d'être passer à `apply` . La fonction `apply` permet quant à elle de déterminer les fonctions à appliquer sur le jeu de données.

Ces fonctions procèdent par disjonction de cas, c'est-à-dire qu'elles déterminent un trai-

tement à effectuer dépendant de la nature de l'objet fourni. Un exemple d'itération de ces fonctions sur l'entrée `(* 2 a)` est visible sur la figure 3.

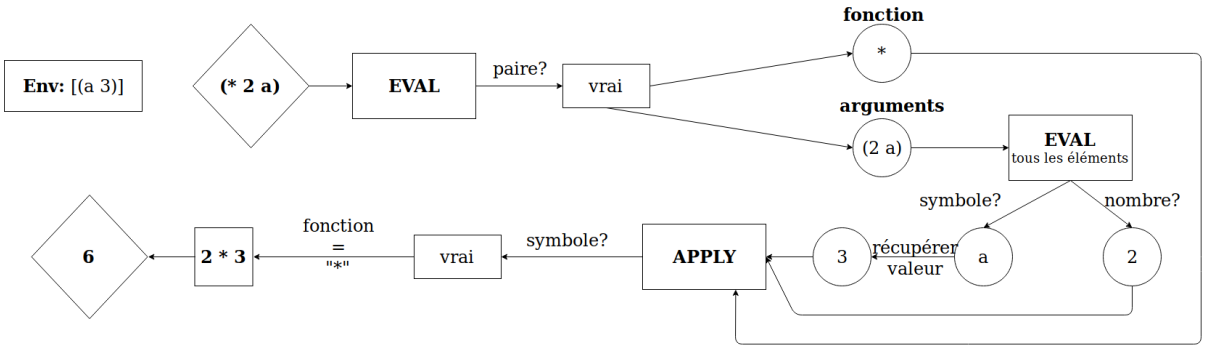

FIGURE 3 – Exemple du traitement de la commande `(* 2 a)` par les fonctions `eval` et `apply`. L'environnement est initialisé avec `a` $= 3$.

2.2 *Subroutines*

L'interpréteur implémente un nombre important de fonctions pré-codées, les *subroutines*. Celles-ci permettent de réaliser des opérations arithmétiques élémentaires (`+`, `*`, `=`, ...), des opérations basiques sur les listes (`cons`, `car`, `cdr`) et d'autres fonctionnalités de base (`display`, identifier le type d'un objet, ...).

Chaque *subroutine* correspond à une fonction C++ dans notre fichier `defs.cc`. `apply` se charge d'identifier l'instance à appeler lorsqu'on lui demande d'appliquer un symbole, par correspondance entre ledit symbole en entrée et une série de symboles associés aux *subroutines*.

Ces fonctions basiques sont donc intégrées à notre interpréteur, ce qui se traduit, à terme, par une une difficulté accrue du procédé d'ajout d'une *subroutine*. Nous avons également considéré la définition des *subroutines* dans l'environnement, mais nous n'avons pas retenu cette option, craignant qu'elle complexifie la gestion future de l'environnement. Ainsi, il est impossible pour l'utilisateur de redéfinir une *subroutine* via l'interpréteur.

3 Évaluateur simple

Nous avons ensuite implémenté les fonctions de la bibliothèque standard concernant l'environnement et la fonction `add_new_binding`, ainsi que l'évaluateur d'expressions arithmétiques avec variables dans l'environnement.

3.1 Les fonctions de l'environnement

De manière interne, l'environnement est géré comme n'importe quelle liste dont les éléments sont des couples *(symbole, valeur)*.

Il dispose néanmoins de fonctions particulières qui lui sont propres :
— `make_env` initialise un environnement ;

— `print_env` affiche l'environnement ;
— `find_value` trouve la valeur associée à un symbole en parcourant l'environnement ;
— `add_new_binding` ajoute un couple *(symbole, valeur)* dans l'environnement.

Notre interpréteur est dynamique, c'est pourquoi la recherche dans l'environnement est nécessairement linéaire. La redéfinition d'une variable masquera ses précédentes définitions.

3.2 La directive `define`

Lors de l'appel de `define` sur un symbole et une expression, l'interprète commence par évaluer l'expression grâce à la fonction `eval`. Il ajoute ensuite à l'environnement le couple *(symbole, expression évaluée)*, à l'aide de la fonction `add_new_binding` définie précédemment.

Cette directive est récupérée au niveau *toplevel*, cela signifie qu'elle est interceptée par la boucle principale avant d'accéder à la fonction `eval`. Un traitement adapté est alors effectué, il consiste ici à mettre à jour l'environnement.

3.3 L'évaluateur d'expressions arithmétiques avec variables d'environnement

À ce stade du projet, nous avons un évaluateur capable de réaliser des opérations arithmétiques, stocker et accéder à des variables dans un environnement. Un exemple de calcul est visible sur la figure 4.

```
>>> (define a 2)      ; Env: [(a 1)]
Define: a = 1
>>> (+ a 1)
3
```

FIGURE 4 – Exemple d'exécution sur l'évaluateur simple.

4 Évaluateur avec λ-expressions

4.1 Traitement des λ-expressions

Les λ-expressions sont des instructions de la forme : `((lambda (args) (body)) values)`. Lorsque l'évaluateur reçoit de telles instructions, il va isoler les arguments (`args`) et les valeurs associées (`values`) ; il va ensuite former des paires à partir des éléments de même rang, qu'il ajoute temporairement à l'environnement courant. Notons que dans le cas où ces deux listes ne sont pas de même longueur, une erreur est levée.

Il suffit par la suite d'évaluer le corps de l'expression (`body`) avec l'environnement étendu précédemment modifié. L'évaluation retourne alors la valeur de la λ-expression pour ce jeu d'arguments. Un exemple d'utilisation est présenté sur la figure 5.

```
; Env: []
>>> ((lambda (x y) (+ x y)) 2 3)
; Ajoute [(x 2), (y 3)] à l'environnement étendu temporaire
5
```

FIGURE 5 – Exemple d'utilisation d'une λ-expression.

4.2 Quelques instructions

Pour enrichir notre interprète, nous avons défini les instructions suivantes :
— `quote` permet de ne pas évaluer une expression ;
— `if` permet de faire des tests ;
— `cond` permet de faire un *matching* ;
— `let` permet de définir et d'évaluer une expression en un point ;
— `lambda` permet de définir des fonctions.
Ces instructions sont identifiées dans la fonction `eval` par une disjonction de cas sur le premier élément des listes.

4.3 Macro-expansion de `let` et `lambda`

Afin de faciliter l'utilisation de notre interprète, nous avons ajouté les macro-expansions de `let` et `lambda`, qui sont présentées dans la table 1.

Expression	Expression macro-expansée
`(define (f x y) (+ x y))`	`(define f (lambda (x y) (+ x y)))`
`(let ((a 3) (b 4)) (* a b))`	`((lambda (a b) (* a b)) 3 4)`

TABLE 1 – Macro-expansion d'expressions.

La macro-expansion de `define` pour les λ-expressions est exécutée au niveau du *toplevel*. La fonction d'évaluation n'a ainsi qu'un seul cas à traiter pour les λ-expressions.
À l'inverse, la macro-expansion de `let` est réalisée lors de l'évaluation. La fonction évaluatrice reçoit la commande `let`, puis la macro-expanse juste avant de l'évaluer.

4.4 Factorielle et Fibonacci

Présentée en figure 6, les fonctions factorielle et de Fibonacci ont été nos premières véritables fonctions à tester l'interprète complet. Elles ont notamment permis de tester la récursivité, ainsi que les fonctions `if` et `cond`. Leurs exécutions dans notre interprète sont visibles sur cette même figure.

5 *Toplevel* avec rattrapage d'erreurs

En l'état, l'interpréteur est fonctionnel, mais s'interrompt à la moindre erreur de l'utilisateur. Afin d'améliorer l'expérience, nous avons implémenté un système de rattrapage d'erreurs

```
; Fonction factorielle ;
(define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
; Fonction de Fibonacci ;
(define fib (lambda (n)
  (cond ((= n 0) 0) ((= n 1) 1) (#t (+ (fib (- n 1)) (fib (- n 2)))))))
; Interprète ;
>>> (fib 0)          |   >>> (fact 1)
0                    |   1
>>> (fib 4)          |   >>> (fact 6)
3                    |   720
```

FIGURE 6 – Exécution des fonctions factorielles et de Fibonacci dans notre interprète.

avec un message assistant dans l'identification et la correction de l'erreur.

Dans la majorité des cas d'erreur, le rattrapage se fait en deux temps. Au niveau de notre librairie, chaque opération illégale (application de `car` à un objet qui n'est pas une liste non-vide par exemple) renvoie une erreur `runtime_error` (de la bibliothèque standard C++). Ces `runtime_error` sont ensuite récupérées au niveau des *subroutines*, où une instance d'une classe d'erreur C++ personnalisée est renvoyée, avec un message destiné à l'utilisateur. Ce message est affiché lors du rattrapage au niveau du *toplevel*.
Cette récupération en deux étapes permet, en positionnant le harnais `try` - `catch` au sein de la fonction d'une *subroutine*, de cerner les actions illégales qui engendrent l'erreur et de rédiger le message à l'utilisateur en conséquence. Afin de garder le code des *subroutines* clair, nous avons tenté de limiter le nombre de messages différents renvoyés (voir figure 7).

```
>>> (+ 1 "Timothée")
Error in evaluation of object (1 "Timothée"):
+ does take two number arguments.
>>>
```

FIGURE 7 – Un message d'erreur qui indique le nombre et type des arguments requis pour `+`.

D'autres erreurs plus spécifiques (caractère inconnu, nombre invalide d'arguments donnés à une fonction de l'utilisateur) se passent de ce rattrapage en deux étapes et renvoient immédiatement une exception personnalisée.

6 Extensions

Après avoir obtenu un interprète opérationnel, nous avons choisi d'implémenter certaines fonctionnalités supplémentaires.

6.1 Fonction de chargement : `load`

Afin de permettre à l'utilisateur une plus grande modularité du code qu'il réalise, nous avons implémenté la fonction `load`. Elle permet de charger en mémoire, soit dans l'environnement courant, les données d'un fichier externe (exemple en figure 8).

```
; File "fib.lsp" ;
(define fib (lambda (n) (if (= n 0) 1 (if (= n 1) 1
(+ (fib (- n 1)) (fib (- n 2))))))
; Interprète ;
>>> (printenv)
[]
>>> (load "fib.lsp")
Loading file: fib.lsp
File loaded.
>>> (printenv)
[(fib = (lambda (n) (if (= n 0) 1 (if (= n 1) 1  (+ (fib (- n 1)) (fib (- n 2))
))))))]
```

FIGURE 8 – Exemple d'utilisation de la directive `load`.

Cette fonction est une directive *toplevel*, c'est-à-dire qu'elle est récupérée dès la boucle principale et ne passe pas par l'évaluateur. Cela signifie qu'il est impossible d'exécuter cette directive depuis des structures telles que `begin`. Ce choix a été réalisé afin de ne pas avoir de conflit lors de chargements successifs de fichiers. En effet, il aurait été possible de commencer une requête dans un fichier et de la finir dans un autre.

Pour réaliser cette fonction, il faut modifier le flux d'entrée de l'analyseur syntaxique et le remplacer par le fichier désiré. L'analyse de ce fichier réalisé, le flux d'entrée est réinitialisé à sa valeur avant modification.

Nous notons que la mémoire tampon n'est pas vidée à chaque modification du flux. Vider la mémoire tampon empêcherait l'exécution des commandes saisies après la commande `load`. Cette approche permet donc le chargement en cascade de fichiers, voir la figure 9.

```
; Fichier A
(load "b.lsp")
; Fichier B
(load "c.lsp") (display "b")
; Fichier C
(display "c")
; == display ==;
cb
```

FIGURE 9 – Exemple d'un chargement en cascade de fichiers.

6.2 Coloration

Pour permettre une meilleure expérience à l'utilisateur et une meilleure compréhension, nous avons décidé de colorer certains messages texte, comme présenté en figure 10. Ainsi, les messages d'erreur sont affichés en rouge, les indications de chargement de fichier sont en vert et les modifications de l'environnement apparaissent en jaune.

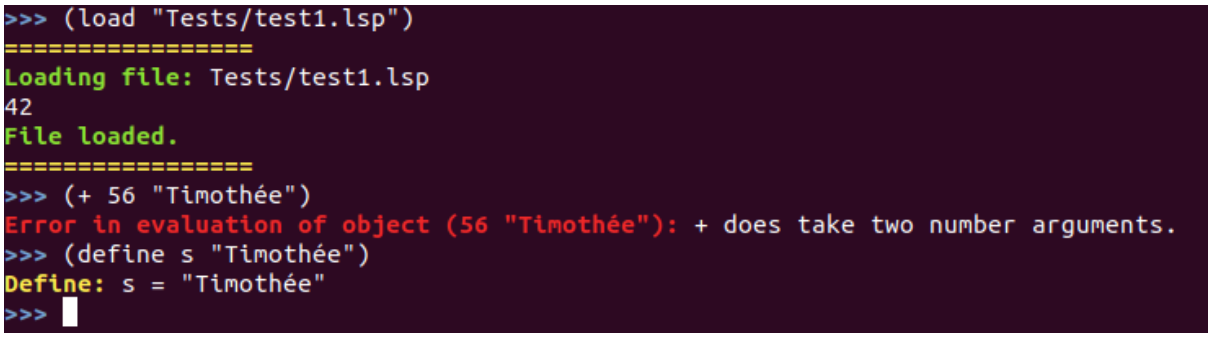

FIGURE 10 – Exemple de coloration du texte affiché dans un terminal.

Conclusion

Pour conclure, nous avons réalisé un interprète Lisp en C++. Pour cela, nous avons progressé par étapes en réalisant tout d'abord un interprète basique sans mémoire et limité aux opérations arithmétiques. Nous l'avons ensuite amélioré en lui ajoutant un environnement et le traitement des λ-expressions. L'objectif de pouvoir exécuter les fonctions de Fibonacci et factorielle a été atteint.

Nous avons pris plaisir à aborder ce projet. Écrire l'interprète Lisp – un langage fonctionnel, sans structure et interprété – en C++ – un langage impératif, orienté objet et compilé – nous a permis d'approcher la notion de sémantique en profondeur.

Notre groupe a bien fonctionné. Notre organisation en amont et une bonne communication nous ont permis de nous impliquer et de nous soutenir dans notre travail. Nous avons ainsi évolué sereinement et avons bien respecté les délais que nous avions estimé en début de projet.

Auto-évaluation

Forces. Nous nous sommes bien organisés, avec un dépôt git correctement utilisé par tous. Une bonne communication nous a permis de nous débarrasser de nos obstacles rapidement.

Faiblesses. Nous avons passé notre code dans *valgrind*, et nous avons remarqué quelques fuites mémoires. Avec plus de temps, nous aurions pu implémenter un *garbage collector* pour résoudre ce problème.

Opportunités. Ce projet nous a permis de nous améliorer en C++ et de découvrir Flex et Bison.

Menaces. Nous avons eu quelques soucis de dispersion.