Compte-Rendu Projet POO
Interpréteur LISP

Quentin Dunand Rémi Gattaz Jules Hablot Elsa Navarro

12 mai 2015

Table des matières

1 Présentation du sujet

Ce projet a été réalisé dans le cadre de la mise en pratique de nos connaissances en programmation orienté objet grâce au langage **JAVA**.

Il consiste à réaliser une modélisation objet en utilisant un interpréteur pour le langage **LISP** en langage **JAVA**. Le but est de concevoir un programme structuré mettant en évidence les concepts majeurs ainsi que d'élaborer un programme qui peut évoluer dans le temps.

2 Récapitulatif rapide

Dans le cadre de ce projet, voici une liste de toutes les fonctionnalités que nous avons développées :

- Lecture d'une expression LISP à l'aide un parser javaCC
- Lecture d'une expression LISP à partir d'une String
- Lecture d'une expression LISP à partir d'un Buffer sur un fichier
- Évaluation d'expressions LISP en mode interactif
- Évaluation d'expressions LISP depuis un fichier

Pour permettre l'évaluation d'expressions LISP, voici alors les différentes primitives qui ont été mises en place :

- Apply
- Atom
- Car
- Cdr
- Cond
- Cons
- De
- Df
- Eprogn
- Eq
- Eval
- Explode
- Implode
- List
- Load
- Print
- Quit
- Scope
- Set
- Toplevel
- Typefn

Il s'agit donc de toutes les primitives qui étaient proposées par le sujet. Nous avons cependant rajouté la primitive **List** car nous n'avons pas trouvé de fonction **Lambda** ou **FLambda** permettant de définir cette fonction.

3 Plus de détails

3.1 Démarrage de l'interpréteur

3.1.1 Initialisation de la machine LISP

Lors de l'initialisation de notre machine LISP, le système rajoute dans le contexte toutes les primitives LISP qui seront reconnues. Il ajoute aussi la fonction **FExpr** *"quote"*. Ceci est nécessaire puisque la grammaire que nous avons définie remplace toutes les instructions $'X$ par $(quote\ X)$.

3.1.2 Ajout de prédicats et fonctions

Nous avons fait le choix de proposer un environnement de départ minimal. Avec la primitive **Load**, il est cependant très facile de charger un grand nombre de fonction usuelles. Le fichier ressources/lisp.txt contient quelques exemples de prédicats et fonctions classiques.

3.1.3 Lancement de la primitive Toplevel

Finalement, le lancement de notre interpréteur se traduit concrètement par l'appel de la primitive **Toplevel** qui permet d'interpréter des instructions LISP. Cette primitive effectue une boucle infinie et évalue toutes les SExpr rentrées par l'utilisateur. C'est l'appel à la primitive *quit* qui permet alors de quitter l'application.

3.2 Évaluation des SExpr

Voici quelques choix que nous avons fais pour l'évaluation des SExpr.

Comme indiqué précédemment, nous avons défini une classe MachineLisp. Contenant le contexte de la LVM, cet objet est transmis à toutes les étapes de l'évaluation d'une SExpr. Cette objet ne contient à l'heure actuelle qu'un contexte. On peut donc penser que remplacer cet object directement par l'objet représentant notre Contexte est possible. Cependant, ce choix avait été fait dans l'optique d'une évolution de cette application. C'est également dans cette optique que nous avons décidé de ne pas définir cet object comme une "variable globale" de notre application mais de la passer à chaque fonction.

Nous avions définis que quelque soit la liste, la première étape de l'évaluation d'une **SCons** était l'évaluation du foncteur. C'est cependant faux car la gestion des Lambdas implique forcément une exception. En effet, nous voyons que dans le cas ou le premier élément de la **SCons** est un symbole lié à une fonction dans le contexte, la symbole est remplacé par un expression lambda. Or, si l'on définit une SExpr à évaluer qui contient directement une expression lambda, avant l"évaluation du premier élément de cette liste, on y trouvera alors une expression lambda. Évaluer cet élément créerait donc une erreur. Avant d'évaluer le premier élément d'une SCons afin de récupérer le foncteur, il faut donc vérifier qu'il ne s'agit pas déjà d'une expression lambda qui dans ce cas impliquera que le premier élément de la liste est déjà le foncteur.

3.3 Précision sur le Contexte

Le contexte est l'élément qui permet de faire la liaison entre les **Symboles** et les **SExpr**. Un symbole LISP peut être différents objets : (les derniers cas ne sont finalement que des cas de listes particulières)

— une *primitive*,
— un *symbole*,
— une *liste*,
— une *lambda*,
— une *flambda*.

Toutes les classes que nous avons définies afin de construire des **SExpr** proposent des objets immuables. C'est à dire qu'ils sont donc impossibles à modifier. Quoi que l'on ajoute dans le contexte, nous savons donc que sa valeur ne sera pas modifiée. Cela permet d'apporter une certaine robustesse à notre application. De plus, cette caractéristique est très intéressante et utile pour la recherche dans le contexte par exemple. Il n'est plus nécessaire de se demander comment manipuler les données du contexte de façon sûre (pas de copie à faire si on a accès à un élément du contexte par exemple) car il est impossible de modifier et donc altérer les données qui y sont présentes.

3.4 Cas particulier des SCons

Les **SCons** nécessitent un traitement spécifique qui diffère des autres cas, c'est pourquoi nous avons décidé de nous attarder dessus.

En effet, lors de l'évaluation d'une **SCons**, plusieurs traitements sont effectués selon les différents cas. Dans un premier temps, on évalue le premier élément de la liste. Ce premier élément est essentiel pour la suite : c'est le foncteur. Selon le foncteur que l'on obtient, notre traitement sera différent :

— Si c'est une primitive, il suffit d'exécuter celle-ci.
— Si il s'agit d'une liste, on va vérifier si il s'agit d'une liste contenant une fonction **Lambda** ou **FLambda**. Si c'est le cas, elle sera exécuté.
— Sinon, il ne s'agit pas d'une fonction et donc il s'agit d'une erreur.

L'interface **Foncteur**, héritée à la fois par la classe **Primitives** et la classe **Fonction**, possède une fonction *exec* qui prend en paramètre une **machineLISP** (qui ne contient que le contexte pour le moment comme nous l'avons expliqué plus tôt) et une **SExpr** qui représente la liste des arguments de la fonction.

C'est dans la gestion de la primitive ou de la fonction que l'on va décider d'évaluer ou non le contenu de cette **SExpr**.

Suivant si la primitive ou fonction est une **Subr** ou une **FSubr** ou une **Lambda** ou une **FLambda**, l'évaluation des paramètres est effectué ou non. Dans le cas ou elle est nécessaire, chaque élément de la liste de paramètre est alors évalué individuellement.

3.5 Quote

Au début de ce projet, lorsque nous arrivions à effectuer quelques primitives simples, nous avions défini la primitive **Quote**. Cependant, lorsque nous avons commencé à évaluer des **Lambda** et des **FLambda**, nous nous sommes rendu compte que cette primitive n'était pas nécessaire. En effet, il est très facile de définir la fonction **Quote** avec une **FLambda**.

Nous avons donc enlevé cette primitive de notre ensemble. Mais notre parser nécessite cette fonction. En effet, il transforme toutes les **SExpr** de la forme 'A en (QUOTE A). Il fallait donc, quoi qu'il arrive, que la fonction QUOTE soit définie. Pour ce faire, avant l'appel à la primitive **toplevel**, nous avons donc effectué l'évaluation d'une chaîne de caractère définissant cette fonction.

3.6 Reader

Notre objet *Reader* permet de lire une **SExpr** à partir de trois entrées différentes. Ces trois méthodes ont chacune un lecteur prenant leurs entrées sur des flux de différentes natures. Nous ne faisons aucune évaluation au sein de notre **Parser**, elles sont toutes faites ailleurs car son seul rôle et de lire des **SExpr**.

Nous avons géré le flux de l'*entrée standard* : le clavier. Chaque expression lue doit finir par un retour à la ligne. Nous avons aussi une fonction qui lit une *String* par le même procédé. Enfin nous avons aussi permit à notre application de lire des **SExpr** par le biais d'un *fichier*. Néanmoins ce n'est pas le *Reader* qui ouvre et ferme le fichier. Pour cette dernière méthode nous lui donnons un *Reader* en paramètre.

Notre grammaire est de plus classique. Il s'agit d'une grammaire synthétisée, c'est dire que pour chaque composante elle retourne un résultat qui est traité plus haut dans la structure de la grammaire. Nous avons défini des **Tokens** avec une simplicité évidente car nous pouvions utiliser des expressions régulières comprenant l'opérateur "+" et "*". Pour la définition de la grammaire, il est parfois facile d'utiliser aussi des expressions régulières mais JavaCC ne veut pas, nous avons donc du expliciter ces opérateurs, ce qui alourdie notre grammaire.

Nous avons la possibilité de faire des affichages pour débuger notre grammaire. Ceci nous à particulière était utile au début pour vérifier que les appels se faisaient bien dans les bonnes catégories du **Parser**.

La gestion des erreurs est faite dans les méthodes du **Parser**. Lorsqu'une erreur au niveau de la grammaire est levée, nous la capturons pour en renvoyer une autre d'un type défini par nos soins. Pour le cas d'*importe*, nous avons du inclure la gestion de la fin de fichier qui pouvait causer des problèmes.

4 Problèmes rencontrés

4.1 SExpr incomplète

Lors de la lecture d'un fichier par notre interpréteur, la lecture d'une **SExpr** incomplète peut poser problème. Le parseur consume le fichier jusqu'à ce qu'il ait pu lire une **SExpr** complète ou atteindre la fin du fichier. Dans le cas ou une **SExpr** est mal formé et qu'il y a un problème de parenthésage, c'est à dire qu'il y a un eu plus de parenthèse ouvrante que fermante, alors le parseur va potentiellement consumer tous le reste du fichier.

De plus, nous n'avons pas trouvé de moyen de remonter l'information comme quoi une **SExpr** était en cours de lecture. Lorsque le caractère 'EOF' est trouvé, le parseur remonte alors une Exception et la **SExpr** qui avait été crée avec ce qui était lu est perdue. Nous ne détectons donc pas cette erreur.

4.2 Terminaison

Avec l'implémentation de la lecture de fichier plutôt que System.in, nous nous sommes confrontés à un problème que nous n'avons pas su expliquer.

Malgré le fait que le fil d'exécution montre bien que le programme sorte de la fonction main, et se dirige vers exit, afin de terminer le programme, le programme ne se termine pas lorsque nous lisons un fichier.

Pour remédier à ce problème, nous avons donc rajouté l'instruction System.exit(0) ; comme dernière instruction de la fonction main().