RAPPORT DE PROJET

THE LAMBDA INTERPRETER

I. INTRODUCTION

En 1936, Church inventa un système formel appelé le Lambda Calcul. Church définit alors la notion de fonction et d'application qui sont des fondamentaux des langages fonctionnels modernes.
En 1936-37, Turing définit la notion d'exécution de ces fonctions sur des machines qu'on appelle aujourd'hui machines de Turing. Le Lambda Calcul est donc considéré comme étant Turing complet.

De nombreux langages de programmation récents dit « fonctionnels purs ou hybrides » ont pour racine le Lambda Calcul (Lisp, Rust, Haskell).
Le lambda calcul peut être considéré comme un langage de programmation théorique. En tant que tel, il est impressionnant de facilité, et représente le langage de programmation le plus simple possible.
En effet, sa grammaire simpliste est la suivante :

<expression> ::= <variable>
| <constante>
| <expression> <expression>
| λ <variable>.<expression>

La grammaire du projet est un peu différente et permet de faire des conditionnelles, de la récursivité, et du calcul de nombres entiers.
Ces raccourcis permettent de ne pas dépendre de certains notions comme le « Y combinator » qui est le « Fixed Point Combinator » découvert par Haskell Curry[1] et qui permet d'obtenir des fonctions récursives en lambda calcul. Nous n'avons pas non plus besoin des « Church Numerals » ou de la représentation du True/False en Lambda Calcul pur.
Voici la grammaire choisie pour le projet :

M ::=	c	(constante)
|	x	(variable)
|	M (+ - *) M	(calcul)
|	if(M = > < ≤ ≥ != M)then{M}else{M}	(conditionnelle)
|	(λx.M)	(abstraction)
|	M M	(application)
|	(rec f x.M)	(récursivité)

Le but du projet est de fournir une implémentation d'un interpréteur de cette nouvelle grammaire en Java. L'interpréteur peut être utilisé dans un mini espace de travail via une interface graphique qui permet d'évaluer des termes du Lambda Calcul selon les étapes de réduction classiques (α et β-Réduction et réduction des règles de grammaire ajoutées).

[1] Curry a aussi démontré le « Currying », qui peut être exprimé en Lambda Calcul par des expressions comme (λx.λy.λz.x) qui est une abstraction qui renvoie une abstraction, puis encore une.

II. Outils utilisés, choix d'implémentation et difficultés rencontrées

1. Lexer/Parser

La première étape dans la création d'un interpréteur est de fournir un Lexer / Parser capable de reconnaître le langage et de fournir un Abre Syntaxique Abstrait (Abstract Syntax Tree en anglais) du terme en entrée.
L'AST est une structure récursive bien connue, qui facilite les traitements lors des étapes de réduction.
Le lambda calcul étant « left associative », l'expression K L M doit être interprétée comme : ((K L) M) et l'AST est donc parcouru selon la stratégie « left-most, outermost ».

Pour l'étape de création du Lexer/Parser, nous avons décidé (après autorisation), d'utiliser la bibliothèque externe ANTLR4. ANTLR4 est une bibliothèque qui à partir d'une grammaire qui décrit le langage généré automatiquement le Lexer et le Parser.
Le Parser est capable de générer automatiquement un AST et ANTLR4 généré aussi automatiquement des interfaces de parcours de cet arbre qu'il faudra implémenter.
L'objectif premier est donc de créer une grammaire qu'ANTLR4 peut comprendre.

Voici la grammaire finale du projet :

```
grammar Lambda;


expression :   LPAR expression RPAR          #parenExpression
             | VAR                           #variable
             | INT                           #integer
             | expression MULT expression    #mult
             | expression op=(PLUS|MINUS) expression      #add
             | IF LPAR expression op=(EQ|NEQ|GT|LT|GTEQ|LTEQ) expression RPAR THEN
LCURL expression RCURL ELSE LCURL expression RCURL     #ifRule
             | expression ' ' expression     #application
             | LAMBDA VAR DOT expression     #abstraction
             | REC ' ' VAR ' ' VAR DOT expression   #recRule
             ;

RPAR      :  ')';
RCURL     :  '}';

MINUS     :  '-';
DOT       :  '.';

NEQ       :  '!=';
LT        :  '<';
LTEQ      :  '<=';

IF        :  'if';
THEN      :  'then';
ELSE      :  'else';
REC       :  'rec';

VAR       :  [a-zA-Z][a-zA-Z0-9]*;
INT       :  [0-9]+;

LAMBDA    :  'λ' | '\\' ;
WS        :  [ \t\n]+ -> skip ;

LPAR      :  '(';
LCURL     :  '{';
PLUS      :  '+';
MULT      :  '*';

EQ        :  '=';
GT        :  '>';
GTEQ      :  '>=';
```

Les entrées en majuscules sont des règles pour le Lexer, le reste sont des règles pour le Parser.
Il y a quelques différences avec la grammaire du sujet, la multiplication et la soustraction sont possibles. Tous les opérateurs de comparaison binaires sont inclus.
Les parenthèses après le then et le else sont remplacées par des accolades.
L'ordre des entrées correspond à la précédence des termes, par exemple la multiplication s'applique avant les additions (comme en mathématiques).
La grammaire ne supporte pas les entiers signés pour la simple raison que la grammaire est plus complexe si on les supporte (et que ce n'est pas le but premier du projet de faire du calcul sur des entiers signés). La raison de cette difficulté vient du fait que l'expression
5-5 est reconnu par le Parser comme une addition. Pour autant si l'on ajoute '-' ? à la règle du Lexer INT, l'expression sera parsée comme étant un 5 et un -5 collé ce qui ne signifie pas grand chose. Le problème est facilement réglé et pourrait faire l'objet d'un futur ajout au projet.

La première difficulté rencontrée sur la réalisation de la grammaire vient du fait que les grammaires ne supportent pas la « left recursion » sauf si elles sont dans la même règle.
Les premières versions de la grammaire partaient du principe que l'on faisait une règle pour chaque règle de la grammaire du projet.

On pouvait donc lire :

expression : VAR | INT | abstraction | application ;
abstraction : LAMBDA VAR DOT expression ;
application : expression expression

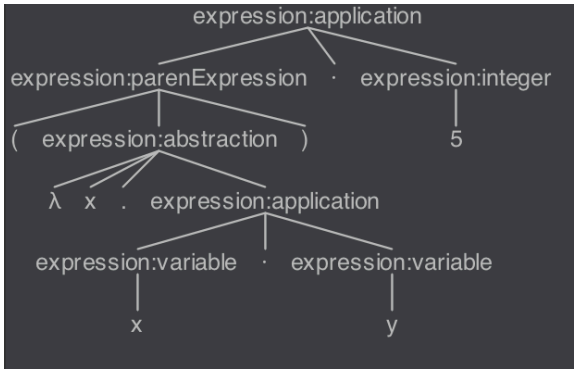
Le problème vient de la règle application. En écrivant cette règle il est possible de faire expression → application → expression → application → expression … en un appel récursif infini.

Pour autant il est possible de faire un appel récursif direct donc :
expression : expression expression ; Est un appel possible qui permet de résoudre simplement le problème. (Et il y a d'autres possibilités pour résoudre le problème de la récursivité des règles d'une grammaire)

Mais en écrivant cela, comment dire dans l'AST que le parent est une application de deux termes ? Il faudra donc fouiller sur internet (la documentation est assez pauvre en dehors d'un livre écrit par l'auteur de la bibliothèque) pour découvrir l'existence des labels # après chaque possibilité d'une règle. Il est bon de préciser que si l'on applique un label pour une alternative il faut mettre un label sur toutes les alternatives.

Une fois ce problème réglé et la bonne précédence des termes respectée, la grammaire du Lambda Calcul restant très simple, aucune autre difficulté n'a été rencontrée.

Voici l'AST généré par ANTLR pour une application d'un entier à une abstraction simple :



2. Réduction d'un terme : l'interpréteur

Nous avons maintenant à notre disposition un AST pour chaque terme syntaxiquement correct. Le plus dur reste pour autant à faire, il faut réduire les termes !

Pour ce faire, il est nécessaire de parcourir l'arbre et d'agir en fonction du nœud où l'on se trouve. Le pattern de « Visitor » créé par ANTLR correspond parfaitement à nos besoins et nous fournit une coquille vide, une interface, qui contient des méthodes pour chaque type de nœud. A nous d'implémenter cette interface en fonction de nos besoins.

Premier problème, les méthodes de l'interface renvoient toutes le même type abstrait.
Pour autant rien que dans notre cas, une variable n'est pas du même type qu'un entier.
Il est donc nécessaire de créer une classe « type » qui sera un « wrapper » de tous nos autres types. Nous avons ainsi créé la classe « wrapper » Value, puis des classes pour chaque type de nœud, afin de renvoyer des informations qui ont du sens pour l'évaluation à chaque nœud. En effet, on ne peut pas traiter une application entre une abstraction simple et une variable comme on traitre une application entre une fonction récursive et la valeur qu'on lui applique.

Pour les cas les plus simples, lorsque l'on visite une variable ou un entier on renvoie cette variable ou cet entier, lorsque l'on visite une addition on fait le calcul si le membre de droite et de gauche sont des entiers et l'on renvoie un Calcul de (droite opérateur gauche) dans le cas contraire. On peut alors comprendre l'utilité de la classe Calcul dans le cas d'abstraction simple comme (λx.λy.x*x+y*y) 5 2 où un simple String ne donne pas assez d'informations sur ce que l'on doit faire avec le corps de notre fonction à deux paramètres.

a. β-réduction

Dans le cas de la β-réduction, c'est donc la méthode qui agit sur un nœud d'une application qui devra réduire notre terme.

La première difficulté (et une des plus grande du projet) pour la β-réduction, porte sur le moyen de remplacement des variables mais aussi sur la condition d'application.
Pour illustrer ces problèmes deux abstractions :

- (λx.xy) 5 : Ici le corps de la fonction ne contient pas la variable x donc l'abstraction renvoie xy. Pour autant si l'on vérifie avec la méthode contains de String on aura bien une réponse positive.
- (λx.x xy) 5 : Cette fois on peut appliquer 5 à la fonction. Mais si l'on utilise bêtement la méthode replace on obtiendra 5 5y, ce qui n'a pas de sens.

La solution la plus simple, et celle retenue pour notre projet vient de la « puissance » des expressions régulières. \bvariable\b match des mots entiers et donc dans x xy ou x+xy ou x) L'expression régulière \bx\b nous trouvera bien x quand il seul ou entouré de caractère qui ne sont pas des lettres ('+' et ')').

Une autre solution qui tire partie des forces de l'AST est d'appliquer les changements en fonction des nœuds parents lorsque l'on visite le nœud de la variable. Cette solution est plus lourde à mettre en place. Pour autant le fait de regarder le ou les nœud(s) parent(s) fait l'objet d'une méthode qui permet de déterminer si le nœud courant est en dessous d'une abstraction (ce qui est une information importante dans certains cas).

La deuxième difficulté découle directement de l'utilisation de la méthode replace pour réduire les applications. En effet replace agit sur des String et renvoie un String et si le corps de l'abstraction est très complexe, on peut perdre beaucoup d'informations durant la transformation. Pour régler ce problème, il est possible si l'on voit un String de dire que l'on renvoie tel quel un terme réduit pour qu'il soit parsé à nouveau[2] ce qui permettra de regagner l'information sur le terme. Pour que cette façon d'agir soit possible, il est très important d'appliquer les bonnes parenthèses aux termes, et c'est pourquoi les méthodes toString de chaque sous classe de type sont réalisées avec le plus grand soin.

b. α-Renommage

L'α-Renommage a été réalisé après que la β-réduction soit fonctionnelle.
La stratégie adoptée pour l'α-Renommage est de visiter l'arbre syntaxique une première fois afin d'obtenir une liste des variables libres. La méthode d'obtention est relativement simple si l'on visite une variable qui n'est pas dans une abstraction, alors elle est libre, sinon lorsque l'on arrive dans une abstraction, on dit que la variable sous le lambda est liée. Puis toutes les variables dans l'expression de l'abstraction qui ne sont pas celles qui est sous le lambda sont libres.

L'avantage d'obtenir cette liste de variable libre avant les étapes de réduction, est de pouvoir faire le renommage peut importe l'endroit où sont situé les abstractions (par rapport aux variables libres) dans le terme. L'étape de renommage à proprement parlé est assurée au cas par cas. Pour chaque abstraction, si elle doit être renommée, on crée une règle de renommage temporaire via une HashMap : Ancienne variable → Nouvelle variable.

c. Réduction de la conditionnelle et des fonctions récursives

La réduction du If/Then/Else et de la fonction récursive ne sont pas difficiles car l'implémentation actuelle est plutôt robuste et supporte bien les ajouts.

La réduction du If se résume à une action au cas par cas, en fonction de l'opérateur de comparaison. La comparaison n'étant faite que sur des entiers, il est alors facile de renvoyer soit le then soit le else, ou de renvoyer l'expression entière si la comparaison n'est pas faite sur des entiers.

Pour la récursivité, une fois les règles de remplacement comprises, il suffit de bien les appliquer pour obtenir des résultats rapidement.

[2] Une autre limitation d'ANTLR4 venant du fait que les AST produits ne sont pas modifiables, c'est une bonne pratique de passer un terme autant de fois que nécessaires à la moulinette.

3. Environnement de travail graphique

L'interface graphique a été crée en Swing, simplement pour comparer avec JavaFX déjà utilisé dans un autre projet. (Et pour être honnête Swing c'est pas si mal).

Le but est de fournir un environnement simple mais pratique pour tester nos termes du lambda calcul. L'archive du projet contient un JAR qui permet de lancer la GUI de la façon la plus simple possible.

Les fonctionnalités du programme sont les suivantes :

- A gauche, une liste des termes sauvegardés. Il est possible de double cliquer sur un terme pour le mettre directement dans la zone d'entrée de texte. Les fonctions d'import / export fonctionne de pair avec cette zone de sauvegarde.
  Pour chaque terme sauvegardé, il est possible de visionner son AST, via le menu.

- Les fonctions d'import/export fonctionnent via des fichiers textes qui contiennent une commande par ligne tout simplement

- Au centre, c'est l'espace de travail, qui est l'endroit où l'on affiche les informations pour l'utilisateur. Les informations sont soit des messages d'erreurs si l'utilisateur entre des termes mal formés, soit le résultat de la réduction du terme entré par l'utilisateur. C'est une simple TextArea non modifiable. Au lancement du programme, la zone affiche un mini tutoriel qui peut être revu plus tard via le menu d'aide.

- Juste en dessous de l'espace de travail, il y a la zone d'entrée des termes. Cette zone d'entrée supporte un historique des termes entrés, dans lequel il est possible de naviguer avec les touches fléchées haut et bas. La commande clear permet aussi de vider la zone de travail. Il est possible d'obtenir le symbole λ par une simple pression sur la touche ';', ce qui reste plus pratique que d'écrire un \ ou d'écrire un λ via son Unicode.

- A droite de la zone d'entrée, une simple « checkbox » permet de choisir entre la sauvegarde des termes ou leur non sauvegarde.

III. CONCLUSION

Si l'on regarde en arrière, le projet ne représente pas beaucoup de code, et son fonctionnement n'a rien de très compliqué. Pour autant, le projet est bel et bien difficile.

L'effort de recherche à fournir pour bien comprendre le Lambda Calcul est assez conséquent. De plus, une chose en entrainant une autre, la moindre petite recherche nous fait découvrir de nombreuses informations intéressantes sur le monde du fonctionnel et sur la complexité non apparente du Lambda Calcul.
La découverte d'un système aussi simple mais pour autant très « puissant », en 1936, est plutôt impressionnante[3].
De nombreux langages de programmation fonctionnels purs existent aujourd'hui, et ils apportent une vision différente de la programmation, (un paradigme) qui arrive même à impacter les langages impératifs classiques (incorporation des lambdas dans Java 8 !).
En effet, pourquoi se priver de ce que le fonctionnel fait mieux que l'impératif ?

Pour en revenir au projet, quelques améliorations sont possibles. La première étant la plus évidente, un système de type simple est l'ajout prioritaire.
L'avantage d'un système de type est indéniable, il permet d'imposer des résultats. Si une abstraction est du type A→B alors elle ne peut pas renvoyer un type A et elle ne peut pas prendre un type B en argument. Le typage est fortement « réducteur » de par le fait qu'il limite les possibilités et permet normalement à une réduction de toujours finir.
Il est bon de noter que certains éléments ne peuvent pas être typés. Le Y Combinator par exemple aurait un type infini. Pour typer ce Combinator, il faut un système de type plus compliqué, appelé le système F ou système de type polymorphique (dont le langage Haskell est inspiré !).
Il est aussi possible d'ajouter de petites fonctionnalités à l'interface graphique, comme plus d'options pour le comportement de l'interpréteur (choix du nombre d'itérations, choix de l'ordre d'évaluation « eager ou lazy »). Pour autant le projet réalisé reste assez complet et fonctionnel dans son état final.

Pour conclure, si le projet était à refaire, il faudrait penser à changer de langage de programmation ☺. En passant sur un langage de programmation fonctionnel pur (Haskell par exemple), non seulement le système de type est plus simple à mettre en place mais le fonctionnement global du programme est aussi plus simple, plus court, etc. Reste seulement à passer la barrière, du changement de paradigme, qui n'est pas évidente quand notre éducation repose principalement sur l'apprentissage des langages impératifs. Et c'est bien pour cela que ce projet est important, car c'est une porte qui s'ouvre sur la découverte du monde fonctionnel !

[3] SI impressionnant, que l'équipe de Church décida de considérer cette découverte comme mineure à l'époque !