# Introduction

Ce présent projet représente notre travaille sur la réalisation d’un compilateur pour le langage R. Et nous avons travaillé avec le langage C et sans l’utilisation des générateurs de l’analyseur lexicale comme FLEX ou les générateurs de l’analyseur syntaxique comme BISON.

Les trois éléments essentiels d’un compilateur sont l’analyseur lexicale, l’analyseur syntaxique et celui de sémantique. Nous nous sommes concentrés premièrement dans l’élaboration de la grammaire du langage pour faciliter la phase de codage. Ensuite nous avons diviser les tâches entre nous. La fin de chaque éléments du compilateur a été testé indépendamment des autres pour s’assurer de son bon fonctionnement premièrement. Après les avoir liés on les a testé sur différents instructions que notre compilateur doit reconnaître sans erreur et qui sont contenus dans le fichier « testing.txt » qui est inclus dans l’archive envoyé.

Le plan de ce rapport sera le suivant :

**I. Grammaire**

**II. Documentation :**

1. **Analyseur Lexicale**
2. **Analyseur Syntaxique**
3. **Analyseur Sémantique**

**III. Fonctionnement du compilateur**

**IV. Limites du compilateur**

**V. Conclusion**

I. Grammaire

Nous avons écrit notre grammaire sans la rendre LL(1) pour qu’elle soit lisible, mais le codage de cette grammaire respecte tous les règles d’une grammaire LL(1).

Prog : [expr | Loop | Function | Decision]\*

expr: id <- ( id | CallFunction | data ) [ operator (id | CallFunction | data) ]\*

| id <- function

| ( id | CallFunction | data ) [ operator (id | CallFunction | data) ]\* -> id

operation: ( id | CallFunction | data ) operator (id | CallFunction | data )

Operator : == | < | > | + | - | \* | /

AssignOp : <- | = | ->

data : INT | FLOAT | STRING | VECTOR

Decision: if ( condition ) { expr }

| if ( condition ) { expr } else { expr }

Loop: while ( condition ) { expr }

| repeat { expr }

| for ( ID in vector ) { expr }

vector : c( id [, id]\* )

Function: function ( None | id [, id ]\* ) { expr | return }

CallFunc id ( None | data [,data]\* )

II. Documentation

# Analyseur Lexical

**char NextChar()**

Description : Lit le prochain charactère et le retourne.

Arguments : NULL

**void ERRORSyntax(char\* ErrorText)**

Description : Affiche l’erreur syntaxique rencontré et termine l’exécution du programme.

Arguments :

**ErrorText :** le texte à afficher.

**void NextToken()**

Description : Après avoir ignorer les séparateurs, elle teste si le prochain Token est un Nombre, Mot réservé (pour les boucles, les décisions, les fonctions ) ou un charactère spécial. Sinon elle le traite comme un invalide Token, et arrête le programme.

Arguments : NULL

**void ignoreSeparators()**

Description : Ignore les commentaires ainsi que les séparateurs comme la tabulation, les espaces et les nouvelles lignes.

Arguments : NULL

**void isNumber()**

Description : Construit le nombre et vérifie si c’est un entier ou un réel. Sinon elle le traite comme un invalide Nombre, et arrête le programme.

Arguments : NULL

**void isWord()**

Description : Construit le mot, et vérifie si c’est un mot réservé pour la fonction, les décisions, les boucles ou pour le vecteur. Sinon elle test si c’est un appel de fonction où le considère comme un identificateur.

Arguments : NULL

**int Operator()**

Description : Vérifie si c’est opérateur tel la division, la multiplication, la soustraction, l’addition.

Arguments : NULL

**int isSpecial()**

Description : Vérifie si c’est un charactère spécial, et en cas ou elle trouve les guillemets, elle vérifie si c’est un texte.

Arguments : NULL

# Analyseur Syntaxique

**int TestToken(TOKEN\_CODE TokenToTest)**

Description : Teste si le Token de l’argument convient avec le Token actuel trouvé par l’analyseur lexicale. En cas de succès il fait appel au prochain Token.

Arguments :

**TokenToTest** : représente le Token qu’on va comparer avec le Token de l’analyseur lexicale.

**void EndExpr()**

Description : Assure qu’il y a un saut de ligne ou un point-virgule à la fin d’une expression. Autre symbole n’est pas permis, et donnera une erreur syntaxique.

Arguments : NULL

**void NoNewline()**

Description : Assure qu’il n’y aura pas un saut de ligne. En cas de trouvaille d’un saut de ligne, il y aura une erreur syntaxique.

Arguments : NULL

**int isExpr()**

Description : Vérifie si la syntaxe représentée dans la grammaire de l’expression est valide. Voir la grammaire de : « expr »;

Arguments : NULL

**int isDecision()**

Description : Teste si l’expression actuel représente une instruction conditionnelle sous la forme de : if (condition) {expressions} ou bien if (condition) {expressions} else {expressions}

Arguments : NULL

**int isFunction()**

Description : Teste si l’expression actuel représente une déclaration de fonction sous la forme : function ( arguments ) { expressions }

Arguments : NULL

**int isReturn()**

Description : Teste si l’expression actuel est une instruction de retour pour les fonctions.

Arguments : NULL

**int isLoop()**

Description : Puisqu’il y a trois instructions de boucle dans R qui sont : for, while, repeat. Alors cette fonction teste laquelle convient avec l’expression actuel.

Arguments : NULL

**int isVector()**

Description : Teste si l’expression actuel représente un vecteur.

Arguments : NULL

**int isID()**

Description : Teste si le Token donné par l’analyseur lexicale est un identificateur.

Arguments : NULL

**int isAssignOp(int assignop)**

Description : Teste si le Token donné par l’analyseur lexicale représente le type d’affectation donné comme argument à cette fonction.

Arguments : NULL

**int isCondition()**

Description : Teste si l’expression actuel représente la syntaxe d’une instruction conditionnelle qui est sous cette forme : un id ou appel fonction ou donnée plus un operateur de comparaison plus un id ou appel fonction ou donnée.

Arguments : NULL

**int isID\_CallFunc\_Data\_Operation()**

Description : Teste si l’expression actuel représente la syntaxe d’une instruction conditionnelle qui est sous cette forme : un id ou appel fonction ou donnée plus un operateur de comparaison plus un id ou appel fonction ou donnée.

Arguments : NULL

**int isOperator(int type)**

Description : Vérifie si le Token est un opérateur normal comme la multiplication, addition, division, soustraction ou un opérateur de comparaison comme inférieur, inférieur ou égal ….

Arguments :

**Type :** type d’opérateur qu’on veut tester, normal ou de condition.

**int isCallFunction()**

Description : Teste si l’expression d’appel fonction avec ses arguments est valide.

Arguments : NULL

**int isData()**

Description : Vérifie si le Token actuel représente une donnée de type : entier, réel, chaine de charactères ou un vecteur.

Arguments : NULL

# Analyseur Sémantique

**void SemanticID(Symbol Input, Symbol Variable)**

Description : S’assure que l’identificateur reçu dans l’argument d’input est déjà déclaré afin qu’elle transfère son type ainsi que sa valeur à l’identificateur reçu dans l’argument de Variable.

Arguments :

**Input :** L’identificateur qui va être affecté à la variable.

**Variable :** L’identificateur qu’on va lui affecter la valeur ainsi que le type d’Input.

**void SemanticData(Symbol Input, Symbol Variable)**

Description : Attribue la valeur de la donnée reçue dans l’argument Input ainsi que son type à la variable reçu dans l’argument Variable.

Arguments :

**Input :** La donnée qu’elle va être affecté à la variable.

**Variable :** L’identificateur qu’on va lui affecter la valeur ainsi que le type d’Input.

**int isIDDeclared(Symbol token)**

Description : S’assure que l’identificateur reçu dans l’argument Token est déjà déclaré. Dans ce cas elle retourne sa position dans la table de symbole, sinon elle retourne la valeur UNDECLARED.

Arguments :

**Token :** L’identificateur qu’on veut s’assurer s’il est déclaré ou non.

void **printSymbolTable**()

Description : Affiche toute la table de symbole.

Arguments : NULL

III.Fonctionnement du compilateur

VI.Limites du compilateur

V.Conclusion

Ce projet nous a permis de parfaitement comprendre la chaîne de compilation d’un code. Et nous a donné l’occasion de découvrir énormément de problèmes liés notamment à l’analyse, qui n’étaient pas aussi évidents dans la théorie du cours.

Nous estimons avoir passé environ 3 semaines complète pour l’achèvement de ce compilateur. Les deux premiers jours on s’est focalisé sur le regroupement de la grammaire, ensuite les tâches se sont divisées en trois, chacun à son analyseur.

Nous ne pensions pas passer autant de temps sur la conception de la grammaire, qui nous a valu de recommencer plusieurs fois et de passer de nombreuses heures à chercher la bonne méthode pour lever les ambiguïtés, sans en soulever d'autres.

La difficulté en général réside dans l’implémentation de la grammaire lors la réalisation de l’analyseur syntaxique, ensuite la sémantique pourra être facilement vérifié en se basant sur la table de symbole généré par l’analyseur syntaxique.