Ce travail a été fait en binômes par :
Nom & Prénom : Mahmoudi Rima
Matricule : 161631076532
MIV M1

Nom & Prénom : Salhi Racha
Matricule : 161631102659

Rapport du Projet de Compilation du Langage R

Analyse Lexicale Avec Flex :

Nous commençons par définir les différents types possibles indiqués dans l'énoncé à l'aide des expressions régulières :

$TYPE$	$REGEX$
$INTEGER$	$\{CHFR\} + \{CFR\} * \|0$
$NUMERIC$	$\{CFR\} + "."\{CFR\} +$
$CHARACTER$	$\backslash'.\backslash'$
$LOGICAL$	$TRUE\|FALSE$
IDF	$[A-Z][a-z0-9] *$

Avec :

CFR	$[0-9]$
$CHFR$	$[1-9]$

Nous avons associé à chaque mot clé du langage son identificateur afin de le reconnaitre comme le montre l'exemple suivant :

$INTEGER \mid NUMERIC \mid$	$mc_{integer} \mid mc_{numeric} \mid$
$CHARACTER \mid LOGICAL$	$mc_{character} \mid mc_{logical}$
$WHILE \ /FOR \ / IN$	$mc_{while} \mid mc_{for} \ / mc_{in}$
$IF \mid "ELSE\ IF" \mid ELSE \mid IFELSE$	$mc_{if} \mid mc_{elseif} \mid mc_{else} \mid mc_{ifelse}$
$AND \mid OR$	$and \mid or$
$= \mid > \mid < \mid \geq \mid \leq \mid ! =$	mc_{comp}
$+ \mid - \mid * \mid / \mid \%$	$mc_{plus} \mid mc_{moins} \mid mc_{mul} \mid mc_{div} \mid mc_{rest}$
"(" | ")" | "[" | "]" | "{" | "}"	$parO \mid parF \mid croO \mid croF \mid accO \mid accF$
" "	$space$
"←"	aff
":" | ","	$pt \mid vrg$

Les identificateurs associés aux types définis :

IDF	id
INTEGER	$id_{integer}$
NUMERIC	$id_{numeric}$
CAR	$id_{character}$
LOGICAL	$id_{logical}$

Pour autoriser les commentaires, nous avons ajouté l'expression régulière : "#".*

Chaque ligne du commentaire doit être précédée par un « # ».

Analyse Syntaxico-sémantique Avec Bison :

Analyse syntaxique :

Dans le but de définir le langage R, nous avons conçu des grammaires conformes aux structures des instructions mentionnées dans l'énoncé.

La grammaire globale qui fait appel aux déclarations des variables ainsi que les différentes instructions est :

$$Start \rightarrow Var\ Inst$$

La grammaire faisant appel à la déclaration d'une ou multiples variables avec ou sans affectation est la suivante :

$$Var \rightarrow Type\ Vars\ Var \mid \mathcal{E}$$

Avec :

- $Type \rightarrow mc_{integer} \mid mc_{numeric} \mid mc_{character} \mid mc_{logical}$
- $Vars \rightarrow id\ aff\ Val \mid id\ croO\ id_{integer} croF \mid id\ Liste$
- $Val \rightarrow Entier \mid Reel \mid id_{character} \mid id_{logical}$
- $Entier \rightarrow\ id_{integer} \mid\ parO\ mc_{moins}\ id_{integer}\ parF$
- $Reel \rightarrow\ id_{numeric} \mid\ parO\ mc_{moins}\ id_{numeric}\ parF$
- $Liste \rightarrow vrg\ id\ Liste \mid \mathcal{E}$

La grammaire constituant les différentes formes d'instructions est :

$$Inst \rightarrow Inst_{Affectation}\ Inst$$
$$|Inst_{Incrémentation} Inst$$
$$|Inst_{if} Inst$$
$$|Inst_{ifelse} Inst$$
$$|Inst_{while} Inst$$
$$|Inst_{for}\ Inst$$
$$|\mathcal{E}$$

- **<u>Affectation</u>** :

$$Inst_{affectation} \rightarrow Var_{aff}\ aff\ Val\ Exp$$
$$|\ Var_{aff} aff\ id\ Exp$$
$$|\ Var_{aff}\ aff\ id\ croO\ id_{integer}\ croF\ Exp$$

Avec Var_{aff} représentant la variable définie recevant l'affectation que ce soit une simple variable ou bien un élément du tableau :

$$Var_{aff} \rightarrow id\ |\ id\ croO\ id_{integer}\ croF$$

Les diverses expressions arithmétiques sont elles aussi formulées dans la grammaire suivante :

$$Exp \rightarrow Opr\ Entier\ Exp$$
$$|Opr\ Reel\ Exp$$
$$|Opr\ id\ Exp$$
$$|\ Opr\ id\ croO\ id_{integer}\ croF\ Exp$$
$$|\ \epsilon$$

Avec :

$$Opr \rightarrow mc_{plus}\ |\ mc_{moins}|\ mc_{mul}|\ mc_{div}|\ mc_{rest}$$

- **Incrémentation** :

$$Inst_{Incrémentation} \rightarrow$$
$$id\ \ mc_{plus}\ aff\ id_{integer}$$
$$|mc_{moins}\ aff\ id_{integer}$$
$$|id\ croO\ id_{integer}\ croF\ \ mc_{plus}\ aff\ id_{integer}$$
$$|\ id\ croO\ id_{integer}\ croF\ mc_{moins}\ aff\ id_{integer}$$

- **Conditions** :

  - **IF…** & **IF…ELSE…** & **IF…ELSE IF…ELSE:**

$$Inst_{If}: \rightarrow\ mc_{if}\ parO\ Condit\ parF\ accO\ Inst\ accF\ Suite$$

Avec :

$$Suite \rightarrow\ mc_{Elseif}\ parO\ Condit\ parF\ accO\ Inst$$
$$|mc_{else}\ accO\ Inst\ accF$$
$$|\mathcal{E}$$

  - **IFELSE :**

$$Inst_{ifelse} \rightarrow$$
$$VAR_{aff}\ aff\ mc_{ifelse}\ parO\ Condit\ vrg\ Membre\ vrg\ Membre\ parF$$

Avec :

$$Condit \rightarrow Membre\ Exp\ mc_{comp}\ Membre$$

$$Membre \rightarrow$$
$$id|\ id\ croO\ id_{integer}\ croF|\ Entier|\ Reel|\ id_{character}|\ id_{logical}$$

- **Boucle** :

  - **WHILE :**

$$Inst_{while} \rightarrow mc_{while}\ parO\ Condit\ parF\ accO\ Inst\ accF$$

  - **FOR :**

$$Inst_{for} \rightarrow\ mc_{for}\ parO\ id\ space\ in\ space\ id_{integer}\ pt\ id_{integer}$$

Analyse sémantique :

Des routines d'analyse sémantique ont été introduites dans toutes les grammaires discutées précédemment afin d'assurer le bon fonctionnement du compilateur :

- **Recherche d'une entité :** indique si une entité existe dans la table des symboles.

- **Insertion :** sert à créer un nouvel élément dans la table des symboles afin d'inclure la variable fraichement déclarée. Présente après chaque vérification de déclaration, si cette dernière ne retourne aucune erreur alors la procédure d'insertion sera exécutée.

- **Double déclaration :** introduite après la déclaration d'une variable, sert à vérifier si la variable existe déjà dans la table des symboles, retourne une erreur de type double déclaration.

- **Vérification de la déclaration** : introduite en cas d'utilisation d'un identificateur inconnu lors de l'affectation ou autre. Retourne une erreur d'absence de déclaration de la variable considérée.

- **Vérification du dépassement de taille :** programmée dans le but de signaler le dépassement de taille lors d'un accès au tableau considéré. Retourne une erreur de type dépassement de taille.

- **Vérification du type** : indique s'il y a une compatibilité entre les types des variables en cas d'affectation ou d'expression arithmétique/logiques. Retourne une erreur de type absence de compatibilité.

Gestion de la table des symboles :

En ce qui concerne la table des symboles, nous avons opté pour une structure de table dynamique programmée à l'aide des pointeurs. Chaque élément de la table est composé de 4 champs :

- Nom de l'entité (ex : A)
- Type (ex : INTEGER)
- Taille (1 sinon le nombre d'éléments du tableau)
- Code (ex : 'IDF' / 'TAB' en cas de tableau / 'TEMP' pour les variables utilisées dans une boucle)

Prenons un exemple simple :

```
NUMERIC A <- (-4.5)
INTEGER B <- 5
CHARACTER M[8]
LOGICAL P <- FALSE
# Ceci est un commentaire
FOR(I IN 0:2){A+<-1; F<-FALSE and TRUE}
```

Tables des Symboles :

```
                    /***************  Table des symboles  ****************/
-------------------------------------------------------------------------------------------------------
|     Nom Entite         |      Code Entite      |        Type Entite       |       Taille          |
-------------------------------------------------------------------------------------------------------
|          A             |                IDF    |             NUMERIC      |              1        |
|          B             |                IDF    |             INTEGER      |              1        |
|          M             |                TAB    |           CHARACTER      |              8        |
|          P             |                IDF    |             LOGICAL      |              1        |
|          I             |               TEMP    |             INTEGER      |              1        |
|          F             |                IDF    |             LOGICAL      |              1        |

                                  *****  Fin affichage  *****
```

Génération du code intermédiaire :

La structure des quadruplets est une matrice Quad ($\boldsymbol{n}$,4), où $\boldsymbol{n}$ est le nombre de quadruplets.

Un quadruplet est composé de 4 champs :

- OP est l'opérateur.
- ARG1 et ARG2 sont les opérandes.
- RES est le résultat.

Pour générer un quadruplet, nous avons créé une fonction « $Générer$ » qui prends en paramètres les 4 champs cités ci-dessus, cette fonction mets à jour le compteur « $QuadSuivant$ » qui nous indique la ligne (adresse) du quadruplet suivant dans la matrice « $Quad$ ».

Pour générer les variables temporaires, la fonction « $CréerTemp$ » nous permet de mettre à jour « $l'indice$ » (une variable globale initialisée à 1) des variables temporaires utilisées pour la génération des quadruplets.

<u>Les quadruplets générés pour l'exemple précèdent :</u>

```
/***************  Les Quadruplets  ****************/
        0 - ( :=  ,  -4.500000  ,     ,  A )
 ---------------------------------------------------
        1 - ( :=  ,  5  ,     ,  B )
 ---------------------------------------------------
        2 - ( BOUNDS  ,  0  ,  8  ,    )
 ---------------------------------------------------
        3 - ( ADEC  ,  M  ,     ,    )
 ---------------------------------------------------
        4 - ( :=  ,  FALSE  ,     ,  P )
 ---------------------------------------------------
        5 - ( :=  ,  0  ,     ,  I )
 ---------------------------------------------------
        6 - ( BGE  ,  @10  ,  I  ,  2 )
 ---------------------------------------------------
        7 - ( +  ,  A  ,  1  ,  x1 )
 ---------------------------------------------------
        8 - ( :=  ,  x1  ,     ,  A )
 ---------------------------------------------------
        9 - ( +  ,     ,  I  ,  1 )
 ---------------------------------------------------
        10 - ( BR  ,  @6  ,     ,    )
 ---------------------------------------------------
        11 - ( and  ,  FALSE  ,  TRUE  ,  x2 )
 ---------------------------------------------------
        12 - ( :=  ,  x2  ,     ,  F )
 ---------------------------------------------------
        *****  Fin Quadruplets  *****
```

Traitement des erreurs :

Afin d'afficher des messages d'erreurs le plus précisément possible à chaque étape du processus de compilation, nous avons rajouté deux variables « nbLigne et nbColonne » partagées entre « Flex » et « Bison ».

Dans Flex :

```
extern nbLigne;
extern nbColonne ;
static int next = 1;
#define compteur nbColonne = next; next += strlen(yytext);
```

Nous ajoutons la procédure compteur pour chaque entité lexicale afin de comptabiliser le nombre de colonnes dans une ligne.

Dans Bison :

```
#define YYERROR_VERBOSE 1
int nbLigne = 1;
int nbColonne = 1;
```

YYERROR_VERBOSE sert à nous donner plus d'informations concernant l'erreur lexicale ou syntaxique.

Exemple d'erreurs lexicales et syntaxiques :

```
Erreur lexicale: <<h>> a la ligne 1, colonne 9.
Erreur lexicale a la ligne 4, colonne 9: Idf trop long.
Erreur Syntaxique << unexpected mc_moins, expecting space >> a la ligne 2, colonne 13.
```

Procédure d'exécution (sous Windows) :

1. Commande pour compiler le programme Flex : « $flex\ lex.l$ »
   Résultat : création d'un fichier C dont le nom est « $lex.yy.c$ »
2. Commande pour compiler le programme Bison : « $bison - d\ \ syn.y$ »
   Cette commande aura comme résultat 2 fichiers :
   - « $syn.tab.c$ » : c'est un analyseur syntaxique de langage.
   - « $syn.tab.h$ » : contient la liste de tous les terminaux de langage.
3. Commande pour créer le fichier exécutable (le compilateur) :

$$« gcc\ lex.yy.c\ syn.tab.c\ -lfl\ -ly\ -o\ Compilateur »$$

4. Commande pour tester le compilateur sur un fichier texte :

$$« .\backslash Compilateur\ < exemple.txt »$$

```
C:\Users\user\Desktop\Projet>flex lex.l

C:\Users\user\Desktop\Projet>bison -d syn.y

C:\Users\user\Desktop\Projet>gcc lex.yy.c syn.tab.c -lfl -ly -o Compilateur

C:\Users\user\Desktop\Projet>.\Compilateur <exemple.txt
```