



# Compte rendu du projet

Compilation 2

Master 1 en informatique visuelle

Année 2019/2020

Rédigé par : BENAISSA Rania

Matricule : 2015 0000 8321

## I. Procédure de lancement du compilateur


Pour pouvoir lancer et exécuter le compilateur réalisé en langage C, il suffit de suivre cette démarche :

- Installer préalablement Mingw, Flex et Bison.
- Renommer le fichier « com.txt » par « com.bat » (parce qu'il m'est impossible d'envoyer un fichier avec l'extension « bat » par email).
- Ouvrir l'invite de commandes, et se situer dans le répertoire du dossier « Benaissa\_Rania\_code ».
- Pour ma part, j'utilise l'éditeur de texte « Visual Studio Code » qui a un terminal intégré :



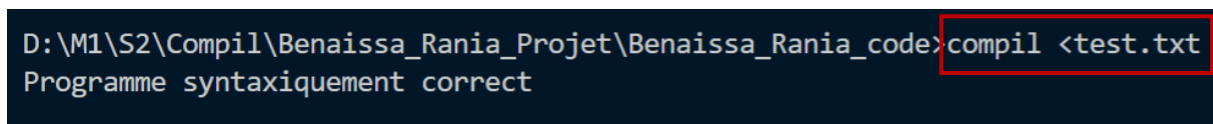
```
Microsoft Windows [version 10.0.18363.1016]
(c) 2019 Microsoft Corporation. Tous droits réservés.
D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code>
```

- Dans l'invite de commandes, exécuter la commande « com » :



```
Microsoft Windows [version 10.0.18363.1082]
(c) 2019 Microsoft Corporation. Tous droits réservés.
D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code>com
D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code>flex lexical.l
D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code>Bison -d syntax.y
D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code>gcc lex.yy.c syntax.tab.c -lfl -ly -o com
pil.exe
D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code>
```

- Dans le dossier « Benaissa\_Rania\_code » se trouve un fichier de test « test.txt » avec un exemple de code en R. Pour l'analyser, il suffit d'exécuter la commande « compil <test.txt » comme suit :



```
D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code>compil <test.txt
Programme syntaxiquement correct
```

- Vous pouvez aussi créer votre propre fichier de test et l'exécuter avec la commande « compil <nom\_du\_fichier.txt »

## II. Parties réalisées

Les parties réalisées sont :

- Analyse lexicale
- Analyse syntaxico-sémantique
- Gestion de la table des symboles
- Génération du code intermédiaire
- Génération du code machine
- Traitement des erreurs

## III. Les phases de réalisation du compilateur

### a. Analyse lexicale

Notre fichier Flex « lexical.l » est décomposé en trois parties :

#### i. Définitions en langage C

Cette partie regroupe les imports de bibliothèques et de headers ainsi que les définitions de variables en C.

```
%{  
    #include <String.h>  
    #include "types.h"  
    #include "syntax.tab.h"  
  
    // compte le nombre de lignes  
    int nb_lines = 1;  
    // compte le nombre de caracteres dans une ligne  
    int nb_chars = 1;  
  
    // Le texte courant à analyser  
    char * currentText = "";  
  
    extern YYSTYPE yylval;  
}%
```

#### ii. Définitions des expressions régulières

On définit les expressions régulières qui serviront à construire nos règles.

```
idf [A-Z]([A-Za-z0-9])  
  
int_pos [0-9]  
  
int_neg "(-"[1-9])  
  
float ([0-9]+."[0-9]+)|("(-"[0-9]+."[0-9]+")  
  
char "'".'"'  
  
bool "TRUE"|"FALSE"  
  
comment "#"^[^n]*
```

### iii. Règles

C'est dans cette partie que l'on définit toutes les entités lexicales possibles de notre compilateur.

Chaque règle est écrite sous forme d'expression régulière à laquelle on associe les actions correspondantes :

```
IN {nb_chars += yyleng;currentText = strdup(yytext);return mc_in;}
IFELSE {nb_chars += yyleng;currentText = strdup(yytext);return mc_ifelse;}
WHILE {nb_chars += yyleng;currentText = strdup(yytext);return mc_while;}
and {nb_chars += yyleng;currentText = strdup(yytext); return mc_and;}
or {nb_chars += yyleng;currentText = strdup(yytext); return mc_or;}
{int_pos} {inserer(yytext,"CONSTANTE","INTEGER",1," / ",nb_lines,nb_chars); nb_cha
{int_neg} {inserer(yytext,"CONSTANTE","INTEGER",1," / ",nb_lines,nb_chars);nb_char
{float} {inserer(yytext,"CONSTANTE","NUMERIC",1," / ",nb_lines,nb_chars);nb_chars
{char} {inserer(yytext,"CONSTANTE","CHARACTER",1," / ",nb_lines,nb_chars);nb_chars
{bool} {inserer(yytext,"CONSTANTE","LOGICAL",1," / ",nb_lines,nb_chars);nb_chars +
```

### b. Analyse syntaxico-sémantique

Pareillement, l'analyseur syntaxique est subdivisé en plusieurs parties que l'on énumère suivant leur ordre de déclaration dans le fichier « syntax.y ».

#### i. Déclarations en langage C

Cette sous partie regroupe les différentes déclarations en langage C :

```
// recupère les variables déclarées dans l'analyseur lexical
extern int nb_lines ;
extern int nb_chars ;
extern char* currentText;
extern int lexical_error;

// type d'un ensemble à sa declaration
char typeEns[20];
// type d'un ensemble à sa declaration
char idfName[20];
// verifie si il recoit 1 cte ou plus
int unique = 1;
//guess if it s an arithm or logical exp
int express = 1;
char type[20];

int yylex(); // Lance l'analyseur lexical
void yyerror(char *s); // fonction d'affichage d'erreurs
```

#### ii. Déclaration des Tokens

Dans ce qui suit, on définit les symboles de la grammaire : les terminaux ainsi que leurs types s'ils doivent en avoir.

```
%token <value>idf <value>mc_float <value>mc_int <value>mc_char
<value>mc_bool mc_if mc_ifelse mc_else mc_for mc_in mc_while
<value>int_pos <value>int_neg <value>float_value <value>char_value
<value>bool_value mc_and mc_or <value>sup_equal <value>inf_equal
<value>equal <value>diff <value>affect saut_ligne
```

### iii. Déclaration des types

Les types créés dans %union servent de types aux entités communes entre Flex et Bison (représentent les types structurés de YYSTYPE dans le fichier « lexical.l »).

```
%union{
    char* value;

    listes bool;

    int number;

    br next;
}
```

De plus, on peut associer ces types aux non-terminaux (ce qui servira lors de l'insertion des routines sémantiques).

```
%type <value> EXP_ARITHME INFO INT_VALUE MGAFF MGAFFEC_COND
| VALUE MDEXP_COMP MGEXP_COMP INCR DECR FOR_LOOP INDICE
%type <bool> EXP_LOG EXP_COMP MGIF_COND MMWHILE_LOOP AFFEC_COND1
%type <number> M MGWHILE_LOOP

%type <next> MDIF_COND BLOC_INSTS IF_COND MDBLOC_INSTS INST
| COND ELSE_COND MDWHILE_LOOP MGFOR_LOOP AFFEC_COND2 AFFEC_COND3
```

### iv. Définition des priorités des opérateurs

Ici on définit les priorités des opérateurs arithmétiques et logiques.

```
%left '+' '-'
%left '*' '/' '%'
%left mc_or mc_and
```

### v. Définition des grammaires

Cette dernière sous partie décrit la grammaire du langage d'autant plus qu'elle associe aux règles définies, les routines sémantiques adéquates (on détaillera les routines dans la section « Génération du code intermédiaire »).

```

INT_VALUE: int_neg {$$ = $1;}
| int_pos {$$ = $1;}
;

ENS: ',' idf ENS {inserer($2,"VARIABLE",typeEns,1,"YES",nb_lines,nb_chars);}
|
;

/* EXPRESSIONS ARITHMETIQUES */

EXP_ARITHME : EXP_ARITHME '+' EXP_ARITHME {
    strcpy(type,checkCompatible($1,$3,nb_lines,nb_chars));
    $$= getTemp();
    inserer($$, "VARIABLE",type,1," / ",nb_lines,nb_chars);
    genererQuad("+",$1,$3,$$);}

```

### c. Gestion de la table des symboles

L'implémentation de la table des symboles se trouve dans le header « tableDesSymboles ».

- La table des symboles est structurée en table de hachage. On la modélise avec un tableau statique de 100 pointeurs.

```

//structure de la table des symboles
typedef struct element *listeH;

typedef struct element
{
    char nom[20];
    char nature[20];
    char type[20];
    long taille;
    char declared[3];
    listeH svt;
} element;

/* table des symboles est un tableau de pointeurs */
listeH ts[100];

```

- Chaque case du tableau pointe vers un objet « element ».
- Un objet « element » est constitué de plusieurs champs :
  - La chaîne de caractères « nom » : représente le nom d'une entité.
  - La chaîne de caractères « nature » : la nature de l'entité peut être un identificateur, un tableau ou une constante.
  - La chaîne de caractères « type » : chaque entité à un type : INTEGER, NUMERIC, CHARACTER ou LOGICAL.
  - L'entier « taille » : si l'entité est un tableau alors sa taille est égale à la taille du tableau. Autrement, la taille est par défaut mise à un.

- La chaine de caractères « declared » : prend comme valeur « Yes » ou « No » et détermine si on a préalablement défini le type de l'entité lors de sa déclaration.
- Le champs « svf » : permet d'accéder au prochain élément de la liste.

Ainsi, pour pouvoir manipuler la table des symboles, nous avons implémenté les fonctions suivantes :

```
// get key according to the idf's name
> int getKey(char entite[]) ...

//hach function
> int hach(int key) ...

//recherche verifie et retourne si une entité existe dans la TS
> listeH recherche(char entite[]) ...

// insere un nouvel element dans la liste chaînée
> listeH insererElement(int indice) ...

//une fonction qui va insérer les entités du programme dans la TS
> void inserer(char entite[], char code[], char type[], long taille, char declared[]) ...

> void checkTableau(char tab[], char indice[], int line, int chars) ...

> char *checkCompatible(char idf1[], char idf2[], int line, int chars) ...

> void divisionParZero(char idf[], int line, int chars) ...

//une fonction pour afficher la TS
> void afficherTS() ...
```

- `getKey ()` : prend en entrée une chaine de caractères, et retourne la somme des codes ASCII de chaque caractère. La valeur retournée représente la clé de la table de hachage.
- `hach ()` : représente la fonction de hachage, elle prend en entrée une clé et retourne l'indice correspondant dans la table des symboles.
- `recherche ()` : prend en entrée une entité et cherche si elle existe dans la table des symboles et retourne l'élément de la liste dans laquelle elle figure.
- `insererElement ()` : insère un nouvel élément à la fin de la liste « indice » de la table des symboles.
- `inserer ()` : insère une nouvelle entité (si elle n'existe pas déjà) dans la table des symboles.
- `checkTableau ()` : vérifie si l'indice du tableau ne dépasse pas sa taille.
- `checkCompatible ()` : vérifie la compatibilité de types entre deux entités de la table des symboles.
- `divisionParZero ()` : vérifie s'il y a bien eu une division par 0.
- `afficherTS ()` : affiche le contenu de la table des symboles.

Et voici un exemple de code en R analysé par notre compilateur et la table des symboles qui en découle :

```

INTEGER Number, Value
LOGICAL BOOL

WHILE( (Value != 5))
{
  IF( (Value != 4 ) )
  {
    Dec <- 5.7
    Hello <- (Value > 5) and (Number == 3)
  }
  ELSE IF((Value > 5))
  {
    Number <- 0
  }
}

```



Nom	Nature	Type	Taille	Déclaré
BOOL	VARIABLE	LOGICAL	1	YES
Hello	VARIABLE	LOGICAL	1	NO
Value	VARIABLE	INTEGER	1	YES
Number	VARIABLE	INTEGER	1	YES
0	CONSTANTE	INTEGER	1	/
3	CONSTANTE	INTEGER	1	/
4	CONSTANTE	INTEGER	1	/
5	CONSTANTE	INTEGER	1	/
5.7	CONSTANTE	NUMERIC	1	/
Dec	VARIABLE	NUMERIC	1	NO

#### d. Génération du code intermédiaire

Le code intermédiaire est représenté par une table de quadruplets dont l'implémentation se trouve dans le header « quadruplets ».

```

typedef struct quad
{
  char op[5];
  char arg1[20];
  char arg2[20];
  char res[20];
} quad;

quad quads[100];

// L indice du quad courant
int currentQuad = 0;

//indice des vars temporaires
int currentTemp = 1;
// taille de la matrice
const int quadsSize = 100;

```

- Le table de quadruplets est un tableau d'au plus 100 objets « quad ».
- Chaque objet « quad » contient les champs suivants :
  - La chaîne de caractères « op » : représente l'opérateur du quadruplet.
  - La chaîne de caractères « arg1 » : définit le premier argument du quadruplet s'il existe.
  - La chaîne de caractères « arg2 » : définit le deuxième argument du quadruplet s'il existe.



- La chaîne de caractères « res » : peut être une variable, une variable temporaire ou une adresse au vu d'un branchement.

Afin d'injecter les routines sémantiques dans notre grammaire, nous aurons besoin des fonctions suivantes :

```

/* converts an int into a char */
> char *intToChar(int val) ...

// genere le quad
> int genererQuad(char op[], char arg1[], char arg2[], char res[])

// modifie l'adresse du quad (la partie resultat)
> void modifyQuad(int pos, int res) ...

// creer une var temporaire
> char *getTemp() ...

// ajoute la position d'un quad dans une des listes
> liste addPosQuad(liste l) ...

> liste removePosQuad(liste l, int value) ...

> liste fusion(liste l1, liste l2) ...

> void afficherQuads() ...

```

- `intToChar()` : convertit un entier en chaîne de caractères.
- `genererQuad()` : ajoute un nouveau quadruplet à la table des quadruplets.
- `modifyQuad()` : modifie le champs « res » d'un quadruplet à la position « pos » de la table des quadruplets.
- `getTemp()` : créer une variable temporaire. Elles sont de la forme : x1, x2, ...
- `addPosQuad()` : rajoute la position du quadruplet courant dans une liste donnée.
- `removePosQuad()` : met à jour les champs « res » des quadruplets d'une liste avec la valeur « value » puis vide cette liste.
- `fusion()` : fusionne deux listes en une seule liste.
- `afficherQuads()` : affiche la table des quadruplets.

Pour le même exemple que celui de la table des symboles, voici la table des quadruplets correspondante :

Quadruplets				
Ligne	OP	ARG1	ARG2	RES
1	!=	Value	5	3
2	BR			19
3	!=	Value	4	5
4	BR			14
5	<-	5.7		Dec
6	>	Value	5	8
7	BR			12
8	==	Number	3	10
9	BR			12
10	<-	TRUE		Hello
11	BR			13
12	<-	FALSE		Hello
13	BR			18
14	>	Value	5	16
15	BR			18
16	<-	0		Number
17	BR			18
18	BR			1
19				

#### e. Génération du code machine

On génère le code machine sous forme de tableau de blocs d'instructions où chaque bloc est une traduction d'un quadruplet donné, comme le montre la figure suivante :

Quadruplets				
Ligne	OP	ARG1	ARG2	RES
1	!=	Value	5	3
2	BR			19
3	!=	Value	4	5
4	BR			14
5	<-	5.7		Dec
6	>	Value	5	8
7	BR			12
8	==	Number	3	10
9	BR			12
10	<-	TRUE		Hello
11	BR			13
12	<-	FALSE		Hello
13	BR			18
14	>	Value	5	16
15	BR			18
16	<-	0		Number
17	BR			18
18	BR			1
19				

Code Objet	
Numero du quadruplet	Instruction assembleur
1	Label_6 : CMP Value, 5
2	JE Label_1
3	CMP Value, 4
4	JE Label_2
5	MOV Dec,5.7
6	CMP Value, 5
7	JBE Label_3
8	CMP Number, 3
9	JNE Label_3
10	MOV AX,TRUE
11	MOV Hello,AX
12	JMP Label_4
13	Label_3 : MOV AX,FALSE
14	MOV Hello,AX
15	Label_4 : JMP Label_5
16	Label_2 : CMP Value, 5
17	JBE Label_5
18	MOV Number,0
19	JMP Label_5
20	Label_5 : JMP Label_6
21	Label_1 :

Pour aboutir à ce résultat, nous avons défini les structures et les fonctions suivantes :

```
typedef struct instruction
{
    char numQuad[10];
    char inst[100];
} instruction;

instruction code[1000];

// L'indice de l'instruction courante
int currentCode = 0;

/* LABELS */
typedef struct label
{
    char name[50];
    char numQuad[10];
} label;

label labels[100];
```

- Où la variable « code » est un tableau de 1000 objets « instruction ».
- Un objet « instruction » est composé d'une instruction à laquelle on associe un numéro de quadruplet.
- La variable « labels » est un tableau de 100 objets « label » où l'on fait correspondre à chaque nom d'étiquette, un numéro de quadruplet.

```

> void afficherCode() ...

> void generateArithm(int i) ...

> char *labelExists(char *numQuad) ...

> char *createLabel(char *numQuad) ...

> int generateCmp(int i) ...

> int isAlpha(char k) ...

> int generateAffect(int i) ...

> void addLabels() ...

// genere le code objet a partir de la table des quadruplets optimisée
> void generateCode() ...

```

- `afficherCode ()` : affiche le table du code machine.
- `generateArithm ()` : génère le code assembleur d'une instruction arithmétique.
- `labelExists ()` : vérifie l'existence d'une étiquette (label).
- `createLabel ()` : crée un label s'il n'existe pas déjà.
- `generateCmp ()` : génère le code assembleur d'une instruction de comparaison.
- `isAlpha ()` : vérifie si un caractère est un caractère alphabétique en majuscule.
- `generateAffect ()` : génère le code assembleur d'une instruction d'affectation.
- `addLabels ()` : rajoute les étiquettes créées à la table du code machine.
- `generateCode ()` : génère la table du code machine à partir de la table des quadruplets.

#### f. Traitement des erreurs

Pour chacune de ces phases de compilation, lorsqu'une erreur est détectée, le compilateur s'arrête tout en affichant un message d'erreur précisant la ligne, la colonne et la nature de l'erreur.

##### i. Erreurs lexicales

Les erreurs lexicales peuvent être engendrées dans deux cas :

- Un identificateur dépasse 10 caractères :

<pre> 1 2  INTEGER Number, Value 3  LOGICA BOOL, JeSuisUnIdfLong </pre>	➡	<pre> D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code&gt;compil &lt;test.txt Erreur lexicale a la ligne 3 colonne 15 sur l'entite  JeSuisUnIdfLong  : identificateur trop long. </pre>
---	---	--

- L'entité en cours d'analyse ne figure pas dans le dictionnaire défini :

<pre> 1 2  INTEGER Number, Value 3  LOGICAL BOOL 4  dsd </pre>	➡	<pre> D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code&gt;compil &lt;test.txt Erreur lexicale a la ligne 4, colonne 1, sur l'entite : d : entite non definie </pre>
--	---	--

## ii. Erreurs syntaxiques

Si la syntaxe ne correspond pas aux grammaires définies alors une erreur syntaxique est signalée :

<pre> 1 2  INTEGER Number, Value 3  LOGICAL BOOL 4 5  WHILE( (Value = 5)) </pre>	➡	<pre> D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code&gt;compil &lt;test.txt Erreur syntaxique a la ligne 5, colonne 16, sur l'entite : = </pre>
--	---	--

## iii. Erreurs sémantiques

Elles peuvent être générées pour les raisons suivantes :

- Une division par la constante zéro.
- Incompatibilité de types entre des variables, constantes et tableaux.
- Identificateur non déclaré.
- Double déclaration d'un identificateur (seulement dans le cas d'une déclaration avec mention du type).
- Dépassement de la taille d'un tableau.

Prenons pour exemple une incompatibilité de types :

<pre> 1 2  INTEGER Number, Value 3  LOGICAL BOOL 4 5  WHILE( (Value == BOOL)) 6  { </pre>	➡	<pre> D:\M1\S2\Compil\Benaissa_Rania_Projet\Benaissa_Rania_code&gt;compil &lt;test.txt Erreur semantique a la ligne 5, colonne 23 :  &lt; Value &gt; et &lt;BOOL&gt; sont incompatibles. </pre>
---	---	---

## IV. Conclusion

En réalisant ce projet, on a pu mettre en pratique les connaissances acquises en théorie et nous avons abouti, à travers différentes phases, à l'implémentation d'un compilateur fonctionnel.