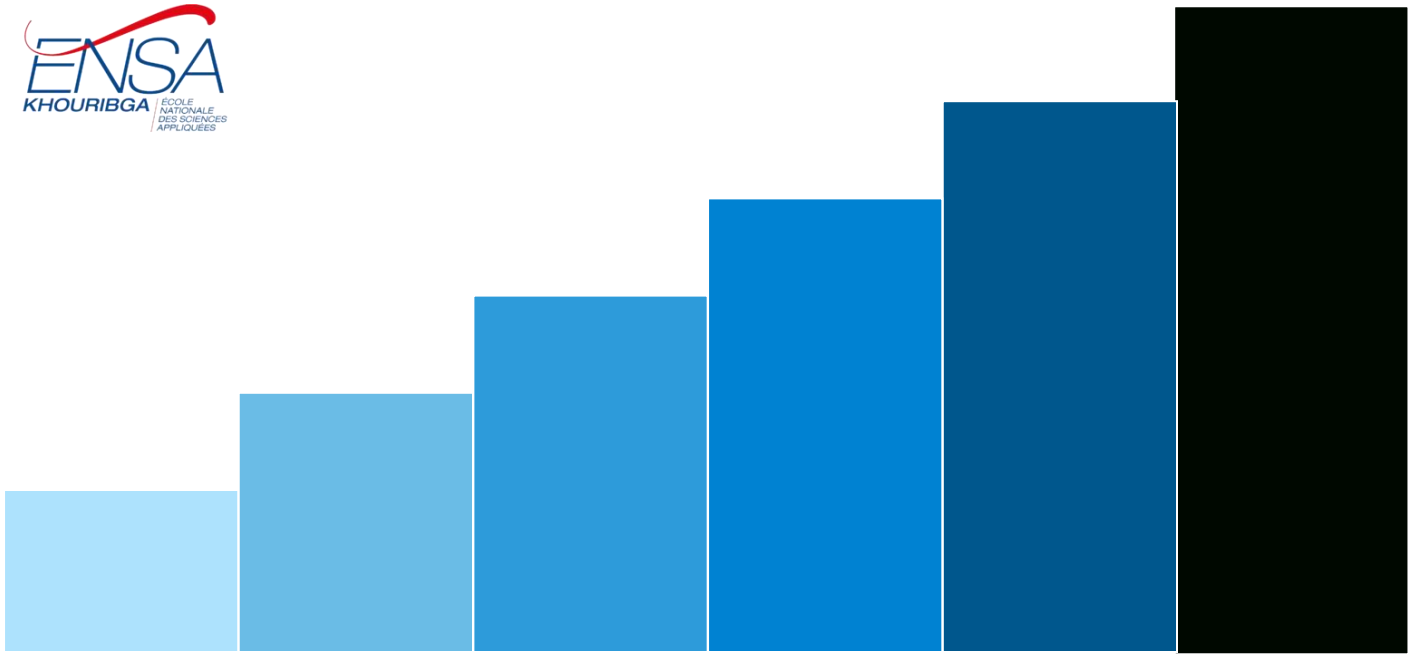


Module Théorie de la calculabilité & Compilation

Mini Projet

Sujet N°2 les ensembles

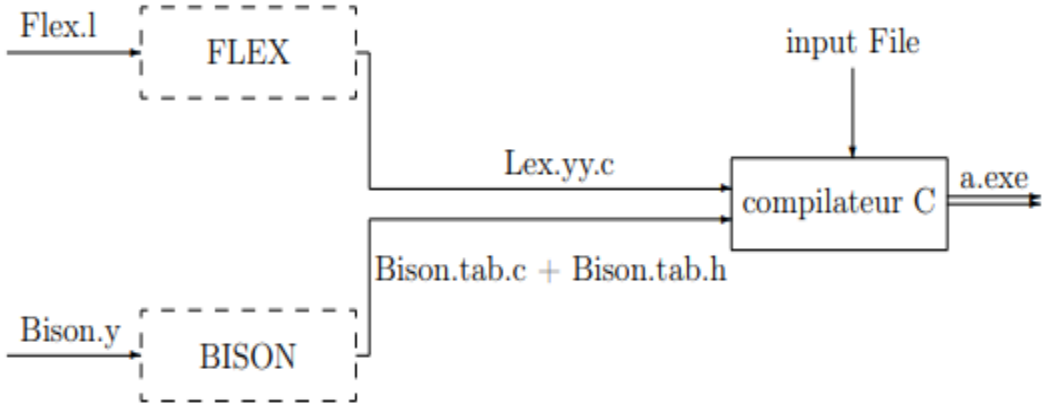
2020-2021

Encadré par : M. Abdelmajid Dargham
Réalisé par : Abderrazak Daoudi N°10
Khadija Belaili

OBJECTIFS :

L'objectif de ce projet de compilation est de se familiariser aux commandes **flex** et **bison,** les deux outils de compilation par défaut sur les systèmes **Unix** depuis plusieurs décennies (déjà !). Le premier outil **flex** (version **Gnu** de la commande **Lex**) construit un ***analyseur lexical*** à partir d'un ensemble de ***règles/actions*** décrites par des ***expressions régulières***. Le second outil **bison** est un compilateur de compilateur, version Gnu de la célèbre commande **yacc** acronyme de « yet another compiler of compilers ». Il construit un compilateur d'un langage décrit par un ensemble de règles et actions d'une ***grammaire LARL*** sous une forme proche de la ***forme BNF*** de Backus-Naur.

Couple Flex Bison



une grammaire hors-contexte pour le langage Ensemble.

Symboles non-terminaux :

- **Les identificateurs : A,B,C,D ou E**

Symboles terminaux :

- **Les constantes**
- **L'opérateur d'affectation**
- **Les opérateurs ensemblistes**
- **Les séparateurs**
- **Les délimiteurs**
- **L'instruction d'affichage**

Variable initiale (axiome) :

- **{I}**

Ensemble des règles :

I=> al.a.M.s|al.a.o.s|aff.s|ε //instruction(affectation , opération ,affichage)

a=> :=| ε //une affectation

p=> { | } | (|) | ε //les délimiteurs

n=> 1|2|3|4|5|6|7|8|9|0| ε|nn //les nombres entiers

s=>,| ;| ε les séparateurs

r=>p.n.s.n.p | ε une ligne de l'ensemble

m=>p.r.s.r.p| ε une identification

op=>+|-|*| ε un operateur

o=>al.op.al.op.o|al| ε les opérations sur les matrices

aff=> afficher.p.al.p| ε la fonction afficher

Analyseur lexical flex.

Flex est un outil pour générer des analyseurs, programmes qui reconnaissent des motifs lexicaux dans du texte. Il lit les fichiers d'entrée donnés, ou bien son entrée standard si aucun fichier n'est donné, pour obtenir la description de l'analyseur à générer. La description est une liste de paires d'expressions rationnelles et de code C, appelées règles. En sortie, flex génère un fichier source en langage C, appelé `lex.yy.c', qui définit une routine `yylex()'. Ce fichier est compilé et lié avec l'option `-lfl' (correspondant à la bibliothèque flex) afin de produire un programme exécutable. Quand l'exécutable est lancé, il analyse son entrée afin d'y trouver des occurrences correspondant aux précédentes expressions régulières. À chaque expression trouvée, il exécute le code C correspondant.

La structure d'un programme **flex** est similaire à celle d'une source **bison**. La source d'un programme **flex** est découpée en 4 zones séparées par les balises **%{** , **%}**, **%%**, **%%**.

L'analyseur lexical de l'exemple ci-dessous recherche le mot le plus long tout en calculant la somme des entiers rencontrés dans le fichier. Il utilise deux variables prédéfinies : yytext et yyleng

- **Header de fichier flex cla.l**

```
%{

#include "simple.h"
unsigned int lineno=1;
bool error_lexical=false;

%}

%option noyywrap

nombre 0|[1-9][[:digit:]]*
variable_identificateur U|V|W|X|Y
```

- **Déclaration du lexique du notre langage.**

```
%%

{nombre} {
        sscanf(yytext, "%ld", &yylval.nombre);
        return TOK_NOMBRE;
}

"BEGIN"      {return TOK_BEGIN;}

"FIN."      {return TOK_FIN;}

"afficher"      {return TOK_AFFICHER;}

":="             {return TOK_AFFECT;}

"+"             {return TOK_ADD;}

"-"             {return TOK_DIFF;}

"*"             {return TOK_MULT;}

"("             {return TOK_PARG;}

")"             {return TOK_PARD;}

"["             {return TOK_PG;}

"]"             {return TOK_PD;}

";"             {return TOK_FINSTR;}

","             {return TOK_VER;}

"\n"            {lineno++;}

{variable_identificateur} {
        yylval.texte = yytext;
        return TOK_VARB;
}
```

```
" "|"\t" {}

. {
        fprintf(stderr,"\tERREUR : Lexeme inconnu a la ligne %d. Il s'agit de %s et comporte %d
        lettre(s)\n",lineno,yytext,yyleng);
        error_lexical=true;
        return yytext[0];
}

%%
```

Analyseur syntaxique Bison

Nous avons créé un analyseur lexical. Maintenant il s'agira de faire l'analyseur syntaxique. C'est la partie la plus complexe du cours. Bison permet de générer un analyseur syntaxique et il est très bien utilisé avec Flex. Notre analyseur syntaxique recevra les tokens qu'enverra l'analyseur lexical. Les tokens sont en fait les entités lexicales lues par Flex. Ils correspondraient aux terminaux du langage. Au fur et à mesure que notre futur analyseur syntaxique recevra les tokens, il regardera si leur ordre est correct ou non. Un fichier Bison génère un fichier en C (tout comme fait Flex aussi quand il génère son analyseur lexical) où il va construire la fonction yyparse(). Cette fonction parse du contenu textuel et fait une analyse syntaxique. Elle appelle automatiquement la fonction yylex() de notre analyseur lexical pour récupérer les tokens. On a donc plus besoin de la fonction main() dans notre fichier Flex.

- **header pour fichier cal.y**

```
%{

#include "simple.h"
#include <string.h>
bool error_syntaxical=false;
extern unsigned int lineno;
extern bool error_lexical;

%}
```

- Declartion

  Nous avons ici les operateurs, ils sont definis par leur ordre de priorite. Si je definis par exemple la multiplication en premier et l'addition apres, le + l'emportera alors sur le * dans le langage. Les parenthese sont prioritaires avec %right

```
/* L'union dans Bison est utilisee pour typer nos tokens ainsi que nos non terminaux. Ici nous avons declare une union avec deux types : nombre de type int
et texte de type pointeur de char (char*) */

%union {
        long nombre;
        char* texte;
}

/* Nous avons ici les operateurs, ils sont definis par leur ordre de priorite. Si je definis par exemple la multiplication en premier et l'addition apres,
le + l'emportera alors sur le * dans le langage. Les parenthese sont prioritaires avec %right */

%left                   TOK_UNION        TOK_DIFF     /* +- */
%left                   TOK_INTER         TOK_DIV         /* /* */
%left                   TOK_VER
%right                  TOK_PARG        TOK_PARD        /* () */
%right                  TOK_PG        TOK_PD        /* {} */
```

  Nous avons la liste de nos expressions (les non terminaux). Nous les typons tous en texte (pointeur vers une zone de char).

```
/* Nous avons la liste de nos expressions (les non terminaux). Nous les typons tous en texte (pointeur vers une zone de char). */

%type<texte>            code
%type<texte>            BEGIN
%type<texte>            FIN
%type<texte>            instruction
%type<texte>            variable_identificateur
%type<texte>            variable_arithmetique
%type<texte>            affectation
%type<texte>            affichage
%type<texte>            expression_arithmetique
%type<texte>            expression_identificateur
%type<texte>            union
%type<texte>            diff
%type<texte>            inter

/* Nous avons la liste de nos tokens (les terminaux de notre grammaire) */

%token<texte>           TOK_NOMBRE
%token                  TOK_BEGIN       /* BEGIN */
%token                  TOK_FIN        /* FIN. */
%token                  TOK_AFFECT      /* := */
%token                  TOK_FINSTR      /* ; */
%token                  TOK_VER         /* , */
%token                  TOK_POINT         /* . */
%token                  TOK_AFFICHER    /* afficher */
%token                  TOK_SUPPR       /* supprimer */
%token<texte>           TOK_VARE        /* variable arithmetique */
%token<texte>           TOK_VARB        /* variable identificateur */

%%
```

ici , Nous definissons toutes les regles grammaticales de chaque non terminal de notre langage. Par defaut on commence a definir l'axiome, c'est a dire ici le non terminal code. Si nous le definissons pas en premier nous devons le specifier en option dans Bison avec %start

```
%%

/* Nous definissons toutes les regles grammaticales de chaque non terminal de notre langage. Par defaut on commence a definir l'axiome, c'est a dire ici le
non terminal code. Si nous le definissons pas en premier nous devons le specifier en option dans Bison avec %start */

code:           %empty{}
                |
                code instruction{
                        printf("\033[92m\t\tResultat pour ligne %d: C'est une instruction valide !\n\n\n\n\n\033[0m",lineno);
                }
                |
                code error{
                        fprintf(stderr,"\033[91m\tERREUR : Erreur de syntaxe a la ligne %d.\n\033[0m",lineno);
                        error_syntaxical=true;
                };

instruction:    affectation{
                        printf("\t\tInstruction type Affectation\n");
                }
                |
                affichage{
                         printf("\t\tInstruction type Affichage\n");
                }
                |
                BEGIN{
                         printf("\t\tInstruction type BEGIN\n");
                }
                |
                FIN{
                        printf("\t\tInstruction type FIN\n");
                };


variable_identificateur: TOK_VARB{
                                printf("\033[33m\t\t\tVariable: \033[0m %s\n",$1);
                                $$=strdup($1);
                        };
```

```
variable_arithmetique:  TOK_VARE{
                                printf("\033[33m\t\t\tVariable %s\n\033[0m",$1);
                                $$=strdup($1);
                        };


affectation:    variable_identificateur TOK_AFFECT expression_identificateur TOK_FINSTR{
                        /* $1 est la valeur du premier non terminal. Ici c'est la valeur du non terminal variable. $3 est la valeur du 2nd non terminal. */
                        printf("\t\tAffectation sur la variable %s\n",$1);
                }
                |
                variable_identificateur TOK_AFFECT expression_arithmetique TOK_FINSTR{
                        printf("\t\tAffectation sur l'identificateur %s\n",$1);
                };


affichage:      TOK_AFFICHER expression_identificateur TOK_FINSTR{
                        printf("\t\tAffichage de la valeur de l'expression %s\n",$2);
                };

BEGIN:          TOK_BEGIN  {
                        printf("\033[96m\t\tDemarrage avec BEGIN\033[0m\n");
                };

FIN:           TOK_FIN {
                        printf("\033[96m\t\tFermeture avec FIN\033[0m\n");
                };
```

```
expression_identificateur:
                                variable_identificateur{
                                        $$=strdup($1);
                                }
                                |
                                union{
                                }
                                |
                                diff{
                                }
                                |
                                inter{
                                }
                                |
                                TOK_PARG expression_identificateur TOK_PARD{
                                        printf("\t\t\tC'est une expression identificateur entre parentheses\n");
                                        $$=strcat(strcat(strdup("("),strdup($2)),strdup(")"));
                                };


expression_arithmetique:        TOK_NOMBRE{
                                        printf("\033[33m\t\t\tNombre :\033[0m %ld\n",$1);
                                        /* Comme le token TOK_NOMBRE est de type entier et que on a type expression_arithmetique comme du texte, il nous
                                        faut convertir la valeur en texte. */
                                        int length=snprintf(NULL,0,"%ld",$1);
                                        char* str=malloc(length+1);
                                        snprintf(str,length+1,"%ld",$1);
                                        $$=strdup(str);
                                        free(str);
                                }
                                |
                                variable_arithmetique{
                                        $$=strdup($1);
                                }
                                |
                                expression_arithmetique TOK_VER expression_arithmetique{
                                        $$=strcat(strcat(strdup($1),strdup(",")),strdup($3));
                                }
                                |
                                TOK_VER expression_arithmetique TOK_VER{

                                        $$=strcat(strcat(strdup("{"),strdup($2)),strdup(","));
                                }
                                |
                                TOK_VER expression_arithmetique TOK_PD{

                                        $$=strcat(strcat(strdup(","),strdup($2)),strdup("}"));
                                }
                                |
                                TOK_PG expression_arithmetique TOK_PD{

                                        $$=strcat(strcat(strdup("{"),strdup($2)),strdup("}"));
                                };
```

ici on définit les opérateurs ensemblistes union, diff ,inter

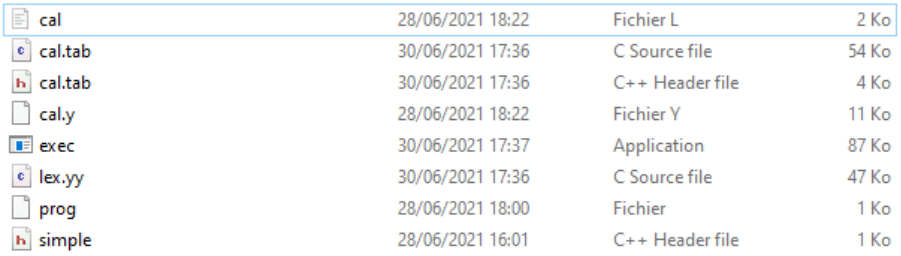
```
union:  expression_identificateur TOK_UNION expression_identificateur{printf("\t\t\tunion\n");$$=strcat(strcat(strdup($1),strdup("+")),strdup($3));};
diff:   expression_identificateur TOK_DIFF expression_identificateur{printf("\t\t\tdifference\n");$$=strcat(strcat(strdup($1),strdup("-")),strdup($3));};
inter: expression_identificateur TOK_INTER expression_identificateur{printf("\t\t\tintersection\n");$$=strcat(strcat(strdup($1),strdup("*")),strdup($3));};

%%
```

Execution

On compile le compilator:

```
C:\Users\user>cd C:\Users\user\Desktop\projet__

C:\Users\user\Desktop\projet__>flex cal.l

C:\Users\user\Desktop\projet__>bison -d cal.y

C:\Users\user\Desktop\projet__>gcc -o exec lex.yy.c cal.tab.c

C:\Users\user\Desktop\projet__>exec < prog
```

On voit qu'après l'exécutions des commandes précédents ; des fichiers .c ont créé .
Lex.yy.c pour .l du fichier flex et cal.tab.c et cal.tab.h pour bison .



cal	28/06/2021 18:22	Fichier L	2 Ko
cal.tab	30/06/2021 17:36	C Source file	54 Ko
cal.tab	30/06/2021 17:36	C++ Header file	4 Ko
cal.y	28/06/2021 18:22	Fichier Y	11 Ko
exec	30/06/2021 17:37	Application	87 Ko
lex.yy	30/06/2021 17:36	C Source file	47 Ko
prog	28/06/2021 18:00	Fichier	1 Ko
simple	28/06/2021 16:01	C++ Header file	1 Ko

On prend comme un programme :

```
BEGIN

B := {2,3,5,7};
C := (A+B)*D;
afficher(C);
A := D*(A+B+C);
afficher(D);

FIN.
```

On l'exécute avec le fichier exec généré par gcc :

```
C:\Users\user\Desktop\projet__>exec < prog
```

On a comme résultat :

```
--------------------------------------------------------------------------------
-----------------------Debut de l'analyse syntaxique :--------------------------
--------------------------------------------------------------------------------
               Demarrage avec BEGIN
               Instruction type BEGIN
               Resultat pour ligne 1: C'est une instruction valide !



                       Variable:  A
                       Nombre : 3
               Affectation sur l'identificateur A
               Instruction type Affectation
               Resultat pour ligne 3: C'est une instruction valide !



                       Variable:  B
                       Nombre : 2
                       Nombre : 3
                       Nombre : 5
                       Nombre : 7
               Affectation sur l'identificateur B
               Instruction type Affectation
               Resultat pour ligne 4: C'est une instruction valide !



                       Variable:  C
                       Variable:  A
                       Variable:  B
                       union
                       C'est une expression identificateur entre parentheses
                       Variable:  D
                       intersection
               Affectation sur la variable C
               Instruction type Affectation
               Resultat pour ligne 5: C'est une instruction valide !

                       Variable:  C
                       C'est une expression identificateur entre parentheses
               Affichage de la valeur de l'expression (C)
               Instruction type Affichage
               Resultat pour ligne 6: C'est une instruction valide !



                       Variable:  A
                       Variable:  D
                       Variable:  A
                       Variable:  B
                       union
                       Variable:  C
                       union
                       C'est une expression identificateur entre parentheses
                       intersection
               Affectation sur la variable A
               Instruction type Affectation
               Resultat pour ligne 7: C'est une instruction valide !



                       Variable:  D
                       C'est une expression identificateur entre parentheses
               Affichage de la valeur de l'expression (D)
               Instruction type Affichage
               Resultat pour ligne 8: C'est une instruction valide !



               Fermeture avec FIN
               Instruction type FIN
               Resultat pour ligne 10: C'est une instruction valide !



--------------------------------------------------------------------------------
-------------------------------Fin de l'analyse !-------------------------------
--------------------------------------------------------------------------------
Resultat :
       -- Succes a l'analyse lexicale ! --
       -- Succes a l'analyse syntaxique ! --
```

Comme vous voyez notre programme est bien vérifier et on a pas des erreurs, par contre on va essayer de vérifier autre programme mais cette fois avec des erreurs :
Notre programme sera :

```
BEGIN

B := {2,3,5,7};
C := (A+B)D;
afficher(C);
A := D*(A+B+C);
aff(D);

FIN.
```

Le résultat sera :

```
--------------------------------------------------------------------------------
----------------------Debut de l'analyse syntaxique :---------------------------
--------------------------------------------------------------------------------
               Demarrage avec BEGIN
               Instruction type BEGIN
               Resultat pour ligne 1: C'est une instruction valide !



                       Variable:  B
                       Nombre : 2
                       Nombre : 3
                       Nombre : 5
                       Nombre : 7
               Affectation sur l'identificateur B
               Instruction type Affectation
               Resultat pour ligne 3: C'est une instruction valide !



                       Variable:  C
                       Variable:  A
                       Variable:  B
                       union
                       C'est une expression identificateur entre parentheses
Erreur de syntaxe a la ligne 4: syntax error
       ERREUR : Erreur de syntaxe a la ligne 4.
                       Variable:  D
       ERREUR : Erreur de syntaxe a la ligne 4.
       ERREUR : Erreur de syntaxe a la ligne 4.
                       Variable:  C
                       C'est une expression identificateur entre parentheses
               Affichage de la valeur de l'expression (C)
               Instruction type Affichage
               Resultat pour ligne 5: C'est une instruction valide !

                      Variable:  A
                      Variable:  D
                      Variable:  A
                      Variable:  B
                      union
                      Variable:  C
                      union
                      C'est une expression identificateur entre parentheses
                      intersection
              Affectation sur la variable A
              Instruction type Affectation
              Resultat pour ligne 6: C'est une instruction valide !



       ERREUR : Lexeme inconnu a la ligne 7. Il s'agit de a et comporte 1 lettre(s)
Erreur de syntaxe a la ligne 7: syntax error
       ERREUR : Erreur de syntaxe a la ligne 7.
       ERREUR : Erreur de syntaxe a la ligne 7.
       ERREUR : Lexeme inconnu a la ligne 7. Il s'agit de f et comporte 1 lettre(s)
       ERREUR : Erreur de syntaxe a la ligne 7.
       ERREUR : Lexeme inconnu a la ligne 7. Il s'agit de f et comporte 1 lettre(s)
       ERREUR : Erreur de syntaxe a la ligne 7.
       ERREUR : Erreur de syntaxe a la ligne 7.
                      Variable:  D
       ERREUR : Erreur de syntaxe a la ligne 7.
       ERREUR : Erreur de syntaxe a la ligne 7.
       ERREUR : Erreur de syntaxe a la ligne 7.
               Fermeture avec FIN
               Instruction type FIN
               Resultat pour ligne 9: C'est une instruction valide !



--------------------------------------------------------------------------------
------------------------------Fin de l'analyse !--------------------------------
--------------------------------------------------------------------------------
Resultat :
       -- Echec : Certains lexemes ne font pas partie du lexique du langage ! --
       -- Echec a l'analyse lexicale --
       -- Echec : Certaines phrases sont syntaxiquement incorrectes ! --
       -- Echec : Certaines phrases sont syntaxiquement incorrectes ! --
       -- Echec a l'analyse syntaxique --
```

Comme vous voyez notre code contient des erreurs dans les lignes 4 et 7

Conclusion :

Ce projet était une occasion pour mettre en œuvre les concepts ´élémentaires de compilation de langage de programmation moderne. Nous nous sommes familiarisées avec des outils d'analyse lexicale tel LEX et d'analyse syntaxique tel YACC. Le compilateur d´evelopp´e, qui colle aux sp´ecifications impos´ees par le cahier des charges, peut certainnement ˆetre améliore.