



Compilation

Réalisation d'un compilateur simplifié

Kristen Manac'h, Jean-Marie Le Yaouanc

Brest, 4 janvier 2005

Enseignant responsable : Mr P. Le Parc

Département informatique

UFR Sciences et Techniques 6 avenue Le Gorgeu 29200 Brest

Master 1 informatique

Table des matières

1	Introduction	1
2	Le langage Easy Frog	2
2.1	Structure générale d'un programme écrit en <i>Easy Frog</i>	2
2.1.1	Les marqueurs #debut et #fin	2
2.1.2	Les commentaires	2
2.2	Variables et constantes	3
2.2.1	Nom de variable ou de constante	3
2.2.2	Les constantes	3
2.2.3	Les variables	3
2.2.4	Utilisation des variables et des constantes	3
2.2.5	Les types	4
2.3	Les instructions	4
2.4	Les expressions	5
2.4.1	L'opérateur d'affectation VAUT	5
2.4.2	Les opérateurs mathématiques	5
2.5	Les instructions de contrôle	7
2.5.1	Les instruction SI et FINSI	7
2.5.2	La clause SINON	7
2.5.3	L'instruction POUR ... JUSQUE ... PAS ... FIN- POUR	8
2.5.4	Les structures imbriquées	8
2.6	Les instructions d'entrées sorties	8
2.6.1	L'instruction LIRE	8
2.6.2	L'instruction ECRIRE	9
3	Commentaires	10
3.1	Analyse lexicale	10
3.2	Analyse syntaxique	10
3.2.1	Le fichier cible	10
3.2.2	Les tables de symboles	11
3.2.3	Le contrôle de type	11
3.2.4	Structures de contrôle	11

4 Conclusion	12
A Annexes	13
A.1 Analyseur lexical	13
A.1.1 Analyseur lexical sans instructions C	13
A.1.2 Analyseur lexical avec instructions C	14
A.2 Analyseur syntaxique	16
A.2.1 Analyseur syntaxique sans instructions C	16
A.2.2 Analyseur syntaxique avec instructions C	19
A.3 Exemples	31
A.3.1 Exemple 1	31
A.3.2 Exemple 2	33
A.3.3 Exemple 3	34

Chapitre 1

Introduction

L'objectif de ce projet est de proposer un langage informatique de type impératif et de construire un compilateur qui partant d'un programme source, génère un programme cible en langage C, compilable et exécutable.

Les fichiers sont disponibles sous manac.kr/projet_compil .

Chapitre 2

Le langage Easy Frog

2.1 Structure générale d'un programme écrit en *Easy Frog*

```
$$ Déclaration de variables et de constantes  
$$ Définition des procédures
```

```
#debut  $$ Début du programme  
        $$ Suite d'instructions  
#fin    $$ Fin du programme
```

2.1.1 Les marqueurs `#debut` et `#fin`

Le marqueur `#debut` est l'instruction obligatoire d'un programme. Il marque le début d'un programme. Le marqueur `#fin` est l'instruction obligatoire de fin d'un programme.

Remarque : les deux lignes suivantes constituent un programme :

```
#debut  
#fin
```

2.1.2 Les commentaires

Un texte qui suit `$$` jusqu'au retour chariot est un commentaire.

```
$$Ceci est un commentaire  
#debut $$Ceci est aussi un commentaire  
#fin
```

2.2 Variables et constantes

2.2.1 Nom de variable ou de constante

Il peut contenir des lettres, des chiffres et le caractère `_`.
Le premier caractère doit être une lettre.

2.2.2 Les constantes

Les constantes sont définies de la manière suivante :

CONSTANTE *TYPE nom valeur*.

Les constantes doivent impérativement être initialisées. Chaque déclaration se termine par un retour chariot.

- constante entier NB 10
- constante reel TAILLE 2.3

2.2.3 Les variables

Les variables sont définies de la manière suivante :

VARIABLE *TYPE nom*.

Il est possible d'initialiser une variable de la manière suivante : **variable** *type nom valeur*. Chaque déclaration se termine par un retour chariot.

- variable chaine nom "toto"
- variable bin test

La déclaration des constantes et des variables se fait au début du programme avant le marqueur `#debut`.

2.2.4 Utilisation des variables et des constantes

Le nom de la variable suffit pour y accéder en lecture et en écriture. Le nom de la constante suffit pour y accéder en lecture.

L'affectation des variables se fait avec le mot-clé **VAUT**.

- `variable entier maVariable_1`
`#debut`
`maVariable_1 VAUT 1`
`#fin`
- `constante entier maConstante_1 1`
`variable entier maVariable_1`
`#debut`
`maVariable_1 VAUT maConstante_1`
`#fin`

2.2.5 Les types

Le type entier ENTIER

Les nombres entiers sont traités par le type **ENTIER**. Ils peuvent être négatifs et sont alors précédés de l'opérateur unaire -.

```
variable entier maVariable_2
#debut
maVariable_2 VAUT -1
#fin
```

Le type entier REEL

Les nombres réels sont traités par le type **REEL**.

```
variable reel maVariable_3
#debut
maVariable_3 VAUT 2.5
#fin
```

Le type booléen BIN

Les booléens sont traités par le type **BIN**. Ce type peut prendre deux valeurs **VRAI** ou **FAUX**. Au contraire des autres mots-clé du langage, les valeurs booléennes **VRAI** et **FAUX** doivent toujours être écrites en majuscules.

```
variable bin maVariable_4
#debut
maVariable_4 vaut VRAI
#fin
```

Le type chaîne de caractères CHAINE

Il permet de coder les chaînes de caractères. Une chaîne de caractères est une suite de caractères entre guillemets, les caractères spéciaux sont ceux du langage C. Exemple : \n est le retour à la ligne.

```
variable chaine maVariable_5
#debut
maVariable_5 vaut "bonjour\n"
#fin
```

2.3 Les instructions

Toute instruction doit se terminer par un retour chariot.

2.4 Les expressions

2.4.1 L'opérateur d'affectation VAUT

L'opérateur d'affectation est le mot-clé **VAUT**.

```
z VAUT w
```

Cela va affecter la valeur *w* à la variable *z*. Les deux opérandes doivent être de même type.

2.4.2 Les opérateurs mathématiques

Les opérateurs unaires

- Le signe **-** inverse la valeur d'une variable ou d'une expression de type **ENTIER**.
- Le signe **++** incrémente d'un pas de un une variable de type **ENTIER**.
- Le signe **--** décrémente d'un pas de un une variable de type **ENTIER**.

Les opérateurs binaires

Les opérations binaires sont définies de la manière suivante :
monNombre **OPERATION** *monAutreNombre*

- L'opérateur **PLUS** permet une addition. Comportement suivant le type des opérandes :


```
operande1 operateur operande2 -> resultat
ENTIER PLUS ENTIER -> ENTIER
ENTIER PLUS REEL -> REEL
REEL PLUS ENTIER -> REEL
REEL PLUS REEL -> REEL
```
- L'opérateur **MOINS** permet une soustraction. Comportement suivant le type des opérandes :


```
operande1 operateur operande2 -> resultat
ENTIER MOINS ENTIER -> ENTIER
ENTIER MOINS REEL -> REEL
REEL MOINS ENTIER -> REEL
REEL MOINS REEL -> REEL
```
- L'opérateur **MULT** permet une multiplication. Comportement suivant le type des opérandes :


```
operande1 operateur operande2 -> resultat
ENTIER MULT ENTIER -> ENTIER
ENTIER MULT REEL -> REEL
REEL MULT ENTIER -> REEL
REEL MULT REEL -> REEL
```

- L'opérateur **DIV** permet une division. Comportement suivant le type des opérandes :


```
operande1 operateur operande2 -> resultat
ENTIER DIV ENTIER -> ENTIER
ENTIER DIV REEL -> REEL
REEL DIV ENTIER -> REEL
REEL DIV REEL -> REEL
```

Les opérateurs de comparaison

- **SUP** supérieur ou égal. Comportement suivant le type des opérandes :


```
operande1 operateur operande2 -> resultat
ENTIER SUP ENTIER -> BIN
ENTIER SUP REEL -> BIN
REEL SUP ENTIER -> BIN
REEL SUP REEL -> BIN
```
- **INF** inférieur ou égal. Comportement suivant le type des opérandes :


```
operande1 operateur operande2 -> resultat
ENTIER INF ENTIER -> BIN
ENTIER INF REEL -> BIN
REEL INF ENTIER -> BIN
REEL INF REEL -> BIN
```
- **EGAL** égal. Comportement suivant le type des opérandes :


```
operande1 operateur operande2 -> resultat
ENTIER EGAL ENTIER -> BIN
ENTIER EGAL REEL -> BIN
REEL EGAL ENTIER -> BIN
REEL EGAL REEL -> BIN
CHAINE EGAL CHAINE -> BIN
BIN EGAL BIN -> BIN
```
- **DIFF** différent. Comportement suivant le type des opérandes :


```
operande1 operateur operande2 -> resultat
ENTIER DIFF ENTIER -> BIN
ENTIER DIFF REEL -> BIN
REEL DIFF ENTIER -> BIN
REEL DIFF REEL -> BIN
CHAINE DIFF CHAINE -> BIN
BIN DIFF BIN -> BIN
```

Les opérateurs logiques

- **ET** et logique, entre deux booléens. Comportement :


```
operande1 operateur operande2 -> resultat
BIN ET BIN -> BIN
```

- **OU** ou logique, entre deux booléens. Comportement :
`operande1 operateur operande2 -> resultat`
`BIN OU BIN -> BIN`
- **NON** non logique, sur un booléen. Comportement :
`operateur operande1 -> resultat`
`NON BIN -> BIN`

NB : la hiérarchie des différents opérateurs présentés ci-dessus correspond à la hiérarchie logique de ces opérateurs dans les mathématiques. Les priorités sont donc les mêmes.

2.5 Les instructions de contrôle

Ces contrôles permettent de modifier le déroulement séquentiel du programme.

2.5.1 Les instruction SI et FINSI

```
SI expression
  $$ instruction
FINSI
```

L'expression doit être une valeur booléenne et doit être sur la même ligne que le mot-clé **SI**.

Si le résultat de l'expression est vrai, l'instruction est exécutée. Dans le cas contraire, l'exécution du programme se poursuit avec l'instruction suivant **FINSI**.

2.5.2 La clause SINON

```
SI expression
  $$ instruction1
SINON
  $$ instruction2
FINSI
```

Si l'expression est évaluée comme étant vraie, l'instruction1 est exécutée, sinon c'est instruction2.

```
variable bin test
#debut
lire test
SI test
  ecrire "ok"
SINON
  ecrire "recommencer"
```

```
FINSI
#fin
```

2.5.3 L'instruction POUR ... JUSQUE ... PAS ... FINPOUR

```
POUR maVariable VAUT init JUSQUE fin PAS lePas
  $$ instructions
FINPOUR
```

Cette instruction exécute un bloc *instructions* tant que *maVariable* est inférieure ou égale à *fin*. Au début, *maVariable* est initialisée à *init*. A chaque boucle, *maVariable* est incrémentée de *lePas*.

```
variable entier i
variable entier j 0
#debut
POUR i vaut 0 JUSQUE 10 PAS 1
  j vaut j plus i
FINPOUR
#fin
```

2.5.4 Les structures imbriquées

Les instructions des structures peuvent contenir d'autres **SI** ou des **POUR ... JUSQUE ... FINPOUR**.

```
variable bin test
variable entier nb
#debut
lire test
SI test
  ecrire "ok"
SINON
lire nb
pour i vaut 0 jusque nb pas 1
  ecrire "bonjour"
finpour
FINSI
#fin
```

2.6 Les instructions d'entrées sorties

2.6.1 L'instruction LIRE

Cette instruction est définie de la manière suivante :
LIRE *varDest*. Cette instruction permet de lire sur l'entrée standard la

valeur d'une variable et l'affecte à la variable *varDest*. Le format de l'entrée doit correspondre au type de *varDest*.

```
variable reel varIn
#debut
lire varIn $$La valeur entrée pourra être par exemple 3.1, mais pas abc.
#fin
```

2.6.2 L'instruction ECRIRE

Cette instruction est définie de la manière suivante :

ECRIRE *unTruc* Cette instruction permet d'écrire *unTruc* sur la sortie standard, *unTruc* pouvant être une variable, une constante ou une chaîne de caractères.

```
variable chaine texte "Bonjour"
constante entier x 25
#debut
ecrire texte
ecrire x
ecrire "On peut aussi ecrire directement du texte.\n"
#fin
```

Chapitre 3

Commentaires

3.1 Analyse lexicale

L'analyseur lexical reconnaît les mots-clés, qu'ils soient écrits en minuscule ou en majuscule, cependant VRAI et FAUX doivent être écrits en majuscule pour être compris comme des valeurs booléennes (vrai et faux seront compris comme des identificateurs). Un nombre réel est un entier, suivi d'un point, suivi d'un entier. Les noms de variables commencent par une lettre et peuvent contenir des lettres, des chiffres ou le caractère underscore `_`. Une chaîne de caractères est nécessairement entre guillemets.

L'analyseur lexical envoie à l'analyseur syntaxique les token correspondants quand il lit un programme écrit en EasyFrog, les commentaires sont reconnus mais ne sont pas traités.

3.2 Analyse syntaxique

3.2.1 Le fichier cible

L'analyseur syntaxique crée 3 fichiers, qui formeront le fichier cible C une fois concaténés :

- `head.c` On y trouve l'entête du fichier : les inclusions de fichiers `.h`, la définition des constantes.
- `var.c` On y trouve la déclaration des variables temporaires pour le code à adresses.
- `fin.c` On y trouve tout le reste du fichier.

Les scripts d'exemples créent ces fichiers dans le répertoire SRC, les concaténent dans SRC/, puis les suppriment. Le fichier résultant est un fichier C compilable.

3.2.2 Les tables de symboles

Nous avons repris le cours de Y. Autret pour gérer les tables des symboles en faisant quelques modifications. Il y a 3 tables des symboles : une pour les constantes, une pour les variables temporaires du code à 3 adresse et une pour les variables du programme. Ces tables contiennent l'identifiant, le type et la valeur des variables (qui n'est utilisée que pour les constantes et les variables initialisées).

C'est à partir des tables des constantes et des variables temporaires que sont écrits les fichiers `head.c` et `var.c`.

Les tables permettent d'effectuer un contrôle sémantique sur le programme, en vérifiant la compatibilité des types pour les opérations, en interdisant de déclarer deux fois la même variable ou encore en vérifiant que les variables ont été déclarées.

3.2.3 Le contrôle de type

Une fonction `test_type(type1, type2)` vérifie la compatibilité de deux types et renvoie le type "dominant" (entier et reel sont compatibles, c'est alors reel qui est renvoyé) ou si les types sont incompatibles, le programme s'arrête sur une erreur.

3.2.4 Structures de contrôle

Les structures de contrôle sont traduites par des structures *C if* et des branchements. Les structures pouvant être imbriquées, il faut maintenir le niveau courant d'imbrication et le prochaine étiquette disponible.

Nous avons séparés les étiquettes des structures *si* et celles des structures *pour*. La variable `etiqs` permet de numéroter les étiquettes des structures *si* sans risque de doublons, le pointeur `setiq` permet de retrouver le niveau courant d'imbrication des structures *si*.

La variable `etiqp` et le pointeur `petiq` jouent respectivement le même rôle pour les structures *pour*.

Chapitre 4

Conclusion

Ce projet était très intéressant à réaliser, mais par manque de temps, l'implémentation du langage n'a pas été complètement terminée. Par exemple, la structure *pour* ne fonctionne pas comme cela a été spécifié dans le langage : le pas et la valeur finale doivent être des entiers.

L'analyse lexicale et l'analyse syntaxique ont été testées et fonctionnent, mais l'analyse sémantique n'a pas été testée comme nous l'aurions voulu (des incohérences subsistent certainement).

Annexe A

Annexes

A.1 Analyseur lexical

A.1.1 Analyseur lexical sans instructions C

```
%{
%}
chiffre      [0-9]
lettre       [a-z]
reel         [+]?[0-9]+[.][0-9]+
entier       [+]?[0-9]+
chaine       ["].*["]
bool         (VRAI|FAUX)
comment      [$$].*[\n]
ident        [a-zA-Z_]+[a-zA-Z0-9_]*
retour       [\n]
%%
[lL] [iI] [rR] [eE]
[eE] [cC] [rR] [iI] [rR] [eE]
[sS] [iI]
[pP] [aA] [sS]
[fF] [iI] [nN] [sS] [iI]
[sS] [iI] [nN] [oO] [nN]
[pP] [oO] [uU] [rR]
[jJ] [uU] [sS] [qQ] [uU] [eE]
[fF] [iI] [nN] [pP] [oO] [uU] [rR]
[cC] [oO] [nN] [sS] [tT] [aA] [nN] [tT] [eE]
[vV] [aA] [rR] [iI] [aA] [bB] [lL] [eE]
[eE] [nN] [tT] [iI] [eE] [rR]
[rR] [eE] [eE] [lL]
[bB] [iI] [nN]
[cC] [hH] [aA] [iI] [nN] [eE]
```

```

[vV] [aA] [uU] [tT]
[eE] [tT]
[oO] [uU]
[nN] [oO] [nN]
[sS] [uU] [pP]
[iI] [nN] [fF]
[eE] [gG] [aA] [lL]
[dD] [iI] [fF] [fF]
[pP] [lL] [uU] [sS]
[mM] [uU] [lL] [tT]
[dD] [iI] [vV]
[mM] [oO] [iI] [nN] [sS]
"#debut"
"#fin" [\n]*
"--"
"++"
{retour}
{reel}
{entier}
{bool}
{ident}
{chaine}
{comment}
"("
")"
.
%%

```

A.1.2 Analyseur lexical avec instructions C

```

%{
#include <stdlib.h>
%}
chiffre      [0-9]
lettre       [a-z]
reel         [+]?[0-9]+[.][0-9]+
entier       [+]?[0-9]+
chaine       ["].*["]
bool         (VRAI|FAUX)
comment      [$$].*[\n]
ident        [a-zA-Z_]+[a-zA-Z0-9_]*
retour       [\n]
%%
[lL] [iI] [rR] [eE]                {return ef_lire;}

```

```

[eE] [cC] [rR] [iI] [rR] [eE]      {return ef_ecrire;}
[sS] [iI]                          {return ef_si;}
[pP] [aA] [sS]                      {return ef_pas;}
[fF] [iI] [nN] [sS] [iI]          {return ef_finsi;}
[sS] [iI] [nN] [oO] [nN]          {return ef_sinon;}
[pP] [oO] [uU] [rR]                {return ef_pour;}
[jJ] [uU] [sS] [qQ] [uU] [eE]      {return ef_jusque;}
[fF] [iI] [nN] [pP] [oO] [uU] [rR] {return ef_finpour;}
[cC] [oO] [nN] [sS] [tT] [aA] [nN] [tT] [eE] {return ef_const;}
[vV] [aA] [rR] [iI] [aA] [bB] [lL] [eE] {return ef_var;}
[eE] [nN] [tT] [iI] [eE] [rR]      {return ef_int;}
[rR] [eE] [eE] [lL]                {return ef_real;}
[bB] [iI] [nN]                     {return ef_bool;}
[cC] [hH] [aA] [iI] [nN] [eE]      {return ef_string;}
[vV] [aA] [uU] [tT]                {return ef_affect;}
[eE] [tT]                          {return ef_et;}
[oO] [uU]                          {return ef_ou;}
[nN] [oO] [nN]                     {return ef_non;}
[sS] [uU] [pP]                     {return ef_sup;}
[iI] [nN] [fF]                     {return ef_inf;}
[eE] [gG] [aA] [lL]                {return ef_egal;}
[dD] [iI] [fF] [fF]                {return ef_diff;}
[pP] [lL] [uU] [sS]                {return ef_add;}
[mM] [uU] [lL] [tT]                {return ef_mult;}
[dD] [iI] [vV]                     {return ef_div;}
[mM] [oO] [iI] [nN] [sS]           {return ef_minus;}
"#debut"                            {return ef_debut;}
"#fin" [\n]*                         {return ef_fin;}
"--"                                 {return ef_aff_moins;}
"++"                                 {return ef_aff_plus;}
{retour}                             {noLigne++; return ef_back;}
{reel}   {strcpy(yylval.chaine, yytext); return ef_reel;}
{entier} {strcpy(yylval.chaine, yytext); return ef_entier;}
{bool}   {strcpy(yylval.chaine, yytext); return ef_booleen;}
{ident}  {strcpy(yylval.chaine, yytext); return ef_ident;}
{chaine} {strcpy(yylval.chaine, yytext); return ef_chaine;}
{comment} {};
 "("     {return ef_par_open;}
 ")"     {return ef_par_closed;}
 "."     {}
 %%

```

A.2 Analyseur syntaxique

A.2.1 Analyseur syntaxique sans instructions C

```
%{  
%union {  
    int entier;  
    double reel;  
    char chaine[80];  
    struct expr exp;  
}  
  
%token ef_pas  
%token ef_lire  
%token ef_ecrire  
%token ef_si  
%token ef_finsi  
%token ef_sinon  
%token ef_pour  
%token ef_jusque  
%token ef_finpour  
%token ef_const  
%token ef_var  
%token ef_int  
%token ef_real  
%token ef_bool  
%token ef_string  
%token ef_affect  
%token ef_et  
%token ef_ou  
%token ef_non  
%token ef_sup  
%token ef_inf  
%token ef_egal  
%token ef_diff  
%token ef_add  
%token ef_mult  
%token ef_div  
%token ef_minus  
%token ef_debut  
%token ef_fin  
%token ef_aff_moins  
%token ef_aff_plus  
%token ef_back
```

```
%token <chaine> ef_reel
%token <chaine> ef_entier
%token <chaine> ef_booleen
%token <chaine> ef_ident
%token <chaine> ef_chaine
%token ef_par_open
%token ef_par_closed

%type <exp> F P
%type <exp> opt_VINIT
%type <exp> B
%type <exp> E
%type <exp> C
%type <exp> T CINIT

%%
//S represente le programme en entier
S : MESDECLs ef_debut Ps ef_fin
  ;
//DECL est la partie déclaration et initialisation des variables

MESDECLs : DECLs ef_back
  |
  ;

DECLs : DECLs ef_back DECL
  | DECL
  ;

DECL : V
  ;

//Déclaration et initialisation des variables
V : ef_var ef_int ef_ident
  | ef_var ef_real ef_ident
  | ef_var ef_bool ef_ident
  | ef_var ef_string ef_ident
  | ef_const CINIT
  ;

opt_VINIT : ef_entier
  | ef_reel
  | ef_booleen
  | ef_chaine
```

```
|  
;  
  
CINIT: ef_int ef_ident ef_entier  
| ef_real ef_ident ef_reel  
| ef_bool ef_ident ef_booleen  
| ef_string ef_ident ef_chaine  
;  
  
//Ps est le corps du programme  
Ps : Ps ef_back P  
| P  
;  
  
P : BLOC  
| B  
| ef_ident ef_affect B  
| L  
|  
;  
  
BLOC : ef_si B ef_back Ps ef_back opt_blocsi ef_finsi  
| ef_pour P ef_jusque ef_entier ef_pas ef_entier Ps ef_finpour  
;  
  
opt_blocsi : ef_sinon Ps ef_back  
|  
;  
  
//Addition ou soustraction  
E : E ef_add T  
| E ef_minus T  
| T  
;  
  
//Multiplication ou division  
T : T ef_mult F  
| T ef_div F  
| F  
;  
  
B : B ef_et C  
| B ef_ou C  
| ef_non C  
| C  
;
```

```
//Expressions booléennes
C : F ef_sup F
  | F ef_inf F
  | F ef_egal F
  | F ef_diff F
  | E
  ;
//Expression de base
F : ef_par_open E ef_par_closed
  | ef_reel
  | ef_entier
  | ef_chaine
  | ef_ident opt_inc
  | ef_booleen
  ;
```

```
opt_inc : ef_aff_plus
  | ef_aff_moins
  |
  ;
```

```
L: ef_lire ef_ident
  | ef_ecrire F
  ;
%%
```

A.2.2 Analyseur syntaxique avec instructions C

```
%{
#include <stdio.h>
#include <stdlib.h>

  /*----- DEFINES----- */
#define TYPEBOOL 0
#define TYPEENTIER 1
#define TYPEREEL 2
#define TYPECHAINE 3

#define MAX_IMBRIQ 16
#define MAXTABLE 100 /* nombre maximal de symboles
dans la table des symboles */

  /*-----Differentes structures -----*/
```

```
/*un element de la table*/
typedef struct attribut_s
{
    int type;
    char * code;
    char * val;
} attribut;

/*la table des symboles*/
struct table {
    int nombre_elements;
    attribut tsym[MAXTABLE];
};

/*----- Fonctions----- */

/*
test_type
Compare deux type et renvoi ce type si c'est le meme,
sinon arrete le programme avec mess d'erreur
*/

int test_type (int t1, int t2)
{
    if (t1!=t2)
{
    if ((t1==TYPEREEL) && (t2==TYPEENTIER))
        return(t1);
    else
        if ((t1==TYPEENTIER)&&(t2==TYPEREEL))
            return (t2);
        else
            {printf("\n***Probleme de type ici***\n,%d,%d",t1,t2);exit(0);}
    }
    else return(t1);}

/* Partie de gestion d'une table des symboles, cours de Licence */

/*
tableErreur
Signale les erreurs et arrete le programme
*/
```

```
*/

int tableErreur(char *s)
{
    printf("\n\n");
    printf("!!! erreur dans la gestion d'une table de symboles\n");
    printf("!!! %s",s);
    printf("\n\n");
    exit(0);
}

/*
tableCreer
Allocation d'une table de symboles
*/

struct table *tableCreer()
{
    struct table *t;
    t = (struct table *)malloc(sizeof(struct table));
    t->nombre_elements = 0;
    return(t);
}

/*
tableTaille
Retourne le nombre d'elements de la table
*/

int tableTaille(struct table *t)
{
    return(t->nombre_elements);
}

/*
tableAjouterElement
Empile un nouveau symbole dans la table
Remarque
Verifie si le symbole existe deja.
*/

void tableAjouterElement(struct table *t, char * code, int type, char * val)
{
    if (t->nombre_elements >= MAXTABLE)
```

```

    {
        tableErreur("table des symbole pleine");
    }
else
    {
        if ((type > 4)|| (type <0))
            {
                tableErreur("Type inconnu");
            }
        else
            {
if (tableChercherType(t,code) != -1)
            {
                tableErreur("Variable deja declaree");
                printf("nom de la variable : %s\n",code);
            }
else
            {
                t->tsym[t->nombre_elements].code=(char *) (malloc(sizeof(code)));
                strcpy(t->tsym[t->nombre_elements].code,code);
                t->tsym[t->nombre_elements].val=(char *) (malloc(sizeof(val)));
                strcpy(t->tsym[t->nombre_elements].val,val);
                t->tsym[t->nombre_elements].type = type;
                t->nombre_elements++;
            }
            }
    }
}

/*
tableChercherType
Recherche un symbole dans la table.
Retourne le type de la variable si la recherche est un succes.
Retourne -1 si la variable n'est pas trouvee.
*/

int tableChercherType(struct table *t, char *code)
{
    int i;
    for (i=0 ; i<t->nombre_elements ; i++) {
        if (strcmp(t->tsym[i].code,code) == 0) return(t->tsym[i].type);
    }
    return(-1);
}

```

```
/*
tableAfficher
Affiche une table de symbole
*/

int tableAfficher(struct table *t)
{
    int i;
    char *s;
    int val;
    printf("affichage d'une table de symboles\n");
    for (i=0 ; i<tableTaille(t) ; i++) {
        printf("element %d :
(%s:%d:%s)\n",i,t->tsym[i].code,t->tsym[i].type,t->tsym[i].val);
    }
    printf("\n");
}

int defAfficher(struct table *t,FILE * f)
{
    int i;
    for (i=0;i<tableTaille(t);i++)
        fprintf(f,"#define %s %s\n",t->tsym[i].code,t->tsym[i].val);
    fprintf(f,"\nint main (void)\n{\n");
}

int varAfficher(struct table *t,FILE * f)
{
    int i;
    fprintf(f,"/* variables temporaires pour le code a 3 adresses */\n");
    for (i=0;i<tableTaille(t);i++)
    {
        switch (t->tsym[i].type)
        {
        case 0 : fprintf(f,"int ");break;
            case 1 : fprintf(f,"int ");break;
            case 2 : fprintf(f,"double ");break;
            case 3 : fprintf(f,"char * ");break;
        }
        fprintf(f,"%s;\n",t->tsym[i].code);
    }
    fclose(f);
}
```

```

}

/*
inittab initialise un tableau de booleen
*/
int * inittab ()
{
    int i;
    int * t;
    t= (int *) malloc (sizeof(int)*MAX_IMBRIQ);
    for (i=1;i<MAX_IMBRIQ-1;i++)
        t[i]=0;
    t[0]=1;
    return t;
}

/*
maxtab renvoi l'index le plus grand qui contient "vrai"
*/
int maxtab ( int* tab)
{
    int i=MAX_IMBRIQ-1;
    while (tab[i]==0)
        i--;
    return i;
}

/*----- Declaration----- */

int etiqs=0;
int etiqp=0;
int * setiq;
int * petiq;

char tmp[5] ="temp0";
FILE * fTemp;
struct table * tds;
struct table * temp;
struct table * tdc;

struct expr {
    int type;
    char val[80];
} exp;
%}

```

```
%union {
    int entier;
    double reel;
    char chaine[80];
    struct expr exp;
}

%token ef_pas
%token ef_lire
%token ef_ecrire
%token ef_si
%token ef_finsi
%token ef_sinon
%token ef_pour
%token ef_jusque
%token ef_finpour
%token ef_const
%token ef_var
%token ef_int
%token ef_real
%token ef_bool
%token ef_string
%token ef_affect
%token ef_et
%token ef_ou
%token ef_non
%token ef_sup
%token ef_inf
%token ef_egal
%token ef_diff
%token ef_add
%token ef_mult
%token ef_div
%token ef_minus
%token ef_debut
%token ef_fin
%token ef_aff_moins
%token ef_aff_plus
%token ef_back
%token <chaine> ef_reel
%token <chaine> ef_entier
%token <chaine> ef_booleen
%token <chaine> ef_ident
%token <chaine> ef_chaine
```

```

%token ef_par_open
%token ef_par_closed

%type <exp> F P
%type <exp> opt_VINIT
%type <exp> B
%type <exp> E
%type <exp> C
%type <exp> T CINIT

%%
//S represente le programme en entier
S : {temp=tableCreer();tds=tableCreer();tdc=tableCreer();
setiq=inittab();petiq=inittab();fTemp=fopen("SRC/var.c","w");}
MESDECLs ef_debut { printf("//debut du programme\n");} Ps
{ printf("//fin du programme\n\n");}ef_fin {defAfficher(tdc,fTemp);
varAfficher(temp,fTemp);}
;
//DECL est la partie déclaration et initialisation des variables

MESDECLs : {printf("//Déclaration des variables du programme\n");}DECLs ef_back{}
|{}
;

DECLs : DECLs ef_back DECL{}
| DECL{}
;

DECL : V {}
;

//Déclaration et initialisation des variables
V : ef_var ef_int ef_ident {printf("int %s",$3);} opt_VINIT
{tableAjouterElement(tds,$3,TYPEENTIER,$5.val);}
| ef_var ef_real ef_ident {printf("double %s",$3);} opt_VINIT
{tableAjouterElement(tds,$3,TYPEEREEL,$5.val);}
| ef_var ef_bool ef_ident {printf("int %s",$3);} opt_VINIT
{tableAjouterElement(tds,$3,TYPEBOOL,$5.val);}
| ef_var ef_string ef_ident {printf("char * %s",$3);} opt_VINIT
{;tableAjouterElement(tds,$3,TYPECHAINE,$5.val);}
| ef_const CINIT {}
;

opt_VINIT : ef_entier {strcpy($$.val,$1);

```

```

    $$ .type=TYPEENTIER;printf(" = %s;\n",$1);}
| ef_reel {strcpy($$.val,$1);
    $$ .type=TYPEREEL;printf(" = %s;\n",$1);}
| ef_booleen {strcpy($$.val,$1);
    $$ .type=TYPEENTIER;printf(" = %s;\n",$1);}
| ef_chaine {strcpy($$.val,$1);
    $$ .type=TYPECHAINE;printf(" = %s;\n",$1);}
| {printf(";\n");}
;

CINIT: ef_int ef_ident ef_entier {tableAjouterElement(tdc,$2,TYPEENTIER,$3);
tableAjouterElement(tds,$2,TYPEENTIER,$3);}
| ef_real ef_ident ef_reel {tableAjouterElement(tdc,$2,TYPEREEL,$3);
tableAjouterElement(tds,$2,TYPEREEL,$3);}
| ef_bool ef_ident ef_booleen {tableAjouterElement(tdc,$2,TYPEBOOL,$3);
tableAjouterElement(tds,$2,TYPEBOOL,$3);}
| ef_string ef_ident ef_chaine {tableAjouterElement(tdc,$2,TYPECHAINE,$3);
tableAjouterElement(tds,$2,TYPECHAINE,$3);}
;

//Ps est le corps du programme
Ps : Ps ef_back P{}
| P{}
;

P : BLOC {}
| B{}
| ef_ident ef_affect B {$$.type=test_type(tableChercherType(tds,$1),$3.type);
printf("%s = %s;\n",$1,$3.val);strcpy($$.val,$1);}
| L {}
| {}
;

BLOC : ef_si B ef_back {etiqs++;*(++setiq)=etiqs;printf("if (! %s )\n
{ \ngoto sinon%d; \n }\n",$2.val,*setiq);} Ps ef_back
{printf("\ngoto finsi%d;\nsinon%d: \n",*setiq,*setiq);} opt_blocsi ef_finsi
{printf("finsi%d: \n",*setiq);setiq--;}
| ef_pour P ef_jusque ef_entier ef_pas ef_entier {etiqp++;*(++petiq)=etiqp;
printf("pour%d: \nif (%s > %s)\ngoto finpour%d;\n",*petiq,$2.val,$4,*petiq);} Ps
ef_finpour{printf("%s+=%s;\ngoto pour%d;\nfinpour%d:\n",$2.val,$6,*petiq,*petiq);
petiq--;}
;

opt_blocsi : ef_sinon Ps ef_back

```

```

    | {}
    ;
//Addition ou soustraction
E : E ef_add T {tableAjouterElement(temp,tmp,test_type($1.type,$3.type),"0");
                printf("%s=(%s + %s);\n",tmp,$1.val,$3.val);
$$ .type=test_type($1.type,$3.type);
strcpy($$.val,tmp); tmp[4]++;}
    | E ef_minus T {tableAjouterElement(temp,tmp,test_type($1.type,$3.type),"0");
                printf("%s=(%s - %s);\n",tmp,$1.val,$3.val);
$$ .type=test_type($1.type,$3.type);
strcpy($$.val,tmp); tmp[4]++;}
    | T {$$=$1;}
    ;
//Multiplication ou division
T : T ef_mult F {tableAjouterElement(temp,tmp,test_type($1.type,$3.type),"0");
                printf("%s=(%s * %s);\n",tmp,$1.val,$3.val);
$$ .type=test_type($1.type,$3.type);
strcpy($$.val,tmp); tmp[4]++;}
    | T ef_div F {tableAjouterElement(temp,tmp,test_type($1.type,$3.type),"0");
                printf("%s=(%s / %s);\n",tmp,$1.val,$3.val);
$$ .type=test_type($1.type,$3.type);
strcpy($$.val,tmp); tmp[4]++;}
    | F {}
    ;

B : B ef_et C {tableAjouterElement(temp,tmp,TYPEBOOL,"0");
                test_type($1.type,$3.type);
printf("%s=(%s&&%s);\n",tmp,$1.val,$3.val);
$$ .type=TYPEBOOL;
strcpy($$.val,tmp); tmp[4]++;}
    | B ef_ou C {tableAjouterElement(temp,tmp,TYPEBOOL,"0");
                test_type($1.type,$3.type);
printf("%s=(%s||%s);\n",tmp,$1.val,$3.val);
$$ .type=TYPEBOOL;
strcpy($$.val,tmp); tmp[4]++;}
    | ef_non C {tableAjouterElement(temp,tmp,TYPEBOOL,"0");
                printf("%s=!%s;\n",tmp,$2.val);
                $$ .type=TYPEBOOL;
                strcpy($$.val,tmp); tmp[4]++;}
    | C {$$=$1;}
    ;
//Expressions booleennes
C : F ef_sup F {tableAjouterElement(temp,tmp,TYPEBOOL,"0");
                test_type($1.type,$3.type);

```

```

printf("%s=(%s>%s);\n",tmp,$1.val,$3.val);
$$ .type=TYPEBOOL;
strcpy($$.val,tmp); tmp[4]++;}
| F ef_inf F {tableAjouterElement(temp,tmp,TYPEBOOL,"0");
               test_type($1.type,$3.type);
printf("%s=(%s<%s);\n",tmp,$1.val,$3.val);
$$ .type=TYPEBOOL;
strcpy($$.val,tmp); tmp[4]++;}
| F ef_egal F {tableAjouterElement(temp,tmp,TYPEBOOL,"0");
               if (test_type($1.type,$3.type)==3)
printf("%s = !strcmp(%s,%s);\n",tmp,$1.val,$3.val);
               else
printf("%s=(%s==%s);\n",tmp,$1.val,$3.val);
$$ .type=TYPEBOOL;
strcpy($$.val,tmp); tmp[4]++;}
| F ef_diff F {tableAjouterElement(temp,tmp,TYPEBOOL,"0");
               if (test_type($1.type,$3.type)==3)
printf("%s = strcmp(%s,%s);\n",tmp,$1.val,$3.val);
else
printf("%s=(%s!=%s);\n",tmp,$1.val,$3.val);
$$ .type=TYPEBOOL;
strcpy($$.val,tmp); tmp[4]++;}
| E {}
;
//Expression de base
F : ef_par_open E ef_par_closed {$$=$2;}
| ef_reel {strcpy($$.val,$1);
          $$ .type=TYPEREEL;}
| ef_entier {strcpy($$.val,$1);
            $$ .type=TYPEENTIER;}
| ef_chaine {strcpy($$.val,$1);
            $$ .type=TYPECHAINE;}
| ef_ident opt_inc {strcpy($$.val,$1);
                  $$ .type=tableChercherType(tds,$1);}
| ef_booleen {strcpy($$.val,$1);
              $$ .type=TYPEBOOL;}
;

opt_inc : ef_aff_plus{printf("++ ");}
| ef_aff_moins{printf("-- ");}
|{}
;

L: ef_lire ef_ident{switch (tableChercherType(tds,$2))

```

```

        {
            case 0 :
printf("scanf(\"%s\",&%s);\n", "%d", $2);break;
            case 1 :
printf("scanf(\"%s\",&%s);\n", "%d", $2);break;
            case 2 :
printf("scanf(\"%s\",&%s);\n", "%lf", $2);break;
            case 3 :
printf("scanf(\"%s\",%s);\n", "%s", $2);break;
            case -1 :
        {}
    }
}
| ef_ecrire F {switch ($2.type)
    {
        case 0 :
            printf("printf(\"%s\",%s);\n", "%d", $2.val);break;
        case 1 :
            printf("printf(\"%s\",%s);\n", "%d", $2.val);break;
        case 2 :
            printf("printf(\"%s\",%s);\n", "%f", $2.val);break;
        case 3 :
            printf("printf(\"%s\",%s);\n", "%s", $2.val);break;
    }
}
;

%%
#include "lex.yy.c"
#include <stdio.h>
main ()
{FILE * entete;
entete = fopen("SRC/head.c","w");
fprintf(entete , "#include <stdio.h>\n\n#define VRAI 1\n#define FAUX 0\n");
yyparse();
fclose(entete);
}

```

A.3 Exemples

A.3.1 Exemple 1

Fichier frachi.ef

```
variable chaine chinois "vous etes chinois"
variable chaine francais "vous etes francais"
variable chaine ni "vous n'etes ni francais ni chinois"
variable bin chi
variable bin fra
variable entier chitot 0
variable entier fratot 0
variable entier autre 0
#debut
ecrire chinois
lire chi
si chi egal VRAI
chitot vaut chitot plus 1
sinon
ecrire francais
lire fra
si fra egal VRAI
fratot vaut fratot plus 1
sinon
ecrire ni
autre vaut autre plus 1
finsi
finsi
ecrire "\nnombre total de chinois : "
ecrire chitot
ecrire "\nnombre total de francais : "
ecrire fratot
ecrire "\nnombre total de autre : "
ecrire autre
#fin
```

Fichier frachi.c

```
#include <stdio.h>

#define VRAI 1
#define FAUX 0

int main (void)
```

```
{
/* variables temporaires pour le code a 3 adresses */
int temp0;
int temp1;
int temp2;
int temp3;
int temp4;
//Déclaration des variables du programme
char * chinois = "vous etes chinois";
char * francais = "vous etes francais";
char * ni = "vous n'etes ni francais ni chinois";
int chi;
int fra;
int chitot = 0;
int fratot = 0;
int autre = 0;
//debut du programme
printf("%s",chinois);
scanf("%d",&chi);
temp0=(chi==VRAI);
if (! temp0 )
{
goto sinon1;
}
temp1=(chitot + 1);
chitot = temp1;

goto finsi1;
sinon1:
printf("%s",francais);
scanf("%d",&fra);
temp2=(fra==VRAI);
if (! temp2 )
{
goto sinon2;
}
temp3=(fratot + 1);
fratot = temp3;

goto finsi2;
sinon2:
printf("%s",ni);
temp4=(autre + 1);
autre = temp4;
```

```
finsi2:
finsi1:
printf("%s","\nnombre total de chinois :");
printf("%d",chitot);
printf("%s","\nnombre total de francais :");
printf("%d",fratot);
printf("%s","\nnombre total de autre :");
printf("%d",autre);
//fin du programme
}
```

Script fraci.csh

```
BIN/proj < TEST/frachi.ef > SRC/fin.c
cat SRC/head.c SRC/var.c SRC/fin.c > SRC/frachi.c
rm SRC/head.c SRC/var.c SRC/fin.c
gcc -o BIN/frachi SRC/frachi.c
BIN/frachi
```

A.3.2 Exemple 2

Fichier hello.ef

```
#debut
$$ affiche bonjour
ecrire "bonjour!\n"
#fin
```

Fichier hello.c

```
#include <stdio.h>

#define VRAI 1
#define FAUX 0

int main (void)
{
/* variables temporaires pour le code a 3 adresses */
//debut du programme
printf("%s","bonjour!\n");
//fin du programme
}
```

Script hello.csh

```
BIN/proj < TEST/hello.ef > SRC/fin.c
```

```
cat SRC/head.c SRC/var.c SRC/fin.c > SRC/hello.c
rm SRC/head.c SRC/var.c SRC/fin.c
gcc -o BIN/hello SRC/hello.c
BIN/hello
```

A.3.3 Exemple 3

Fichier tataille.ef

```
variable chaine tataille "combien mesures-tu?"
variable reel taille
variable reel tailletot 0
variable entier nbeleve 10
variable entier t
variable reel taille moy
#debut
pour t vaut 1 jusque 10 pas 1
  ecrire tataille
  lire taille
  tailletot vaut tailletot plus taille
finpour
taille moy vaut tailletot div nbeleve
ecrire "la taille moyenne des élèves de la classe est :"
ecrire taille moy
#fin
```

Fichier tataille.c

```
#include <stdio.h>

#define VRAI 1
#define FAUX 0

int main (void)
{
  /* variables temporaires pour le code a 3 adresses */
  double temp0;
  double temp1;
  //Déclaration des variables du programme
  char * tataille = "combien mesures-tu?";
  double taille;
  double tailletot = 0;
  int nbeleve = 10;
  int t;
  double taille moy;
```

```
//debut du programme
t = 1;
pour1:
if (t > 10)
goto finpour1;
printf("%s",tataille);
scanf("%lf",&taille);
temp0=(tailletot + taille);
tailletot = temp0;
t+=1;
goto pour1;
finpour1:
temp1=(tailletot / nbeleve);
taillemoy = temp1;
printf("%s","la taille moyenne des élèves de la classe est :");
printf("%f",taillemoy);
//fin du programme
}
```

Script tataille.csh

```
BIN/proj < TEST/tataille.ef > SRC/fin.c
cat SRC/head.c SRC/var.c SRC/fin.c > SRC/tataille.c
rm SRC/head.c SRC/var.c SRC/fin.c
gcc -o BIN/tataille SRC/tataille.c
BIN/tataille
```