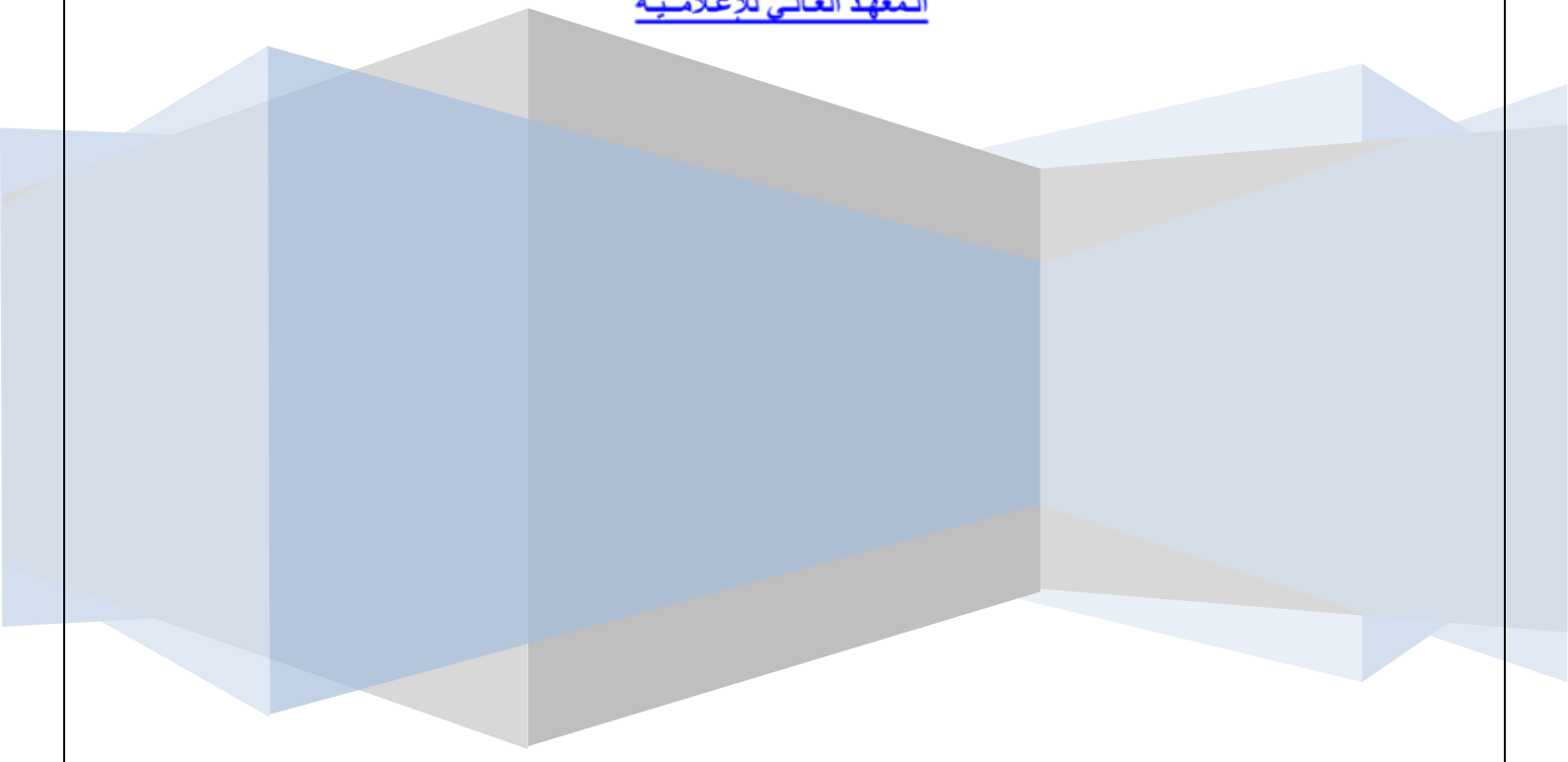
Compte rendu du mini-projet (Techniques de compilation)

Réalisé par : Mustapha Sahli

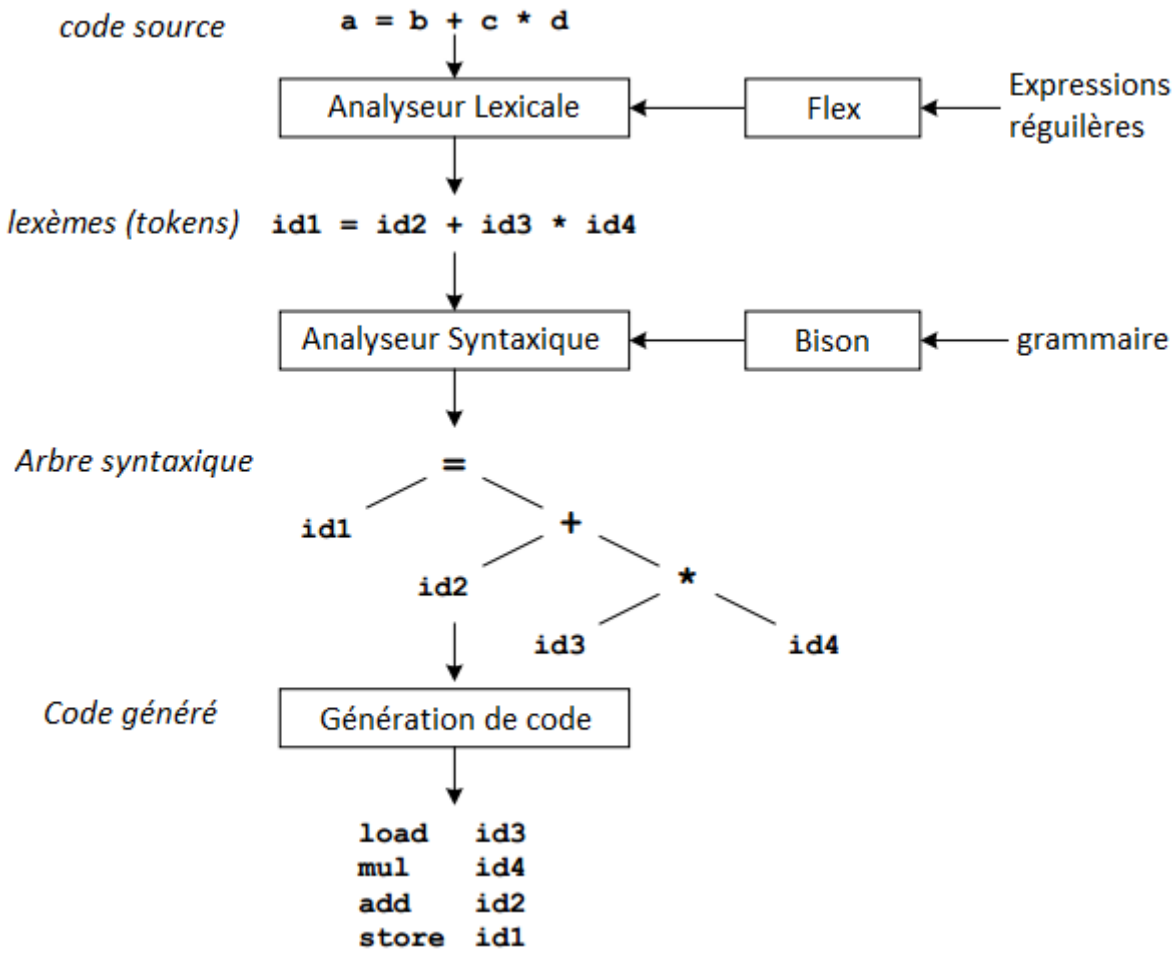
1ING4 – sous-groupe B

INSTITUT SUPERIEUR INFORMATIQUE ISI
المعهد العالي للإعلامية

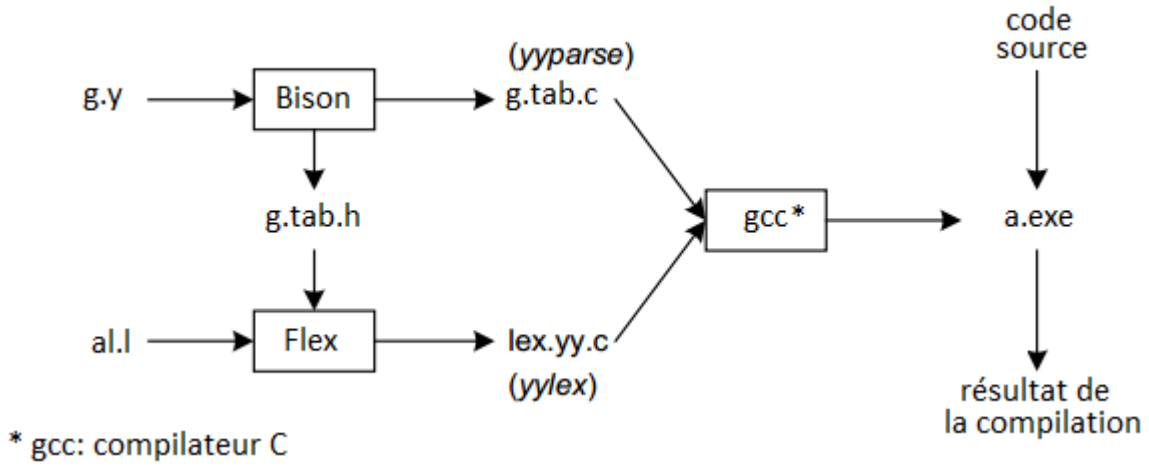


Introduction

L'intérêt de ce mini-projet est de créer une version simplifié d'un compilateur du langage Pascal.
Pour cela, on est amené à créer un analyseur lexicale : pour reconnaître les différents lexèmes (*tokens*) du code source et un analyseur syntaxique : afin de vérifier l'ordre des lexèmes reconnues dans le code.
Les outils utilisés sont : *flex* pour la génération de l'analyseur lexicale, *bison* pour la génération de l'analyseur syntaxique et *code::blocks* pour compiler les fichiers C générés par *flex* et *bison* et pour traiter la phase de l'analyse sémantique.



Flex doit obtenir un fichier (.l) pour générer un analyseur lexicale, ce fichier contient les expressions régulières qui définissent les différents lexèmes du langage, *bison* doit obtenir un fichier (.y) qui contient la description de la grammaire du langage.



Analyse lexicale

Le fichier (.l) qui sera par la suite traité par *flex* a le format suivant :

... définitions ...
%%
... définitions et expressions régulières ...
%%
... fonctions en C ...

Définitions : contient la définition des variables et/ou fonctions Globales à utiliser.
Définitions et expressions régulières : cette partie regroupe L'ensemble des définitions et expressions régulières qui vont Implémenter la stratégie de détection des lexèmes dans le code source
Fonctions en C : cette partie contient les fonctions nécessaires pour le bon fonctionnement de l'analyseur lexicale, notamment, les fonctions *yylex()* qui déclenche l'analyse lexicale et *yywrap()* qui retourne un entier 1 pour terminer l'analyse 0 pour indiquer que l'analyse n'est pas encore terminée.

L'analyseur lexical doit reconnaitre les mots clés (*begin, do, else, end, function, if, int, procedure, program, then, var, while*), les identificateurs (*id*), les nombres (*nb*), les opérateurs relationnels (*oprel*), les opérateurs arithmétiques (*opadd*, *opmul*) et l'opérateur d'affectation (*opaffect*).

Dans un premier temps, on va juste afficher que le lexème en question est reconnu, après on doit ajouter permettre l'analyseur lexicale de retourner les unités lexicales reconnues pour être ensuite utilisées par l'analyseur syntaxique.

```
    /* Définitions */

%{

    /* Code C */
int outputLexical = 1;  /* flag qui permet d'afficher(1)/masquer(0) les messages de l'analyseur lexicale, par défaut (1) */
    /* fonction qui affiche un message passé en paramétre */
void output(const char* msg) {
    if(outputLexical == 1) {
        printf("Analyseur lexicale: %s\n", msg);
    }
}
    /* chaine de caractéres utilisé avec la fonction output() */
char buffer[50];

%}

lettre  [a-zA-Z]
chiffre [0-9]
id      {lettre}({lettre}|{chiffre})*
nb      {chiffre}+

blanc   [ \t\n]+

%%
```

Partie des définitions du fichier pascal.l

L'entier *outputLexical* est définie pour afficher/masquer les messages de l'analyseur lexical, la fonction *output()* permet d'afficher un message passé en paramètre si la variable *outputLexical* a la valeur 1, la chaine *buffer* est utilisée pour afficher un message plus détaillée, elle est utilisée avec la fonction *output().*

```
%%

    /* Définitions et expressions régulières */

    /*
        lexèmes à reconnaître:
            mots clés:
                begin
                do
                else
                end
                function
                if
                int
                procedure
                program
                then
                var
                while
    */

[bB][eE][gG][iI][nN]                { output("mot cle: BEGIN");     }
[dD][oO]                            { output("mot cle: DO");        }
[eE][lL][sS][eE]                    { output("mot cle: ELSE");      }
[eE][nN][dD]                        { output("mot cle: END");       }
[fF][uU][nN][cC][tT][iI][oO][nN]    { output("mot cle: FUNCTION");  }
[iI][fF]                            { output("mot cle: IF");        }
[iI][nN][tT]                        { output("mot cle: INT");       }
[pP][rR][oO][cC][eE][dD][uU][rR][eE] { output("mot cle: PROCEDURE"); }
[pP][rR][oO][gG][rR][aA][mM]        { output("mot cle: PROGRAM");   }
[tT][hH][eE][nN]                    { output("mot cle: THEN");      }
[vV][aA][rR]                        { output("mot cle: VAR");       }
[wW][hH][iI][lL][eE]                { output("mot cle: WHILE");     }
```

Partie Définitions et expressions régulières du fichier pascal.l (1/3)

Chaque caractère doit être traite dans ces deux variantes (en majuscule et en minuscule) d'où l'utilisation de *[bB][eE][gG][iI][nN]* au lieu de *begin.* Entre autres, pour accepter toutes les variantes du mot clé en question : *begin, Begin, BEGIN, …*

```
    /*
            nb          (nombres: chiffre chiffre*)
    */
{nb}                                    {
                                            sprintf(buffer, "NB: %s (%d caractere(s))", yytext, yyleng);
                                            output(buffer);
                                        }
    /*
            id          (identificateurs)
    */
{id}                                    {
                                            sprintf(buffer, "ID: %s (%d caractere(s))", yytext, yyleng);
                                            output(buffer);
                                        }
    /*
            oprel       (opérateurs relationnels: == <> < > <= >= not)
    */
==|<>|<|>|<=|>=|not                     {
                                            sprintf(buffer, "OPREL: %s (%d caractere(s))", yytext, yyleng);
                                            output(buffer);
                                        }
    /*
            opadd       (+ - or)
    */
\+|-|or                                 {
                                            sprintf(buffer, "OPADD: %s (%d caractere(s))", yytext, yyleng);
                                            output(buffer);
                                        }
```

Partie Définitions et expressions régulières du fichier pascal.l (2/3)

NB : pour les opérateurs d'additions (opadd) on a mis \+ au lieu de + puisque + a un sens sous *flex,* on doit soit ajouter un antislash avant le caractère (comme on a choisi de le faire), soit on entoure le caractère en question par des guillemets *« + »* sinon une erreur du type (*unrecognized rule* : *règle non reconnue*) se produira !

```
83  +|-|or                              {
84                                          sprintf(buffer, "OPADD: %s (%d caractere(s))", yytext, yyleng);
85                                          output(buffer);
86                                      }
```

```
"test.pascal.v1.l", line 83: unrecognized rule
"test.pascal.v1.l", line 83: unrecognized rule
"test.pascal.v1.l", line 83: unrecognized rule
"test.pascal.v1.l", line 83: unrecognized rule
"test.pascal.v1.l", line 83: unrecognized rule
"test.pascal.v1.l", line 83: unrecognized rule
"test.pascal.v1.l", line 83: unrecognized rule
```

```
    /*
        opaffect    (=)
    */
=                                   {
                                        sprintf(buffer, "OPAFFECT: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                    }
    /*
        opmul       (* / div mod and)
    */
\*|\/|div|mod|and                   {
                                        sprintf(buffer, "OPMUL: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                    }

{blanc}                             {
                                        sprintf(buffer, "BLANC: %s (%d caractere(s))", yytext, yyleng);
                                        /* output(buffer); */
                                    }
    /*
        Les caractéres ne sont pas reconnus
    */
.                                   {
                                        sprintf(buffer, "AUTRE: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                    }

%%
```

Partie Définitions et expressions régulières du fichier pascal.l (3/3)

* et / ont aussi des significations dans la syntaxe de *flex,* c'est pour cela qu'on a ajouté un antislash avant.

```
    /* Code C */

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

Partie Fonctions en C

Remarque : On peut améliorer l'analyseur par l'ajout d'autres fonctionnalités à savoir : la reconnaissance des chaines de caractères et des noms de fonctions (et non pas les traiter en tant que identificateurs), ignorer les commentaires, …

```
chaine  \'[^\']*\'
commentaire     "{"[^{}]*"}"
```

A ajouter dans la partie *Définitions*

```
    /*
            * nom de fonction: identificateur suivi par des parentheses
    */
{id}/{blanc}*\((.|\n)*\)                {
                                            sprintf(buffer, "NOM_FONCTION: %s (%d caractere(s))", yytext, yyleng);
                                            output(buffer);
                                        }
    /*
            * chaine de caractéres
    */
{chaine}                                {
                                            sprintf(buffer, "CHAINE: %s (%d caractere(s))", yytext, yyleng);
                                            output(buffer);
                                        }
```

```
    /*
            * commentaires: placés entre { ... }
    */
{commentaire}                           {
                                            sprintf(buffer, "COMMENTAIRE: %s (%d caractere(s))", yytext, yyleng);
                                            output(buffer);
                                        }
```

A ajouter dans la partie *Définitions et expressions régulières*

Analyseur lexicale: test

Pour tester notre analyseur lexical on doit tout d'abord générer le fichier (*.c*) à l'aide de *flex* et l'exécutable (*.exe*) à l'aide d'un compilateur C.

```
C:\Program Files (x86)\GnuWin32\bin>flex pascal.l

C:\Program Files (x86)\GnuWin32\bin>gcc lex.yy.c

C:\Program Files (x86)\GnuWin32\bin>
```

Génération du fichier .c (*flex*) et de l'exécutable .exe (*compilateur c: gcc*)

L'analyseur est prêt pour le test, on va utiliser un simple programme pascal dans le fichier *helloWorld.pas*

```
program HelloWorld;

begin
    writeln('Hello World');
end.
```

Contenu du fichier *helloWorld.pas*

```
C:\Program Files (x86)\GnuWin32\bin>a < codesPascal\helloWorld.pas
```

La commande de compilation (*gcc lex.yy.c*) produit l'exécutable (*a.exe*), le symbole(<) est utilisée pour modifier l'entrée standard (qui est par défaut le clavier), dans notre exemple on ne souhaite pas saisir le code nous même mais le lire à partir d'un fichier

```
Analyseur lexicale: mot cle: PROGRAM
Analyseur lexicale: ID: HelloWorld (10 caractere(s))
Analyseur lexicale: AUTRE: ; (1 caractere(s))
Analyseur lexicale: mot cle: BEGIN
Analyseur lexicale: ID: writeln (7 caractere(s))
Analyseur lexicale: AUTRE: ( (1 caractere(s))
Analyseur lexicale: AUTRE: ' (1 caractere(s))
Analyseur lexicale: ID: Hello (5 caractere(s))
Analyseur lexicale: ID: World (5 caractere(s))
Analyseur lexicale: AUTRE: ' (1 caractere(s))
Analyseur lexicale: AUTRE: ) (1 caractere(s))
Analyseur lexicale: AUTRE: ; (1 caractere(s))
Analyseur lexicale: mot cle: END
Analyseur lexicale: AUTRE: . (1 caractere(s))
```

Résultat de l'analyse lexicale du fichier *helloWorld.pas*

Analyse syntaxique

Pour l'analyse syntaxique on doit définir la grammaire qu'utilisent les compilateurs du langage pascal pour vérifier la syntaxe des codes entrés.

Le fichier *bison* (*.y*) a le même format que le fichier *flex* (*.l*)

Définitions: Où on peut inclure les librairies C et déclarer des Variables auxiliaires ou alors des signatures de fonctions à Implémenter dans la dernière partie du fichier.

Définition de la grammaire: Dans cette partie on doit inscrire les Différents règles de production de la grammaire.

```
... définitions ...
%%
... définition de la grammaire ...
%%
... fonctions en C ...
```

Fonctions en C: La dernière partie du fichier *bison* est consacré à l'implémentation des fonctions principales et optionnels utiles dans la phase de l'analyse sémantique.

```
%{

    #include <stdio.h>
    #include <stdlib.h>
    int yylex(void);
    int yyerror(char*);

%}
```

Partie définitions du fichier pascal.y (1/2)

```
    /* l'axiome de la grammaire */
%start programme

%token ID
%token NB
%token OPADD
%token OPAFFECT
%token OPMUL
%token OPREL

    /* Mots clés */
%token BEGIN
%token DO
%token ELSE
%token END
%token FUNCTION
%token IF
%token NOT
%token PROCEDURE
%token PROGRAM
%token THEN
%token VAR
%token WHILE
%token INT

%%
```

Partie définitions du fichier pascal.y (2/2)

```
programme:
    PROGRAM ID ';' declaration instruction_composee '.'
;

declaration:
    declaration_var declarations_sous_programmes
;

declaration_var:
    declaration_var VAR liste_identificateurs ':' INT ';'
|   /* chaîne vide */
;

liste_identificateurs:
    ID
|   liste_identificateurs ',' ID
;

declarations_sous_programmes:
    declarations_sous_programmes declarations_sous_programme ';'
|   /* chaîne vide */
;

declarations_sous_programme:
    entete_sous_programme declaration instruction_composee
;

entete_sous_programme:
    FUNCTION ID arguments ':' INT ';'
|   PROCEDURE ID arguments ';'
;
```

Partie définition de la grammaire du fichier pascal.y (1/4)

```
arguments:
    '(' liste_parametres ')'
|   /* chaîne vide */
;

liste_parametres:
    parametre
|   liste_parametres ';' parametre
;

parametre:
    ID ':' INT
|   VAR ID ':' INT
;

instruction_composee:
    BEGIN instructions_optionnelles END
;

instructions_optionnelles:
    liste_instructions
|   /* chaîne vide */
;

liste_instructions:
    instruction
|   liste_instructions ';' instruction
;

instruction:
    variable OPAFFECT expression
|   appel_de_procedure
|   instruction_composee
|   IF expression THEN instruction ELSE instruction
|   WHILE expression DO instruction
;
```

Partie définition de la grammaire du fichier pascal.y (2/4)

```
variable:
    ID
;

appel_de_procedure:
    ID
|   ID '(' liste_expressions ')'
;


liste_expressions:
    expression
|   liste_expressions ',' expression
;

expression:
    expression_simple
|   expression_simple OPREL expression_simple
;

expression_simple:
    terme
|   signe terme
|   expression_simple OPADD terme
;

terme:
    facteur
|   terme OPMUL facteur
;

facteur:
    ID
|   ID '(' liste_expressions ')'
|   NB
|   '(' expression ')'
|   NOT facteur
;
```

Partie définition de la grammaire du fichier pascal.y (3/4)

```
signe:
    '+'
|   '-'
;

%%
```

Partie définition de la grammaire du fichier pascal.y (4/4)

```
#include "lex.yy.c"

int yyerror(char *s) {
    printf ("erreur: %s", s);
    return 0;
}

int main() {
    yyparse();
    return 0;
}
```

Partie fonctions en C du fichier pascal.y

NB: la ligne (*#include "lex.yy.c"*) permet d'inclure le fichier (.*c*) généré par *flex* pour qu'il soit compilé avec le fichier (.*c*) généré par *bison,* plus précisément, le fichier *lex.yy.c* contient la fonction *yylex()* vu dans la partie de l'analyse lexicale qui lit le texte en entrée caractère par caractère, reconnait et retourne une suite d'unités lexicales, cette fonction sera appelée par la fonction *yyparse()* de l'analyseur syntaxique pour obtenir l'unité lexicale suivante. On peut remarquer que la signature de la fonction *yylex()* se trouve dans la première partie du fichier *bison* alors que son implémentation est dans le fichier *lex.yy.c*

A ce niveau, on doit apporter quelques modifications sur le fichier *flex* pour permettre la coopération entre l'analyseur lexicale et l'analyseur syntaxique: l'option –d de *bison* permet de générer un fichier (.*h*) contenant les différents symboles terminales codés chacun par un nombre afin de les identifier, ce fichier doit être inclus dans le fichier spécification de l'analyseur lexicale, ensuite, à chaque identification d'une unité lexicale, on doit retourner son code correspondant pour qu'il soit reconnu par *bison,* et si le symbole terminal est écrit à la main (entre guillemets), dans le fichier (.*l*) on doit retourner le caractère lui-même.

```
/* Tokens.  */
#define ID 258
#define NB 259
#define OPADD 260
#define OPAFFECT 261
#define OPMUL 262
#define OPREL 263
#define BEGIN 264
#define DO 265
#define ELSE 266
#define END 267
#define FUNCTION 268
#define IF 269
#define NOT 270
#define PROCEDURE 271
#define PROGRAM 272
#define THEN 273
#define VAR 274
#define WHILE 275
#define INT 276
```

Les différents codes de symboles terminaux dans le fichier *pascal.tab.h*

```
    /* Définitions */

%{

    /* Code C */
int outputLexical = 1;  // flag qui permet d'afficher(1)/masquer(0)
    /* fonction qui affiche un message passé en paramétre */
void output(const char* msg) {
    if(outputLexical == 1) {
        printf("Analyseur lexicale: %s\n", msg);
    }
}
    /* chaine de caractéres utilisé avec la fonction output() */
char buffer[50];

#include "pascal.tab.h"
```

Partie définitions du fichier (.*l*) après modification
(Inclusion du fichier *pascal.tab.h*)

```
[bB][eE][gG][iI][nN]                { output("mot cle: BEGIN");     return BEGIN;       }
[dD][oO]                            { output("mot cle: DO");        return DO;          }
[eE][lL][sS][eE]                    { output("mot cle: ELSE");      return ELSE;        }
[eE][nN][dD]                        { output("mot cle: END");       return END;         }
[fF][uU][nN][cC][tT][iI][oO][nN]    { output("mot cle: FUNCTION");  return FUNCTION;    }
[iI][fF]                            { output("mot cle: IF");        return IF;          }
[iI][nN][tT]                        { output("mot cle: INT");       return INT;         }
[pP][rR][oO][cC][eE][dD][uU][rR][eE] { output("mot cle: PROCEDURE"); return PROCEDURE;   }
[pP][rR][oO][gG][rR][aA][mM]        { output("mot cle: PROGRAM");   return PROGRAM;     }
[tT][hH][eE][nN]                    { output("mot cle: THEN");      return THEN;        }
[vV][aA][rR]                        { output("mot cle: VAR");       return VAR;         }
[wW][hH][iI][lL][eE]                { output("mot cle: WHILE");     return WHILE;       }
```

Partie définitions et expressions régulières après modification

```
    /*
          nb          (nombres: chiffre chiffre*)
    */
{nb}                                {
                                        sprintf(buffer, "NB: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                        return NB;
                                    }
    /*
          id          (identificateurs)
    */
{id}                                {
                                        sprintf(buffer, "ID: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                        return ID;
                                    }
    /*
          oprel       (opérateurs relationnels: == <> < > <= >=)
    */
==|<>|<|>|<=|>=                     {
                                        sprintf(buffer, "OPREL: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                        return OPREL;
                                    }
[nN][oO][tT]                        {
                                        sprintf(buffer, "OPREL/NOT: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                        return NOT;
                                    }
```

Remarque: une autre petite modification sur le fichier *flex* est faite: l'opérateur not est traité séparément puisque qu'il est traité séparément au niveau de la grammaire.
On préfère utiliser *[nN][oO][tT]* au lieu de *not* afin de détecter toutes les combinaisons possibles des mots clés (dans le cas du mot *not*: not, Not, NOT, ...)
* la règle de l'opérateur *not* doit être mise avant celle des identificateurs sinon un avertissement s'affiche pour prévoir que toutes les instances de l'opérateur *not* peuvent être considérées comme identificateurs !

```
C:\Program Files (x86)\GnuWin32\bin>flex pascal.l
"pascal.l", line 87: warning, rule cannot be matched
```

```
    /*
            opadd       (+ - or)
    */
\+
|   -
|   [oO][rR]                          {
                                        sprintf(buffer, "OPADD: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                        return OPADD;
                                    }
```

```
    /*
            opaffect    (=)
    */
=                                   {
                                        sprintf(buffer, "OPAFFECT: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                        return OPAFFECT;
                                    }
```

```
    /*
            opmul       (* / div mod and)
    */
\*
|   \/
|   [dD][iI][vV]
|   [mM][oO][dD]
|   [aA][nN][dD]                      {
                                        sprintf(buffer, "OPMUL: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                        return OPMUL;
                                    }
```

```
    /*
            * caractéres blancs: espace, tabulation, retour chariot
    */
{blanc}                             {
                                        /* les caractéres blancs sont à ignorer, on ne retourne rien */
                                    }
```

```
    /*
            * commentaires: placés entre { ... }
    */
{commentaire}                       {
                                        sprintf(buffer, "COMMENTAIRE: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                        /* les commentaires sont à ignorer aussi */
                                    }
```

Remarque: bien évidemment, les caractères blancs sont à ignorer, les commentaires le sont aussi: leur rôle est de clarifier le code, ils sont utiles pour les humains non pas pour la machine ! Pour la machine, ils sont totalement inutiles.

```
    /*
        Les caractéres restants
    */
.                                   {
                                        sprintf(buffer, "AUTRE: %s (%d caractere(s))", yytext, yyleng);
                                        output(buffer);
                                        /* dans le cas des autres caractéres, on doit le retourner tel qu'il est */
                                        return *yytext; /* ou alors, return yytext[0] */
                                    }
```

Remarque: tout les autres caractères sont à retourner tels quels.

```
C:\Program Files (x86)\GnuWin32\bin>gcc pascal.tab.c
In file included from pascal.y:152:0:
lex.yy.c:79:0: warning: "BEGIN" redefined
 #define BEGIN yy_start = 1 + 2 *
 ^
```

Lors de la compilation du fichier *pascal.tab.c* une erreur s'affiche pour signaler qu'il y a un problème avec la déclaration de la constante BEGIN! Il parait que *flex* possède une macro du même nom (définie dans le fichier *lex.yy.c*)

```
/* Enter a start condition.  This macro really ought to take a parameter,
 * but we do it the disgusting crufty way forced on us by the ()-less
 * definition of BEGIN.
 */
#define BEGIN yy_start = 1 + 2 *
```

Définition de BEGIN dans le fichier *lex.yy.c*

Pour corriger ce problème, on doit changer le nom du *token* BEGIN.

```
    /* Mots clés */
%token BEGIN_TOKEN
```

```
instruction_composee:
    BEGIN_TOKEN instructions_optionnelles END
;
```

Correction du fichier *bison* pascal.y

```
[bB][eE][gG][iI][nN]                    { output("mot cle: BEGIN");     return BEGIN_TOKEN;     }
```

Correction du fichier *flex* pascal.l

Lors de la compilation, une nouvelle erreur apparait qui indique que la fonction *main()* est redéfinie, pour corriger cet erreur, il faut supprimer la fonction *main()* au niveau du fichier *flex*

```
C:\Program Files (x86)\GnuWin32\bin>gcc pascal.tab.c
pascal.y:159:5: error: redefinition of 'main'
 int main() {
     ^
In file included from pascal.y:152:0:
pascal.l:149:5: note: previous definition of 'main' was here
 int main(void) {
     ^
```

L'erreur de compilation

```
%token CHAINE
```

```
    /*
        * chaine de caractéres
    */
{chaine}                        {
                                    sprintf(buffer, "CHAINE: %s (%d caractere(s))", yytext, yyleng);
                                    output(buffer);
                                    return CHAINE;
                                }
```

Modifications sur le fichier (*.l*)

```
facteur:
    ID
|   ID
    '('
    liste_expressions
    ')'
|   NB
|   '('
    expression
    ')'
|   NOT
    facteur
|   CHAINE
;
```

Modification sur le fichier (*.y*)

Le facteur peut être dérivé en chaine de caractère d'où la nécessité de le définir dans le fichier de spécification de l'analyseur lexicale.
Exemple: l'appel de la fonction *println('hello world');*

```
instruction:
    variable
    OPAFFECT
    expression
|   appel_de_procedure
|   instruction_composee
|   IF
    expression
    THEN
    instruction
    ELSE
    instruction
|   WHILE
    expression
    DO
    instruction
|   /* chaine vide */
;
```

On doit aussi remarquer que l'instruction peut être vide.
Exemple: l'appel de la fonction *readln();* cette fonction n'a pas d'arguments, si la règle de production instruction → ε n'est pas permise dans la grammaire, une erreur syntaxique sera généré !
Suite à cette modification un conflit du type réduction/réduction (reduce/reduce) apparait.

```
test.pascal.y: conflits: 1 réduction/réduction
```

Il parait que notre grammaire est ambigüe ! *bison* nous signale qu'il a trouvé un conflit du type réduction/réduction
Remarque: *bison* reconnaît 2 types de conflits:
* réduction/réduction (reduce/reduce)
Exemple:
A → a
B → a
Le conflit se produit en lisant le symbole 'a', on réduit au non terminal A ou B ? Par défaut *bison* choisit la première règle qui apparait dans le fichier.
* décalage/réduction (shift/reduce)

Exemple:
A → A + A | A * A | nb
Le conflit se produit par exemple en lisant: nb + nb * nb

Pile	Entrée	Action
ε	nb + nb * nb	décalage
nb	+ nb * nb	réduction
A	+ nb * nb	décalage
A +	nb * nb	décalage
A + nb	* nb	réduction
A + A	* nb	décalage ou réduction ?

2 scénarios possibles: A + A *, A + A * nb, A + A * A, A + A, A ou alors, A, A *, A * nb, A * A, A
Lequel choisir ? *bison* dans ce type de conflit choisit de décaler, dans cet exemple le conflit est du au non-respect de la priorité de la multiplication par rapport à l'addition.

Comment trouver ces conflits ?

En utilisant l'option –v de *bison* qui permet de générer un fichier (*.output*)

Revenons au conflit trouvé, il est dû aux deux chemins possibles pour avoir une liste d'instructions vide:

① instructions_optionnelles → ε
② instructions_optionnelles → liste_instructions →instruction → ε

Où ε désigne la chaine vide.
La solution est d'éliminer la première règle qui parce qu'elle est redondante et ne traite pas tout les cas possibles.

Analyse syntaxique : test

Pour essayer notre analyseur syntaxique on va lui fournir deux codes le premier est syntaxiquement correct, le second est syntaxiquement incorrect pour voir s'il va détecter l'erreur ou pas.
Pour améliorer le compilateur, on peut ajouter une fonctionnalité qui permet de suivre les lignes dans le code, quand l'analyseur trouve une erreur il peut aussi afficher dans quelle ligne du code il l'a trouvé

```
%}

%option yylineno
```

Modification du fichier *flex*

```
%error-verbose
```

Cette option permet d'avoir des messages d'erreurs plus significatives

```
int main() {
    yydebug = 1;
    yyparse();
    return 0;
}
```

Pour mieux voir le déroulement de l'analyse syntaxique et la manipulation de l'automate à pile (vérification de la grammaire), on affecte une valeur non nulle à la variable yydebug

```
void yyerror(char *s) {
    extern int yylineno;
    char str[100];
    sprintf(str,"Erreur (ligne n %d): %s\n", yylineno, s);
    printf(str);
}
```

Modification du fichier *bison*

Détection d'erreurs

```
programme:
    PROGRAM
    ID
    ';'
    declaration
    instruction_composee
    '.'                     { output_syn("fin du programme"); }
|   PROGRAM ID error        { yyerror("point virgule omis"); }
;
```

Pour détecter l'erreur de l'absence du point virgule, on doit ajouter une règle dans la grammaire.
Testons avec le code suivant :

```
program HelloWorld

begin
    writeln('Hello World');
end.
```

On obtient le résultat attendu :

```
Erreur (ligne n 3): syntax error, unexpected BEGIN_TOKEN, expecting ';'
Erreur (ligne n 3): point virgule omis
```

Pour détecter l'erreur de l'absence du point virgule à la fin des instructions, on doit modifier les règles de la liste des instructions comme suit :

```
liste_instructions:
    instruction ';'
|   liste_instructions
    instruction ';'
|   instruction
    error                   { yyerror("point virgule d'instruction omis"); }
;
```

Testons avec le code suivant :

```
program HelloWorld;

begin
    writeln('Hello World')
end.
```

On obtient le résultat attendu :

```
Erreur (ligne n 5): syntax error, unexpected END, expecting ';'
Erreur (ligne n 5): point virgule d'instruction omis
Analyseur syntaxique: fin du programme
```

Pour détecter l'erreur dans le cas d'absence ou de modification du mot clé 'int', on doit ajouter une règle qui détecte cette erreur parmi les régles de déclaration de variables comme suit :

```
declaration_var:
    declaration_var
    VAR
    liste_identificateurs
    ':'
    INT
    ';'
|   /* chaîne vide */
|   declaration_var
    VAR
    liste_identificateurs
    ':'
    error                   { yyerror("mot cle 'int' introuvable !"); }
;
```

Testons avec le code suivant (*intt* au lieu de *int*) :

```
program HelloWorld;

var
    i : intt;
begin
    writeln('Hello World');
end.
```

On obtient le résultat attendu :

```
Erreur (ligne n 4): syntax error, unexpected ID, expecting INT
Erreur (ligne n 4): mot cle 'int' introuvable !
Erreur (ligne n 4): point virgule omis
```

Dans le cas d'un programme correct, l'analyseur affiche le message 'fin du programme'
Prenons par exemple le code suivant :

```
program HelloWorld;

begin
    writeln('Hello World');
end.
```

On obtient à la sortie de l'analyseur syntaxique :

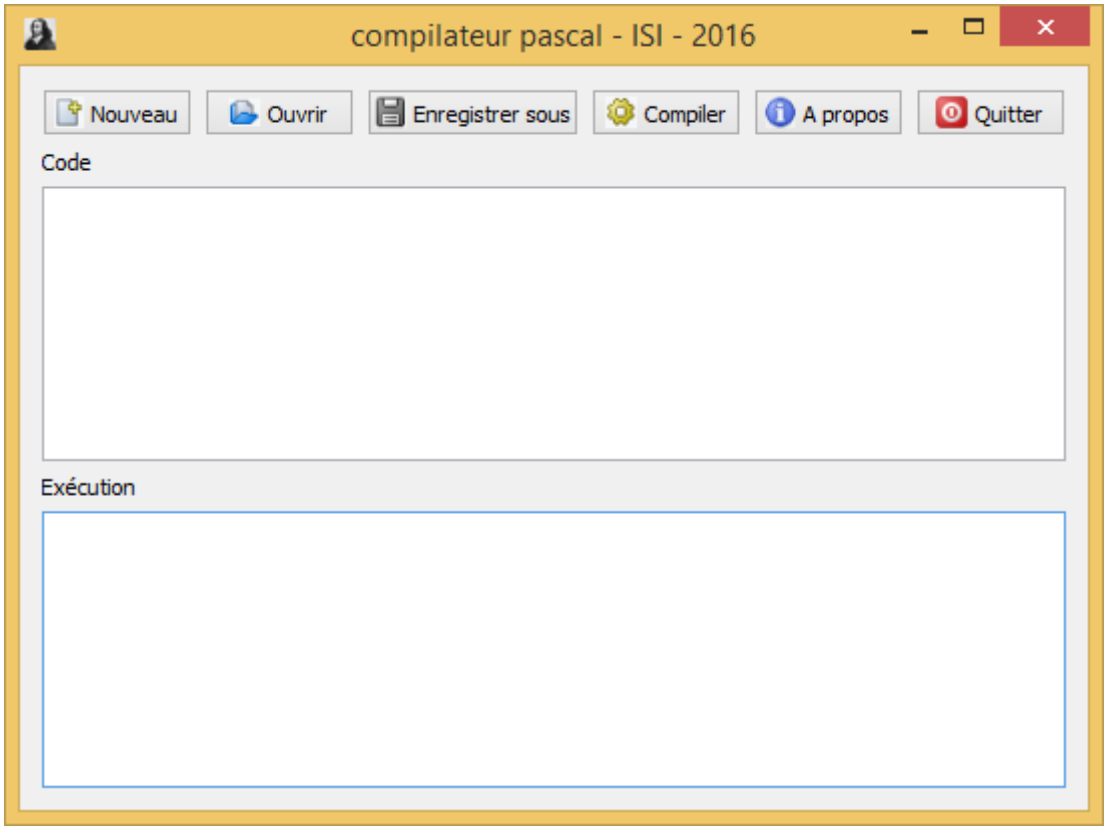
```
Analyseur syntaxique: fin du programme
```

Interface graphique

L'interface graphique est crée à l'aide de la bibliothèque Qt.

Logo de la bibliothèque Qt

Le principe de l'interface graphique est assez simple, sauvegarder le contenu écrit dans la partie du code dans un fichier *code.pas* qui va être ensuite envoyé au compilateur, après sa traitement tout ce qui va être affiché sur la console précédemment va être affiché à présent dans la partie exécution de l'interface graphique.



L'interface graphique

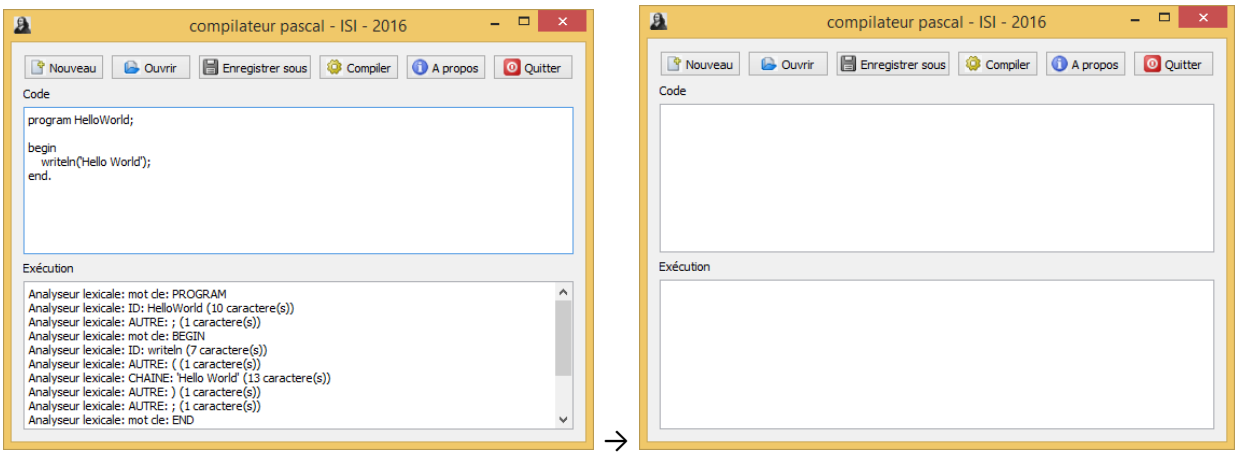
Pour cela on doit modifier le point d'entrée du compilateur pour qu'il ouvre le fichier dont le nom est passé en paramètres et donc de ne plus considéré l'entrée standard (*clavier*)

```c
int main(int argc, char *argv[]) {
    yydebug = 0;
    argc--, argv++; /* ignorer le 1er paramétre: le nom du fichier C actuel */
    if(argc > 0) {
        yyin = fopen(argv[0], "r");
    } else {
        printf("fichier introuvable !");
        return 0;
    }
    yyparse();
    return 0;
}
```

La nouvelle version de la fonction *main()* dans le fichier *bison*

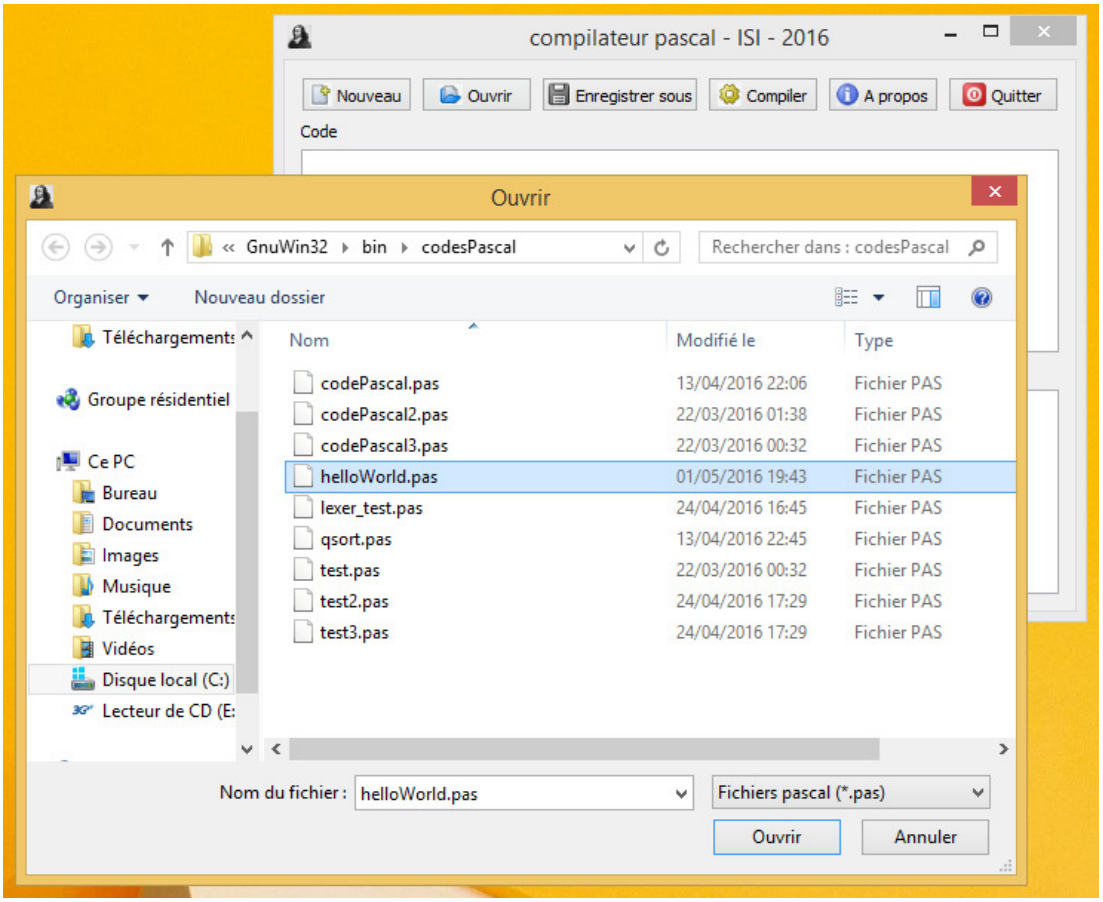
L'analyseur lit à présent le fichier dont le nom est passé en paramètre.

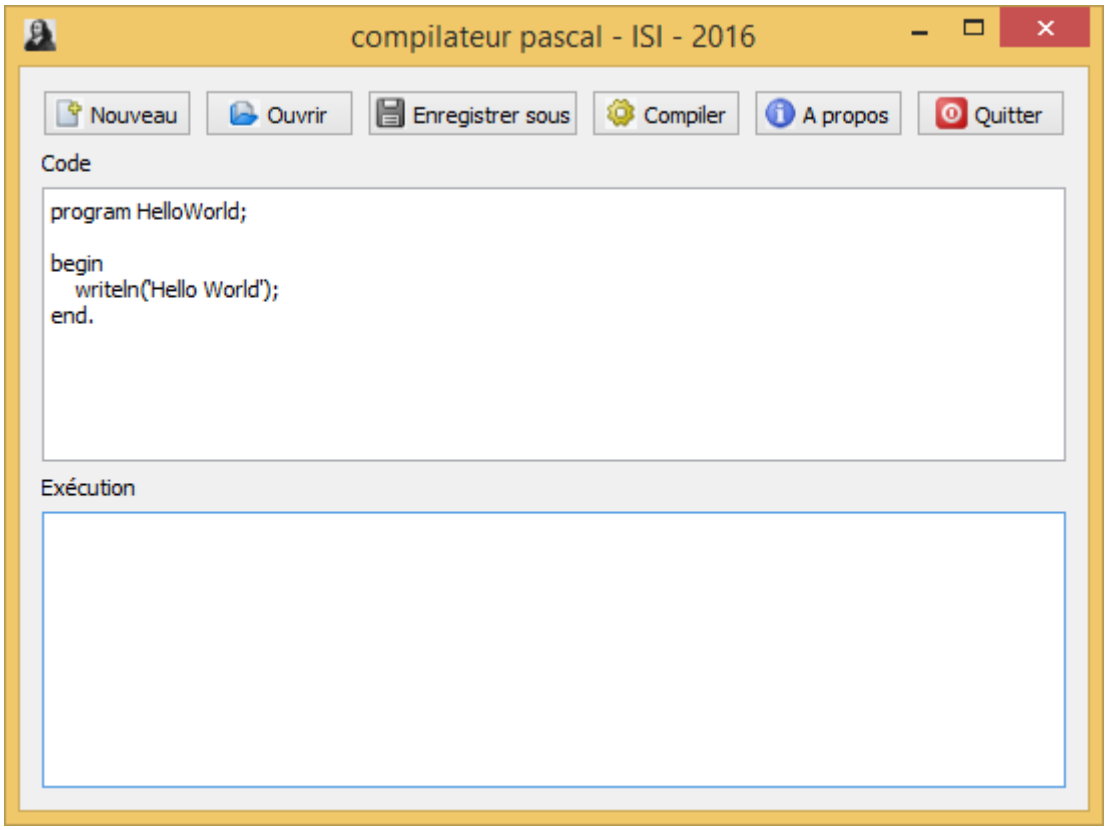
Le bouton *Nouveau* permet de vider les champs de code et d'exécution pour une nouvelle compilation.



Avant et après le click sur le bouton *Nouveau*

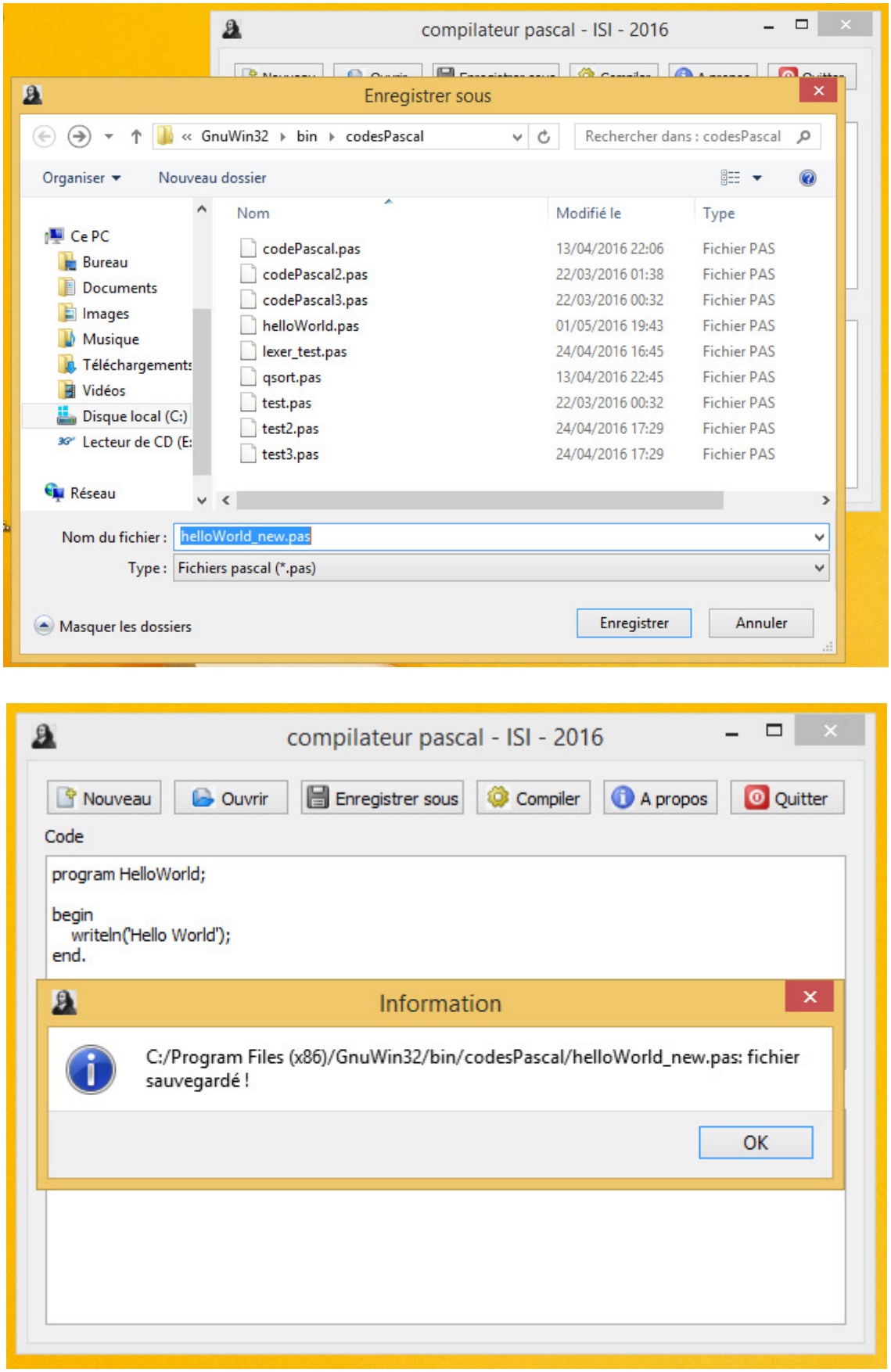
Le bouton *Ouvrir* permet de parcourir le système de fichiers de votre ordinateur pour ouvrir un fichier (l'extension est *.pas* par défaut) , comme on peut écrire le code directement dans la partie code.





Bouton *Ouvrir*

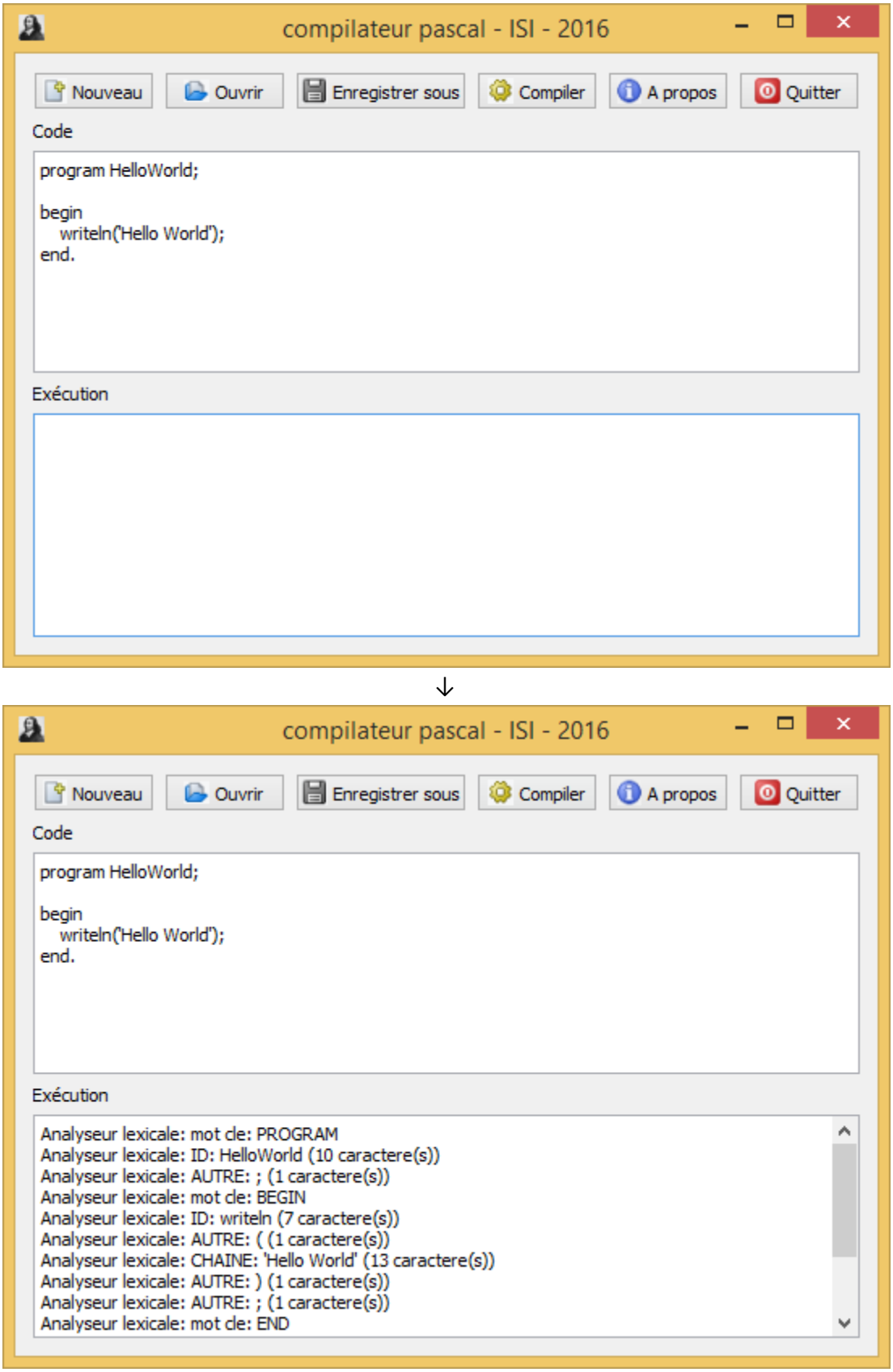
Le bouton *Enregistrer sous* permet d'enregistrer le code de affiché dans un fichier sur votre ordinateur.



Bouton *Enregistrer sous*

Le bouton *compiler* permet de sauvegarder le contenu de la partie *code* dans un fichier *code.pas* et de lancer l'analyseur *analyseur.exe* en lui passant en argument le nom du fichier *code.pas.*

Après la fin de la compilation, l'application récupère tout ce qui était affiché sur la console précédemment pour l'afficher à présent dans la partie *Exécution* de l'interface graphique et un son s'entend pour signaler la fin de la compilation.



Bouton Compiler

Fichier *flex* : pascal.l

```
    /* Définitions */

%{

    /* Code C */
int outputLexical = 1;  // flag qui permet d'afficher(1)/masquer(0) les messages de
l'analyseur lexicale, par défaut (1)
    /* fonction qui affiche un message passé en paramétre */
void output(const char* msg) {
    if(outputLexical == 1) {
        printf("Analyseur lexicale: %s\n", msg);
    }
}
    /* chaine de caractéres utilisé avec la fonction output() */
char buffer[50];

#include "pascal.tab.h"

%}

%option yylineno

lettre          [a-zA-Z]
chiffre         [0-9]
id              {lettre}({lettre}|{chiffre})*
nb              {chiffre}+

blanc           [ \t\n]+
chaine          \'[^\']*\'
commentaire     "{"[^{}]*"}"

%%

    /* Définitions et expressions régulières */

    /*
        lexèmes à reconnaître:
            mots clés:
```

```
                begin
                do
                else
                end
                function
                if
                int
                procedure
                program
                then
                var
                while
    */

[bB][eE][gG][iI][nN]                    { output("mot cle: BEGIN");     return
BEGIN_TOKEN;     }
[dD][oO]                                { output("mot cle: DO");        return DO;
}
[eE][lL][sS][eE]                        { output("mot cle: ELSE");      return ELSE;
}
[eE][nN][dD]                            { output("mot cle: END");       return END;
}
[fF][uU][nN][cC][tT][iI][oO][nN]        { output("mot cle: FUNCTION");  return
FUNCTION;        }
[iI][fF]                                { output("mot cle: IF");        return IF;
}
[iI][nN][tT]                            { output("mot cle: INT");       return INT;
}
[pP][rR][oO][cC][eE][dD][uU][rR][eE]    { output("mot cle: PROCEDURE"); return
PROCEDURE;       }
[pP][rR][oO][gG][rR][aA][mM]            { output("mot cle: PROGRAM");   return
PROGRAM;         }
[tT][hH][eE][nN]                        { output("mot cle: THEN");      return THEN;
}
[vV][aA][rR]                            { output("mot cle: VAR");       return VAR;
}
[wW][hH][iI][lL][eE]                    { output("mot cle: WHILE");     return WHILE;
}

    /*
            nb          (nombres: chiffre chiffre*)
```

```
    */
{nb}                                    {
                                            sprintf(buffer, "NB: %s (%d
caractere(s))", yytext, yyleng);
                                            output(buffer);
                                            return NB;
                                        }
[nN][oO][tT]                            {
                                            sprintf(buffer, "OPREL/NOT: %s (%d
caractere(s))", yytext, yyleng);
                                            output(buffer);
                                            return NOT;
                                        }
    /*
            id          (identificateurs)
    */
{id}                                    {
                                            sprintf(buffer, "ID: %s (%d
caractere(s))", yytext, yyleng);
                                            output(buffer);
                                            return ID;
                                        }
    /*
            oprel       (opérateurs relationnels: == <> < > <= >=)
    */
==|<>|<|>|<=|>=                         {
                                            sprintf(buffer, "OPREL: %s (%d
caractere(s))", yytext, yyleng);
                                            output(buffer);
                                            return OPREL;
                                        }
    /*
            opadd       (+ - or)
    */
\+|-|[oO][rR]                           {
                                            sprintf(buffer, "OPADD: %s (%d
caractere(s))", yytext, yyleng);
                                            output(buffer);
                                            return OPADD;
                                        }
    /*
```

```
            opaffect    (=)
    */
=                                       {
                                            sprintf(buffer, "OPAFFECT: %s (%d
caractere(s))", yytext, yyleng);
                                            output(buffer);
                                            return OPAFFECT;
                                        }
    /*
            opmul       (* / div mod and)
    */
\*|\/|[dD][iI][vV]|[mM][oO][dD]|[aA][nN][dD]    {
                                            sprintf(buffer, "OPMUL: %s (%d
caractere(s))", yytext, yyleng);
                                            output(buffer);
                                            return OPMUL;
                                        }
    /*
            * chaine de caractéres
    */
{chaine}                                {
                                            sprintf(buffer, "CHAINE: %s (%d
caractere(s))", yytext, yyleng);
                                            output(buffer);
                                            return CHAINE;
                                        }

    /*
            * caractéres blancs: espace, tabulation, retour chariot
    */
{blanc}                                 {
                                            /* les caractéres blancs sont à ignorer,
on ne retourne rien */
                                        }
    /*
            * commentaires: placés entre { ... }
    */
{commentaire}                           {
                                            sprintf(buffer, "COMMENTAIRE: %s (%d
caractere(s))", yytext, yyleng);
                                            output(buffer);
```

```
                                        /* les commentaires sont à ignorer aussi
*/
                                    }
    /*
        Les caractéres restants
    */
.                                   {
                                        sprintf(buffer, "AUTRE: %s (%d
caractere(s))", yytext, yyleng);
                                        output(buffer);
                                        /* dans le cas des autres caractéres, on
doit le retourner tel qu'il est */
                                        return *yytext; /* ou alors, return
yytext[0] */
                                    }

%%

    /* Code C */

int yywrap(void) {
    return 1;
}
```

Fichier *bison* : pascal.y

```
%{

    #include <stdio.h>
    #include <stdlib.h>
    int yylex(void);
    void yyerror(char*);

    int outputSyntaxique = 1;   // flag qui permet d'afficher(1)/masquer(0) les
messages de l'analyseur syntaxique, par défaut (1)
        /* fonction qui affiche un message passé en paramétre */
    void output_syn(const char* msg) {
        if(outputSyntaxique == 1) {
            printf("Analyseur syntaxique: %s\n", msg);
        }
```

```
    }

%}

%debug

    /* l'axiome de la grammaire */
%start programme

%token ID
%token NB
%token OPADD
%token OPAFFECT
%token OPMUL
%token OPREL

    /* Mots clés */
%token BEGIN_TOKEN
%token DO
%token ELSE
%token END
%token FUNCTION
%token IF
%token NOT
%token PROCEDURE
%token PROGRAM
%token THEN
%token VAR
%token WHILE
%token INT

%token CHAINE

%error-verbose

%%

programme:
    PROGRAM
    ID
    ';'
```

```
    declaration
    instruction_composee
    '.'                     { output_syn("fin du programme"); }
|   PROGRAM ID error        { yyerror("point virgule omis"); }
;

declaration:
    declaration_var
    declarations_sous_programmes
;

declaration_var:
    declaration_var
    VAR
    liste_identificateurs
    ':'
    INT
    ';'
|   /* chaîne vide */
|   declaration_var
    VAR
    liste_identificateurs
    ':'
    error                   { yyerror("mot cle 'int' introuvable !"); }
;

liste_identificateurs:
    ID
|   liste_identificateurs
    ','
    ID
;

declarations_sous_programmes:
    declarations_sous_programmes
    declarations_sous_programme
    ';'
|   /* chaîne vide */
;

declarations_sous_programme:
```

```
    entete_sous_programme
    declaration
    instruction_composee
;

entete_sous_programme:
    FUNCTION
    ID
    arguments
    ':'
    INT
    ';'
|   PROCEDURE
    ID
    arguments
    ';'
;

arguments:
    '('
    liste_parametres
    ')'
|   /* chaîne vide */
;

liste_parametres:
    parametre
|   liste_parametres
    ';'
    parametre
;

parametre:
    ID
    ':'
    INT
|   VAR
    ID
    ':'
    INT
;
```

```
instruction_composee:
    BEGIN_TOKEN
    instructions_optionnelles
    END
;

instructions_optionnelles:
    liste_instructions
;

liste_instructions:
    instruction ';'
|   liste_instructions
    instruction ';'
|   instruction
    error                   { yyerror("point virgule d'instruction omis"); }
;

instruction:
    variable
    OPAFFECT
    expression
|   appel_de_procedure
|   instruction_composee
|   IF
    expression
    THEN
    instruction
    ELSE
    instruction
|   WHILE
    expression
    DO
    instruction
|   /* chaîne vide */
;

variable:
    ID
;
```

```
appel_de_procedure:
    ID
|   ID
    '('
    liste_expressions
    ')'
;


liste_expressions:
    expression
|   liste_expressions
    ','
    expression
;

expression:
    expression_simple
|   expression_simple
    OPREL
    expression_simple
;

expression_simple:
    terme
|   signe terme
|   expression_simple
    OPADD
    terme
;

terme:
    facteur
|   terme
    OPMUL
    facteur
;

facteur:
    ID
```

```
|   ID
    '('
    liste_expressions
    ')'
|   NB
|   '('
    expression
    ')'
|   NOT
    facteur
|   CHAINE
;

signe:
    '+'
|   '-'
;

%%

#include "lex.yy.c"

void yyerror(char *s) {
    extern int yylineno;
    char str[100];
    sprintf(str,"Erreur (ligne n %d): %s\n", yylineno, s);
    printf(str);
}

int main(int argc, char *argv[]) {
    yydebug = 0;
    argc--, argv++; /* ignorer le 1er paramétre: le nom du fichier C actuel */
    if(argc > 0) {
        yyin = fopen(argv[0], "r");
    } else {
        printf("fichier introuvable !");
        return 0;
    }
    yyparse();
    return 0;
}
```