DEUG MIAS2
Langages formels.

PROJET

M.M. Institut Galilée
1999–2000

Le but de ce projet est de construire, à l'aide de Lex et Yacc, un traducteur du langage *Galileo* en un P–code. La grammaire de *Galileo* et l'ensemble des P–instructions qui sont utiles à sa traduction sont présentés dans ce qui suit.

- Le projet doit être fait par groupes de 2 ou 3 étudiants.
- Un programme capable d'éditer un *P*–code de façon lisible vous sera fourni.
- Un programme capable d'exécuter un *P*–code vous sera fourni. Afin que nous soyons d'accord, vous devez impérativement respecter certaines déclarations, qui seront encore signalées dans le schéma du fichier `GAL.Y` ci–joint.
- Il vous revient d'écrire les programmes suivants :
  - Un programme Lex, dans un fichier appelé `GALLEX.L`, qui doit produire un analyseur lexical du langage *Galileo* (une fonction `yylex`).
  - Un programme Yacc, dans un fichier appelé `GAL.Y`, qui doit produire un analyseur syntaxique de la grammaire de *Galileo* (une fonction `yyparse`).

Le programme `GAL.EXE` que vous obtiendrez devra être capable de traduire un programme écrit en langage *Galileo* (dans un fichier qui doit avoir le suffixe `.GAL`) en un programme écrit en *P*–code (dans un fichier ayant le même nom que le précédent, mais avec le suffixe `.PCO`).

Les pages qui suivent vous proposent des schémas de ces fichiers. Ils ne sont pas contractuels : soyez donc vigilants ...

- Il vous revient de les compléter et de les mettre au point (des points de suspension (...) signalent les endroits où il reste du travail à faire) :
  - en définissant les procédures et les fonctions qui ont été signalées dans le sujet,
  - en complétant les actions de l'analyseur lexical et de l'analyseur syntaxique.
- Vous testerez votre traducteur sur des programmes de votre choix, pas trop simples (ils devront en particulier, comporter des tableaux).
- Le traitement des erreurs lexicales est assez facile; prévoyez des messages d'erreur qui soient utilisables : chacun d'entre eux devra comporter le texte erroné (utilisez `yytext`) et la ligne où il se trouve (utilisez `yylineno`).
- Le traitement des erreurs syntaxiques est difficile! Yacc possède un `token` appelé `error` qui est prévu à cet effet (ne considérez ce problème qu'à l'extrême fin de votre travail).

Vous devrez présenter, sous forme de listings, vos programmes `GALLEX.L`, `GAL.Y` et les fichiers `.GAL` et `.PCO` issus de vos meilleurs tests. Une "soutenance" aura lieu en salle machines.

Présentation de la grammaire.

Comme son nom l'indique, *Galileo* est un langage "maison" : c'est, en français, un langage impératif très simple inspiré de Pascal, dont la sémantique est par conséquent connue de tous.

La grammaire de Galileo

Programmes.

1 : $P \to$ **programme** Dv Li **fin**

Déclarations de **v**ariables.

2 : $Dv \to \varepsilon$	3 : $Dv \to$ **variables** Lid **fin**

Listes d'**id**entificateurs.

4 : $Lid \to$ **id** : Ty	5 : $Lid \to Lid$ **id** : Ty

Types.

6 : $Ty \to$ **entier**	7 : $Ty \to$ **tableau** [**nb**]

Listes d'**i**nstructions.

8 : $Li \to I$	9 : $Li \to Li$; I

Instructions.

10 : $I \to V$:= E	13 : $I \to$ **tantque** E **faire** Li **fin**
11 : $I \to$ **si** E **alors** Li **fin**	14 : $I \to$ **ecrire** (E)
12 : $I \to$ **si** E **alors** Li **sinon** Li **fin**	15 : $I \to$ **retourne**

Variables.

16 : $V \to$ **id**	17 : $V \to$ **id** [ES]

Expressions.

18 : $E \to ES$	25 : $F \to$ (E)
19 : $E \to ES$ **oprel** ES	26 : $F \to - F$
20 : $ES \to T$	27 : $F \to$ **non** F
21 : $ES \to ES$ **opadd** T	28 : $F \to$ **id** [ES]
22 : $ES \to ES - T$	29 : $F \to$ **lire**
23 : $T \to F$	30 : $F \to$ **id**
24 : $T \to T$ **opmult** F	31 : $F \to$ **nb**

• Les types autorisés sont les entiers et les tableaux d'entiers (on n'a donc pas jugé utile de mentionner **entier** dans la règle 7). L'introduction d'une expression de type tableau se produisant par application de la règle 30 est à prohiber : il n'est pas question de faire des calculs sur un tel objet (les calculs sur un tableau se font sur ses composantes!)

• Pour en finir avec les tableaux : le domaine d'un tableau de type **tableau** `[ n ]` est l'ensemble des entiers `0, ..., n - 1`.

• Les entiers sont interprétés comme des booléens lorsque c'est nécessaire : 0 pour faux et $\neq 0$ pour vrai.

• Une liste d'instructions ne peut pas être vide : le retour (c'est–à–dire la sortie) d'un programme se fait nécessairement par une instruction **retourne**.

• Il peut arriver, pour certaines valeurs des données, que l'exécution n'aboutisse à aucune instruction de retour : ceci est le fruit d'une erreur grave dans le programme mais qui n'est pas décelable pendant la traduction. La P–instruction `ERREUR` qui doit figurer à la fin de toute traduction permet d'interrompre l'exécution dans ce cas.

Unités lexicales de Galileo.

- Chaque mot–clef forme une unité lexicale à soi seul.
- Il en est de même pour les symboles de ponctuation, les parenthèses, ... (le symbole d'affectation comportant deux caractères.
- **nb** : `[-+]?[0-9]+`
- **id** : `[A-Za-z][A-Za-z0-9]*`
- **opadd** : les opérateurs additifs `+`, `ou`
- **opmult** : les opérateurs multiplicatifs `*`, `et`
- **oprel** : les opérateurs relationnels `=`, `>=`, ...

La grammaire et cette description des unités lexicales constituent proprement la définition du langage Galileo.

Modifications en vue de la traduction.

- Il est commode d'introduire une unité lexicale **affect** ne comportant que le symbole d'affectation `:=` comme élément.
- Des nouvelles variables, que nous noterons M, $Malors$, $Msinon$, $Mtantque$ et $Mfaire$, sont nécessaires lors de la traduction, pour appliquer la "méthode de reprise".

Voici les modifications qu'il faut apporter à la grammaire pour son utilisation par Yacc :

Modifications de la grammaire pour Yacc

Règles modifiées.

9 : $Li \rightarrow Li\ M\ I$
10 : $I \rightarrow V$ **affect** E
11 : $I \rightarrow$ **si** $E\ Malors\ Li$ **fin**
12 : $I \rightarrow$ **si** $E\ Malors\ Li\ Msinon\ Li$ **fin**
13 : $I \rightarrow Mtantque\ E\ Mfaire\ Li$ **fin**

Marqueurs.

32 : $M \rightarrow$;
33 : $Malors \rightarrow$ **alors**
34 : $Msinon \rightarrow$ **sinon**
35 : $Mtantque \rightarrow$ **tantque**
36 : $Mfaire \rightarrow$ **faire**

Présentation du P–code.

Un P–code se présente sous la forme d'un fichier typé : c'est une liste de P–instructions, dont la plupart agit directement sur **la pile d'exécution**.

• Une P–instruction comporte un opérateur (représenté par un mnémonique de quelques lettres, qui devra être codé par un entier) et éventuellement un argument.

• On dispose de "registres" :

`CP` (Pointeur de Code)
`SP` (Sommet de la Pile)

• Le registre `CP` est l'adresse, dans le P–code, de l'instruction à exécuter.

Les instructions d'un *P*–code s'exécutent séquentiellement, à partir de la première.

La pile d'exécution.

La pile d'exécution est désignée comme une partie initiale d'un tableau `PILE` : l'indice de son sommet est la valeur du registre `SP`.

• Le principe d'exécution d'une opération sur une pile est bien connu :

- les arguments de l'opération sont empilés au sommet, dans l'ordre correspondant à la déclaration de cette opération,
- l'exécution de l'opération a pour effet global de dépiler les arguments et d'empiler le résultat, si l'opération en produit un.

• CONSEIL. Il faut que vous compreniez parfaitement le mode d'action de chaque P–instruction avant de définir vos règles sémantiques!

Les P–instructions.

Opérations arithmétiques.

`ADD` : (E, E) → E	`PILE[SP - 1] := PILE[SP - 1] + PILE[SP]` `SP := SP - 1 CP := CP + 1`
`SUB` : (E, E) → E	`PILE[SP - 1] := PILE[SP - 1] - PILE[SP]` `SP := SP - 1 CP := CP + 1`
`MUL` : (E, E) → E	`PILE[SP - 1] := PILE[SP - 1] * PILE[SP]` `SP := SP - 1 CP := CP + 1`
`NEG` : E → E	`PILE[SP] := - PILE[SP]` `CP := CP + 1`

Opérations booléennes.

`ET` : (E, E) → E	`PILE[SP - 1] := PILE[SP - 1] et PILE[SP]` `SP := SP - 1 CP := CP + 1`
`OU` : (E, E) → E	`PILE[SP - 1] := PILE[SP - 1] ou PILE[SP]` `SP := SP - 1 CP := CP + 1`
`LNEG` : E → E	`PILE[SP] := non PILE[SP]` `CP := CP + 1`

Expressions booléennes.

`EQU` : (E, E) → E	`PILE[SP - 1] := PILE[SP - 1] = PILE[SP]` `SP := SP - 1 CP := CP + 1`
`GEQ` : (E, E) → E	`PILE[SP - 1] := PILE[SP - 1] >= PILE[SP]` `SP := SP - 1 CP := CP + 1`
`LEQ` : (E, E) → E	`PILE[SP - 1] := PILE[SP - 1] <= PILE[SP]` `SP := SP - 1 CP := CP + 1`
`LES` : (E, E) → E	`PILE[SP - 1] := PILE[SP - 1] < PILE[SP]` `SP := SP - 1 CP := CP + 1`
`GRT` : (E, E) → E	`PILE[SP - 1] := PILE[SP - 1] > PILE[SP]` `SP := SP - 1 CP := CP + 1`

NEQ : $(\mathrm{E},\mathrm{E}) \rightarrow \mathrm{E}$	PILE[SP - 1] := PILE[SP - 1] <> PILE[SP] SP := SP - 1 CP := CP + 1

Adressage et transfert de données.

LDC p : vide $\rightarrow$ E	SP := SP + 1 PILE[SP] := p CP := CP + 1
LDO p : vide $\rightarrow$ E	SP := SP + 1 PILE[SP] := PILE[p] CP := CP + 1
STO : $(\mathrm{E},\mathrm{E}) \rightarrow$ vide	PILE[PILE[SP - 1]] := PILE[SP] SP := SP - 2 CP := CP + 1
MOV : E $\rightarrow$ E	PILE[SP] := PILE[PILE[SP]] CP := CP + 1

Test de domaine.

CHK p : E $\rightarrow$ E	si (PILE[SP] < 0 ou PILE[SP] $\geq$ p) alors erreur('hors du domaine') sinon CP := CP + 1 fin

Entrées et sorties.

PECRIRE : E $\rightarrow$ vide	écrire l'entier PILE[SP] à l'écran SP := SP - 1 CP := CP + 1
PLIRE : vide $\rightarrow$ E	SP := SP + 1 PILE[SP] := entier lu à l'écran CP := CP + 1

Branchement.

UJP p	CP := p
FJP p : E $\rightarrow$ vide	si PILE[SP] = 0 alors CP := p sinon CP := CP + 1 fin SP := SP - 1

Initialisation.

INIT	SP := -1 CP := CP + 1

Réservation de place.

RES p : vide $\rightarrow$ (U,...,U)	SP := SP + p CP := CP + 1

Fin de programme.

RET	retourne (en TP : halt(0))
ERREUR	erreur('sortie imprévue') (en TP : halt(1))

Quelques indications.

Voici quelques indications utiles pour écrire vos programmes Lex et Yacc : la séparation des tches attribuées à l'analyse lexicale de celles qui sont attribuées à l'analyse syntaxique est importante.

La table des symboles.

La table des symboles est définie par les déclarations suivantes :

```
type  symbole =  record
                      nom :  string ;
                      dim :  integer ;
                     depl :  integer
                 end ;

var     table :  array[1..TMax] of symbole ;
     longueur :  integer ;
```

`function ranger ( s : string ) : boolean ;`

Cette fonction tente de ranger la chane de carctères `s` dans la `table` :
- si `s` est déjà présente dans la `table` alors elle renvoie `false` et ne la modifie pas,
- sinon, elle renvoie `true` et range effectivement `s` dans la `table` :

```
longueur := longueur + 1 ;
table[longueur].namestr := s ;
```

`function rechercher ( s : string ; var place : integer ) : boolean ;`

Cette fonction recherche la chane de caractères `s` dans la `table` :
- si `s` en est absente alors elle renvoie `false`,
- sinon, elle donne à `place` la valeur correspondant à la position de `s` dans la `table` et renvoie la valeur `true`.

`var recherche : boolean ;`

Cette variable a la valeur `false` pendant toute la partie "déclaration" du programme : la fonction `ranger` est utilisée. Elle devient `true` pendant la partie "exécution" du programme : c'est alors la fonction `rechercher` qui est utilisée.

Pendant la partie "déclaration" du programme, on doit réserver la place, au début de la `PILE` pour chacune des variables (une case pour une variable de type entier, n cases pour une variable de type tableau [n], comme le montre l'exemple suivant)

```
variables
    x1 : entier
    x2 : tableau[4]
    x3 : entier
    x4 : entier
    ...
fin
```

0 :	`x1`	`depl( x1 ) = 0`
1 :	`x2[0]`	`depl( x2 ) = 1`
2 :	`x2[1]`	
3 :	`x2[2]`	
4 :	`x2[3]`	
5 :	`x3`	`depl( x3 ) = 5`
6 :	`x4`	`depl( x4 ) = 6`
etc.	...	

Le nombre de cases (composante `dim`) et la valeur (calculée) du déplacement (composante `depl`) doivent être enregistrés dans la `table` : cette opération est exécutée par l'appel de

`procedure Completertable ( place , dimension : integer ) ;`

Le P–code.

Un P–code est une liste de P–instructions qui doit impérativement se présenter sous la forme d'un fichier typé de la façon suivante :

```
type  pinstruction =  record
                                operation :  integer ;
                                 argument :  longint
                      end ;

            liste =  ^cliste ;
           cliste =  record
                                  adresse :  longint ;
                                  suivant :  liste
                      end ;


var      pcode : file of pinstruction ;
```

Les P–opérations doivent impérativement être considérées comme étant les constantes :

`ADD = 1 ; SUB = 2 ; MUL = 3 ; ... ; RET = 25 ; ERREUR = 26 ;`

Le P–code, c'est–à–dire le fichier assigné à variable `pcode`, est obtenu en utilisant les fonctions et les procédures décrites ci–dessous :

`procedure em ( voperation : integer ; vargument : longint ) ;`
: Ecrit la P–instruction `voperation  argument` à la position courante du fichier assigné à la variable `pcode`.

`procedure em1( voperation : integer ) ;`
: Ecrit la P–instruction`voperation`, sans argument, à la position courante du fichier assigné à la variable `pcode`.

Au moment où l'on écrit une P–instruction de saut, l'adresse de destination n'est en général pas encore connue : ceci explique les variables de "marquage" (*M*, *Malors*, ...) qui ont été introduites dans la grammaire Yacc de *Galileo*. Les fonctions et la procédure suivantes permettent de faire les adressages corrects par des petits retours en arrière appelés "reprises".

```
function position : longint ;
    begin
        position := filepos ( pcode )
    end ;
```

Renvoie la position courante.

`function liste0 : liste ;`
: Renvoie une liste vide.

`function liste1 ( p : longint ) : liste ;`
: Renvoie une liste dont le seul élément est `p`.

`function concatener ( l1 , l2 : liste ) : liste ;`
: Renvoie la concaténée de `l1` et `l2`.

`procedure reprendre ( var l : liste ; vadresse : longint ) ;`
: Ecrit l'adresse `vadresse` comme argument des P–instructions de saut dont la position est dans la liste `l`.

Extensions possibles : pour les rapides et les courageux.

• Les majuscules et les minuscules. Tel qu'il vous est proposé, le programme Lex produira un analyseur lexical qui fait la distinction entre les majuscules et les minuscules. Plus précisément :

- les mot–clefs devront être écrits en minuscules : si vous en écrivez un avec une seule majuscule, il sera reconnu comme un identificateur ordinaire,
- deux identificateurs qui ne diffèrent que par la hauteur de certains de leurs caractères, seront considérés comme étant distincts.

Vous pouvez accepter ces principes (très contraignants lorsque l'on écrit un programme), ou les refuser : dans ce dernier cas, il vous faudra modifier votre analyseur lexical en conséquence. Si vous désirez traiter votre table de symboles de façon efficace, vous pouvez vous inspirer du fichier PASLEX.L qui figure dans les environs de TPlex.

• Les entrées et les sorties. La définition du langage *Galileo* qui vous a été donnée ne permet que la lecture et l'écriture d'entiers. Or, il est souvent utile d'afficher des messages à l'écran pour "dialoguer" avec l'utilisateur. Je vous propose l'extension suivante de notre langage :

- introduire un nouveau mot–clef **ecrirec** et une nouvelle règle syntaxique $I \rightarrow$ **ecrirec** (**car**) où **car** est l'unité lexicale de tous les caractères, écrits sous l'une des formes classiques `'c', \nnn, etc` ... (attention : on veut que `c` puisse, par exemple, être un passage à la ligne).
  Une instruction `ecrirec( 'c' )` signifiera "écrire **le** caractère `c` à l'écran".
- introduire une nouvelle P–instruction `PECRIREC p` où `p < 256` est un entier, et qui aura comme effet d'écrire **le** caractère de code ASCII `p` à l'écran.

Si vous faites cette extension, vous devrez impérativement :

- coder `PECRIREC` comme étant la constante 27,
- commencer vos P– codes par `INIT 1` (au lieu de `INIT` tout court),

et gérer entièrement vos affichages à l'écran par vous–même.

LE FICHIER LEX : GALLEX.L

```
%{
   (* METTEZ VOS NOMS ET VOS GROUPES DE TD ICI, EN COMMENTAIRE ...*)

      var code : integer ;              (* pour la procédure standard Val *)
      function ranger ( s : string ) : boolean ; ...
      function rechercher ( s : string ; var place : integer ) : boolean ; ...

   (* Les variables table et longueur sont déclarées dans GAL.Y
      On pourrait écrire une seule fonction jouant à la fois le rle de ranger
      et celui de rechercher ...
      Les autres déclarations sont faites dans GAL.Y :
      l'analyseur lexical que Lex va produire à partir du présent programme
      c'est-à-dire la fonction yylex
      n'est qu'un sous programme de celui que va produire Yacc ...*)
%}
%%
[ \t\n]                ;
programme              return( programme ) ;
variables              return( variables ) ;
...                    ...etc ...
ou                     begin yylval.yyinteger := OU ; return( opadd ) end ;
...                    ...etc ...
'='                    begin yylval.yyinteger := EQU ; return( oprel ) end ;
...                    ...etc ...
[-+]?[0-9]+            begin
                          Val(yytext, yylval.yyinteger, code) ;
                          if code = 0 then return( nb )
                          else begin
                             (* message d'erreur *) ; halt
                          end
                       end ;
[A-Za-z][A-Za-z0-9]*   (* voir NOTE ci-dessous *)
':='                   return( affect ) ;
'-'|'('|')'|
'['|']'|':'|';'        returnc( yytext[1] ) ;
.                      begin
                          (* message d'erreur *) ; halt
                       end ;
   /* NOTE : ranger ou rechercher yytext, suivant la circonstance.
      En cas de succès : return( id ) et yylval.yyinteger := la place où se trouve
      yytext dans la table, sinon : (* message d'erreur *) et halt ! */
%%
(* Ainsi se termine GALLEX.L *)
```

LE FICHIER YACC : GAL.Y

```
%{
  (* METTEZ VOS NOMS ET VOS GROUPES DE TD, EN COMMENTAIRE ...*)
     uses crt , dos , yacclib , lexlib ;
  (* IL FAUT IMPERATIVEMENT CODER LES 26 P-OPERATIONS DE CETTE FACON : *)
     Const
     ADD = 1 ;       SUB = 2 ;       MUL = 3 ;       NEG = 4 ;       ET = 5 ;
     OU = 6 ;        LNEG = 7 ;      ...
     RET = 25 ;      ERREUR = 26 ;    (* et éventuellement PECRIREC = 27 *)
  (* Déclarations relatives à la table des symboles. *)
     Const
     TMax = ...                      (* la longueur maximale de la table *)
     Type
          symbole =  record
                                  nom :  string ;
                                  dim :  integer ;
                                 depl :  integer
                     end ;
     Var
            table :  array[1..TMax] of symbole ;
         longueur :  integer ;
        recherche :  boolean ;
     procedure completertable ( place , dimension : integer ) ; ...
  (* Les fonctions ranger et rechercher sont déclarées dans GALLEX.L *)
  (* Déclarations relatives au fichier de P-code :
     IL FAUT IMPERATIVEMENT RESPECTER LA DEFINITION DU TYPE pinstruction,
     IL FAUT IMPERATIVEMENT RESPECTER LA DEFINITION DE LA VARIABLE pcode. *)
     Type
     pinstruction =  record
                            operation :  integer ;
                             argument :  longint
                     end ;
            liste =  ^cliste ;
           cliste =  record
                              adresse :  longint ;
                              suivant :  liste
                     end ;
     Var
             pcode : file of pinstruction ;

     procedure em ( voperation : integer ; vargument : longint ) ;
     (* émettre une P-instruction à la fin du fichier assigné à pcode *)
     var v : pinstruction ;
     begin
         v.operation := voperation ;
         v.argument := vargument ;
         write( pcode , v )
     end ;

     procedure em1 ( voperation : integer ) :
     begin em( voperation , 0 ) end ;

     function position : longint ;
     begin position := filepos ( pcode ) end ;

     function liste0 : liste ; ...
```

```
     function liste1 ( p : longint ) : liste ; ...
     function concatener ( l1 , l2 : liste ) : liste ; ...
     procedure reprendre ( var l : liste ; vadresse : longint ) ; ...
%}
%token          <integer>        programme        retourne         variables
                                 id               entier           tableau
                                 nb               si               alors
                                 sinon            tantque          faire
                                 ecrire           lire             fin
                                 oprel            opadd            opmult
                                 non              affect

%type           <integer>        Lid              Ty               V
                                 E                ES
                                 T                F

%type           <longint>        M                Malors           Msinon
                                 Mtantque         Mfaire

%type           <liste>          I                Li


%%
P          :    programme Dv Li fin            {begin
                                     reprendre( $3, position ) ;
                                     em1(ERREUR)
                                          end ; }
           ;
Dv         :    /* vide*/                 {recherche := true ; }
           |    variables Lid fin           {...}
           ;
Lid        :    id ':' Ty                 {begin
                                     completertable( $1 , $3 ) ;
                                     em( RES , $3 )
                                          end ; }
           |    Lid id ':' Ty             {...}
           ;
Ty         :    entier                    {$$ := 1 ; }
           |    tableau '[' nb ']'          {begin
                                     if $3 < 1 then begin
                                         (* message d'erreur *) ; halt
                                     end ;
                                     $$ := $3
                                          end ; }
           ;
Li         :    I                         { ; }
           |    Li M I                    {begin
                                     reprendre( $1 , $2 ) ; $$ := $3
                                          end ; }
           ;
I          :    V affect E                {begin
                                     em1( STO ) ; $$ := liste0
                                          end ; }
           |    si E Malors Li fin          {
                                     $$ := concatener( liste1( $3 ) , $4 )
                                          ; }
           |    si E Malors Li Msinon Li fin{...}
           |    Mtantque E Mfaire Li fin   {...}
```

```
          |    ecrire '(' E ')'          {begin
                                    em1( PECRIRE ) ; $$ := liste0
                                         end ; }
          |    retourne                  {...}
          ;
V         :    id                   {if table[$1].dim = 1 then
                                        em( LDC , table[$1].depl ) ;
                                    else begin
                                        (* message d'erreur *) ; halt
                                         end ; }
          |    id '[' ES ']'             {begin
                                    em( CHK , table[$1].dim ) ;
                                    em( LDC , table[$1].depl ) ;
                                    em1( ADD )
                                         end ; }
          ;
E         :    ES                        {...}
          |    ES oprel ES               {em1( $2 ) ; }
          ;
ES        :    T                         {...}
          |    ES opadd T                {em1( $2 ) ; }
          |    ES '-' T                  {...}
          ;
T         :    F                         {...}
          |    T opmult F                {...}
          ;
F         :    '(' E ')'                 {...}
          |    '-' F                     {em1( NEG ) ; }
          |    non F                     {em1( LNEG ) ; }
          |    id '[' ES ']'             {begin
                                    em( CHK , table[$1].dim ) ;
                                    em( LDC , table[$1].depl ) ;
                                    em1( ADD ) ; em1( MOV )
                                         end ; }
          |    lire                      {em1( PLIRE ) ; }
          |    id                   {if table[$1].dim = 1 then
                                        em( LDO , table[$1].depl )
                                    else begin
                                        (* message d'erreur *) ; halt
                                         end ; }
          |    nb                        {em( LDC , $1) ; }
          ;
M         :    ';'                       {$$ := position ; }
          ;
Malors    :    alors                     {begin
                                    $$ := position ; em1( FJP )
                                         end ; }
          ;
Msinon    :    sinon                     {...}
          ;
Mtantque  :    tantque                   {...}
          ;
Mfaire    :    faire                     {...}
          ;
 %%
```

```
{$I GALLEX }                                 (* inclut l'analyseur lexical *)
  (* Déclarations utiles pour récolter le nom du fichier à traduire
     et pour construire le nom du fichier qui contiendra la traduction. *)
     Var                                  (* types définis dans l'unité Dos *)
           fichier : pathstr ;
        repertoire : dirstr ;
               nom : namestr ;
         extension : extstr ;

begin
    clrscr ;
    if paramcount = 0 then begin
        writeln( 'Commande : GAL nomDufichier[.GAL]' ) ;
        halt( 1 )
    end
    else begin
        fichier := fexpand( paramstr( 1 ) ) ;
        fsplit( fichier , repertoire , nom , extension ) ;
        if nom + extension = '' then begin
            writeln( 'Commande : GAL nomDufichier[.GAL]' ) ;
            halt( 1 )
        end ;
        if extension = '' then extension := '.GAL' ;
        {$I-}
        assign( yyinput , repertoire + nom + extension ) ;
        reset( yyinput ) ;
        {$I+}
        if IOresult <> 0 then begin
            write( 'Impossible d''ouvrir le fichier : ' ) ;
            writeln( repertoire + nom + extension ) ;
            halt( 1 )
        end
    end ;
    {$I-}
    assign( pcode , repertoire + nom + '.PCO' ) ;
    rewrite( pcode ) ;
    {$I+}
    if IOresult <> 0 then begin
        writeln( 'Impossible d''ouvrir un fichier pour la traduction.' ) ;
        halt( 1 )
    end ;   (* si l'on est arrivé ici : les fichiers sont bien ouverts ...*)
    longueur := 0 ;
    recherche := false ;
    em1( INIT ) ;                              (* début obligé d'un P-code *)
    if yyparse = 0 then writeln( 'Traduction terminée.' ) ;
end.
(* ainsi se termine GAL.Y ...*)
```