Gilles HUNAULT

An 2001

galg, manuel

du Programmeur

Université d'Angers

Table des matières

Chapitre 1.

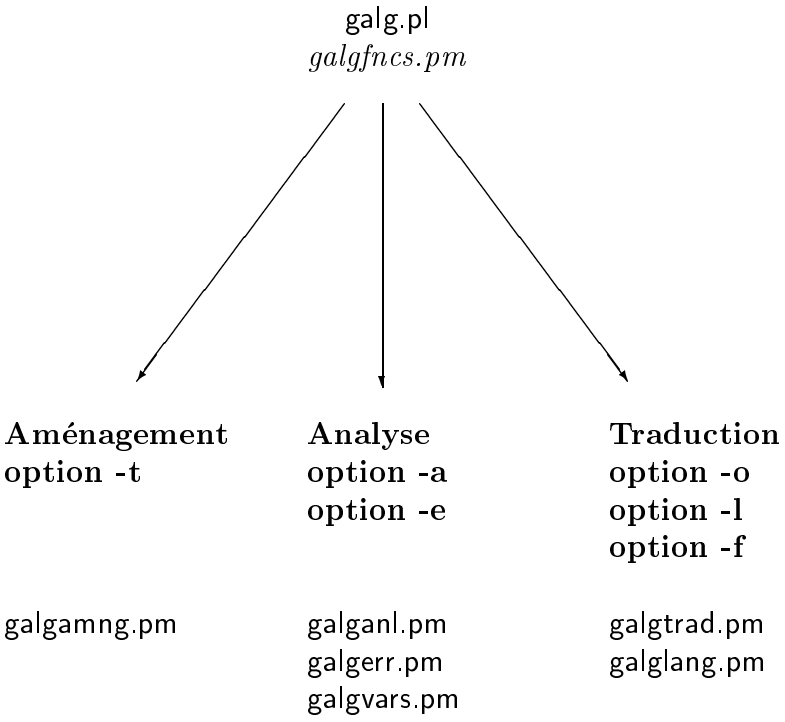
Programme principal : galg.pl

1.1 Organisation générale

Le programme galg se compose d'un programme principal `galg.pl` et de sept modules généraux, à savoir, par ordre alphabétique :

- `galgamng.pm`
  pour le mode aménagement (option `-t`),
- `galganl.pm`
  pour le mode analyse (option `-a`),
- `galgerr.pm` qui contient la liste des erreurs détectées à l'analyse et les fonctions liées aux erreurs (option `-e`),
- `galgfncs.pm`
  qui contient des fonctions générales principalement de traitement de chaines de caractères,
- `galglang.pm` qui contient des fonctions générales pour les langages, les fonctions de post-comparaison et post-traduction et l'affichage des listes de correspondances internes (options `-l` et `-f`),
- `galgtrad.pm`
  pour le mode traduction (option `-o`),
- `galgvars.pm`
  qui gère les listes de modules, variables et tableaux.

De façon plus visuelle, on peut séparer les modules généraux en trois groupes, selon le schéma suivant :



De plus galg utilise un module spécifique par langage pour la traduction (option -o). Ce sont, par ordre alphabétique, les modules :

- `galg2c.pm` pour le langage C,
- `galg2cpp.pm` pour le langage C++,
- `galg2dbase.pm` pour le langage Dbase,
- `galg2java.pm` pour le langage Java,
- `galg2pascal.pm` pour le langage Pascal,
- `galg2perl.pm` pour le langage Perl,
- `galg2rexx.pm` pour le langage Rexx,
- `galg2tcl.pm` pour le langage Tcl/Tk.

1.2 Déroulement du programme galg.pl

Le programme principal galg.pl se contente de charger les modules généraux, de tester les divers paramètres et d'appeler les fonctions correspondantes. Il contient le sous-programme `Aide` qui rappelle la syntaxe de galg.

La fonction `galgcpy` qui est appelée par le sous-programme d'aide, en cas d'erreur sur les paramètres et en fin de programme contient le rappel du copyright, l'email de l'auteur, l'adresse *Web* de la documentation. La fonction `galgcpy` est dans le module `galgfncs.pm`.

galg commence par tester tous les paramètres avant de lancer les sous-programmes correspondants aux options associées aux paramètre. Chaque option de traitement des fichiers a sa variable, à savoir :

`$modeAmng`	pour gérer l'option `-t` (Aménagement)
`$modeAnl`	pour gérer l'option `-a` (Analyse)
`$modeTrad`	pour gérer l'option `-o` (Traduction)
`$modeExec`	pour gérer l'option `-x` (Exécution)

Lorsque la variable d'option est à 1, l'action correspondante est à exécuter ; sinon, elle vaut 0.

Il y a de plus une variable nommée `$modePostc` pour gérer le mode post-comparaison qui est une des étapes de la traduction.

On peut schématiser le déroulement du programme principal par l'appel des sous-programmes suivants :

`galglang::initNomLangages`	dans tous les cas
`galgerr::listeDesErreurs`	pour l'option `-e`
`galglang::listeLangages`	pour l'option `-l`
`galglang::fonctionsDuLangage`	pour l'option `-f`
`galgamng::amenageTexte`	pour l'option `-t`
`galganl::analyseAlgorithme`	pour l'option `-a`
`galgtrad::TradAlg`	pour l'option `-o`
`galglang::postTraduction`	pour l'option `-o`
`galglang::postCompare`	pour l'option `-o`

Comme pour de nombreuses fonctions et sous-programmes de galg, il y a une variable de "debug". Pour le programme principal, elle se nomme `$dbg_Galg`. Comme pour les autres variables de debug, si sa valeur est 0, il n'y a aucun affichage de debug. Si elle vaut 1, il y a de nombreux affichages de debug, en particulier avant et après les appels principaux des fonctions correspondants aux options.

Le déroulement de galg peut être divisé en parties, facilement repérables dans le texte par des "commentaires en boite" ; ce sont les parties :

1 - vérification des paramètres

2 - aménagement ou analyse

3 - traduction et exécution éventuelle

Pour la partie 1, *vérification des paramètres*, s'il n'y a pas de paramètre, on appelle la fonction `Aide`. Sinon, si le premier paramètre ne commence pas par un tiret, on dit que ce n'est pas possible car toute option commence par un tiret et on s'arrête.

galg appelle alors la fonction `galglang::initNomLangages` puis teste ensuite les options `-e`, `-l` et `-f`. Si on rencontre une de ces options, on exécute la fonction correspondante et on s'arrête. Sinon, on vérifie que le paramètre un correspond soit à `-t` soit à `-a` qui sont les deux seules options principales possibles.

On initialise ensuite les variables `mode*` déjà présentées puis les variables suivantes

`$nomLang`	(nom du langage à utiliser si traduction)
`$nbParmRst`	(nombre de paramètres à transmettre si exécution)
`$parmRst`	(ligne des paramètres à transmettre si exécution)
`$codeRet`	(code retour final de galg)
`$main::nbErreurs`	(nombre global d'erreurs)

L'intérêt d'un code retour est de permettre à des scripts de tester si l'exécution de galg s'est bien passée.

Lorsque l'option `-t` est sélectionnée, galg vient vérifier qu'il y a bien un paramètre pour le nom du fichier à aménager et qu'il n'y a pas d'autre paramètre sur la ligne de commande. Sinon, on dit qu'il y a une erreur, on appelle la fonction `galgcpy` et on quitte galg.

Lorsque l'option `-t` est sélectionnée, galg vérifie que le nom du fichier-algorithme à analyser a bien été passé en paramètre.

Pour l'option `-t` comme pour l'option `-a`, on vient ensuite vérifier que l'identificateur du fichier (variable `$idFourni`) à traiter comporte bien une partie "nom" et une partie "extension" reliés par un point, ce qui permet d'affecter à la variable `$baseNomFichier` la partie nom du fichier grâce à la fonction `nomFichierSansExtension` du module gamlgfncs.pm.

galg vérifie ensuite les paramètres restants sur la ligne de commande : on doit d'abord trouver `-o` puis le nom d'un langage connu, ce qu'on teste en regardant si le tableau associatif `%tabLang` a une entrée pour la variable `$nomLang` qui correspond au paramètre qui suit `-o` (le tableau `%tabLang` a été initialisé précédemment par `galglang::initNomLangages`).

La partie 1 de galg se termine par la récupération de la liste (éventuellement vide) des paramètres restants si l'option `-x` est sélectionnée. Puisqu'on a utilisé l'option `-a` avec le nom du fichier à analyser, puis l'option `-o` avec le nom du langage et l'option `-x` on doit commencer la liste (variable `$parmRst`) à partir du paramètre numéro 5 (car Perl commence les paramètres à l'indice 0).

La partie 2, *aménagement ou analyse*, commence par mettre dans la variable `$nomFichierAlg` l'identificateur du fichier algorithme à traiter. Elle se compose ensuite de deux tests successifs afin de savoir si on est en mode aménagement ou en mode analyse. Pour l'aménagement, on vérifie que le nom du fichier à traiter (variable `$idFourni`) n'est pas le nom du fichier algorithme à produire et si ce n'est pas le cas, galg exécute la commande

```
&galgamng::amenageTexte($idFourni,$nomFichierAlg) ;
```

avant d'indiquer le nom du fichier algorithme à consulter.

Pour l'analyse, galg exécute la commande

```
($nomLst,$varLst,@lignesAlg)
            = &galganl::analyseAlgorithme($nomFichierAlg) ;
```

avant d'indiquer le nom des fichiers produits à consulter, à savoir :

- le listing numéroté (variable `$nomLst`),
- la liste des variables, tableaux et modules (`$varLst`).

La variable `$main::nbErreurs` contient le nombre d'erreurs détectées. S'il y a eu des erreurs à l'analyse, elles sont affichées avec la ligne d'algorithme incriminée grâce au tableau `@$main::tabErr` et on met le code retour de galg (variable `$codeRet`) à -1.

Enfin, la partie 3, *traduction et exécution éventuelle*, qui n'est n'est exécutée que s'il n'y a pas d'erreurs, commence par préparer le nom du programme dans la variable `$main::nomDuProgramme` en utilisant la correspondance entre nom du langage et extension du fichier fournie par le tableau associatif `$main::extLang` qui a été initialisé par `galglang::initNomLangages`.

La traduction est exécutée par `&galgtrad::TradAlg` et passe en paramètres

- la liste des modules (variable `$lstModules`),
- la liste des variables (variable `$lstVars`),
- la liste des tableaux (variable `$lstTableaux`),
- le nom de base des fichiers (variable `$baseNomFichier`),
- le nom du langage (variable `$nomLang`),
- la valeur du mode exécution (variable `$modeExec`),
- la valeur du mode post-comparaison (variable `$modePostc`),
- le nom du fichier de correspondance externe (variable `$fichierCorresp`),
- le tableau des lignes de l'algorithme (variable `@lignesAlg`).

Il y a ensuite deux appels de fonctions. Le premier est

```
&galglang::postTraduction($main::nomDuProgramme,$nomLang) ;
```

pour réaliser une post-traduction éventuelle (c'est à dire un remplacement de certaines expressions avec la table de correspondance interne) et le second est

```
&galglang::postCompare(
          $main::nomDuProgramme,$fichierCorresp,$nomLang) ;
```

pour réaliser une post-comparaison éventuelle (c'est à dire un remplacement de certaines expressions avec la table de correspondance externe contenue dans le fichier dont l'identificateur est `$fichierCorresp`).

L'exécution du programme se fait en réalisant appel au système d'exploitation grâce à l'instruction

```
$retExec = system($cmd)
```

La variable `$cmd` est composée à partir du nom du script à exécuter (expression `$cmdLang{$nomLang}`, initialisée par `galglang::initNomLangages`), du nom du programme (variable`$main::nomDuProgramme`) et des paramètres éventuels (variable `$parmRst`).

Juste d'avant d'exécuter la commande de la variable `$cmd`, galg affiche la commande et une ligne de symboles "=" de même longueur. Une autre ligne de de symboles "=" est affiché après l'exécution de la commande.

galg se termine alors en affichant le copyright avec la fonction `galgcpy` et renvoie le code retour renvoyé par l'appel système.

Afin de présenter les divers affichages, nous incluons ici ce qui apparait à l'écran lors de la traduction en Rexx et de l'exécutiond'un algorithme contenu dans leTest.alg et qui ne contient, en dehors des commentaires, que l'instruction `\ECRIRE " ceci est un test"` ; la commande utilisée est bien sur `galg -a leTest.alg -o rexx -x` :

```
galg (gH) 2001 ; Gestion d'ALGorithmes

-a : Analyse du fichier-algorithme leTest.alg
      001  |           # auteur : gh
      002  |  001      ECRIRE " ceci est un test"

vous pouvez consulter les fichiers :

     leTest.lst (listing numéroté)
     leTest.lvm (liste des variables et modules)

Algorithme sans erreur détectée.

-o : Traduction du fichier-algorithme leTest.alg en rexx

-x : Exécution du fichier produit : leTest.rex

Exécution de la commande  regina   leTest.rex
================================================
ceci est un test
================================================

Copyright  2001  - email : gilles.hunault@univ-angers.fr
                   http://www.info.univ-angers.fr/pub/gh/

Documentation      http://www.info.univ-angers.fr/pub/gh/vitrine/Galg.htm
```

Si on met la variable de debug, `$dbg_Galg`, à 1, l'affichage est beaucoup plus long. Il permet de vérifier comment les paramètres ont été récupérés, de voir explicitement le nom des fichiers mis en jeu, de savoir quand se passent la traduction, la post-traduction, l'exécution :

```
   ===> 5 paramètres fournis à savoir :

    paramètre   1  : -a
    paramètre   2  : leTest.alg
    paramètre   3  : -o
    paramètre   4  : rexx
    paramètre   5  : -x


galg (gH) 2001 ; Gestion d'ALGorithmes

le nom du fichier fourni est leTest.alg
la base des noms de fichiers est donc leTest
   ===> demande de Traduction en  Rexx (implémentation : regina 2.0)

   ===> mode Exécution demandé.

-a : Analyse du fichier-algorithme leTest.alg

      001  |           # auteur : gh
      002  |  001      ECRIRE " ceci est un test"
   ===> fin d'analyse

vous pouvez consulter les fichiers :

     leTest.lst (listing numéroté)
     leTest.lvm (liste des variables et modules)

Algorithme sans erreur détectée.

-o : Traduction du fichier-algorithme leTest.alg en rexx

   ===> fin de traduction

   ===> début de post-traduction
   ===> fin   de post-traduction
```

```
-x : Exécution du fichier produit : leTest.rex

Exécution de la commande  regina   leTest.rex
================================================
ceci est un test
================================================
   ===> fin d'exécution

Copyright  2001  - email : gilles.hunault@univ-angers.fr
                   http://www.info.univ-angers.fr/pub/gh/

Documentation      http://www.info.univ-angers.fr/pub/gh/Galg.htm
```

Chapitre 2.

Mode Analyse : galganl.pm

2.1 Grandes lignes de l'analyse

Le programme principal (fichier galg.pl) commence par vérifier la validité des différents paramètres passés en ligne de commande. Si les paramètres sont valides, il lance, lorque le paramètre `-a` est présent, l'analyse du fichier algorithme. Cette analyse se fait à l'aide du sous-programme `analyseAlgorithme` du fichier galganl.pm et prend comme paramètre l'identificateur complet du fichier algorithme (avec l'extension `.alg` en fin d'identificateur).

Elle exécute les étapes suivantes :

- initialisation des listes de mots clés, de symboles, des codes-erreurs,
- mise en mémoire des lignes du fichier dans le tableau `@lignesAlg`,
- élimination des dernières lignes de l'algorithme si elles sont vides,
- construction des noms de fichiers de listage et des variables associés à l'algorithme (variables `$nomLst` et `$varLst`),
- parcours des lignes de l'algorithme une à une avec détection des erreurs immédiates,
- détection d'erreurs non flagrantes lorsque toutes les lignes de l'algorithme ont été analysées,
- mise en forme de la ligne d'algorithme avec numéro d'instruction et de profondeur

Le nom du fichier de listage reprend le nom du fichier algorithme mais avec l'extension .lst alors que le nom du fichier des variables, modules et tableaux reprend le nom du fichier algorithme avec l'extension .lvm.

Par exemple, au fichier-algorithme `Matrices/produit.alg` l'analyse de l'algorithme associe les fichiers `Matrices/produit.lst` et `Matrices/produit.lvm`.

Les listes utilisées par galg pour tester les mots clés, les conditions... sont des chaines de caractères contenant des mots écrits en majuscules et séparés par des espaces. Elles sont initialisées par la fonction `initListes` du module galganl.pm. Ces listes et les variables associées sont les suivantes :

`$lddI` : liste des débuts d'instruction qui n'imbriquent pas

```
affecter appeler écrire écrire_ écrire_sur fermer
lire lire_sur ouvrir ouvrir_lec ouvrir_ecr quitter
```

`$lddP` : liste des débuts d'instruction à profondeur (ou "qui imbriquent")

```
pour répéter repeter répeter repéter si tant_que
```

`$ldmI` : liste des milieux d'instruction

```
alors sinon
```

`$ldfI` : liste des fins d'instruction

```
fin_si fin_pour fin_tant_que jusqu'à jusqu'a
```

`$main::masqI` : expression régulière de construction des identificateurs

```
^[_a-zA-Z]+[a-zA-Z_0-9]*\$
```

`$ldmdC` : liste des mots de condition

```
< = > <= >= <> ET OU NON
```

`$ldmdCo` : liste précédente convertie en interne

```
< = > {  }  !  ;  :  ~
```

`$ldmdCop` liste précédente en opérateurs mono-caractères

```
<=>{}!;:~
```

La variable `$ldpmI` contient la liste des premiers mots invalides après ALORS/SINON. C'est la concaténation des listes `$lddP`, `$ldmI` et `$ldfI`.

La variable `$ldpmV` contient la liste des premiers mots valides. C'est la concaténation des listes `$lddI`, `l$ddP`, `$ldmI`, `$ldfI`, `$ldmdC` et `$ldmdCo`.

Un identificateur (de variable simple, de tableau, de fonction...) peut donc commencer par le caractère de soulignement. En principe, seules les fonctions doivent l'utiliser pour indiquer qu'il faut utiliser les tables de correspondance internes.

L'analyse vérifie qu'il est possible d'ouvrir le fichier `.alg` en lecture et les fichiers `.lst` et `.lvm` en écriture (ce qui n'est pas le cas si on analyse des fichiers-algorithmes dans des répertoires pour lesquels on n'a pas de droit d'écriture comme par exemple les fichiers sur Cdrom...).

Le parcours du tableau des lignes de l'algorithme vérifie qu' il y a bien le mot "auteur" dans les 5 premières lignes dans le but de forcer à mettre des commentaires en début de fichier (si on a écrit "auteur", on peut bien rajouter quelques mots pour commenter ce que fait l'algorithme...). Ensuite, dans ce parcours on décompose chaque ligne d'instruction en trois parties :

- une partie **instruction** avec un mot-clé unique (variable `$mu`) qui est souvent le premier mot (ou mot "un") de la ligne,
- une partie **paramètres** de l'instruction, éventuellement vide ou composée de plusieurs mots (variable `$finIns`),
- une partie **variable de fichier**, éventuellement vide. (variable `$varFic`)

Cette décomposition en assurée par le sous-programme `decoupeLigneAlgo`.

Voici quelques exemples de telles parties, une fois le découpage terminé :

Instruction	*Paramètres*	*Variable de fichier*
`affecter`	`x <-- 2`	
`si`	`(a=5)`	
`lire`	`x`	
`lire_sur`	`x , y`	`matent`
`fermer`		`matent`
`quitter`		

Chaque instruction est repérée par un mot géré en majuscules et encadré par un espace de début et un espace de fin ce qui permet de le chercher dans la liste des instructions qui est une liste de mots écrits en majuscules et séparés par des espaces.

Le nom de l'instruction courante est dans la variable `$insCour` et le nom de l'instruction précédente est dans la variable `$insPrec`. Il faut se méfier car le mot un n'est pas forcément le nom de l'instruction en cours. Par exemple dans `ALORS ECRIRE x` le mot un est `ALORS` mais l'instruction est `ECRIRE` (et l'instruction précédente `SI`).

Ce découpage oblige parfois à donner comme nom d'instruction un mot autre que le mot un de l'instruction. Ainsi le découpage de l'instruction `LIRE x SUR f` vient nommer `LIRE_SUR` l'instruction et attribue `x` comme seul paramètre de l'instruction et `f` comme variable de fichier.

Une fois le découpage assuré, il y a vérification du mot un via le sous-programme `gereMotUn` puis une vérification que le reste de l'instruction est correct avec le sous-programme `valideInstruction` et enfin un ajout éventuel de la variable de fichier grâce au sous-programme `ajouteVarFic`.

Il ne reste alors qu'à afficher la ligne bien présentée par le sous-programme `formateLigne` avec affichage du code erreur et du texte de l'erreur lorsqu'une erreur est détectée. Une fois arrivé à la dernière ligne du tableau des lignes de l'algorithme, on passe en revue les erreurs possibles suivantes :

- n'avoir aucune instruction dans l'algorithme (erreur `059`),
- ne pas avoir de nom d'auteur dans les 5 premières lignes (erreur `057`),
- ne pas finir toutes les structures `SI`, `POUR`... (erreur `058`).

On peut donc schématiser le déroulement du sous-programme `analyseAlgorithme` par la suite des appels

- `decoupeLigneAlgo`
- `gereMotUn`
- `valideInstruction`
- `galgvars::ajouteVarFic`

avec le déclenchement éventuel du sous-programme

- `galgerr::erreur`.

L'analyse de l'algorithme se termine par l'affichage des listes de modules, variables et tableaux et un test d'homonymie des variables grâce aux sous-programmes

- `galgvars::afficheListeVariables`
- `galgvars::afficheListeVarFic`
- `galgvars::afficheListeTableaux`
- `galgvars::afficheListeModules`
- `galgvars::testeHomonymie`

2.2 Le sous-programme decoupeLigneAlgo

Tout d'abord, il faut noter que ce sous-programme `decoupeLigneAlgo` du module galganl.pm est également utilisé aussi par le sous-programme `TradAlg` dans la phase de traduction gérée par le module galgtrad.pm.

Il reçoit comme paramètre la ligne courante à analyser. Il commencer par séparer le premier mot (variable `$motUn`) du reste de la ligne (variable `$resteDeLaLigne`) à l'aide du sous-programme `premierEtReste` du module galgfncs.pm.

Il y a quatre cas particuliers à traiter : celui de `FERMER`, celui de `OUVRIR..COMME` et enfin ceux de `ECRIRE... SUR` et `LIRE... SUR` qui peuvent être traités ensemble.

Le sous-programme commence par gérer le cas FERMER qui est le plus simple : après `FERMER` il doit y avoir un seul mot qui devient la variable de fichier (identificateur `$varF`). La partie "reste de la ligne" est alors la chaine vide.

On traite ensuite le cas de `OUVRIR` : si l'avant dernier mot (variable `$avdm`) n'est pas `COMME`, on a affaire à l'erreur `055`. Si c'est bien `COMME`, la variable de fichier (identificateur `$varF`) est obtenue comme le dernier mot du reste de la ligne grâce au sous-programme `dernierMot` du module galgfncs.pm. Il reste à savoir s'il faut ouvrir en lecture ou en écriture. Pour cela, on commence par restreindre les paramètres de l'instruction en ne conservant que la sous-chaine avant le mot `COMME` dans la chaine `$resteDeLaLigne`. Si le dernier mot est `EN_LECTURE` ou `EN_ECRITURE` alors l'instruction est `OUVRIR_LEC` ou `OUVRIR_ECR` et l'identificateur du fichier est le premier mot du reste de la ligne. Sinon, c'est l'erreur `056`.

Enfin, on traite le cas des instructions `ECRIRE` et `LIRE`. Si l'avant-dernier mot (variable `$avdm`) est `SUR` alors les instructions `ECRIRE` et `LIRE` doivent etre renommées en `ECRIRE_SUR` et `LIRE_SUR`, la variable de fichier (identificateur `$varF`) est obtenue comme le dernier mot du reste de la ligne grâce au sous-programme `dernierMot` du module galgfncs.pm. Les paramètres de l'instruction sont alors obtenus en ne que gardant la chaine `$resteDeLaLigne` que la sous-chaine avant le mot `SUR`.

Voici quelques exemples d'instructions et de découpage associé :

- l'instruction `AFFECTER abc <-- 10` est découpée en :

instruction	`AFFECTER`
paramètre(s)	`abc <-- 10`
variable de fichier	`""`

- l'instruction `ECRIRE " x vaut " , x` est découpée en :

instruction	`ECRIRE`
paramètre(s)	`" x vaut " , x`
variable de fichier	`""`

- l'instruction `FERMER ficmat` est découpée en :

instruction	`FERMER`
paramètre(s)	`""`
variable de fichier	`ficmat`

- l'instruction `OUVRIR "ventes.janvier" EN_LECTURE COMME ficVentes` est découpée en :

instruction	`OUVRIR_LEC`
paramètre(s)	`"ventes.janvier"`
variable de fichier	`ficVentes`

- l'instruction `LIRE mois , an SUR ficMois` est découpée en :

instruction	`LIRE_SUR`
paramètre(s)	`mois , an`
variable de fichier	`ficMois`

- l'instruction `ECRIRE " -- " , date() SUR ficMois` est découpée en :

instruction	`ECRIRE_SUR`
paramètre(s)	`" -- " , date()`
variable de fichier	`ficMois`

2.3 Le sous-programme gereMotUn

Le premier test qu'effectue le sous-programme `gereMotUn` c'est de vérifier que le mot le mot un est valide (sinon, on a l'erreur `002`) grâce à la position de ce mot un (variable `$mu`) dans la liste des premiers mots valides (variable `$ldpmV`). Si le mot un est dans la liste (variable `$lddP`) des structures ou "instructions emboitantes" comme `SI`, `POUR`... on incrémente la variable `$insNf` des instructionss non finies. Comme dépasser 3 niveaux d'imbrication est considéré comme une erreur (plus que comme une faute de gout), si la variable `$insNf` est plus grande que trois, on déclenche l'erreur `017`.

Le listage des instructions ne numérote que les instructions principales, et donc on n'incrémente le numéro d'instruction (variable `$nbIns`) que si on n'est pas déjà dans une instruction emboitante. Pour garder une trace du type de structure, on met dans la variable `$curStru` l'initiale du mot un et on garde dans la variable `$lastStru` la liste des ces initiales. `POUR` est donc gardé comme `P`, `TANT_QUE` comme `T` etc.

La structure `SI` est spéciale puisqu'elle met en jeu une partie `ALORS` et puis éventuellement une partie `SINON` c'est pourquoi on garde dans la variable `$mdmP` (mot du milieu de pour) la suite des initiales de `SI` seulement.

Si le mot un fait partie de la liste (variable `$lddI`) des débuts d'instructions et si on n'est pas à l'intérieur d'une instruction non finie, il s'agit d'une nouvelle instruction à traiter et donc on incrémente la variable `$nbIns`.

On teste ensuite si le mot un indique une fin de structure, ce qui arrive dans deux cas : soit le mot un est `JUSQU'A` et le type de structure est `R` (pour `REPETER`) soit le mot est dans la liste (variable `$ldfI`) des fins d'instructions (qui commencent toutes par `FIN_`) auquel cas le type de structure est la première lettre après le caractère de soulignement, c'est à dire la lettre en position 5. Finir une structure alors qu'une structure plus interne n'est pas finie est une erreur, de code `016`.

Gérer une fin de structure consiste simplement à éliminer des chaines structures courantes (variables `$lastStru` et `$mdmP`) le dernier caractère.

Le sous-programme `gereMotUn` effectue ensuite quelques tests particuliers pour l'instruction `SI` à cause des cas particulier de `ALORS` et `SINON` : les mots `ALORS` et `SINON`, repérés parce que le mot un est dans la liste (variable `$ldmI`) des milieux d'instruction, sont bien placés si la structure courante (dernier caractère de la chaine `$lastStru`) est `S` ; dans le cas contraire, il s'agit de l'erreur `024`. Une autre erreur, de code `021` est de ne pas mettre de `ALORS` après un `SI` ; de même, remettre un `ALORS` après un `ALORS` est l'erreur `045`, de même, remettre un `SINON` après un `SINON` est l'erreur `046`.

Enfin, le sous-programme `gereMotUn` teste des erreurs qui sont plutot des quelques fautes de gout :

- si la première instruction a lieu en ligne 1, il n'y a pas de commentaire en début, et on considère qu'il s'agit de l'erreur `001`,
- une instruction `ECRIRE` doit précéder une instruction `LIRE` (ce qui se teste avec les variables `$insCour` et `$insPrec`) sinon, c'est l'erreur `003`,
- une fin de structure non commentée est considérée comme une erreur, de code `012` pour la structure `POUR`, `013` pour la structure `SI`, `014` pour la structure `TANT_QUE`.

2.4 Le sous-programme valideInstruction

Le sous-programme `valideInstruction` reçoit trois paramètres, à savoir et dans cet ordre :

- le nom de l'instruction
- le corps (ou "paramètres") de l'instruction
- la variable de fichier associée à l'instruction

Ce sous-programme commencer par éliminer les espaces de début et de fin grâce à la fonction `sansEspaceDebutEtFin` du module galgfncs.pm pour le nom et le corps de l'instruction. Si l'instruction se termine par un point-virgule, on déclenche l'erreur `048`. Sinon, on commence par tester si les guillemets dans la partie paramètres de l'instruction sont bien en nombre pair grâce à la fonction `testeGuillemets` du module galgfncs.pm, sinon, c'est l'erreur `026`. Pour éviter de mauvaises surprises, on remplace les chaines de caractères constantes par la valeur 0 puis on teste le parenthésage et le crochetage des chaines avec les fonctions `oteChainesCst`, `testeParenthesage` et `testeCrochetage` toutes trois dans le module galgfncs.pm.

Si l'expression possède plus de deux niveaux de parenthèses imbriquées, il s'agit de l'erreur `038` et une expression mal parenthèsée déclenche l'erreur `018`. Une expression mal crochetisée déclenche l'erreur `027` ; de plus galg considère qu'emboiter les crochets est une erreur (de code `041`) et qu'utiliser directement un appel de fonction ou une expression parenthèsée comme indice dans les crochets est également une erreur (de code `042`).

S'il n'y a pas de problème de guillemets, de parenthèses et de crochets, le sous-programme `valideInstruction` calcule ensuite le nombre de mots (variable `$nbm`) du corps de l'instruction aménagé dans la variable `$rdp` (comme reste des paramètres). S'il reste quelquechose après les mots `FIN_SI`, `FIN_POUR` ou `FIN_TANT_QUE`, on déclenche l'erreur `036`.

`valideInstruction` vérifie ensuite que pour les mots de milieu d'instruction `ALORS` et `SINON` on ne retrouve que des instructions simples (sinon, c'est l'erreur `019`) et en tous cas des instructions connues(sinon, c'est l'erreur `050`). S'il n'y a pas d'erreur on effectue un nouvel appel à `valideInstruction` pour le reste de l'instructions après le mot de milieu (le découpage ayant été assuré par la fonction `premierEtReste` du module galgfncs.pm.

`valideInstruction` analyse ensuite les instructions `LIRE` et `LIRE_SUR`. La variable `$rdp$` voit ses virgules "aérées" c'est à dire encadrées d'espaces grâce à la fonction `aereVirgules` du module galgfncs.pm ce qui permet de calculer correctement le nombre de mots suivant le nom de l'instruction. Il faut au moins un mot sinon c'est l'erreur `005`, mais pas de tableau (erreur `25`) ni fonction (erreur `20`) mais pas plus d'un mot (erreur `06`) car nous encourageons les algorithmes à lire dans les fichiers une variable de type caractère pour la découper ensuite en les variables voulues.

Comme pour `LIRE`, on aére les virgules mais aussi les crochets (avec la fonction `aereCrochets` du module galgfncs.pm) et on calcule le nombre de mots. Il en faut au moins 1, sinon c'est l'erreur `004`. L'expression à écrire est ensuite transmise à la fonction `gereListeExpressions` du même module (galganl.pm) que `valideInstruction`.

`valideInstruction` teste alors l'instruction `POUR`. Pour cela, grâce à la fonction `premierEtReste` on sépare l'identificateur de la boucle du reste de la ligne, on teste cet identificateur grâce à la fonction `testeIdentificateur` puis on sépare les valeurs de début et de fin de boucle avec la fonction `decoupePour`. Ces valeurs de début et de fin devant correspondre à une expression unique, elles sont testées par la fonction `testeUneSeuleExpression`.

Le validation suivante est celle des conditions dans les instructions `SI`, `TANT_QUE` et `REPETER` grâce à la fonction `testeCondition` mais à cause du langage Rexx qui veut la condition dès la ligne du mot `REPETER` on incrémente, pour cette instruction seulement, la variable `$main::NbRep` qui contient le nombre de structures `REPETER` et ont met dans le tableau `@$main::globRep` la condition correspondante.

Enfin, pour tester l'affectation et l'appel de modules, `valideInstruction` fait appel à des fonctions spécialisées du du même module (galganl.pm) nommées respectivement `valideAppeler` et `valideAffecter`.

Chapitre 3.

Mode Traduction : galgtrad.pm et galg2*.pm

3.1 Principe général de traduction

Le module galgtrad.pm qui gère la traduction commence par charger trois modules généraux de galg, à savoir

1. `galgfncs` pour les fonctions sur chaines de caractères
2. `galgvars` pour la gestion des variables
3. `galglang` pour la gestion générale des langages

Puis il charges les modules spécifiques aux langages, c'est à dire

- `galg2c.pm` pour le langage C ;
- `galg2cpp.pm` pour le langage C++ ;
- `galg2dbase.pm` pour le langage Dbase ;
- `galg2java.pm` pour le langage Java ;
- `galg2pascal.pm` pour le langage Pascal ;
- `galg2perl.pm` pour le langage Perl ;
- `galg2rexx.pm` pour le langage Rexx.
- `galg2tcl.pm` pour le langage Tcl/Tk.

galgtrad.pm charge aussi le module standad `FileHandle` utile pour la gestion du fichier de sortie correspondant au programme à écrire.

La traduction est effectuée par le sous-programme `TradAlg` qui traduit ligne par ligne en séparant chaque ligne d'instruction en ses différentes composantes et en appelant une fonction qui correspond au nom de l'instruction. Ainsi l'instruction `ECRIRE` appelle la fonction `Ecrire` qui elle-même appele la fonction correspondante dans le langage choisi (`Ecrire_rexx`, `Ecrire_perl`...).

3.2 Le sous-programme TradAlg

`TradAlg` commence par récupérer les différents paramètres passés par le programme principal, à savoir :

- la liste des modules (variable `$::LModules`),
- la liste des variables (variable `$::LVar`),
- la liste des tableaux (variable `$::LTab`),
- le nom de base des fichiers (variable `$NomFichSansExt`),
- le nom du langage (variable `$Lang`),
- la valeur du mode exécution (variable `$::modeExec`),
- la valeur du mode post-comparaison (variable `$::modePostc`),
- le nom du fichier de correspondance externe (variable `$nfpc`),
- le tableau des lignes de l'algorithme (variable `@TLigneAlg`).

`TradAlg` définit ensuite la chaine `_£_` comme séparateur de données (variable `$::SeparDon`), vérifie que le langage est connu grâce à la fonction `testeNomLangage` du module galglang et vérifie aussi qu'il est possible d'écrire dans le fichier-programme.

Si c'est le cas, on commence par écrire dans le fichier-programme de sortie qu'il s'agit d'une traduction automatique par galg. Voici ce qui est produit pour la traduction en Rexx de l'algorithme `bonjour_Rex03.alg` :

```
/* ##-#   Fichier bonjour_Rex03.rex issu de galg -a bonjour_Rex03.alg -o rex */
/* ##-#  ================================================================== */
/* ##-#      16/07/2001 19:29.40 */
```

La date et l'heure sont affichées grâce au sous-programme `gereLeTemps` du module galgfncs.pm. Les commentaires sont écrits suivant le modèle

```
&{"Commentaire_".$Lang}(...) ;
```

et comme la variable `$Lang` contient le nom du langage, c'est l'instruction du module spécifique au langage qui est appelée (toutes les fonctions des modules spécifiques aux langages sont "exportées" ce qui évite de les préfixer par le nom du module).

Comme il arrive que les dernières lignes d'un algorithme soient vides, `TradAlg` les élimine par une boucle "tant_que" sur les dernières lignes du tableau. La traduction véritable est alors effectuée par une boucle "pour" sur chacune des lignes du tableau de l'algorithme `@TLigneAlg`.

Dans cette boucle, on transfère par l'instruction

```
$ligAlgCour = $TLigneAlg[$IndL] ;
```

la ligne du tableau en cours dans la variable `$ligAlgCour`.

Lorsqu'on détecte la première ligne qui n'est pas un commentaire (ce qui inclut le cas d'une ligne blanche ou vide), on exécute la fonction `DebutProgramme` pour le langage. Cette fonction est vide pour des langages interprétés comme Rexx ou Perl mais elle contient `PROGRAM ...` pour le Pascal, `int main() {` pour le C... La détection se fait en regardant le premier caractère de la ligne, qui doit être `#` pour un commentaire et à l'aide de la variable `$insereDebut` qui vaut 0 tant qu'on n'a pas inséré le début de programme.

On ne doit pas mettre le début de programme en début de traduction car en C notamment les sous-programmes peuvent être mis avant le programme principal. Le fait de mettre le début après les premiers commentaires permet d'inclure avec les commentaires spéciaux `#:` et `#>` des instructions, des paramétrages propres au langage avant le programme principal comme par exemple `#include <stdio.h>` pour le C.

La ligne courante est ensuite découpée via l'instruction

```
($Indent, $LCourante)=&IndentEtReste($TLigneAlg[$IndL]) ;
```

ce qui permet de respecter dans le programme l'indentation de l'algorithme (la fonction `IndentEtReste` provient du module galgfncs.pm).

Si la ligne courante est un commentaire spécial d'inclusion ligne ou fichier, elle est gérée par la fonction correspondante, soient les instructions

```
if ( &cmtInclusionFichier($TLigneAlg[$IndL])==1 ) {
    &inclusionFichier($TLigneAlg[$IndL])

} elsif ( &cmtInclusionLigne($TLigneAlg[$IndL])==1 ) {
    &inclusionLigne($TLigneAlg[$IndL])  }
```

les fonctions `cmtInclusionFichier`, `InclusionFichier`, `cmtInclusionLigne` et `InclusionLigne` étant définies à la fin du module galgtrad.pm.

S'il ne s'agit pas d'un commentaire spécial, on découpe la ligne en une partie commentaire et une partie instruction à l'aide des instructions

```
$LSansCmt = &sansCmt($LCourante);
$LSansCmt = &sansEspaceDebutEtFin($LSansCmt) ;
$Comment  = &Cmt($LCourante);
```

où les fonctions `sansCmt`, `sansEspaceDebutEtFin` et `Cmt` proviennent du module galgfncs.pm.

S'il ne s'agit pas d'un commentaire, on découpe alors la chaine d'instruction grâce à la fonction `decoupeLigneAlgo` du module galganl.pm en

- une partie **instruction** avec un mot-clé unique (variable `$Instruction`) qui est souvent le premier mot (ou mot "un") de la ligne,
- une partie **paramètres** de l'instruction, éventuellement vide ou composée de plusieurs mots (variable `$ResteL`),
- une partie **variable de fichier**, éventuellement vide. (variable `$var_Fi`)

puis le nom de l'instruction est alors mis sous une forme standard (initiale majuscule, le reste en minuscule, les mots reliés par le caractère de soulignement) via la fonction `MizANormInstr` contenue dans le module galgtrad.pm et l'instruction spécifique au langage correspondant est ensuite appelée par l'instruction

```
&$Instruction(&sansEspaceDebutEtFin($ResteL),$var_Fi) ;
```

on rajoute ensuite en fin de ligne la partie commentaire, si elle n'était pas vide à l'aide de la fonction commentaire spécifique par l'instruction

```
&{"Commentaire_".$Lang}($Comment) ;
```

et on finit la boucle "pour" en mettant un saut de ligne à la fin de la ligne traduite dans le fichier programme par

```
print $::SymbFLang "\n" ;
```

`TradAlg` vient enfin écrire la partie spécifique en fin de programme (souvent vide pour des langages interprétés comme Rexx ou Perl mais qui contient `END.` pour le Pascal, `} // fin de main` pour le C... puis ferme le fichier programme et indique qu'on peut le consulter s'il ne doit pas y avoir exécution.

3.3 Le sous-programme MizANormInstr

Comme indiqué précédemment, le sous-programme `MizANormInstr` a pour but de normaliser l'écriture des noms d'instructions puisque ceux-ci peuvent être écrits en majuscules, minuscules ou en une combinaison des deux. `MizANormInstr` traduit tout en majuscules à l'aide la fonction `maju` du module galgfncs.pm puis remplace chaque nom d'instruction par un nom d'instruction normalisé ce qui signifie : chaque mot est en minuscule mais avec l'initiale en majuscule, les mots sont reliés par le caractère de soulignement). Par exemple, `TANT_QUE` est normalisé en `Tant_Que`.

3.4 Traduction des instructions

Pour traduire un commentaire, la fonction `Commentaire` se réduit à l'instruction

```
&{"Commentaire_".$Lang}($_[0]) ;
```

c'est à dire qu'elle se contente de transmettre le texte du commentaire à la fonction spécifique du langage qui viendra mettre le ou les symboles de commentaire là où il faut.

Par contre pour la fonction `Affecter`, il faut d'abord repérer la variable à gauche du symbole d'affectation (variable `$VarG`) et la partie à droite de l'affectation (variable `$ParD`) ce qui se fait à l'aide de la position (variable `$PosAff`) de la chaine `<--` dans l'instruction d'affectation. On élimine dans `$VarG` et `$ParD` les espaces de début et de fin de chaine via la fonction `sansEspaceDebutEtFin` du module galgfncs.pm puis on prépare les deux parties de l'expression en on appelant respectivement `GereTerme` (contenue dans le module galgtrad.pm) et `GereExpression` (également contenue dans le module galgtrad.pm) avant d'exécuter l'instruction spécifique d'affectation au langage :

```
&{"Affecter_".$Lang}($VarG, $ParD) ;
```

Les appels des fonctions `GereTerme` et `GereExpression` sont obligatoires pour mettre éventuellement des symboles devant les variables (comme `$` en Perl et Tcl/Tk), pour mettre les bons symboles d'opération et d'appels de fonctions...

La traduction de `ECRIRE`, `ECRIRE_` et `ECRIRE_SUR` aménage le paramètre grâce à la fonction `GerePhrase` du module galgtrad.pm alors que la traduction des instructions `LIRE` et `LIRE_SUR` utilise le paramètre traité par la fonction `GereVariable` du module galgtrad.pm.

Pour traduire `OUVRIR_LEC` et `OUVRIR_ECR`, si le nom du fichier à ouvrir est une chaine (ce qu'on teste à l'aide de la fonction `EstChaine` contenue dans le mo-

dule galgtrad.pm), on enlève les guillements avant d'appeler dans tous les cas la fonction spécifique du langage.

`FERMER` par contre ne pose aucun problème : on se contente d'appeler la fonction spécifique du langage avec le paramètres fournis (il y en a toujours deux car une instruction comporte toujours une partie paramètres de l'instruction et une partie variable de fichier, éventuellement vide, l'une ou l'autre).

Quand on traduit `APPELER`, le paramètre est géré par `GereFonction` contenue dans le module galgtrad.pm.

L'instruction `POUR` est décomposée entrois parties (variables `$IndB`, `$Deb`, et `$Fin`) de façon à gérer l'identificateur de l'indice de boucle avec la fonction `GereVarSimple` et les deux expressions de début et fin avec la fonction `GereTerme`.

La fin de `POUR` notée `Fin_Pour` ne présente aucune difficulté, comme `Fin_Si` et `Fin_Tant_Que` puisqu'il n'y a aucun paramètre à transmettre contrairement à `SI` et `TANT_QUE` qui doivent gérer la condition passée en paramètre avec la fonction `GereExpression`.

Comme `ALORS` et `SINON` peuvent contenir une instruction complète, on redécompose le paramètre (qui est le reste de la ligne courante après le mot clé) avec la fonction `premierEtReste` du module galgfncs.pm en une partie instruction (variable `$Instruction`) et une partie "reste de la ligne" (variable `$ResteL`). On vient alors mettre à la norme le nom de l'instruction avec la fonction `MizANormInstr` du module galgtrad.pm avant de relancer l'appel de l'instruction avec les paramètres débarassés des espaces en début et fin de chaine avec `sansEspaceDebutEtFin`.

La traduction de l'instruction `REPETER` consiste seulement à appeler la fonction spécifique correspondante du langage alors que la traduction de `JUSQU'A` oblige à gérer la condition par la fonction `GereExpression`.

La dernière instruction autorisée, `QUITTER` est traduire en appelant la fonction spécifique du langage avec comme paramètre celui fourni par l'algorithme. géré par `GereExpression` car le code-retour (qui est un nombre) peut être transmis via une variable.

Chapitre 4.

Mode aménagement : galgamng.pm

Le mode aménagement est géré par la fonction `amenageTexte` du module galgamng.pm. Cette fonction prend comme paramètres et dans cet ordre l'identificateur du fichier texte à analyser (variable `$nomFichierTexte`) et l'identificateur du fichier algorithme à produire (variable `$nomFichierAlg`). Elle se réduit, après avoir vérifié qu'il était possible de lire dans le premier fichier et d'écrire dans le second, à un parcours du fichier associé à `$nomFichierTexte` avec lecture ligne par ligne (variable `nLigne`) et modification éventuelle de cette ligne par les fonctions

- `ajoutEventuel_Affecter`,
- `ajoutEventuel_Appeler`,
- `amenageMotsClefs`.

Ces trois fonctions sont également dans le module galgamng.pm. La fonction `amenageTexte` se termine par la fermeture des deux fichiers.

La fonction `ajoutEventuel_Affecter` se contente de tester si la chaine `"<--"` est présente dans la ligne. Si c'est le cas, on rajoute le mot `affecter` après les espaces blancs de début de façon à respecter l'indentation originale du fichier-texte.

De même, pour la fonction `ajoutEventuel_Appeler`, si le premier mot de la ligne contient une parenthèse ouvrante, on rajoute `appeler` en début de ligne.

Comme galg autorise la présence d'une instruction non imbriquante dans le `ALORS` et le `SINON` de la structure `SI`, il faut éventuellement permuter les mots un et deux des lignes après l'ajout éventuel, comme on le voit sur l'exemple suivant :

Instruction avant ajout d'affecter	`"alors x <-- 2"`
Instruction après l'ajout	`"affecter alors x <-- 2"`

La fonction `permuteMotsUnEtDeuxPourAlorsSinon`, qui est la dernière fonction du module galgamng.pm, réalise justement cette permutation éventuelle pour les fonctions `ajoutEventuel_Affecter` et `ajoutEventuel_Appeler`.

On notera que cette fonction utilise la variable globale `$main::ldmI` qui ne contient en principe que les mots `ALORS` et `SINON`. Cette variable est initialisée par la fonction `initListes` du module galganl.pm.

La troisième fonction appelée pour chaque ligne du fichier texte à aménager est la fonction `amenageMotsClefs`. Elle prend comme paramètre toute la ligne et remplace "bestialement" les différentes fins de structure possibles par une fin normalisée. Par exemple, pour la fin de la boucle "TANT QUE", on effectue les substitutions suivantes :

Chaine lue	Chaine écrite
`fin tant que`	`fin_tant_que`
`fintantque`	`fin_tant_que`
`Fin Tant Que`	`Fin_Tant_Que`
`Fin tant que`	`Fin_tant_que`
`FinTantQue`	`Fin_Tant_Que`
`Fintantque`	`Fin_tant_que`
`FIN TANT QUE`	`FIN_TANT_QUE`
`FINTANTQUE`	`FIN_TANT_QUE`

Chapitre 5.

Les fonctions du module galgfncs.pm

Ce sont les fonctions

```
attends
avantDernierMot
chaineVide
Cmt
dernierePosition
galgcpy
gereLeTemps
IndentEtReste
maju
mot
nbMots
nomFichierSansExtension
oteChainesCst
phraseSansPremierMot
PosMotDsPhraz
premierEtReste
premierMot
sansCmt
sansEspaceAuDebut
sansEspaceDebutEtFin
testeCrochetage
testeGuillemets
testeParenthesage
```

Chapitre 6.

Les fonctions du module galgvars.pm

On peut séparer les fonctions du module galgvars.pm en trois groupes :

1. les fonctions d'ajouts de variables

   ```
   ajouteModule
   ajouteTableau
   ajouteVar
   ajouteVarEnpartiegauche
   ajouteVarFic
   proposeArite
   proposeDimension
   ```

2. les fonctions d'affichages de variables

   ```
   afficheListeModules
   afficheListeTableaux
   afficheListeVariables
   afficheListeVarFic
   ```

3. les fonctions de tests sur les variables

   ```
   testeHomonymie
   testeIdentificateur
   ```

Chapitre 7.

Les fonctions du module galgerr.pm

Ce sont les fonctions

```
codesErreur
erreur
listeDesErreurs
```

Chapitre 8.

Les fonctions du module galglang.pm

On peut séparer les fonctions du module galglang.pm en trois groupes :

1. les fonctions globales sur la gestion des langages :

   ```
   fonctionsDuLangage
   initNomLangages
   listeLangages
   testeNomLangage
   ```

2. les fonctions de post-traitement, c'est à dire de traitement après la traduction des instructions

   ```
   postComparaison
   postTraduction
   ```

3. les fonctions spécifiques aux langages d'initialisation des tables de correspondance

   ```
   InitTdc_c
   InitTdc_cpp
   InitTdc_dbase
   InitTdc_java
   InitTdc_pascal
   InitTdc_perl
   InitTdc_rexx
   InitTdc_tcl
   ```