Gilles HUNAULT

An 2001

galg, manuel

de l'Utilisateur

Université d'Angers

Table des matières

Chapitre 1.

Présentation de galg

1.1 Principe de fonctionnement

Le logiciel galg gère des fichiers textes correspondant à des algorithmes. Il peut servir à écrire et valider des algorithmes ou à les traduire dans un langage de programmation comme C, Rexx, Perl, Java... et à les exécuter dans la foulée.

On peut l'utiliser de trois façons différentes grâce à des options entrées en ligne de commande :

- *soit avec l'option -t*
  pour transformer, aménager des fichiers textes avec une syntaxe assez libre en des algorithmes avec une syntaxe déposée,
- *soit avec l'option -a*
  pour analyser et vérifier un algorithme écrit dans la syntaxe déposée, avec production d'un listage numéroté des instructions et de la liste des variables, tableaux, modules et variables de fichiers,
- *soit avec l'option -o*
  pour traduire un algorithme vérifié dans un langage de programmation ; la traduction ne sera effectuée que si l'analyse n'a détecté aucune erreur. Il est alors possible d'exécuter dans la foulée le programme *avec l'option -x* ; si sur la ligne de commande il reste d'autres paramètres ils sont au programme traduit.

Il y a trois autres options :

l'option -e qui fournit la liste des erreurs standards détectées,

l'option -l qui donne la liste des langages implémentés ainsi que les commandes associées aux langages,

l'option -f qui donne la liste pour un langage considéré des fonctions algorithmiques élémentaires traduites automatiquement.

De plus :

- galg permet de post-modifier la traduction d'un algorithme en programme, c'est à dire de remplacer une expression grace à une table de correspondance (fichier d'exension .tdc) liée au fichier algorithme,
- galg peut inclure des sous-programmes déjà validés après la traduction via le commentaire spécial `#>`, ou insérer en ligne des instructions particulières si le langage le requiert via le commentaire spécial `#:`.

1.2 Syntaxe d'appel et exemples d'utilisation

galg s'utilise en ligne de commande, dans une "fenêtre de terminal" sous *Unix*, dans une session *Dos* sous *Dos/Windows*.

galg est écrit en Perl version 5 et donc Perl doit être installé sur l'ordinateur où on utilise galg mais aucune connaissance sur Perl n'est requise. Il faut juste savoir exécuter un programme Perl.

Pour vérifier le numéro de version de Perl installé sur le système, il faut taper, en ligne de commande

```
perl -v
```

Pour les utilisateurs de *Dos/Windows*, comme pour les utilisateurs de *Linux*, si Perl n'est pas installé, on peut l'obtenir gratuitement auprès du CPAN à l'adresse :

http://www.cpan.org

On suppose de plus pour ce qui suit que la commande galg est un script qui appelle galg.pl avec le bon chemin d'accès. Par exemple, sous *Unix*, si galg.pl est dans le répertoire `~/Perl_Macros/` le script peut être

```
perl ~/Perl_Macros/galg.pl $*
```

et sous *Dos/Windows*, si galg.pl est dans `D:\Perl_Mac\`

```
@echo off
perl D:\Perl_Mac\galg.pl %1 %2 %3 %4 %5 %6 %7 %8 %9
```

galg traite des fichiers textes écrits en "clair", sans codage interne. On peut vérifier qu'un fichier est lisible et reconnu grâce à la commande cat sous *Unix*, type sous *Dos*. C'est le cas des fichiers écrit avec *vi*, *emacs*, *edit*, *notepad*.

Si par contre on utilise *Word* pour écrire les fichiers, il faut les enregistrer avec la commande "Enregistrer sous..." et choisir le format "Texte Dos Seul".

galg accepte les textes avec des lettres accentués pour les instructions mais pas pour les variables. Les instructions peuvent être écrites indifféremment en majuscules ou en minuscules ou dans une combinaison des deux.

Les noms de fonctions avec un caractère de soulignement en début comme par exemple `_date()` ont un sens particulier : elles servent à indiquer qu'on veut utiliser (via une table de correspondance interne) la fonction correspondante du langage.

Les commentaires sont repérés par le caractère # mais certains commentaires sont spéciaux et ne servent que pour la traduction. Ce sont

`#>`
: pour indiquer qu'il faut inclure le fichier texte dont le nom suit le commentaire spécial juste après la traduction,

`#:`
: pour inclure le texte qui suit le commentaire spécial directement dans le programme après traduction.

Si on n'entre en ligne de commandes que galg, le rappel succint de la syntaxe est fourni avec des exemples d'utilisation, à savoir

```
galg.pl, version 2.67 (gH) 2001 : gestion de fichiers algorithmes

syntaxe :  galg [ -t ft | -a fa [ -o lang [ -x parms ] ] ] | -e | -f [lang] ]

    -t    traite un fichier texte pour en faire un algorithme

    -a    analyse un fichier algorithme

          -o  traduit, après analyse dans le langage passé en paramètre

          -x  exécute, après traduction avec passage
              éventuel des paramètres restants.

    -e    affiche la liste des erreurs reconnues

    -l    affiche la liste des langages connus pour traduction

    -f    donne les fonctions algorithmiques reconnues par le langage choisi


exemples :  galg -e
            galg -l
            galg -f rexx
            galg -t triRapide.txt
            galg -a produitMatrices.alg
            galg -a nbParfaits.alg      -o rex
            galg -a simulation.alg      -o java -x 10 "ventes" 3.15


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
```

La liste des exemples d'utilisation est explicite : ansi, pour afficher la liste des langages, il suffit de taper

```
galg -l
```

De même, la liste des erreurs détectables est fournie par

```
galg -e
```

et pour obtenir la liste des fonctions algorithmiques reconnues par le langage Rexx, on tape

```
galg -f rexx
```

Pour aménager le fichier texte, disons maxocc.txt en le fichier maxocc.alg, on tape

```
galg -t maxocc.txt
```

L'analyse simple d'un algorithme , disons polynom.alg se fait par

```
galg -a polynom.alg
```

alors que l'analyse du même algorithme, suivie de la traduction en Rexx se fait par

```
galg -a polynom.alg -o rex
```

Enfin, si on veut exécuter la traduction en C++ de l'algorihtme ventes.alg, il faut écrire

```
galg -a ventes.alg -o cpp -x
```

S'il y a des paramètres à passer au programme, on peut les indiquer après `-x`. Par exemple, si on veut exécuter la traduction en C++ de l'algorihtme ventes.alg avec les trois paramètres `"Chemises"`, `100` et `175.50`, il faut écrire

```
galg -a ventes.alg -o cpp -x Chemises 100 175.50
```

Pour que les traductions soit exécutables, il faut toutefois créer les commandes correspondants aux choix de galg, affichés par l'option `-l`. On consultera le chapitre 6, *Installation de galg* pour plus de détails.

galg renvoie un code-retour qui vaut 0 en cas de réussite (d'aménagement, d'analyse, de traduction), ce qui permet d'inclure galg dans des scripts. On trouvera ausi dans le chapitre 6 des exemples de tels scripts.

1.3 Détail des options

Le programme galg ne fonctionne pas sans option. Il est donc obligatoire d'utiliser au moins une option. Les options -a et -o sont les deux options les plus importantes. L'option -t est annexe mais peut rendre quelques services. Les options -e, -l et -f servent à connaitre les possibilités de galg. L'option -x requiert l'installation des langages à utiliser et la création de scripts mais elle seule permet de "voir" ce que fait un algorithme qaund on exécute le programme correspondant.

L'option -a prend un fichier algorithme en entrée ; l'identificateur du fichier doit donc se terminer par les quatre caractères ".alg".

Par contre, l'option -t prend un fichier texte quelconque pour en faire un fichier algorithme. Pour cette option, le fichier proposé en entrée doit avoir une extension, c'est à dire comporter dans son identificateur un nom puis un point et des caractères ensuite, sauf les lettres .alg puisque ce sera l'identificateur du fichier produit.

On peut donc écrire

```
galg -t ventes.demo
galg -a ventes.alg
```

mais pas

```
galg -a ventes.txt
galg -t ventes
galg -t ventes.alg
galg -a ventes
```

Il faut noter que l'option -o est une sous-option de l'option -a et que que l'option -x est une sous-option de l'option -o. Il n'est donc pas possible d'écrire directement

```
galg -o ventes.alg "langage"
galg -x ventes.alg "langage"
```

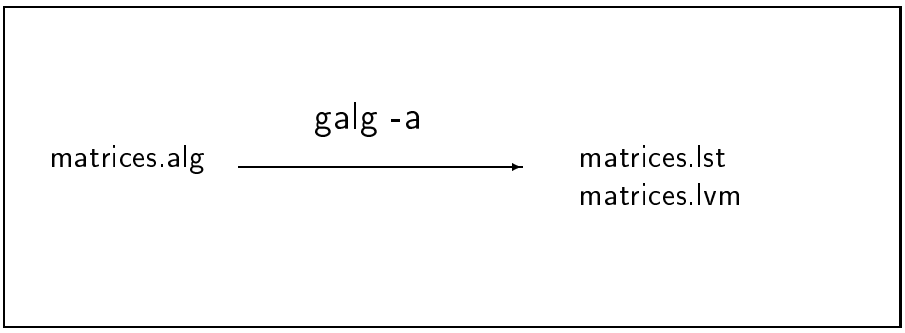
Il faut pour traduire écrire -a puis -o pour traduire et il faut écrire -a puis -o puis -x pour exécuter soit respectivement les commandes :

```
galg -a ventes.alg -o "langage"
galg -a ventes.alg -o "langage" -x [...]
```

1.3.1 ce que fait l'option -a

Lorqu'on utilise l'option -a, le programme commence par vérifier que l'identificateur de fichier proposé est bien celui d'un fichier-algorithme puis il vérifie l'existence du fichier. Le programme lit ensuite le fichier ligne par ligne, et teste chaque ligne. Le résultat de l'analyse est mis dans un fichier de même nom que le fichier algorithme mais avec l'extension .lst ; de plus un fichier avec le même nom mais avec l'extension .lvm est aussi créé, qui contient la liste des variables, des tableaux et des modules (fonctions) utilisés.

Par exemple si on analyse le fichier-algorithme `matrices.alg`, alors galg avec l'option -a produit les fichiers `matrices.lst` et `matrices.lvm`, comme indiqué sur le schéma ci-dessous :



Il faut signaler que le programme utilise le chemin d'accès du fichier d'entrée pour créer les noms des fichiers de sortie. Par exemple, si on analyse le fichier `~/Dev/Algos/matrices.alg`, les fichiers produits sont respectivement `~/Dev/Algos/matrices.lst` et `~/Dev/Algos/matrices.lvm`, ce qui est susceptible de poser problème 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.

Chaque ligne analysée est numérotée, chaque instruction aussi. Toutefois, un commentaire n'est pas considéré comme une instruction. les instructions imbriquantes comme `SI`, `POUR`, `TANT_QUE`, `REPETER` s'étendent sur plusieurs lignes avec le même numéro d'instruction.

C'est pourquoi l'algorithme suivant comporte 41 lignes pour seulement 5 instructions :

```
#########################################################
#                                                       #
# tabmult.alg -- un exemple simple d'algorithme :       #
#                la table de multiplication             #
#                                                       #
#  auteur : gh                                          #
#                                                       #
#########################################################


# demande initiale

écrire  " Donner un entier "
lire nbChoisi


# relance éventuelle

tant_que (non entier(nbChoisi))
    écrire " nombre incorrect. Redonner un nombre ENTIER "
    lire nbChoisi
fin_tant_que # nombre invalide


# boucle d'affichage

écrire " Table de " , nbChoisi
pour indb de1a 10
    affecter produit <-- nbChoisi*indb
    affecter find    <-- format(indb,2,0)
    affecter fpro    <-- format(produit,5,0)
    écrire find , " fois " , nbChoisi , " = " , fpro
fin_pour # indb de1a 10
```

Les cinq instructions avec leur numéro de ligne dans le fichier-algorithme sont :

```
Ligne   Numéro        Instruction

  021     001         écrire " Donner un entier "

  022     002         lire nbChoisi

  027     003         tant_que ( non entier(nbChoisi) )

  035     004         écrire " Table de " , nbChoisi

  036     005         pour indb de1a 10
```

Lors de l'analyse, galg détecte les variables simples et les variables de type tableau, les fonctions et sous-programmes. Voici par exemple pour l'algorithme précédent le fichier des variables, tableaux et modules :

```
Fichier tabmult.lvm issu de galg -a  tabmult.alg
================================================
  02/08/2001 23:27.26


* Liste des variables (par ordre alphabétique)

Variable                              Ligne de première     Nombre de
                                         utilisation       références
 #  find                                     38                2
 #  fpro                                     39                2
 #  indb                                     37                2
 #  nbChoisi                                 22                6
 #  produit                                  37                2

**** Liste des modules (par ordre alphabétique)

Module                   Arité        Ligne de première     Nombre de
                                         utilisation       références
 #  entier                  1                27                1
 #  format                  3                38                2

-- Fin du fichier tabmult.lvm issu de galg -a  tabmult.alg
```

Nous reproduisons également ici le listing complet de l'analyse produit par galg :

```
 Fichier tabmult.lst issu de galg -a  tabmult.alg
 ================================================
   02/08/2001 23:27.26

LIG  |  INS PRO
001  |           ########################################################
002  |           #                                                      #
003  |           #  tabmult.alg -- un exemple simple d'algorithme :     #
004  |           #  auteur : (gH) pour le manule de l'utilisateur       #
005  |           #                                                      #
006  |           ########################################################
007  |           #                                                      #
008  |           #  1. on demande à l'utilisateur un nombre entier et #
009  |           #     on le relance tant que le nombre n'est pas       #
010  |           #     entier.                                          #
011  |           #                                                      #
012  |           #  2. lorsque le nombre est entier, on afiche sa      #
013  |           #     table avec cadrage des unités sous les unités, #
014  |           #     des dizaines sous les dizaines etc.              #
015  |           #                                                      #
016  |           ########################################################
017  |
018  |
019  |           # demande initiale
020  |
021  |  001      écrire  " Donner un entier "
022  |  002      lire nbChoisi
023  |
024  |
025  |           # relance éventuelle
026  |
027  |  003 001  tant_que (non entier(nbChoisi))
028  |  003 001      écrire " nombre incorrect. Redonner un nombre ENTIER "
029  |  003 001      lire nbChoisi
030  |  003 001  fin_tant_que # nombre invalide
031  |
032  |
```

```
033  |           # boucle d'affichage
034  |
035  |  004      écrire " Table de " , nbChoisi
036  |  005 001  pour indb de1a 10
037  |  005 001      affecter produit <-- nbChoisi*indb
038  |  005 001      affecter find    <-- format(indb,2,0)
039  |  005 001      affecter fpro    <-- format(produit,5,0)
040  |  005 001      écrire find , " fois " , nbChoisi , " = " , fpro
041  |  005 001  fin_pour # indb de1a 10

 -- Fin du fichier tabmult.lst issu de galg -a  tabmult.alg
```

La troisième colonne de chiffres est la profondeur d'imbrication, c'est à dire le nombre d'instructions emboitées. Par exemple, une instruction `SI` à l'intérieur d'une instruction `POUR` est en profondeur 2. galg considére que c'est une erreur (plus exactement une faute de gout quant à la lisibilité) de dépasser la profondeur 3.

On trouvera dans le chapitre 2 *Mode "Analyse des algorithmes"* le détail de l'autre fichier produit, ce qui est affiché en cas d'erreur et ce que teste en détail galg.

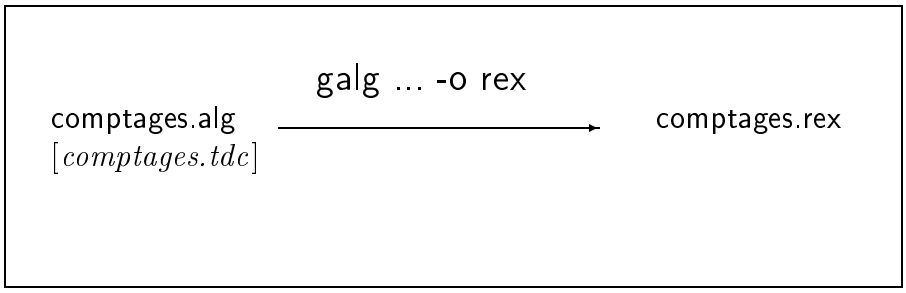
1.3.2 ce que fait l'option -o

Lorqu'on utilise l'option `-o`, `galg` commence par tester si le mot passé en paramètre après `-o` est celui d'un langage reconnu et implémenté. Il se comporte ensuite comme avec l'option `-a` et vient alors traduire ligne par ligne dans le langage considéré s'il n'y a pas eu d'erreurs d'analyse. L'identificateur du fichier-programme produit a le même nom que le fichier-algorithme mais une extension différente, celle traditionnellement utilisée avec le langage.

Par exemple si on demande la traduction du fichier `comptages.alg` en Rexx, `galg` avec l'option `-o` produit le fichier `comptages.rex` ; par contre, si on exécute la commande `galg -a comptages.alg -o cpp` le fichier programme est nommé `comptages.cpp` ; de même, si on exécute la commande `galg -a comptages.alg -o dbase` le fichier programme est nommé `comptages.prg`.

Les mêmes restrictions que pour l'option `-a` s'appliquent quant au stockage du fichier produit.

Comme il est expliqué en détail au chapitre 4, *Mode "Traduction des algorithmes"*, si une table de correspondance est trouvée, elle est est utilisée après que la traduction ait été effectuée. La table de correspondance doit avoir le même nom que le fichier-algorithme mais l'extension `.tdc` ; on peut résumer ce qu'effectue l'option `-o` dans ce cas sur le fichier précédent par le schéma suivant :



La table de correspondance n'est pas obligatoire. Le fait qu'elle n'existe pas ne provoque aucune erreur. Par contre, lorsqu'elle est utilisée, un commentaire portant la mention *"fichier post-modifié via la table de correspondance"* est inséré dans le fichier programme (voir également le chapitre 4 à ce sujet).

L'algorithme suivant affiche les dix premières puissances de deux :

```
# pddRexx.alg : puissances de deux ; auteur (gH)

affecter puiss <-- 1
pour i de1a 10
     affecter puiss <-- puiss + puiss
     écrire _format(i,2), _format(puiss,10)
fin_pour # i de1a 10
```

La commande `galg -a pddRexx.alg -o rexx` traduit l'algorithme en Rexx et fournit le fichier suivant :

```
/* ##-#   Fichier pddRexx.rex issu de galg -a pddRexx.alg -o rexx */
/* ##-#  ================================================== */
/* ##-#      08/08/2001 15:44.00 */

/* # pdd.alg : puissances de deux ; auteur (gH) */

puiss = 1
do i=1 to 10
     puiss = puiss  +  puiss
     say  format(i ,  2)  format(puiss ,  10)
end /* # i de1a 10 */

/* ##-#   Fin de traduction pour pddRexx.rex via de galg -a pddRexx.alg -o rexx */
```

On notera que la fonction notée `_format` qui assure le cadrage des nombres entiers a été traduite en `format` grâce à la table de correspondance interne.

Pour traduire en Perl il faut légèrement modifier l'algorithme : il faut inclure le fichier qui contient la définition du sous-programme `format`. L'algorithme modifié est alors :

```
# pddPerl.alg : puissances de deux ; auteur (gH)

affecter puiss <-- 1
pour i de1a 10
     affecter puiss <-- puiss + puiss
     écrire format(i,2), format(puiss,10)
fin_pour # i de1a 10

#> format.pl
```

On obtient comme fichier programme le fichier dont le contenu est :

```
##-#   Fichier pddPerl.pl issu de galg -a pddPerl.alg -o perl
##-#  ======================================================
##-#      08/08/2001 15:58.22

     # pddPerl.alg : puissances de deux ; auteur (gH)

     $puiss = 1  ;
     for ( $i = 1  ;  $i <= 10  ; $i++ ) {
          $puiss = $puiss + $puiss  ;
          print &format( $i , 2  ) ." ". &format( $puiss , 10  )."\n" ;
     } ; # i de1a 10

##-#  Ajout de format.pl via le code #>

      ## -- fichier format.pl
      sub  format  { return( sprintf("%$_[1]d",$_[0]) ) ; } ;

##-#  Fin d'ajout de format.pl via le code #>

##-#   Fin de traduction pour pddPerl.pl via de galg -a pddPerl.alg -o perl
```

Pour traduire en Pascal, il faut là encore modifier un peu l'algorithme initial : on rajoute l'inclusion du sous-programme `function`, la déclaration des variables `i` et `puiss` et le mot `BEGIN`, soit l'algorithme :

```
     # pddpascal.alg : puissances de deux ; auteur (gH)

     #> format.pas

     #: var puiss, i : integer ;

     #: BEGIN

     affecter puiss <-- 1
     pour i de1a 10
          affecter puiss <-- puiss + puiss
          écrire format(i,2), format(puiss,10)
     fin_pour # i de1a 10
```

On notera que le nom du fichier ne comporte que des minuscules car la version de Pascal utilisée (ppc386, free pascal compiler) produit des fichiers tout en minuscules.

La commande `galg -a pddpascal.alg -o pascal` traduit l'algorithme en Pascal et fournit le fichier suivant :

```
(* ##-#   Fichier pddpascal.pas issu de galg -a pddpascal.alg -o pascal *)
(* ##-#  ========================================================== *)
(* ##-#      08/08/2001 16:4.42 *)


PROGRAM pddpascal ;

     (* # pddpascal.alg : puissances de deux ; auteur (gH) *)

(* ##-#  Ajout de format.pas via le code #>  *)

function format( nombre : integer ; longueur : integer ) : string ;

{ le nombre passé en paramètre est cadré à droite }
{ avec la longueur demandée                       }

  var chaine : string ;

begin (* début de la fonction format *)

  str(nombre:longueur,chaine) ;
  format := chaine ;

end ;  (* fin de la fonction format *)

(* ##-#  Fin d'ajout de format.pas via le code #>  *)

(* ##-#  La ligne suivante provient d'un ajout via le code #:  *)
#: var puiss, i : integer ;

(* ##-#  La ligne suivante provient d'un ajout via le code #:  *)
#: BEGIN

     puiss := 1 ;
     for i := 1 to 10 do begin
          puiss := puiss  +  puiss ;
          writeln(  format(i ,  2) , format(puiss ,  10)  ) ;
     end ; (* # i de1a 10 *)

END.

(* ##-#   Fin de traduction pour pddpascal.pas via de galg -a pddpascal.alg -o pascal *)
```

Pour en finir avec l'option `-o` de traduction, voici l'algorithme des puissances de deux pour Tcl/Tk. Comme il existe une fonction nommée `format` en Tcl/Tk mais qui n'utilise pas la même syntaxe que notre fonction algorithmique, nous utilisons la fonction `_format`. Ayant une fonction `cadre` définie en Tcl/Tk dans le fichier `cadre.tcl` qui réalise ce que fait notre fonction algorithmique, nous écrivons l'algorithme :

```
# pddTcl.alg : puissances de deux ; auteur (gH)

#> cadre.tcl

affecter puiss <-- 1

pour i de1a 10
     affecter puiss <-- puiss + puiss
     écrire _format(i,2), _format(puiss,10)
fin_pour # i de1a 10
```

galg remplace tous les appels de `_format` en des appels de `cadre` grâce à la table de correspondance nommée `pddTcl.tdc` qui contient :

```
     \[_format       [cadre
```

Le fichier programme obtenu par la commande `galg -a pddTcl.alg -o tcl` est alors

```
##-#   Fichier pddTcl.tcl issu de galg -a pddTcl.alg -o tcl
##-#  ====================================================
##-#      08/08/2001 16:29.33

##-#  (fichier post-modifié via la table de correspondance pddTcl.tdc)

# pddTcl.alg : puissances de deux ; auteur (gH)

##-#  Ajout de cadre.tcl via le code #>
      proc  cadre  { nombre longueur } {
      ...
##-#  Fin d'ajout de cadre.tcl via le code #>

set puiss 1

for  { set i 1  }    { $i  <= 10  }   { set i [ incr i ] }  {
     set puiss [expr "$puiss + $puiss " ]
     puts "[cadre $i 2 ]  [cadre $puiss 10 ]  "
} ; # i de1a 10

##-#   Fin de traduction pour pddTcl.tcl via de galg -a pddTcl.alg -o tcl
```

1.3.3 ce que fait l'option -x

Pour exécuter le programme traduit par galg, l'option -x exécute la commande interne associée au langage avec comme paramètres tout ce qui suit le mot -x. La liste des commandes internes est fournie par l'option -l. Par exemple la commande associée au langage Pascal est pa, ce qui signifie que si on tape `galg -a bonjour.alg -a pascal -x "oui" 10`, galg demande au système d'exploitation d'exécuter la commande `pa bonjour "oui" 10` (voir le chapitre 6, *Installation de galg* pour plus de détails sur les commandes associées aux langages)..

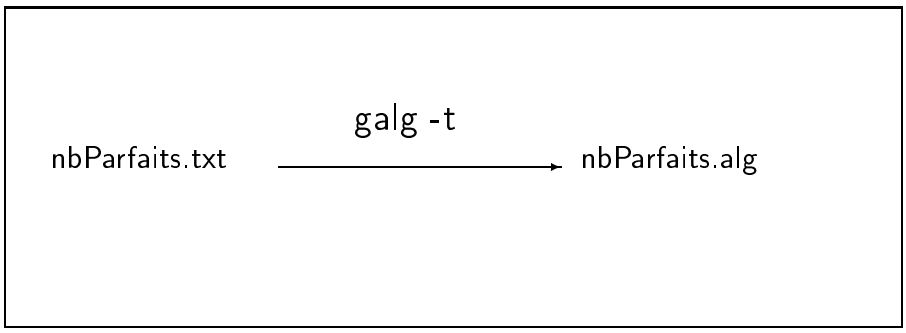
La commande est en général un script exécutable qui teste les erreurs éventuelles de pré-compilation, de compilation, de pré-interprétation etc. Par exemple la commande pa peut être un script écrit en Rexx (implémentation *Regina* 2.0) dont un contenu possible est

```
#!/usr/bin/regina

/* ce script teste que la compilation d'un programme PASCAL est */
/* ok avant de lancer l'exécution et de lister les fichiers de  */
/* même nom ; le compilateur est ppc386 (free pascal compiler). */

parse arg fn prms

if words(fn)=0 then do
   say "     syntaxe : pa nomfic parms "
   say "     exemple : pa demo oui 10 "
   say "(ce script rajoute .pas là où il faut)."
   exit
end

fnp = fn".pas"
"\rm -f " fn
"\rm -f " fn".o"
"ppc386 " fnp
sovRc = rc
if sovRc=0 then do
   "./"||fn prms
   "ls -al "fn" " fn".*"
end ; else ; do ;
   say " erreur de compilation "
   exit sovRc
end ;
```

1.3.4 ce que fait l'option -t

Pour que la syntaxe sous-jacente à galg soit respectée, il faut écrire les algorithmes de façon peu "naturelle". Par exemple, les expressions `finsi` et `Fin Si` sont considérées comme incorrectes. Seul `Fin_si` est accepté, avec le caractère de soulignement entre le mot `Fin` et le mot `si`. Ces contraintes, obligatoires pour l'analyse et la traduction, peuvent gêner la lisibilité. C'est pourquoi l'option `-t` prend un texte avec une syntaxe moins stricte et vient rajouter le caractère de soulignement là où il le faut. Un autre aménagement consiste à rajouter le mot `affecter` au début de toute instruction d'affectation. Un troisième aménagement rajoute le mot `appeler` au début de toute instruction qui effectue un appel de sous-programme.

Le fichier produit par l'option `-t` de galg est un fichier dont l'extension est `.alg` qui peut ensuite être traité par l'option `-a` de galg comme indiqué sur le schéma ci-dessous où on aménage le fichier `nbParfaits.txt` :



On trouvera dans le chapitre 5 *Mode "Aménagement des algorithmes"* le détail de ce que fait l'option `-t` avec des exemples de fichiers avant et après l'utilisation de l'option.

1.3.5 ce que fait l'option -e

L'option -e de galg fournit la liste des erreurs standards détectées. Cette liste permet de se faire une idée de ce que sait détecter galg.

Nous la reproduisons ici sans commentaire car le chapitre 5 est consacré entièrement aux erreurs détectées et à la correction des instructions incriminées.

```
Liste des erreurs reconnues

  001 :  la première ligne du fichier-algorithme n'est pas un commentaire
  002 :  instruction inconnue (mot Un non reconnu)
  003 :  une instruction ECRIRE doit précéder l'instruction LIRE
  004 :  instruction ECRIRE incorrecte
  005 :  après LIRE il faut mettre le nom d'une variable
  006 :  une instruction LIRE ne lit qu'une variable à la fois
  007 :  partie gauche d'affectation incorrecte
  008 :  expression unique correcte requise
  009 :  identificateur égal au mot reservé 'A' ou 'DE'
  010 :  partie gauche de condition vide
  011 :  partie droite de condition vide
  012 :  instruction POUR incorrecte : fin_pour non commenté
  013 :  instruction SI incorrecte : fin_si non commenté
  014 :  instruction TANT_QUE incorrecte : fin_tant_que non commenté
  015 :  opération incorrecte
  016 :  mauvaise imbrication des structures SI, POUR ou TANT_QUE
  017 :  trop de niveaux d'imbrication (maximum autorisé : 3)
  018 :  expression mal parenthésée
  019 :  mot-clef non autorisé sur la même ligne que ALORS ou SINON
  020 :  identificateur incorrect
  021 :  ALORS absent
  022 :  partie gauche de 'A' invalide dans boucle pour
  023 :  partie droite de 'A' invalide dans boucle pour
  024 :  ALORS/SINON mal placé
  025 :  on ne peut pas lire directement une valeur de tableau
  026 :  expression mal guillemetisée
  027 :  expression mal crochetisée
  028 :  affectation interdite dans instruction ECRIRE
  029 :  expression booléenne interdite dans instruction ECRIRE
  030 :  expression incorrecte ou calcul incorrect
  031 :  opération, liste ou test non terminé
  032 :  appel de module sans parenthèse ouvrante
  033 :  appel de module avec des caractères après la parenthèse fermante
```

```
 034 :  syntaxe d'appel de module incorrecte
 035 :  identificateur homonyme d'un mot clef
 036 :  fin de structure incorrecte
 037 :  affectations multiples interdites sur une seule ligne
 038 :  trop de niveaux de parenthèses
 039 :  instruction AFFECTER sans symbole <--
 040 :  il doit manquer un paramètre
 041 :  il est interdit d'emboiter les crochets
 042 :  aucun appel de fonction n'est toléré à l'intérieur des crochets
 043 :  indice de tableau manquant
 044 :  partie gauche d'affectation de tableau incorrecte
 045 :  mot-clef ALORS en double
 046 :  mot-clef SINON en double
 047 :  partie droite d'affectation incorrecte
 048 :  points-virgules illicites
 049 :  caractère(s) invalide(s) dans expression
 050 :  instruction non reconnue après ALORS/SINON
 051 :  mot 'DE' dans boucle POUR non vu ou mal placé
 052 :  partie 'DE' invalide dans boucle pour
 053 :  arité du module incohérent avec arité précédente
 054 :  dimensions du tableau incohérentes avec dimensions précédentes
 055 :  il manque le mot COMME dans l'instruction OUVRIR
 056 :  il manque le mot EN_LECTURE ou EN_ECRITURE dans l'instruction OUVRIR
 057 :  pas de nom d'auteur fourni dans les 5 premières lignes de l'algorihtme
 058 :  les structures SI, POUR, REPETER et TANT_QUE ne sont pas toutes finies.
 059 :  l'algorithme ne comporte aucune instruction.
 060 :  ALORS mal placé

-- fin de la liste des erreurs reconnues

 Les fichier testNN.alg sur le site Web permettent de voir les erreurs NN.


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
```

1.3.6 ce que fait l'option -l

L'option -l de galg donne la liste des langages implémentés ainsi que les commandes associées aux langages. Le chapitre 6 détaille le paramétrage éventuel des commandes associées aux langages lorsque ceux-ci sont déjà installés. Voici l'affichage correspondant :

```
Valeurs du paramètre de l'option -o, langages et commandes associées

  Paramètre        Langage                                  Commande
  ---------        -------                                  --------
      c            C    (norme ansi)                        pc
      cpp          C++                                      ppc
      dbase        Dbase (version Max20)                    db
      java         Java                                     ja
      perl         PERL                                     perl -W
      rexx         REXX (implémentation : regina)           regina
      tcl          tcl                                      tcl


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
```

1.3.7 ce que fait l'option -f

L'option -f de galg permet de savoir quelles fonctions algorithmiques sont facilement traduites dans le langage choisi. Par exemple, pour le langage Rexx, c'est à dire si on exécute la commande `galg -f rexx` on obtient :

```
Table des 13 fonctions algorithmiques reconnues par le langage rexx :

              ALGORITHMIQUE                |     REXX
                                           |
 1  / 13  :   _codeAscii()                 |     c2d()
 2  / 13  :   _date()()                    |     date("E")
 3  / 13  :   _entierAuHasard()            |     random()
 4  / 13  :   _format()                    |     format()
 5  / 13  :   _heure()()                   |     time()
 6  / 13  :   _longueur()                  |     length()
 7  / 13  :   _maju()                      |     translate()
 8  / 13  :   _max()                       |     max()
 9  / 13  :   _nbParametres()              |     arg()
10  / 13  :   _parametre()                 |     arg()
11  / 13  :   _partieEntiere()             |     trunc()
12  / 13  :   _sousChaine()                |     substr()
13  / 13  :   _valeur()                    |     ()

 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
```

1.4 Syntaxe du langage algorithmique utilisé

Le langage utilisé est celui décrit dans le manuel d'Algorithmiques Raisonnées, disponible à l'adresse

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

à quelques exceptions près :

- un algorithme doit commencer par un commentaire,
- un algorithme doit contenir un commentaire avec le mot `AUTEUR` dans les 5 premières lignes du fichier,
- les affectations commencent par le mot-clé `AFFECTER`,
- un appel de sous-programme commence par le mot-clé `APPELER`,
- toute instruction imbriquante n'utilise qu'un seul mot pour indiquer le début et la fin d'instruction (par exemple la fin de la boucle "pour" s'écrit `FIN_POUR` et non pas `FIN POUR`),
- les niveaux d'imbrication ne doivent pas être de profondeur supérieure à trois,
- on n'a pas le droit d'écrire un deuxième `SI` sur la ligne du `ALORS` du premier `SI`,
- il est interdit de mettre une expression avec appel de fonction comme indice de tableau,
- il est interdit d'emboiter les crochets dans les indices de tableau,
- l'instruction `ECRIRE_` n'effectue pas de saut à la ligne.

Un fichier-algorithme contient soit des lignes vides (pour faciliter la lecture), soit des commentaires répérés par le symbole `#` soit des instructions, réparties sur une ou plusieurs lignes. Certains commentaires ont un sens particulier, comme par exemple le commentaire `#>` qui indique le nom d'un fichier à inclure lors de la traduction (voir le chapitre 4 à ce sujet).

Les instructions peuvent être écrites en majuscules, en minuscules ou dans une combinaison des deux. Par contre, les noms de variables, tableaux et modules ne peuvent pas comporter de caractères accentués. Le caractère de soulignement peut être utilisé pour ces mêmes noms mais il a un sens particulier pour la traduction (voir le chapitre 4 à ce sujet).

galg distingue les instructions non imbriquantes qui tiennent sur une seule ligne et les instructions imbriquantes qui sont impérativement réparties sur plusieurs lignes.

Les mots qui commencent une instruction non imbriquante sont exclusivement (par ordre alphabétique)

```
AFFECTER APPELER ECRIRE FERMER LIRE OUVRIR QUITTER
```

Les mots qui commencent une instruction imbriquante sont exclusivement (par ordre alphabétique)

```
POUR REPETER SI TANT_QUE
```

Les mots qui terminent une instruction imbriquante sont exclusivement (par ordre alphabétique)

```
FIN_POUR FIN_SI FIN_TANT_QUE JUSQU'A
```

Les règles suivantes, de bon sens, sont impératives pour galg :

- l'imbriquante imbriquante qui commence par le mot `POUR` doit se terminer par le mot `FIN_POUR`,
- l'imbriquante imbriquante qui commence par le mot `SI` doit se terminer par le mot `FIN_SI`,
- l'imbriquante imbriquante qui commence par le mot `TANT_QUE` doit se terminer par le mot `FIN_TANT_QUE`,
- l'imbriquante imbriquante qui commence par le mot `REPETER` doit se terminer par le mot `JUSQU'A`,
- l'instruction `SI` requiert le mot `ALORS` sur la ligne suivante et peut comporter éventuellement le mot `SINON` situé après les instructions du `ALORS`,
- l'instruction `LIRE` n'accepte qu'un seul nom de variable (sauf pour la lecture sur fichier),
- l'instruction `ECRIRE` permet d'écrire plusieurs expressions à condition de les séparer par des virgules,
- tout appel de fonction utilise des parenthèses ; les paramètres doivent êtres séparé par des virgules,
- lors de l'appel de module par l'instruction `APPELER` les paramètres doivent êtres séparés par des virgules.

Chapitre 2.

Mode "Analyse d'algorithmes"

2.1 Un exemple simple sans erreur

Commençons par un exemple simple, celui de la conversion minimale d'un montant en francs en euros dont l'algorithme est

```
################################################
#                                              #
#   cfemin.alg  ;  conversion francs en euro   #
#               -- auteur : gh                 #
#                                              #
################################################


# 1. Saisie du montant en Francs

écrire_  " Quel est le montant en Francs ? "
lire     mntF


# 2. Conversion et affichage

affecter mntE <-- mntF / 6.55957
écrire   mntF , " Francs font " , mntE , " euros."
écrire   " et si on arrondit : " , partieEntiere( 0.5 + mntE ), " euros."
```

Lorsqu'on analyse cet algorithme, disons dans le fichier `cfemin.alg` avec la commande `galg -a cfemin.alg`, le programme galg produit deux fichiers :

- `cfemin.lst` qui contient une copie de `cfemin.alg` avec numérotation des lignes, des instructions et code d'erreur et détail éventuel de l'erreur ;
- `cfemin.lvm` qui contient les listes alphabétiques des modules, variables et tableaux avec l'arité (nombre d'arguments) des modules, le nombre de fois où chaque identificateur est utilisé et le numéro de la ligne où il apparait pour la première fois.

Voici tout d'abord le fichier des instructions :

```
 Fichier cfemin.lst issu de galg -a  cfemin.alg
 ==============================================
   03/08/2001 0:9.26

LIG  |  INS PRO
001  |           ##################################################
002  |           #                                                #
003  |           #   cfemin.alg  ;  conversion francs en euro     #
004  |           #               -- auteur : gh                   #
005  |           #                                                #
006  |           ##################################################
007  |
008  |
009  |           # 1. Saisie du montant en Francs
010  |
011  |  001      écrire_  " Quel est le montant en Francs ? "
012  |  002      lire     mntF
013  |
014  |
015  |           # 2. Conversion et affichage
016  |
017  |  003      affecter mntE <-- mntF / 6.55957
018  |  004      écrire   mntF , " Francs font " , mntE , " euros."
019  |  005      écrire   " et si on arrondit : " ,
                            partieEntiere( 0.5 + mntE ), " euros."

 -- Fin du fichier cfemin.lst issu de galg -a  cfemin.alg
```

Le fichier des modules, variables et tableaux, contient, quant à lui

```
Fichier cfemin.lvm issu de galg -a  cfemin.alg
================================================
  03/08/2001 0:9.26

* Liste des variables (par ordre alphabétique)

Variable                                Ligne de première     Nombre de
                                           utilisation        références
 #  mntE                                       17                 3
 #  mntF                                       12                 3

**** Liste des modules (par ordre alphabétique)

Module                    Arité         Ligne de première     Nombre de
                                           utilisation        références
 #  partieEntiere            1                19                 1

-- Fin du fichier cfemin.lvm issu de galg -a  cfemin.alg
```

2.2 Détail de l'analyse

L'analyse de l'algorithme passe chaque ligne en revue, les unes après les autres. Chaque ligne d'instruction est décomposée en trois parties :

- une partie **instruction** avec un mot-clé unique qui est souvent le premier mot de la ligne,
- une partie **paramètres** de l'instruction, éventuellement vide ou composée de plusieurs mots,
- une partie **variable de fichier**, éventuellement vide.

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		

Il est clair que certains noms d'instructions sont composés à partir des lignes d'instruction : par exemple `LIRE_SUR` qui est un nom d'instruction valide s'obtient en détectant `LIRE` puis `SUR`. De même, `OUVRIR_LEC` résulte de `OUVRIR` et de `EN_LECTURE`. Voici d'autres exemples de découpage avec le nom interne des instructions utilisées :

- 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 Détail des fichiers produits

Le fichier des instructions contient, dans cet ordre, pour chaque ligne du fichier-algorithme :

- un code erreur (éventuellement vide),
- le numéro de la ligne dans le fichier-algorithme
- le numéro de l'instruction,
- la profondeur d'imbrication de la ligne,
- le texte de la ligne de l'algorithme.

Lorsqu'il y a une erreur pour une ligne, le texte associé à l'erreur est écrit sous la ligne (voir la section suivante pour les fichiers-résultats correspondants).

Le fichier des modules, variables et tableaux contient, dans cet ordre :

- la liste des modules,
- la liste des variables,
- la liste des tableaux.

Pour chaque liste, on dispose du nom du module (ou de la variable ou du tableau), du premier numéro de ligne dans le fichier-algorithme où il apparait et du nombre de références à ce module. L'intérêt du nombre de références permet de détecter une faute de frappe. Par exemple si dans l'algorithme précédent, si on utilise l'identificateur `PRODUITT` (avec deux `T`, donc) au lieu de `PRODUIT`, le nombre de références à la variable `PRODUITT` vaut 1.

Pour la liste des modules, galg affiche aussi l'arité c'est à dire le nombre de paramètres à passer au module.

galg ne considére pas comme une erreur de n'avoir qu'une seule référence à une variable. Il y a deux raisons principales à cela : l'algorithme peut partiellement écrit au moment où on le teste, certaines références sont uniques car elles sont utilisées à l'intérieur d'une boucle.

2.4 Un exemple avec des erreurs

L'intérêt de cet exemple est de montrer ce qu'affiche galg en cas d'erreur et de discuter ce que fait galg s'il y a plusieurs erreurs pour une même ligne dans un fichier-algorithme.

galg détecte un certain nombre d'erreurs. Le détail des erreurs et des indications de correction sont fournis au chapitre 3.

Voici l'algorithme :

```
# bad.alg : un algorithme avec des erreurs
# auteur  : (anonyme, ça vaut mieux !)

affecter  x <--
initVar( x ; t )
écrire    " x = "   x
écrire    " y = "   f( x
écrire    " y = "   , x ( t )
écrire    " y = "   , y ;
```

Il a plusieurs erreurs dans cet algorithme :

- la première erreur est l'absence de partie droite dans l'affectation ;
- la deuxième erreur (moins flagrante) est la présence d'un point-virgule pour séparer les paramètres d'appel dans un module (il faut mettre des virgules et non pas des points-virgules) ;
- la troisième erreur (peu flagrante aussi) est l'absence de virgule pour séparer la chaine `" x = "` de la variable `x` pour l'instruction `ECRIRE`. Au lieu de

  ```
  écrire    " x = "   x
  ```

  on aurait du avoir

  ```
  écrire    " x = "   ,  x
  ```

- l'instruction suivante comporte deux erreurs : comme pour l'instruction précédente, il manque une virgule comme séparateurs des composants de l'instruction `ECRIRE` ; mais il manque aussi une parenthèse fermante pour l'appel de la fonction `f`.
- l'avant-dernière instruction réalise une erreur d'homonymie : l'identificateur `x` désigne à la fois une variable et une fonction.
- la dernière instruction comporte un point-virgule en fin de ligne.

Le fichier des instructions ne liste pas toutes les erreurs car galg ne revient pas en arrière sur les instructions analysées. La détection de l'homonymie pour `X` comme variable et comme fonction est donc indiquée dans le fichier des variables (ce fichier est listé un peu plus loin dans la section) et non pas dans le fichier des instructions.

```
Fichier bad.lst issu de galg -a  bad.alg
========================================
  05/08/2001 16:19.53

 ERR  LIG  |  INS PRO
      001  |              # bad.alg : un algorithme avec des erreurs
      002  |              # auteur  : (anonyme, ça vaut mieux !)
      003  |

 ***  004  |  001         affecter  x <--
    ERREUR 047 :  PARTIE DROITE D'AFFECTATION INCORRECTE


 ***  005  |  001         initVar( x ; t )
    ERREUR 002 :  INSTRUCTION INCONNUE (MOT UN NON RECONNU)


 ***  006  |  002         écrire    " x = "   x
    ERREUR 030 :  EXPRESSION INCORRECTE OU CALCUL INCORRECT


 ***  007  |  003         écrire    " y = "   f( x
    ERREUR 018 :  EXPRESSION MAL PARENTHESEE

      008  |  004         écrire    " y = "   , x ( t )

 ***  009  |  005         écrire    " y = "   , y ;
    ERREUR 048 :  POINTS-VIRGULES ILLICITES


-- Fin du fichier bad.lst issu de galg -a  bad.alg
```

Voici maintenant le fichier des modules, variables et tableaux :

```
Fichier bad.lvm issu de galg -a  bad.alg
========================================
  05/08/2001 16:19.53


* Liste des variables (par ordre alphabétique)

Variable                                Ligne de première     Nombre de
                                           utilisation        références
 #  t                                          8                 1
 #  x                                          4                 1

**** Liste des modules (par ordre alphabétique)

Module                   Arité        Ligne de première     Nombre de
                                           utilisation        références
 #  x                       1                 8                 1

Il y a un identificateur en homonymie à savoir :

   x                     Module et Variable

-- Fin du fichier bad.lvm issu de galg -a  bad.alg
```

2.5 Des exemples plus conséquents

L'algorithme présenté ici est un algorithme classique qui détermine le plus grand élément d'un tableau d'entiers et le nombre de fois où il apparait dans le tableau. Une particularité de l'algorithme est qu'il réalise la détermination et le comptage du maximum dans la même boucle.

Pour initialiser le tableau, on suppose qu'il existe une fonction algorithmique nommée `entierAuHasard` qui prend deux paramètres, disons `vmin` et `vmax` et qui renvoie un nombre entier au hasard (ou plus exactement pseudo-aléatoire uniformément réparti) entre `vmin` et `vmax`.

Voici tout d'abord une version monobloc, sans aucun sous-programme :

```
################################################
#                                              #
#                                              #
#   détermination du maximum dans un tableau   #
#   et comptage du nombre d'occurences de ce   #
#   maximum  en une seule boucle               #
#                                              #
#                                              #
################################################
#                                              #
#                                 auteur : gh  #
#                                              #
################################################


# initialisation du tableau avec 15 éléments
# entiers entre 10 et 20

affecter nbElt <-- 15
pour indb de1a nbElt
     affecter monT[indb] <-- entierAuHasard( 10, 20 )
fin_pour # indb de1a nbElt-1


# initialisation de la valeur du maximum (valMax)
# et de son nombre d'occurences (nbMax)

affecter valMax  <-- monT[ nbElt ]
affecter nbMax   <-- 1
```

```
# on parcourt alors le tableau
# sans utiliser le dernier élément déja comptabilisé

pour indb de1a nbElt-1
     affecter eltCourant <-- monT[ indb ]
     si eltCourant > valMax
        alors # nouveau maximum local
              affecter valMax  <-- eltCourant
              affecter nbMax   <-- 1
        sinon
              si eltCourant = valMax
                 alors # une fois de plus le maximum
                       affecter nbMax <-- nbMax + 1
              fin_si # nouvelle occurence du maximum
     fin_si # nouveau maximum
fin_pour # indb de1a 10


# affichage de fin

écrire " Le maximum dans le tableau est : " , valMax
écrire " et il apparait " , nbMax , " fois."
écrire " Pour vérification, voici les éléments du tableau : "

pour indb de1a nbElt
    écrire " élément " , format(indb,3) , ":" , format(monT[indb],4)
fin_pour # indb de1a nbElt-1
```

Voici le listage de l'algorithme une fois analysé :

```
 Fichier maxoccmono.lst issu de galg -a  maxoccmono.alg
 ======================================================

LIG  |  INS PRO
001  |           ################################################
002  |           #                                              #
003  |           #                                              #
004  |           #   détermination du maximum dans un tableau   #
005  |           #   et comptage du nombre d'occurences de ce   #
006  |           #   maximum  en une seule boucle               #
007  |           #                                              #
008  |           #                                              #
009  |           ################################################
010  |           #                                              #
011  |           # auteur : gh                                  #
012  |           #                                              #
013  |           ################################################
```

```
014  |
015  |
016  |           # initialisation du tableau avec 15 éléments
017  |           # entiers entre 10 et 20
018  |
019  |  001      affecter nbElt <-- 15
020  |  002 001  pour indb de1a nbElt
021  |  002 001       affecter monT[indb] <-- entierAuHasard( 10, 20 )
022  |  002 001  fin_pour # indb de1a nbElt-1
023  |
024  |
025  |           # initialisation de la valeur du maximum (valMax)
026  |           # et de son nombre d'occurences (nbMax)
027  |
028  |  003      affecter valMax  <-- monT[ nbElt ]
029  |  004      affecter nbMax   <-- 1
030  |
031  |
032  |           # on parcourt alors le tableau
033  |           # sans utiliser le dernier élément déja comptabilisé
034  |
035  |  005 001  pour indb de1a nbElt-1
036  |  005 001       affecter eltCourant <-- monT[ indb ]
037  |  005 002       si eltCourant > valMax
038  |  005 002          alors # nouveau maximum local
039  |  005 002                affecter valMax  <-- eltCourant
040  |  005 002                affecter nbMax   <-- 1
041  |  005 002          sinon
042  |  005 003                si eltCourant = valMax
043  |  005 003                   alors # une fois de plus le maximum
044  |  005 003                         affecter nbMax <-- nbMax + 1
045  |  005 003                fin_si # nouvelle occurence du maximum
046  |  005 002       fin_si # nouveau maximum
047  |  005 001  fin_pour # indb de1a 10
048  |
049  |
050  |           # affichage de fin
051  |
052  |  006      écrire " Le maximum dans le tableau est : " , valMax
053  |  007      écrire " et il apparait " , nbMax , " fois."
054  |  008      écrire " Pour vérification, voici les éléments du tableau : "
055  |
056  |  009 001  pour indb de1a nbElt
057  |  009 001      écrire "élément " , format(indb,3) , ":" , format(monT[indb],4)
058  |  009 001  fin_pour # indb de1a nbElt-1

 -- Fin du fichier maxoccmono.lst issu de galg -a  maxoccmono.alg
```

Voici également le fichier des variables, tableaux et modules :

```
Fichier maxoccmono.lvm issu de galg -a  maxoccmono.alg
=====================================================
  05/08/2001 17:16.40


* Liste des variables (par ordre alphabétique)

Variable                              Ligne de première     Nombre de
                                         utilisation        références
 #  eltCourant                              36                 4
 #  indb                                    21                 7
 #  nbElt                                   19                 6
 #  nbMax                                   29                 5
 #  valMax                                  28                 5

*** Liste des tableaux (par ordre alphabétique)

Tableau                 Dims.         Ligne de première     Nombre de
                                         utilisation        références
 #  monT                    1               21                 4

**** Liste des modules (par ordre alphabétique)

Module                  Arité         Ligne de première     Nombre de
                                         utilisation        références
 #  entierAuHasard          2               21                 1
 #  format                  2               57                 2

-- Fin du fichier maxoccmono.lvm issu de galg -a  maxoccmono.alg
```

Algorithmiquement, il serait mieux d'utiliser deux sous-algorithmes (ou "sous-programmes"), un pour initialiser le tableau et un autre pour l'afficher.

On trouvera sur les pages suivantes l'algorithme principal (les deux sous-programmes sont nommés respectivement `init_Tab` et `affiche_Tab`), son listage et le fichier des ses variables, tableaux et modules.

```
 Fichier maxocc.lst issu de galg -a  maxocc.alg
 ==============================================
   05/08/2001 16:21.21

LIG  |  INS PRO
001  |           ############################################################
002  |           #                                                          #
003  |           #  auteur : (gH)                                           #
004  |           #  maxocc.alg : calcul du plus grand élément d'un          #
005  |           #               tableau avec comptage du nombre            #
006  |           #               d'occurences en une seule boucle.          #
007  |           #                                                          #
008  |           ############################################################
009  |
010  |           # 1. initialisation du tableau avec 15 éléments
011  |           #    entiers entre 10 et 20
012  |
013  |
014  |  001      affecter nbElt <-- 15
015  |  002      appeler init_Tab( monT , nbElt , 10 , 20 )
016  |
017  |           # 2. détermination du max et comptage du nb d'occurences
018  |           #    du max en une seule boucle
019  |
020  |           #    2.1 initialisation de la valeur du maximum (valMax)
021  |           #        et de son nombre d'occurences (nbMax)
022  |
023  |  003      affecter valMax  <-- monT[ nbElt ]
024  |  004      affecter nbMax   <-- 1
025  |
026  |           #    2.2 on parcourt alors le tableau
027  |           #        sans utiliser le dernier élément déja comptabilisé
028  |
029  |  005 001  pour indb de1a nbElt-1
030  |  005 001       affecter eltCourant <-- monT[ indb ]
031  |  005 002       si eltCourant > valMax
032  |  005 002          alors # nouveau maximum local
033  |  005 002                affecter valMax  <-- eltCourant
034  |  005 002                affecter nbMax   <-- 1
035  |  005 002          sinon
036  |  005 003                si eltCourant = valMax
037  |  005 003                   alors # une fois de plus le maximum
038  |  005 003                         affecter nbMax <-- nbMax + 1
039  |  005 003                fin_si # nouvelle occurence du maximum
040  |  005 002       fin_si # nouveau maximum
041  |  005 001  fin_pour # indb de1a 10
042  |
043  |           # 3. affichage de fin
```

```
044  |
045  |  006      écrire " Le maximum dans le tableau est : " , valMax
046  |  007      écrire " et il apparait " , nbMax , " fois."
047  |  008      écrire " Pour vérification, voici les éléments du tableau : "
048  |
049  |  009      appeler affiche_Tab( monT , 1 , nbElt , 4)

 -- Fin du fichier maxocc.lst issu de galg -a  maxocc.alg


 Fichier maxocc.lvm issu de galg -a  maxocc.alg
 ==============================================
   05/08/2001 16:21.21


 * Liste des variables (par ordre alphabétique)

 Variable                              Ligne de première     Nombre de
                                          utilisation        références
  #  eltCourant                              30                4
  #  indb                                    30                2
  #  monT                                    15                2
  #  nbElt                                   14                6
  #  nbMax                                   24                5
  #  valMax                                  23                5

 *** Liste des tableaux (par ordre alphabétique)

 Tableau                 Dims.         Ligne de première     Nombre de
                                          utilisation        références
  #  monT                    1               23                2

 **** Liste des modules (par ordre alphabétique)

 Module                  Arité         Ligne de première     Nombre de
                                          utilisation        références
  #  affiche_Tab             4               49                1
  #  init_Tab                4               15                1

 Il y a un identificateur en homonymie à savoir :

    monT                 Tableau et Variable

 -- Fin du fichier maxocc.lvm issu de galg -a  maxocc.alg
```

Un algorithme long

Pour finir ce chapitre voici un algorithme long et son analyse par galg. Il est à noter que pour des questions de cadrage certains lignes du listage ont étés remaniées afin de tenir dans la largeur de la page, comme par exemple

```
094 | 015 002 affecter machineTacheDuree[ machineCour , tacheCour ] <--
                                                              resteTemps
```

ce qui gêne un peu la lisibilité mais évite d'utiliser des très petites polices de caractères.

```
#################################################################
#                                                               #
#   macNaughton.alg : affectation de taches selon               #
#                     la méthode de mac Naughton                #
#   auteur (gH)                                                 #
#                                                               #
#                                                               #
#################################################################
#                                                               #
#   références :  Cormen, Leiserson et Rivest                   #
#                 Introduction to algorithms, MIT Press         #
#                                                               #
#################################################################
#                                                               #
# avec nbMachines qui doivent effectuer nbTaches et sachant     #
# que la tache numéro i dure dureeTache[i] comment répartir     #
# les taches ? la durée est en heure et les taches sont         #
# morcelables heure par heure.                                  #
#                                                               #
#################################################################



# 0.  Affectations arbitraires pour tester l'exemple fourni
#     dans le cours vu sur le web à l'adresse
#

affecter nbMachines  <-- 3
affecter nbTaches    <-- 6

affecter dureeTache[ 1 ]  <--   5
affecter dureeTache[ 2 ]  <--   3
affecter dureeTache[ 3 ]  <--   4
affecter dureeTache[ 4 ]  <--   5
affecter dureeTache[ 5 ]  <--   2
affecter dureeTache[ 6 ]  <--   5
```

```
# 1. Calcul de la durée minimale

affecter somDuree <-- 0
pour indTache de1a nbTaches
  affecter dureeCour <-- dureeTache[ indTache ]
  affecter somDuree  <-- somDuree + dureeCour
  si indTache=1
     alors affecter dureeMax <-- dureeCour
     sinon
      si dureeCour > dureeMax
         alors affecter dureeMax <-- dureeCour
      fin_si # dureeCour > dureeMAx
  fin_si # indTache=1
fin_pour # indTache de1a nbTaches
affecter dureeMin <-- max( somDuree / nbMachines , dureeMax)

# 2. Affectation des taches aux machines

# 2.1 Initialisation du tableau de répartititon

pour indTache de1a nbTaches
  affecter nbTache[indTache] <-- 0
  pour indMachine de1a nbMachines
    affecter machineTacheNumero[indTache,indMachine] <-- 0
    affecter machineTacheDuree[indTache ,indMachine] <-- (-1)
  fin_pour # indMachine de1a nbMachines
fin_pour # indTache de1a nbTaches

# 2.2 Boucle d'affectation

affecter tempsPris   <-- 0
affecter machineCour <-- 1

pour indTache de1a nbTaches
 affecter dureeCour <-- dureeTache[indTache]
 si tempsPris+dureeCour <=  dureeMin
 alors # on peut affecter toute la tache à la machine courante

   affecter tacheCour                                <--  nbTache[machineCour]+ 1
   affecter nbTache[machineCour]                     <--  tacheCour
   affecter machineTacheNumero[machineCour,tacheCour] <--  indTache
   affecter machineTacheDuree[machineCour,tacheCour]  <--  dureeCour
   affecter tempsPris                                <--  tempsPris + dureeCour
   si tempsPris = dureeMin
      alors affecter machineCour <-- machineCour + 1
            affecter tempsPris   <-- 0
   fin_si # tempsPris = dureeMin
```

```
sinon # on n'a le droit d'affecter qu'une partie à la machine courante

 affecter tacheCour                                  <--  nbTache[machineCour]+1
 affecter nbTache[machineCour]                       <--  tacheCour
 affecter machineTacheNumero[machineCour,tacheCour] <--  indTache
 affecter resteTemps                                 <--  dureeMin - tempsPris
 affecter machineTacheDuree[machineCour,tacheCour]  <--  resteTemps

 # et on affecte ce qui reste à la machine suivante

 affecter machineCour                                <--  machineCour + 1
 affecter tacheCour                                  <--  nbTache[machineCour]+1
 affecter nbTache[machineCour]                       <--  tacheCour
 affecter machineTacheNumero[machineCour,tacheCour] <--  indTache
 affecter machineTacheDuree[machineCour,tacheCour]  <--  dureeCour - resteTemps
 affecter tempsPris                                  <--  dureeCour - resteTemps

 fin_si # tempsPris+dureeTache[tacheCour]<=  dureeMin
fin_pour # indTache de1a nbTaches

# 3. Affichage des données et des répartitions

écrire " Il y avait " , nbMachines , " machines et " , nbTaches ," taches à répartir."
écrire " Voici la durée des taches : "
écrire "   Tache    Durée (en heures)"
pour indTache de1a nbTaches
     écrire  format(indTache,8) , format( dureeTache[indTache ],8 )
fin_pour # indTache de1a nbTaches

écrire ""
écrire " La durée minimale calculée est " , dureeMin ," heures."
écrire ""

pour indMachine de1a nbMachines
 écrire "  -- machine " , indMachine , " : "
 affecter nbTacheCour <-- nbTache[indMachine]
 si nbTacheCour > 0
    alors
     pour indTache de1à nbTacheCour
       affecter dureeCour <-- machineTacheDuree[indMachine,indTache]
       affecter tacheCour <-- machineTacheNumero[indMachine,indTache]
       écrire   copies("  ",6) , " tache " , tacheCour , " durée " , dureeCour
     fin_pour # indTache de1à nbTacheCour
 fin_si # numMachine > 0
fin_pour # indTache de1a nbTaches
```

```
 Fichier macnaughton.lst issu de galg -a  macnaughton.alg
 ========================================================
   05/08/2001 18:27.27

LIG   |  INS PRO
001   |           ######################################################################
002   |           #                                                                    #
003   |           #   macNaughton.alg : affectation de taches selon                    #
004   |           #                     la méthode de mac Naughton                     #
005   |           #   auteur (gH)                                                      #
006   |           #                                                                    #
007   |           #                                                                    #
008   |           ######################################################################
009   |           #                                                                    #
010   |           #   références :  Cormen, Leiserson et Rivest                       #
011   |           #                  Introduction to algorithms, MIT Press             #
012   |           #                                                                    #
013   |           ######################################################################
014   |           #                                                                    #
015   |           #                                                                    #
016   |           # avec nbMachines qui doivent effectuer nbTaches et sachant         #
017   |           # que la tache numéro i dure dureeTache[i] comment répartir         #
018   |           # les taches ? la durée est en heure et les taches sont             #
019   |           # morcelables heure par heure.                                      #
020   |           #                                                                    #
021   |           #                                                                    #
022   |           ######################################################################
023   |
024   |           # 0.  Affectations arbitraires pour tester l'exemple fourni
025   |           #     dans le cours vu sur le web à l'adresse
026   |           #
027   |
028   |  001      affecter nbMachines  <-- 3
029   |  002      affecter nbTaches    <-- 6
030   |
031   |  003      affecter dureeTache[ 1 ]  <--   5
032   |  004      affecter dureeTache[ 2 ]  <--   3
033   |  005      affecter dureeTache[ 3 ]  <--   4
034   |  006      affecter dureeTache[ 4 ]  <--   5
035   |  007      affecter dureeTache[ 5 ]  <--   2
036   |  008      affecter dureeTache[ 6 ]  <--   5
037   |
038   |           # 1. Calcul de la durée minimale
039   |
040   |  009      affecter somDuree <-- 0
041   |  010 001  pour indTache de1a nbTaches
042   |  010 001    affecter dureeCour <-- dureeTache[ indTache ]
043   |  010 001    affecter somDuree  <-- somDuree + dureeCour
```

```
044  |  010 002    si indTache=1
045  |  010 002       alors affecter dureeMax <-- dureeCour
046  |  010 002       sinon
047  |  010 003        si dureeCour > dureeMax
048  |  010 003           alors affecter dureeMax <-- dureeCour
049  |  010 003        fin_si # dureeCour > dureeMAx
050  |  010 002    fin_si # indTache=1
051  |  010 001  fin_pour # indTache de1a nbTaches
052  |  011       affecter dureeMin <-- max( somDuree / nbMachines , dureeMax)
053  |
054  |            # 2. Affectation des taches aux machines
055  |
056  |            # 2.1 Initialisation du tableau de répartititon
057  |
058  |  012 001  pour indTache de1a nbTaches
059  |  012 001    affecter nbTache[ indTache ] <-- 0
060  |  012 002    pour indMachine de1a nbMachines
061  |  012 002      affecter machineTacheNumero[ indTache , indMachine ] <-- 0
062  |  012 002      affecter machineTacheDuree[ indTache , indMachine ]  <-- (-1)
063  |  012 002    fin_pour # indMachine de1a nbMachines
064  |  012 001  fin_pour # indTache de1a nbTaches
065  |
066  |            # 2.2 Boucle d'affectation
067  |
068  |  013       affecter tempsPris   <-- 0
069  |  014       affecter machineCour <-- 1
070  |
071  |  015 001  pour indTache de1a nbTaches
072  |  015 001   affecter dureeCour <-- dureeTache[ indTache ]
073  |
074  |  015 002   si tempsPris+dureeCour <=  dureeMin
075  |
076  |  015 002   alors # on peut affecter toute la tache à la machine courante
077  |
078  |  015 002     affecter tacheCour                                          <--
                                                 nbTache[ machineCour ] + 1
079  |  015 002     affecter nbTache[ machineCour ]                             <--
                                                                        tacheCour
080  |  015 002     affecter machineTacheNumero[ machineCour , tacheCour ]  <--
                                                                        indTache
081  |  015 002     affecter machineTacheDuree[  machineCour , tacheCour ]  <--
                                                                        dureeCour
082  |  015 002     affecter tempsPris                                          <--
                                                       tempsPris + dureeCour
083  |  015 003     si tempsPris = dureeMin
084  |  015 003        alors affecter machineCour <-- machineCour + 1
085  |  015 003              affecter tempsPris   <-- 0
086  |  015 003     fin_si # tempsPris = dureeMin
```

```
087  |
088  |  015 002   sinon # on n'a le droit d'affecter qu'une partie à la machine courante
089  |
090  |  015 002     affecter tacheCour                                          <--
                                                  nbTache[ machineCour ] + 1
091  |  015 002     affecter nbTache[ machineCour ]                             <--
                                                                    tacheCour
092  |  015 002     affecter machineTacheNumero[ machineCour , tacheCour ] <--
                                                                      indTache
093  |  015 002     affecter resteTemps                                         <--
                                                        dureeMin - tempsPris
094  |  015 002     affecter machineTacheDuree[  machineCour , tacheCour ] <--
                                                                    resteTemps
095  |
096  |              # et on affecte ce qui reste à la machine suivante
097  |
098  |  015 002     affecter machineCour                                        <--
                                                           machineCour + 1
099  |  015 002     affecter tacheCour                                          <--
                                                  nbTache[ machineCour ] + 1
100  |  015 002     affecter nbTache[ machineCour ]                             <--
                                                                    tacheCour
101  |  015 002     affecter machineTacheNumero[ machineCour , tacheCour ] <--
                                                                      indTache
102  |  015 002     affecter machineTacheDuree[  machineCour , tacheCour ] <--
                                                        dureeCour - resteTemps
103  |  015 002     affecter tempsPris                                          <--
                                                        dureeCour - resteTemps
104  |
105  |  015 002    fin_si # tempsPris+dureeTache[ tacheCour ] <=  dureeMin
106  |  015 001   fin_pour # indTache de1a nbTaches
107  |
108  |            # 3. Affichage des données et des répartitions
109  |
110  |  016       écrire " Il y avait " , nbMachines , " machines et " ,
                                          nbTaches ," taches à répartir."
111  |  017       écrire " Voici la durée des taches : "
112  |  018       écrire "   Tache    Durée (en heures)"
113  |  019 001   pour indTache de1a nbTaches
114  |  019 001        écrire  format(indTache,8) , format( dureeTache[ indTache ],8 )
115  |  019 001   fin_pour # indTache de1a nbTaches
116  |
117  |  020       écrire ""
118  |  021       écrire " La durée minimale calculée est " , dureeMin ," heures."
119  |  022       écrire ""
120  |
121  |  023 001   pour indMachine de1a nbMachines
122  |  023 001    écrire "  -- machine " , indMachine , " : "
123  |  023 001    affecter nbTacheCour <-- nbTache[ indMachine ]
```

```
124  |  023 002   si nbTacheCour > 0
125  |  023 002      alors
126  |  023 003       pour indTache de1à nbTacheCour
127  |  023 003         affecter dureeCour <-- machineTacheDuree[  indMachine ,
                                                                    indTache ]
128  |  023 003         affecter tacheCour <-- machineTacheNumero[ indMachine ,
                                                                    indTache ]
129  |  023 003         écrire   copies("  ",6) , " tache " , tacheCour ,
                                                          " durée " , dureeCour
130  |  023 003       fin_pour # indTache de1à nbTacheCour
131  |  023 002    fin_si # numMachine > 0
132  |  023 001  fin_pour # indTache de1a nbTaches

 -- Fin du fichier macnaughton.lst issu de galg -a  macnaughton.alg


 Fichier macnaughton.lvm issu de galg -a  macnaughton.alg
 ========================================================
   05/08/2001 18:27.27

 * Liste des variables (par ordre alphabétique)

 Variable                                Ligne de première     Nombre de
                                            utilisation       références
  #  dureeCour                                  42               13
  #  dureeMax                                   45                4
  #  dureeMin                                   52                5
  #  indMachine                                 61               11
  #  indTache                                   42               21
  #  machineCour                                69               29
  #  nbMachines                                 28                5
  #  nbTacheCour                               123                3
  #  nbTaches                                   29                6
  #  resteTemps                                 93                4
  #  somDuree                                   40                4
  #  tacheCour                                  78               20
  #  tempsPris                                  68                8

 *** Liste des tableaux (par ordre alphabétique)

 Tableau                 Dims.        Ligne de première     Nombre de
                                            utilisation       références
  #  dureeTache              1               31                9
  #  machineTacheDuree       2               62                5
  #  machineTacheNumero      2               61                5
  #  nbTache                 1               59                8
```

```
**** Liste des modules (par ordre alphabétique)

Module                  Arité        Ligne de première     Nombre de
                                        utilisation        références
 #  copies                2               129                1
 #  format                2               114                2
 #  max                   2                52                1

-- Fin du fichier macnaughton.lvm issu de galg -a  macnaughton.alg
```

Chapitre 3.

Erreurs détectées

galg détecte une soixantaine d'erreurs. Nous n'allons pas les présenter toutes ici car on trouvera sur le *Web* à l'adresse

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

des mini-algorithmes qui mettent en oeuvre chacune des erreurs : ainsi le fichier `test57.alg` produit l'erreur `57`, le fichier `test32.alg` produit l'erreur `32` etc.

Nous présenterons par contre deux types d'erreurs standards :

1. les erreurs les plus fréquentes,
2. les erreurs délicates à corriger.

3.1 Erreurs les plus fréquentes

L'erreur la plus fréquente de toutes qui est en même temps une faute de gout est de ne pas commmencer un algorithme par un commentaire. C'est pourquoi c'est l'erreur numéro `001`. L'algorithme minimal qui dit "bonjour" avec la seule ligne

```
ECRIRE "bonjour"
```

déclenche en particulier cette erreur.

La deuxième erreur sans doute la plus fréquente est aussi une faute de gout et consiste à ne pas mettre le mot "auteur" dans les 5 premières lignes du fichier. Rappelons que forcer les commentaires de début et le mot "auteur" a pour but de rappeler que les algorithmes et les programmes sont rarement assez commentés. Corriger l'algorithme précédent en

```
# un bonjour minimal
ECRIRE "bonjour"
```

ne déclenche donc pas l'erreur `001` mais l'erreur `057`.

La troisième erreur la plus fréquente est sans doute celle qui consiste à ne pas utiliser un des mots réservés. Elle porte le numéro `002`. Elle survient avec des instructions qui semblent correctes comme

```
x <-- 2
```

car pour galg une affectation doit commencer par le mot `AFFECTER`. Il faut donc écrire explicitement

```
AFFECTER x <-- 2
```

Cette erreur numéro `02` survient aussi avec une ligne comme

```
init_Tab( montT, 1, 10 )
```

car pour galg une appel de module doit commencer par le mot `APPELER` et la ligne précédente doit donc être corrigée en

```
APPELER init_Tab( montT, 1, 10 )
```

L'erreur numéro `02` survient bien sûr aussi dans le cas d'une faute de frappe comme pour

```
ECIRE " il n'y a pas de solution ".
```

Mettre des points-virgules en fin de ligne (ou même seulement enfin d'instruction) ne sert à rien pour galg ; utiliser des points-virgules comme dans

```
ECRIRE " il n'y a pas de solution " ;
```

déclenche donc l'erreur `048`.

Si on utilise un autre jeu de symboles de commentaire que `#`, l'erreur `002` apparait pour chacune des lignes mal commentées, c'est à dire ici pour les 4 lignes suivantes

```
/*
    un commentaire "à la C" est considéré
    comme incorrect par galg
*/
```

Une erreur très fréquente aussi est de vouloir lire deux valeurs sur une seule ligne, comme dans

```
LIRE x,y
```

ce qui a pour effet de déclencher l'erreur numéro `006`.

Oublier de poser une question avec l'instruction `ECRIRE` avant d'effectuer la lecture au clavier avec l'instruction `LIRE` provoque l'erreur numéro `003`.

Au niveau des lectures de valeurs de tableau, galg interdit la lecture directe. L'instruction

```
LIRE monTableau[ indC ]
```

est incorrecte et provoque l'erreur numéro `025`. Elle doit être corrigée en

```
LIRE eltC
AFFECTER monTableau[ indC ] <-- eltC
```

galg n'accepte pas d'écrire des expressions booléennes. L'instruction

```
ECRIRE " le test vaut " , x = 2
```

est donc considérée comme incorrecte et le numéro de l'erreur est `029`.

Pour l'appel d'un module, galg requiert des parenthèses, même si le module n'a pas de paramètres. Il ne faut donc pas écrire

```
APPELER initMatrices
```

sous peine de déclencher l'erreur `032`. Il faut écrire :

```
APPELER initMatrices()
```

Rappelons que le nom d'un module se compose d'un seul mot. Ecrire

```
APPELER Initialise Matrices()
```

déclenche l'erreur `034`. On peut utiliser le caractère de soulignement pour relier deux mots. `Initialise_Matrices` est ainsi un nom correct de module et on peut écrire `APPELER Initialise_Matrices()`.

L'ouverture de fichier a une syntaxe simple :

```
OUVRIR ... [EN_LECTURE | EN_ECRITURE] COME ...
```

Mais il arrive qu'on se trompe sur l'ordre des mots à écrire. Ainsi

```
OUVRIR "matrices.dat"  EN_LECTURE
```

déclenche l'erreur `055` et

```
OUVRIR "matrices.dat"  COMME fic_mat EN_LECTURE
```

déclenche l'erreur `056`.

Comme galg est pointilleux sur les structures emboitantes, mettre un `SI` sur la même ligne que le `ALORS` ou le `SINON` est considéré comme l'erreur `019`. En général, mettre un commentaire sur la même ligne que `ALORS` ou `SINON` permet de corriger cette erreur. Par exemple la partie d'algorithme

```
...
si f(x)>0
   alors si g(1+x)<0
...
```

peut être simplement corrigée en

```
...
si f(x)>0
   alors # cas f(x) positif
         si g(1+x)<0
...
```

L'erreur `036` dont le texte est `FIN DE STRUCTURE INCORRECTE` indique souvent qu'on a oublié le symbole de commentaire comme dans `fin_si x=1` ce qu'on ne doit pas confondre avec l'absence de commentaire, détecté par les erreurs `012`, `013`, `014` respectivement pour les fins de `POUR`, `SI` et `TANT_QUE`. Par exemple

```
fin_si
```

provoque l'erreur `013` puisque non commenté.

3.2 Erreurs délicates à corriger

L'erreur numéro 04 a pour texte : `INSTRUCTION ECRIRE INCORRECTE`. Cela peut provenir de l'instruction écrire sans aucun paramètre (ce qui, en Pascal, effectuerait l'affichage d'une ligne). Pour forcer l'affichage d'une ligne vide, on doit écrire une chaine vide avec l'instruction :

```
ECRIRE ""
```

Si ce n'est pas le cas, c'est à dire si on voulait afficher une variable ou un texte et qu'on a oublié de les indiquer, la correction est délicate car il faut trouver (ou se rappeler) ce qu'on voulait mettre derrière le mot `ECRIRE`.

Le message `PARTIE GAUCHE D'AFFECTATION INCORRECTE` de l'erreur 007 signifie qu'on a oublié de donner l'identificateur de la variable dans l'affectation, comme dans

```
AFFECTER <-- 1
```

Là encore, il est difficile de corriger car galg ne teste pas si les variables sont initialisées avant de les utiliser. Toutefois la liste des variables dans le fichier .lvm doit guider dans le mot à utiliser après `AFFECTER`.

Ne pas utiliser une seule expression dans l'affectation comme dans

```
AFFECTER maVar <-- "nous avons fait" , "un beau voyage"
```

fait afficher le texte `EXPRESSION UNIQUE CORRECTE REQUISE` qui correspond à l'erreur 008.

Une opération incorrecte comme dans

```
ECRIRE  f(  / 3 )
```

ou dans

```
AFFECTER x <--  g(  / 3 )
```

déclenche l'erreur 15 à ne pas confondre avec l'erreur de numéro 31 de texte `OPERATION, LISTE OU TEST NON TERMINE` obtenue avec l'instruction

```
AFFECTER x <--  g(  3 / )
```

ni avec l'erreur 030 de texte `EXPRESSION INCORRECTE OU CALCUL INCORRECT` obtenue par exemple avec

```
tant_que x > si x > 0
```

L'erreur 040 est difficile à corriger aussi puisqu'il doit manquer une variable ou une expression. Cette erreur apparait lorsqu'on écrit deux virgules à la suite dans une liste d'expressions, par exemple dans

```
APPELER  f(x,,z)
```

Même remarque en ce qui concerne l'erreur 043 qui se produit si on ne trouve rien entre les crochets pour un élément de tableau, comme dans

```
ECRIRE " Valeur deux : " , monTab[ ]
```

Si on utilise plusieurs mots à gauche du symbole d'affectation, comme dans

```
AFFECTER t[ i ]  j <-- 1
```

galg le signale avec l'erreur 044 mais là encore comment corriger ? Il ne s'agit pas d'une bête faut d'orthographe...

L'erreur 047 signifie aussi qu'il manque quelque chose après le symbole d'affectation, comme pour

```
AFFECTER j <--
```

ce qui ne peut se corriger sans savoir précisément ce que fait l'algorithme. Enfin, les caractères non autorisés comme ^ pour désigner la puissance déclenche l'erreur 049, ce qu'on peut vérifier en demandant à galg d'analyser un algorithme qui contient l'instruction

```
AFFECTER ind <-- 2 ^ 3
```

Les opérateurs "à la C" sont eux aussi interdits mais sont interprétés comme des opérations incomplètes, soit l'erreur 031 comme pour

```
AFFECTER  y <-- x++
```

Chapitre 4.

Mode "Traduction d'algorithmes"

4.1 Remarques générales sur la traduction

La traduction se fait avec galg grâce à l'option -o. Elle est effectuée ligne par ligne après l'analyse si aucune erreur n'a été détectée à l'analyse. galg décompose chaque instruction ou partie d'instruction en ses différents composants (conditions, variables...) et les traduit terme à terme.

Lorsque galg recontre des instructions ou des parties d'instruction imbriquantes, il rajoute systématiquement les débuts ou fin de bloc correspondantes, comme `do... end` en Rexx, `begin...end` en Pascal ou `{... }` en C, C++, Java même si ce n'est pas obligatoire.

Lorsque galg rencontre le commentaire spécial `#>` il recopie, à l'endroit où il est rendu, le fichier correspondant tel quel, sans aucune vérification. De même, lorsque galg rencontre le commentaire spécial `#:` il recopie, à l'endroit où il est rendu, le reste de la ligne, telle quelle, sans aucune vérification.

La traduction se fait en trois temps :

- d'abord une traduction mot à mot pour les instructions et leurs composants,
- ensuite une post-traduction des modules et fonctions commençant par le caractère de soulignement,
- enfin une post-comparaison terme à terme éventuelle via un fichier de correspondance de même nom que l'algorithme.

Par défaut, galg ne traduit aucune fonction, même les plus élémentaires. La fonction `LONGUEUR` par exemple, qui donne la longueur d'une chaine de caractères, sera recopiée telle que. Ainsi l' instruction

```
affecter lngPhrase  <-- longueur( Phrase )
```

aura pour traduction automatique

```
lngPhrase  =  longueur(  Phrase )    /* en Rexx */
lngPhrase  = &longueur( $Phrase ) ;  #  en Perl
```

Par contre, si on utilise la fonction `_LONGUEUR` avec le caractère de soulignement en début de nom, galg va traduire avec sa table de correspondance interne en :

```
lngPhrase  =  length(  Phrase )    /* en Rexx */
lngPhrase  =  length( $Phrase ) ;  /* en Perl */
```

Si maintenant pour la traduction en Rexx, on décide que `LONGUEUR` correspond au nombre de mots (ce qui se dit `WORDS` en Rexx), il suffit de mettre la ligne

```
longueur(              words(
```

dans la table de correspondance associée à l'algorithme.

Par exemple, si on traduit le fichier-algorithme `patterns.alg`, celle-ci correspond au fichier `patterns.tdc` et galg produit automatiquement la ligne

```
lngPhrase = words( Phrase )
```

Il est obligatoire de mettre dans le fichier de correspondance `longueur(` avec sa parenthèse ouvrante pour éviter que galg ne vienne remplacer d'autres identificateurs contenant la chaine de caractères *"longueur"*.

Si par contre le nom `longueur` convient, il faut donner sa définition en rajoutant par exemple en fin d'algorithme `#> longueur.rex` ou `#> longueur.pl` suivant le langage choisi pour la traduction.

Pour obtenir la liste des fonctions algorithmiques reconnues par un langage, rappelons qu'il suffit d'utiliser l'option `-f` de galg.

4.2 Algorithmes de référence pour traduction

La traduction effectuée par galg est une traduction automatique. Elle est donc mécanique, parfois imparfaite. Il y a souvent lieu de modifier légèrement l'algorithme de base pour profiter du langage sous-jacent. Afin de comparer la traduction pour les divers langages supportés par galg, nous utiliserons plusieurs algorithmes de référence.

Il serait fastidieux de fournir pour chaque algorithme sa version aménagée pour chaque langage. Nous nous contentons ici de détailler les algorithmes de référence, de fournir quelques indications sur leur aménagement car tous les algorithmes et leurs aménagements sont disponibles à l'adresse

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

Le premier, que nous nommerons "algorithme bonjour" est une version améliorée du fameux "Hello Word" de Kernighan et Richie :

```
##########################################################
#                                                        #
#  auteur : (gH)                                         #
#  bonjour.alg : un algorithme de bonjour                #
#                (avec dialogue et appels systèmes)      #
#                                                        #
##########################################################

# la demande = une question + une réponse

écrire   " Bonjour. Quel est ton nom ? "
lire     id

# ce qu'on fait répondre par la machine

écrire "Le " ,date(), " à " ,heure(), "au revoir" , maju(id)
```

Cet algorithme met en oeuvre les entrées et sorties au clavier, les appels de fonction `date` et `heure` ainsi que la conversion en majuscules, qu'on peut décider d'implémenter au choix soit avec la fonction correspondante du langage (quand elle existe) soit avec un sous-programme personnalisé, de façon à traduire les caractères accentués du français.

Le second algorithme, nommé "algorithme cfe" reprend la conversion de francs en euros de la section 2.1 mais avec une gestion de paramètres, la conversion d'une chaine en nombre, l'affichage sans saut de ligne, le calcul de la partie entière.

```
################################################
#                                              #
#   cfe.alg  ;  conversion francs en euro      #
#            -- auteur : gh                    #
#                                              #
################################################


# 1. Saisie ou conversion du paramètre

si nbParametres()=0
   alors écrire_  " Quel est le montant en Francs ? "
         lire     mntF
   sinon affecter mntF <-- valeur( parametre( 1 ) )
fin_si # nbParametres()=0


# 2. Conversion et affichage

affecter mntE <-- mntF / 6.55957
écrire   mntF , " Francs font " , mntE , " euros."
écrire   " et si on arrondit : " , partieEntiere( 0.5+mntE ), " euros."
```

Le troisième algorithme, nommé "algorithme macnaugthon" est exactement l'algorithme long de la section 2.6 d'affectation de tâches avec durée pour des machines. Il utilise des tableaux à une et deux dimensions, les fonctions `max`, `copies` et `format`.

Rappelons que la fonction `max(x,y)` renvoie le plus grand des deux entiers `x` et `y`, que la fonction `copies(c,r)` renvoie `r` copies de la chaine `c` et qu'enfin la fonction `format(n,l)` renvoie le nombre entier `n` cadré sur `l` caractères.

Le quatrième algorithme, nommé "algorithme maxoccmono" est exactement l'algorithme la section 2.5 qui calcule en une seule boucle le maximum d'un tableau d'entiers et le nombre de fois où ce maximum apparait.

Le cinquième algorithme, nommé "algorithme rchbrute" vient rechercher si un sous-tableau est présent dans un tableau selon la méthode de recherche brute (d'où le nom de l'algorithme) présenté par Sedgewick dans le chapitre 19 (*String Searching*) dans son ouvrage célèbre *Algorithms* paru chez Addison-Wesley.

```
############################################################
#                                                          #
#                                                          #
#  rchbrut.alg : brutesearch, auteur (gH)                  #
#                                                          #
#                                                          #
############################################################
#                                                          #
# source : Algorithms, second edition                      #
#          pages 279, 280                                  #
#          R. Sedgewick, Addison-Wesley 1988               #
#                                                          #
############################################################
#                                                          #
#                                                          #
#  recherche "brute" du tableau P dans le tableau A        #
#  (en anglais : P pour pattern, A pour array)             #
#  les variables sont renommées tabP et tabA               #
#  pour plus de lisibilité.   le tableau tabP contient     #
#  les codes ascii de la phrase                            #
#                                                          #
#      "A STRING SEARCHING EXAMPLE CONSISTING OF..."       #
#                                                          #
#  et le tableau tabA ceux du mot   "STING"                #
#                                                          #
#                                                          #
############################################################


affecter chP <--  "STING"
affecter chA <--  "A STRING SEARCHING EXAMPLE CONSISTING OF..."

# conversion des chaines en tableaux

affecter M <-- longueur(chP)
pour ind de1a M
  affecter tabP[ ind ] <--  codeAscii( sousChaine(chP,ind,1) )
fin_pour # ind de1a longueur(chP)
```

```
affecter N <-- longueur(chA)
pour ind de1a N
  affecter tabA[ ind ] <--  codeAscii( sousChaine(chA,ind,1) )
fin_pour # ind de1a longueur(chA)

# algorithme de recherche

affecter i <-- 1
affecter j <-- 1

répéter

  si tabA[i] = tabP[j]
     alors affecter i <-- i + 1
           affecter j <-- j + 1
     sinon affecter i <-- i - j + 2
           affecter j <-- 1
  fin_si # tabA[i] = tabP[i]

jusqu'à (j>M) ou (i>N)


# affichage du résultat

si j>M
   alors écrire chP , " vue dans " , chA , " à partir de la position " , (i-M)
   sinon écrire chP , " non vue dans ", chA
fin_si # j>M


# détail des tableaux (on suppose M < N)

écrire " Elément tableau P  tableau A "

pour ind de1a  N
   si (ind<=M)
      alors affecter milieu <-- concatene(
                    format(tabP[ind],8) , sousChaine(chP,ind,1) )
      sinon affecter milieu <-- copies(" ",10)
   fin_si # (ind<=M)
   écrire format(ind,3) , " : " , milieu , format(tabA[ind],8) ,
                                           sousChaine(chA,ind,1)
fin_pour # ind de1a N
```

L'algorithme rchbrute met en jeu les fonctions `format`, `codeAscii`, `concatene`, `copies`, `longueur`, `sousChaine`.

Rappelons que `codeAscii(c)` donne le code ascii du premier caractère de la chaine `c`, que `concatene(a,b)` met bout à bout les deux chaines `a` et `b`, que `longueur(c)` renvoie le nombre de caractères de la chaine `c` et enfin que `sousChaine(c,d,l)` renvoie la sous-chaine de `l` caractères pris dans la chaine `c` à partir du caractère numéro `d` (les caractères étant numérotés à partir de 1).

Une fois traduit, le programme correspondant doit afficher

```
STING  vue dans  A STRING SEARCHING EXAMPLE CONSISTING OF...
à partir de la position  33

 Elément tableau P  tableau A

  1  :        83 S       65 A
  2  :        84 T       32
  3  :        73 I       83 S
  4  :        78 N       84 T
  5  :        71 G       82 R
  6  :                   73 I
  7  :                   78 N
  8  :                   71 G
  9  :                   32
 10  :                   83 S
 11  :                   69 E
 12  :                   65 A
 13  :                   82 R
 14  :                   67 C
 15  :                   72 H
 16  :                   73 I
 17  :                   78 N
 18  :                   71 G
 19  :                   32
 20  :                   69 E
 21  :                   88 X
 22  :                   65 A
 23  :                   77 M
 24  :                   80 P
 25  :                   76 L
 26  :                   69 E
...
```

Le sixième et avant-dernier algorithme, nommé "algorithme tabmult" vient demander un nombre entier, relancer si le nombre n'est pas entier jusqu'à en obtenir un puis affiche la table de multiplication bien cadrée, unité sous unité, dizaine sous dizaine, etc. Il met en jeu la fonction `entier(n)` qui renvoie "vrai" si `n` est un entier.

```
#########################################################
#                                                       #
# tabmult.alg -- un exemple simple d'algorithme :       #
#                la table de multiplication             #
#                                                       #
#  auteur : gh                                          #
#                                                       #
#########################################################


# demande initiale

écrire  " Donner un entier "
lire nbChoisi


# relance éventuelle

tant_que (non entier(nbChoisi))
    écrire " nombre incorrect. Redonner un nombre ENTIER "
    lire nbChoisi
fin_tant_que # nombre invalide


# boucle d'affichage

écrire " Table de " , nbChoisi
pour indb de1a 10
    affecter produit <-- nbChoisi*indb
    affecter find    <-- format(indb,2,0)
    affecter fpro    <-- format(produit,5,0)
    écrire find , " fois " , nbChoisi , " = " , fpro
fin_pour # indb de1a 10
```

Enfin, le septième et dernier algorithme, nommé "algorithme trifusion" vient lire les fichier triFus.f1 et fichier triFus.f2 qui sont déjà triés sur leur deuxième mot et mettre dans le fichier triFus.f3 le fichier trié correspondant à la fusion des deux autres fichiers.

Par exemple si le fichier triFus.f1 contient

```
Pierre          ALBAN
Jacques         BOUYER
Irène           MALUN
```

et si le fichier triFus.f2 contient

```
Line            ARMA
Sophie          DAMART
Philippe        ROBIN
Serge           TARSKY
```

alors le fichier triFus.f3 devra contenir

```
Pierre          ALBAN
Line            ARMA
Jacques         BOUYER
Sophie          DAMART
Irène           MALUN
Philippe        ROBIN
Serge           TARSKY
```

Cet algorithme est de loin le plus délicat à traduire, compte-tenu de la particularité de certains langages à traiter ensemble la lecture sur fichier et le test de positionnement en fin de fichier. Nous avons tenu à traduire correctement et proprement les instructions

```
tant que non fin_de_fichier( f )
     lire ligne sur f
...
```

ce qui produit des traductions correctes mais souvent éloignées des traductions standards du C à savoir `while (fgets(ligne,LNG,f)!=NULL) {...` ou de Perl à savoir `while (<f>) { ...` (avec `$_` comme ligne lue).

Voici l'algorithme proposé

```
############################################################
#                                                          #
# triFus.alg : trifusion de 2 fichiers déjà triés          #
#              auteur -- gH                                #
#                                                          #
############################################################
#                                                          #
#  les deux fichiers sont censés contenir dans cet         #
#  ordre des prénoms et des noms ; les fichiers            #
#  d'entrée doivent être déja triés sur le nom, qui        #
#  est le deuxième mot de chaque ligne                     #
#                                                          #
############################################################


# nommage des variables pour noms de fichier

affecter nomFic1 <-- "triFus.f1"
affecter nomFic2 <-- "triFus.f2"
affecter nomFic3 <-- "triFus.f3"


# initialisation des nombres de lignes lues et écrites

affecter nbl1 <-- 0
affecter nbl2 <-- 0
affecter nbl3 <-- 0


# ouverture des 3 fichiers

ouvrir nomFic1 en_lecture  comme fic1
ouvrir nomFic2 en_lecture  comme fic2
ouvrir nomFic3 en_écriture comme fic3


# lecture du premier enregistrement sur fic1

si non fin_de_fichier(fic1)
   alors # lecture possible
         lire ligne1 sur fic1
         affecter nbl1 <-- nbl1 + 1
fin_si # lecture du fic1
```

```
# lecture du premier enregistrement sur fic2

si non fin_de_fichier(fic2)
   alors # lecture possible
         lire ligne2 sur fic2
         affecter nbl2 <-- nbl2 + 1
fin_si # lecture du fic1


# boucle de tri-fusion

tant_que (non fin_de_fichier(fic1)) et (non fin_de_fichier(fic2))

  si mot(ligne1,2) <= mot(ligne2,2)

     alors  # écriture de fic1 vers fic3
            écrire " (fic1) " , ligne1 sur fic3
            affecter nbl3 <-- nbl3 + 1
            si (non fin_de_fichier(fic1))
               alors # lecture possible
                     lire ligne1 sur fic1
                     affecter nbl1 <-- nbl1 + 1
               sinon affecter ligne1 <-- ""
            fin_si # lecture du fic1

     sinon  # écriture de fic2 vers fic3
            écrire " (fic2) " , ligne2 sur fic3
            affecter nbl3 <-- nbl3 + 1
            si (non fin_de_fichier(fic2))
               alors  # lecture possible
                     lire ligne2 sur fic2
                     affecter nbl2 <-- nbl2 + 1
               sinon affecter ligne2 <-- ""
            fin_si # lecture du fic2
  fin_si # comparaison ligne 1 et ligne 2

fin_tant_que # lecture sur les deux fichiers

# il est possible que la dernière lecture
# sur fic1 ou fic2 mène à fin_de_fichier ; il faut donc
# commencer par "épuiser" les lignes lues de
# part et d'autre
```

```
si (_longueur(ligne1)>0) et (_longueur(ligne2)>0) alors
          si mot(ligne1,2) <= mot(ligne2,2) alors
                    écrire " (fic1) " , ligne1 sur fic3
                    affecter nbl3 <-- nbl3 + 1
                    écrire " (fic2) " , ligne2 sur fic3
                    affecter nbl3 <-- nbl3 + 1
             sinon  # écriture de fic2 vers fic3
                    écrire " (fic2) " , ligne2 sur fic3
                    affecter nbl3 <-- nbl3 + 1
                    écrire " (fic1) " , ligne1 sur fic3
                    affecter nbl3 <-- nbl3 + 1
          fin_si # comparaison ligne 1 et ligne 2
fin_si # on compare en fin de fichier

# s'il reste des lignes dans f1, on recopie f1

tant_que non fin_de_fichier(fic1)
     lire   ligne1 sur fic1
     affecter nbl1 <-- nbl1 + 1
     écrire " [fic1] " , ligne1 sur fic3
     affecter nbl3 <-- nbl3 + 1
fin_tant_que # lecture sur f1

# s'il reste des lignes dans f2, on recopie f2

tant_que non fin_de_fichier(fic2)
     lire   ligne2 sur fic2
     affecter nbl2 <-- nbl2 + 1
     écrire " [fic2] " , ligne2 sur fic3
     affecter nbl3 <-- nbl3 + 1
fin_tant_que # lecture sur f1

# fermeture des fichiers

fermer fic1
fermer fic2
fermer fic3

# affichage du nombre de lignes lues et écrites :

écrire "    "  , format(nbl1,4) , " lignes lues    sur " , nomFic1
écrire "    "  , format(nbl2,4) , " lignes lues    sur " , nomFic2
écrire "    "  , format(nbl3,4) , " lignes écrites sur " , nomFic3
```

4.3 Traduction en Rexx

La traduction en Rexx pose en général une seule difficulté, celle de l'instruction `REPETER...JUSQU'A` car la condition du `JUSQU'A` doit être donnée dans la traduction de la ligne `REPETER`. Heureusement, galg détecte les conditions dans l'analyse et est capable de les restituer lors de la traduction.

La traduction sans aucune modification de l'algorithme bonjour n'est pas exécutable à cause de la dernière instruction traduite, à savoir

```
say " Le " date() " à " heure() " au revoir " maju(id)
```

car les fonctions `HEURE` et `MAJU` n'existent pas en Rexx.

Si l'on remplace la dernière instruction de l'algorithme par

```
écrire   " Le " , _date() , " à " , _heure() , " au revoir " , _maju(id)
```

c'est un peu mieux car l'instruction Rexx correspondante est éxécutable grâce à la table de correspondance interne :

```
say " Le " date("E") " à " time() " au revoir " translate(id)
```

Toutefois, nous préférons terminer l'algorithme par

```
écrire   " Le " , _date() , " à " , _heure() , " au revoir " , maju(id)
quitter 0
```

```
#> maju.rex
```

car alors nous avons une traduction des caractères accentués grâce à notre fonction `MAJU` comme le montre la fin de la traduction avec l'inclusion demandée (le fichier algorithme était nommé ici `rex03.alg`) :

```
   say " Le " date("E") " à " time() " au revoir " maju(id)
   exit 0

/* ##-#  Ajout de maju.rex via le code #>  */
   maju: procedure
         arg chen
   return translate(chen,"AACEEEIOUU","àâçéèêîôùû")
/* ##-#  Fin d'ajout de maju.rex via le code #>  */

/* ##-#   Fin de traduction pour rex03.rex via de galg -a rex03.alg -o rex */
```

Traduire en Rexx est souvent simple car Rexx est interprété et non typé explicitement. De plus il dispose de nombreuses fonctions. On ne s'étonnnera donc pas que la table de correspondance interne, consultable par la commande `galg -f rexx` contienne plus d'une dizaine de fonctions algorithmiques :

```
Table des 13 fonctions algorithmiques reconnues par le langage rexx :

              ALGORITHMIQUE                  |     REXX
                                             |
 1  / 13  :   _codeAscii()                   |     c2d()
 2  / 13  :   _date()()                      |     date("E")
 3  / 13  :   _entierAuHasard()              |     random()
 4  / 13  :   _format()                      |     format()
 5  / 13  :   _heure()()                     |     time()
 6  / 13  :   _longueur()                    |     length()
 7  / 13  :   _maju()                        |     translate()
 8  / 13  :   _max()                         |     max()
 9  / 13  :   _nbParametres()                |     arg()
10  / 13  :   _parametre()                   |     arg()
11  / 13  :   _partieEntiere()               |     trunc()
12  / 13  :   _sousChaine()                  |     substr()
13  / 13  :   _valeur()                      |     ()

 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
```

Pour traduire l'algorithme cfe, il suffit seulement de rajouter un caractère de soulignement devant chaque nom de fonction. La même remarque s'applique aux algorithmes macnaughton et maxoccmono.

Par contre, pour l'algorithme rchbrute, on ajoute en fin d'algorithme les lignes

```
quitter

#> concatene.rex
```

car la concaténation en Rexx est obtenue avec les symboles `||` et non pas par une fonction.

Le sous-programme utilisé, mis dans le fichier `concatene.rex` se réduit à :

```
concatene: procedure
   parse arg debut , fin
return debut||fin
```

Pour inclure ce sous-programme en Rexx, il est obligatoire d'utiliser l'instruction `QUITTER` avant de réaliser l'inclusion car sinon l'interpréteur Rexx signale une erreur, comme nous l'avions fait pour l'algorithme bonjour.

Pour l'avant-dernier algorithme, tabmult, il faut utiliser une table de correspondance externe afin de traduire la fonction `entier`. Comme Rexx dispose d'une fonction `datatype`, on met dans le fichier `tabmult.tdc` la ligne

```
entier(nbChoisi)          datatype(nbChoisi,"W")
```

et la traduction du fichier `tabmult.alg` produit automatiquement la ligne

```
do while (   \datatype(nbChoisi,"W"))
```

comme traduction de la ligne algorithmique

```
tant_que (non entier(nbChoisi))
```

Passons finalement à l'algorithme nommé triFusion. Rexx n'utilise pas de "poignées" ou *FileHandle* pur gérer les fichiers. Il est possible d'ouvrir un fichier en utilisant directement son nom, c'est pourquoi l'instuction algorithmique

```
ouvrir nomFic en_lecture comme fic
```

est traduite par galg en Rexx sous la forme

```
fic = nomFic
```

et la même traduction est utilisée pou traduire

```
ouvrir nomFic en_ecriture comme fic
```

puisque Rexx n'effectue l'ouverture véritable du fichier seulement lors de la première opération de lecture ou d'écriture sur le fichier.

Pour traduire la fonction `fin_de_fichier`, nous utilisons la fonction Rexx nommée *lines* qui renvoie pour $lines(f)$ le nombre de lignes restant à lire dans f.

L'instruction

```
tant_que (non fin_de_fichier(fic))
```

est alors traduite en

```
do while \fin_de_fichier(fic1)
```

où `fin_de_fichier` est une fonction externe chargée par

```
#> fdf.rex
```

en fin d'algorithme et dont le contenu est :

```
fin_de_fichier: procedure
  parse arg nomfic
return lines(nomfic)=0
```

Il n'y a pas besoin d'inventer une traduction pour la fonction `mot` car elle existe en Rexx : on écrit donc `_mot` pour qu'elle soit gérée par la table de correspondance interne.

Pour en finir avec la traduction en Rexx, signalons que l'appel des sous-programmes tels que nous les conseillons en algorithmique restreignent un peu la puissance du "parser" d'arguments de Rexx. Celui prend en principe l'ensemble des arguments sous forme d'une chaine et permet de les redécouper. Le style algorithmique impose de séparer les paramètres par des virgules, ce qui impose le même séparateur en Rexx.

Par exemple, l'instruction algorithmique

```
appeler affiche_Tab("monT." , 1 , nbElt , 4)
```

oblige la déclaration suivante du module `affiche_Tab`

```
affiche_Tab: procedure expose (nomTab)
parse arg nomTableau , indMin , indMax , largeurF
```

Avec une écriture Rexx traditionnelle, on aurait plutot écrit l'appel

```
affiche_Tab( "monT." 1 nbElt 4)
```

avec comme définition des paramètres

```
parse arg  nomTableau  indMin  indMax  largeurF
```

4.4 Traduction en Perl

Pour traduire l'algorithme bonjour en Perl, il faut traduire trois fonctions : `date`, `heure` et `maju`. Pour `date` et `heure`, Perl ne dispose pas de fonctions équivalentes en standard. Il faut les écrire (en Perl) puis les inclure grâce aux lignes

```
#> date.pl
#> heure.pl
```

mises en fin d'algorithme. Un contenu possible pour ces fichiers est :

```
## -- fichier date.pl

sub date { # présente proprement la date d'aujourdhui

   ($sec,$min,$hour,$mday,$mmon,$year,$jda)=localtime();
   $mmon = $mmon + 1 ;
   $year = $year + 1900 ;
   if (length($sec)<2)  { $sec = "0$sec" } ;
   if (length($mday)<2) { $mday = "0$mday" } ;
   if (length($mmon)<2) { $mmon = "0$mmon" } ;

   return "$mday/$mmon/$year" ;

 } ; # fin sub date


## -- fichier heure.pl

sub heure { # présente proprement l'heure courante

   ($sec,$min,$hour,$mday,$mmon,$year,$jda)=localtime();
   $mmon = $mmon + 1 ;
   $year = $year + 1900 ;
   if (length($sec)<2)  { $sec = "0$sec" } ;
   if (length($mday)<2) { $mday = "0$mday" } ;
   if (length($mmon)<2) { $mmon = "0$mmon" } ;

   return "$hour:$min.$sec" ;

 } ; # fin sub heure
```

Pour traduire la fonction `maju`, on on peut se contenter de la fonction Perl nommée `uc` (comme Upper Case). Il suffit alors de mettre un caractère de soulignement devant le mot `maju`. Les caractères accentués ne sont alors pas traduits.

L'algorithme complet correspondant est alors

```
#####################################################################
#                                                                   #
#                                                                   #
#  auteur : (gH)                                                    #
#  bonjour.alg : un algorithme de bonjour conséquent.               #
#                                                                   #
#  -- ADAPTATION POUR TRADUCTION EN PERl PAR GALG --                #
#                                                                   #
#####################################################################
#                                                                   #
#                                                                   #
#                                                                   #
#      cet algorithme comporte un dialogue où on demande à          #
#      l'utilisateur de donner son nom ;                            #
#      il affiche ensuite la date et l'heure puis dit au            #
#      revoir à la personne après avoir le nom en majuscules.       #
#                                                                   #
#                                                                   #
#                                                                   #
#####################################################################


   # la demande = une question + une réponse

   écrire   " Bonjour. Quel est ton nom ? "
   lire     id


   # ce qu'on fait répondre par la machine

   écrire   " Le ", date(), " à ", heure(), " au revoir " , _maju(id)

#> date.pl
#> heure.pl
```

Si on veut inventer une fonction `maju` "à la française", il faut écrire une fonction Perl dans le fichier `maju.pl` et l'inclure par la ligne

```
#> maju.pl
```

à la suite des autres lignes d'inclusion. Un contenu possible pour ce fichier `maju.pl` est :

```
## -- fichier maju.pl

sub maju {
  # passe en majuscules y compris les caractères accentués

  my $parm = $_[0] ;

  $parm =~ s/à/A/g ;
  $parm =~ s/â/A/g ;
  $parm =~ s/ç/C/g ;
  $parm =~ s/é/E/g ;
  $parm =~ s/è/E/g ;
  $parm =~ s/ê/E/g ;
  $parm =~ s/î/I/g ;
  $parm =~ s/ô/O/g ;
  $parm =~ s/ù/U/g ;
  $parm =~ s/û/U/g ;

  return uc($parm) ;

} ; # fin sub maju
```

Pour traduire l'algorithme cfe en Perl, il faut traduire les quatre fonctions `nbParametres`, `valeur`, `parametre`, et `partieEntiere`.

La table de correspondance interne, consultable par la commande `galg -f perl` et dont le contenu est :

```
Table des  5 fonctions algorithmiques reconnues par le langage perl :

                ALGORITHMIQUE                |     PERL
                                             |
 1  /  5  :   &_codeAscii()                  |     ord()
 2  /  5  :   &_longueur()                   |     length()
 3  /  5  :   &_maju()                       |     uc()
 4  /  5  :   &_partieEntiere()              |     int()
 5  /  5  :   &_valeur()                     |     ()
```

montre qu'il suffit de mettre un caractère de soulignement devant les mots `valeur` et `partieEntiere` pour que galg sache traduire ces fonctions.

Pour traduire les deux autres fonctions, on vient inclure les fichiers correspondants grâce aux lignes

```
#> nbParametres.pl
#> parametre.pl
```

mises en fin d'algorithme. Un contenu possible pour ces fichiers est :

```
## -- fichier nbParametres.pl
sub  nbParametres  {
  return( 1 + $#ARGV ) ;
} ; # fin sub nbParametres

## -- fichier parametre.pl
sub  parametre  {
  return( $ARGV[ -1 + $_[0] ] ) ;
} ; # fin sub parametre
```

La traduction en Perl de l'algorithme macnaughton met en jeu les trois fonctions `max`, `format` et `copies`. Là encore cs fonctions n'existent pas en Perl mais elles sont très simples à inventer. On vient mettre en fin de fichier algorithme les lignes

```
#> max.pl
#> format.pl
#> copies.pl
```

Un contenu possible pour les fichiers correspondants est :

```
## -- fichier max.pl
sub  max  {
  if ($_[0]>$_[1]) { return $_[0] ; } else { return $_[1] ; } ;
} ; # fin sub max

## -- fichier format.pl
sub  format  { return( sprintf("%$_[1]d",$_[0]) ) ; } ;

## -- fichier copies.pl
sub  copies {  return( $_[0] x $_[1] ) ; } ;
```

Lors de la traduction de l'algorithme maxoccmono, galg a besoin des deux fonctions `format` et `entierAuHasard`. La fonction `format` a été présentée pour la traduction de l'algorithme précédent. Pour `entierAuHasard`, on peut inclure le fichier suivant

```
## -- fichier entierAuHasard.pl
sub  entierAuHasard  {

 my ($hvmin,$hvmax) = @_ ;
 return int($hvmin + rand($hvmax-$hvmin+1))  ;

} ; # fin sub entierAuHasard
```

car Perl dispose d'une fonction `rand` qui renvoie un nombre réel au hasard entre 0 et 1.

Pour traduire l'algorithme rchbrute, il faut traduire les fonctions `codeAscii`, `concatene`, `copies`, `format`, `longueur` et `sousChaine`.

Les fonctions `codeAscii` et `longueur` ne posent pas de problème : renommées respectivement `_codeAscii` et `_longueur`, elles sont remplacées par leur équivalent Perl grâce à la table de correspondance interne.

Les fonctions `copies` et `format` ont déja été vues lors de la traduction de l'algorithme précédent. Pour `concatene` on vient inclure le fichier

```
## -- fichier concatene.pl
sub  concatene  { # met bout à bout deux chaines
  return( join(" ",@_) ) ;
} ; # fin sub concatene
```

Pour `sousChaine`, on vient inclure le fichier `sousChaineZero.pl`. Volontairement, le nom de ce fichier n'est pas `sousChaine.pl` car Perl compte les caractères à partir de zéro. Ce fichier contient :

```
## -- fichier sousChaineZero.pl
sub  sousChaine  {
  return( substr($_[0],$_[1]-1,$_[2]) ) ;
} ; # fin sub sousChaine
```

L'avant-dernier algorithme, tabmult utilise les fonctions `format` et `entier`. Le cas de `format` a déjà été vu précédemment. Pour `entier`, on peut inclure le fichier

```
## -- fichier entier.pl
sub  entier  {
  # renvoie "vrai" si le paramètre est une chaine
  # qui ne contient qu'un entier éventuellement signé
  my $parm_entier = $_[0] ;
  return $parm_entier =~ /^[+-]?\d+$/ ;
} ; # fin sub entier
```

qui exploite la gestion des expressions régulières en Perl.

Passons maintenant à l'algorithme triFusion. Comme Perl dispose de raccourcis pour les opérations sur fichiers, nous devons inclure le module *FileHandle* avec la ligne

```
#: use FileHandle ;
```

en début d'algorithme afin de disposer de fonctions "lisibles" équivalentes à nos instructions algorithmiques. Ainsi

```
fermer f
```

devient assez naturellement

```
$f->close()
```

Comme notre algorithme utilise les fonction `format` et `mot`, on vient inclure, grâce aux lignes

```
#> format.pl
#> mot.pl
```

une définition de ces fonctions. Comme Perl n'utilise pas les mêmes opéarteurs pour la comparaison de chaines et de nombres, on remplace l'instuction algorithmique

```
si mot(ligne1,2) <= mot(ligne2,2)
```

par l'instruction

```
si chaineAvant(mot(ligne1,2),mot(ligne2,2)) = 1)
```

et on vient inclure également en fin d'algorithme la définition de la fonction correspondante par `#> chaineAvant.pl`.

Voici le détail des fonctions utilisées :

```
sub mot { # Retourne le MOT n dans la PHRAse ;
          # verison courte : le seul séparateur est l'espace
  my @TPhraz;
  @TPhraz=split(" ", $_[0]);
  return $TPhraz[$_[1]-1];
} ; # fin de sub mot


sub  chaineAvant  {
  my $val ;
  if  ($_[0] le $_[1]) { $val = 1 ; } else { $val = 0 ; } ;
  return( $val ) ;
} ; # fin sub chaineAvant
```

Terminons ce passage en revue de la traduction en Perl par l'inclusion de modules. Perl permet de regrouper des définitions de fonctions dans des fichiers modules (d'extension `.pm`). Par exemple nous avons regroupé sur le site *Web* de galg toutes les fonctions présentées pour les six exemples dans le module `algoFuncs.pm`. Il suffit alors d'indiquer qu'il faut utiliser le module par la ligne

```
#: use algoFuncs ;
```

Ainsi, la traduction de l'algorithme bonjour, au lieu d'être

```
écrire   " Bonjour. Quel est ton nom ? "
lire     id
écrire   " Le ", date(), " à ", heure(), " au revoir " , maju(id)

#> date.pl
#> heure.pl
#> maju.pl
```

devient le fichier

```
#: uses algoFuncs.pm ;

écrire   " Bonjour. Quel est ton nom ? "
lire     id
écrire   " Le ", date(), " à ", heure(), " au revoir " , maju(id)
```

On y gagne en concision, en nombre de fichiers mais encore faut-il savoir écrire des modules et "exporter" les noms de fonctions...

4.5 Traduction en Tcl/Tk

Pour traduire l'algorithme bonjour en Tcl/Tk, il suffit de rajouter la définition des fonctions `date`, `heure` et `maju` via les lignes

```
#> date.tcl
#> heure.tcl
#> maju.tcl
```

Ces lignes doivent être mises avant la première instruction de l'algorithme (par exemple après le premier bloc de commentaires) parce que Tcl/Tk a besoin de connaitre les sous-programmes avant le programme principal. Une autre solution est de mettre

```
#: source date.tcl
#: source heure.tcl
#: source maju.tcl
```

ce qui indique à Tcl/Tk de charger les fichiers correspondants. Dans ce cas, le fichier traduit ne contient pas la définition des sous-programmes mais seulement une demande de chargement dynamique.

Voici un contenu possible pour ces fichiers :

```
######################################## date.tcl

proc date { }  {

 set  ladate [ clock format [clock seconds] -format " %d/%m/%y " ]
 return $ladate

} ; # fin de date

######################################## heure.tcl

proc heure { }  {

 set  lheure [ clock format [clock seconds] -format " %H:%M" ]
 return $lheure

} ; # fin de heure
```

```
###################################### maju.tcl

proc maju { chaine }  {

 regsub -all "à" $chaine "A" chaine
 regsub -all "â" $chaine "A" chaine
 regsub -all "ç" $chaine "C" chaine
 regsub -all "è" $chaine "E" chaine
 regsub -all "é" $chaine "E" chaine
 regsub -all "ê" $chaine "E" chaine
 regsub -all "î" $chaine "I" chaine
 regsub -all "ô" $chaine "O" chaine
 regsub -all "ù" $chaine "U" chaine
 regsub -all "û" $chaine "U" chaine

 return [string toupper $chaine]

} ; # fin de maju
```

Si on veut utiliser la traduction en majuscules du langage (instruction `[string toupper ...]` il faut écrire `_maju` au lieu de `maju` dans l'algorithme et retirer la ligne d'inclusion de `maju.tcl`.

Pour traduire l'algorithme cfe, on rajoute un caractère de soulignement devant la fonction `nbParametres()` car cette fonction est dans la table de correspondance de Tcl/Tk (ce qu'on peut vérifier en tapant `galg -f tcl`) et on inclut les fonctions `parametre`, `valeur` et `partieEntiere`.

Malheureusement Tcl/Tk requiert un paramètre supplémentaire pour la fonction `parametre` : la syntaxe algoritmique pour le i-ème paramètre est `parametre( i )` alors qu'en Tcl/Tk, on doit utiliser `[parametre $argv $i]` en précisant que la liste des paramètres est `$argv`. Pour arriver à cette traduction, il faut donc utiliser un fichier de post-traduction. Si l'algorithme est dans le fichier `cfeTcl.alg`, ce fichier de post-traduction (ou fichier de correspondance externe) est nommé `cfeTcl.tdc` et contient

```
[parametre          [parametre $argv
```

Il est à noter que cette modification est effectuée après la traduction ligne à ligne effectuée par galg. C'est pourquoi on remplace des expressions Tcl/Tk par d'autres expressions Tcl/Tk et non pas des expressions algorithmiques.

Pour traduire l'algorithme macnaughton, on doit définir la fonctions `copies` et la fonction `format`. Comme il existe déjà une fonction `format` en Tcl/Tk, on mettra `_format` au lieu de `format`. Le début du fichier-algorithme contient donc

```
#> copies.tcl
#> _format.tcl
```

Le contenu de ces fichiers peut être :

```
######################################## copies.tcl

proc copies { chaine nb } {
  set repete ""
  fo  {set ind 1} {$ind <= $nb}  {set ind [ incr ind]} {
    set repete "$repete$chaine"
  } ; # fin de pour ind
  return $repete
} ; # fin proc copies

######################################### _format.tcl

proc  _format  { nombre longueur } {
  set chformat [append "%" $longueur "d"]
  return [format %$chformat $nombre]
} ; # fin proc _format
```

Le même sous-programme `_format.tcl` doit être utilisé pour l'algorithme maxoccmono. Il faut lui adjoindre le sous-programme `entierAuHasard`, ce qui oblige donc à inclure en début d'algorithme les lignes

```
#> copies.tcl
#> _format.tcl
```

Le contenu du fichier pour la génération d'un nombre aléatoire entre les deux bornes passées en paramètres peut ressembler à

```
######################################## entierAuHasard.tcl

proc  entierAuHasard  { vmin vmax } {
  return [ expr int($vmin + ($vmax-$vmin)*rand()) ]
} ; # fin proc entierAuHasard
```

Passons maintenant à l'algorithme rchbrute. On doit encore y adjoindre les fonctions `_format`, `copies` mais aussi les fonctions `sousChaine`, `codeAscii` et `concatene`, ce qui oblige donc à inclure en début d'algorithme les lignes

```
#> sousChaine.tcl
#> codeAscii.tcl
#> concatene.tcl
#> copies.tcl
#> _format.tcl
```

dont le texte peut être :

```
######################################## sousChaine.tcl
proc  sousChaine  { chaine debut longueur } {
  return [string range $chaine [expr $debut-1] \
                               [expr $debut+$longueur-2] ]
} ; # fin proc sousChaine

######################################## codeAscii.tcl
proc  codeAscii  { caract } {
   return [scan $caract "%c"]
} ; # fin proc codeAscii

######################################## concatene.tcl
proc  concatene  { debut fin } {
  return "$debut$fin" ;
} ; # fin proc concatene
```

Par contre, la fonction `longueur` existe dans la table de correspondance interne et doit être écrite `_longueur`. Comme de plus il n'y pas de structure REPETER en Tcl/Tk, on la remplace par une boucle TANT QUE, c'est à dire que les lignes

```
répéter
    ...
jusqu'à (j>M) ou (i>N)
```

sont remplacées par

```
tant_que (j <= M) et (i <= N)
   ...
fin_tant_que # (j<=M) et (i<=N)
```

Pour l'algorithme tabmult, à part la fonction `_format` on doit insérer la fonction `entier` par la ligne

```
#> entier.tcl
```

dont le texte peut être :

```
proc  entier  { chaine } {
  set res  [regexp "^\[0-9\]+$" $chaine]
  return [expr $res==1]
} ; # fin proc entier
```

Enfin, pour l'algorithme triFus, on invente les fonctions `eof` et `mot` qu'on vient inclure en début d'algorithme par

```
#> mot.tcl
#> eof.tcl
```

La comparaison de chaines se faisant avec `[string compare`, on utilise l'instruction

```
si _compareChaine( mot(ligne1,2) , mot(ligne2,2) ) < 1
```

au lieu de l'instruction

```
si mot(ligne1,2) <= mot(ligne2,2)
```

Comme de plus, Tcl/Tk indique la fin de fichier après avoir lu la variable, on rajoute un test sur la longueur de la chaine afin d'être sur d'avoir lu quelquecchose. Ainsi, au lieu de `lire ligne sur fic` on écrit

```
lire ligne sur fic
si _longueur(ligne>0) alors ...
```

Signalons enfin qu'il est possible avec l'instruction `source` de charger un fichier de sous-progammes. Par exemple, si on met tous les sous-programmes présentés dans le fichier `algoFuncs.tcl`, il suffit de ne mettre que la ligne

```
#: source algoFuncs.tcl
```

en début d'algorithme pour que tous les sous-programmes soient chargés. On trouvera ce fichier, comme tous les algorithmes modifiés pour Tcl/Tk sur notre site *Web* à l'adresse

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

4.6 Traduction en Dbase

Avant de traduire les algorithmes de référence en Dbase, il faut noter que notre version de Dbase, gratuite sous *Dos* et sous *Unix* est Max que l'on peut trouver sur le site

`http://www.plugsys.com/`

Cette version de Dbase impose quelques restrictions puisqu'il faut notamment ne pas utiliser des noms de fichiers de plus de 10 caractères, utiliser des noms de fichiers en minuscules seulement. Par contre cette version fonctionne sous *Dos* et sous *Unix*, ce qui n'est pas encore le cas pour *Db2k*, la verison "officielle" de Dbase pour *Windows*.

Ainsi l'algorithme d'affectation des tâches qui selon nos conventions aurait du être nommé `macnaughtonDbase.alg` est renommé en `macndb.alg`.

Pour l'algorithme bonjour, les fonctions `date` et `heure` existent, mais n'affichent l'année sur 4 chiffres et l'heure sur 24 heures qu'avec des options `SET ... TO...` ; il faut donc mettre en début d'algorithme

```
#: set date  to french
#: set hours to 2
```

pour obtenir un "bel" affichage.

Si on écrit la fonction `maju` avec un caractère de soulignement, la fonction `UPPER` est utilisée grâce à la table de correspondance interne (la liste des fonctions traduites est accessible via la commande `galg -f dbase`) mais si on laisse le mot `maju` tel quel, il faut en fin d'algorithme mettre la définition de la fonction par exemple dans le fichier `maju.prg` qu'on inclut par la ligne `#> maju.prg`. Voici un contenu possible pour ce fichier :

```
function maju

  parameter chaineAvant
  store trim(chaineAvant) to chaineAVant
  store "" to chaineApres
  store len(chaineAvant) to longCh
  for i := 1 to len(chaineAvant)
      store substr(chaineAvant,i,1) to carCour
      store asc(carCour) to codCour

      do case
      case codCour = 133 // "à"
```

```
            carCour = "A"
         case codCour = 131 // "â"
            carCour = "A"
         case codCour = 135 // "ç"
            carCour = "C"
         case codCour = 130 // "é"
            carCour = "E"
         case codCour = 138 // "è"
            carCour = "E"
         case codCour = 136 // "ê"
            carCour = "E"
         case codCour = 140 // "î"
            carCour = "I"
         case codCour = 147 // "ô"
            carCour = "O"
         case codCour = 150 // "û"
            carCour = "U"
         case codCour = 151 // "ù"
            carCour = "U"
         endcase
         store chaineApres+carCour to chaineApres
   next
return upper(chaineApres)
```

Il n'y a aucun fichier à inclure pour l'algorithme `cfe` car toutes les fonctions peuvent être traduites via la table de correspondance interne (la liste des fonctions traduites est accessible via la commande `galg -f dbase`) à condition de leur ajouter le caractère de soulignement en début de fonction. On pourra, s'il l'on veut, ajouter

```
#: set decimals to 5
```

pour n'afficher que 5 décimales. Par contre il est obligatoire de mettre

```
#: parameters parametre1
```

pour que le paramètre éventuel soit pris en compte.

Voici l'algorithme de conversion des francs en euros :

```
################################################
#                                              #
#   cfe.alg  ;  conversion francs en euro      #
#            -- auteur : gh                    #
#                                              #
################################################

#: set decimals to 5
#: parameters parametre1

# 1. Saisie ou conversion du paramètre

affecter nbp <--  _nbParametres()

si nbp = 0
   alors écrire_  " Quel est le montant en Francs ? "
         lire     mntF
   sinon # il ne faut pas mettre d'instruction sur la ligne
         # du sinon en Dbase
         affecter mntF <-- _valeur( _parametre( 1 ) )
fin_si # nbParametres()=0

# 2. Conversion et affichage

affecter mntE <-- mntF / 6.55957
écrire   mntF , " Francs font " , mntE , " euros."
affecter a_mntE <-- _partieEntiere( 0.5 + mntE )
écrire   " et si on arrondit : " , a_mntE , " euros."
```

On remarquera qu'il n'y a qu'un commentaire sur la ligne du sinon car Dbase ignore ce qui suit le mot sinon.

Pour l'algorithme macnaugthon, on vient d'abord renommer le fichier-texte de l'algorithme en `macndb.alg` car comme déjà signalé, une restriction imposée par *Max* est de ne pas utiliser des noms longs. Les tableaux doivent être déclarés grâce aux instructions

```
#: declare dureeTache [ nbTaches ]
#: declare nbTache [ nbTaches ]
#: declare machineTacheNumero[ nbTaches , nbMachines ]
#: declare machineTacheDuree[  nbTaches , nbMachines ]
```

dès que les variables `nbTaches` et `nbMachines` ont une valeur. Les fonctions mises en jeu (`copies`, `format` et `max`) ne requièrent aucun fichier à charger ; par contre il faut mettre un caractère de soulignement devant `copies` et `format` pour que les fonctions correspondantes de Dbasesoient reconnues, à savoir `replicate` et `str`.

On renomme aussi le fichier `maxoccmonoDbase.al` en `mxomdb.alg` de façon à traduire l'algorithme maxoccmono dans un fichier de nom court. Pour que la traduction automatique de cet algorithme en Dbase soit possible, il faut déclarer le tableau utilisé par

```
#: declare monT[ nbElt ]
```

une fois qu'on donné une valeur à la variable `nbElt` et rajouter en fin d'algorithme

```
#> entierAuHasard.prg
```

afin de calculer automatiquement des nombres au hasard. Le contenu de ce fichier peut être :

```
function entHasard

  parameters vmin , vmax
  store vmin + rand( vmax-vmin ) to _le_nombre_

return _le_nombre_
```

ce qui signifie que nous avons aussi modifié la fonction `entierAuHasard` en `entHasard` toujours pour une raison de longueur d'identificateur.

Pour adapter l'algorithme rchbrute mis dans le fichier `rchdb.alg`, il faut

- déclarer les tableaux `tabP` et `tabA`,
- mettre un caractère de soulignement devant les fonctions `codeAscii`, `sousChaine` et `format` pour qu'elles correspondent respectivement à `asc`, `substr` et `str`,
- transformer la boucle `REPETER` en boucle `TANT_QUE`,
- insérer le fichier `concatene.prg`.

Ce dernier fichier, `concatene.prg`, contient :

```
function concatene
         parameters debut , fin
return debut + fin
```

Terminons la traduction en Dbase avec l'algorihtme tabmult mis dans le fichier `tdmdb.alg`. Pour profiter des masques d'écran de Dbase, nous remplaçoons les lignes de demande initiale et de relance éventuelle par des commandes `get` et `say` soit le texte :

```
affecter nbChoisi <-- 0
#: @ 3,05  say  " Donner un entier "
#: @ 3,25  get nbChoisi picture "999"
#: read
```

Le reste de l'algorithme est alors traduit sans aucun changement autre que l'ajout du caractère de soulignement en au début du mot `format`.

Si on veut completement masquer la saisie dans l'algorithme on met les lignes précédentes dans un fichier nommé par exemple `demandeN.prg` et on se contente de mettre en début d'algorithme la seule ligne

```
#> demandeN.prg
```

avant la boucle d'affichage.

4.7 Traduction en Pascal

Pascal est le premier des langages compilables que nous présentons ici. La version du compilateur utilisée est `ppc386`, suffisamment compatible avec Turbo Pascal Dos, Turbo Pascal Objet, Turbo Pascal Windows et Delphi pour pouvoir être utilisée un peu partout.

Les instructions d'un programme Pascal s'écrivent entre les mots `Begin` et `End.` et la première instruction d'un programme doit être le mot `Program`. galg déduit du nom du fichier le nom du programme : le fichier `test.alg` produira donc comme première ligne de programme l'instruction `PROGRAM test` (comme Pascal ne fait pas la distinction entre majuscules et minuscules, `PROGRAM` et `program` sont équivalents).

Il y a souvent une partie déclaration entre `PROGRAM` et le `BEGIN` du programme principal. C'est pourquoi galg laisse le soin à l'utilisateur de placer le mot `BEGIN`, soit par une insertion "en ligne" avec le commentaire spécial `#:` soit en insérant tout le texte de début de ligne avec le commentaire spécial `#>`.

La même remarque s'applique aux unités précompilées et de de `ppc386` et en particulier pour l'unité `Linux`. C'est pourquoi la ligne `#: Uses linux ;` doit souvent faire partie du début de l'algorithme.

En ce qui concerne la traduction de l'algorithme `bonjour`, les fonctions `date`, `heure` et `maju` doivent être insérées soit l'algorihtme :

```
#####################################################################
#                                                                   #
#                                                                   #
#  auteur : (gH)                                                    #
#  bonjour.alg : un algorithme de bonjour conséquent.               #
#                                                                   #
#                                                                   #
#####################################################################
#                                                                   #
#                                                                   #
#      cet algorithme comporte un dialogue où on demande à          #
#      l'utilisateur de donner son nom ;                            #
#      il affiche ensuite la date et l'heure puis dit au            #
#      revoir à la personne après avoir le nom en majuscules.       #
#                                                                   #
#                                                                   #
#####################################################################

#: Uses linux ;
```

```
#> date.pas
#> heure.pas
#> maju.pas

#: var id : string ;

#:BEGIN


   # la demande = une question + une réponse

   écrire_   " Bonjour. Quel est ton nom ? "
   lire      id


   # ce qu'on fait répondre par la machine

   écrire   " Le ", date(), " à ", heure(), " au revoir " , maju(id)
```

Le détail des fonctions ajoutées peut être :

```
     (* fichier date.pas *)

     function date : string ;

          var Year, Month, Day : word ;
              chaineY, chaineM, chaineD : string ;

     begin

      GetDate (Year, Month, Day) ;
      str(Year,chaineY) ;
      str(Month,chaineM) ;
      str(Day,chaineD) ;
      if length(chaineD) = 1 then chaineD := '0' + chaineD ;
      if length(chaineM) = 1 then chaineM := '0' + chaineM ;

      date := chaineD+'/'+chaineM+'/'+chaineY ;

     end ;
```

```
(* fichier heure.pas *)

function heure : string ;

     Var Hour, Minute, Second : Word ;
         chaineH, chaineM, chaineS : string ;
begin

 GetTime (Hour, Minute, Second);
 str(Hour,chaineH) ;
 str(Minute,chaineM) ;
 str(Second,chaineS) ;
 if length(chaineH) = 1 then chaineH := '0' + chaineH ;
 if length(chaineM) = 1 then chaineM := '0' + chaineM ;
 if length(chaineS) = 1 then chaineS := '0' + chaineS ;

 heure := chaineH+':'+chaineM+'.'+chaineS ;

end ;

(* fichier maju.pas *)

function maju(avant : string ) : string ;
     var ind   : integer ;
         apres : string  ;
         carc  : char    ;
begin
 apres := '' ;
 for ind := 1 to length(avant) do begin
     carc  :=  upcase( avant[ ind ] ) ;
     if carc = 'à' then carc := 'A' ;
     if carc = 'â' then carc := 'A' ;
     if carc = 'ç' then carc := 'C' ;
     if carc = 'é' then carc := 'E' ;
     if carc = 'è' then carc := 'E' ;
     if carc = 'ê' then carc := 'E' ;
     if carc = 'î' then carc := 'I' ;
     if carc = 'ô' then carc := 'O' ;
     if carc = 'ù' then carc := 'U' ;
     if carc = 'û' then carc := 'U' ;
     apres := apres + carc ;
 end ;
 maju := apres ;
end ;
```

L'algorithme cfe requiert en début d'algorithme :

```
#: Uses linux ;

#> valeur.pas
#> format.pas

#: var mntF,mntE : real ;
#:     mntA : integer ;
```

de façon à déclarer les variables et à inclure les sous-programmes `valeur` et `format`. La fonction `nbParametres` est connue dans la table interne de correspondance (et doit donc être écrite `_nbParametres` avec un caractère de soulignement en début pour que galg la traduise correctement en `paramcount`). Les fonctions `parametre` et `partieEntiere` sont également reconnues. Par contre la fonction `format` doit être fournie extérieurement comme la fonction `valeur`, baties autour des fonctions Pascal standards `str` et `val`.

Un contenu possible pour ces fonctions est donc :

```
(* fichier format.pas *)

function format( nombre : integer ; longueur : integer ) : string ;
  var chaine : string ;
begin
  str(nombre:longueur,chaine) ;
  format := chaine ;
end ;

(* fichier valeur.pas *)

function valeur( chaine : string) : real ;
 var rc : integer ;
     nombre : real ;
begin
 val(chaine,nombre,rc) ;
 if rc=0 then valeur := nombre
         else valeur := -999.99 ;
end ;
```

On notera que comme la fonction `valeur` doit renvoyer un nombre, on a décidé arbitrairement de renvoyer `-999.99` si la chaine transmise ne correspond pas à un nombre.

En plus de la fonction `format` déjà vue, il faut inclure la fonction `max` et la fonction `copies` pour traduire l'algorithme `macnaugthon`. De plus il y a de nombreuses variables.

On pourra donc au choix soit regrouper toutes les déclarations dans un fichier, par exemple `macn.var` et se contenter d'écrire

```
#> max.pas
#> format.pas
#> copies.pas

#> macn.var

#: BEGIN
```

soit mettre des insertions lignes bout à bout comme

```
#> max.pas
#> format.pas
#> copies.pas

#: var  nbMachines : integer ;
#:      nbTaches     : integer ;
#:      dureeTache   : array[1..10] of integer ;
#:      somDuree     : integer ;
#:      indTache     : integer ;
#:      dureeCour    : integer ;
#:      dureeMax     : integer ;
#:      dureeMin     : integer ;
#:      machineTacheNumero : array[1..10,1..10] of integer ;
#:      machineTacheDuree  : array[1..10,1..10] of integer ;
#:      nbTache      : array[1..10] of integer ;
#:      indMachine   : integer ;
#:      machineCour  : integer ;
#:      tempsPris    : integer ;
#:      resteTemps   : integer ;
#:      tacheCour    : integer ;
#:      nbTacheCour  : integer ;

#: BEGIN
```

Pour l'algorithme maxoccmono, on devra inclure la fonction `format` déjà vue et la fonction `entierAuHasard` dont le contenu peut être :

```
(* fichier entierAuHasard.pas *)

function entierAuHasard( vmin, vmax : integer ) : integer ;
begin
  entierAuHasard := trunc( vmin + random( vmax-vmin+1 ) ) ;
end ;
```

Signalons que pour avoir un comportement aléatoire il faut rajouter

```
#: Randomize ;
```

avant la première affectation. Si on ne met pas cette instruction, les nombres sont bien tirés au hasard, mais toujours dans le même ordre.

On pourra utiliser comme fonction `copies` la fonction définie par le texte suivant :

```
(* fichier copies.pas *)

function copies( motif : string ; repet : integer ) : string ;
 var ind : integer ; chaine : string ;
begin
  chaine := '' ;
  for ind := 1 to repet do begin
      chaine := chaine + motif ;
  end ;
  copies := chaine ;
end ;
```

Il n'y a pas plus de difficulté à traduire l'algorithme rchbrute si ce n'est qu'il faut en plus inclure les fonctions `codeAscii`, `concatene` et `sousChaine` dont le contenu peut être :

```
(* fichier codeAscii.pas *)

function codeAscii( chaine : string ) : integer ;
begin
  codeAscii := ord( chaine[ 1 ] ) ;
end ;
```

```
(* fichier concatene.pas *)

function concatene( chaineA, chaineB : string ) : string ;
begin
 concatene := chaineA + chaineB ;
end ;

(* fichier sousChaine.pas *)

function sousChaine(chaine:string ; debut,longueur:integer) : string ;
  var ind : integer ; sChaine : string ;
begin
  sChaine := '' ;
  for ind := 1 to longueur do begin
      sChaine := sChaine + chaine[ debut + ind - 1 ] ;
  end ;
  sousChaine := sChaine ;
end ;
```

Pour l'avant-dernier algorithme, nommé`tabmult`, on modifie un peu l'algorithme de départ : on introduit une chaine `chNbChoisi` et si cette chaine est convertible en un entier, on l'utilise pour initialiser `nbChoisi`. On vient donc dans l'algorithme remplacer les lignes

```
     ...
     lire nbChoisi
# relance éventuelle
     tant_que (non entier(nbChoisi))
          écrire "nombre incorrect. Redonner un nombre ENTIER "
          lire nbChoisi
     fin_tant_que # nombre invalide
     ...
```

par les lignes

```
     ...
     lire chNbChoisi
 # relance éventuelle
     tant_que (non entier(nbChoisi))
        écrire "nombre incorrect. Redonner un nombre ENTIER "
        lire chNbChoisi
     fin_tant_que # nombre invalide
     affecter nbChoisi <-- valeurEntiere( chNbChoisi)
     ...
```

On a besoin d'inclure les fonctions `format`, `entier` et `valeur`. Comme `format` et `valeur` ont déjà été présentées, nous n'incluons ici que le détail de la fonction `entier` :

```
(* fichier entier.pas *)

function entier( chaine : string ) : boolean ;

 var nbMauvaisCar : integer ;
     ind          : integer ;

begin
 nbMauvaisCar := 0 ;
 for ind := 1 to length( chaine ) do begin
     if (chaine[ ind ] < '0') or (chaine[ ind ] > '9')
        then nbMauvaisCar := nbMauvaisCar + 1 ;
 end ;
 entier := nbMauvaisCar = 0 ;
end ;
```

Pascal admet suffisamment d'instructions proches de notre langage algorithmique pour la traduction du dernier algorithme nommé triFus soit simple : la fin de fichier, notée `fin_de_fichier` en algorithmique doit être écrite avec un caractère de soulignement et la table de correspondance interne la transforme en eof, l'ouverture en lecture est facilement gérée par les instructions $open(...)$ et $reset(...)$ alors que l'ouverture en écriture est gérée par $open(...)$ et $rewrite(...)$. La seule fonction à inclure est donc la fonction `mot`.

On écrira donc

```
#> trifuspascal.var
```

en début d'algorithme juste avant `#: BEGIN` et en plus de la déclaration des variables, on mettra la fonction `mot` telle qu'on peut la trouver sur notre site *Web*.

Pascal autorise la compilation séparée. Il est donc possible de regrouper tous les sous-programmes présentés en une seule unité, disons `algofuncs.pas` et de ne mettre alors que la ligne

```
#: Uses algofuncs ;
```

pour inclure tous les sous-programmes. Cela ne dispense malheureusement pas de déclarer les variables...

4.8 Traduction en C

La traduction en C impose les mêmes contraintes qu'en Pascal puisqu'il faut déclarer des variables. On notera qu'à la suite de la première série de commentaires, galg insère automatiquement les lignes :

```
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>
#include <ctype.h>
#include <time.h>
typedef char chaine[250] ;
```

Comme pour le Pascal, il faut choisir où mettre le début du programme principal et où on veut inclure les sous-programmes. Par exemple pour l'algorithme bonjour, le texte du fichier peut être :

```
######################################################################
#                                                                    #
#                                                                    #
#  auteur : (gH)                                                     #
#  bonjour.alg : un algorithme de bonjour conséquent.                #
#                                                                    #
#                                                                    #
######################################################################
#                                                                    #
#                                                                    #
#      cet algorithme comporte un dialogue où on demande à           #
#      l'utilisateur de donner son nom ;                             #
#      il affiche ensuite la date et l'heure puis dit au             #
#      revoir à la personne après avoir le nom en majuscules.        #
#                                                                    #
#                                                                    #
#                                                                    #
######################################################################

#> maju.c
#> date.c
#> heure.c

#: int main() {

#: chaine id ;
```

```
# la demande = une question + une réponse

écrire   " Bonjour. Quel est ton nom ? "
lire     id


# ce qu'on fait répondre par la machine

écrire   " Le ", date(), " à ", heure(), " au revoir " , maju(id)
```

à condition de disposer des fichiers suivants :

```
/* fichier date.c */

     char* date(void) {
          time_t now   = time(NULL) ;
          struct tm *t = localtime(&now);
          char laDate[100];
          strftime(laDate, 100, "%d/%m/%Y", t);
          return(laDate) ;
     } ; /* fin de date */

/* fichier heure.c */

     char* heure(void) {
          time_t now   = time(NULL);
          struct tm *t = localtime(&now);
          char lHeure[100];
          strftime (lHeure, 100, "%H:%M.%S", t);
          return(lHeure) ;
     } ; /* fin de heure */

/* fichier maju.c */

     char*  maju(chaine f_iden) {
      int idc = 0 ;
      while (idc<strlen(f_iden)) {
         f_iden[idc] = toupper(f_iden[idc]) ;
         idc++ ;
      } ; /* fin de pour # idc */
      return( f_iden ) ;
     } ; /* fin de maju */
```

Pour traduire l'algorithme cfe les choses se compliquent : le tableau des paramètres est fourni par la ligne `... main(...` qu'il faut inclure. Pour que les paramètres soient accessibles, cette ligne doit être impérativement

```
#: int main(int argc, char *argv[]) {
```

La fonction `parametre` doit alors être légèrement modifiée (il faut lui passer le nom du tableau des paramètres). galg s'en occupe si on met le caractère de soulignement devant le mot `parametre`. Il faut bien sur déclarer les variables `mntF` et `mntE` avec des inclusions simples (voir l'algorithme un peu plus loin).

Pour lire le nombre réel `mntF`, il y a une petite difficulté : le C est typé et comme galg ne gère pas (volontairement) les types, il faut passer par un sous-programme qui lit une chaine avec `fgets`, la convertit en réel et renvoie le nombre obtenu. On pourrait penser que `lireReel( mntF )` est simple, mais au contraire : en C les passages de paramètre se font **tous** par valeur. Il faudrait mettre l'adresse de la variable, soit `&mntF` (ce qui pourrait se faire avec la table de correspondance) pour que la variable soit effectivement affectée. Nous avons préféré utiliser une solution plus "propre", à savoir l'appel d'une fonction `lireReel` sans paramètre dont le résultat est un réel : il faut donc remplacer la ligne `lire mntF` par `affecter mntF <-- lireReel()` et bien sur inclure le sous-programme `lireReel.c`

Par contre, aucun problème pour traduire `nbParametres` et `valeurReelle` : il suffit de leur rajouter un caractère de soulignement pour que galg utilise sa table de correspondance interne (affichable par `galg -f c`) et les remplace donc respectivement par `argc-1` (car le C commence à compter à partir de zéro) et `atof`.

Restent maintenant quelques petits soucis pour l'écriture des résultats. Le C ne supporte pas les affichages par concaténation de termes à afficher. Il faut donc avoir recours à des appels successifs de `ecrire_` sur les premiers termes à afficher puis utiliser `ecrire` pour le dernier terme. Pire : comme C est typé, chaque appel de écrire doit indiquer s'il faut écrire un entier ou un réel avec le `printf` correspondant. C'est pourquoi nous avons créé les fonctions `ecrireReel` et `ecrireEntier` qu'il faut bien sur inclure en début de programme.

Finalement, l'algorithme modidié pour le C est

```
################################################
#                                              #
#   cfe.alg  ;  conversion francs en euro      #
#            -- auteur : gh                    #
#                                              #
################################################

#> lireReel.c
#> parametre.c
#> ecrireEntier.c
#> ecrireReel.c

#: int main(int argc, char *argv[]) {

#:   float mntF ;
#:   float mntE ;

# 1. Saisie ou conversion du paramètre

si _nbParametres()=0
   alors écrire_  " Quel est le montant en Francs ? "
         affecter mntF <-- lireReel()
   sinon affecter mntF <-- _valeurReelle( _parametre(1) )
fin_si # nbParametres()=0


# 2. Conversion et affichage

affecter mntE <-- mntF / 6.55957
appeler ecrireReel( mntF )
écrire_ "  Francs font "
appeler ecrireReel( mntE )
écrire "  Euros "

écrire_   " et si on arrondit : "
appeler   ecrireEntier(  _partieEntiere( 0.5+mntE ) )
écrire    " euros."
```

Voici le détails des sous-programmes utilisés :

```
/* fichier lireReel.c  */

     float lireReel() {
       float  f_lu ;
       chaine f_ch ;
       fgets(f_ch,130,stdin) ;
       f_lu = (float) atof( f_ch ) ;
       return( f_lu) ;
     } /* fin de lireReel */

/* fichier parametre.c */

     char*  parametre(char *t_argv[], int parm_num) {
      return( t_argv[ parm_num ] ) ;
     } /* fin de parametre */

/* fichier ecrireEntier.c  */

     void ecrireEntier(int i) {
       printf("%d",i) ;
       return ;
     } /* fin de ecrireEntier */

/* fichier ecrireReel.c */

     void ecrireReel(float f) {
       printf("%f",f) ;
       return ;
     } /* fin de ecrireReel */
```

Afin de "masquer" pour une fois les déclarations du C dans la traduction de l'algorithme macnaugthon, nous venons inclure en début d'algorithme, après les fonctions `maxEntier.c`, `ecrireEtier.c`, `formatEntier.c`, et `copies.c`, le fichier `macnaughtonC.inc` qui contient toutes les déclarations.

Si la traduction des affectations, des boucles `pour` ne demande aucune modification, comme précédemment, les sorties écran avec `ECRIRE` doivent être remaniées, puisque le C n'a pas de type *chaine* (string) élémentaire.

Il faut donc "bricoler", et remplacer par exemple une instruction comme

```
écrire "Durée minimale : " , dureeMin , " heures."
```

par les trois instructions

```
écrire_ "Durée minimale : "
appeler ecrireEntier( dureeMin )
écrire  "heures."
```

Ce n'est certes pas la meilleure solution. Une traduction propre, même minimaliste en C pourrait être notamment

```
printf("Durée minimale : %d heures." , dureeMin ) ;
```

mais il faudrait pour cela disposer du type des variables, ce qui est contre nos principes algorithmiques. Une solution envisageable (qui est à l'étude) serait d'ajoindre un fichier de typage suffisamment général pour qu'il puisse servir à la traduction de tous les langages typés explicitement à déclaration obligatoire et en particulier pour le C...

Les lignes suivantes, qui forment le fichier `macnaugthonC.inc` seraient alors automatiquent produites :

```
int main(void) {

    int nbMachines   = -1 ;
    int nbTaches     = -1 ;
    int indTache     = -1 ;
    int indMachine   = -1 ;
    int somDuree     =  0 ;
    int dureeCour    =  0 ;
    int dureeMax     =  0 ;
    int dureeMin     =  0 ;
    int tempsPris    =  0 ;
    int machineCour  =  0 ;
    int tacheCour    =  0 ;
    int resteTemps   =  0 ;
    int nbTacheCour  =  0 ;

    int dureeTache[10] ;
    int nbTache[10] ;
    int machineTacheNumero[10][10] ;
    int machineTacheDuree[10][10] ;
```

Comme l'algorithme macnaughton est long, nous n'en reproduisons que le début et la fin :

```
    ##################################################################
    #                                                                #
    #   macNaughton.alg : affectation de taches selon                #
    #                     la méthode de mac Naughton                 #
    #   auteur (gH)                                                  #
    #                                                                #
    #                                                                #
    ##################################################################
    #                                                                #
    #   références :  Cormen, Leiserson et Rivest                    #
    #                 Introduction to algorithms, MIT Press          #
    #                                                                #
    ##################################################################
    #                                                                #
    #                                                                #
    # avec nbMachines qui doivent effectuer nbTaches et sachant      #
    # que la tache numéro i dure dureeTache[ i] comment répartir     #
    # les taches ? la durée est en heure et les taches sont          #
    # morcelables heure par heure.                                   #
    #                                                                #
    #                                                                #
    ##################################################################

    #> maxEntier.c
    #> ecrireEntier.c
    #> formatEntier.c
    #> copies.c

    #> macnaughtonC.inc
  ...

    # 3. Affichage des données et des répartitions

    écrire_ " Il y avait "
    appeler ecrireEntier(nbMachines)
    écrire_ " machines et "
    appeler ecrireEntier(nbTaches)
    écrire " taches à répartir."

    écrire " Voici la durée des taches : "
    écrire "   Tache    Durée (en heures)"
```

```
pour indTache de1a nbTaches
     écrire_  formatEntier(indTache,8)
     écrire   formatEntier( dureeTache[ indTache ],8 )
fin_pour # indTache de1a nbTaches


écrire ""
écrire_ " La durée minimale calculée est "
appeler ecrireEntier(dureeMin)
écrire " heures."
écrire ""

pour indMachine de1a nbMachines
    écrire_ "  -- machine "
    appeler ecrireEntier(indMachine)
    écrire ""
    affecter nbTacheCour <-- nbTache[ indMachine ]
    si nbTacheCour > 0
       alors
             pour indTache de1à nbTacheCour
                  affecter dureeCour <-- machineTacheDuree[  indMachine , indTache ]
                  affecter tacheCour <-- machineTacheNumero[ indMachine , indTache ]
                  écrire_ copies("  ",6)
                  écrire_  " tache "
                  appeler  ecrireEntier( tacheCour )
                  écrire_  " durée "
                  appeler ecrireEntier( dureeCour )
                  écrire ""
             fin_pour # indTache de1à nbTacheCour
    fin_si # numMachine > 0
fin_pour # indTache de1a nbTaches
```

Nous listons aussi, bien sûr, les sous-programmes utilisés sauf `ecrireEntier` qui a déjà été vu :

```
/* fichier maxEntier.c */

     int maxEntier(float un, float deux) {
       if (un>deux) { return (int) un   ; }
               else { return (int) deux ; } ;
     } /* fin de maxEntier */
```

```
/* fichier formatEntier.c */

     char *formatEntier(int nombre,int longueur) {

       int      next      ;  int      flag = 0 ;
       char     qbuf[100] ;  char     qbuf2[100] ;
       register int r, k  ;

       next = 0 ;
       if (nombre <  0) { qbuf[next++] = '-' ; nombre = -nombre ; }
       if (nombre == 0) { qbuf[next++] = '0' ;   } else {
           k = 10000 ;
           while (k > 0) {
              r = nombre / k ;
              if (flag || r > 0) { qbuf[next++] = '0' + r ; flag = 1 ; } ;
              nombre -= r * k ;
              k       = k / 10 ;
             } ; /* fin de tant que k > 0 */
       } ; /* fin de si nombre non nul */
       qbuf[next] = 0 ;
       while (strlen(qbuf)<longueur) {
           strcpy(qbuf2," ")  ; strcat(qbuf2,qbuf) ; strcpy(qbuf,qbuf2) ;
       ; } ; /* fin de tant que chaine trop courte */

       return(qbuf);

     } /* fin de formatEntier */

/* fichier copies.c */

     char *copies(char *motif,int repet ) {

       int   k = 1 ;
       char  chaineRes[100] ;
       strcpy(chaineRes,"")  ;
       while (k <= repet) {
         strcat(chaineRes,motif) ;
         k++;
       ; } ; /* fin de tant que sur k */

       return(chaineRes) ;
     } /* fin de copies */
```

Par comparaison, l'algorithme maxoccmono est plus simple à traduire : à part l'inclusion de la ligne `#: srand( time(NULL) );` pour changer de valeurs aléatoires à chaque appel du programme, toutes les autres modifications ont déja été vues.

```
################################################
#                                              #
#                                              #
#   détermination du maximum dans un tableau   #
#   et comptage du nombre d'occurences de ce   #
#   maximum  en une seule boucle               #
#                                              #
#                                              #
################################################
#                                              #
# auteur : gh                                  #
#                                              #
################################################

#> ecrireEntier.c
#> entierAuHasard.c
#> formatEntier.c

#: int main(int argc, char *argv[]) {

#: int nbElt      = 0 ;
#: int indb       = 0 ;
#: int valMax     = 0 ;
#: int nbMax      = 0 ;
#: int eltCourant = 0 ;
#: int monT[50] ;

#: srand( time(NULL) ) ;

# initialisation du tableau avec 15 éléments
# entiers entre 10 et 20

affecter nbElt <-- 15
pour indb de1a nbElt
     affecter monT[indb] <-- entierAuHasard( 10, 20 )
fin_pour # indb de1a nbElt-1


# initialisation de la valeur du maximum (valMax)
```

```
# et de son nombre d'occurences (nbMax)

affecter valMax  <-- monT[ nbElt ]
affecter nbMax   <-- 1


# on parcourt alors le tableau
# sans utiliser le dernier élément déja comptabilisé

pour indb de1a nbElt-1
     affecter eltCourant <-- monT[ indb ]
     si eltCourant > valMax
        alors # nouveau maximum local
              affecter valMax  <-- eltCourant
              affecter nbMax   <-- 1
        sinon
              si eltCourant = valMax
                 alors # une fois de plus le maximum
                       affecter nbMax <-- nbMax + 1
              fin_si # nouvelle occurence du maximum
     fin_si # nouveau maximum
fin_pour # indb de1a 10


# affichage de fin

écrire_ " Le maximum dans le tableau est : "
appeler ecrireEntier( valMax )
écrire ""

écrire_ " et il apparait "
appeler ecrireEntier( nbMax )
écrire  " fois."

écrire " Pour vérification, voici les éléments du tableau : "

pour indb de1a nbElt
    écrire_   "    élément  "
    écrire_   formatEntier(indb , 3)
    écrire_   "  : "
    écrire   formatEntier(monT[indb],4)
fin_pour # indb de1a nbElt-1
```

Il ne reste donc que le fichier `entierAuHasard.c` à lister :

```
int entierAuHasard(int vmin, int vmax) {
  double res = 0  ;
  res = (double) rand() / ((double)RAND_MAX + 1) * (vmax-vmin+1) ;
  return( (int) vmin + res ) ;
} /* fin de entierAuHasard) */
```

Pour l'algorithme `rchbrute` nous avons encore des petites surprises : C distingue caractère et chaine de 1 caractère. Il faut donc adjoindre à la fonction `sousChaine` une fonction `caractereNumero` de façon à pouvoir transférer un caractère dans un tableau. Par contre `codeAscii` est alors inutile : on laisse la fonction, précédée d'un caractère de soulignement et galg vient supprimer l'appel de la fonction grâce à sa table de correspondance interne.

Comme il n'est pas autorisé en C d'affecter des chaines de caractère avec le symbole d'affectation usuel, il faut remplacer les affectations algorithmiques comme

```
chn <-- "mon texte..."
```

par l'instruction

```
appeler _affecterChaine(chn,"mon texte...")
```

et galg se charge alors de traduire avec `strcpy`.

Pas de problème en revanche pour la fonction `longueur` : précédée d'un caractère de soulignement, galg la transforme en `strlen` (les en-têtes de bibliothèques comme `#include <strings.h>` sont systématiquement mises par galg et le script conseillé pour compiler utilise `gcc ... -lm` pour assurer le "linkage").

La boucle `REPETER` est remplacée, pour d'autres langages en une boucle `TANT_QUE` et la fin de l'algorithme consiste en une multitude d'écritures partielles puisqu'il n'est pas possible de concatener des expressions en Cavec un seul `printf` sans connaitre le type explicite des données.

L'algorithme modifié pour être traduit en C est donc :

```
#############################################################
#                                                           #
#                                                           #
#  rchbrut.alg : brutesearch, auteur (gH)                   #
#                                                           #
#                                                           #
#############################################################
#                                                           #
# source : Algorithms, second edition                       #
#          pages 279, 280                                   #
#          R. Sedgewick, Addison-Wesley 1988                #
#                                                           #
#############################################################
#                                                           #
#                                                           #
#  recherche "brute" du tableau P dans le tableau A         #
#  (en anglais : P pour pattern, A pour array)              #
#  les variables sont renommées tabP et tabA                #
#  pour plus de lisibilité.   le tableau tabP contient      #
#  les codes ascii de la phrase                             #
#                                                           #
#      "A STRING SEARCHING EXAMPLE CONSISTING OF..."        #
#                                                           #
#  et le tableau tabA ceux du mot   "STING"                 #
#                                                           #
#                                                           #
#############################################################


#> caractereNumero.c
#> formatEntier.c
#> sousChaine.c
#> concatene.c
#> copies.c
#> ecrireEntier.c

#> rchbruteC.inc

appeler _affecterChaine(chP,"STING")
appeler _affecterChaine(chA,"A STRING SEARCHING EXAMPLE CONSISTING OF...")
```

```
# conversion des chaines en tableaux

affecter M <-- _longueur(chP)
pour ind de1a M
  affecter tabP[ ind ] <--  _codeAscii( caractereNumero(chP,ind) )
fin_pour # ind de1a longueur(chP)

affecter N <-- _longueur(chA)
pour ind de1a N
  affecter tabA[ ind ] <--  _codeAscii( caractereNumero(chA,ind) )
fin_pour # ind de1a longueur(chA)

# algorithme de recherche

affecter i <-- 1
affecter j <-- 1

#répéter
tant_que (j<=M) et (i<=N)

  si tabA[i] = tabP[j]
     alors affecter i <-- i + 1
           affecter j <-- j + 1
     sinon affecter i <-- i - j + 2
           affecter j <-- 1
  fin_si # tabA[i] = tabP[i]

#jusqu'à (j>M) ou (i>N)
fin_tant_que # (j<=M) et (i<=N)

# affichage du résultat

si j>M
   alors écrire_ chP
         écrire_ " vue dans "
         écrire_ chA
         écrire_ " à partir de la position "
         appeler ecrireEntier(i-M)
         écrire ""
   sinon écrire_ chP
         écrire_ chP
         écrire_ " non vue dans "
         écrire  chA
fin_si # j>M
```

```
# détail des tableaux (on suppose M < N)

écrire " Elément tableau P  tableau A "

appeler _affecterChaine(espb," ")

pour ind de1a  N

   si (ind<=M)

      alors appeler _affecterChaine(feltp,formatEntier(tabP[ind],8))
            appeler _affecterChaine(seltp,sousChaine(chP,ind,1) )
            appeler _affecterChaine(fe,concatene(feltp,espb))
            appeler _affecterChaine(milieu,concatene(fe,seltp))

      sinon appeler _affecterChaine(milieu,copies(" ",10))

   fin_si # (ind<=M)

   écrire_  formatEntier(ind,3)
   écrire_  " : "
   écrire_ milieu
   écrire_ formatEntier(tabA[ind],8)
   écrire_ " "
   écrire  sousChaine(chA,ind,1)

fin_pour # ind de1a N
```

Les sous-programmes utilisés sont :

```
/* fichier concatene */

      char* concatene(chaine un, chaine deux) {
       chaine res ;
       strcpy(res,un) ;
       strcat(res,deux) ;

       return( res ) ;

     } /* fin de concatene */
```

```
/* fichier caractereNumero.c */

     int caractereNumero(chaine ancienne, int debut) {
       return(ancienne[debut-1]) ;
     } /* fin de caractereNumero */

/* fichier sousChaine.c */

     char* sousChaine(chaine ancienne, int debut, int longueur) {
     chaine nouvelle ;
     int ic ;
       strcpy(nouvelle,ancienne) ;
       if (debut<0) { debut = 0 ; } ;
       if (longueur>strlen(ancienne)) { longueur = strlen(ancienne) ; } ;
       nouvelle[longueur] = 0 ;
       for (ic=0;ic<strlen(nouvelle);ic++) {
         nouvelle[ic] = ancienne[debut+ic-1] ;
       } ; /* fin de pour ic */
       return(nouvelle) ;
     } /* fin de sousChaine */
```

et le fichier inclus en début d'algorithme contient le texte :

```
int main(int argc, char *argv[]) {

chaine chP ;
chaine chA ;
int tabP[250] ;
int tabA[250] ;
int ind ;
int i ;
int j ;
int M ;
int N ;
chaine milieu ;
chaine feltp ;
chaine seltp ;
chaine espb ;
chaine fe ;
```

Passons maintenant à la traduction en C avec l'algorithme tabmult. A cause du typage des variables, on s'inspire de la version de l'algorithme modifié pour Pascal : on lit une chaine nommée `chNbChoisi` qu'on convertit éventuellement en `nbChoisi`. On rajoute donc la fonctions `entier`, soit l'algorithme :

```
#########################################################
#                                                       #
# tabmult.alg -- un exemple simple d'algorithme :       #
#                la table de multiplication             #
#                                                       #
#  auteur : gh                                          #
#                                                       #
#########################################################
#                                                       #
#  on demande à l'utilisateur un nombre entier et       #
#  on le relance tant que le nombre n'est pas           #
#  entier.                                              #
#                                                       #
#  lorsque le nombre est entier, on afiche sa           #
#  table avec cadrage des unités sous les unités,       #
#  des dizaines sous les dizaines etc.                  #
#                                                       #
#########################################################


#> ecrireEntier.c
#> formatEntier.c
#> affecterChaine.c
#> entier.c

#: int main(int argc, char *argv[]) {

#: int nbChoisi = 0 ;
#: int produit  = 0 ;
#: int indb     = 0 ;
#: chaine find ;
#: chaine fpro ;
#: chaine chNbChoisi ;

# demande initiale

écrire_  " Donner un entier < 3276 : "
lire    chNbChoisi
```

```
# relance éventuelle

tant_que (non entier(chNbChoisi))
    écrire ""
    écrire_ " nombre incorrect. Redonner un nombre ENTIER < 3276 : "
    lire    chNbChoisi
fin_tant_que # nombre invalide

affecter nbChoisi <-- _valeurEntiere( chNbChoisi )

# boucle d'affichage

écrire_ " Table de "
appeler ecrireEntier( nbChoisi )
écrire ""


pour indb de1a 10
    affecter produit <-- nbChoisi*indb
    appeler affecterChaine(find, formatEntier(indb,2))
    appeler affecterChaine(fpro, formatEntier(produit,5))
    écrire_ find
    écrire_ " fois "
    appeler ecrireEntier( nbChoisi )
    écrire_ " = "
    écrire  fpro
fin_pour # indb de1a 10
```

On notera qu'avec un typage explicite en `int`, il faut insister auprès de l'utilisateur pour obtenir un nombre entier inférieur à 3267 car le plus grand produit correspond à 10 fois le nombre entré et la plus grande valeur pour un `int` est 32767.

Le fichier `entier.c` contient les lignes :

```
int entier(chaine chen) {
  int res = 0 ; /* résultat */
  int ic  = 0 ; int nb  = 0 ;
  /* on vérifie qu'il n'y a que des chiffres ou "return" */
  while (ic<strlen(chen)) {
    if ( (!isdigit(chen[ic])) && (chen[ic]!=10) ) { res++ ; } ;
    ic++ ;
  } ; /* fin tant que sur ic */
```

```
  /* puis on vérifie que le nombre n'est pas trop grand */
  if (res==0) {
     nb = atoi( chen ) ;
     if (nb>3276) { res++ ; } ;
  } ; /* fin de si res= */
  return( res==0 ) ;
} /* fin de entier */
```

Finalement, pour la traduction du dernier algorithme nommé triFus, il y a aussi de nombreuses modifications à apporter. Tout d'abord on remplace les affectations de chaines de caractères par des appels au sous-programme `_affecterChaine`, au lieu de venir lire simplement la chaine nommé `ligne` sur le fichier `fic`, on passe par un sous-programme nommé `bonnneLecture` et donc l'instruction

```
lire ligne sur fic
```

est remplacée par l'instruction

```
si bonneLecture(ligne,fic)=1
```

Il faut aussi inventer une fonction `mot` et une fonction `sousChaine`, sans compter les déclarations de variables...

Le mieux est donc sans doute ici d'inclure tout un fichier, disons `triFusC.inc` en début d'algorithme par

```
#> triFusC.inc
```

pour y mettre toutes les déclarations et les sous-programmes. Voici un contenu possible de ce fichier (seul `formatEntier` n'y est pas inclus, car il a déjà été vu précédemment) :

```
chaine nomFic1,
       nomFic2,
       nomFic3,
       ligne1,
       ligne2,
       mot1,
       mot2
       ;

int nbl1,
    nbl2,
    nbl3,
    rc
    ;
```

```
FILE *fic1,
     *fic2,
     *fic3   ;

/* ################################################## */

              char* sousChaine(chaine ancienne, int debut, int longueur) {

/* ################################################## */

chaine nouvelle ;
int ic ;
  strcpy(nouvelle,ancienne) ;
  if (debut<0) { debut = 0 ; } ;
  if (longueur>strlen(ancienne)) { longueur = strlen(ancienne) ; } ;
  nouvelle[longueur] = 0 ;
  for (ic=0;ic<strlen(nouvelle);ic++) {
    nouvelle[ic] = ancienne[debut+ic-1] ;
  } ;
  return(nouvelle) ;
} /* fin de sousChaine */

/* ################################################## */

              char* mot(chaine ancienne, int numero) {

/* ################################################## */

chaine PHR ;
chaine lemot ;
chaine sep   ;
int    nb,i,L,debut_mot,fin_mot ;
  strcpy(lemot,ancienne) ;
  strcpy(sep," ") ;
  strcpy(PHR,ancienne) ;
  strcat(PHR,sep) ;
  L    = strlen(PHR) ;
  i    = 0 ;
  nb   = 0 ;
  while ( (i <= L) && (nb < (numero-1)) ) {
     if (PHR[i]==sep[0]) {
        nb = nb + 1 ;
        while ((i <= L ) && (PHR[i]==sep[0])) { i = i + 1 ; } ;
     } ; /*  finsi PHR[i]=sep */
```

```
      i = i + 1 ;
    } ; /* fin while (i < L) and ... */
    /* le début est donc trouvé ; cherchons la fin */
    debut_mot = i ;
    while (i<=L ) {
       if (PHR[i] != sep[0]) {  i = i + 1 ; }
                         else { fin_mot  = i ; i = L + 1 ; }
    } ; /* fin tant que i< L */
    if (fin_mot > debut_mot) {
       strcpy(lemot,sousChaine(PHR,debut_mot,fin_mot-debut_mot+1)) ;
    } else { strcpy(lemot,"") ; } ;
    return(lemot) ;

  } /* fin de mot */

  /* ################################################## */

                int bonneLecture(chaine laligne, FILE *lefichier) {

  /* ################################################## */

    if (fgets(laligne,250,lefichier) != 0) {
       laligne[ strlen(laligne)-1] = 0 ;
       return( 1 ) ;
    } else { return( 0 ) ; } ;
  } /* fin de bonneLecture */
```

Remarques sur la traduction en C :

Comme on a pu le lire, la traduction en C est la plus difficile à effectuer. De plus elle n'est pas toujours "propre" : par exemple la traduction de

```
ECRIRE  " nous sommes le " , date()
```

ne résiste à une compilation en *gcc* si on utilise les options

```
-ansi -pedantic -Wall -Werror
```

car une *function* (un sous-programme) ne doit pas renvoyer l'adresse d'une variable locale, en l'occurence la chaine qui contient la date formatée. Pour que l'appel soit correct en C-ANSI, il faudrait écrire `date(chaineDate)` au lieu de `date()` où `chaineDate` est une chaine de caractères.

Il serait possible de s'arranger pour que galg effectue automatiquement ce genre de modifications mais il faudrait alors rajouter de nombreuses déclarations de variables en début de programme, ce qui alourdirait le code C. Ainsi le programme suivant est valide avec les options de compilation précitées :

```
typedef char chaine[250] ; chaine la_Date ;

char* date(chaine l_d) {
     time_t now   = time(NULL);
     struct tm *t = localtime(&now);
     strftime(l_d, 100, "%d/%m/%Y", t);
     return(l_d) ;
} /* fin de date */

int main(void) {
   printf(" nous sommes le %s \n", date(la_Date) ) ;
return(0);
} /* fin de main */
```

De même, de nombreuses fonctions algorithmiques renvoient des chaines de caractères, ce que le C n'accepte pas. Au lieu de l'instruction algorithmique

```
affecter maChaine <--  fonction( p1, p2 ... )
```

que nous avons souvent remplacée par

```
appeler affecterChaine( maChaine, fonction( p1, p2 ... ) )
```

il aurait fallu en fait substituer l'instruction

```
appeler fonction( maChaine, p1, p2 ... )
```

ce qui bien sur serait plus correct sous l'angle du C mais moins lisible selon nos critères et surtout cela aurait changé la syntaxe de nos "bonnes" fonctions algorithmiques de base.

Enfin, nous n'avons pas inclus, là encore pour des raisons de longueur de code, les prototypes des fonctions utilisées (par exemple dans `algofns.h`). Nous demandons donc aux puristes C de nous excuser pour ces abus d'écriture et nous rappelons que la seule option de compilation obligatoire avec *gcc* pour les algorithmes est `-lm` lorsqu'on utilise des fonctions mathématiques.

Par contre, on notera que galg traduit très proprement la lecture au clavier d'une chaine via `fgets` plutot que par `gets` ce qui évite des débordement dans les saisies clavier.

Signalons que les mêmes programmes traités par le le compilateur pour C++ avec les options "dures" de compilation détecte encore des erreurs, dues à des comparaisons entre nombres entiers signés et nonsignés (ce qui ne devrait pas se faire, en toute rigueur). Voici les lignes incriminées :

```
(idc<strlen(f_iden)) { ... }
for (ic=0;ic<strlen(nouvelle);ic++) { ... }
if (longueur>strlen(ancienne)) { longueur = strlen(ancienne) ; } ;
return( (int) vmin + res ) ;
while (ic<strlen(chen)) { ... }
while (strlen(qbuf)<longueur) { ... }
```

Le compilateur C++ signale aussi une erreur de conversion d'entier double en simple pour la ligne `return( (int) vmin + res ) ;` utilisée avec l'algorithme maxoccmono lors du calcul d'un nombre entier au hasard.

Comme vous pouvez vous en rendre compte, les avertissements (*warnings*) ne sont pas justifiés, mais enfin...

4.9 Traduction en C++

La traduction en C++ est plus simple qu'en C pour deux raisons :

- il est possible de concaténer des expressions en sortie via `cout ...>>`
- il existe un type *string* standard
- les opérateurs `<<` et `>>` peuvent s'appliquer aux fichiers.

Ainsi pour l'algorithme bonjour, puisque galg rajoute de lui même les lignes

```
#include <iostream>
#include <string>
#include <fstream>
#include <time>
```

après la première série de commentaire, il n'y a qu'à inclure les lignes d'inclusion pour les fichiers `date.c`, `heure.c` et `maju.c` , la ligne de debut de programme "`int main(void) {`" et la ligne de déclaration "`string id ;`" pour que la traduction soit assurée.

Notre nouveau programme de passage en majuscules est :

```
// fichier maju.cpp

       template <class charT, class traits, class Allocator>
       inline
       basic_string<charT, traits, Allocator>

       maju(const basic_string<charT,traits, Allocator>& str) {
               basic_string<charT, traits, Allocator> newstr(str);
               for (size_t index = 0; index < str.length(); index++) {
                       if (islower(str[index])) {
                           newstr[index] = toupper(str[index]);
                       } ; // fin de si minuscule
               } ; // fin de pour index
               return newstr;
       } // fin de maju
```

Un familier du C++ y reconnaitra la STL (*Standard Template Library*) c'est à dire la marque du standard C++.

Pour traduire le dernier algorithme, triFus on vient, comme en C remplacer les affectations chaines de caractères par des appels au sous-programme `_affecterChaine`. Par contre, contrairemet au C, on peut garder les lignes comme

```
lire ligne sur fic
```

car elles s'accordent bien à la syntaxe "propre" du C++, à savoir

```
fic.getline(ligne,250)
```

La comparaison de chaines peut se faire avec la fonction *strcmp* qui renvoie un nombre négatif si son premier argument est lexicographiquement avant son deuxième argument. On peut soit laisser la fonction telle quelle avec ce nom dans l'algorithme ou l'écrire `_chaineAvant\verb` car elle est dans la table de correspondance interne de galg. On peut lui préferrer la fonction *strncmp* qui admet trois paramètres, le troisième étant le nombre de caractères sur lequel porte la comparaison.

4.10 Traduction en Java

Un programme Java comporte en général des définitions d'importation (instructions `import`) avant la définition du programme principal avec l'instruction `class`. galg vient ne vient incorporer l'instruction `class` du programme principal qu'après la première série de commentaire et donc après tous les les commentaires spéciaux liés à la première ligne.

Par contre, il peut y avoir d'autre déclarations de sous-programmes avant l'instruction `main` et c'est donc à l'auteur de l'algorithme de choisir où inclure cette instruction `main` dans l'algorithme.

Le texte de l'algorithme bonjour ressemble à :

```
#####################################################################
#                                                                   #
#                                                                   #
#  auteur : (gH)                                                    #
#  bonjour.alg : un algorithme de bonjour conséquent.               #
#                                                                   #
#                                                                   #
#####################################################################
#                                                                   #
#                                                                   #
#                                                                   #
#      cet algorithme comporte un dialogue où on demande à          #
#      l'utilisateur de donner son nom ;                            #
#      il affiche ensuite la date et l'heure puis dit au            #
#      revoir à la personne après avoir le nom en majuscules.       #
#                                                                   #
#                                                                   #
#                                                                   #
#####################################################################
#
#:     import java.io.* ;
#:     import java.text.DateFormat ;
#:     import java.text.SimpleDateFormat ;
#:     import java.util.Date ;
#:     import java.util.Locale ;

#> date.java
#> heure.java
#> maju.java
```

```
     #: public static void main(String args[]) {
     #: // début de la méthode main()


     #:   String id = "gh" ;

        # la demande = une question + une réponse

        écrire   " Bonjour. Quel est ton nom ? "
        lire     id


        # ce qu'on fait répondre par la machine

        écrire   " Le ", date(), " à ", heure(), " au revoir " , maju(id)
```

Comme indiqué, si les commentaires spéciaux ne font pas partie de la première série de commentaires, par exemple si on laisse une ligne vide après la première série de commentaires avant les commentaires spéciaux d'ajout, soit le texte

```
     ######################################################################
     #                                                                    #
     #  auteur : (gH)                                                     #
     #  bonjour.alg : un algorithme de bonjour conséquent.                #
     #                                                                    #
     ######################################################################

     # bjr.alg, un bonjour minimal ; auteur : (gH)
     #:     import java.io.* ;
...
```

alors la traduction est incorrecte puisqu'elle aboutit aux instructions

```
...
     class bonjour {
...
     import java.io.* ;
```

qui est l'ordre inverse de l'ordre normal pour java.

On notera que l'instruction `lire` est équivalent à la lecture d'une chaine de caractères. Java étant très soigneux sur les entrées et sorties, l'instruction `lire` de l'algorithmique ne pourra lire aucun entier ni nombre réel.

Pour traduire la lecture de nombres, nous devons donc modifier les algorithmes. C'est notamment pour le cas de l'algorithme `cfe` qui vient lire une somme réelle au clavier lorsqu'il n'y a pas de paramètre. L'instruction `lire variable` doit être remplacée selon les cas par

```
affecter variable <-- lireEntier()
```

ou par

```
affecter variable <-- lireReel()
```

Comme `Java` est typé explicitement, il faut aussi déclarer les variables et inclure les sous-programmes à utiliser. Ceux-ci sont au nombre de 4, à savoir : `parametre`, `lireReel`, `valeurReelle` et `partieEntiere`. On notera que la fonction `nbParametres` est connue dans la table de correspondance interne (donc il faut lui adjoindre un caractère de soulignement). Il faut également mettre un caractère de soulignement devant le mot `parametre` (même si la fonction `parametre` est lue dans un fichier externe).

Voici donc l'algorithme :

```
################################################
#                                              #
#   cfe.alg  ;  conversion francs en euro      #
#            -- auteur : gh                    #
#                                              #
################################################
#
#: import java.io.* ;
#: import java.lang.* ;
#


#> parametre.java
#> lireReel.java
#> valeurReelle.java
#> partieEntiere.java

#: public static void main(String tab_args[]) {
#: // début de la méthode main()

#: double mntF = 10 ;
#: double mntE = 20.0 ;
```

```
# 1. Saisie ou conversion du paramètre

écrire " il y a " , _nbParametres() ," parametres."

si _nbParametres()=0
   alors écrire_  " Quel est le montant en Francs ? "
         affecter mntF  <-- lireReel()
   sinon affecter mntF  <-- valeurReelle( _parametre( 1 ) )
fin_si # _nbParametres()=0


# 2. Conversion et affichage

affecter mntE <-- mntF / 6.55957
écrire   mntF , " Francs font " , mntE , " euros."
écrire   " et si on arrondit : " , partieEntiere( 0.5 + mntE ), " euros."
```

Le texte des sous-programme à inclure peut être :

```
// fichier parametre.java


     static String parametre(String tab_args[], int num_parm) {

       return tab_args[ num_parm - 1] ;
     } // fin de parametre

// fichier lireReel.java

     static double lireReel() {

       double nb_reel = -999.99 ;
       String chen = "" ;

       System.out.flush() ;
       try { BufferedReader clavier =
                new BufferedReader(new InputStreamReader(System.in)) ;
             chen = clavier.readLine() ;
       } // fin de try
       catch (IOException erreurES) {
             System.out.println(erreurES.getMessage()) ;
             System.exit(-1) ;
       } // fin de catch
```

```
            try {  nb_reel = Double.valueOf(chen).doubleValue() ; }
            catch (NumberFormatException erreurConv) {
                   System.out.println(" erreur de conversion en réel ") ;
                   System.exit(-1) ;
            } // fin de catch

            return nb_reel ;

          } // fin lireReel

    // fichier valeurReelle.java

          static double valeurReelle(String chen) {

            double valeur_reelle = -999.99 ;

            try {  valeur_reelle = Double.valueOf(chen).doubleValue() ; }
            catch (NumberFormatException erreurConv) {
                   System.out.println(" Erreur de conversion en réel ") ;
                   System.exit(-1) ;
            } // fin de catch

            return valeur_reelle ;

          } // fin de valeurReelle

    // fichier partieEntiere.java

          static int partieEntiere(double nb_r) {

            return (int) nb_r ;

          } // fin de partieEntiere
```

Pour traduire l'algorithme macnaugthon, il faut déclarer des tableaux en Java puis les créer avec l'instruction `new`.

De plus, il faut définir un sous-programme du calcul du maximum de deux nompres entier, nommé ici `maxEntier`. Pour forcer la conversion en nombre entier d'un réel, on utilise la fonction `partieEntiere`. Les fonctions `copies` et `format` n'existant pas en Java, nous venons aussi les inclure en début d'algorithme.

Nous ne reproduisons ici que le début de l'algorithme car le reste est presque totalement identique à l'algorithme initial :

```
#################################################################
#                                                               #
#   macNaughton.alg : affectation de taches selon               #
#                     la méthode de mac Naughton                #
#   auteur (gH)                                                 #
#                                                               #
#################################################################
#
#:     import java.io.* ;


#> maxEntier.java
#> partieEntiere.java
#> format.java
#> copies.java

#: public static void main(String args[]) {
#: // début de la méthode main()


#:     int     nbMachines    = -1  ;
#:     int     nbTaches      = -1  ;
#:     int[]   dureeTache          ;
#:     int     somDuree      =  0  ;
#:     int     dureeCour     =  0  ;
#:     int     dureeMax      =  0  ;
#:     int     dureeMin      =  0  ;
#:     int[]   nbTache             ;
#:     int[][] machineTacheNumero  ;
#:     int[][] machineTacheDuree   ;
#:     int     tempsPris     =  0  ;
#:     int     machineCour   =  0  ;
#:     int     tacheCour     =  0  ;
#:     int     resteTemps    =  0  ;
#:     int     nbTacheCour   =  0  ;


# 0.  Affectations arbitraires pour tester un
#     exemple élémentaire
#
```

```
     affecter nbMachines  <-- 3
     affecter nbTaches    <-- 6

     #:     dureeTache         = new int[ nbTaches + 1 ] ;
     #:     nbTache            = new int[ nbTaches + 1 ] ;
     #:     machineTacheNumero = new int[ nbTaches + 1 ][ nbMachines + 1 ] ;
     #:     machineTacheDuree  = new int[ nbTaches + 1 ][ nbMachines + 1 ] ;

     affecter dureeTache[ 1 ]  <--   5
...
```

Par rapport à l'algorithme initial, la seule ligne à changer est la ligne

```
     dureeMin <-- max( somDuree / nbMachines , dureeMax)
```

qui doit devenir :

```
     dureeMin <-- maxEntier( partieEntiere( somDuree / nbMachines) , dureeMax)
```

de façon à ne pas avoir de problème de typage entre nombres entiers et nombres réels (car une division donne toujours un nombre réel, même si on peut croire naivement que 6/2 donne un nombre entier).

Voici le texte des sous-programmes utilisés :

```
     // fichier maxEntier.java

          public static int maxEntier( int un, int deux) {

            if (un>deux) { return un ; } else { return deux ; } ;

          } // fin de maxEntier


     // fichier format.java

          public static String format(int nombre, int longueur) {

            String chen = Integer.toString(nombre) ;
            while (chen.length() < longueur) { chen = " " + chen ; } ;
            return chen ;

          } // fin de format
```

```
// fichier copies.java

     public static String copies(String motif, int repet) {

       String chen = "" ;
       for (int ifois =1 ; ifois <= repet ; ifois++) {
           chen = motif + chen ;
       } ; // fin de pour

       return chen ;

     } // fin de copies
```

Pour l'algorithme maxoccmono, une fois qu'on a importé la fonction `entierAuHasard` et déclaré le tableau des valeurs, l'algorithme orginal est repris tel quel. Nous ne présentons donc là encore que le début de l'algorithme :

```
################################################
#                                              #
#                                              #
#   détermination du maximum dans un tableau   #
#   et comptage du nombre d'occurences de ce   #
#   maximum  en une seule boucle               #
#                                              #
#                                              #
################################################
#                                              #
# auteur : gh                                  #
#                                              #
################################################
#
#:  import java.io.* ;

#> entierAuHasard.java
#> format.java

#: public static void main(String args[]) {
#: // début de la méthode main()

#:    int nbElt      = 0 ;
#:    int valMax     = 0 ;
#:    int nbMax      = 0 ;
#:    int eltCourant = 0 ;
```

```
# initialisation du tableau avec 15 éléments
# entiers entre 10 et 20

affecter nbElt <-- 15

#: int[] monT ;
#: monT = new int[ nbElt + 1 ] ;
```

et son sous-programme associé :

```
// fichier entierAuHasard.java

     public static int entierAuHasard( int vmin, int vmax) {
       return   (int) (vmin + Math.random()*(vmax-vmin)) ;
     } // fin de entierAuHasard
```

Pour traduire l'algorithme rchbrute, il faut bien sur déclarer les variables et les tableaux. Comme on travaille avec des chaines de caractères, on a besoin des fonctions `longueur`, `codeAscii`, `sousChaine` et `concatene`.

Le texte des sous-programmes peut être :

```
// fichier longueur.java

     public static int longueur(String chen) {

       return chen.length() ;

     } // fin de longueur

// fichier codeAscii.java

     public static int codeAscii(String chen) {
       char uncar = chen.charAt(0) ;
       return Character.getNumericValue(uncar) ;
     } // fin de codeAscii

// fichier sousChaine.java

     public static String sousChaine(String chen, int debut, int longueur) {
       String sous_chaine = ""  ;
       try   { sous_chaine = chen.substring(debut-1,debut+longueur-1) ; }
```

```
          catch (StringIndexOutOfBoundsException erreurSC) {
             System.out.println(erreurSC.getMessage());
             System.exit(-1) ;
          } // fin de catch
          return sous_chaine ;
        } // fin de sousChaine

// fichier concatene.java

        public static String concatene(String un, String deux) {
          return un + deux ;
        } // fin de concatene
```

En ce qui concerne l'avant-dernier algorithme, tabmult, on s'inspire de la version de l'algorithme modifié pour Pascal : on lit une chaine nommée `chNbChoisi` qu'on convertit éventuellement en `nbChoisi`. On rajoute les fonctions `valeurEntiere`, `format` et `entier`. Nous ne donnons ici que ces deux dernières fonctions car `format` a déja été vue :

```
// fichier valeurEntiere.java

        static int valeurEntiere(String chen) {

          int valeur_entiere = 100 ;

          try {  valeur_entiere = Integer.valueOf(chen).intValue() ; }
          catch (NumberFormatException erreurConv) {
                 System.out.println(" Erreur de conversion en entier ") ;
                 System.exit(-1) ;
          } // fin de catch

          return valeur_entiere ;

        } // fin de valeurEntiere

// fichier entier.java

        public static boolean entier(String chen) {

          boolean resultat = false ;
          int nb_entier = -1 ;
          try {  nb_entier = Integer.valueOf(chen).intValue() ;
             resultat = true ;
          } // fin de try
```

```
            catch (NumberFormatException erreurConv) {
               resultat = false ;
            } // fin de catch
             return resultat ;
          } // fin de entier
```

Le dernier algorithme à traduire en Java est triFus. Comme Java est très pointilleux sur les opérations sur fichiers, il faut modifier l'algorithme initial. En particulier, la traduction de

```
ouvrir nomfic en_lecture comme fic
...
fermer fic
```

est traduit automatiquement en Java par galg comme les lignes

```
BufferedReader fic =  new BufferedReader(new FileReader(nomFic)) ;
try {
...
fic.close() ;
} // fin de try : surtout pas de ;
catch(IOException exc) {
         System.out.println("Erreur de lecture de fichier ");
         exc.printStackTrace() ;
} ; // fin de catch
```

L'instruction *try...catch* assure qu'il n'y aura pas d'erreur de lecture non détectée. Comme elle englobe toutes les instructions entre `ouvrir` et `fermer`, il faut fermer les fichiers dans l'ordre inverse de leur ouverture.

La même s'applique pour `ouvrir nomfic en_ecriture comme fic` qui déclenche le code

```
PrintWriter fic =
   new PrintWriter(new BufferedWriter(new FileWriter(nomFic))) ;
try {
```

Le reste de l'algorithme est aisé à traduire si on vient inclure, comme pour les autres langages ressemblant, les sous-programmes `longueur`, `sousChaine`, `format`, `chaineAvant` et `mot`.

Chapitre 5.

Mode "Aménagement d'algorithmes"

L'aménagement d'un algoritme consiste en 3 améliorations possibles :

- l'ajout éventuel du mot `AFFECTER` là où il faut,
- l'ajout éventuel du mot `APPELER` là où il faut,
- la normalisation des mots-clefs de fin de structures.

Pour savoir s'il faut ajouter le mot `AFFECTER`, le mode aménagement se contente de tester si la chaine `"<--"` est présente dans la ligne à aménager. 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, 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"`

L'aménagement des mots-clefs de fin de structures vient remplacer les écritures possibles comme `fin tant que` ou `fin tantque` en la seule écriture acceptée `fin_tant_que`.

L'intêret du mode aménagment est qu'il permet de conserver des algorithmes plus rapidement écrits et avec une syntaxe plus lisible tout en restant utilisable par galg.

Ainsi le fichier texte maxoccdem.txt dont le contenu suit devient, via la commande `galg -t maxoccdem.txt`, le fichier maxoccdem.alg qui est directement utilisable par la commande `galg -a` et ses différentes options comme `-o` et `-x`.

Voici le contenu du fichier original

```
##########################################################
#                                                        #
#  auteur : (gH)                                         #
#  maxocc.alg : calcul du plus grand élément d'un        #
#               tableau avec comptage du nombre          #
#               d'occurences en une seule boucle.        #
#                                                        #
##########################################################

# 1. initialisation du tableau avec 15 éléments
#    entiers entre 10 et 20
     nbElt <-- 15
     init_Tab( "monT" , nbElt , 10 , 20 )
# 2. détermination du max et comptage du nb d'occurences
#    du max en une seule boucle
#    2.1 initialisation de la valeur du maximum (valMax)
     valMax  <-- monT[ nbElt ]
     nbMax   <-- 1
#    2.2 on parcourt alors le tableau
#        sans utiliser le dernier élément déja comptabilisé
pour indb de1a nbElt-1
     eltCourant <-- monT[ indb ]
     si eltCourant > valMax
        alors # nouveau maximum local
              valMax  <-- eltCourant
              nbMax   <-- 1
        sinon
              si eltCourant = valMax
                 alors # une fois de plus le maximum
                       nbMax <-- nbMax + 1
              fin si # nouvelle occurence du maximum
     finsi # nouveau maximum
fin pour # indb de1a 10

# 3. affichage de fin

écrire " Le maximum dans le tableau est : " , valMax
écrire " et il apparait " , nbMax , " fois."
```

```
     écrire " Pour vérification, voici les éléments du tableau : "

     affiche_Tab( "monT" , 1 , nbElt , 4)
```

Et le fichier produit est :

```
#-#   Fichier maxoccdem.alg issu de galg -t maxoccdem.txt
#-#  ================================================
#-#      14/08/2001 19:58.40

#########################################################
#                                                       #
#  auteur : (gH)                                        #
#  maxocc.alg : calcul du plus grand élément d'un       #
#               tableau avec comptage du nombre         #
#               d'occurences en une seule boucle.       #
#                                                       #
#########################################################

#
# 1. initialisation du tableau avec 15 éléments
#    entiers entre 10 et 20
#

     affecter nbElt <-- 15
     appeler init_Tab( "monT" , nbElt , 10 , 20 )

#
# 2. détermination du max et comptage du nb d'occurences
#    du max en une seule boucle

#
#    2.1 initialisation de la valeur du maximum (valMax)
#        et de son nombre d'occurences (nbMax)
#

     affecter valMax  <-- monT[ nbElt ]
     affecter nbMax   <-- 1



#    2.2 on parcourt alors le tableau
#        sans utiliser le dernier élément déja comptabilisé
```

```
pour indb de1a nbElt-1
     affecter eltCourant <-- monT[ indb ]
     si eltCourant > valMax
        alors # nouveau maximum local
              affecter valMax  <-- eltCourant
              affecter nbMax   <-- 1
        sinon
              si eltCourant = valMax
                 alors # une fois de plus le maximum
                       affecter nbMax <-- nbMax + 1
              fin_si # nouvelle occurence du maximum
     fin_si # nouveau maximum
fin_pour # indb de1a 10

# 3. affichage de fin

écrire " Le maximum dans le tableau est : " , valMax
écrire " et il apparait " , nbMax , " fois."
écrire " Pour vérification, voici les éléments du tableau : "

appeler affiche_Tab( "monT" , 1 , nbElt , 4)
```

Chapitre 6.

Installation de galg et appel des langages

6.1 Installation de galg

Pour installer galg, il y a deux choses à faire :

1. recopier le programme galg.pl et les modules associés galg*.pm dans un même répertoire (par exemple le répertoire `D:\Galg` pour *Dos*, `~/Galg` pour *Unix*) ;
2. définir la variable d'environnement `PERL_MACROS`.

Pour que galg puisse appeler les langages, il faut évidemment que ceux-ci soient déjà installés. L'installation des langages ne fait pas partie de ce manuel.

Comme indiqué au chapitre 1, il y a 7 modules généraux et 8 modules spécifiques, (un par langage), il faut donc recopier les 16 fichiers :

```
galg.pl                 galg2c.pm
galgamng.pm             galg2cpp.pm
galganl.pm              galg2dbase.pm
galgerr.pm              alg2java.pm
galgfncs.pm             galg2pascal.pm
galglang.pm             galg2perl.pm
galgtrad.pm             galg2rexx.pm
galgvars.pm             galg2tcl.pm
```

Pour indiquer que la variable d'environnement `PERL_MACROS` correspond au répertoire choisi, on tapera :

- sous *Dos* :

```
set perl_macros=D:\Galg
```

- sous *Unix*, en *bash* :

```
export PERL_MACROS=~/Galg
```

Si on se place dans ce répertoire, il suffit alors de taper `perl galg.pl` suivi des paramètres voulus pour appeler galg.

Pour une utilisation plus intensive, il vaut mieux définir un script d'appel de galg accessible par le *PATH*. Sous *Dos*, on construira un fichier `batch`; sous *Unix* un script `bash` ou une commande exécutable suffira.

Par exemple sous *Dos*, on pourra écrire le fichier `galg.bat` dont le contenu est

```
perl D:\Galg\galg.pl %1 %2 %3 %4 %5 %6 %7 %8 %9
```

alors que sous *Unix*, on pourra écrire le fichier `galg` dont le contenu est

```
perl ~/Galg/galg.pl $*
```

La localisation exacte des fichiers `galg.bat` et `galg` pour qu'ils soient accessibles par le *PATH* dépend bien évidemment de la machine et du système utilisé. En principe sous *Dos*, le répertoire `C:\Windows` est toujours accessible par le *PATH* mais ce n'est pas forcément le meilleur endroit pour mettre la commande galg (nous conseillons plutot d'utiliser un répertoire `C:\Outils` ou `C:\BIN` pour toutes les commandes et scripts). Sous *Unix*, le répertoire `~/bin` ou `~/Bin` doit pouvoir convenir. En cas de doute, le mieux est de demander à quelqu'un de compétent de gérer ce *PATH*.

Pour modifier les commandes proposées pour appeler les langages, il faut consulter le manuel du programmeur afin de savoir quelles lignes changer dans le fichier galglang.pm qui gère les langages pour galg.

On trouvera sur notre site *Web*, à l'adresse

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

les différents scripts présentés ici nécessaires à l'appel des langages.

6.2 Appel du langage Rexx

La commande appelée par galg est `regina`. Il s'agit de l'exécutable compilé (supposé accessible par le $PATH$) et non pas d'un script. Il n'y a donc aucune précaution particulière à prendre hormis le fait que, lors de l'inclusion de fichiers, ceux-ci doivent bien sûr être dans le même répertoire que le fichier-algorithme.

Nous vous conseillons fortement d'installer Rexx, ne serait-ce que pour gérer les autres scripts d'appel des langages. L'implémentation *regina* est gratuite pour *Dos* et pour *Unix* et en particulier pour *Linux*. Rexx est disponible sur de nombreux sites *Web* et en particulier à l'adresse

http://www.lightlink.com/hessling/

De plus de nombreux algorithmes écrits en Rexx sont accessibles sur le *Web*. On pourra notamment consulter *the Album of Algorithms and Techniques for Standard Rexx* écrit par Vladimir Zabrodsky à l'adresse

http://www.geocities.com/zabrodskyvlada/aat/index.html

6.3 Appel du langage Perl

La commande appelée par galg est `perl -W` ce qui signifie que les *warnings* de précompilation sont activés. Cela peut aider lorsque la traduction a "oublié" un symbole $ devant le nom d'une variable, si les modules et fonctions n'ont pas été trouvés...

Puisque galg est écrit en Perl, Perldoit être installé sur toute machine qui veut utiliser galg. Perl est disponible gratuitement pour *Dos* et pour *Unix* (dont *Linux*) à l'adresse

http://www.cpan.org

Les algorithmes du livre *Mastering Algorithms with Perl* de Orwant, Hietaniemi et Macdonald parus chez O'Reilly sont disponibles par un clic sur le mot *Examples* à l'adresse *Web*

http://www.oreilly.com/catalog/maperl/

6.4 Appel du langage Tcl/Tk

Le langage Tcl/Tk est disponible gratuitement pour *Dos* et pour *Unix* (dont *Linux*) à l'adresse

http://www.activestate.com/ASPN/Downloads/ActiveTcl/

Pour exécuter un programme Tcl/Tk, galg utilise la commande `tcl`. Il s'agit de l'appel à l'interpréteur standard installé sous *Unix* lors de l'installation du langage.

Pour *Dos*, l'installation standard crée un répertoire *bin* sous le répertoire *tcl* qui doit contenir un fichier exécutable (de type `.exe` dont le nom commence par `tclsh`) qui doit être appelé par la commande `tcl`.

Sous *Dos*, on viendra donc créer un fichier "batch" nommé `tcl.bat` dont le contenu ressemble à

```
@echo off
C:\tools\tcl\bin\tclsh80.exe %1 %2 %3 %4 %5 %6 %7 %8 %9
```

6.5 Appel du langage Dbase

La version de *Dbase* choisie est `MAX`, disponible gratuitement pour *Dos* comme pour *Unix* à partir du site *Web* :

http://www.plugsys.com/

On notera que cette version compile en en respectant majuscule et minuscules mais que le module d'exécution `maxrun` ne sait utiliser que les fichiers en minuscules. Ainsi la commande

```
max monProgramme.prg
```

produit le fichier `monProgramme.max` mais la tentative d'exécution par la commande

```
maxrun monProgramme.max
```

produit l'erreur *File Not Found*. Il est donc fortement conseillé d'utiliser des noms de fichiers en majuscules (par exemple le fichier précédent devrait être nommé `monprogramme.prg`), mais galg ne l'impose ni le détecte.

La commande appelée par galg pour compiler et exécuter en Dbase est `db`. Il s'agit d'un script accessible par le *PATH*. Ce script est censé compiler le fichier avec

la commande `max` puis exécuter, si le compilation est correcte, par la commande `maxrun`. On notera qu'il faut rajouter le mot clé `-p` s'il y a des paramètres à transmettre. Nous vous recommandons d'utiliser comme script l'appel au programme Rexx suivant nommé `dbmax.rex` :

```
/* compilation et exécution dans la foulée d'un
   fichier dbase (.prg) avec max version 20 */

parse arg fn rst

/* si pas de paramètre, rappel de l'aide */

   if words(fn)=0 then do
      say " syntaxe  : ma nom_fichier "
      say " exemple  : ma bonjour "
      exit
   end

/* élimination de l'ancien résultat de compilation */

   "\rm -f " fn||".max"

/* compilation et récupération du code-retour */

"max " fn||".prg"
sovRc = rc

if sovRc=0 then do
   if words(rst)>0 then do /* il faut -p devant les paramètres */
       finparm = "-p " rst
   end ; else ; do
       finparm = ""
   end ;
   /* exécution éventuelle */
   "maxrun " fn||".max" finparm
end ; else ; do
    say " erreur de la compilation max " fn
end
```

Le script peut être, sous *Dos* :

```
@echo off
regina C:\Max20Win\dbmax.rex %1 %2 %3 %4 %5 %6 %7 %8 %9
```

et sous *Unix* :

```
regina ~/Bin/dbmax.rex $*
```

Mais si *Max* est installé dans d'autres répertoires, ou si d'autres versions du compilateur Dbase sont installés, on modifiera les chemins d'accès et les commandes en conséquence. Là encore, en cas de doute, le mieux est de demander à quelqu'un de compétent de gérer le script.

6.6 Appel du langage C

Il existe de nombreux compilateurs C sous *Dos* comme sous *Unix*. La commande utilisée pc par galg peut appeler le compilateur voulu. En particulier *gcc* est un bon choix gratuit pour *Dos* et pour *Unix* (dont *Linux*) à disponible l'adresse

http://gcc.gnu.org/

Notre configuration personnelle utilise gcc et la commande pc est un script écrit en Rexx qui gère l'appel de *gcc*. Son contenu pour *Linux* ressemble à

```
#!/usr/bin/regina

/* de quoi tester que la compilation d'un programme C
   sous Linux avec gcc est ok */

parse arg fn prms

/* si pas de paramètre, rappel de la syntaxe */

if words(fn)=0 then do
   say "     syntaxe : pc nomfic [ parms ] "
   say "     exemple : pc demo. oui 10 "
   exit
end

/* élimination de l'ancien résultat de compilation */

"\rm -f " fn

/* compilation et récupération du code-retour */

"gcc -o" fn  " " fn||".c"
sovRc = rc
if sovRc=0 then do
   say  " exécution de : ./"||fn prms
   "./"||fn prms
   "ls -al "fn".*"
end ; else ; do ;
   say " erreur dans la compilation gcc -o" fn  " " fn||".c"
   exit sovRc
end ;
```

6.7 Appel du langage C++

Comme pour le langage C, il existe de nombreux compilateurs pour C++ sous *Dos* comme sous *Unix*. La commande utilisée ppc par galg peut appeler le compilateur voulu.

Nous vous conseillons d'utiliser g++ et de l'appeler via un script Rexx dont le contenu pour *Linux* peut ressembler à :

```
#!/usr/bin/regina

/* de quoi tester que la compilation d'un programme C++
   sous Linux avec g++ est ok */

parse arg fn prms

/* si pas de paramètre, rappel de la syntaxe */

if words(fn)=0 then do
   say "     syntaxe : ppc nomfic parms "
   say "     exemple : ppc demo. oui 10 "
   exit
end

/* élimination de l'ancien résultat de compilation */

"\rm -f " fn

/* compilation et récupération du code-retour */

"g++ -o" fn  " " fn||".cpp"
sovRc = rc
if sovRc=0 then do
   say  " exécution de : ./"||fn prms
   "./"||fn prms
   "ls -al "fn".*"
end ; else ; do ;
   say " erreur dans la compilation g++ -o" fn  " " fn||".cpp"
   exit sovRc
end ;
```

6.8 Appel du langage **Pascal**

La version du langage Pascal utilisée est ppc386. Il s'agit d'une version gratuite, disponible pour *Dos* et pour *Unix* (dont *Linux*) à l'adresse

http://www.freepascal.org/

Pour l'appeler, galg utilise la commande pa. Ce peut être un script Rexx dont le contenu peut ressembler à

```
#!/usr/bin/regina

/* ce script teste que la compilation d'un programme PASCAL est */
/* ok avant de lancer l'exécution et de lister les fichiers de  */
/* même nom ; le compilateur est ppc386 (free pascal compiler). */

parse arg fn prms

/* si pas de paramètre, rappel de l'aide */

if words(fn)=0 then do
   say "     syntaxe : pa nomfic parms "
   say "     exemple : pa demo oui 10 "
   exit
end

/* élimination des anciens résultats de compilation */

"\rm -f " fn
"\rm -f " fn".o"

/* compilation et récupération du code-retour */

"ppc386 " fn
sovRc = rc
if sovRc=0 then do
   "./"||fn prms
   "ls -al "fn" " fn".*"
end ; else ; do ;
   say " erreur de la compilation "ppc386 " fn
   exit sovRc
end ;
```

6.9 Appel du langage Java

Java est gratuit, disponible pour *Dos* et pour *Unix* (dont *Linux*) à l'adresse

http://java.sun.com/

Pour l'appeler, galg utilise la commande `ja`. Ce peut être un script Rexx dont le contenu peut ressembler à

```
#!/usr/bin/regina

/* de quoi tester que la compilation d'un programme Java est ok */

parse arg fn prms

 /* si pas de paramètre, rappel de l'aide */

if words(fn)=0 then do
   say "     syntaxe : ja nomfic [parms] "
   say "     exemple : ja demo oui 10 "
   exit
end

/* élimination de l'ancien résultat de compilation */

"\rm -f " fn||".class"

/* compilation et récupération du code-retour */

"javac " fn||".java"
sovRc = rc
if sovRc=0 then do
   say  " exécution de : ./"||fn||"(.class)" prms
   " java -classpath .:/usr/local/bin/JavaIbm/lib/classes.zip " fn prms
end ; else ; do ;
   say " erreur de la compilation javac " fn||".java"
   exit sovRc
end ;
```

BIBLIOGRAPHIE

A. Aho, J. Hopcroft, J. Ullman
Structures de données et Algorithmes
InterEditions, 1987.

Cormen, Leiserson et Rivest
Introduction to algorithms
M.I.T. Press, 1990.

C. Froidevaux, M. C. Gaudel, M. Soria
Types de donnnées et Algorithmes
Ediscience, 1993.

B. Kernighan, R. Pike
The practice of programming
Addison-Wesley, 1999.

D. E. Knuth
The art of computer programming
Addison-Wesley, 1973.

P. Naudin, C. Quitté
Algorithmique algébrique
Masson, 1992.

J. Orwant, J. Hietaniemi, J. Macdonald
Mastering Algorithms with Perl
O'Reilly, 1999.

R. Sedgewick
Algorithms, second edition
Addison-Wesley, 1989.

S. Skiena
The algorithm design manual
Springer-Verlag, 1998.

H. Wilf
Algorithmes et complexité
Masson, 1989.

N. Wirth
Algorithmes et structures de données
Eyrolles, 1987.