Gilles Vasseur © 2014-2015

Apprendre les bases de la programmation avec un dérivé de LOGO ou se perfectionner en bâtissant l’interpréteur de A à Z…

GVLOGO

Réaliser un interpréteur en Pascal V1.0.0 - 19/02/2015

Table des matières

[I - Le projet 4](#_Toc412110554)

[**01 - Les objectifs du projet** 4](#_Toc412110555)

[**Qu'est-ce que GVLOGO ?** 5](#_Toc412110556)

[**En quoi ce projet peut-il être utile ?** 5](#_Toc412110557)

[**Combien dois-je débourser pour acquérir GVLOGO ?** 6](#_Toc412110558)

[**De quoi ai-je besoin pour travailler ?** 6](#_Toc412110559)

[**Plan du document** 7](#_Toc412110560)

[**Conventions** 7](#_Toc412110561)

[**Contacter l’auteur** 8](#_Toc412110562)

[II - Les objets de GVLOGO 9](#_Toc412110563)

[**02 - Les fichiers communs** 9](#_Toc412110564)

[**La centralisation des données** 9](#_Toc412110565)

[**GVDefines.inc** 9](#_Toc412110566)

[**GVConsts** 10](#_Toc412110567)

[**GVErrConsts** 11](#_Toc412110568)

[**GVPrimConsts** 11](#_Toc412110569)

[**03 - Le traitement des erreurs** 12](#_Toc412110570)

[**L’unité GVErrors** 12](#_Toc412110571)

[**L’utilisation de l’unité** 14](#_Toc412110572)

[**Le programme d’exemple** 16](#_Toc412110573)

[**04 - Les mots** 18](#_Toc412110574)

[**Définitions** 18](#_Toc412110575)

[**Exemples de mots** 18](#_Toc412110576)

[**Opérations sur les mots** 20](#_Toc412110577)

[**Implémentation des mots** 23](#_Toc412110578)

[**05 - Les listes** 32](#_Toc412110579)

[**Définitions** 32](#_Toc412110580)

[**Exemples de listes** 32](#_Toc412110581)

[**Opérations sur les listes** 33](#_Toc412110582)

[**Implémentation des listes** 36](#_Toc412110583)

[**06 - Les listes de propriétés** 42](#_Toc412110584)

[**Définitions** 42](#_Toc412110585)

[**Exemples de listes de propriétés** 42](#_Toc412110586)

[**Opérations sur les listes de propriétés** 42](#_Toc412110587)

[**Implémentation des listes de propriétés** 43](#_Toc412110588)

[**07 - La tortue graphique** 48](#_Toc412110589)

[**Présentation** 48](#_Toc412110590)

[**Opérations avec la tortue** 49](#_Toc412110591)

[**Implémentation de la tortue** 58](#_Toc412110592)

[III - Récréation : EasyTurtle (logiciel de dessin) 75](#_Toc412110593)

[**A - Le projet EasyTurtle** 75](#_Toc412110594)

[**B - Mode d’emploi rapide** 75](#_Toc412110595)

[**L’écran d’accueil** 75](#_Toc412110596)

[**La tortue** 76](#_Toc412110597)

[**Couleurs et formes** 77](#_Toc412110598)

[**Ordres généraux** 77](#_Toc412110599)

[**L’aide** 78](#_Toc412110600)

[**Boîte « À propos »** 79](#_Toc412110601)

[**Boîte des préférences** 79](#_Toc412110602)

[**Autres éléments** 80](#_Toc412110603)

[**C - La programmation** 81](#_Toc412110604)

[**La fiche principale** 82](#_Toc412110605)

[**Les autres fiches** 87](#_Toc412110606)

[IV - Les outils de programmation 88](#_Toc412110607)

[**08 - Les piles** 88](#_Toc412110608)

[**Définition** 88](#_Toc412110609)

[**Exemples** 88](#_Toc412110610)

[**Opérations sur les piles (et les queues)** 91](#_Toc412110611)

[**Implémentation des piles** 92](#_Toc412110612)

[09 - L’évaluation d’une expression mathématique 98](#_Toc412110613)

[Définition 98](#_Toc412110614)

[Opérations dans l’évaluateur 98](#_Toc412110615)

[Implémentation de l’évaluateur 98](#_Toc412110616)

[Les composants du langage 99](#_Toc412110617)

[Le noyau 99](#_Toc412110618)

[Les variables 99](#_Toc412110619)

[Les variables globales 99](#_Toc412110620)

[Les variables locales 99](#_Toc412110621)

[Les procédures 99](#_Toc412110622)

[Les paquets 99](#_Toc412110623)

[Les primitives 99](#_Toc412110624)

[L’interpréteur 99](#_Toc412110625)

[Le programme final 100](#_Toc412110626)

[L’organisation du logiciel 100](#_Toc412110627)

[L’interface utilisateur 100](#_Toc412110628)

[Ce qu’il reste à faire 100](#_Toc412110629)

[Mode d’emploi de GVLOGO 101](#_Toc412110630)

[Installer GVLOGO 101](#_Toc412110631)

[Interface et menus 101](#_Toc412110632)

[Menu Fichier 101](#_Toc412110633)

[Menu Editer 101](#_Toc412110634)

[Menu Voir 101](#_Toc412110635)

[Menu Exécuter 101](#_Toc412110636)

[Menu Fenêtres 101](#_Toc412110637)

[Menu Aide 101](#_Toc412110638)

[Les boutons 101](#_Toc412110639)

[Créer un programme 101](#_Toc412110640)

[Exécuter un programme 101](#_Toc412110641)

[Sauvegarder et récupérer un programme 101](#_Toc412110642)

[Modifier un programme 102](#_Toc412110643)

[Les messages d’erreur 102](#_Toc412110644)

[Le débogage 102](#_Toc412110645)

[Mettre à jour le logiciel 102](#_Toc412110646)

[Programmes exemples 103](#_Toc412110647)

[Travailler avec la tortue 103](#_Toc412110648)

[Travailler avec les listes 103](#_Toc412110649)

[Travailler avec les listes de propriétés 103](#_Toc412110650)

[Manipuler l’espace de travail 103](#_Toc412110651)

[Table des figures 105](#_Toc412110652)

# I - Le projet

## **01 - Les objectifs du projet**

Le projet **GVLOGO** est né en 2014. Il s'agissait d'implémenter le langage LOGO sur un PC (avec Windows ou Linux).

À présent que l’écriture des composants du logiciel est terminée et que ce dernier fonctionne de manière satisfaisante, l'objectif est triple :

* proposer aux programmeurs ou apprentis programmeurs la description de la réalisation complète d'un interpréteur et de son environnement à l'aide d'outils gratuits, évolués, stables et facilement disponibles ;
* offrir un langage de programmation adapté à des enfants, mais dont les possibilités sont étendues grâce au traitement des listes ;
* inviter les nostalgiques des années 70 à des moments de rêverie…

Le langage de programmation choisi pour le développement est un avatar de PASCAL qui est connu pour sa grande lisibilité et sa puissance : Free Pascal sous l’environnement de développement intégré Lazarus. Non seulement Pascal a lui aussi des vertus pédagogiques, mais il fonctionne sur de nombreuses plates-formes (dont Windows et Linux) et est gratuit. Les sources de tout le projet sont par ailleurs largement exploitables avec Delphi, un autre environnement plus puissant que le premier, mais payant.

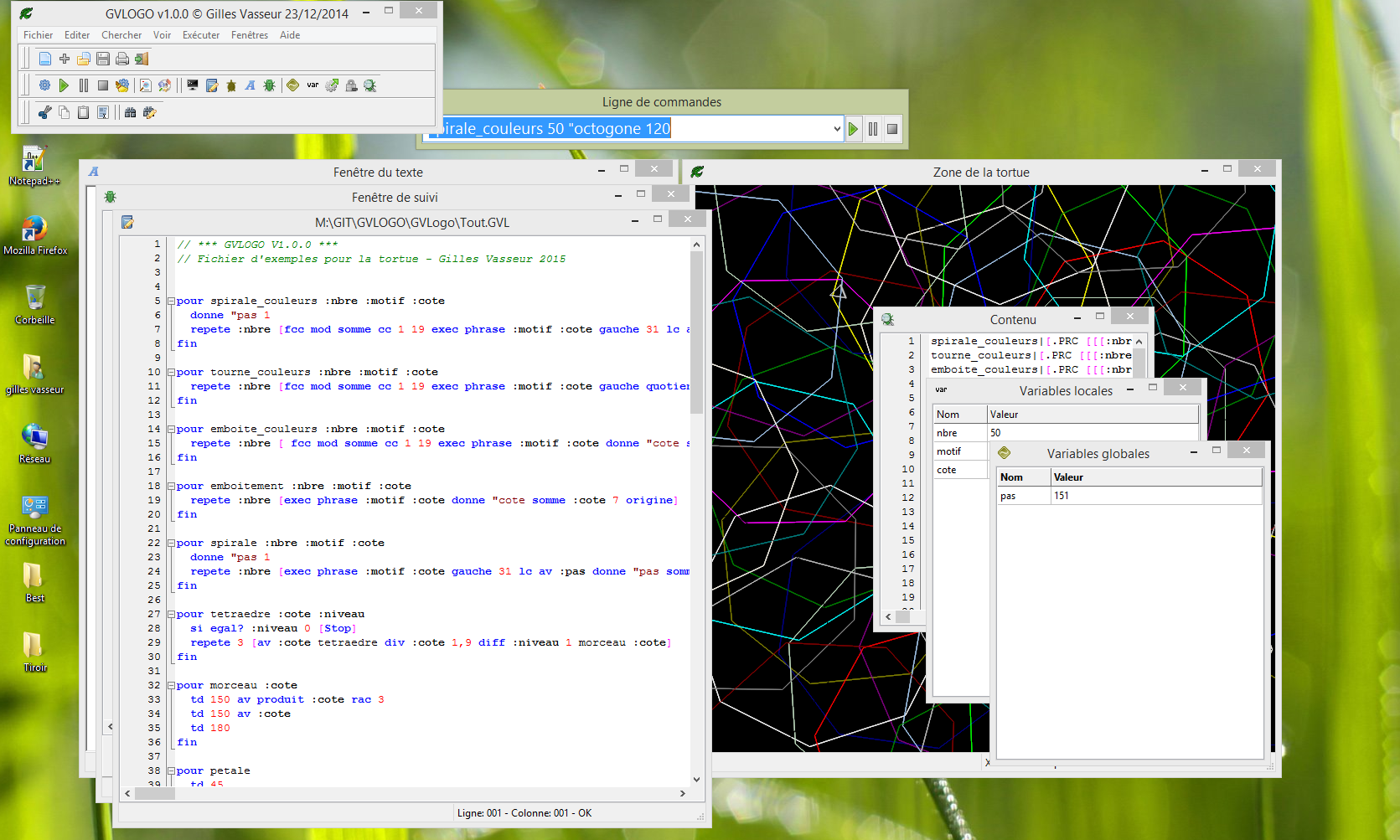


Figure - exemple de GVLOGO en action

## **Qu'est-ce que GVLOGO ?**

**GVLOGO** est un langage de programmation. Il descend de LOGO, lui-même apparu dans les années 60 à la suite de recherches menées par des universitaires du M.I.T. (U.S.A.) autour du mathématicien Seymour Papert.

S'appuyant sur des travaux de Piaget relatifs à l'acquisition de connaissances et de méthodes chez l'enfant, la visée du projet était pédagogique : rendre l'élève acteur de son apprentissage, y compris et surtout lorsque cet apprentissage devait passer par un ordinateur. En cela le projet s'opposait à l'Enseignement Assisté par Ordinateur alors en vogue qui se contentait le plus souvent d'un scénario figé.

« Dans bien des écoles, aujourd’hui, l’expression "Enseignement Assisté par Ordinateur" signifie que l’ordinateur est programmé pour enseigner à l’enfant. On pourrait dire que l’ordinateur sert à programmer l’enfant. Dans ma vision des choses, l’enfant programme l’ordinateur et, ce faisant, acquiert la maîtrise de l’un des éléments de la technologie la plus moderne et la plus puissante, tout en établissant un contact intime avec certaines des notions les plus profondes de la science, des mathématiques, et de l’art de bâtir des modèles intellectuels. »

(S. Papert - Jaillissement de l’esprit, ordinateurs et apprentissage, Flammarion 1981)

Peut-être avez-vous utilisé ce langage au cours de votre scolarité en pilotant sur l’écran de l’ordinateur un triangle (baptisé "tortue") grâce à une série de commandes telles que : AVANCE, RECULE, GAUCHE, DROITE... Comme cette tortue pouvait laisser une trace de son passage, vous obteniez des dessins en programmant ses déplacements. Plus tard, peut-être avez-vous aussi manipulé les listes LOGO pour traiter des problèmes d'intelligence artificielle... Et quand bien même vous n’auriez rien fait de tout cela, peu importe, car programmer en LOGO est très simple !

## **En quoi ce projet peut-il être utile ?**

Si vous ne connaissez rien à la programmation, **GVLOGO** (le produit fini) vous initiera à ses grands principes.

Si vous êtes en contact avec des enfants ou des adolescents, **GVLOGO** leur permettra une première approche de la programmation tout en développant leur sens logique, leur représentation dans l'espace et leur capacité à résoudre des problèmes par eux-mêmes.

Si vous êtes programmeur ou si vous aspirez à l'être, vous pourrez être intéressé par le logiciel lui-même dont les fichiers sources sont fournis et largement commentés. Bien qu’il ne s’agisse pas d’un tutoriel d’apprentissage de la programmation, ce projet met en œuvre de nombreuses techniques *in vivo*. J’ai souvent regretté en pêchant sur Internet de rapporter dans mon filet des poissons trop maigres pour me nourrir ou beaucoup trop gros pour mon estomac : entre l’exemple trivial et la bibliothèque absconse, j’ai pensé qu’il devait y avoir l’espace pour un projet moyennement complexe, à portée d’un programmeur solitaire.

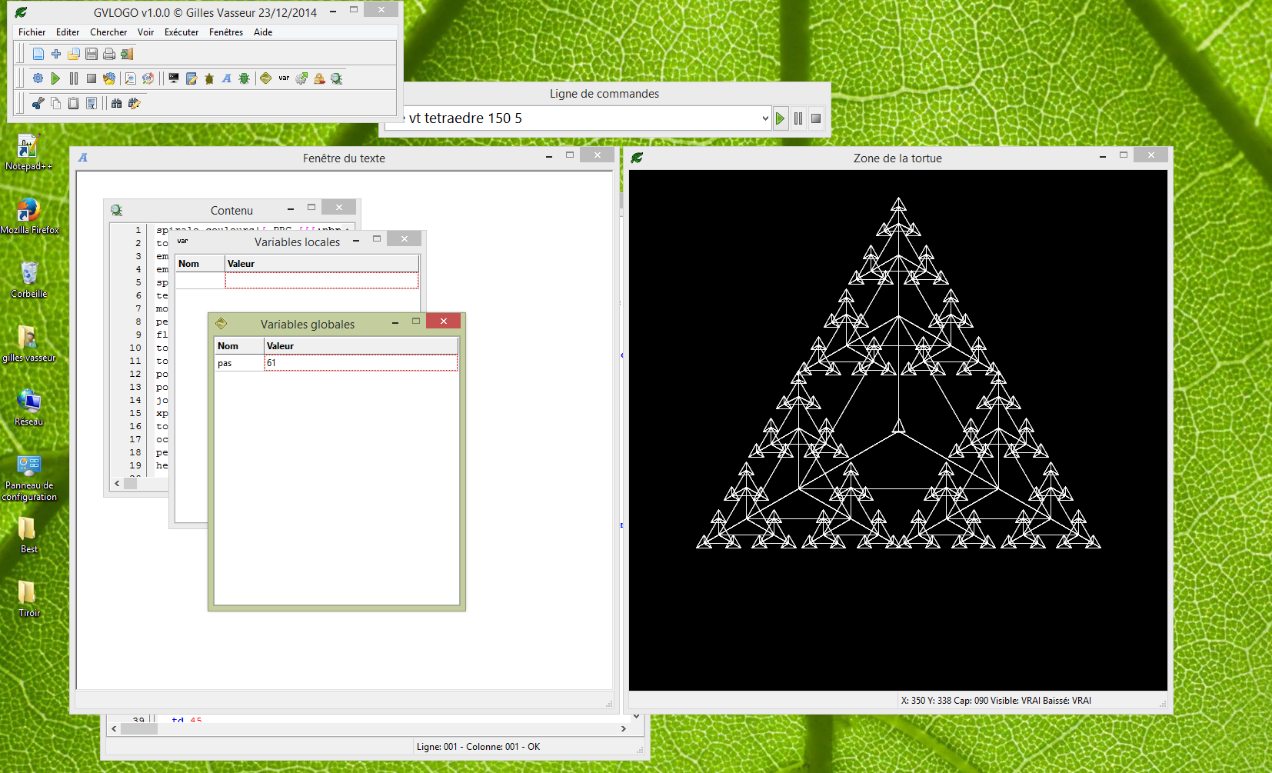


Figure - exemple de GVLOGO en action (2)

## **Combien dois-je débourser pour acquérir GVLOGO ?**

Le logiciel est fourni avec tous les fichiers sources et compilés. **GVLOGO** est protégé par la licence GNU qui garantit le droit de disposer des fichiers fournis sans avoir de redevance à verser à l’auteur. Cependant, ce dernier est protégé quant à l’origine du développement.

La documentation, en particulier le manuel que vous avez devant les yeux, est protégée par un copyright si bien que toute reproduction la concernant, même partielle, doit être soumise à l’auteur pour approbation. Elle est offerte gracieusement avec **GVLOGO**.

Pour résumer, **GVLOGO** est entièrement gratuit. Cela dit, si vous jugez ce logiciel utile et que vous mesurez le temps de travail qu’il représente, rien ne vous interdit de verser une contribution, même symbolique, *via* mon site et mon compte Paypal !

## **De quoi ai-je besoin pour travailler ?**

Tous les outils présentés sont entièrement gratuits.

Pour examiner les fichiers sources fournis, un simple éditeur suffit : je vous conseille un outil adapté aux développeurs comme NotePad++[[1]](#footnote-1) (pour Windows) ou Geany[[2]](#footnote-2) (pour Linux). Il aura au moins l’avantage de faciliter la lecture des fichiers grâce à la coloration syntaxique.

Pour compiler les mêmes fichiers, Lazarus est le choix qui s’impose puisque **GVLOGO** a été développé avec lui. Il fonctionne aussi bien sous Windows que sous Linux. La seule bibliothèque extérieure, elle aussi commune aux deux systèmes d’exploitation, est la bibliothèque graphique **BGRABitmap**[[3]](#footnote-3) : au **TCanvas** standard de Free Pascal, elle ajoute entre autres la possibilité de dessiner avec anticrénelage, de contrôler la transparence et de fusionner des couches, ce qui sera très utile au moment d’implémenter la tortue graphique.

## **Plan du document**

La réalisation de **GVLOGO** est modulaire. Tout d’abord, nous créerons une unité de traitement des **erreurs** qui sera utilisée dans tous les modules. Dans un second temps seront étudiés les objets **GVLOGO** et leur implémentation : les **mots**, les **listes**, les **listes de propriétés** et la **tortue graphique**. Les objets sont en quelque sorte les briques qui fondent le langage.

L’étape suivante consistera à coder les outils nécessaires à l’interpréteur, à savoir les **piles**, un **évaluateur d’expressions mathématiques** et une unité traitant les **variables locales**.

Une place conséquente sera alors consacrée au **noyau**. Comme la mission de ce dernier est de gérer le **contenu des** **éditeurs**, les **variables**, les **procédures** et les **paquets**, tous ces éléments seront tour à tour étudiés

Viendra alors l’**interpréteur** lui-même dont la tâche sera grandement simplifiée par tout le travail réalisé en amont. Pour clore ce qui se rapporte aux fonctionnalités, les **primitives** seront regroupées dans leur propre unité. Il restera à s’occuper de l’architecture de l’**interface,** depuis l’**éditeur** jusqu’au **débogueur**, en passant par la **zone de saisie**, celle de **texte** et celle réservée à la **tortue**.

Enfin, une fois le logiciel opérationnel, un **mode d’emploi** et des **exemples** écrits en **GVLOGO** illustreront quelques-unes de ses capacités.

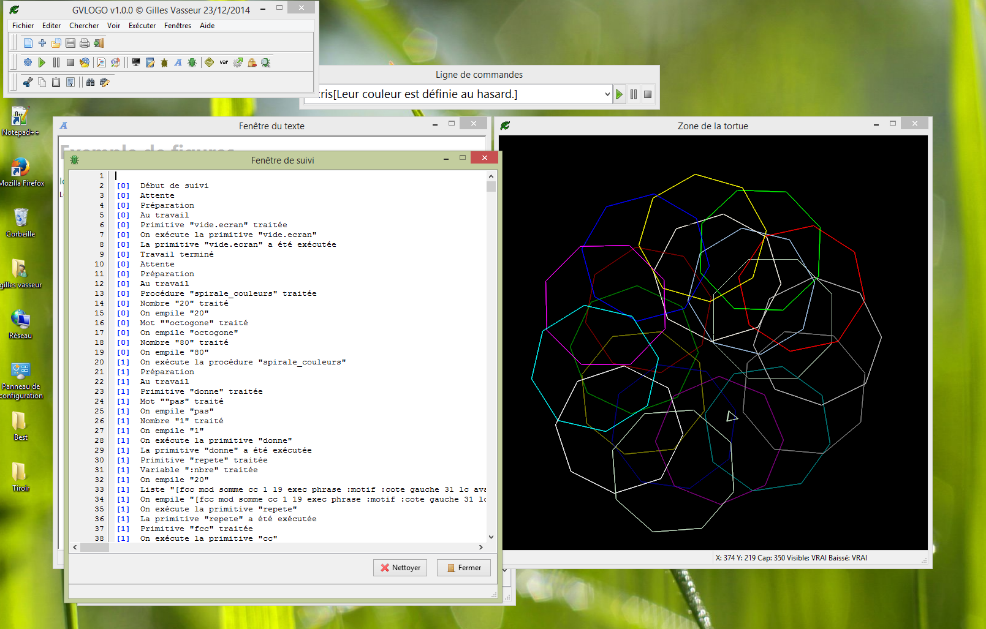


Figure - exemple de GVLOGO en action (3)

## **Conventions**

Les éléments de programmation en **GVLOGO** sont écrits en bleu.

Les éléments de programmation de Pascal, en dehors des listings, sont écrits en rouge. Les classes sont en **rouge gras**.

Les unités utilisées ou créées sont en **noir gras**.

Ce pictogramme indique une astuce, un point intéressant pour faciliter la programmation.

Ce pictogramme indique un point à souligner.

 Ce pictogramme attire l’attention sur un point sensible, une difficulté particulière.

## **Contacter l’auteur**

Il existe sans doute des méthodes plus efficaces que celles que j'ai employées pour résoudre les problèmes liés à l'implémentation d'un interpréteur LOGO, mais je tenterai de justifier mes choix. Par exemple, l’interpréteur de formules mathématiques a été traité à partie de l’algorithme de « shunting-yard » alors que les solutions courantes font plutôt appel à des arbres : il s’agissait justement de varier les approches et non de reproduire sans cesse les mêmes schémas, car c’est confronté à quelques difficultés que le programmeur, quel que soit son niveau, peut espérer progresser.

Bien entendu, certains aspects du programme sont à améliorer : je pense notamment à la vitesse d’exécution de l’ensemble, même si elle reste tout à fait acceptable. En général, les approches pédagogiques ont été privilégiées, souvent au détriment des performances. Il sera toujours possible de poser des questions, de critiquer les réalisations et de proposer des solutions plus élégantes et/ou plus efficaces que les miennes !

Enfin, l’écriture de milliers de lignes est une aventure à la limite des possibilités humaines : des erreurs émaillent forcément les unités produites. Aussi peu nombreux que possibles mais forcément présents, les bogues empoisonnent la vie des utilisateurs et celle des programmeurs : que chacun se fasse un devoir de les signaler et de proposer des solutions pour leur éradication !

Pour cela, n’hésitez pas à envoyer un mail à : [gillesvasseur58@gmail.com](mailto:gillesvasseur58@gmail.com)

Vous pouvez aussi me joindre *via* mon site personnel : [www.lettresenstock.org](http://www.lettresenstock.org)

Bonne lecture, bonne réflexion… et bon LOGO !

**Gilles Vasseur, jeudi 19 février 2015**.

# II - Les objets de GVLOGO

## **02 - Les fichiers communs**

### **La centralisation des données**

Les avantages d’une centralisation de certaines données sont de deux types :

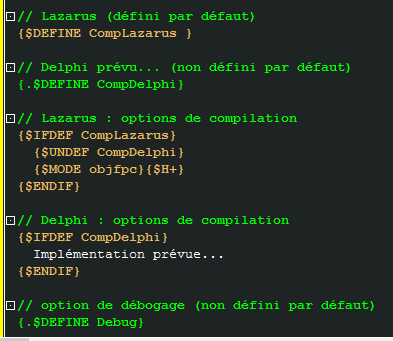
* il est possible modifier un élément à partir d’un seul point et donc inutile de parcourir tous les fichiers sources éparpillés dans plusieurs unités : on imagine le gain de temps dans la situation où l’on devrait changer un contenu utilisé plusieurs dizaines de fois ;
* le risque d’erreurs difficiles à déceler est moindre puisqu’il n’y a pas à recopier un élément pour l’utiliser ailleurs : coder en dur une chaîne de caractères, par exemple, expose à une copie erronée qui donnera des résultats inattendus lors de comparaisons.

La centralisation des données s’organise autour de quatre fichiers : **GVDefines.inc**, **GVConsts**, **GVErrConsts** et **GVPrimConsts**. Le projet final **GVLOGO** utilise par ailleurs un fichier qui lui est propre et sur lequel nous reviendrons en temps utile : **GVLOGOConsts**.

### **GVDefines.inc**

Le fichier **GVDefines.inc** contient les options de compilation. Il permettra par exemple d’optimiser la compilation lors d’une transposition du logiciel vers Delphi. Son contenu s’organise par conséquent autour de directives.

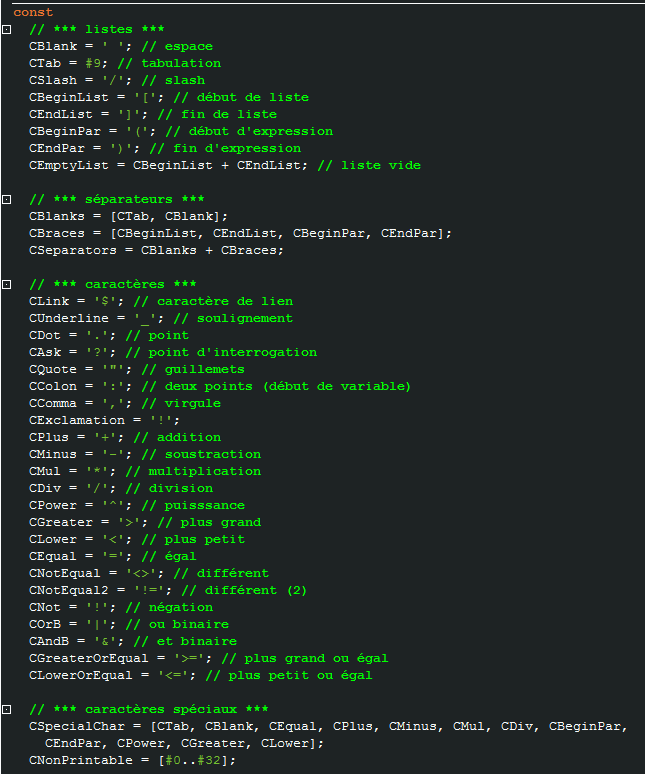
Actuellement, il ne prend en compte que la compilation avec Lazarus, provoquant une erreur si l’on tente d’activer Delphi :



### **GVConsts**

L’unité **GVConsts** servira pour toutes les unités du projet. Elle contient en effet les constantes, types et chaînes de caractères partagés par l’ensemble du projet.

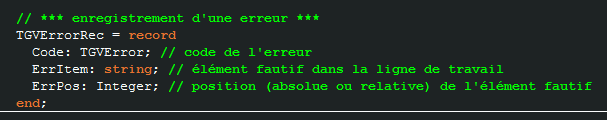
Dans un premier temps, on notera les constantes les plus fréquemment utilisées dans les différentes unités :



Les autres éléments de cette unité seront étudiés lors de la description des unités qui y font appel.

### **GVErrConsts**

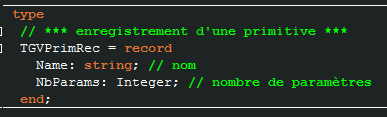
L’unité **GVErrConsts** abrite les éléments de base utiles pour la gestion des erreurs. Elle contient les chaînes des messages, un type énumération qui contient les erreurs possibles et un tableau reprenant ces messages. Un enregistrement décrit une erreur :



### **GVPrimConsts**

L’unité **GVPrimConsts** fait de même, mais pour ce qui concerne les primitives du langage **GVLOGO,** c’est-à-dire les mots qui renvoient à une action/commande comprise d’origine par le langage.

L’enregistrement d’une primitive comprend son nom et le nombre de paramètres qu’elle attend :



L’unité contient alors la liste des noms possibles et un tableau des enregistrements des primitives.

On remarquera que la partie implémentation de ces unités est vide, le rôle de ces dernières étant seulement de déclarer des types, des chaînes et des constantes.

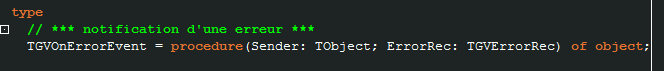
 L’utilisation de resourcestring facilitera la traduction du logiciel en une autre langue. Toutefois, l’utilisation de tableaux fait que les chaînes traitées selon ce mode ne seront pas traduites automatiquement[[4]](#footnote-4).

## **03 - Le traitement des erreurs**

Il peut paraître surprenant de commencer par la gestion des erreurs, mais la réalisation d’une unité utilisée par tous les modules nécessaires au projet permettra de simplifier leur écriture. L’objectif est de fournir un outil simple de centralisation des erreurs : à partir d’un minimum de déclarations, tout module gèrera de manière homogène une éventuelle erreur.

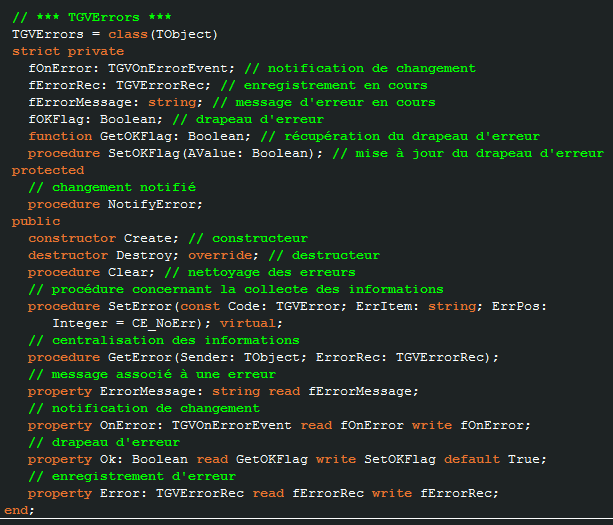
### **L’unité GVErrors**

L’unité **GVErrors** concentre les outils nécessaires à cette gestion des erreurs. L’élément fondamental de communication est le gestionnaire d’événements relatifs aux erreurs :



Comme tout gestionnaire d’événements, il propose un modèle de procédure (ou, plus rarement, de fonction) avec une signature particulière (ici, un Sender de type **TObject** et un enregistrement d’erreur ErrorRec de type **TGVErrorRec** tel que défini plus haut dans l’unité **GVErrConsts**).

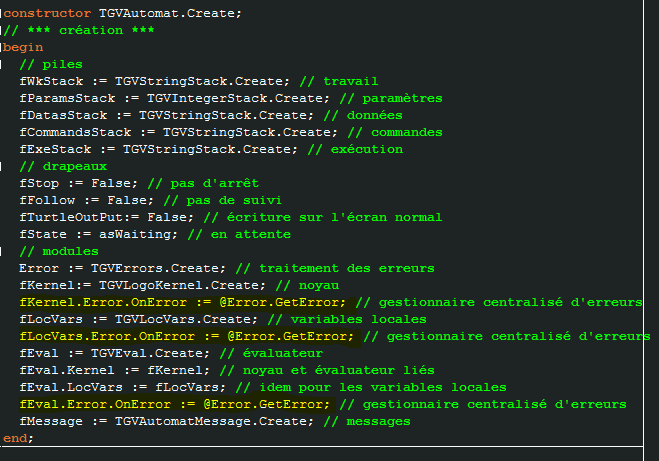
Voici l’interface de la classe **TGVErrors** définie dans **GVErrors** :



ErrorMessage transforme le numéro d’une erreur en un message lisible. La propriété Error permet d’accéder à l’enregistrement de l’erreur. La propriété OK permet de tester si une erreur a été rencontrée ou non.

Plus intéressant est le gestionnaire OnError qui sera activé à chaque erreur rencontrée. C’est par son intermédiaire que les erreurs seront traitées par l’unité appelante. SetError et GetError permettent de gérer les données associées aux erreurs. La méthode GetError a une signature compatible avec le type **TGVOnErrorEvent** si bien qu’elle peut récupérer les erreurs d’une unité qui gérerait elle-même un objet de type **TGVErrors**!

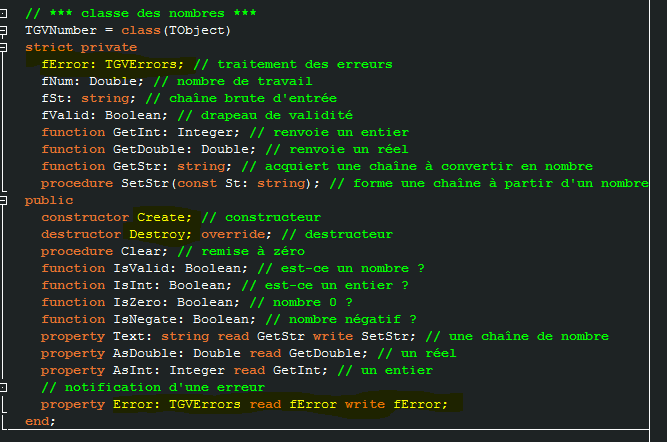
Voici par exemple comment l’interpréteur récupère et centralise les erreurs détectées dans les objets qu’il manipule :



Une fois assuré le traitement local des erreurs par la création de l’objet Error (instance de la classe **TGVErrors**), les objets fKernel, fLocVars et fEval voient leurs gestionnaires d’erreurs reliés à celui de leur hôte. À présent, toute erreur déclenchée dans leur sein sera répercutée dans Error de l’interpréteur.

### **L’utilisation de l’unité**

Le processus de déclaration est toujours le même. Voici, par exemple, la classe **TGVNumber** qui traite les nombres :

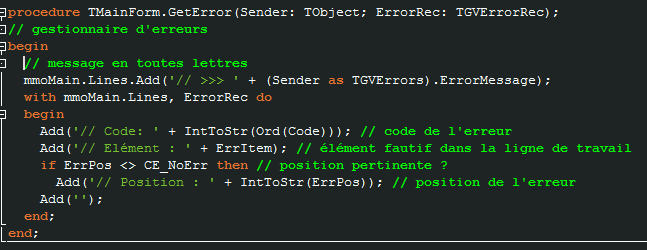


On crée une propriété qui renvoie à une variable privée. Dans le constructeur **Create**, on crée l’instance de classe tandis qu’on la libère dans le destructeur **Destroy**. Il suffit à présent de générer quand nécessaire une erreur pour qu’elle puisse être exploitée : on utilisera la méthode **SetError** qui requiert un numéro d’erreur, le nom de l’élément ayant déclenché l’erreur et facultativement un emplacement de l’erreur dans l’élément.

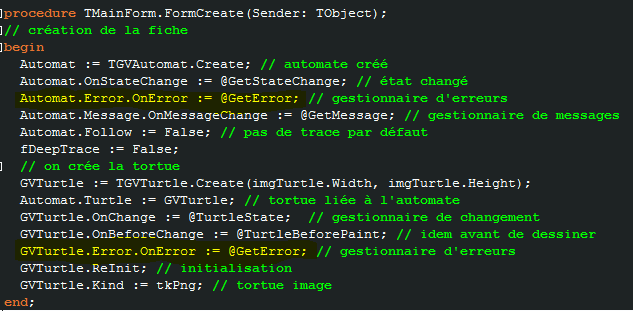
Dans le programme appelant, on crée une méthode d’affichage des erreurs :



Voici le listing de cette méthode qui affiche les données associées à l’erreur :



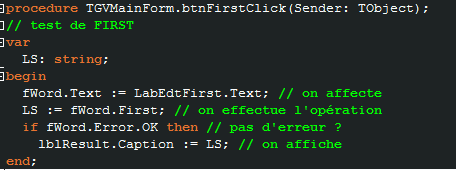
Lorsqu’on crée la fiche de travail, on relie cette méthode à l’unité de gestion des erreurs :



 On remarquera que le même gestionnaire est ici partagé par deux unités : c’est tout l’intérêt de notre centralisation des erreurs que de n’avoir qu’une seule méthode de traitement à coder.

A présent, tout ce qu’il reste à faire, c’est de vérifier l’état d’erreur de l’unité. Si une erreur a été levée, elle a été traitée.

Il n’est par conséquent pas besoin de tester quoi que ce soit d’autre :



Dans la méthode donnée en exemple, on récupère le contenu d’un composant **TLabelEdit** qu’on transforme en mot formaté. Le second mot issu lui aussi d’un **TLabelEdit** est comparé au premier. S’il n’y a pas eu d’erreur, on affiche le résultat, à savoir le mot qui précède l’autre par ordre alphabétique. En cas d’erreur, un message aura automatiquement été affiché.

### **Le programme d’exemple**

Les versions proposées pour le programme d’exemple sont au nombre de deux :

* une version Lazarus Win32 ;
* une version Lazarus Linux.

On trouvera les fichiers sources et les exécutables dans le sous-dossier « 00 – testgverrors » du dossier « tests ».

Toutes les unités produites seront ainsi testées par un programme autonome. Cette démarche permet d’isoler les éventuels problèmes : il sera bien plus facile de repérer une erreur dans un module que dans le programme final qui comportera des milliers de lignes !

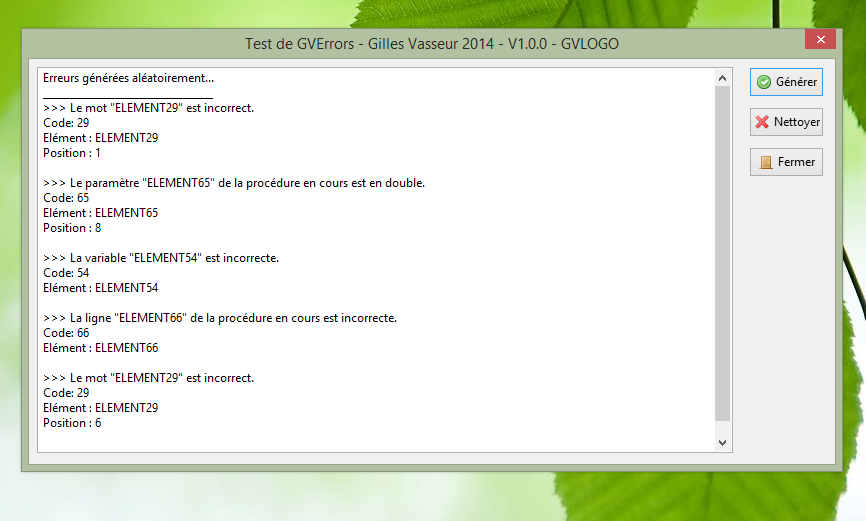


Figure - Test de GVErrors

En dehors de la gestion des erreurs déjà étudiée, le programme est tout à fait trivial. Un composant **TMemo** affiche les messages d’erreurs. Chacune d’entre elles est générée de manière aléatoire.

## **04 - Les mots**

### **Définitions**

**GVLOGO** travaille essentiellement à partir de mots et de listes. Ce chapitre se propose d’étudier les mots.

Un *mot* est une suite quelconque de caractères.

Un *mot* est délimité par un espace, les signes **[**, **]**, **(**, **)**. Ces cinq caractères particuliers sont appelés des *délimiteurs* : ils indiquent à **GVLOGO** comment séparer les éléments du langage.

Afin d'inclure un délimiteur dans un mot, on le fait précéder d'un autre caractère appelé caractère d’échappement : c'est le **$**. Si ce caractère d'échappement doit apparaître en tant que tel dans un mot, il faut le faire précéder de lui-même !

Il existe un *mot* particulier qui ne comprend aucun caractère : on l'appelle le *mot vide*. On le représente par un guillemet anglais suivi d'un espace.

 Les mots "VRAI et "FAUX, qui peuvent s'écrire simplement VRAI et FAUX, sont réservés au renvoi de valeurs booléennes. Pour **GVLOGO**, la valeur « vrai » est représentée par le nombre -1 alors que la valeur « faux » l’est par le nombre 0.

### **Exemples de mots**

#### **Mots simples**

Voici des mots simples qui ne poseront pas de problèmes particuliers :

* libellule
* porte-avions
* entendre
* EnTenDRe

Les nombres ne sont que des mots particuliers qui seront traités comme des mots ordinaires ou comme des nombres, suivant l'opération en cours :

* 1245
* 1E2[[5]](#footnote-5)
* 13456,58

Les mots accentués et les caractères exotiques sont utilisables eux aussi, à condition de ne pas être choisis en tant qu’identificateur:

* éléphant
* %&/ç\

 Les caractères particuliers comme ceux qui sont accentués ne sont pas toujours bien gérés par les langages de programmation et les systèmes d'exploitation. Les normes se sont multipliées et des hiéroglyphes étranges peuvent parfois apparaître. Le travail de développement de Lazarus est largement déterminé par ce problème. Nous y reviendrons lorsque nous aborderons l'implémentation des mots. Disons que, pour le moins, il s’agit d’un point délicat susceptible de troubler le programmeur.

#### **Mots avec caractère d'échappement**

Un caractère pourtant interdit dans un mot **GVLOGO** peut toujours être introduit s'il est précédé du caractère d'échappement :

* Victor$ Hugo
* GVLOGO$ est$ un$ langage$ facile$ à$ apprendre
* $[encore$ un$ mot$]
* $(un$ autre$)
* 15$ $$

Lors de l'écriture des mots ainsi construits, le caractère d'échappement, sauf contre-ordre, n'apparaîtra pas à l'écran :

* ECRIS $[encore$ un$ mot$] donnera [encore un mot]
* ECRIS 15$ $$ donnera 15 $

 Dès qu'une suite de mots est en jeu, l'emploi des listes est préconisé. Le caractère d'échappement est une commodité, sans plus. Il est plus facile d'écrire et de comprendre ECRIS [15 $] que ECRIS 15$ $$.

#### **Autres caractères particuliers**

En plus des délimiteurs, certains caractères sont utilisés par le langage lui-même et déterminent son comportement selon le contexte :

* le **"** (guillemet anglais) indique que le mot qui suit doit être pris tel quel, sans chercher à l'interpréter (par exemple, ECRIS "Bonjour.) ;
* les **:** (deux-points) indiquent que le mot qui suit est une variable dont la valeur est à rechercher (par exemple, ECRIS :salutation affichera, si elle existe, la valeur qui correspond à la variable « salutation ») ;
* le **.** (point) en début de mot indique une primitive dont l'emploi est risqué, car elle touche au cœur de **GVLOGO** (par exemple .EFFACE.TOUT détruit tous les objets en réinitialisant le noyau).
* le **?** (point d'interrogation) est utilisé à la fin des primitives qui renvoient une valeur booléenne (VRAI/FAUX), mais, comme le point, son utilisation est une question d’habitude et n'a pas de caractère obligatoire.

### **Opérations sur les mots**

Ci-après, le lecteur trouvera les primitives connues de **GVLOGO** qui concernent les mots. Pour chaque primitive, avant sa définition, il est indiqué le nombre de paramètres attendus, leurs types et le type de valeur qu'elle renvoie. Au moins un exemple illustre l'emploi de la primitive décrite.

Les primitives énumérées peuvent évidemment se combiner pour obtenir le résultat escompté : c’est même ainsi qu’elles prennent tout leur intérêt ! On y reviendra dans la partie qui traitera de la réalisation de programmes en **GVLOGO**.

#### **Fabriquer des mots**

* **METS.PREMIER** (raccourci : **MP**) : attend deux mots en entrée – renvoie un mot – le mot rendu est composé du premier paramètre placé avant le second.

Exemple :

ECRIS METS.PREMIER "tourne "dos → tournedos

* **METS.DERNIER** (raccourci : **MD**) : attend deux mots en entrée – renvoie un mot – le mot rendu est composé du premier paramètre placé après le second.

Exemple :

ECRIS METS.DERNIER "tourne "dé → détourne

* **MOT**: attend deux mots en entrée – renvoie un mot – le mot rendu est composé du second paramètre placé après le premier.

Exemple :

ECRIS MOT "dé "tourne → détourne

 Il s'agit donc d'un synonyme de METS.DERNIER. Cependant, cette dernière primitive fonctionne aussi avec les listes alors que MOT déclenchera une erreur si une liste est fournie en entrée.

#### **Modifier des mots**

* **INSERE**: attend un entier suivi deux mots en entrée – renvoie un mot – Le mot rendu est composé du deuxième paramètre inséré dans le dernier paramètre à la position précisée par l’entier.

Exemple :

ECRIS INSERE 4 "en "atttion → attention

* **INVERSE**: attend un mot en entrée – renvoie un mot – le mot rendu est celui d'entrée dont les lettres ont été inversées.

Exemple :

ECRIS INVERSE "billet → tellib

* **MAJUSCULES**: attend un mot en entrée – renvoie un mot – le mot rendu est en majuscules.

Exemple :

ECRIS MAJUCULES "éléphant → ĒLĒPHANT

* **MELANGE**: attend un mot en entrée – renvoie un mot – le mot rendu a les mêmes lettres que celui d'origine, mais dans un ordre aléatoire.

Exemple :

ECRIS MELANGE "alouette → eoetault

* **MINUSCULES**: attend un mot en entrée – renvoie un mot – le mot rendu est en minuscules.

Exemple :

ECRIS MINUSCULES "ESSAI → essai

* **REMPLACE**: attend un entier suivi de deux mots en entrée – renvoie un mot – le mot rendu est composé de toutes les lettres du second mot en entrée, sauf celle visée par l'entier qui a été remplacée par le premier mot.

Exemple :

ECRIS REMPLACE 3 "é "éliment → élément

* **TRIE**: attend un mot en entrée – renvoie un mot – les lettres du mot sont triées par ordre alphabétique[[6]](#footnote-6).

Exemple :

ECRIS TRIE "important → aimnoprtt

* **ROTATION** : attend un mot en entrée – renvoie un mot – la primitive renvoie le mot dont le premier caractère est placé à la fin.

Exemples :

ECRIS ROTATION "essai → ssaie

#### **Extraire depuis des mots**

* **ELEMENT**: attend un entier suivi d'un mot en entrée – renvoie un mot – le mot rendu est composé de la lettre du mot en entrée indiquée par l'entier fourni.

Exemple :

ECRIS ELEMENT 4 "Zorro → r

* **DERNIER** (raccourci : **DER**): attend un mot en entrée – renvoie un mot – le mot renvoyé est composé de la dernière lettre du mot en entrée.

Exemple :

ECRIS DERNIER "important → t

* **HASARD**: attend un mot en entrée – renvoie un mot – le mot renvoyé est composé d'une lettre du mot en entrée, tirée au hasard. Si le mot en entrée est un nombre, le mot renvoyé est un nombre compris entre 1 et ce nombre.

Exemples :

ECRIS HASARD "destin → i

ECRIS HASARD "destin → d

ECRIS HASARD "10 → 8

* **PREMIER** (raccourci : **PREM**): attend un mot en entrée – renvoie un mot – le mot renvoyé est composé de la première lettre du mot en entrée.

Exemple :

ECRIS PREMIER "important → i

* **SAUF.DERNIER** (raccourci : **SD**): attend un mot en entrée – renvoie un mot – le mot renvoyé est le mot en entrée amputé de sa dernière lettre.

Exemple :

ECRIS SAUF.DERNIER "important → importan

* **SAUF.PREMIER** (raccourci : **SP**): attend un mot en entrée – renvoie un mot – le mot renvoyé est le mot en entrée amputé de sa première lettre.

Exemple :

ECRIS SAUF.PREMIER "important → mportant

#### **Tester des mots**

* **PLG?** : attend deux mots en entrée – renvoie un booléen – la primitive renvoie "VRAI si le premier mot vient strictement après le second selon l'ordre alphabétique, "FAUX sinon.

Exemples :

ECRIS PLG? "petit "grand → -1

ECRIS PLG? 1245 138 → -1

Comme EGAL? et PLP?, PLG? se comporte différemment si les deux mots sont des nombres : ce sont leurs valeurs qui sont alors comparées.

* **PLP?** : attend deux mots en entrée – renvoie un booléen – la primitive renvoie "VRAI si le premier mot vient strictement avant le second selon l'ordre alphabétique, "FAUX sinon.

Exemple :

ECRIS PLP? "petit "grand → 0

* **COMPTE**: attend un mot en entrée – renvoie un entier – la primitive renvoie le nombre de caractères du mot en entrée.

Exemple :

ECRIS COMPTE "urticaire → 9

* **EGAL?** : attend deux mots en entrée – renvoie un booléen – la primitive renvoie "VRAI si le premier mot est égal au second selon l'ordre alphabétique, "FAUX sinon.

Exemple :

ECRIS EGAL? "100 "1E2 → -1

* **IDENTIFICATEUR?** : attend un mot en entrée – renvoie un booléen – la primitive renvoie "VRAI si le mot en entrée est un identificateur correct, "FAUX sinon.

Exemples :

ECRIS IDENTIFICATEUR? "MaFonction12 → -1

ECRIS IDENTIFICATEUR? "12MaFonction → 0

Un identificateur est correct quand il ne comprend que des lettres non accentuées, des chiffres non placés en première position, le signe **\_** (souligné) et le signe **.** (point) en début de mot, ou le **?** (point d'interrogation) à la fin.

* **MEMBRE?** : attend deux mots en entrée – renvoie un booléen – la primitive renvoie "VRAI si le premier mot est compris dans le second, "FAUX sinon.

Exemple :

ECRIS MEMBRE? "an "étonnant → -1

* **MOT?** : attend un objet en entrée – renvoie un booléen – la primitive renvoie "VRAI si l'objet est un mot, "FAUX sinon.

Exemple :

ECRIS MOT? [coucou] → 0

* **NOMBRE?** : attend un mot en entrée – renvoie un booléen – la primitive renvoie "VRAI si le mot est un nombre, "FAUX sinon.

Exemple :

ECRIS NOMBRE? "12458 → -1

Le nombre est ici précédé d’un guillemet anglais, comme le serait un mot ordinaire. Pour les nombres, cette écriture est facultative.

* **PRECEDENT** : attend deux mots en entrée – renvoie un mot – la primitive renvoie le mot qui vient le premier selon l'ordre alphabétique.

Exemple :

ECRIS PRECEDENT "un "deux → deux

* **SUIVANT** : attend deux mots en entrée – renvoie un mot – la primitive renvoie le mot qui vient le dernier selon l'ordre alphabétique.

Exemple :

ECRIS SUIVANT "un "deux → un

* **VIDE?** : attend un mot en entrée – renvoie un booléen – la primitive renvoie "VRAI si le mot en entrée est le mot vide, "FAUX sinon.

Exemple :

ECRIS VIDE? " → -1

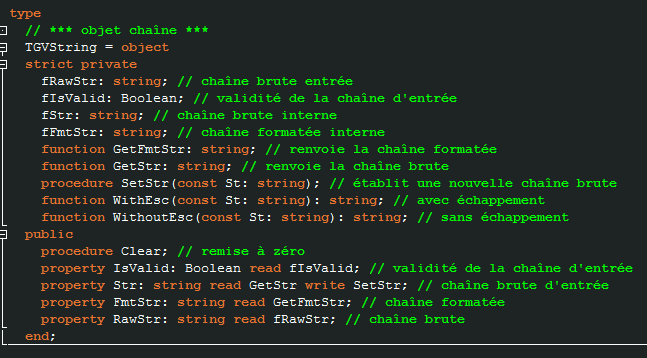
### **Implémentation des mots**

L’implémentation des mots passe par l’unité **GVConsts** qui contient les constantes centralisées du projet et une unité baptisée **GVWords** qui abrite les classes nécessaires à la gestion proprement dite des mots.

#### **L’objet TGVString**

La difficulté essentielle dans le traitement des mots en **GVLOGO** réside dans la présence d’un caractère d’échappement dont il faut tenir compte pour toutes les opérations portant sur eux. L’objet **TGVString** accomplit le travail de normaliser tout mot qui lui est proposé. Il est capable de restituer le mot d’origine ainsi que celui transformé.

Voici sa partie interface :



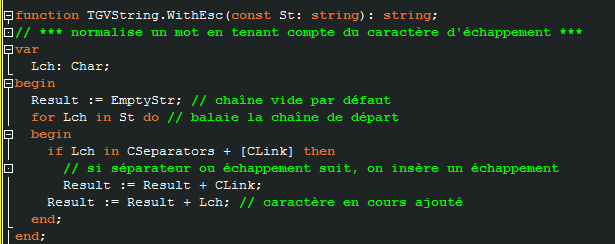
L’interface est bâtie de manière « classique » en proposant des propriétés dont l’implémentation est protégée par des méthodes privées. La propriété Str lit une chaîne afin de la transformer en chaîne normalisée et de la restituer ainsi. La propriété en lecture seule RawStr permet de retrouver la chaîne brute, telle qu’elle a été introduite.

La méthode SetStr est chargée de transformer la chaîne fournie en entrée. Elle invoque successivement WithoutEsc qui retire un éventuel formatage et WithEsc qui normalise la chaîne. Cette façon de procéder évite de formater une chaîne qui le serait déjà : en revanche, elle exige deux analyses de la chaîne et ne repère pas les mots normalisés par hasard !

Un objet a été préféré à une classe dans le cas bien précis où aucun appel à un constructeur et/ou à un destructeur n’était nécessaire. En effet, l’objet n’utilise pas de méthodes virtuelles et n’utilise aucune classe particulière.

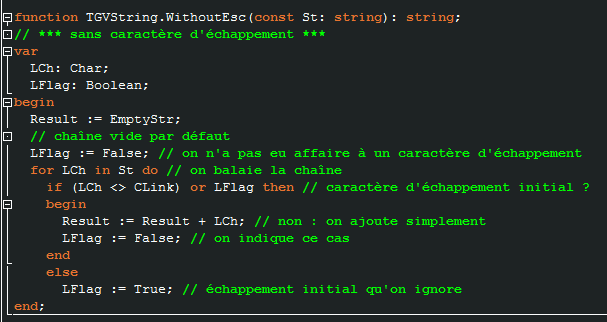
 Encore un avantage à déléguer un travail particulier à un objet ou à une classe : un meilleur algorithme pourra être implémenté dans cette classe sans avoir à modifier la moindre ligne des autres fichiers. Autrement dit, une fois l’interface bien définie, il n’importe en rien aux classes qui s’en servent de savoir comment le problème est traité : seul le résultat compte. Il est même très judicieux d’imposer qu’un objet ou une classe n’ait aucun présupposé à faire avant d’utiliser une autre classe.

L’algorithme utilisé par WithEsc est trivial. On remarquera cependant l’utilisation de l’énumération for…in qui balaie la chaîne en lieu et place d’une boucle traditionnelle for…next moins lisible :



Les utilisateurs de Delphi devront remplacer in CSeparators par la fonction CharInSet s’ils ne veulent pas obtenir un message d’avertissement.

La méthode WithoutEsc n’intéressera que pour l’utilisation d’un drapeau (variable LFlag) qui mémorise le fait d’avoir trouvé précédemment un caractère d’échappement. Ce drapeau est remis à zéro lorsqu’on a traité son repérage :

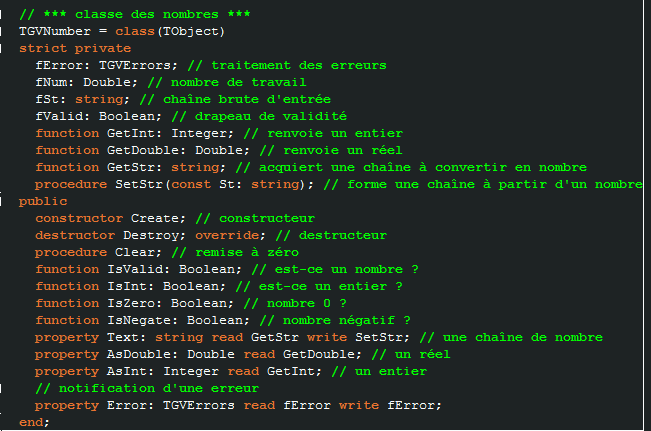


Les utilisateurs de Delphi pourront envisager de transformer cet objet en enregistrement. En effet, cet EDI accepte des enregistrements comprenant des méthodes.

#### **La classe TGVNumber**

La classe **TGVNumber** traite les chaînes susceptibles de représenter des nombres.

Voici sa partie interface :



La partie publique comprend, outre le constructeur et le destructeur, une procédure Clear qui remet à zéro le nombre interne, quatre fonctions utiles et quatre propriétés.

L’entrée dans la classe se fait par la propriété AsString. Si possible, la chaîne est de manière interne transformée en nombre. En cas d’échec, la fonction IsValid retournera False comme résultat. L’accès en lecture d’un nombre erroné déclenchera une erreur.

La lecture peut être effectuée de trois manières : sous forme de chaîne (AsString), d’entier (AsInt) ou de nombre réel (AsDouble). Il est par ailleurs possible de tester le nombre interne pour savoir s’il s’agit d’un entier grâce à la fonction IsInt. La fonction IsNegate permet de savoir si le nombre est négatif. Enfin, la fonction IsZero compare de manière correcte des nombres réels qui seraient jugés inégaux pour des raisons de représentation interne.

L’implémentation ne pose pas de problèmes particuliers. Voici celle de SetStr qui gère l’entrée d’une nouvelle chaîne à traduire :



On conserve le paramètre d’entrée et on le teste grâce à la fonction TryStrToFloat. Le drapeau privé fValid sera vérifié avant de renvoyer une valeur : l’utilisateur est ainsi assuré de toujours disposer d’une valeur correcte ou d’obtenir le déclenchement d’une erreur.

 En règle générale, il faut éviter de court-circuiter les erreurs et toujours avertir l’utilisateur que la valeur dont il dispose est erronée. En croyant bien faire pour l’utilisateur final, on le prive dans le cas contraire de tout repère sur la stabilité de son système et sur la confiance qu’il peut accorder aux résultats qu’il obtient.

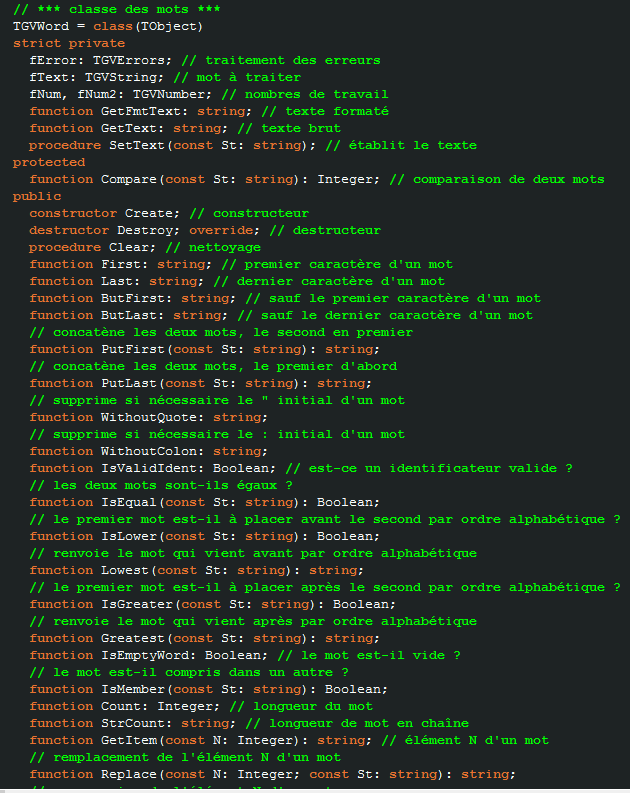
#### **La classe TGVWord**

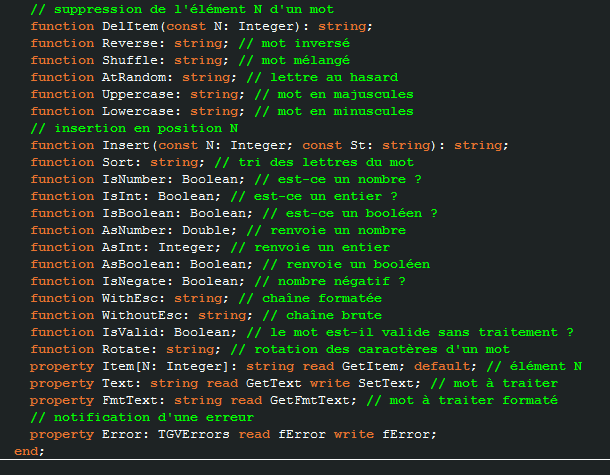
La classe **TGVWord** est de loin la plus fournie de cette unité. Elle gère les mots, qu’ils soient des chaînes de caractères ou des nombres, en reprenant toutes les fonctionnalités définies dans la description du langage **GVLOGO** qui concernent les mots.

La classe utilise deux champs privés de la classe **TGVString** et autant de la classe **TGVNumber**: ils servent d’outils de travail et de comparaisons. L’emploi de ces classes évite d’avoir constamment à vérifier la présence des caractères spéciaux.

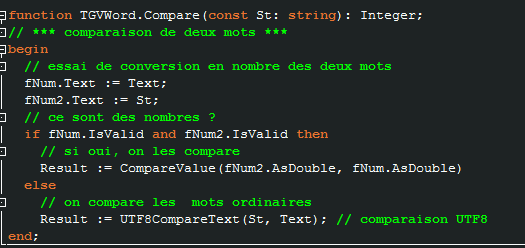
 En fait, la difficulté essentielle tient aux lacunes de Lazarus dans la gestion des caractères accentués. Les fonctions préfixées par UTF8 présentes dans l’unité **lazutf8** constituent une solution possible qui règle les problèmes, sauf ceux relatifs à l’ordre alphabétique : ainsi, la fonction Sort ne triera pas correctement les lettres d’un mot, rejetant les caractères accentués après les caractères normaux. Delphi ne présente pas ce problème. On notera par ailleurs que la fonction Sort utilise un tri à bulles bien suffisant pour des quantités très limitées d’objets à trier.

En voici l’interface :





La méthode protégée Compare s’occupe des comparaisons : il faut tenir compte de la comparaison de chaînes ordinaires et de nombres. En effet, 99 vient après 121 par ordre alphabétique, mais avant par ordre de grandeur mathématique !



#### **Test de l’unité GVWords**

Le meilleur moyen de s’approprier les méthodes de la classe **TGVWord** est de les voir fonctionner dans un programme.

Les versions proposées sont au nombre de deux :

* une version Lazarus Win32 ;
* une version Lazarus Linux.

On trouvera les fichiers sources et les exécutables dans le sous-dossier « 01 – testgvwords » du dossier « tests ».

Si Delphi offre de meilleures capacités à traiter les caractères Unicode et s’il facilite une présentation plus moderne grâce aux styles, Lazarus se montre à la hauteur dans sa portabilité sur différents OS : il n’a pas été nécessaire d’apporter la moindre modification aux fichiers pour faire fonctionner le programme aussi bien sur Windows que sur Linux.

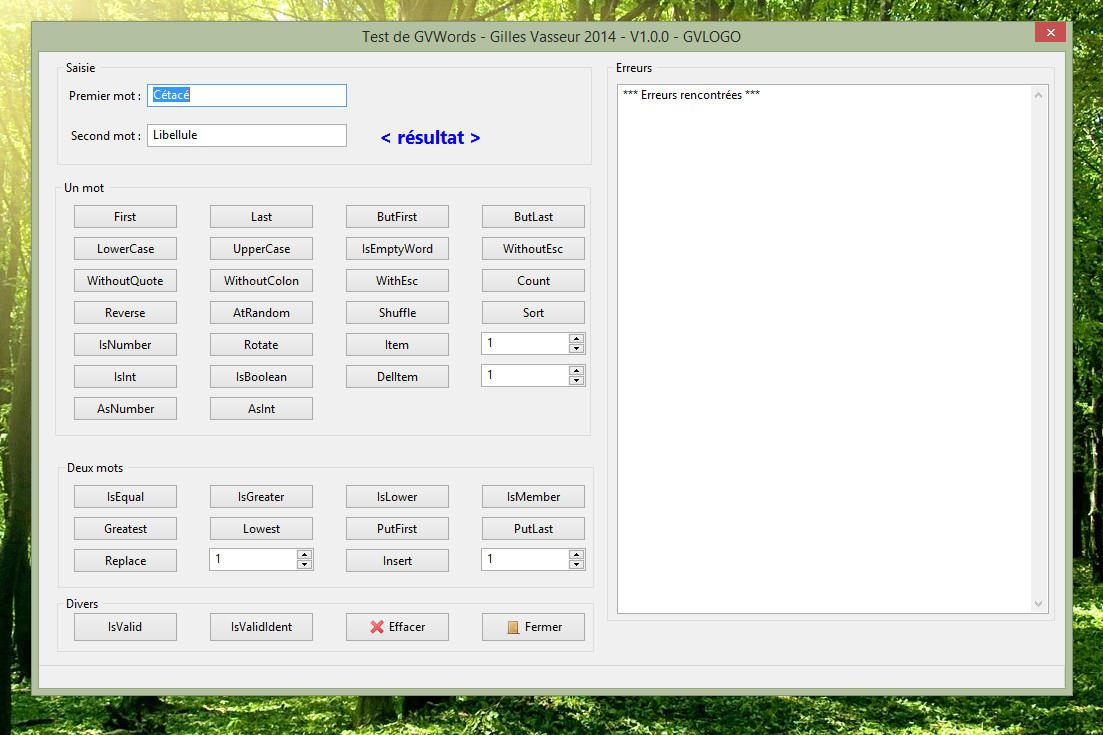


Figure -Test de GVWords

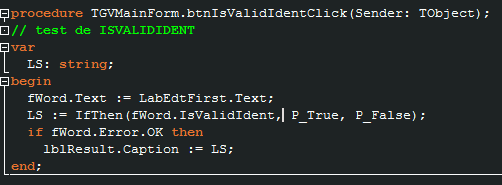
Le programme lui-même est trivial puisqu’il se contente d’affecter le résultat d’une méthode de **TGVWord** à un **TLabel** si aucune erreur n’a été déclenchée. C’est exactement ce qui était recherché : l’unité remplit parfaitement son rôle en limitant au possible les complications pour l’utilisateur final. On notera en particulier que les erreurs sont affichées correctement sans que les méthodes d’affichage aient à s’occuper de leur traitement.

Il est intéressant de modifier les paramètres des éditeurs pour voir le comportement de l’unité. Par exemple, on pourra entrer des mots tels que "Essai, cou[cou… On peut aussi entrer une phrase complète et constater qu’elle sera normalisée pour être comprise comme un mot ! Dans tout test, il est conseillé de provoquer des erreurs afin de voir comment se comporte l’unité.

 Dans ce petit programme d’exemple, une erreur inhibe toutes les fonctions. Il faut cliquer sur le bouton « Effacer » pour réinitialiser l’ensemble.

Une fonction souvent méconnue est utilisée largement dans ce programme : il s’agit de IfThen. La version traitant les chaînes de caractères est définie dans l’unité **strutils**. Elle prend deux ou trois paramètres en entrée : le premier est une valeur booléenne, les deux suivants des chaînes de caractères. Si le troisième est omis, il est par défaut défini à la chaîne vide. Le principe de fonctionnement est semblable à celui d’un test avec if…then…else : si la valeur booléenne est vraie, le second paramètre est renvoyé, sinon c’est le troisième.

Par exemple :



Le test de la fonction IsValidIdent est effectué sur le composant **TLabelEdit** : si le résultat est correct, c’est la constante P\_True qui est affectée au Caption du composant **TLabel**, sinon c’est P\_False.

 Un conseil ("hint") indique que l’unité **strutils** n'est pas utilisée alors que la compilation échouera si elle est omise ! En effet, elle est indispensable pour le traitement de IfTthen avec des chaînes. Il semblerait que Lazarus soit trompé par l’autre fonction IfThen présente dans l'unité **math** utilisée par ailleurs.

## **05 - Les listes**

### **Définitions**

Une liste simple est composée de mots séparés par des espaces[[7]](#footnote-7). Elle est délimitée par des crochets : **[** puis **]**.

Une *liste complexe* comprend des mots et des listes. Les imbrications ne sont pas limitées, pourvu que l’ordre d’ouverture et de fermeture des listes imbriquées soit correct.

Il existe une liste particulière qui ne comprend ni mot ni liste. On la nomme *la liste vide* et elle est représentée par des crochets **[]** accolés.

### **Exemples de listes**

#### **Listes simples**

Voici quelques listes simples :

* [un deux trois quatre cinq]
* [1 2 3 4 5]
* [GVLOGO est un langage facile à apprendre.]

#### **Listes imbriquées**

Voici des listes complexes :

* [élément1 [liste imbriquée 1] élément3]
* [un [deux imbriquée [encore deux mais imbriquée]] trois]
* [[]]
* [un [deux [deux bis [deux ter]]] trois]
* [un (12\*45+789) trois]

La dernière liste contient une expression. Une expression est une suite de caractères à interpréter comme un calcul à effectuer[[8]](#footnote-8).

#### **Listes fautives**

Une liste est fautive si elle ne contient pas des crochets correctement agencés et en bon nombre.

Voici des listes incorrectes :

* [oups ! [où est la fermeture ?]
* [fermée trop tôt][ceci est une autre liste]
* [le caractère d’échappement masque la fin de liste !$]
* [Une parenthèse se promène seule (]

Pour contourner le problème des crochets ou des parenthèses mal appariés, il suffit d’utiliser le caractère d’échappement, comme avec les mots :

* [le caractère d’échappement ne masque plus la fin de liste !$]]
* [Une parenthèse se promène seule sans problème $(]

Le caractère d’échappement disparaîtra à l’écriture grâce aux primitives ECRIS et TAPE. ECRIST le fera apparaître si nécessaire.

### **Opérations sur les listes**

Les opérations possibles sur les listes sont en majorité celles déjà vues à propos des mots. Elles sont mentionnées ici avec des exemples propres aux listes.

La primitive ECRIS retire les crochets lors de l’envoi d’une liste à l’écran, tout comme elle fait disparaître le caractère d’échappement. Ce comportement explique pourquoi les exemples qui utilisent cette primitive donnent des résultats sans crochets. Si l’on veut conserver les crochets, il faut utiliser la primitive ECRIST.

#### **Fabriquer des listes**

* **METS.PREMIER** (raccourci : **MP**) : attend deux listes en entrée ou un mot et une liste – renvoie une liste – la liste rendue est composée du premier paramètre placé avant le second.

Exemples :

ECRIS METS.PREMIER [Coucou] [Tortue !] → [Coucou] Tortue !

ECRIS METS.PREMIER "Coucou [Tortue !] → Coucou Tortue !

* **METS.DERNIER** (raccourci : **MD**) : attend deux listes en entrée ou un mot suivi d’une liste – renvoie une liste – la liste rendue est composée du premier paramètre placé après le second.

Exemples :

ECRIS METS.DERNIER [Coucou] [Tortue !] → Tortue ! [Coucou]

ECRIS METS.DERNIER "Coucou [Tortue !] → Tortue ! Coucou

* **PHRASE** (raccourci : **PH**): attend deux listes ou deux mots ou une combinaison des deux en entrée – renvoie une liste – la liste rendue est composée des deux éléments séparés par un espace sans les crochets et/ou sans les guillemets anglais.

Exemples :

ECRIS PHRASE "je "viens → je viens

ECRIS PHRASE [je] [pars] → je pars

ECRIS PHRASE "je [reviens] → je reviens

ECRIS PHRASE [je] "repars → je repars

#### **Modifier des listes**

* **INSERE**: attend un entier, un mot ou une liste, puis une liste en entrée – renvoie une liste – la liste rendue est composée du deuxième paramètre inséré à la position précisée par l’entier dans la liste finale.

Exemples :

ECRIS INSERE 2 "très [c’est bien] → c’est très bien

ECRIS INSERE 1 [zéro] [[un][deux][trois]] → [zéro][un][deux][trois]

* **INVERSE**: attend une liste en entrée – renvoie une liste – la liste rendue est celle d'entrée dont les éléments ont été inversés.

Exemple :

ECRIS INVERSE [un deux trois] → trois deux un

* **MAJUSCULES**: attend une liste en entrée – renvoie une liste – la liste rendue est en majuscules.

Exemple :

ECRIS MAJUCULES [le soleil se lève] → LE SOLEIL SE LÈVE

* **MELANGE**: attend une liste en entrée – renvoie une liste – la liste rendue a les mêmes éléments que celle d'origine, mais dans un ordre aléatoire.

Exemple :

ECRIS MELANGE [un deux [troisa troisb]] → deux [troisa troisb] un

* **MINUSCULES**: attend une liste en entrée – renvoie une liste – la liste rendue est en minuscules.

Exemple :

ECRIS MINUSCULES [LES ENFANTS SE LÈVENT] → Les enfants se lèvent

* **REMPLACE**: attend un entier, une liste ou un mot, puis une liste en entrée – renvoie une liste – la liste rendue est composée de toutes les éléments de la liste finale, sauf celui visé par l'entier qui a été remplacé par le deuxième paramètre.

Exemples :

ECRIS REMPLACE 4 "couchent [Les enfants se lèvent] → Les enfants se couchent

ECRIS REMPLACE 2 [deuxième] [[premier][seconde][troisième]] → [premier][deuxième][troisième]

* **TRIE**: attend une liste en entrée – renvoie une liste – les éléments de la liste sont triés par ordre alphabétique[[9]](#footnote-9).

Exemple :

ECRIS TRIE [zèbre tartine accent dommage] → accent dommage tartine zèbre

* **ROTATION** : attend une liste en entrée – renvoie une liste – la primitive renvoie la liste dont le premier élément est placé à la fin.

Exemple :

ECRIS ROTATION [un deux trois quatre] → deux trois quatre un

#### **Extraire de listes**

* **ELEMENT**: attend un entier suivi d'une liste en entrée – renvoie une liste ou un mot – l’objet rendu est l’élément de la liste en entrée indiqué par l'entier.

Exemple :

ECRIS ELEMENT 2 [un deux [trois]] → deux

* **DERNIER** (raccourci : **DER**): attend une liste en entrée – renvoie un mot ou une liste – l’objet renvoyé est composé du dernier élément de la liste en entrée.

Exemple :

ECRIS DERNIER [un deux] → deux

* **HASARD**: attend une liste en entrée – renvoie un mot ou une liste – l’objet renvoyé est composé d’un élément de la liste en entrée, tiré au hasard.

Exemple :

ECRIS HASARD [un [deux] trois quatre] → trois

* **PREMIER** (raccourci : **PREM**): attend une liste en entrée – renvoie un mot ou une liste – l’objet renvoyé est composé du premier élément de la liste en entrée.

Exemple :

ECRIS PREMIER [beau laid] → beau

* **PREMS**: attend une liste en entrée – renvoie une liste – l’objet renvoyé est composé de chaque premier élément des éléments de la liste en entrée.

Exemples :

ECRIS PREMS [beau [laid]] → b laid

ECRIS PREMS [[moi beau] [toi laid]] → [moi] [toi]

* **SAUF.DERNIER** (raccourci : **SD**): attend une liste en entrée – renvoie un mot ou une liste – l’objet renvoyé est composé de la liste en entrée sans son dernier élément.

Exemple :

ECRIS SAUF.DERNIER [pas cette liste stupide] → pas cette liste

* **SAUF.PREMIER** (raccourci : **SP**): attend une liste en entrée – renvoie un mot ou une liste – l’objet renvoyé est composé de la liste en entrée sans son premier élément.

Exemple :

ECRIS SAUF.PREMIER [pas cette liste stupide] → cette liste stupide

* **SAUF.PREMS**: attend une liste en entrée – renvoie une liste – l’objet renvoyé est composé de la liste en entrée sans chaque premier élément de ses éléments.

Exemples :

ECRIS SAUF.PREMS [pas cette liste stupide] → as ette iste tupide

ECRIS SAUF.PREMS [[un chat] [une chouette]] → [chat] [chouette]

#### **Tester des listes**

* **PLG?** : attend deux listes en entrée – renvoie un booléen – la primitive renvoie "VRAI si la première liste vient strictement après la seconde selon l'ordre alphabétique, "FAUX sinon.

Exemple :

ECRIS PLG? [un petit chat] [un chat] → -1

* **PLP?** : attend deux listes en entrée – renvoie un booléen – la primitive renvoie "VRAI si la première liste vient strictement avant la seconde selon l'ordre alphabétique, "FAUX sinon.

Exemple :

ECRIS PLP? [le renard] [un renard] → -1

* **COMPTE**: attend une liste en entrée – renvoie un entier – la primitive renvoie le nombre d’éléments de la liste en entrée.

Exemple :

ECRIS COMPTE [un [deux encore deux] trois] → 3

* **EGAL?** : attend deux listes en entrée – renvoie un booléen – la primitive renvoie "VRAI si la première liste est identique à la seconde, "FAUX sinon.

Exemple :

ECRIS EGAL? [c’est la même] METSPREMIER "c’est [la même] → -1

* **MEMBRE?** : attend une liste ou un mot, puis une liste en entrée – renvoie un booléen – la primitive renvoie "VRAI si le premier objet est compris dans le second, "FAUX sinon.

Exemples :

ECRIS MEMBRE? [oui] [un deux [oui]] → -1

ECRIS MEMBRE? "oui [un deux [oui]] → 0

* **LISTE?** : attend un objet en entrée – renvoie un booléen – la primitive renvoie "VRAI si l'objet est une liste, "FAUX sinon.

Exemples :

ECRIS LISTE? [coucou] → -1

ECRIS LISTE? "coucou → 0

* **PRECEDENT** : attend deux listes en entrée – renvoie une liste – la primitive renvoie la liste qui vient la première selon l'ordre alphabétique.

Exemple :

ECRIS PRECEDENT [je suis premier] [je suis second] → je suis premier

* **SUIVANT** : attend deux listes en entrée – renvoie une liste – la primitive renvoie la liste qui vient la dernière selon l'ordre alphabétique.

Exemple :

ECRIS SUIVANT [je suis premier] [je suis second] → je suis second

* **VIDE?** : attend une liste en entrée – renvoie un booléen – la primitive renvoie "VRAI si la liste en entrée est la liste vide, "FAUX sinon.

Exemples :

ECRIS VIDE? [] → -1

ECRIS VIDE? SAUF.PREMIER [un] → -1

### **Implémentation des listes**

L’implémentation des listes passe par un complément de l’unité **GVConsts** qui contient les constantes centralisées du projet et une unité baptisée **GVLists** qui abrite les classes nécessaires à la gestion proprement dite des listes. Cette unité utilise par ailleurs l’unité **GVWords** précédemment étudiée.

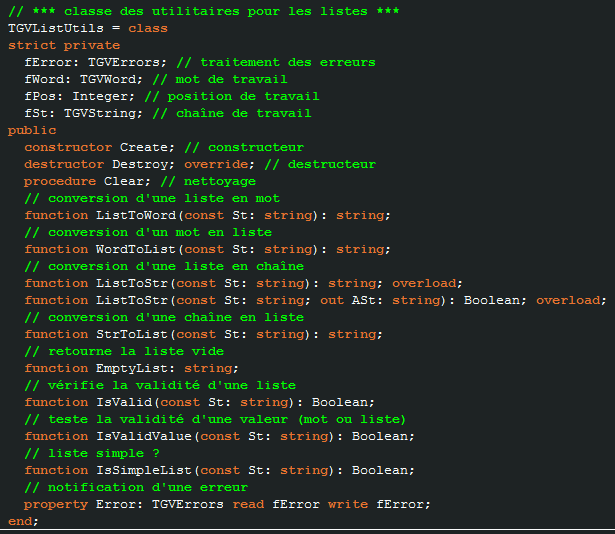
#### **Constantes**

Les constantes ont simplement été complétées avec les chaînes nécessaires au signalement des erreurs et un type énuméré baptisé **TGVError** qui reprend les erreurs sous forme codée.

#### **La classe TGVListUtils**

La classe **TGVListUtils** concentre les méthodes utiles au traitement des listes. Il aurait été possible de les implémenter en tant que procédures et fonctions indépendantes, mais l’utilisation d’une classe et de son interface permet une meilleure maîtrise des outils : la classe définit précisément ce pour quoi elle est faite.

En voici l’interface :



En dehors de la fonction EmptyList qui se contente de renvoyer la liste vide, il s’agit essentiellement de méthodes de conversion et de validation.

Ainsi ListToStr et StrToList convertissent une liste en chaîne et réciproquement. ListToWord et WordToList prennent en compte le caractère d’échappement. Pour ces quatre méthodes, un contrôle de la validité des listes est effectué.

Le contrôle de la validité d’une liste s’effectue grâce à IsValid qui est de loin la plus complexe. Elle analyse une chaîne et contrôle caractère par caractère que les règles relatives aux listes sont respectées : autant de crochets ouvrants que de crochets fermants, dans le bon ordre ; autant de parenthèses ouvrantes que de parenthèses fermantes, dans le bon ordre ; prise en compte du caractère d’échappement ; interdiction des sous-listes à l’intérieur d’une expression parenthésée.

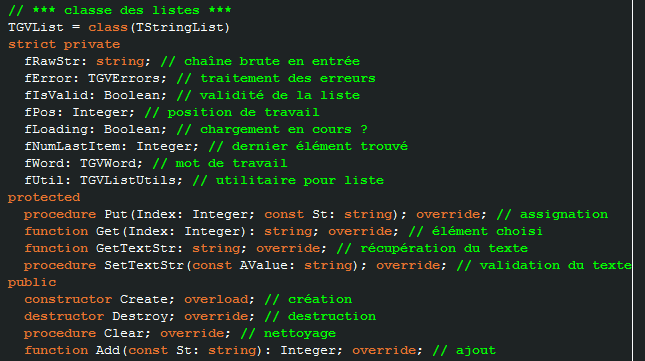
Les autres méthodes de contrôle sont plus simples. IsSimpleValue se contente de vérifier la présence des crochets pour une liste. IsValidValue exploite l’unité **GVWords** pour tester un mot et la méthode précédente pour une liste. TestValue est une procédure qui déclenche une exception si la chaîne en entrée n’est ni un mot ni une liste au sens de **GVLOGO**.

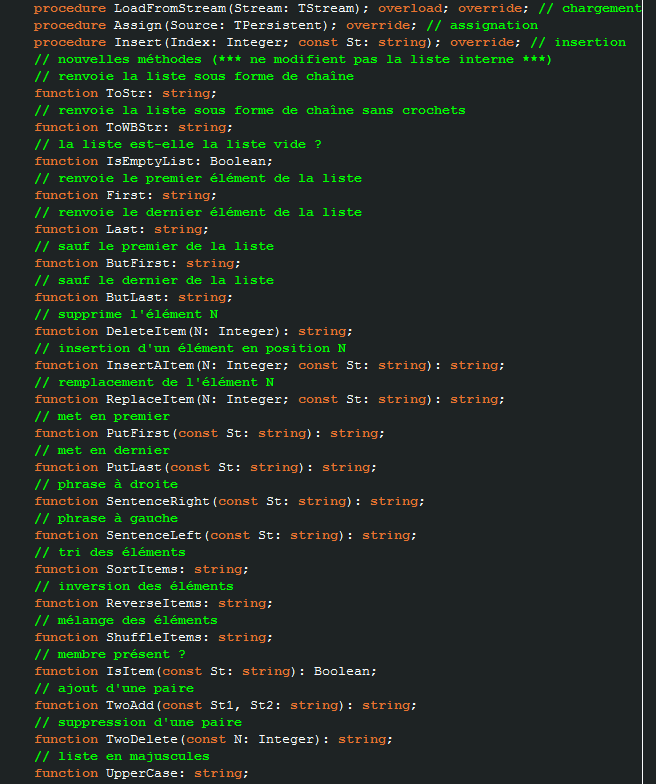
#### **La classe TGVList**

La classe **TGVList** implémente la gestion des listes pour **GVLOGO**. Elle hérite de **TStringList** qui gère les listes de chaînes. En fait, une liste en LOGO est une chaîne particulière de caractères : elle nécessite d’être entourée de crochets et, si elle comprend d’autres listes, de comprendre à l’intérieur autant de crochets ouvrants que de crochets fermants (dans le bon ordre évidemment !). Cette dernière remarque s’applique aussi pour les éventuelles expressions imbriquées dans la liste. Pour parvenir à cet effet, il faut surcharger les méthodes d’entrée : Add, LoadFromStream, Assign, Insert et Put. Chaque élément de la liste (mot ou sous-liste) est stocké dans une chaîne de la **TStringList**.

Toutes les nouvelles méthodes renvoient une chaîne ou une valeur sans modifier la liste d’entrée. En cas d’erreur lors de la construction de la liste, la liste interne est la chaîne vide et la propriété LastErrorPos renvoie l’indice de l’erreur détectée.

En voici l’interface[[10]](#footnote-10) :







La méthode la plus complexe est certainement Add. Elle prend en charge l’ensemble du contrôle de la validité de la liste. Cette méthode comprend une partie générale qui teste les mots et les listes avant de renvoyer une éventuelle position d’erreur ou de stocker les chaînes trouvées, et une procédure imbriquée qui examine le contenu de la chaîne. Cette dernière lit les caractères un par un et répartit le travail suivant ce qu’elle rencontre : blanc, crochet ouvrant, parenthèse ouvrante, échappement, crochet et parenthèse fermants mais orphelins, autre caractère.

Elle est en cela similaire à la méthode IsValid de **TGVListUtils**, sinon qu’elle stocke les éléments en vue de leur exploitation. Du point de vue de l’utilisateur, l’entrée d’une nouvelle liste se fait grâce à la propriété héritée Text de l’unité : il suffit d’affecter une chaîne à cette propriété pour qu’elle soit analysée et stockée.

La propriété LastItem n’est affectée que par l’utilisation de la méthode IsItem. Si cette dernière renvoie True, LastItem contiendra le numéro de l’élément de la liste qui correspond à celui recherché.

On notera que les éléments des listes en Pascal sont numérotés avec une base de 0 alors qu’en **GVLOGO** la base est de 1, ce qui paraîtra plus naturel à l’utilisateur.

#### **Test de l’unité TGVLists**

Comme lors du test de **TGVWords**, les versions proposées sont au nombre de deux :

* une version Win32 ;
* une version Linux.

Le programme lui-même ne pose pas de problèmes particuliers. Il ne s’agit en effet que de mettre à jour la liste de travail et d’appliquer la méthode voulue.

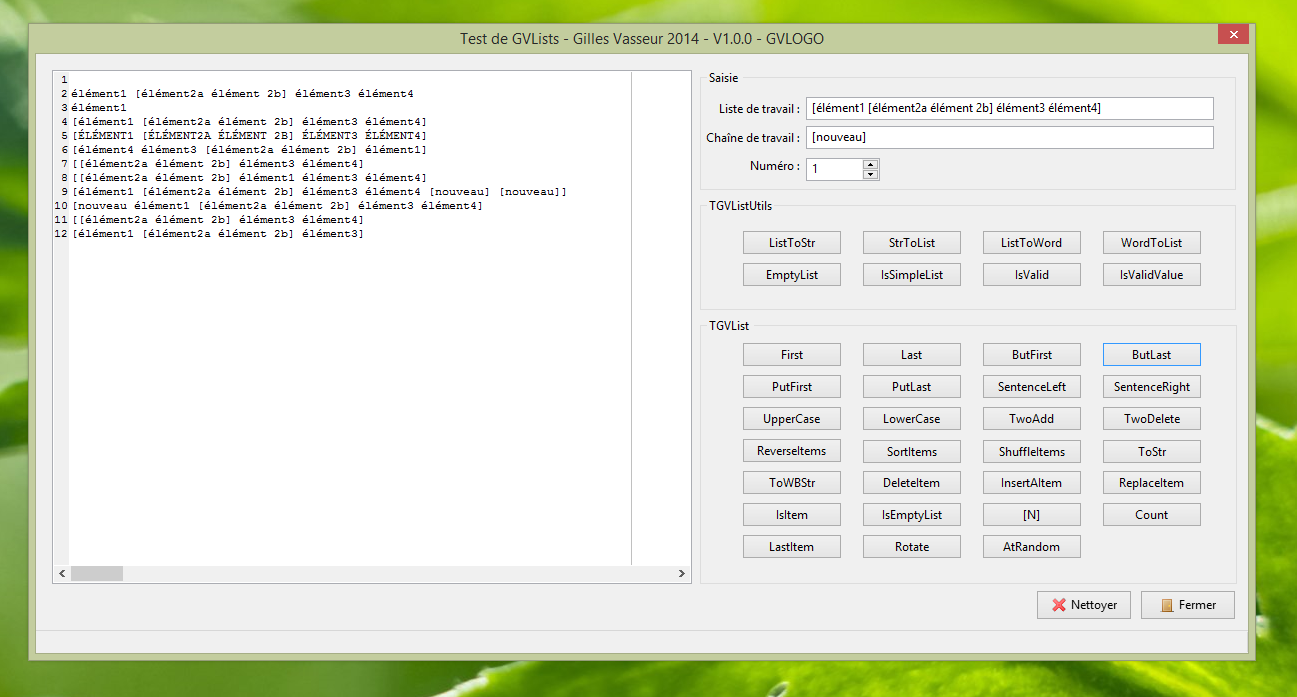


Figure - Test de GVLists

Il est encore une fois conseillé de modifier les différents paramètres pour observer les réactions de l’unité.

## **06 - Les listes de propriétés**

### **Définitions**

Les listes de propriétés sont des listes particulières associant chacune à un nom des couples propriété-valeur.

Par exemple, on peut imaginer répertorier des chiens dont les caractéristiques seront stockées ainsi : Zoé aura pour propriétés race, sexe, couleur, taille et âge dont les valeurs respectives seront caniche, femelle, blanc, très petit, 5. Médor aura les propriétés race, sexe, âge, caractère dont les valeurs seront teckel, mâle, 3, tranquille. On voit que l‘agencement des propriétés est libre, contrairement à ce qui se passerait pour un tableau.

**GVLOGO** fait un usage interne intensif des listes de propriétés. En effet, aussi bien les variables, les procédures, les paquets que les lignes des éditeurs sont des listes de propriétés.

Les noms donnés aux propriétés tout comme le nom des listes est indifférent aux majuscules et minuscules. Par conséquent, les propriétés MAPROP, MaProp ou encore maprop sont équivalentes. En revanche, ce nom doit être un identificateur correct.

### **Exemples de listes de propriétés**

* Liste associée à "caniche" : [couleur blanc taille [tout petit]]
* Liste associée à "labrador" : [couleur blanc vitesse [rapide] ans 8]

Les propriétés d’une liste de propriétés sont organisées en deux éléments consécutifs : le premier définit le nom de la propriété tandis que le second indique la valeur qui lui est associée. Le premier est toujours un mot unique pour la liste considérée alors que le second peut être un mot ou une liste quelconque.

### **Opérations sur les listes de propriétés**

* **DPROP**: attend deux mots en entrée suivis d’un mot ou une liste – ne renvoie rien – la primitive crée ou met à jour la propriété de la liste de propriétés fournie en paramètre. Le premier mot est la liste de propriétés et le second la propriété. Le dernier paramètre est soit un mot soit une liste qui définit la valeur attribuée à la propriété.

Exemple :

DPROP "MALISTE "MAPROP [une valeur] → -

* **RPROP**: attend deux mots en entrée – renvoie un mot ou une liste – la primitive renvoie la valeur de la propriété (second paramètre) de la liste visée (premier paramètre).

Exemple :

ECRIS RPROP "MALISTE "MAPROP → une valeur

* **ANPROP**: attend deux mots en entrée – ne renvoie rien – la propriété (second paramètre) de la liste spécifiée (premier paramètre) est supprimée.

Exemple :

ANPROP "MALISTE "MAPROP → -

* **PROPS**: attend un mot en entrée – renvoie une liste – la primitive renvoie la liste des propriétés de la liste en entrée.

Exemple :

ECRIS PROPS "MALISTE → maprop

* **ANNULE**: attend un mot en entrée – ne renvoie rien – la liste spécifiée par le mot est détruite avec toutes ses propriétés.

Exemple :

ANNULE "MALISTE → -

* **COMPTE.PROPS**: attend un mot en entrée – renvoie un entier – la primitive renvoie le nombre de propriétés associées à la liste en entrée.

Exemple :

COMPTEPROPS "MALISTE → 1

* **PROP?**: attend deux mots en entrée – renvoie un booléen – la primitive renvoie "VRAI si le second mot en entrée est une propriété du premier, "FAUX sinon

Exemple :

ECRIS PROP? "MALISTE "MAPROP → -1

* **LISTE.PROP?**: attend un mot en entrée – renvoie un booléen – la primitive renvoie "VRAI si le mot en entrée est une liste de propriétés, "FAUX sinon.

Exemple :

ECRIS LISTEPROP? "MALISTE → -1

* **PROCEDURE?**: attend un mot en entrée – renvoie un booléen – la primitive renvoie "VRAI si le mot en entrée est une procédure, "FAUX sinon.

Exemple :

ECRIS PROCEDURE? "MALISTE → 0

* **PRIMITIVE?**: attend un mot en entrée – renvoie un booléen – la primitive renvoie "VRAI si le mot en entrée est une primitive, "FAUX sinon.

Exemple :

ECRIS PRIMITIVE? "ECRIS → 0

On notera que la primitive (ou la procédure) est précédée des guillemets anglais. Si elle ne l’était pas, **GVLOGO** chercherait à l’exécuter, ce qui n’est pas l’effet voulu.

* **NOM?**: attend un mot en entrée – renvoie un booléen – la primitive renvoie "VRAI si le mot en entrée est une variable, "FAUX sinon.

Exemple :

ECRIS NOM? "MALISTE → 0

### **Implémentation des listes de propriétés**

L’unité **GVPropLists** implémente essentiellement deux classes : **TGVPropListEnumerator** qui gère l’énumération et **TGVPropList** qui gère les listes elles-mêmes.

#### **Constantes**

Comme pour les unités précédentes, l’unité **GVConsts** a été complétée afin de prendre en compte les nouveaux besoins. Les ajouts ont été effectués au niveau des constantes elles-mêmes.

On remarquera l’extension pour les fichiers et surtout l’en-tête qui permettra de vérifier qu’un fichier est conforme au format attendu : il précèdera les données proprement dites.

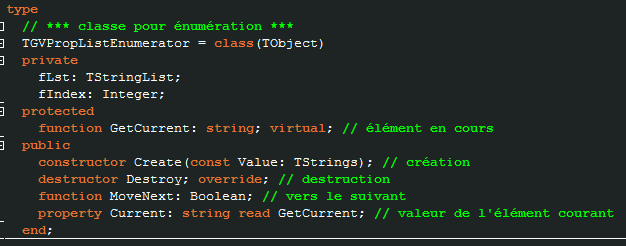
Les chaînes de ressources ont aussi été complétées, de même pour l’énumération des erreurs possibles.

#### **La classe TGVPROPLISTENUMERATOR**

Comme son nom l’indique, l’énumération est un moyen de dresser la liste des éléments contenus dans un objet. Aussi bien Lazarus que Delphi en font un usage étendu à travers les objets tels que les listes.

La technique d’implémentation, finalement assez simple, obéit à un schéma type : on doit fournir l’élément en cours (pointé par une propriété en lecture seule) et être capable d’indiquer si l’on peut se déplacer vers l’élément suivant.

Cela donne dans le cas des listes de propriétés l’interface suivante :



Comme les listes de propriétés fonctionnent *via* une liste interne, il suffit d’en manipuler les éléments grâce à un index.

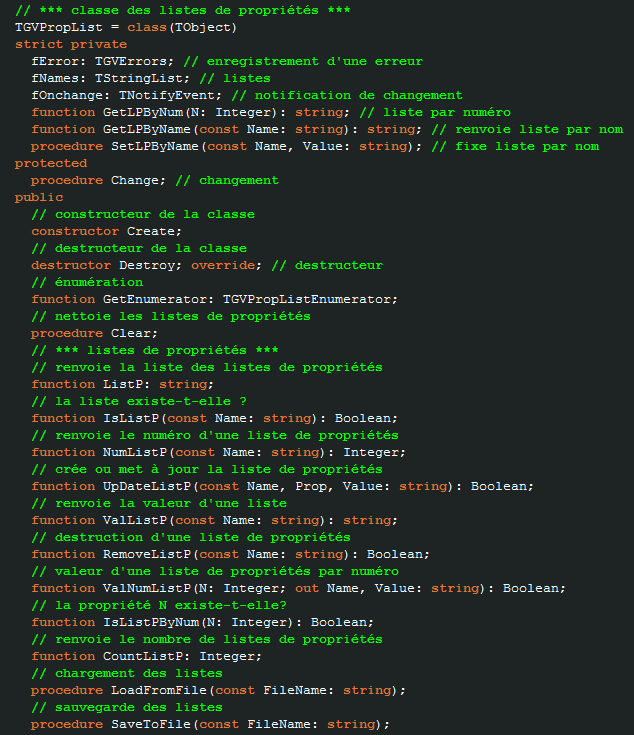
 La gestion se fait au moyen du système intégré dans **TStringList** des paires Names/Values, mais utilise un caractère séparateur personnalisé : **|[[11]](#footnote-11)** . Ce caractère ne doit par conséquent pas figurer dans la liste elle-même.

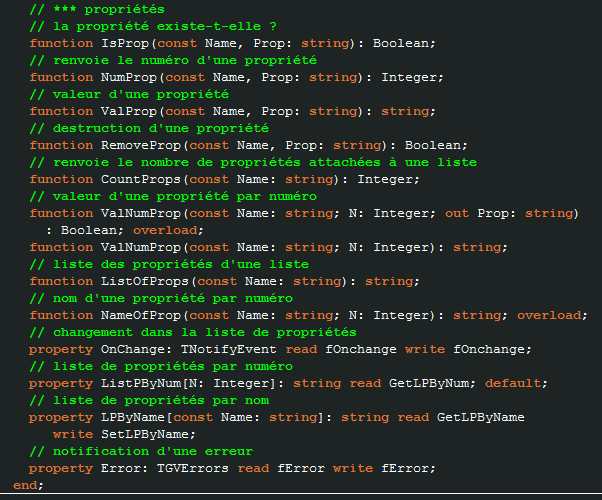
Les utilisateurs de Delphi pourront imaginer une unité fondée sur la classe **TDictionary**.

#### **La classe TGVPROPLIST**

La classe **TGVPropList** fonctionne grâce à un champ privé fNames de type TStringList. Les méthodes sont organisées en trois blocs : méthodes générales, celles concernant les listes de propriétés et celles concernant les propriétés.

En voici l’interface :





On remarquera que la propriété ListPByNum est définie comme default, ce qui signifie qu’elle peut être omise en employant simplement le numéro de la liste entre crochets.

Parmi les difficultés, on relèvera :

* Les listes de propriétés sont numérotées à partir de 1 et non de 0 comme en Pascal ;
* La méthode RemoveProp détruit aussi la liste si elle ne contient plus aucune propriété ;
* La méthode UpdateListP crée et met à jour les listes et les propriétés ;
* Il est fait un usage intensif de la fonction Odd qui vérifie qu’un nombre est impair : en effet, comme les propriétés sont organisées par paires, il est nécessaire de bien différencier le nom de la propriété (élément impair) de sa valeur (élément pair).

#### **Test de l’unité TGVPROPLists**

Comme d’habitude, le programme de test est décliné en deux versions :

* Lazarus pour Windows 32 ;
* Lazarus pour Linux.

Un fichier de listes préenregistrées permet de tester immédiatement LoadFromFile : « ListPs.GPL ». Il peut être édité par n’importe quel lecteur de textes simples (genre NotePad).



Figure - Test des listes de propriétés

## **07 - La tortue graphique**

### **Présentation**

La tortue graphique est sans doute l’objet le plus connu du langage LOGO. Elle a fait son succès, mais l’a aussi très longtemps limité à une utilisation avec des enfants, conduisant les utilisateurs à négliger les listes qui permettent d’aborder des sujets plus ardus.

La tortue graphique est habituellement symbolisée par un triangle sur une surface de dessin. **GVLOGO** propose par ailleurs une tortue plus réaliste au format « png ». Des ordres permettent de la faire dessiner. Les commandes disponibles sont multiples : l’utilisateur maîtrise son orientation, son trait, sa forme, sa couleur…

L’originalité de ce mode de dessin réside dans sa capacité à partir d’une orientation dans l’espace déterminée par la tortue elle-même : même s’il est possible de dessiner dans un repère orthonormé traditionnel, la plupart des commandes sont dépendantes de l’état en cours de la tortue. Ainsi, tourner de 90° à gauche se fera par rapport à l’orientation actuelle de la tortue : si son cap était de 45° par rapport à la surface de dessin, son nouveau cap sera de 135°.

Figure 8 - Tortue avec cap 90 Tortue 180 Tortue 90 png

Par défaut, la tortue est au centre de la surface de dessin qui se mesure en pixels et dépend de la taille du composant **TPanel** d’accueil : sa position d’origine est donc le centre de ce composant. Ses déplacements se comptent en pixels si les échelles des abscisses et des ordonnées sont fixées à 100 (valeur par défaut). Un déplacement négatif fait reculer la tortue.

Le cap de la tortue indique la direction des déplacements. Il est exprimé en degrés et va de 0 à 360. Le cap est de 90° par défaut et pointe vers le sommet de l’écran afin d’être en conformité avec le cercle trigonométrique. On visualise ce cap grâce à l’orientation du triangle qui symbolise la tortue : l’arrière de la tortue est un trait plus épais ; sa tête est donc le sommet du triangle qui marque l’intersection des deux lignes plus fines.

### **Opérations avec la tortue**

#### **Champ de la tortue**

Le champ de la tortue est la surface sur laquelle elle peut évoluer. Le coin inférieur gauche de l’écran est la position d’abscisse 0 et d’ordonnée 0. Par défaut, le point supérieur droit a pour abscisse la largeur du composant d’accueil et pour ordonnée sa hauteur[[12]](#footnote-12).

* **CLOS**: n’attend rien en entrée – ne renvoie rien – le champ de la tortue est limité à l’écran visible. Toute tentative de sortir de l’écran se soldera par un échec.

Exemple :

CLOS → -

Le champ est défini à FENETRE par défaut.

* **ENROULE**: n’attend rien en entrée – ne renvoie rien – le champ de la tortue s’enroule sur lui-même : quand la tortue atteint un bord, elle réapparaît sur le bord opposé.

Exemple :

ENROULE → -

* **FENETRE (raccourci : FEN)**: n’attend rien en entrée – ne renvoie rien – le champ de la tortue s’étend au-delà de l’écran. La tortue peut donc disparaître du champ de vision de l’utilisateur.

Exemple :

FENETRE → -

* **ETAT.ECRAN**: n’attend rien en entrée – renvoie une liste – la liste renvoyée indique l’état actuel de l’écran avec les données suivantes : couleur, hauteur et largeur.

Exemple :

ECRIS ETAT.ECRAN → 0 700 677

* **FIXE.ECHELLE**: attend une liste de deux entiers en entrée – ne renvoie rien – la primitive détermine les proportions du déplacement de la tortue. Par défaut, la valeur pour les abscisses (premier entier) et pour les ordonnées (second paramètre) est de 100. Une valeur moindre réduira les déplacements dans la proportion indiquée et une valeur supérieure l’augmentera. Par exemple, fixer le premier entier à 200 et le second à 50 fera que la tortue se déplacera deux fois plus vite horizontalement que la normale et deux fois plus lentement verticalement.

Exemple :

FIXE.ECHELLE [200 50] → -

* **FIXE.ECHELLEX**: attend un entier en entrée – ne renvoie rien – la primitive détermine la proportion du déplacement de la tortue selon l’axe des abscisses.

Exemple :

FIXE.CHELLEX 200 → -

* **FIXE.ECHELLEY**: attend un entier en entrée – ne renvoie rien – la primitive détermine la proportion du déplacement de la tortue selon l’axe des ordonnées.

Exemple :

FIXE.ECHELLEY 50 → -

* **ECHELLE**: n’attend rien en entrée – renvoie une liste de deux entiers – la liste renvoyée contient un premier entier qui indique l’échelle des déplacements de la tortue selon l’axe des abscisses et un second entier qui indique l’échelle selon l’axe des ordonnées.

Exemple :

ECRIS ECHELLE → 100 100

* **VIDE.ECRAN (raccourci : VE) :** n’attend rien en entrée – ne renvoie rien – l’espace de la tortue est nettoyé et la tortue retrouve sa place et son cap d’origine, ainsi que toutes ses valeurs par défaut.

Exemple :

VIDE.ECRAN → -

* **ORIGINE :** n’attend rien en entrée – ne renvoie rien – la tortue retrouve sa position d’origine en [300 300] et son cap initial de 90. L’état du crayon n’est pas modifié. Si le crayon est baissé, le déplacement de la tortue laisse une trace.

Exemple :

ORIGINE → -

* **NETTOIE :** n’attend rien en entrée – ne renvoie rien – l’espace de la tortue est nettoyé, sans changer la position de la tortue ni son cap, pas plus que les valeurs du crayon.

Exemple :

NETTOIE → -

* **FIXE.COULEUR.FOND (raccourci : FCF)**: attend un entier en entrée – ne renvoie rien – la primitive détermine la couleur de fond de l’écran de la tortue.

Exemple :

FIXE.COULEUR.FOND 19 → -

Tableau 1 - Couleurs standards de la tortue

|  |  |  |
| --- | --- | --- |
| Noir | 0 |  |
| Bleu ciel | 1 |  |
| Bleu | 2 |  |
| Crème | 3 |  |
| Gris foncé | 4 |  |
| Fuchsia | 5 |  |
| Gris | 6 |  |
| Vert | 7 |  |
| Vert citron | 8 |  |
| Gris clair | 9 |  |
| Marron | 10 |  |
| Gris moyen | 11 |  |
| Vert menthe | 12 |  |
| Bleu marine | 13 |  |
| Vert olive | 14 |  |
| Violet | 15 |  |
| Rouge | 16 |  |
| Argent | 17 |  |
| Bleu ciel | 18 |  |
| Sarcelle | 19 |  |
| Blanc | 20 |  |
| Jaune | 21 |  |

* **COULEUR.FOND (raccourci : CF)**: n’attend rien en entrée – renvoie un entier – l’entier renvoyé correspond à la couleur du fond de l’écran de la tortue.

Exemple :

ECRIS COULEUR.FOND → 19

* **DISTANCE**: attend deux entiers en entrée – renvoie un entier – l’entier renvoyé correspond à la distance de la tortue au point de coordonnées fournies en entrée (abscisse puis ordonnée).

Exemple :

ECRIS DISTANCE 50 50 → 190

#### **Déplacements de la tortue**

* **AVANCE** (raccourci : **AV**): attend un entier en entrée – ne renvoie rien – la tortue avance du nombre de pixels indiqués par le paramètre en entrée, sauf si les échelles ont été changées. Elle laisse une trace sur la zone de dessin à moins que son crayon ne soit levé ou qu’elle soit en mode gomme.

Exemple :

AVANCE 45 → -

* **RECULE** (raccourci : **RE**): attend un entier en entrée – ne renvoie rien – la tortue recule du nombre de pixels indiqués par le paramètre en entrée, sauf si les échelles ont été changées. Elle laisse une trace sur la zone de dessin à moins que son crayon ne soit levé ou qu’elle soit en mode gomme.

Exemple :

RECULE 50 → -

* **GAUCHE** (raccourci : **TG**): attend un réel en entrée – ne renvoie rien – la tortue tourne à gauche du nombre de degrés indiqué en entrée.

Exemple :

GAUCHE 90 → -

Pour rappel : la gauche indiquée est celle par rapport à la tortue et non la gauche de l’écran ! Cette remarque vaut évidemment pour la primitive DROITE.

* **DROITE** (raccourci : **TD**): attend un réel en entrée – ne renvoie rien – la tortue tourne à droite du nombre de degrés indiqué en entrée.

Exemple :

DROITE 90 → -

* **FIXE.POS** (raccourci : **FPOS**): attend une liste de deux entiers en entrée – ne renvoie rien – la tortue est envoyée à la position indiquée par la liste : le premier élément fixe l’abscisse tandis que le second détermine l’ordonnée.

Exemple :

FIXE.POS [200 100] → -

* **FIXE.XY** (raccourci : **FXY**): attend deux entiers en entrée – ne renvoie rien – la tortue est envoyée à la position indiquée par les paramètres : le premier élément fixe l’abscisse tandis que le second détermine l’ordonnée.

Exemple :

FIXE.XY 200 100 → -

* **FIXE.X** (raccourci : **FX**): attend un entier en entrée – ne renvoie rien – l’abscisse de la tortue est fixée à l’entier fourni en entrée.

Exemple :

FIXE.X 200 → -

* **FIXE.Y** (raccourci : **FY**): attend un entier en entrée – ne renvoie rien – l’ordonnée de la tortue est fixée à l’entier fourni en entrée.

Exemple :

FIXE.Y 100 → -

* **POS**: n’attend rien en entrée – renvoie une liste de deux entiers – la liste renvoyée contient un premier entier qui indique l’abscisse de la tortue et un second entier qui indique son ordonnée.

Exemple :

ECRIS POS → 200 100

* **XCOOR**: n’attend rien en entrée – renvoie un entier – l’entier renvoyé indique l’abscisse de la tortue.

Exemple :

ECRIS XCOOR → 200

* **YCOOR**: n’attend rien en entrée – renvoie un entier – l’entier renvoyé indique l’ordonnée de la tortue.

Exemple :

ECRIS YCOOR → 100

* **FIXE.VITESSE**: attend un entier en entrée – ne renvoie rien – la vitesse de la tortue est fixée à l’entier fourni en entrée.

Exemple :

FIXEVITESSE 100 → -

La vitesse est relative aux performances de l’ordinateur sur lequel est installé **GVLOGO**. Elle est automatiquement limitée à 100.

* **VITESSE**: n’attend rien en entrée – renvoie un entier – l’entier renvoyé indique la vitesse de dessin de la tortue.

Exemple :

ECRIS VITESSE → 100

#### **Cap de la tortue**

* **FIXE.CAP** (raccourci : **FCAP**): attend un réel en entrée – ne renvoie rien – le cap de la tortue est fixé au nombre fourni en entrée. Le cap est compris entre 0 et 360.

Exemple :

FIXECAP 90 → -

* **CAP**: n’attend rien en entrée – renvoie un entier – l’entier renvoyé indique le cap de la tortue.

Exemple :

ECRIS CAP → 90

* **VERS :** attend une liste de deux entiers en entrée – renvoie un entier – L’entier renvoyé indique le cap que devrait avoir la tortue pour pointer vers le point dont les coordonnées sont fournies en entrée.

Exemple :

VERS [0 0] → 214

Pour rappel : le cap 90 est celui qui pointe vers le haut de l’écran. Le cap 0 pointe vers sa droite. En fait, le cap augmente en tournant dans le sens inverse des aiguilles d’une montre.

#### **Etat de la tortue**

* **MONTRE.TORTUE (raccourci : MT)**: n’attend rien en entrée – ne renvoie rien – la tortue devient visible.

Exemple :

MONTRE.TORTUE → -

* **CACHE.TORTUE (raccourci : CT)**: n’attend rien en entrée – ne renvoie rien – la tortue devient invisible.

Exemple :

CACHE.TORTUE → -

La tortue dessine plus rapidement lorsqu’elle est invisible.

* **VISIBLE?** : n’attend rien en entrée – renvoie un booléen – la primitive renvoie "VRAI si la tortue est visible, "FAUX sinon.

Exemple :

ECRIS VISIBLE? → -1

* **ETAT.TORTUE**: n’attend rien en entrée – renvoie une liste – la liste renvoyée fournit les données relatives à la tortue : son abscisse, son ordonnée, son cap, sa taille, sa vitesse, sa visibilité et si elle est figurée par un triangle.

Exemple :

ECRIS ETAT.TORTUE → 300 300 98 8 100 -1 0

* **FIXEETATTORTUE**: attend une liste en entrée – ne renvoie rien – l’état de la tortue est fixé selon le contenu de la liste : abscisse, ordonnée, orientation, taille, vitesse, visibilité et type de tortue. La tortue dessine durant le changement si le crayon est baissé.

Exemple :

FIXE.ETAT.TORTUE [100 100 0 8 100 -1 -1] → -

* **TORTUE.NORMALE**: n’attend rien en entrée – ne renvoie rien – quand elle est visible, la tortue est symbolisée par un triangle. C’est le mode par défaut.

Exemple :

TORTUE.NORMALE → -

* **TORTUE.VERTE**: n’attend rien en entrée – ne renvoie rien – quand elle est visible, la tortue est dessinée à partir d’images au format PNG.

Exemple :

TORTUE.VERTE → -

 L’image est plus sympathique, mais le dessin est un peu plus lent et la précision pour les angles moindre (5°).

* **FIXE.TAILLE**: attend un entier en entrée – ne renvoie rien – la taille de la tortue est fixée au nombre fourni en entrée. La taille ne peut excéder 20 et ne concerne que le type par défaut, à savoir le triangle. La valeur par défaut est de 8.

Exemple :

FIXE.TAILLE 10 → -

 La taille ne peut excéder 20 et ne concerne que le type par défaut, à savoir le triangle. La valeur par défaut est de 8.

* **TAILLE**: n’attend rien en entrée – renvoie un entier – l’entier renvoyé indique la taille actuelle de la tortue.

Exemple :

ECRIS TAILLE → 8

La taille renvoyée si la tortue n’est pas la tortue par défaut sera toujours indépendante de celle de la tortue PNG qui ne bouge pas !

#### **Le crayon**

* **BAISSE.CRAYON (raccourci : BC)**: n’attend rien en entrée – ne renvoie rien – le crayon est baissé et donc dessine avec les attributs qui lui ont été affectés. C’est la position par défaut du crayon.

Exemple :

BAISSE.CRAYON → -

* **LEVE.CRAYON (raccourci : LC)**: n’attend rien en entrée – ne renvoie rien – le crayon est levé et donc ne dessine pas lorsque la tortue est déplacée.

Exemple :

LEVE.CRAYON → -

* **BAISSE?** : n’attend rien en entrée – renvoie un booléen – la primitive renvoie "VRAI si le crayon est baissé, "FAUX sinon.

Exemple :

ECRIS BAISSE? → -1

* **FIXE.COULEUR.CRAYON (raccourci : FCC)**: attend un entier en entrée – ne renvoie rien – la primitive détermine la couleur d’écriture du crayon de la tortue[[13]](#footnote-13).

Exemple :

FIXE.COULEUR.CRAYON 5 → -

* **COULEUR.CRAYON (raccourci : CC)**: n’attend rien en entrée – renvoie un entier – l’entier renvoyé correspond à la couleur du crayon de la tortue.

Exemple :

ECRIS COULEUR.CRAYON → 5

* **FIXE.TAILLE.CRAYON**: attend un entier en entrée – ne renvoie rien – l’épaisseur du trait du crayon de la tortue est fixée au nombre fourni en entrée.

Exemple :

FIXE.TAILLE.CRAYON 2 → -

* **TAILLE.CRAYON**: n’attend rien en entrée – renvoie un entier – l’entier renvoyé correspond à l’épaisseur de trait du crayon de la tortue.

Exemple :

ECRIS TAILLE.CRAYON → 2

* **INVERSE.CRAYON**: n’attend rien en entrée – ne renvoie rien – le crayon écrit dans la couleur complémentaire de la couleur en cours.

Exemple :

INVERSE.CRAYON → -

* **GOMME**: n’attend rien en entrée – ne renvoie rien – le crayon efface en avançant.

Exemple :

GOMME → -

 Si la couleur de l’écran est modifiée, les traits effacés réapparaîtront.

* **NORMAL**: n’attend rien en entrée – ne renvoie rien – le crayon n’inverse plus et n’efface plus.

Exemple :

NORMAL → -

* **ETAT.CRAYON**: n’attend rien en entrée – renvoie une liste – la liste renvoyée fournit les données relatives au crayon de la tortue : sa couleur, son épaisseur, s’il écrit ou non, si l’écriture est inversée ou non.

Exemple :

ECRIS ETAT.CRAYON → 1 1 -1 0

* **FIXE.CRAYON** ou **FIXE.ETAT.CRAYON** (raccourci : **FEC**): attend une liste en entrée – ne renvoie rien – l’état du crayon est fixé selon le contenu de la liste : couleur, épaisseur, écriture ou non, inversion ou non.

Exemple :

FIXE.ETAT.CRAYON [1 2 -1 0] → -

#### **Formes prédéfinies**

**GVLOGO** fournit plusieurs formes prédéfinies qui seront dessinées soit à une position indiquée par l’utilisateur soit par rapport à la position de la tortue. Ces formes ne sont pas orientables suivant un angle (par exemple, celui de l’orientation de la tortue), mais de telles formes sont facilement programmables en LOGO lui-même.

Afin d’éviter un conflit entre le nom des éventuelles procédures définies par l’utilisateur et celui des primitives de dessins de formes, on emploie pour nommer ces dernières le préfixe F\_.

* **F.RECTANGLE**: attend une liste en entrée – ne renvoie rien – un rectangle est dessiné avec les coordonnées de deux points fournies en entrée[[14]](#footnote-14).

Exemple :

F.RECTANGLE [100 100 200 200] → -

* **F.RECTANGLE.ARRONDI**: attend une liste en entrée – ne renvoie rien – un rectangle arrondi est dessiné avec les coordonnées de deux points fournies en entrée.

Exemple :

F.RECTANGLE.ARRONDI [50 100 150 200] → -

* **F.CARRE**: attend une liste en entrée – ne renvoie rien – un carré est dessiné avec les coordonnées du point supérieur gauche suivies de la longueur de son côté.

Exemple :

F.CARRE [20 20 50] → -

* **F.ELLIPSE**: attend une liste en entrée – ne renvoie rien – une ellipse est dessinée avec les coordonnées de deux points fournies en entrée.

Exemple :

F.ELLIPSE [100 100 200 200] → -

* **F.CERCLE**: attend une liste en entrée – ne renvoie rien – un cercle est dessiné avec les coordonnées du premier point du cercle suivies de la longueur de son rayon.

Exemple :

F.CERCLE [20 20 50] → -

* **REMPLIS**: n’attend rien en entrée – ne renvoie rien – les figures seront pleines et par conséquent recouvriront celles qui seront placées dans leur espace de dessin.

Exemple :

REMPLIS → -

* **LAISSE.VOIR**: n’attend rien en entrée – ne renvoie rien – les figures seront transparentes et par conséquent ne recouvriront pas celles qui seront placées dans leur espace de dessin.

Exemple :

LAISSE.VOIR → -

#### **Texte avec la tortue**

La tortue peut écrire à son emplacement. Le texte aura alors sa couleur et son orientation. L’utilisateur peut aussi écrire à un emplacement défini par lui avec l’orientation, la taille et la couleur de son choix.

* **TEXTE.TORTUE**: attend un mot ou une liste en entrée – ne renvoie rien – le texte spécifié en paramètre est écrit sur l’écran de la tortue en utilisant ses coordonnées, sa taille, sa couleur et son orientation.

Exemple :

TEXTE.TORTUE [Bonjour mon ami !] → -

* **FIXE.ANGLE.TEXTE**: attend un entier en entrée – ne renvoie rien – le texte sera écrit avec l’angle spécifié en entrée, selon le même principe que l’orientation de la tortue.

Exemple :

FIXE.ANGLE.TEXTE 90 → -

* **ANGLE.TEXTE**: n’attend rien en entrée – renvoie un entier – la primitive renvoie l’angle actuel d’écriture d’un texte sur l’écran de la tortue.

Exemple :

ECRIS ANGLE.TEXTE → 90

* **FIXE.TAILLE.FONTE**: attend un entier en entrée – ne renvoie rien – le texte sera écrit avec la taille de caractère spécifiée en entrée, si cette taille est possible pour la police en cours.

Exemple :

FIXE.TAILLE.FONTE 16 → -

* **TAILLE.FONTE**: n’attend rien en entrée – renvoie un entier – la primitive renvoie la taille actuelle d’écriture d’un texte sur l’écran de la tortue.

Exemple :

ECRIS TAILLE.FONTE → 16

* **FIXE.COULEUR.TEXTE (raccourci : FCT)**: attend un entier en entrée – ne renvoie rien – le texte sera écrit avec la couleur de caractère spécifiée en entrée.

Exemple :

FIXE.COULEUR.TEXTE 5 → -

* **COULEUR.TEXTE**: n’attend rien en entrée – renvoie un entier – la primitive renvoie la couleur actuelle d’écriture d’un texte sur l’écran de la tortue.

Exemple :

ECRIS COULEUR.TEXTE → 5

* **TEXTE.LIBRE.TORTUE**: attend deux entiers puis un mot ou une liste en entrée – ne renvoie rien – le texte indiqué par le mot ou la liste fourni en paramètre est écrit sur l’écran de la tortue en utilisant les entiers comme coordonnées (abscisse et ordonnée). La taille, la couleur et l’orientation sont celles définies par les primitives définies ci-dessus.

Exemple :

TEXTE.LIBRE.TORTUE 100 150 [Bonjour mon ami !] → -

### **Implémentation de la tortue**

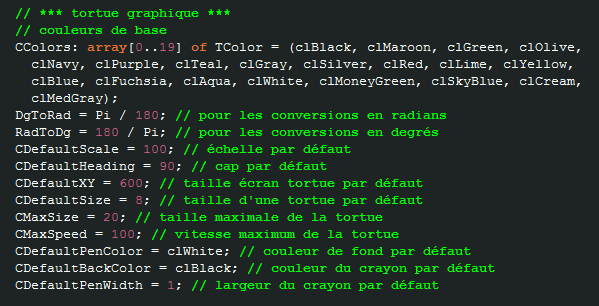
L’implémentation de la tortue pose trois types de problèmes :

* mathématiques : il faut se plonger dans quelques formules de trigonométrie afin de tenir compte de l’orientation de la tortue pendant ses déplacements ;
* graphiques : il faut gérer un fond indépendant du tracé et une tortue qui doit laisser une trace en se déplaçant à la manière d’un *sprite[[15]](#footnote-15)*;
* de communication par événements : une tortue personnalisée sera tributaire des dessins envoyés par l’application tandis que le fond dépendra du contrôle sous-jacent à la surface de la tortue (généralement un panneau ou une surface d’image). Il est aussi intéressant de notifier tout changement intervenu en rapport avec la tortue, par exemple pour afficher son état dans la barre de statut.

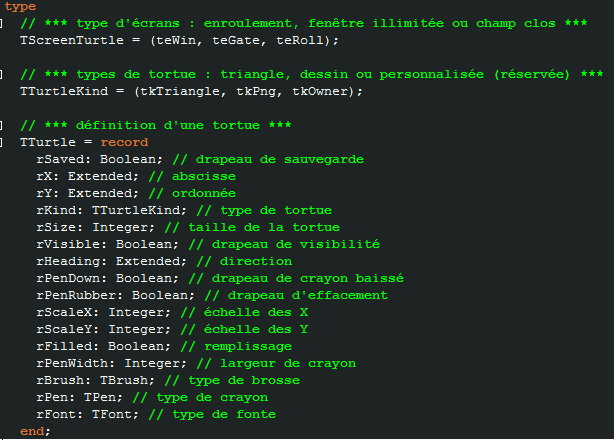
Afin de tenir compte de ces problèmes, l’implémentation met en œuvre une bibliothèque tierce qui ne fonctionne qu’avec Lazarus : **BGRABitmap**. Cette bibliothèque propose un ensemble d’outils pour créer et manipuler des images avec transparence. Elle est libre de droits et régulièrement mise à jour *via* le projet LazPaint. Outre les améliorations citées, résolues à travers la gestion d’un canal alpha pour la transparence, elle prend en charge l’anti-alias qui permet de réaliser des dessins de meilleure qualité, sans crénelages désagréables. Le seul point noir est qu’elle ne fonctionne qu’avec Lazarus, mais ceci n’est pas vraiment gênant puisque le projet s’appuie avant tout sur cet IDE.

#### **Constantes**

L’unité **GVConsts** n’est modifiée que modestement par l’ajout de quelques constantes :



Quelques types viennent compléter le tout afin de définir le type d’écran et de tortue, ainsi qu’un enregistrement possible de l’état complet de la tortue :



#### **Un peu de mathématiques**

Afin de déterminer un déplacement de la tortue sur la surface de travail, il faut connaître ses coordonnées pas à pas en fonction de son cap. Pour cela, on utilisera les fonctions sinus et cosinus :



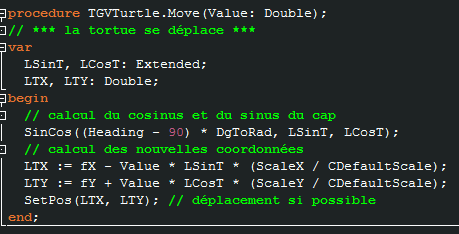
Figure - Point avec sinus et cosinus

Aussi bien Lazarus que Delphi fournissent une procédure bien pratique[[16]](#footnote-16) pour les déterminer : SinCos définie pour un réel étendu de la manière suivante :



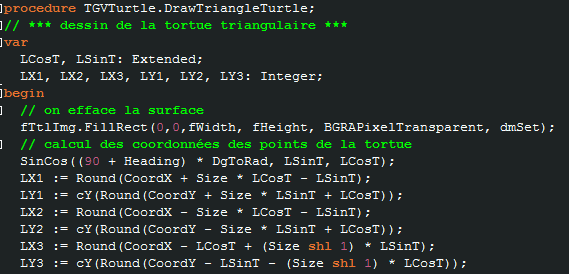
Theta est l’angle exprimé en radians et les sinus et cosinus sont renvoyés dans les paramètres appropriés.

Voici par exemple comment elle est utilisée dans la procédure Move qui déplace la tortue :



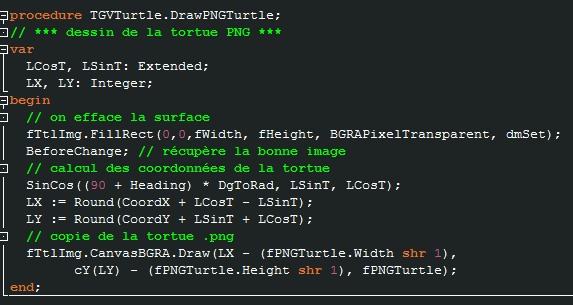
fX et fY sont les coordonnées avant le déplacement. On remarquera la transformation des degrés en radians pour l’orientation ainsi que l’utilisation des échelles pour déterminer le nombre réel de pixels à franchir. La position sera par la suite arrondie puisqu’on ne peut dessiner que des pixels à des positions entières.

De la même manière, lorsqu’on voudra dessiner le triangle représentant la tortue, des formules similaires seront utilisées :



L’utilisation de shl au lieu d’une multiplication par deux traditionnelle est très efficace pour ce qui est de la vitesse d’exécution.

En ce qui concerne la tortue personnalisée, les données sont tout à fait différentes. Le choix a été fait de partir d’une série de « clichés » de la tortue au format « png », car il supporte très bien la transparence. Malheureusement, les algorithmes qui permettent les rotations sont complexes : c’est pourquoi la solution d’images préenregistrées a été choisie. Elle est par ailleurs plutôt rapide.

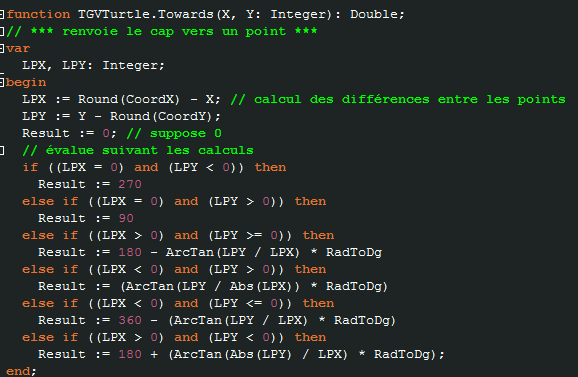


Une autre fonction complexe est Towards qui permet de rendre l’orientation nécessaire de la tortue pour qu’elle pointe vers un point. Elle utilise la fonction mathématique ArcTan qui détermine un angle dont la tangente est donnée.



Figure - Arc Tangente

En voici le listing[[17]](#footnote-17) :



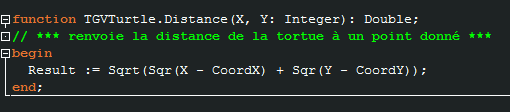
Enfin, la fonction Distance permet de calculer la distance entre la tortue et un point donné. Elle fait appel à de simples calculs à partir de triangles rectangles, donc en appliquant le théorème de Pythagore.

La formule bien connue est :



avec xa, ya les coordonnées du premier point etxb, yb celles du second.

La fonction en Pascal donne[[18]](#footnote-18) :



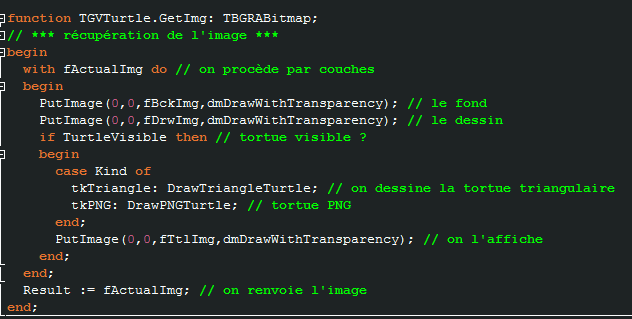
#### **Se déplacer sur l’écran**

En dehors de ces quelques notions de mathématiques, le déplacement de la tortue à l’écran pose des problèmes d’affichage. La solution adoptée est celle de la transparence bien maîtrisée par la bibliothèque **BGRABitmap**.

 Sans cette bibliothèque, le comportement de la tortue avec Lazarus était erratique : la transparence fonctionne avec Lazarus dès qu’il s’agit de dessiner une forme simple sur le canevas ; mais si l’on désire déplacer la tortue sur une surface transparente qui prend sa couleur de fond sur un contrôle, le dessin en entier disparaît… du moins, dans la limite de mes connaissances !

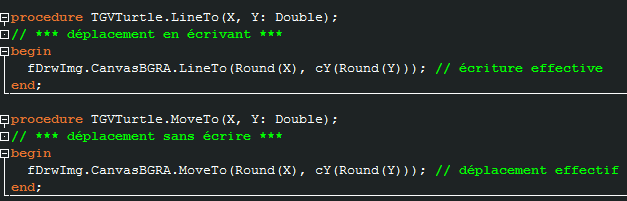
Le dessin met en jeu trois couches : la première s’occupe du fond, la deuxième du dessin laissé par la tortue, la troisième par la représentation de cette dernière. On est ainsi assuré que les objets manipulés n’interféreront pas. La bibliothèque fournit en effet les éléments qui permettent de superposer ces couches en gérant la transparence nécessaire.

La méthode centrale se présente ainsi :

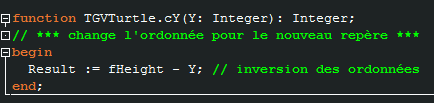


On utilise fBckImg pour afficher le fond, fDrwImg pour le dessin et enfin fTtlImg pour la tortue. Les trois éléments sont du type **TBGRABitmap**. C’est la bibliothèque **BGRABitmap** qui s’occupe de fondre ces trois couches afin d’obtenir le dessin souhaité.

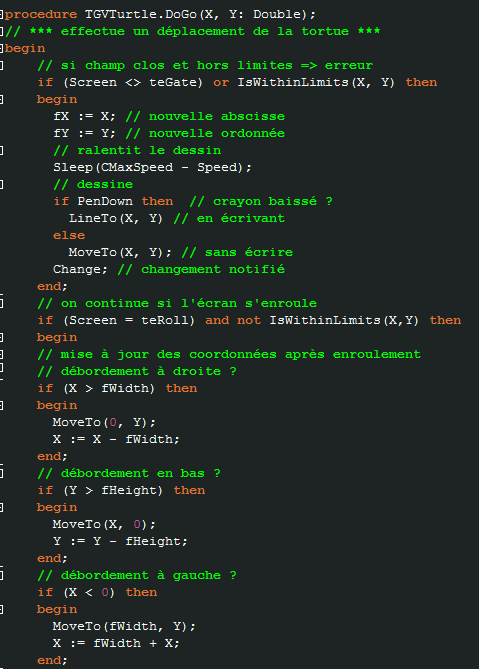
On notera par ailleurs que les procédures LineTo et MoveTo adaptent les coordonnées fournies afin de les rendre cohérentes avec un repère dont l’origine est située en bas à gauche de la surface de dessin et non en haut à gauche comme en Pascal.

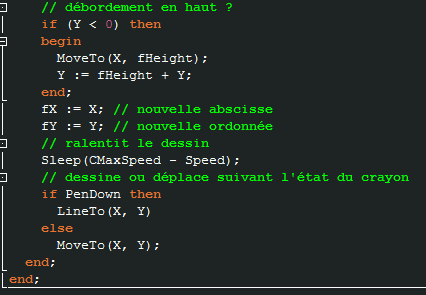


C’est la fonction cY qui est chargée du travail de modification du repère :



Le déplacement proprement dit de la tortue est traité par la méthode DoGo qui prend en compte le type d’écran teRoll, c’est-à-dire l’enroulement. En voici l’implémentation :





Bien que ce soit une erreur que de vouloir faire sortir la tortue de son espace alors qu’elle est en mode champ clos, **GVLOGO** se contente de ne pas déplacer la tortue dans ce cas. On pourrait aussi déclencher une exception, mais, d’un point de vue pédagogique, il a semblé plus intéressant de laisser à l’utilisateur le soin de comprendre ce qu’il se passe dans ce cas.

#### **Les événements**

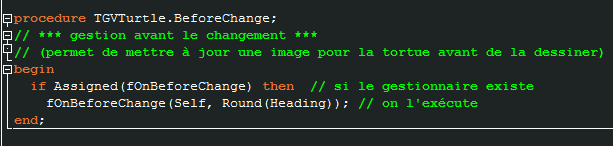
La récupération de l’image désirée de la tortue est effectuée par le programme qui exploite la classe T**GVTurtles**. Même si les multiples liens entre les procédures et les unités rendent l’ensemble plutôt complexe, le principe de fonctionnement est simple : quand l’instance de la classe **TGVTurtles** a besoin de dessiner la tortue, elle exécute la procédure BeforeChange qui envoie en paramètre essentiel l’orientation de la tortue. En retour, elle s’attend à ce que l’unité appelée ou le programme ait affecté la bonne image à sa propriété TurtleImg.

Autrement dit, il s’agit d’une sorte d’aller-retour entre les deux unités : la première envoie un pointeur vers elle-même et l’orientation de la tortue si elle trouve une procédure affectée à son champ privé fOnBeforeChange ; la seconde récupère l’orientation de la tortue, cherche l’image correspondante et l’affecte en retour à la propriété TurtleImg de l’unité génératrice de l’événement.

La méthode BeforeChange est déclarée ainsi, dans la partie protected de la définition :



Et voici sa propre définition :



On voit que la procédure fait elle-même référence au champ privé fOnBeforeChange défini ainsi :

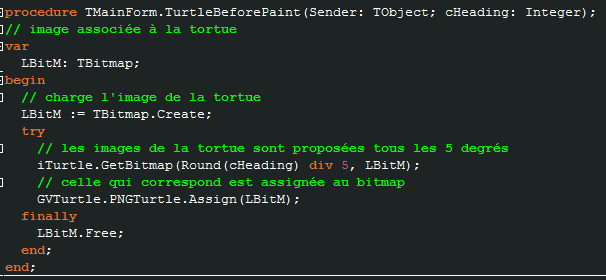


Quant au type **TTurtleBeforeEvent**, il fournit un squelette de la procédure appelante :



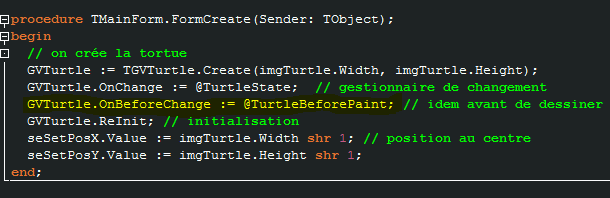
Ce squelette est essentiel puisqu’en fournissant un modèle commun il permet aux unités de communiquer entre elles.

Voici à présent comment le programme de test opère pour répondre à la requête de l’unité **GVTurtles** :



On remarquera que fOnBeforeChange de l’unité **GVTurtles** et TurtleBeforePaint du programme de test ont exactement le même en-tête calqué sur **TTurtleBeforeEvent**. Peu importe que les noms soient différents : ce qui compte, ce sont les paramètres absolument identiques dans leur nombre, leur agencement et leurs types.

Pour que l’ensemble fonctionne, il faut évidemment affecter la procédure appelée au gestionnaire d’événements approprié avec une ligne du genre :



Ici, GVTurtle est une instance de la classe **TGVTurtle**. Cette ligne figure dans la méthode FormCreate de la fiche. Le programme « sait » à présent quelle procédure il doit appeler lorsque l’événement est déclenché.

 Contrairement à Delphi, Lazarus exige l’emploi de **@** devant la méthode pour l’affecter au gestionnaire d’événements.

Un second événement plus simple est déclenché à chaque modification de la tortue : il s’agit de **TTurtleEvent**:



Cet événement ne transmet qu’un pointeur vers l’instance de la classe qui a déclenché l’événement, mais permet ainsi d’accéder aux propriétés de la tortue susceptibles d’avoir été modifiées : ses coordonnées, son cap, l’état de sa visibilité, celui de sa capacité à écrire et la couleur d’écriture du crayon. L’intérêt d’une telle notification est, par exemple, de renseigner en temps réel une barre de statut qui affichera en clair ces données pour informer l’utilisateur.

Du côté de l’unité **GVTurtles**, on retrouve :

* un champ privé qui conservera la procédure affectée si elle existe :



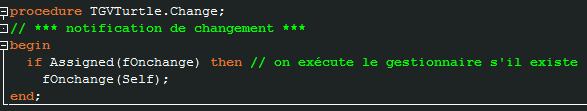
* une propriété en accès public et renvoyant à ce champ privé :



* une procédure dans la partie protected de la définition de la classe, appelée quand nécessaire et en charge d’appeler elle-même la procédure de l’unité qui exploite la tortue :



* la définition de cette procédure :



Il ressemble tout à fait au gestionnaire vu précédemment : si le champ privé fOnChange est affecté, on exécute la méthode à laquelle il renvoie.

Du côté du programme de test, on retrouve :

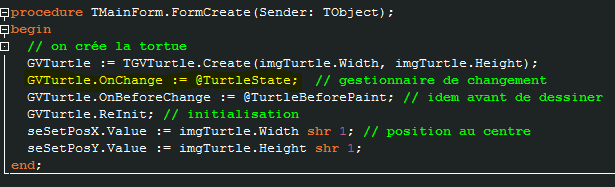
* la déclaration de la procédure qui sera appelée en se conformant au squelette fourni par **TTurtleEvent** :



* sa définition qui affiche les données fournies sur la barre de statut :



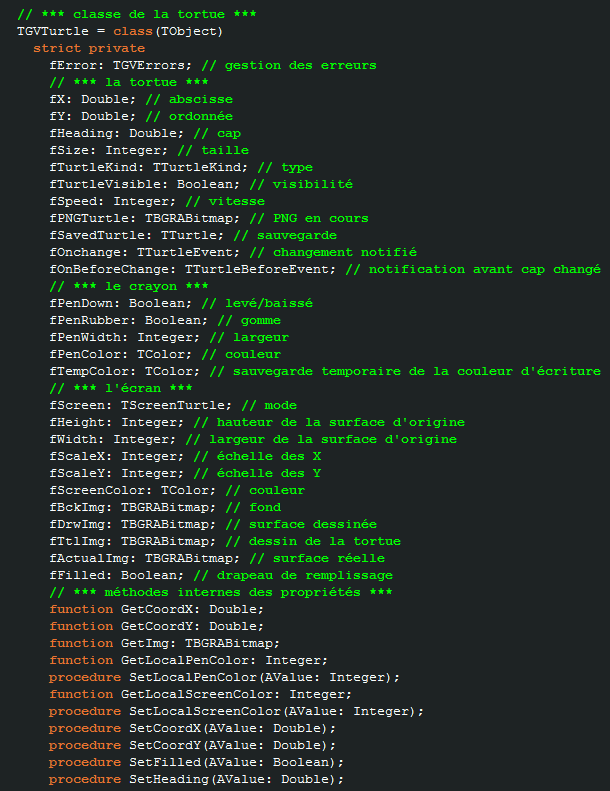
* afin de créer le lien entre les deux unités, son affectation lors de la création de la fiche :

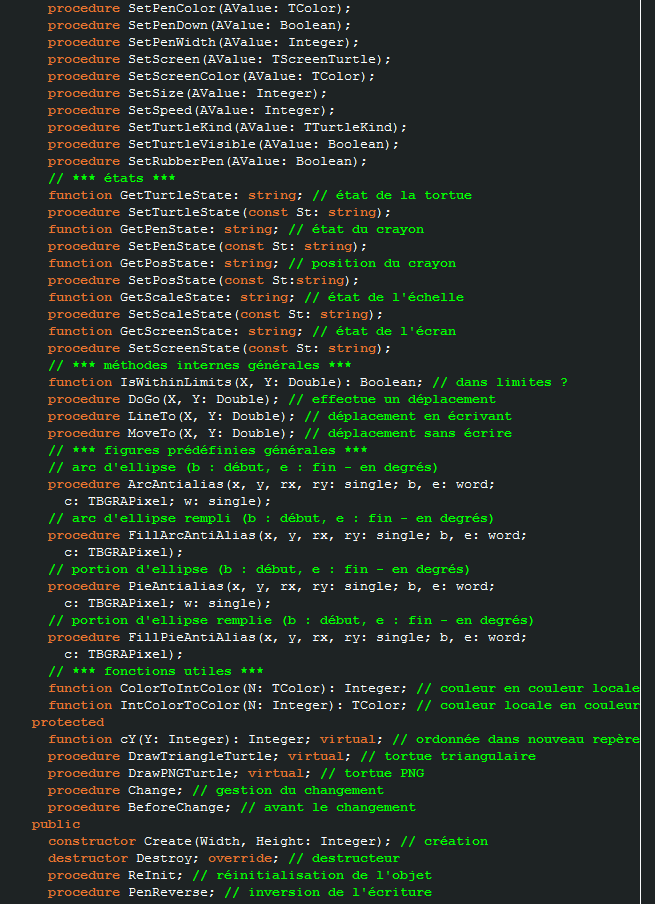


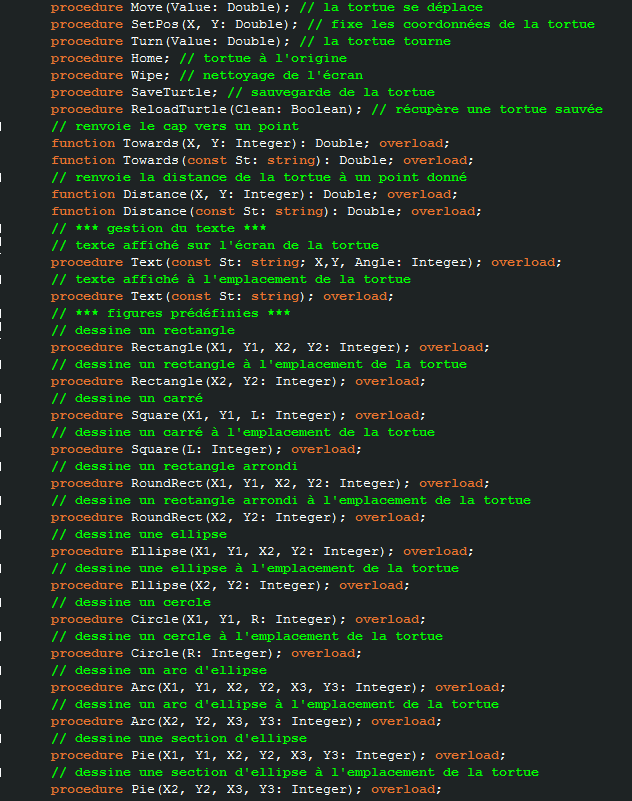
 On a là un cas très classique de gestion d’événements, avec toutes les étapes de sa mise en place. Le code source de l’unité est alors émaillé d’appels à Change lorsqu’intervient un changement concernant la tortue.

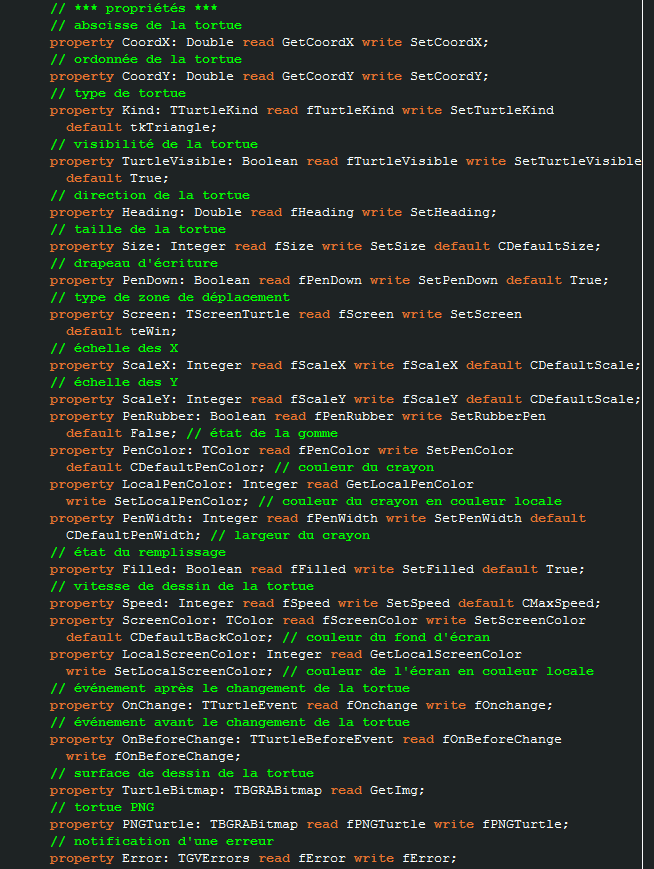
#### **La classe TGVTURTLE**

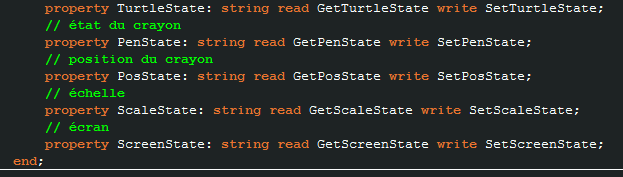
Voici l’interface de la copieuse classe **TGVTurtle** :











En dehors des aspects déjà étudiés, la classe ne présente pas de problèmes majeurs. La tortue fait bien sûr appel à des propriétés afin que l’utilisateur puisse en maîtriser l’aspect et le comportement. L’interface est aussi alourdie par de nombreuses fonctions de dessin : une première version dessine en se référant à des coordonnées traditionnelles tandis qu’une seconde version surchargée grâce à overload utilise les coordonnées de la tortue pour amorcer son dessin.

On notera que les coordonnées sont des réels alors que la position finale est un entier : cela permet de calculer au plus près la position de la tortue avant de l’arrondir, évitant ainsi des approximations cumulées.

#### **Test de l’unité TGVTurtles**

Comme toujours, le programme de test est décliné en deux versions :

* Lazarus pour Windows 32 ;
* Lazarus pour Linux.

L’utilisateur pourra tester la plupart des fonctions et propriétés définies pour manipuler la tortue. Le programmeur sera plus intéressé par le mécanisme des notifications d’événements tel que décrit plus haut.

Pour avoir une idée des possibilités de dessin de la tortue, un bouton marqué « dessin » dessine une série de 36 carrés en rosace. Pour se rendre compte de la rapidité de la tortue triangulaire par rapport à celle graphique, il est judicieux d’utiliser ce bouton spécial dans trois configurations : graphique, triangle et tortue cachée.

On remarquera aussi quatre boutons : le premier met en œuvre la méthode Text dans sa version indépendante de la tortue tandis que le second la teste avec des coordonnées fournies par cette même tortue ; le troisième illustre ce que peut réaliser la méthode Text avec une fleur de mots en couleurs ; enfin, le dernier, accompagné d’un **TSpinEdit**, contrôle l’épaisseur du trait de la tortue lorsqu’elle dessine. De plus, trois boutons **TRadioEdit** permettent de contrôler le type de champ de la tortue : illimité avec teWin (valeur par défaut), limité par les bords de l’écran (teGate) et enfin enroulé (teRoll).

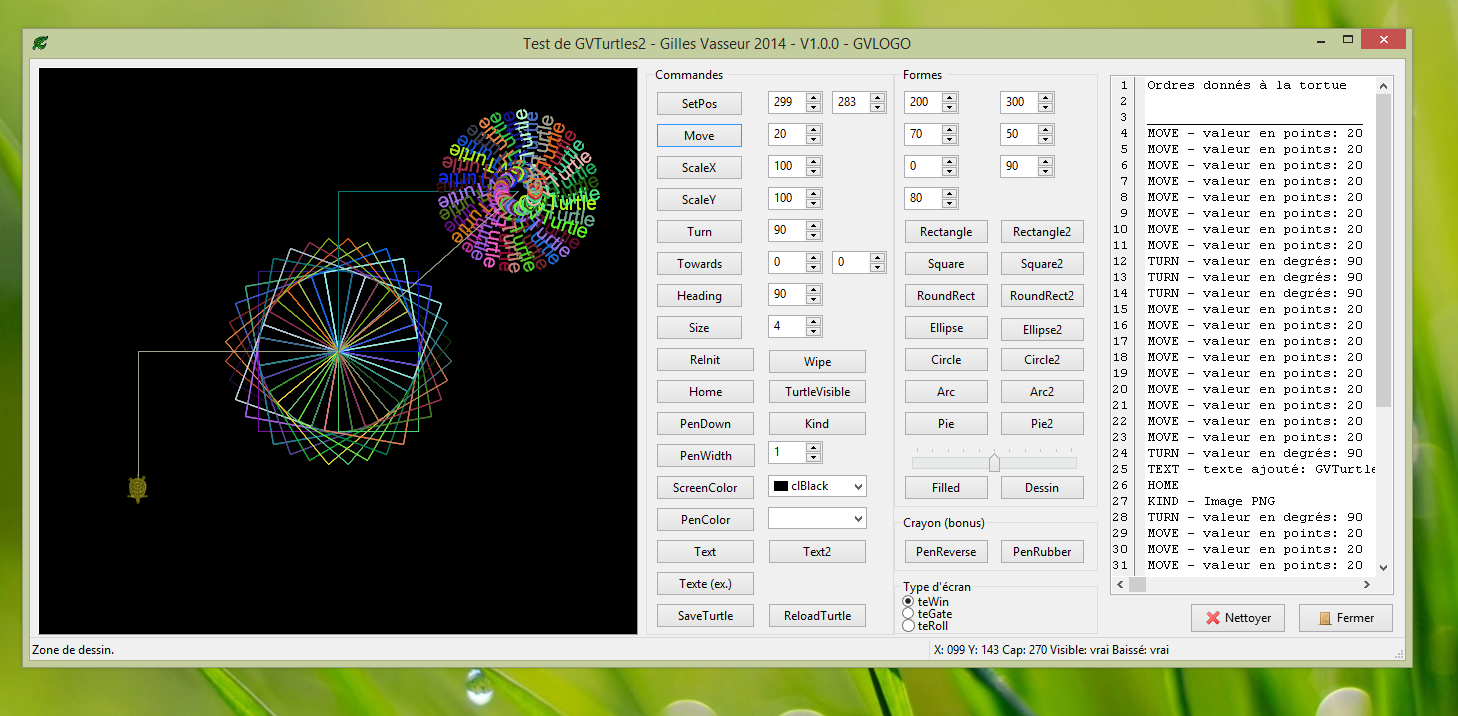


Figure - Test de la tortue

# III - Récréation : EasyTurtle (logiciel de dessin)

## **A - Le projet EasyTurtle**

Avant d’aborder les outils de programmation et de rentrer dans le cœur de l’interpréteur, une récréation s’impose avec la création d’un petit logiciel de dessin baptisé **EasyTurtle**. Il s’agit de mettre en œuvre l’unité **GVTurtles** en permettant à un enfant de dessiner avec la tortue, de rejouer l’ensemble des ordres qu’il lui aura donnés, ou encore de charger et de sauvegarder ses réalisations.

Pour le programmeur, il s’agira d’utiliser quelques outils particulièrement utiles, en particulier les listes d’actions.

## **B - Mode d’emploi rapide**

### **L’écran d’accueil**

L’écran d’accueil se présente comme ceci :



Figure - Écran principal de EasyTurtle

Sur l’essentiel de la fenêtre se situe la zone d’affichage de la tortue : c’est ici que seront réalisés les dessins. Afin de faciliter sa manipulation, la zone de dessin est en mode « clos », ce qui signifie que la tortue ne peut pas être perdue de vue, car tout ordre tendant à la faire disparaître conduira à la faire buter contre le bord de son champ comme s’il y avait une barrière.

À droite de cette zone d’affichage, on aperçoit toute une série de boutons : ce sont les actions mises à disposition de l’utilisateur.

### **La tortue**

On distingue ainsi les actions concernant la tortue :



**Figure 13 - Actions de la tortue (EasyTurtle)**

La tortue peut donc avancer, reculer, tourner à gauche et à droite. Chacune de ces opérations se fait selon une valeur par défaut modifiable grâce à une fenêtre de réglage des préférences.

Si elle est de forme triangulaire, **l**a tortue peut aussi grossir et rapetisser. Dans le cas contraire, ces boutons sont grisés, et par conséquent inactifs.

L’utilisateur peut effacer l’écran, renvoyer la tortue à son origine, l’autoriser ou lui interdire d’écrire, la rendre visible ou invisible.

 Un bouton indique toujours l’action à réaliser, et non l’état de la tortue. Ainsi, si le bouton comporte le message « N’écris plus », c’est que la tortue laisse actuellement une trace et qu’une pression sur le bouton lui demandera de ne plus écrire.

### **Couleurs et formes**

Le panneau suivant s’occupe des couleurs et des formes :

****

**Figure 14 - Couleurs et formes (EasyTurtle)**

Le premier bouton permute l’apparence de la tortue, entre le triangle et le dessin au format png. Le second ouvre une fenêtre de choix de la couleur du crayon, tout comme le suivant pour la couleur de fond. Les deux derniers dessinent respectivement un carré et un cercle à l’emplacement de la tortue.

 Avec Lazarus, lorsque la couleur de fond de la surface de dessin de la tortue change, l’écran est entièrement réinitialisé.

### **Ordres généraux**

Le panneau suivant regroupe les ordres généraux concernant le travail effectué par la tortue :



**Figure 15 - Ordres (EasyTurtle)**

**EasyTurtle** propose un enregistrement des ordres donnés à la tortue. C’est à partir de cet enregistrement qu’opéreront les boutons, « Charge », « Sauve », « RAZ » et « Défais ».

L’utilisateur peut sauvegarder et charger son travail. Il peut ouvrir des fenêtres spécialisées : « Outils », « À Propos » et « Aide ». Il peut aussi rejouer une séquence enregistrée et interrompre cette répétition : ce sont les deux boutons sans légende qui apparaissent en bas à gauche de la copie d’écran. Il peut encore annuler le dernier ordre donné à la tortue (bouton « Défais ») ou remettre à zéro toute la séquence (bouton « RAZ »). Enfin, il peut quitter le logiciel.

Le fait de charger une suite d’ordres l’exécute immédiatement après le chargement.

 Suivant l’état du logiciel, certains boutons seront désactivés : par exemple, les boutons « Défais », « Sauve » et « Rejoue » ne seront activés que si des ordres ont été enregistrés. Le bouton « Stop » ne sera activé que si le bouton « Rejoue » a été pressé et seulement le temps de la répétition. Lorsque le bouton « Rejoue » a été pressé, tous les boutons, sauf « Quitter » et « Stop », sont désactivés.

Pour réinitialiser la séquence d’ordres, plusieurs possibilités sont offertes : chargement d’un nouveau fichier d’ordres, pression sur le bouton « RAZ » et modification du fond de l’écran. Contrairement au bouton « Efface l’écran », le bouton « RAZ » conserve la couleur d’écriture et celle du fond de l’écran.

### **L’aide**

Une aide peut être demandée :



**Figure 16 - Aide (EasyTurtle)**

### **Boîte « À propos »**

De même, on peut accéder à une boîte « À propos » :



**Figure 17 - A propos (EasyTurtle)**

### **Boîte des préférences**

Enfin, une boîte d’outils permet de modifier les valeurs par défaut de certains ordres concernant le dessin effectué par la tortue :



**Figure 18 - Outils (EasyTurtle)**

Les modifications apportées grâce à la boîte des préférences sont enregistrées avec le fichier des ordres. Les valeurs sont données en pixels ou en degrés.

### **Autres éléments**

Tous les boutons décrits ci-dessus ont leur double dans la barre d’outils située en haut de la fenêtre principale :



Les pictogrammes sont évidemment les mêmes afin de renforcer la cohérence du logiciel. De plus, la plupart des ordres peuvent être donnés par une combinaison de touches indiquée en aide ponctuelle près du bouton avant de le presser et dans la barre de statut.

Pour terminer cette présentation rapide, il faut noter des indicateurs fournis par la barre de statut et par deux disques situés en bas à droite de la fenêtre principale :

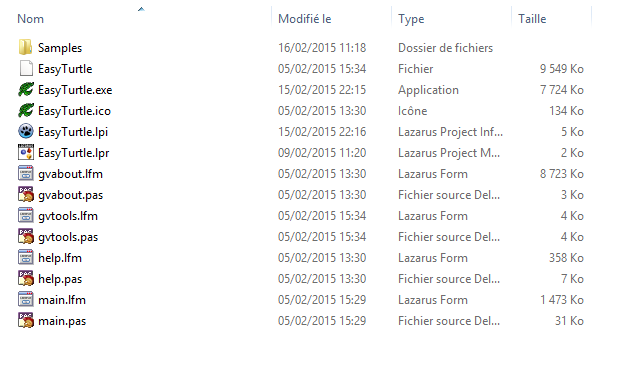


**Figure 19 - Barre de statut (EasyTurtle)**

On prend ainsi connaissance d’une aide succincte concernant le bouton survolé par la souris, des principales valeurs associées à la tortue et des valeurs attribuées à la couleur du crayon de la tortue (premier disque) et au fond de l’écran (second disque), ainsi que l’état du logiciel : « enregistrement en cours », « sauvegarde en cours », « chargement en cours » et « répétition des ordres ».

## **C - La programmation**

La suite de cette partie décrit le fonctionnement d’**EasyTurtle**. Contrairement aux autres logiciels d’exemples, celui-ci fonctionne à partir de plusieurs fiches :



**Figure 20 - Répertoire de EasyTurtle.**

L’unité principale s’appelle **Main.pas**. **GVAbout.pas** contient la boîte « À propos », **Help.pas** l’aide et **GVTools.pas** la boîte des préférences.

L’appel d’une fiche externe se fait selon un modèle courant depuis la fiche « Main ». Voici, par exemple, l’appel de la boîte « À propos » :



**Figure 21 - Appel d'une autre fiche (EasyTurtle).**

### **La fiche principale**

La fiche « Main » contient toutes les méthodes nécessaires au fonctionnement d’**EasyTurtle**. Les éléments les plus complexes sont ceux relatifs à la mémorisation des ordres donnés à la tortue : le logiciel utilise à cette fin un tableau ouvert géré par une méthode nommée Memorize :



**Figure 22 - Mémorisation des ordres (EasyTurtle).**

Cette méthode ajuste la taille du tableau avant d’enregistrer la nouvelle donnée. Elle indique aussi que la séquence a été modifiée en vue d’un futur enregistrement.

La plupart des ordres sont gérés de manière identique. Voici par exemple l’ordre ActionForwardExecute qui fait avancer la tortue :



**Figure 23 - La tortue avance (EasyTurtle).**

On exécute l’ordre et on l’enregistre, rien de plus facile ! Simplement, afin de de permettre aux boutons, à la barre d’outils et aux combinaisons de touches de fonctionner de la même manière sans dupliquer le code, on utilise un composant **TActionList** qui centralise les actions.

En plus de l’exécution, on a prévu une mise à jour (disponibilité, visibilité, affichage) de chaque fonction suivant l’état du logiciel :



**Figure 24 - Mise à jour (EasyTurtle).**

Ici, l’objet appelant (qui doit être une action) n’est activé que si le mode est celui de l’enregistrement. En effet, il ne faut pas continuer à dessiner au cours de la sauvegarde, du chargement ou si l’on est en train de rejouer toute la séquence.

On aurait pu indiquer directement ActionForward dans la méthode, mais le transtypage (Sender as **TAction**) permet de partager le même gestionnaire avec d’autres actions au comportement identique (ActionBackward, par exemple).

Rejouer la séquence exige de dispatcher les ordres en fonction de leur enregistrement :



**Figure 25 - Rejouer une séquence (EasyTurtle).**

Le champ privé fCmd est le pointeur utilisé sur l’ordre en cours. Avant de rejouer les ordres, on s’occupe de l’en-tête qui contient les valeurs modifiables depuis la fenêtre des paramètres. Ces valeurs sont enregistrées avec l’éventuel fichier de sauvegarde.

On remarquera la présence de Application.ProcessMessages qui permet à l’application de réagir aux événements tels que l’appui sur le bouton « Stop ». On profite aussi de cette boucle pour animer certains boutons : ainsi, deux roues seront animées. Certaines données ne sont utiles qu’en cas d’action de correction, en remontant dans le temps : pour rejouer la séquence, on les ignore simplement (voir le traitement de CT\_Home, par exemple).

La partie la plus complexe de cette unité est celle relative à la fonction « Défaire » :



**Figure 26 - Annuler la dernière action (EasyTurtle).**

En effet, pour être annulées, certaines fonctions exigent que l’état de la tortue soit enregistré au préalable : ainsi, pour annuler un retour à l’origine, il faut se souvenir de l’ancien emplacement de la tortue, mais aussi de son orientation. En amont, il faut donc que chaque ordre mémorise ce qu’attendra une éventuelle correction.

 La solution adoptée ici est de repasser sur le dernier trait en l’effaçant. Cette méthode est très rapide, mais elle peut effacer un trait qui devrait être conservé parce qu’il est recouvrant.

La méthode ActionSaveExecute d’enregistrement de la séquence doit tenir compte des remarques précédentes :



**Figure 27 - Sauvegarde (EasyTurtle).**

Cette méthode vérifie l’existence du fichier avant d’éventuellement l’écraser puis enregistre les données dans l’ordre du tableau des commandes. Elle anime aussi deux roues en permettant à l’application de gérer les événements grâce à la méthode Application.ProcessMessages.

Le chargement d’un fichier est un brin plus complexe :



**Figure 28 - Chargement d'un fichier (EasyTurtle).**

Le premier élément du fichier doit contenir le numéro de version actuelle, soit 100. Il est suivi de l’en-tête puis des données proprement dites. Au fur et à mesure de la lecture, on fait tourner les roues habituelles et on enregistre les ordres grâce à la méthode Memorize. À la fin de la méthode, on lance l’exécution de la séquence afin d’avoir un écran à jour : sans ce dernier point, la fonction « Défaire » serait erronée puisqu’elle corrigerait un écran non dessiné !

On peut enfin mentionner la méthode CloseQuery :



**Figure 29 - Demande de fermeture (EasyTurtle).**

Lors d’une demande de fermeture du logiciel, on vérifie préalablement si la séquence en cours, lorsqu’elle a été modifiée, doit être enregistrée. Les messages diffèrent suivant les cas.

Le test de CanClose qui modifie si nécessaire pState s’explique par le fait que l’utilisateur peut avoir demandé la fermeture du logiciel alors qu’on rejoue une séquence. On interrompt alors cette répétition et l’on ferme le logiciel.

### **Les autres fiches**

La boîte des préférences comprise dans l’unité **GVTools** est d’une simplicité enfantine : elle modifie ou non les données qu’elle doit gérer suivant le bouton pressé.

L’unité **Help** permet d’introduire quelques animations très simples : une image suit le déplacement de la souris en modifiant son champ Left, les composants TLabel ont leur texte en gras ou non suivant l’entrée/la sortie de la souris de leur surface…

L’unité **GVAbout** comprend aussi une animation aléatoire de la tortue.

# IV - Les outils de programmation

## **08 - Les piles**

### **Définition**

Une pile informatique peut être figurée par une pile d’assiettes posées sur une table : lorsque j’empile une donnée (*push* en anglais), je pose une assiette et lorsque je dépile une donnée (*pull* en anglais), je retire une assiette de la pile. La dernière donnée est donc la première à pouvoir être extraite : on parle de pile LIFO (*Last In First Out* = dernier entré, premier sorti). Une pile permet en fait de mémoriser les données manipulées dans l’ordre où elles seront à utiliser.



Figure - Notion de pile LIFO

Les piles sont très utilisées par **GVLOGO**. Elles permettent tout à la fois de faire fonctionner l’interpréteur en stockant des données à récupérer plus tard dans un certain ordre, et d’effectuer des calculs complexes, en particulier pour l’évaluation d’expressions arithmétiques telles que (2 \* 4,5) / 8.

Comme la lecture d’une ligne de programme s’effectue par convention de la gauche vers la droite, si l’on rencontre une opération rien n’indique *a priori* que les arguments nécessaires sont fournis : mentalement, nous découvrons les éléments au fur et à mesure de notre lecture. Un programme fait de même !

### **Exemples**

Les exemples qui suivent seront très détaillés. En effet, le fonctionnement d’une pile est au cœur de celui de l’interpréteur lui-même.

#### **Un exemple simple**

On souhaite effectuer l’opération SOMME 24 35 qui additionne les deux nombres fournis en paramètres. Pour cet exemple, on utilisera deux piles, la première contenant les données, la seconde les opérations à effectuer et le nombre de paramètres attendus.

Tout d’abord, on rencontre une opération à effectuer : ici, une addition. On sait qu’elle nécessite deux données : on empile cette opération et le nombre de paramètres attendus.

|  |  |
| --- | --- |
| **<pile vide>** | **2** |
|  | **SOMME** |

Ensuite, on rencontre une donnée. Afin de la mémoriser, on empile 24. SOMME n’attend plus qu’un paramètre : on décrémente le sommet de la seconde pile :

|  |  |
| --- | --- |
| **24** | **1** |
|  | **SOMME** |

On rencontre encore une donnée. On empile 35  et on décrémente le sommet de l’autre pile :

|  |  |
| --- | --- |
| **35** | **0** |
| **24** | **SOMME** |

Le 0 au sommet de la seconde pile indique que les données nécessaires sont acquises. On retire le sommet de la seconde pile pour en extraire l’opération qui peut à présent être effectuée.

|  |  |
| --- | --- |
| **35** | **SOMME** |
| **24** |  |

On effectue alors l’addition sur la pile :

|  |  |
| --- | --- |
| **59** | **<pile vide>** |

On récupère le résultat (59), par exemple pour l’afficher, et la première pile est vide.

Cette technique permet par conséquent de mémoriser des résultats intermédiaires : on peut imaginer que 35 est le résultat d’une autre opération et sa présence sur la pile fournira toujours le second élément nécessaire à l’addition initiale.

#### **Un exemple plus complexe**

On souhaite à présent effectuer l’opération SOMME 24 PRODUIT 5 7, sachant que PRODUIT multiplie deux données sur la pile.

Tout d’abord, on rencontre l’opération à effectuer : ici, une addition. On sait qu’elle nécessite deux données :

|  |  |
| --- | --- |
| **<pile vide>** | **2** |
|  | **SOMME** |

Ensuite, on rencontre une donnée. Afin de la mémoriser, on empile 24.

SOMME n’attend plus qu’un paramètre.

On décrémente le sommet de la seconde pile :

|  |  |
| --- | --- |
| **24** | **1** |
|  | **SOMME** |

On rencontre une seconde opération qui est PRODUIT, elle-même nécessitant deux données. Il faut empiler ces informations :

|  |  |
| --- | --- |
| **24** | **2** |
|  | **PRODUIT** |
|  | **1** |
|  | **SOMME** |

La donnée suivante est 5 qu’on empile en décrémentant par ailleurs le sommet de l’autre pile :

|  |  |
| --- | --- |
| **5** | **1** |
| **24** | **PRODUIT** |
|  | **1** |
|  | **SOMME** |

On a donc une opération SOMME pendante qui attend encore une donnée et une opération PRODUIT qui elle aussi attend une donnée.

La donnée suivante est 7 qu’on empile :

|  |  |
| --- | --- |
| **7** | **0** |
| **5** | **PRODUIT** |
| **24** | **1** |
|  | **SOMME** |

Le 0 au sommet de la seconde pile indique que l’opération pendante qui la suit a les paramètres qu’elle attend à sa disposition. On dépile ce 0 devenu inutile.

|  |  |
| --- | --- |
| **7** | **PRODUIT** |
| **5** | **1** |
| **24** | **SOMME** |

On récupère alors PRODUIT. On peut donc effectuer cette opération et déposer son résultat sur le sommet de la première pile en décrémentant le sommet de la seconde pile.

|  |  |
| --- | --- |
| **35** | **0** |
| **24** | **SOMME** |

On en a terminé avec la multiplication. La pile contient les deux données nécessaires à SOMME : on peut donc l’effectuer comme dans l’exemple 1.

Le 0 au sommet de la seconde pile indique que les données nécessaires sont acquises. On retire le sommet de la seconde pile pour en extraire l’opération qui peut à présent être effectuée.

|  |  |
| --- | --- |
| **35** | **SOMME** |
| **24** |  |

On effectue alors l’addition sur la pile :

|  |  |
| --- | --- |
| **59** | **<pile vide>** |

On récupère le résultat (59), par exemple pour l’afficher, et la première pile est vide.

Il est bien entendu que cet enchaînement d’opérations exige des éléments non encore réalisés :

* le flux du texte à exécuter doit être acquis et analysé pour en définir les éléments de base ;
* il faut pouvoir distinguer une opération d’une donnée ;
* le nombre de paramètres d’une opération doit être retrouvé pour être empilé ;
* une pile fournit au minimum les moyens d’empiler et de dépiler une donnée. D’autres opérations élémentaires sont souvent utiles : duplication de la dernière donnée, test de la profondeur de la pile, incrémentation et décrémentation du sommet… Les piles d’entiers et de réels fournissent d’autre part l’essentiel des opérations qu’on peut effectuer sur les données numériques qu’elles gèrent : addition, soustraction…

### **Opérations sur les piles (et les queues)**

**GVLOGO** n’a pas pour vocation de traiter des opérations de bas niveau comme les piles, mais il en fournit cependant une structure minimale. De plus, il implémente une structure de queue : les éléments sont alors stockés les uns après les autres, et c’est le plus ancien qui est déstocké le premier.



Figure - Notion de queue

Les primitives ci-après sont décrites dans ce chapitre, mais relèvent du traitement des listes. Elles simulent le fonctionnement d’une pile ou d’une queue à partir d’une liste.

* **EMPILE**: attend le nom d’une variable puis un mot ou une liste – ne renvoie rien – la primitive place le second paramètre au début de la variable qui doit être une liste.

Exemples :

EMPILE "MAPILE 124 → -

EMPILE "MAPILE 568 → -

* **SOMMET** : attend le nom d’une variable en entrée – renvoie une liste ou un mot – la primitive renvoie le dernier élément stocké dans la pile indiquée par le paramètre qui doit renvoyé à une liste sans le retirer de cette dernière.

Exemple :

ECRIS SOMMET "MAPILE → 568

* **DEPILE** : attend le nom d’une variable en entrée – renvoie une liste ou un mot – la primitive renvoie le dernier élément stocké dans la pile indiquée par le paramètre qui doit renvoyé à une liste en le retirant de cette dernière.

Exemples :

ECRIS DEPILE "MAPILE → 568

ECRIS DEPILE "MAPILE → 124

* **QUEUE** : attend une variable puis une liste ou un mot en entrée – ne renvoie rien – la primitive stocke le second paramètre dans la première liste pour former une queue.

Exemple :

QUEUE "MAQUEUE [ceci est stocké] → -

* **DEQUEUE** : attend une variable en entrée – renvoie une liste ou un mot – la primitive renvoie le premier élément stocké dans une queue en le retirant de cette dernière.

Exemple :

ECRIS DEQUEUE "MAQUEUE → un ancien élément stocké

Les notions de queues et de piles sont très souples pour **GVLOGO** si bien qu’il est possible de traiter n’importe quelle variable pourvu qu’elle renvoie une liste. De plus, les primitives ENQUEUE et EMPILE sont strictement équivalentes : on peut donc empiler une donnée et traiter la liste comme une pile, une queue ou une liste ordinaire, et réciproquement !

### **Implémentation des piles**

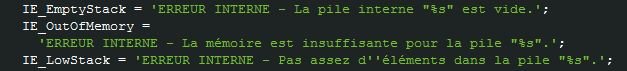
L’unité **GVStacks** offre une classe de pile générique et les classes nécessaires pour manipuler des entiers, des réels et des chaînes de caractères.

On aura tout intérêt, si l’on utilise Delphi, à utiliser les piles fournies dans **System.Generics.Collections**[[19]](#footnote-19). Comme Lazarus ne sait pas encore traiter les classes génériques de manière aussi satisfaisante que Delphi, la solution adoptée pour le projet est de travailler à partir d’une classe générique commune. Elle est plus limitée dans la mesure où elle ne connaît pas d’énumération. D’autre part, la notification est moins complète aussi, Lazarus n’acceptant que les classes comme support des génériques. En revanche, plusieurs fonctionnalités intéressantes ont été ajoutées.

#### **Constantes**

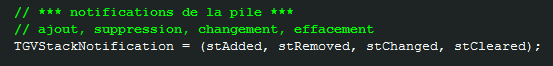
L’unité **GVConsts** a été complétée. On a fixé la taille minimale de la pile à 8, ce qui permet de ne pas réallouer en permanence de la mémoire pour l’accroissement de la pile.

De nouvelles erreurs possibles ont été par ailleurs prévues dans **GVErrConsts**. On retrouve ces erreurs dans les chaînes de ressources :



On remarquera que ces erreurs sont marquées comme internes, car elles ne se produiront que si notre programme final connaît un problème de conception.

Enfin, des constantes énumérées ont été définies pour les notifications :



La première indique qu’un élément a été ajouté à la pile, la deuxième qu’un élément a été retiré, la troisième qu’un changement autre est intervenu, la dernière que la pile a été remise à zéro.

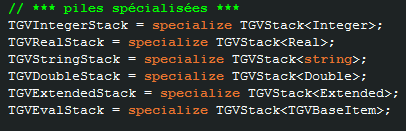
#### **La classe TGVSTack**

Les piles sont implémentées en utilisant les classes génériques. Pour rappel, ces classes fournissent un modèle sur lequel viendront s’appuyer des classes spécialisées. Ainsi, à partir d’une seule classe de pile, on peut créer toutes les piles imaginables simplement en précisant le type à utiliser.

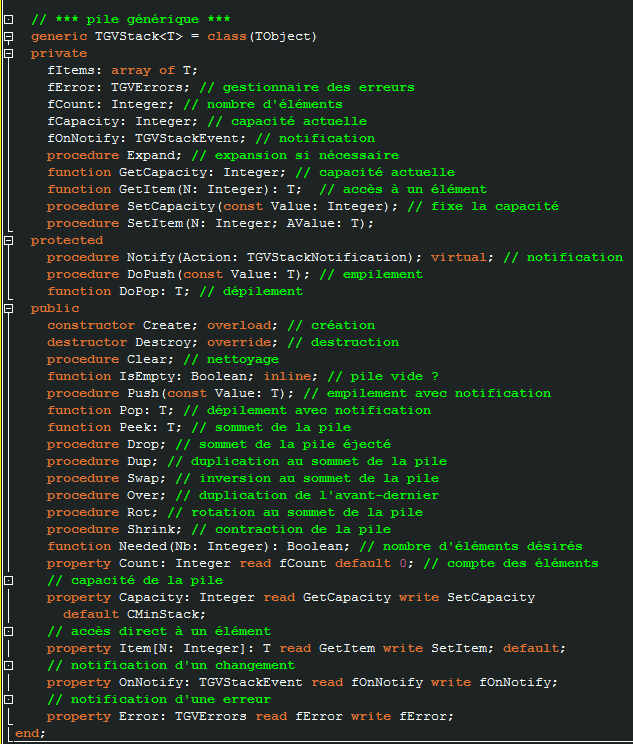
Les avantages évidents de cette méthode sur celle plus classique de définitions de classes au cas par cas sont un gain de temps, une souplesse largement accrue, une sécurité renforcée puisque le code n’a pas à être ressaisi pour chacune des classes et la possibilité de modifier le comportement de toutes les classes spécialisées à partir de la seule modification de la classe générique :



Les piles spécialisées prédéfinies sont les suivantes :



En ce qui concerne l’interface de la classe TGVStack, en voici le listing :



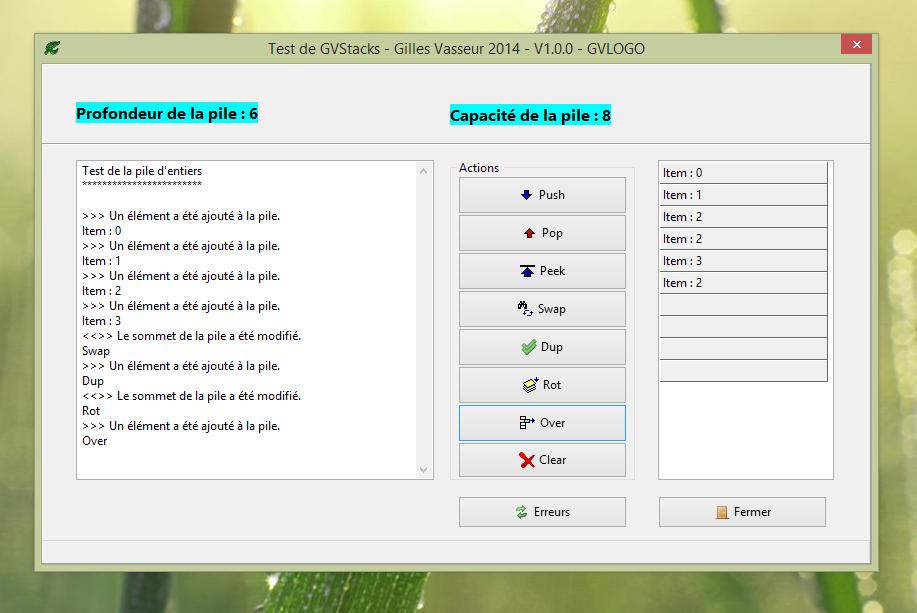
Cette interface appelle les remarques suivantes :

* le paramètre **T** qui apparaît souvent sera celui remplacé par le type réel mis en œuvre par la classe spécialisée ;
* le fondement de la pile est le tableau ouvert fItems qui verra son dernier élément pointé par le champ privé fCount ;
* la notion de capacité (à travers le champ privé fCapacity) complique un peu la classe, mais permet d’accélérer les empilements en ne procédant pas sans cesse à des allocations de mémoire ;
* la propriété Item est définie par défaut, ce qui signifie par conséquent que l’utilisateur n’a pas besoin de la spécifier  (les deux écritures sont équivalentes : MaPile.Item[2] et Mapile[2]) ;
* la plupart des méthodes définies sont destinées à manipuler la pile : Push, Peek, Pop, Drop, Dup, Swap, Over et Rot. Les programmeurs en Forth reconnaîtront là des outils très familiers ;
* la présence des méthodes DoPush et DoPop se justifie par le fait que ces opérations de base sur la pile sont employées par les autres méthodes qui ne doivent déclencher l’événement OnNotify qu’une seule fois : ce sont donc deux méthodes qui ne notifient rien, contrairement à Push et à Pop.

#### **Test de l’unité TGVStacks**

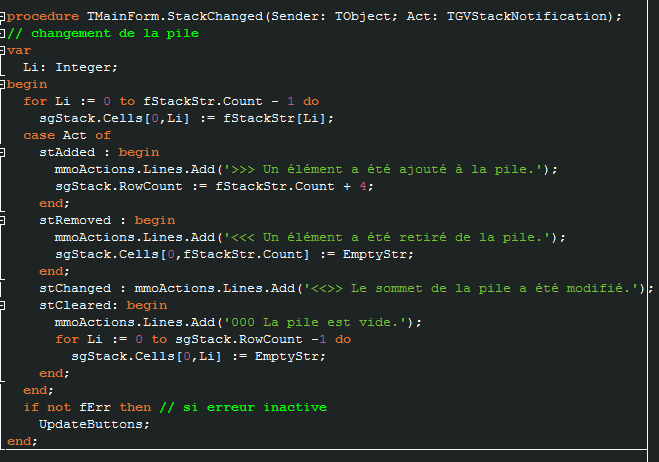
Pour ne pas changer, le programme de test est décliné en deux versions :

* Lazarus pour Windows 32 ;
* Lazarus pour Linux.



**Figure 32 - Test des piles avec Lazarus**

La méthode StackChanged exige quelques commentaires :

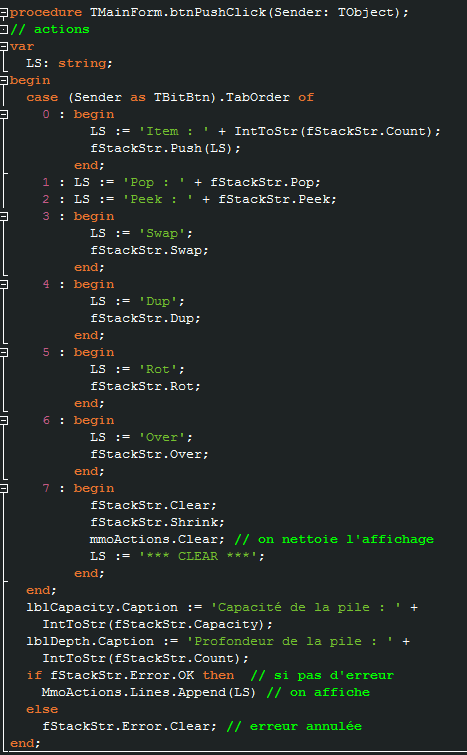


Cette méthode a été affectée à la propriété OnNotify de la pile. Elle remplit le composant **TStringGrid** avec les éléments de la pile, répartit son travail suivant l’action notifiée et met à jour les boutons qui seront actifs ou non selon la profondeur de la pile. On remarquera la mise à jour permanente de la **TStringGrid** afin de ne pas laisser de chaînes visibles après un dépilement, et l’augmentation si nécessaire du nombre de lignes disponibles sans quoi une exception serait déclenchée pour un index hors limites.

Une autre méthode peut poser problème : il s’agit de celle appliquée lorsque l’utilisateur presse un bouton (btnPushClick). Cette méthode est partagée par tous les boutons de la fenêtre. Afin de déterminer quel bouton a été pressé, on se sert de la propriété TabOrder qui renvoie l’ordre du composant si l’utilisateur utilise la touche de tabulation.

On utilise plutôt la propriété particulière Tag qui a le mérite de ne pas dépendre de l’ordre des composants, mais qui ne vérifie pas qu’elle est unique. Avec TabOrder, on devra revoir le fichier source si l’on modifie l’ordre des composants ou si l’on en ajoute/supprime un. Ici, le choix de TabOrder ne se justifie que par le parti pris pédagogique de montrer que plusieurs solutions existent quand il s’agit de résoudre un problème !

En voici la définition :



Une chaîne LS est préparée avant d’être ajoutée au composant **TMemo**. Comme la taille de la pile peut être modifiée, les composants **TLabel** correspondants sont ajustés.

Il pouvait être plus judicieux de placer cette dernière mise à jour dans la méthode StackChanged qui couvre automatiquement tous les changements de la pile. Le lecteur pourra s’amuser à réaliser ce petit changement. Toutefois, il faut prendre garde de ne pas alourdir démesurément les gestionnaires d’événements qui sont susceptibles d’être appelés de manière très fréquente : on observerait alors un ralentissement général des performances du programme.

## 09 - L’évaluation d’une expression mathématique

### Définition

### Opérations dans l’évaluateur

### Implémentation de l’évaluateur

# Les composants du langage

## Le noyau

## Les variables

### Les variables globales

### Les variables locales

## Les procédures

## Les paquets

## Les primitives

## L’interpréteur

# Le programme final

## L’organisation du logiciel

## L’interface utilisateur

## Ce qu’il reste à faire

# Mode d’emploi de GVLOGO

## Installer GVLOGO

## Interface et menus

### Menu Fichier

### Menu Editer

### Menu Voir

### Menu Exécuter

### Menu Fenêtres

### Menu Aide

### Les boutons

## Créer un programme

## Exécuter un programme

## Sauvegarder et récupérer un programme

## Modifier un programme

## Les messages d’erreur

## Le débogage

## Mettre à jour le logiciel

# Programmes exemples

## Travailler avec la tortue

## Travailler avec les listes

## Travailler avec les listes de propriétés

## Manipuler l’espace de travail

# Table des figures

[Figure 1 - exemple de GVLOGO en action 4](#_Toc409557541)

[Figure 2 - exemple de GVLOGO en action (2) 6](#_Toc409557542)

[Figure 3 - exemple de GVLOGO en action (3) 7](#_Toc409557543)

[Figure 4 -Test de TGVWords avec Lazarus (Windows) 29](#_Toc409557544)

[Figure 5 - Test de TGVLists avec Delphi 40](#_Toc409557545)

[Figure 6 - Test de TGVLists avec Lazarus 40](#_Toc409557546)

[Figure 7 - Test des chaînes avec Lazarus 41](#_Toc409557547)

[Figure 8 - Test des listes de propriétés avec Delphi 47](#_Toc409557548)

[Figure 9 - Test des listes de propriétés avec Lazarus 48](#_Toc409557549)

[Figure 10 - Tortue avec cap 90 Tortue 180 Tortue 90 png 49](#_Toc409557550)

[Figure 11 - Point avec sinus et cosinus 60](#_Toc409557551)

[Figure 12 - Arc Tangente 61](#_Toc409557552)

[Figure 13 - Test de la tortue avec Delphi 73](#_Toc409557553)

[Figure 14 - Test de la tortue avec Lazarus 74](#_Toc409557554)

[Figure 15 - Test de GVTurtles2 (Lazarus) 75](#_Toc409557555)

[Figure 16 - Écran principal de EasyTurtle (Windows) 79](#_Toc409557556)

[Figure 17 - Actions de la tortue (EasyTurtle) 80](#_Toc409557557)

[Figure 18 - Couleurs et formes (EasyTurtle) 81](#_Toc409557558)

[Figure 19 - Ordres (EasyTurtle) 81](#_Toc409557559)

[Figure 20 - Aide (EasyTurtle) 82](#_Toc409557560)

[Figure 21 - A propos (EasyTurtle) 83](#_Toc409557561)

[Figure 22 - Outils (EasyTurtle) 83](#_Toc409557562)

[Figure 23 - Barre de statut (EasyTurtle) 84](#_Toc409557563)

[Figure 24 - Répertoire de EasyTurtle. 85](#_Toc409557564)

[Figure 25 - Appel d'une autre fiche (EasyTurtle). 85](#_Toc409557565)

[Figure 26 - Mémorisation des ordres (EasyTurtle). 86](#_Toc409557566)

[Figure 27 - La tortue avance (EasyTurtle). 86](#_Toc409557567)

[Figure 28 - Mise à jour (EasyTurtle). 86](#_Toc409557568)

[Figure 29 - Rejouer une séquence (EasyTurtle). 87](#_Toc409557569)

[Figure 30 - Annuler la dernière action (EasyTurtle). 88](#_Toc409557570)

[Figure 31 - Sauvegarde (EasyTurtle). 89](#_Toc409557571)

[Figure 32 - Chargement d'un fichier (EasyTurtle). 90](#_Toc409557572)

[Figure 33 - Demande de fermeture (EasyTurtle). 91](#_Toc409557573)

[Figure 34 - Notion de pile LIFO 93](#_Toc409557574)

[Figure 35 - Notion de queue 96](#_Toc409557575)

[Figure 36 - Test des piles avec Delphi 100](#_Toc409557576)

[Figure 37 - Test des piles avec Lazarus (Windows) 101](#_Toc409557577)

[Figure 38 - Test des piles avec Lazarus (Linux) 101](#_Toc409557578)

1. *Cf.* <http://notepad-plus-plus.org/fr/> [↑](#footnote-ref-1)
2. *Cf.* <http://geany.org/> [↑](#footnote-ref-2)
3. *Cf*. <http://lazarus.developpez.com/telecharger/detail/id/433/Bibliotheque-BGRABitmap> [↑](#footnote-ref-3)
4. *Cf*. <http://www.freepascal.org/docs-html/ref/refse11.html> [↑](#footnote-ref-4)
5. Il s’agit de la notation scientifique pour 102, soit 100. [↑](#footnote-ref-5)
6. Le tri des caractères accentués sera correct avec Delphi, contrairement à Lazarus qui les classera après les caractères normaux. [↑](#footnote-ref-6)
7. L’espace séparant les éléments d’une liste, une suite d’espaces sera par conséquent ignorée. [↑](#footnote-ref-7)
8. On reviendra sur les expressions lors de l’étude de leur évaluation. [↑](#footnote-ref-8)
9. Le tri des caractères accentués sera correct avec Delphi, contrairement à Lazarus qui les classera après les caractères normaux. [↑](#footnote-ref-9)
10. On remarquera que les primitives concernant les piles et les queues ne sont pas implémentées dans cette unité : elles le seront dans l’unité **TGVStacks**. [↑](#footnote-ref-10)
11. Ce caractère est celui qui figure sous le 6 du pavé alphanumérique. Il s’obtient en pressant la touche AltGr et la touche du 6. Il est choisi grâce à la propriété NameValueSeparator. [↑](#footnote-ref-11)
12. En Pascal, les graphiques ont des coordonnées inversées par rapport aux habitudes en mathématiques. Ainsi le coin supérieur gauche a pour abscisse 0 et ordonnée 0. Plus l’ordonnée augmente plus le point visé se place vers le bas de l’écran. [↑](#footnote-ref-12)
13. Pour les couleurs admises, se référer au tableau n°1 fourni avec FIXE.COULEUR.FOND. [↑](#footnote-ref-13)
14. Le premier point est celui situé en haut à gauche tandis que le second est celui dessiné en bas à droite. [↑](#footnote-ref-14)
15. Un *sprite* (parfois appelé lutin en français) est un objet qui se déplace sur une surface. On doit avoir l’impression qu’il glisse sur cette surface. [↑](#footnote-ref-15)
16. … et deux fois plus rapide qu’un appel successif aux fonctions voulues ! [↑](#footnote-ref-16)
17. La détermination des cas a été établie dans une ancienne FAQ concernant Delphi dont je ne trouve plus trace… Par ailleurs, il existe une seconde version de Towards qui travaille avec une liste en entrée. [↑](#footnote-ref-17)
18. CoordX et CoordY sont les coordonnées actuelles de la tortue. [↑](#footnote-ref-18)
19. Voir le dossier « EasyStack » qui contient un exemple de programme avec Delphi : on remarquera qu’en quelques lignes les extensions désirées sont implémentées. [↑](#footnote-ref-19)