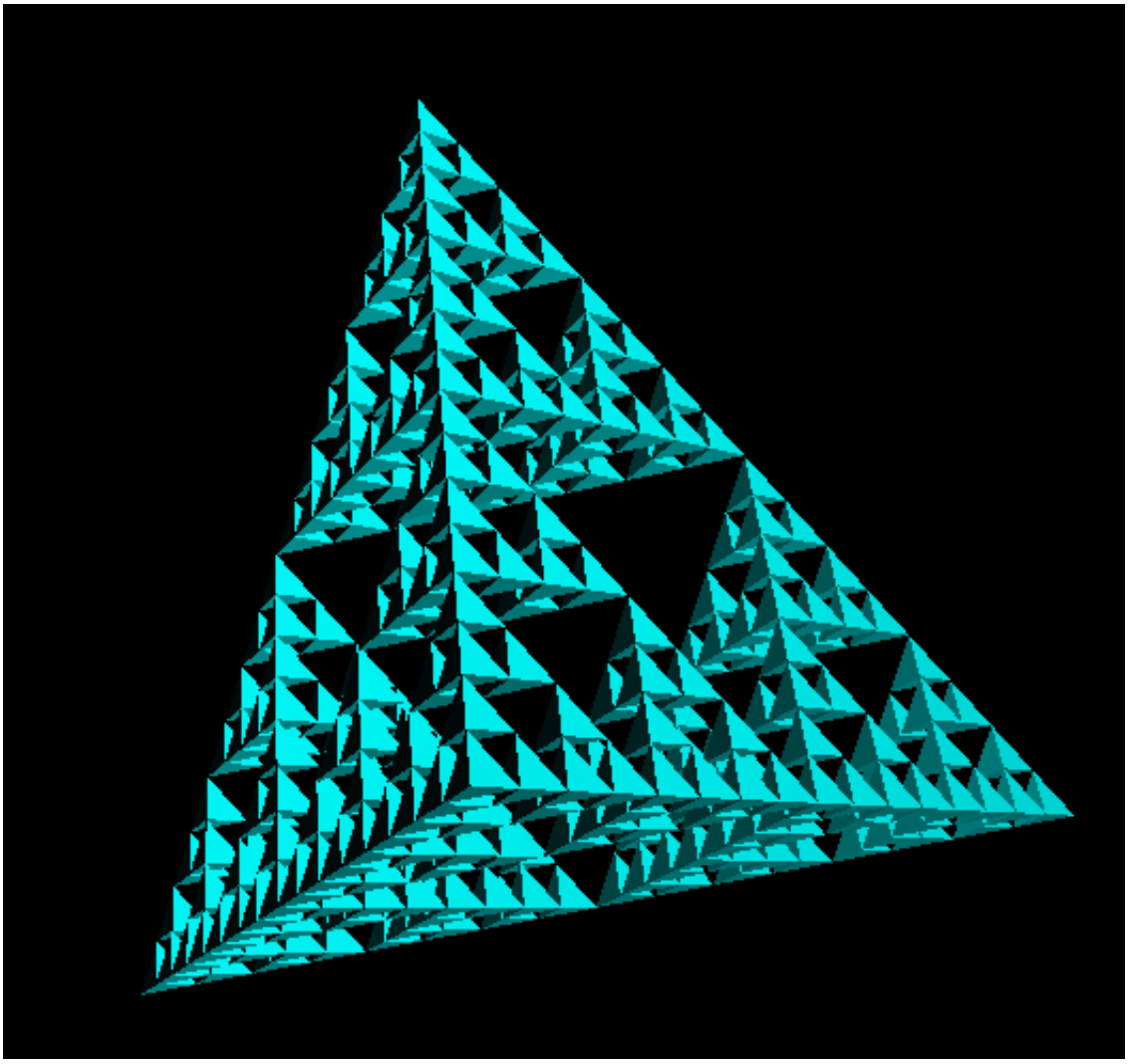
XLogo: Manuel de référence

Loïc Le Coq

8 juillet 2009



http://xlogo.tuxfamily.org

Introduction

Le Logo est un langage qui a été développé dans les années 60 par Seymour Papert. Celui-ci s'appuyait sur une théorie originale de l'apprentissage, appelée le constructionisme, dont on peut résumer le concept en anglais par l'expression « *learning-by-doing* ».

Le langage Logo permet de développer incontestablement certaines compétences mathématiques et logiques, c'est un excellent langage pour débuter avec la programmation et apprendre les rudiments tels que les boucles, les tests, les procédures... L'utilisateur peut déplacer un objet appelé « tortue » sur l'écran à l'aide de commandes aussi simples que `avance`, `recule`, `tournedroite` et autres. A chaque déplacement, la tortue laisse un trait derrière elle et ainsi on peut créer des dessins. Le fait de pouvoir donner les instructions dans un langage quasiment usuel facilite grandement l'apprentissage. Une utilisation plus avancée est aussi possible, on peut ainsi manipuler des objets tels que les listes, les mots ou encore les fichiers.

Logo est un langage interprété : c'est à dire que les commandes écrites par l'utilisateur sont exécutées immédiatement par l'ordinateur. On se rend compte directement lors du déroulement du programme des erreurs commises ce qui favorise l'apprentisage.

XLogo est donc un interpréteur du langage Logo. L'adresse du site principal de l'application est :

`http ://xlogo.tuxfamily.org/`

Vous pourrez télécharger le logiciel et la documentation. Une gallerie de multiples exemples permet de mieux cerner les aptitudes du logiciel.

XLogo supporte actuellement 10 langues (anglais, allemand, arabe, asturien, espagnol, espéranto,français, galicien, grec et portugais) et a été écrit en Java. Ce langage présente l'avantage d'être multiplateforme c'est à dire que l'application XLogo tournera indépendamment du système d'exploitation installé. Que vous soyez sous Linux, sous Windows ou encore sous MAC, pas de problèmes, la petite tortue s'offre à vous !

XLogo est placé sous licence GPL :

C'est donc un logiciel libre ce qui garantit à l'utilisateur :

1. la liberté d'exécuter le logiciel, pour n'importe quel usage ;
2. la liberté d'étudier le fonctionnement d'un programme et de l'adapter à ses besoins, ce qui passe par l'accès aux codes sources ;
3. la liberté de redistribuer des copies ;
4. la liberté d'améliorer le programme et de rendre publiques les modifications afin que l'ensemble de la communauté en bénéficie.

Structure du manuel :

Ce manuel va vous permettre de découvrir XLogo.

- La première partie sera consacrée à la description de l'interface et des différents menus.
- Ensuite, une série de chapitres vous présentera les premières instructions de base de XLogo. La difficulté dans l'enchaînement des notions se veut graduée. Des exercices d'application sont proposés en fin de chapitre, vous trouverez leurs corrigés en annexe.

- Enfin, une série de thèmes particuliers est abordée pour les utilisateurs avancés.
- En annexe, vous trouverez notamment la description de la totalité des primitives ainsi que les différentes options de lancement de XLogo.

Ce manuel est disponbile sous divers formats :

- PDF : http ://downloads.tuxfamily.org/xlogo/downloads-fr/manual-fr.pdf
- HTML zippé : http ://downloads.tuxfamily.org/xlogo/downloads-fr/manual-html-fr.zip
- LaTeX2$_\varepsilon$: Source du manuel : http ://downloads.tuxfamily.org/xlogo/downloads-fr/manual-src-fr.zip
- JavaHelp : Via le menu Aide-Manuel en ligne de XLogo

Table des matières

Chapitre 1

Installation de XLogo

- Premièrement, il vous faut installer un environnement d'exécution JAVA sur votre ordinateur. Rendez-vous sur cette page :
  `http ://java.sun.com/javase/downloads/index.jsp`
  Télécharger le JRE (Java Runtime Environment) correspondant à votre systèmee d'exploitation (windows, Linux ...) puis installez-le.
- Deuxièmement, il faut télécharger le fichier `xlogo.jar` situé à l'adresse :
  `http ://downloads.tuxfamily.org/xlogo/common/xlogo.jar`
  Sinon, plus simplement, rendez-vous sur le site de XLogo, à l'adresse `http ://xlogo.tuxfamily.org` puis choisir le français et le menu téléchargement.

1.1 Configuration de XLogo

1.1.1 Environnement Linux

Sous Ubuntu 8.04 :

1. Pour installer JAVA :
   - Système -> Administration -> Gestionnaire de paquets Synaptic
   - Installer le paquet `sun-java6-jre`
2. Pour ouvrir le fichier `xlogo.jar` par double-clic :
   - Clic droit sur `xlogo.jar`, Propriétés
   - Onglet « Ouvrir avec » : Choisir Sun Java Runtime
3. Asscocier les extensions `lgo` à XLogo :
   - Clic droit sur `xlogo.jar`, Propriétés
   - Onglet « Ouvrir avec » :
   - Bouton « Ajouter »
   - Dans « Utiliser une comande personnalisée : », taper :
     `java -jar chemin_vers_xlogo.jar`

Remarque : XLogo est inclus dans la distribution OpenSuse.

1.1.2 Environnement Windows

En principe, si vous double-cliquez sur l'icône de XLogo, le logiciel doit se lancer. Si c'est le cas, passer au paragraphe suivant. Sinon, c'est qu'un autre programme prend en charge les fichiers d'extension « `jar` » (Souvent, des programmes de décompression, style WinZip et autres).

Voilà comment associer le programme « `java` » aux fichiers d'exetnsion « `jar` ». (Certains chemins peuvent différés selon que vous possédiez Windows 98, 2000, XP ...)

1. Démarrer–> Paramètres —> Options des dossiers ...
2. Cliquez ensuite sur l'onglet « Types de fichiers » (le $3^{\text{ème}}$).

3. Repérez alors dans la liste des options proposées celles se rapportant au fichiers JAR (Fichiers JAR, Fichiers exécutables JAR, archive JAR ...)
4. Sélectionner ce type de fichiers et cliquez sur « Modifier... »
5. Une nouvelle fenêtre apparaît alors choisissez « Modifier... »
6. Séectionner alors « Parcourir... »
7. Il faut indiquer le chemni vers `javaw.exe`, en principe

   `c :\Program Files\java\j2re1.4.1\bin\javaw.exe`
8. Une fois ceci fait, il apparaît dans le champ Application utilisée pour réaliser l'action :

   `c :\Program Files\java\j2re1.4.1\bin\javaw.exe`

   Il faut alors rajouter au bout :

   `"c :\Program Files\java\j2re1.4.1\bin\javaw.exe" -jar "%1" %*`

   (Attention il faut un espace de chaque côté de -jar)
9. Ensuite, il n'y a plus qu'à refermer toutes les fenêtres puis double-cliquer sur l'icône de XLogo.

Si ca ne fonctionne toujours pas, il y a une deuxième possiblité : Vous ouvrez une console MSDOS (Démarrer –> Programmes –> Commandes MSDOS ou Démarrer—> Programmes—-> Accessoires—-> Invite MSDOS) puis vous tapez l'instruction suivante :

`java -jar le_chemin_ou_se_trouve_le_fichier`

Par exemple : `java -jar c :\xlogo\xlogo.jar`

Si cela vous ennuie de systématiquement devoir taper cette instruction. Taper là dans un fichier texte et enregistrer le par exemple sous le nom de `xlogo.bat`. Il ne reste plus qu'à double cliquer sur `xlogo.bat` pour lancer XLogo.

Associer les fichiers d'extension `lgo` avec XLogo

Je n'ai pas réussi à configurer cela sous Vista. (Je n'ai pas trop cherché non plus... avis aux amateurs ! Merci de me communiquer la solution)

En principe, les fichiers d'extension `.lgo` ne sont pas reconnus par votre ordinateur, lorsque vous double-cliquez dessus, une boîte de dialogue apparaît vous demandant quelle application il faut utiliser pour ouvrir le fichier.

- Indiquer « Autre »puis indiquer le chemin vers l'application `javaw.exe`
  Généralement, `c :\Program Files\java\j2re1.4.1\bin\javaw.exe`
- Doner un nom pour désigner les fichiers d'extension `lgo`.
  Par exemple : Fichiers Logo
- Démarrer -> Paramèters -> Options des dossiers
- Onglet « Type de Fichiers »
- Repérer dans la liste les fichiers `lgo`
- Sélectionner ce type de fichiers puis cliquer sur « Modifier »
- Une nouvelle fenêtre apparaît, Encore « Modifier »
- Dans le champ « Application utilisée pour réaliser l'action »,
  `"c :\Program Files\java\j2re1.4.1\bin\javaw.exe" -jar xlogo.jar "%1" %*`
- Reste à fermer les fenêtres

1.2 Mises à jour

`http ://xlogo.tuxfamily.org/rss.xml`

Pour mettre à jour XLogo, il suffit de remplacer le fichier `xlogo.jar` par sa nouvelle version. Si vous souhaitez être prévenu de la parution de chaque nouvelle version, ou de chaque amélioration, il est possible de s'abonner au fil RSS de XLogo. L'adresse du fill RSS est :

`http ://xlogo.tuxfamily.org/rss.xml`

Il existe plusieurs logiciels permettant de gérer les fils RSS, si vous n'êtes pas coutumier de cette technique, le plus simple est d'utiliser Mozilla Thunderbird :

- Menu Edition - Paramètre des comptes
- Bouton « Ajouter un compte »
- « Nouvelles RSS et Blogs »
- Nom du compte : « Fils RSS » par exemple
- Boutons « Suivant » et « Terminer »
- Dans la fenêtre « Paramètre des comptes », sélectionner alors « Fils RSS » dans le menu de gauche puis cliquer sur le bouton « Gérer les abonnements ».
- Bouton « Ajouter »
  - URL du fil : `http ://xlogo.tuxfamily.org/rss.xml`
  - Cocher la case « Aficher le résumé de l'article plutôt que de télécharger la page Web »

Voilà, avec le bouton « Envoyer-Recevoir », vous recevrez les nouvelles de XLogo de la même façon que vous gérez vos mails.

1.3 Désinstallation

Pour désinstaller XLogo, il suffit de supprimer le fichier `xlogo.jar` et le fichier de démarrage `.xlogo` (il est situé dans votre répertoire utilisateur c'est à dire `/home/votre_login` pour les linuxiens ou `c :\windows\.xlogo`

Chapitre 2

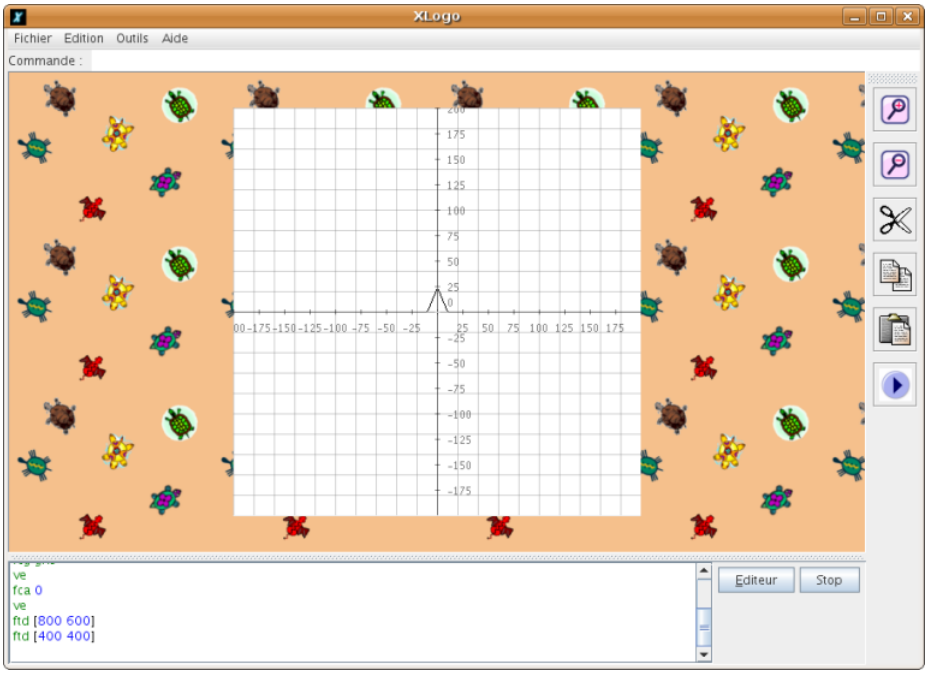
Présentation de l'interface :

2.1 Au premier lancement

La première fois que vous lancez XLogo (ou si vous avez effacé le fichier `.xlogo` - Voir section 1.3), une boîte de dialogue apparaîtra pour vous permettre de choisir la langue utilisée.

Ce choix n'est pas définitif bien sûr, il peut être corrigé ensuite à l'aide de la boîte de dialogue de Préférences. (Voir section 3.3)

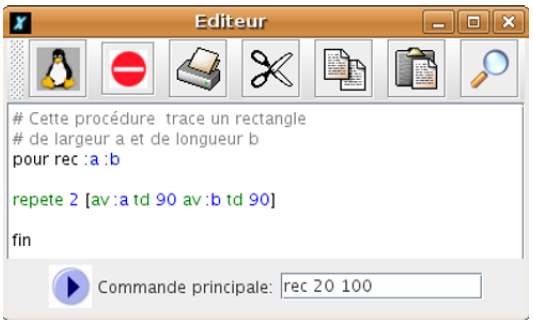
2.2 Fenêtre principale



- En haut, les traditionnels menus **Fichier, Edition, Outils et Aide**
- Juste en dessous, **la ligne de commande** qui permet de saisir les instructions logo.
- Au centre, **la zone de dessin**.
- A la droite de la zone de dessin, **une barre d'outils** vous permet de réaliser divers actions :

  - Zoom avant/arrière.
  - Diverses fonctionnalités d'édition (couper/copier/coller).
  - Le bouton « Lecture » permet de lancer la commande principale définie dans l'éditeur.
- En bas, **la zone « historique »** qui rappelle toutes les dernières commandes tapées et les réponses associées. Pour rappeler rapidement une instruction déjà tapée, il y a deux solutions : ou bien vous cliquez sur l'ancienne instruction dans l'historique, ou bien vous appuyez plusieurs fois sur la flèche du haut jusqu'à ce que l'instruction désirée apparaisse. Les deux flèches haut et bas permettent en fait de se déplacer dans toute l'historique des commandes tapées précédemment (Très pratique).
- A la droite de l'historique, deux boutons : **STOP et EDITEUR** .
  - Le bouton STOP interrompt toute exécution en cours.
  - Le bouton EDITEUR permet d'ouvrir l'éditeur de procédures.

2.3 L'éditeur de procédures



Pour ouvrir l'éditeur, trois possibilités :

- Taper `ed` dans la ligne de commandes. L'éditeur s'ouvrira alors avec toutes les procédures déjà définies. Si vous ne souhaitez éditer que certaines procédures particulières, taper alors :
  `ed [procedure_1 procedure_2 ...]`
- Appuyer sur le bouton Editeur de la fenêtre principale.
- Utiliser le raccourci clavier Alt+E

Voici les différents boutons que vous trouverez dans l'éditeur :

Sauve les modifications apportés au contenu de l'éditeur puis ferme celui-ci. C'est sur ce bouton qu'il faut appuyer à chaque fois que vous voulez enregistrer les nouvelles procédures tapées. Si vous le préférez, vous pouvez utiliser le raccourci clavier ALT+Q.

Quitte l'éditeur en n'enregistrant aucune des modifications apportées à celui-ci. On peut également utiliser le raccourci ALT+C.

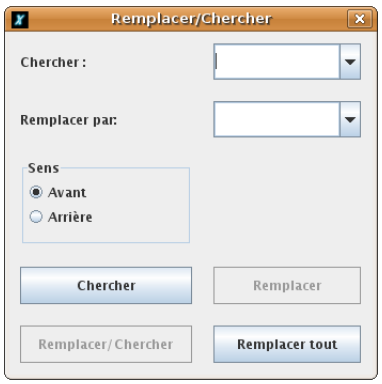
Imprime le contenu de l'éditeur.

Copie le texte sélectionné dans le presse-papiers

Coupe le texte sélectionné dans le presse-papiers

Colle le texte sélectionné dans le presse-papiers

Ouvre une boîte de dialogue permettant de chercher ou de remplacer du texte dans l'éditeur.



 Tout en bas de l'éditeur, un champ texte permet de définir une commande principale. Celle-ci représente la commande générale qui permet de lancer un programme. Elle est accessible via le bouton « lecture » de la barre d'outils de la fenêtre principale. Lorsqu'on sauve le contenu de l'éditeur dans un fichier au format `.lgo`, cette commande est également enregistrée

IMPORTANT :

- Cela ne sert à rien d'appuyer sur la croix en haut à droite pour fermer la fenêtre ! Seuls les deux premiers boutons vous permettent de quitter l'éditeur.
- Pour effacer une ou plusieurs procédures indésirables, utiliser la primitive `efp, effaceprocedure` ou alors utiliser dans la barre de menus **Outils - Gestionnaire de procédures**.

2.4 Quitter

Pour quitter XLogo, dans la barre de menu, **Fichier - Quitter** ou alors cliquer sur la croix de fermeture de la fenêtre. Une boîte de dialogue de confirmation apparaît à ce moment.

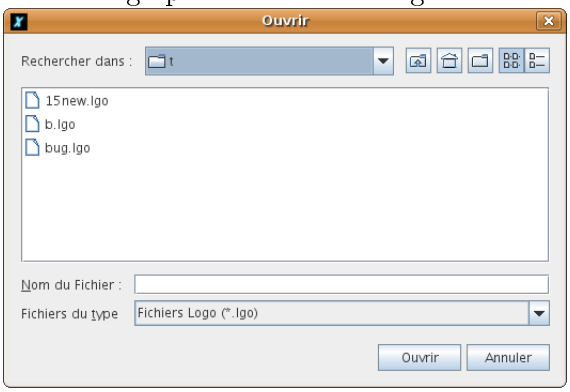
Chapitre 3

Options des menus :

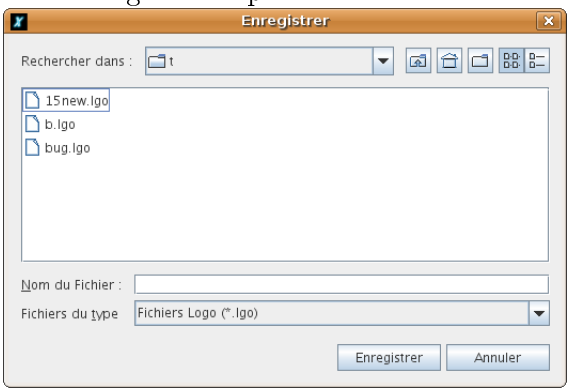
3.1 Menu « Fichier »

- **Fichier→Nouveau** : Détruit l'ensemble des procédures et variables définies pour créer ainsi un nouvel espace de travail.

- **Fichier→Ouvrir** : ouvre un fichier logo précédemment enregistré.



- **Fichier→Enregistrer sous ...** : enregistre les procédures en cours sous un nom précis.



- **Fichier→Enregistrer** : enregistre les procédures dans le fichier actuellement utilisé.

- **Fichier→Capturer l'image→Enregistrer l'image sous...** : permet d'enregistrer l'image sous le format jpg ou png. Si vous souhaitez sélectionner seulement une partie de l'image, vous avez la possibilité de définir un rectangle de sélection en faisant glisser la souris sur la zone de dessin.

- **Fichier→Capturer l'image→Imprimer l'image** : permet d'imprimer l'image. De même que précédemment, vous pouvez sélectionner une zone précise à imprimer.

- **Fichier→Capturer l'image→Copier l'image dans le presse-papier** : Permet d'envoyer l'image dans le presse-papier système. De même que pour l'impression et l'enregistrement, vous pouvez ne

sélectionner qu'une zone de l'image. Cette fonctionnalité fonctionne très bien sous les environnements de type Windows. En revanche, elle ne marche pas sous Linux (Le presse-papier n'a pas le même type de fonctionnement). Non testé sous Mac.

- **Fichier→Zone de texte→Enregistrer au format RTF** : Permet d'enregistrer la zone d'historique au format RTF (conserve les couleurs et le formatage des caractères).

- **Fichier→Quitter** : quitte l'application XLOGO.

3.2 Menu « Edition »

- **Edition→Copier** : copie le texte sélectionné dans le presse-papiers.

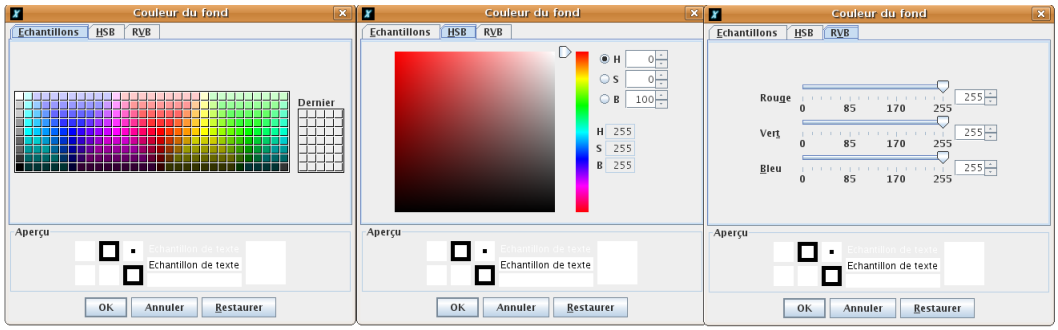
- **Edition→Couper** : coupe le texte sélectionné dans le presse-papiers.

- **Edition→Coller** : colle le texte contenu dans le presse-papiers dans la ligne de commande.

- **Edition→Sélectionner tout** : Sélectionne l'ensemble du texte de la zone de commande.

3.3 Menu « Outils »

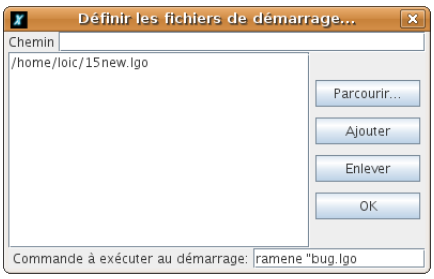
- **Outils→Choisir la couleur du crayon** : permet de choisir la couleur avec laquelle écrit la tortue à l'aide d'une palette de couleurs.



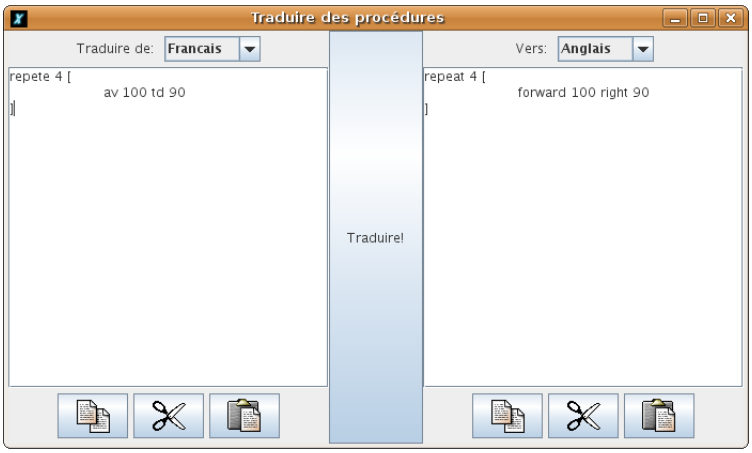
Disponible également avec la primitive `fcc` (Voir annexe A.1.2).

- **Outils→Choisir la couleur du fond** : même chose avec le fond d'écran. Disponible avec la primitive `fcfg`. (Voir annexe A.1.2).

- **Outils→Définir les fichiers de démarrage** : permet de définir des chemins vers des fichiers dit « de démarrage ». Toutes les procédures contenues dans ces fichiers au format *.lgo deviendront alors des « pseudo-primitives » du langage XLogo. Elles ne sont pas éditables ni modifiables par l'utilisateur. Vous pouvez ainsi définir des primitives personnalisées. Vous pouvez de plus lui donner une commande (en logo) à effectuer au démarrage de XLogo. Vous avez ainsi la possibilité de lancer un programme que vous avez conçu dès l'ouverture de XLogo.



- **Outils→Traduire des procédures** : Ouvre une boîte de dialogue qui permet de traduire des commandes XLogo dans une langue désirée. (Très utile en particulier lorsqu'on récupère des sources Logo en anglais sur internet pour les remettre en français)

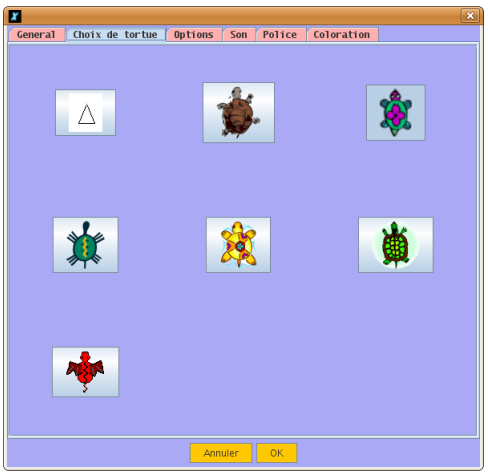


- **Outils→Gestionnaire de procédures** : Ouvre une boîte de dialogue permettant d'effacer des procédures. Elle permet également de changer l'ordre d'apparition des procédures dans l'éditeur.

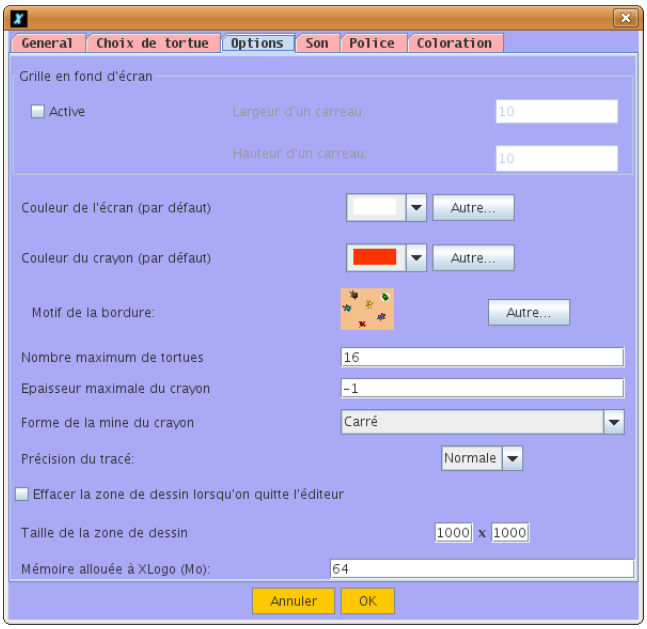
- **Options→Préférences** : Ouvre une boîte de dialogue dans laquelle vous pouvez configurer plusieurs choses :
  - **Onglet général :**
    → **Langue** : permet de choisir entre le français, l'anglais, l'espagnol, le portugais, l'arabe, l'allemand et l'espéranto. Attention, les primitives changent d'une langue à, l'autre.

    → **Aspect :** permet de définir le « look » de la fenêtre XLogo. Soit style natif, style Java (Métal) ou style Motif

    → **Choisir la vitesse de défilement**. Si vous souhaitez voir tous les déplacement de la tortue, vous pouvez la ralentir à l'aide de la barre prévue à cet effet.

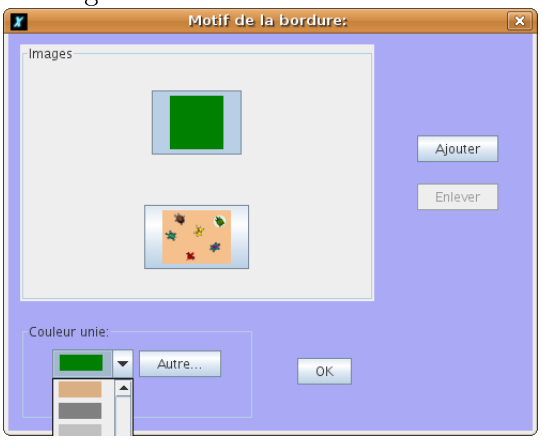


– **Onglet Choix de la tortue** : vous pouvez choisir votre tortue préférée.

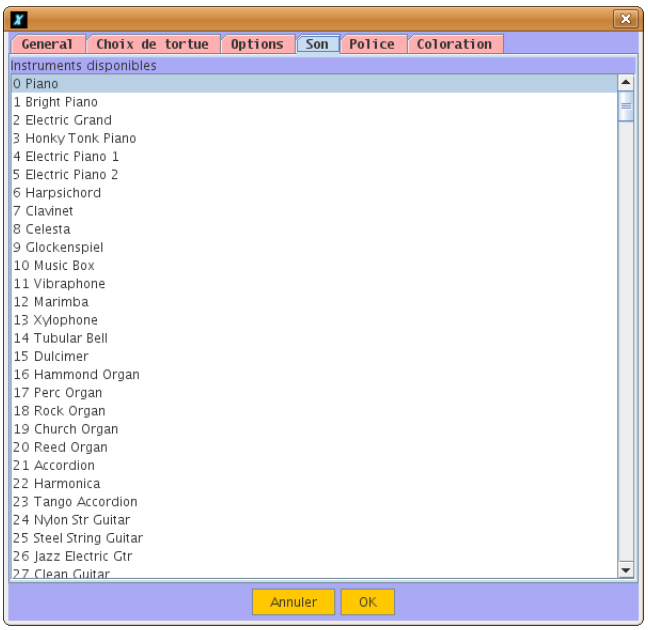


– **Onglet Options** : Plusieurs choses peuvent être fixées.
  → **Grille :** vous pouvez choisir de tracer une grille en fond d'écran. Il est possible de choisir la largeur et la hauteur d'un carreau de la grille ainsi que sa couleur.
  → **Axes :** vous pouvez choisir de tracer l'axe vertical ou l'axe vertical en fond d'écran. Vous pouvez définir la distance entre deux graduations ainsi que la couleur de chaque axe.
  → **Couleur de fond d'écran** : Possibilité de définir une couleur de fond d'écran par défaut.

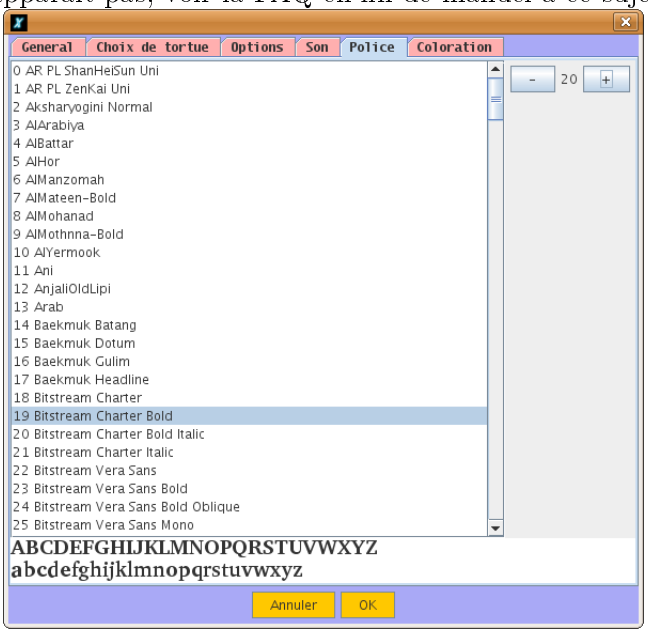
→ **Couleur de crayon** : Possibilité de définir une couleur de crayon par défaut.
→ **Motif de bordure** : Possibilité de définir un motif précis pour la bordure encadrant la zone de dessin (soit sous forme d'une image soit sous forme d'une couleur unie.)



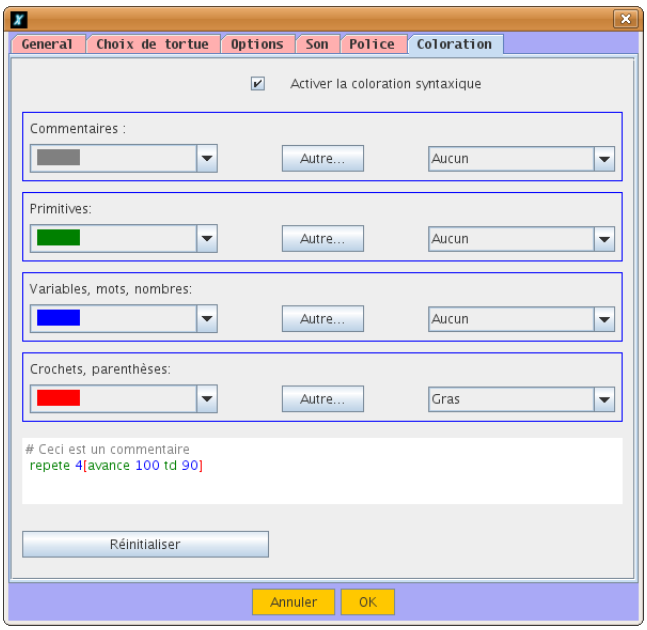
→ **Epaisseur du crayon** : On peut fixer une taille limite à l'épaisseur du crayon. Si l'on ne veut pas utiliser cette limitation, mettre le nombre -1 dans la zone de texte associée.
→ **Forme du crayon** : Ensuite, on peut choisir la forme du crayon de la tortue, on ne se rend compte du choix de cette option que lorsque l'on choisit une épaisseur de crayon supérieure à 1.
→ **Nombre maximal de tortues** : On peut également fixer le nombre de tortues maximum en mode multitortues. (Par défaut 16)
→ **Précision du tracé** : Vous pouvez choisir la qualité du tracé. En haute qualité, vous n'aurez plus d'effet de crénelage des lignes. En revanche, bien repérer qu'en augmentant la qualité, vous perdrez en rapidité d'exécution.
→ **Effacer la zone de dessin en sortie d'éditeur** : On peut choisir d'effacer automatiquement la zone de dessin lorsqu'on sort de l'éditeur.
→ **Effacer les variables en sortie d'éditeur** : Certains utilisateurs apprécient qu'à chaque changement dans l'éditeur, les variables globales soient automatiquement détruites. C'est possible en activant cette option.
→ **Taille de la zone de dessin** : Vous pouvez choisir une taille personnalisée pour la zone de tracé. Par défaut XLogo se lance avec une zone de 1000 pixels sur 1000 pixels. Attention, lorsque vous agrandissez l'image, il peut-être nécessaire d'augmenter la taille mémoire attribuée à XLogo. Un message d'erreur vous en avertira.
→ **Mémoire allouée à XLogo** : Vous pouvez par conséquent également changer la valeur correspondant à l'espace mémoire alloué à XLogo. Par défaut, cette valeur est fixée à 64 Mo. Il se peut que vous soyez obligé de l'augmenter si vous souhaitez travailler sur une zone de dessin plus grande. Lorsqu'on modifie ce paramètre, le changement n'est effectif qu'après redémarrage de XLogo. Attention, n'augmentez pas abusivement sans raison cette valeur, cela peut considérablement ralentir votre système.
→ **Numéro du port TCP :** Permet de choisir une valeur particulière pour le port utilisé lors des communications réseau. Voir p.128



– **Onglet Son** : vous trouverez la liste des instruments que peut imiter votre carte son au travers de l'interface MIDI. Vous pouvez sélectionner un instrument précis en cliquant dessus. (Vous pouvez également également sélectionner un instrument avec la primitive `fixeinstrument numéro`. Si la liste des instruments n'apparait pas, voir la FAQ en fin de manuel à ce sujet.



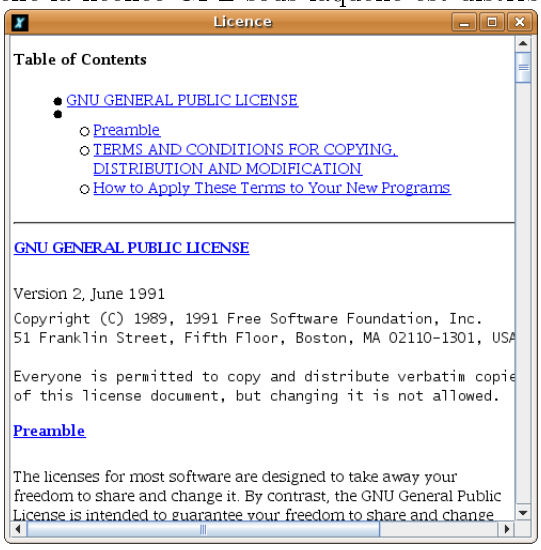
– **Onglet Police** : Dans le cinquième onglet, vous pouvez choisir la police de l'interface graphique ainsi que sa taille. Attention ceci n'affecte pas la police rendue par les primitives `ecris` et `etiquette`.



– **Onglet Coloration syntaxique** : Possibilité d'activer ou non la coloration syntaxique et de définir des couleurs personnalisées.

3.4 Menu « Aide »

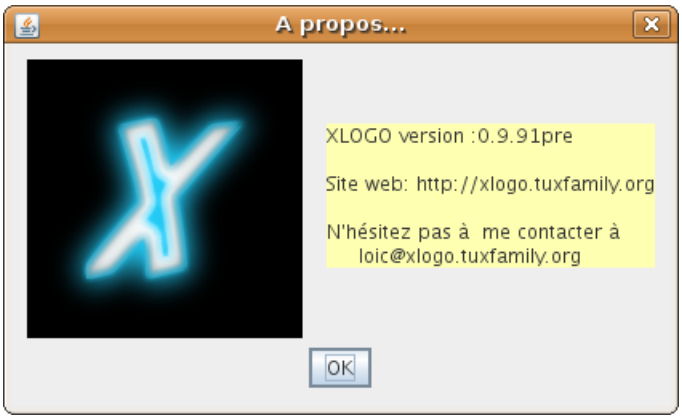
- **Menu Aide→Manuel en ligne** : Affiche le manuel de référence de XLogo, accessible uniquement avec une connexion internet.
- **Menu Aide→Licence** : Affiche la licence GPL sous laquelle est distribué ce logiciel.



- **Menu Aide→Traduction française** : affiche une traduction de ladite licence. Cette traduction n'a aucune valeur officielle, seule la version anglaise a ce rôle.
- **Menu Aide→Traduire XLogo** : Ouvre une boîte de dialogue permettant de consulter/modifier/compléter l'ensemble des traductions de XLogo (messages et primitives).

Il est également possible de créer les traductions pour une nouvelle langue. Dans chacun des cas le fichier généré est à envoyer à `loic@xlogo.tuxfamily.org`.

- **Menu Aide→A propos** : Classique .... et http ://xlogo.tuxfamily.org pour vos mises à jours ! ! o :)



Chapitre 4

Conventions adoptées dans XLOGO

Voici la présentation de certaines choses à savoir concernant le langage LOGO lui-même et d'autres concernant XLOGO spécifiquement.

4.1 Commandes et interprétation

Le langage LOGO est composé de commandes internes : On appelle ces commandes les **primitives**. Chaque primitive attend un certain nombre de paramètres que l'on appelle **arguments**. Par exemple la primitive `ve` qui permet d'effacer l'écran ne prend aucun argument alors que la primitive `somme` attend deux arguments.

`somme 2 3` écrira 5 en retour.

Les arguments sont de trois types en LOGO :

- **Les nombres :** certaines primitives attendent des nombres comme argument. Exemple `avance 100`
- **Les mots :** Les mots commencent tous par ". Un exemple de primitive pouvant travailler avec les mots est la primitive `ecris`.

  ```
  ecris "bonjour
  ```

  Cette commande provoque l'affichage du mot `bonjour` dans la zone de texte.
  A noter que si vous oubliez le ", l'interpréteur vous renverra un message d'erreur. En effet, `ecris` attend un argument, or pour l'interpréteur, `bonjour` ne représente rien puisque ce n'est ni un nombre, ni un mot, ni une liste ni une procédure déjà définie.
- **Les listes :** Elles sont définies entre crochets.

Remarque : Les nombres sont traités soit en tant que valeurs numériques, soit en tant que mots.
Exemple : `ecris premier 12` renvoie 1

Certaines primitives admettent une forme généralisée, c'est à dire qu'elles peuvent recevoir un nombre indéfini d'arguments. Voici la liste de ces primitives ci-dessous :

`ecris`	`somme`	`produit`	`ou`
`et`	`liste`	`phrase`	`mot`

Pour notifier à l'interpréteur que l'on va les utiliser sous leur forme généralisée, on tape la commande entre parenthèses, voici quelques exemples :

```
 ecris (somme 1 2 3 4 5)
15

(liste [a b] 1 [c d])
Que faire de [[a b] 1 [c d]]?

si (et 1=1 2=2 8=5+3) [av 100 td 90]
```

4.2 Procédures

En plus de ces primitives, vous pouvez définir vos propres commandes. On les appelle les *procédures*. Les procédures sont introduites à l'aide du mot `pour` et se terminent par le mot `fin`. On utilise l'éditeur de procédures interne à XLOGO pour les taper. Voici un petit exemple :

```
pour carre
repete 4[avance 100 tournedroite 90]
fin
```

Ces procédures ont le droit d'admettre également des arguments. Pour cela, on utilise des variables. Une variable est un mot auquel on peut affecter une valeur. voici un exemple très simple :

```
pour total :a :b
ecris somme :a :b
fin

total 2 3 -----> 5
```

4.3 Le caractère spécial « \ »

Le caractère « \ » (backslash) permet en particulier de créer des mots contenant des espaces ou contenant un retour à la ligne. « \n » provoque un retour à la ligne et « \␣ » désigne une espace dans un mot.
Exemple :

```
ecris "xlogo\ xlogo
xlogo xlogo
ecris "xlogo\nxlogo
xlogo
xlogo
```

Il s'ensuit que l'on ne peut plus accéder au caractère « \ » en le tapant il faudra taper « \\ ».

De même, les caractères « ( ) [ ] # » sont des délimiteurs du langage Logo qui ne peuvent être utilisés dans des mots. On pourra les introduire en rajoutant un caractère « \ » devant.

Tout caractère « \ » tout seul est ignoré. Cette remarque est très importante en particulier pour la gestion des fichiers

Pour fixer le répertoire courant à `C :\Mes Documents`, il faudra taper :

```
fixerepertoire "c:\\Mes\ Documents.
```

Noter l'utilisation du « \␣ » pour notifier l'espace entre « Mes » et « Documents ». Si d'autre part, vous omettez le double backclash, le chemin défini sera alors `c :Mes Documents` et l'interpréteur rendra un message d'erreur.

4.4 Règles concernant les majuscules et minuscules

XLogOne fait pas la différence majuscule-minuscule en ce qui concerne les noms de procédures et de primitives. Ainsi, avec la procédure `carre` définie précédemment, que vous tapiez `CARRE` ou `cArRe`, l'interpréteur de commande traduira correctement et exécutera `carre`. En revanche, XLogOrespecte les majuscules

dans les listes et les mots :

```
ecris "Bonjour ----> "Bonjour (on garde le B majuscule)
```

4.5 Opérateurs et syntaxe

Il y a deux façons d'écrire certaines commandes. Par exemple, pour effectuer l'addition de 4 et de 7, il y a deux possibilités :

- soit on se sert de la primitive `somme` qui attend deux arguments : on écrit `somme 4 7`
- soit on se sert de l'opérateur + : on écrit `4+7`.

Les deux ont le même effet. Voici la liste des correspondances entre opérateurs et primitives :

somme	difference	produit	divise
+	-	*	/
ou	et	egal ?	
| (ALT GR+6)	&	=	

Il existe également deux opérateurs de tests numériques ne correspondant à aucune primitive :

- Opérateur « Inférieur ou égal »`<=`
- Opérateur « Supérieur ou égal »`>=`

Attention : pas d'espace entre les symboles > et = !

Remarque : Les deux opérateurs | et & sont deux opérateurs spécifiques à XLOGO. Ils n'existent pas dans les versions traditionnelles de LOGO. Voici quelques exemples d'utilisation :

```
ec 3+4=7-1 ----> vrai
ec 3=4 | 7>=49/7 ----> vrai
ec 3=4 & 7=49/7 ----> faux
```

Chapitre 5

Découvrir les primitives de base

Niveau : débutant

Pour déplacer la tortue sur l'écran, on utilise des commandes prédéfinies appelées « primitives ». Dans ce chapitre, nous allons découvrir certaines primitives de base permettant de piloter la tortue.

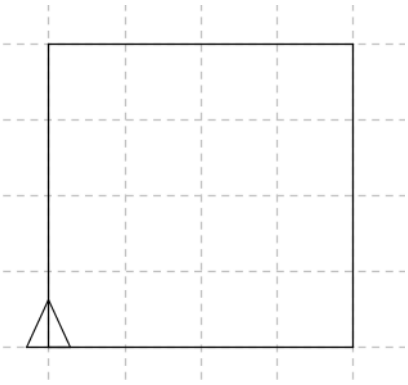
5.1 Primitives nouvelles utilisées :

1. `av nombre` `av 50`
   Fait avancer la tortue du nombre de pas de tortues indiqué.
2. `re nombre` `re 100`
   Fait reculer la tortue du nombre de pas de tortues indiqué.
3. `td nombre` `td 90`
   La tortue tourne vers la droite de l'angle indiqué.
4. `tg nombre` `tg 45`
   La tortue tourne vers la gauche de l'angle indiqué.
5. `ve` `ve`
   Efface l'écran et initialise la tortue au centre de l'écran.
6. `mt` `mt`
   La tortue est visible à l'écran.
7. `ct` `ct`
   Cache la tortue. Peut permettre de rendre l'affichage plus rapide.
8. `lc` `lc`
   Lève le crayon. La tortue ne laisse plus de trait derrière elle lorsqu'elle se déplace.
9. `bc` `bc`
   Baisse le crayon. La tortue écrit lorsqu'elle se déplace.
10. `repete nombre liste` `repete 5[av 50 td 45]`
    Répète les instructions contenues dans la liste le nombre de fois indiqué.

5.2 Dessiner un polygone régulier

Ici, nous allons apprendre à tracer un carré, un triangle équilatéral, un pentagone régulier etc

5.2.1 Le carré



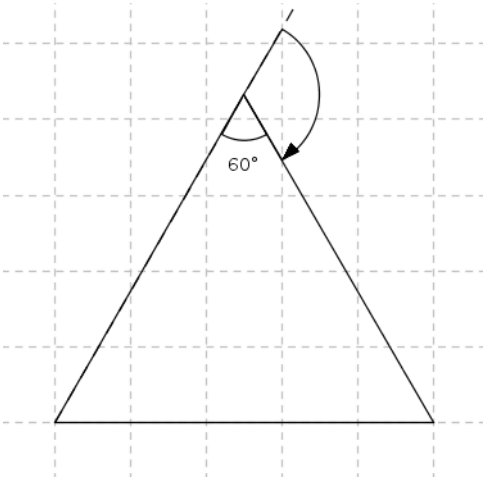
Un carreau représente 50 pas de tortue. Pour dessiner le carré ci-contre, on va donc taper :

```
av 200 td 90 av 200 td 90 av 200 td 90 av 200 td 90
```

On s'aperçoit ainsi que l'on répète 4 fois la même instruction d'où une syntaxe plus rapide :

```
repete 4[av 200 td 90]
```

5.2.2 Le triangle équilatéral



Ici, un carreau représente 30 pas de tortues. Nous allons voir ici comment tracer ce triangle équilatéral de 150 pas de tortue de côté.

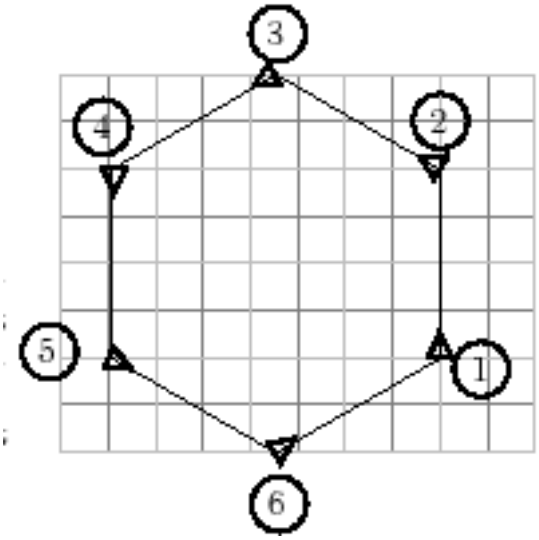
La commande ressemblera à quelque chose du style :

```
repete 3[av 150 td ....]
```

Reste à déterminer le bon angle. Dans un triangle équilatéral, les angles valent tous 60 degrés. Comme la tortue doit tourner à l'extérieur du triangle. L'angle vaudra 180-60=120 degrés. La commande est donc :

```
repete 3[av 150 td 120]
```

5.2.3 L'hexagone



Ici, un carreau représente 20 pas de tortues.

```
repete 6[av 80 td ....]
```

On s'aperçoit que lors de son déplacement, la tortue effectue en fait un tour complet sur elle même. (Elle part orientée vers le haut puis revient dans cette position). Cette rotation de 360 degrés s'effectue en 6 étapes. Par conséquent, à chaque fois, elle tourne de $\frac{360}{6} = 60^\circ$.

La commande est donc : `repete 6[av 80 td 60]`

5.2.4 Tracer un polygone régulier en général

En fait, en réitérant le petit raisonnement précédent, on s'aperçoit que pour tracer un polygone à n côtés, l'angle s'obtiendra en divisant 360 par n. Par exemple :

- Pour tracer un pentagone régulier de côté 100 :
  `repete 5[av 100 td 72]  (360:5=72)`
- Pour tracer un ennagone régulier (9 côtés) de côté 20 :
  `repete 9[av 20 td 40]  (360:9=40)`
- Pour tracer un euh... 360-gone régulier de côté 2 : (ça ressemble fortement à un cercle, ça !)
  `repete 360[av 2 td 1]`
- Pour tracer un heptagone de côté 120 :
  `repete 7[av 120 td 360/7]`

5.3 Enregistrer une procédure

Pour éviter d'avoir à retaper à chaque fois les instructions pour dessiner un carré, un triangle ... on peut définir des instructions personnelles appelées « procédures ». Une procédure commence par le mot-clé `pour` et se termine par le mot-clé `fin`. On ouvre l'éditeur, on tape par exemple

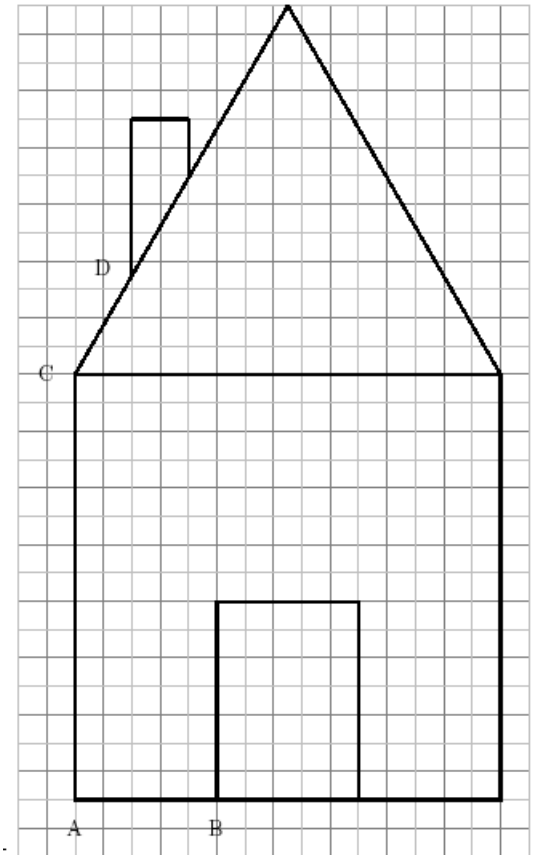
```
pour carre
repete 4[av 100 td 90]
fin
```

puis on ferme l'éditeur en enregistrant les modifications en cliquant sur le bouton tortue. Maintenant à chaque fois que l'on tape `carre`, un carré apparaît à l'écran !

5.4 Exercice ...

Un petit carreau représente 10 pas de tortue.
Essayer de réaliser le dessin ci-dessous en définissant huit procédures :

- Une procédure « `carre` » qui tracera le carre de base de la maison.
- Une procédure « `tri` » qui tracera le triangle équilatéral représentant le toit de la maison.
- Une procédure « `porte` » qui tracera le rectangle représentant la porte.
- Une procédure « `che` » qui tracera la cheminée
- Une procédure « `dep1` » qui permettra à la tortue de se déplacer de la position A à la position B.
- Une procédure « `dep2` » qui permettra à la tortue de se déplacer de la position B à la position C.
- Une procédure « `dep3` » qui permettra à la tortue de se déplacer de la position C à la position D. (Attention, il faudra peut-être lever le crayon de la tortue...)
- Une procédure « `ma` » qui permettra de tracer la maison en entier en s'aidant de toutes les autres procédures.



Chapitre 6

Se servir des coordonnées

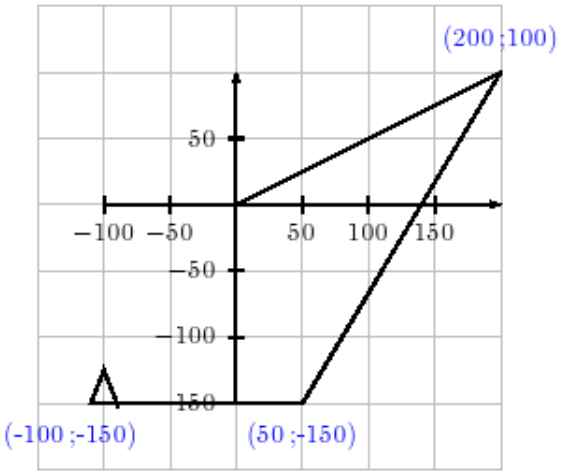
Niveau : débutant

6.1 Présentation

Dans ce chapitre, nous allons découvrir la primitive `fixeposition`. La zone de dessin est en fait muni d'un repère dont l'origine est située au centre de l'écran. On peut ainsi atteindre chacun des points de la zone de dessin à l'aide de ses coordonnées.

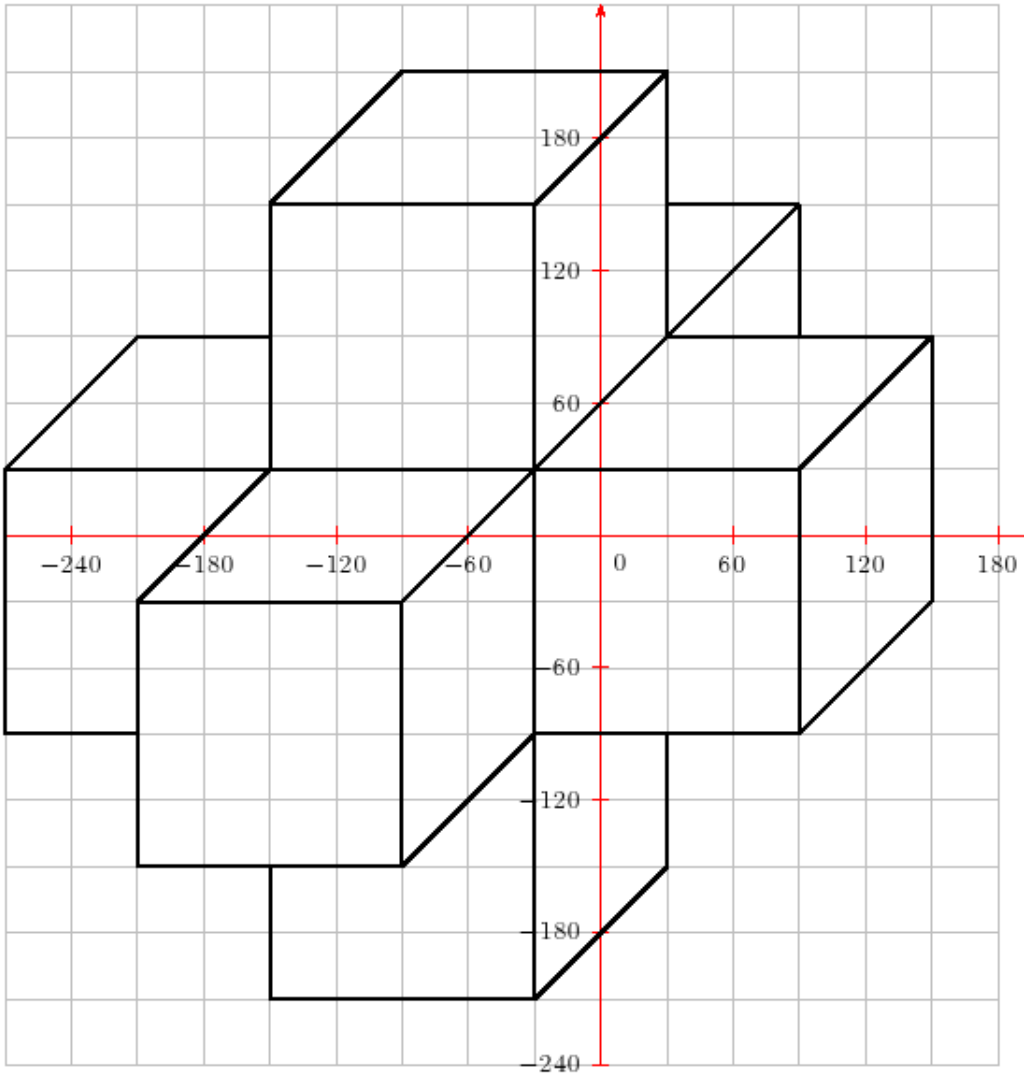
`fpos liste` `fpos [100 -250]`
Déplace la tortue au point dont les coordonnées sont définis dans la liste.

Un petit exemple d'utilisation :
`ve fpos [200 100] fpos [50 -150] fpos [-100 -150]`



6.2 Exercice :

Réaliser cette figure en n'utilisant que les primitives : `fpos`, `ve`, `lc`, `bc`.



Chapitre 7

Les variables

Niveau : débutant

Parfois, on souhaite tracer une figure à différentes échelles. Par exemple, si on souhaite dessiner un carré de côté 100, un carré de côté 200 et un carré de côté 50, actuellement on définirait trois procédures différentes correspondant à chacun de ces carrés.

```
pour carre1
repete 4 [av 100 td 90]
fin
pour carre2
repete 4 [av 200 td 90]
fin
pour carre3
repete 4 [av 50 td 90]
fin
```

On s'aperçoit immédiatement qu'il serait plus simple de définir une seule procédure à laquelle on préciserait juste la longueur du côté à tracer. Par exemple, `carre 200` tracerait le carré de côté 200, `carre 100` tracerait le carré de côté 100 etc. C'est précisément ce que vont permettre de réaliser les variables.

7.1 Exemples d'utilisation

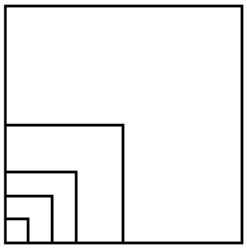
Pour tracer un carré de côté 100, on utilise :

```
pour carre
repete 4[av 100 td 90]
fin
```

Nous allons modifier cette procédure afin qu'elle reçoive un paramètre (on dit également « argument ») indiquant la longueur du côté à tracer.
Un nom de variable est toujours précédée du symbole « : ». Lorsqu'on veux indiquer que la procédure `carre` dépend de la variable `:c`, on rajoute `:c` à la fin de la ligne de défnition.
Par conséquent, ensuite, on avancera non plus de 100 pas de tortue mais de `:c` pas de tortues. La procédure devient alors :

```
pour carre :c
repete 4[av :c td 90]
fin
```

Ainsi, en tapant : `carre 100 carre 50 carre 30 carre 20 carre 10`



7.2 Tracer un rectangle de longueur et largeur déterminée

On définit ici une procedure nommée `rec` qui dépend de deux variables représentant les deux dimensions du rectangles. `rec 200 100` trace ainsi un rectangle de hauteur 200 et largeur 100.

```
pour rec :lo :la
repete 2[av :lo td 90 av :la td 90]
fin
```

Faites des essais :

```
rec 200 100 rec 100 300 rec 50 150 rec 1 20 rec 100 2
```

Bien sûr, si vous ne donnez qu'un argument à la procédure `rec`, l'interpréteur vous signalera par un message d'erreur que la procédure attend un autre argument.

7.3 Tracer une forme à des tailles diverses

Nous avons vu comment tracer un carré, un rectangle à des tailles différentes. Nous allons reprendre l'exemple de la maison p. 32 et voir comment modifier le code pour tracer la maison à l'échelle souhaitée.

L'objectif est de passer un argument à la procédure `ma` pour que selon le paramètre, la maison soit plus ou moins grande. Nous souhaitons que `ma 1` trace la maison en taille réelle.
`ma 0,5` tracera une maison à l'échelle 0,5.
`ma 2` tracera une maison aux dimensions deux fois plus grandes etc

La notion de proportionnalité est bien sûr sous-jacente. En vraie grandeur, la procédure `carre` était la suivante :

```
pour carre
repete 4[av 150 td 90]
fin
```

Toutes les dimensions originales de la maison sont multipliées par l'échelle. La procédure `carre` devient :

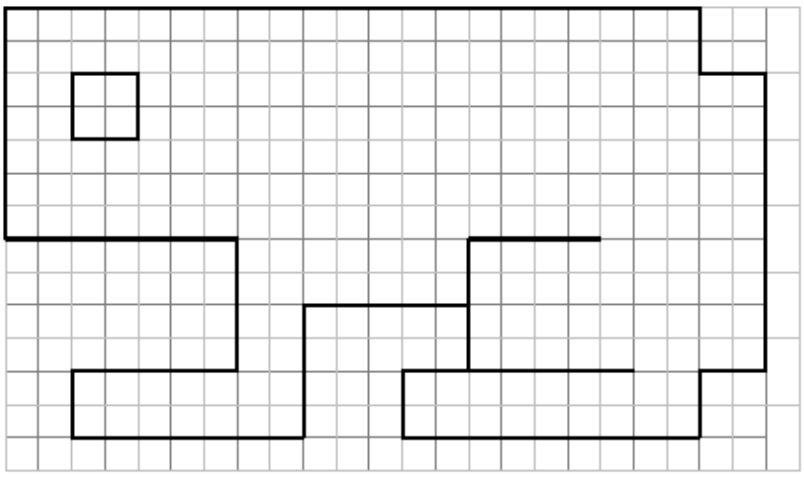
```
pour carre :c
repete 4[av 150*:c td 90]
fin
```

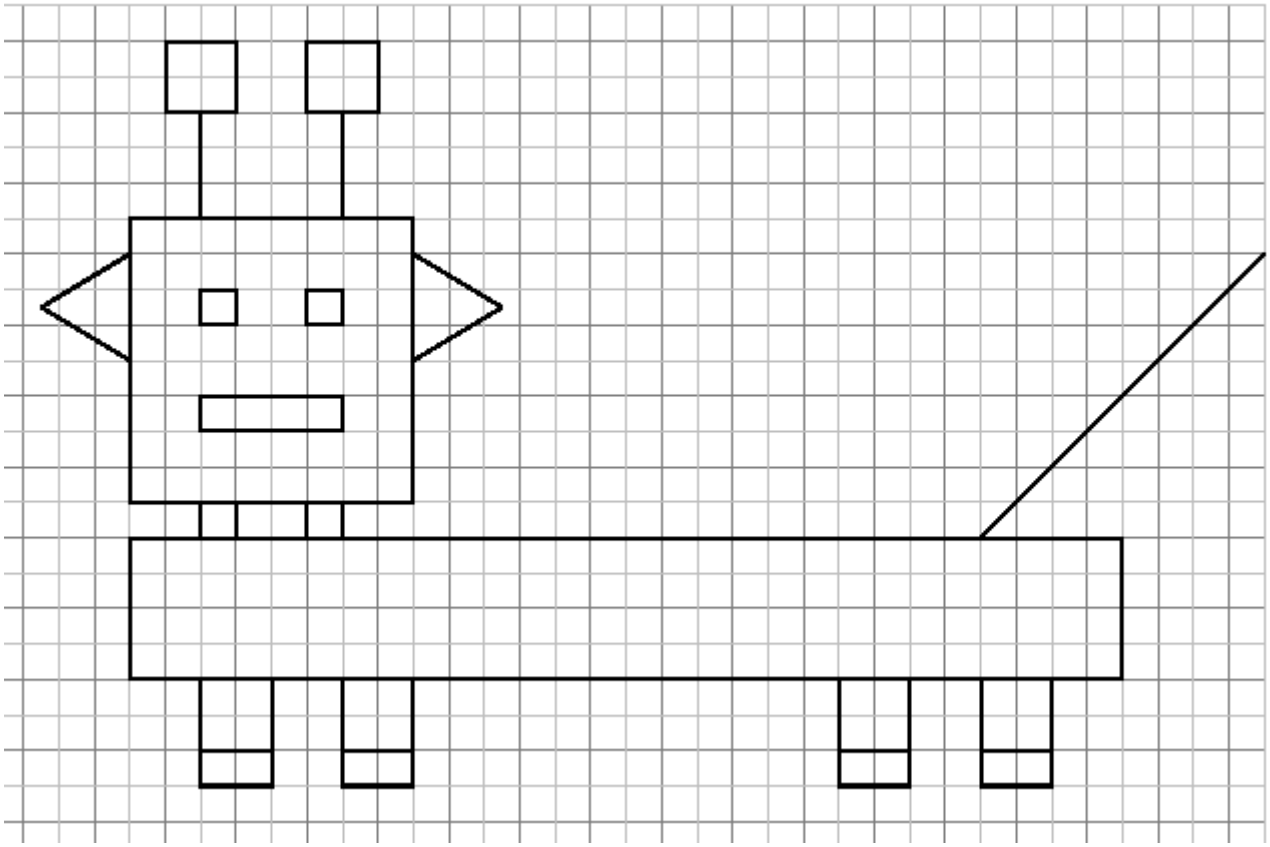
Ainsi quand on tapera `carre 2`, le carré aura pour côté $150 \times 2 = 300$. les proportions sont bien respectées ! En fait, on s'aperçoit qu'il va juste falloir reprendre toutes les procédures et changer les longueurs de déplacement de la manière suivante :
`av 70` devient `av 70* :c`
`av 45` devient `av 45* :c`
etc

```
pour carre :c
repete 4[av 150*:c  td 90]
fin

pour tri :c
repete 3[av 150*:c td 120]
fin

pour porte :c
repete 2[av 70*:c td 90 av 50*:c td 90]
fin

pour che :c
av 55*:c td 90 av 20*:c td 90 av 20*:c
fin

pour dep1 :c
td 90 av 50*:c tg 90
fin

pour dep2 :c
tg 90 av 50*:c td 90 av 150*:c td 30
fin

pour dep3 :c
lc td 60 av 20*:c tg 90 av 35*:c bc
fin

pour ma :c
carre :c dep1 :c porte :c dep2 :c tri :c dep3 :c che :c
fin
```

7.4 Exercice :

Réaliser les dessins suivants avec des variables de telle sorte que l'on puisse les obtenir à des tailles diverses.





Chapitre 8

La récursivité

Niveau : moyen

Le langage Logo utilise très souvent une technique de programmation appelée la récursivité. Dans ce chapitre, nous découvrirons tout d'abord cette notion sur des exemples simples pour ensuite approfondir avec notamment le tracé d'une fractale appelée le flocon de Van Koch. Pour commencer, petite explication :

Une procédure est récursive si elle s'appelle elle-même.

8.1 Avec la zone de dessin.

8.1.1 Premier exemple :

```
pour ex1
td 1
ex1
fin
```

Cette procédure est récursive puisque la procédure `ex1` est appelée à la dernière ligne. A l'exécution, on constate que la tortue ne cesse de tourner sur elle-même. Pour interrompre le programme, on est obligé de se servir du bouton STOP.

8.1.2 Deuxième exemple :

Tout d'abord, voici trois nouvelles primitives :

- `attends nombre` `attends 60`
  Bloque le programme pendant le nombre de $60^{\text{ième}}$ de secondes indiqué.
  Par exemple, `attends 120` bloquera le programme pendant deux secondes.
- `gomme` gomme
  Lorsque la tortue se déplace, elle efface tout au lieu de laisser un trait derrière elle.
- `dessine,de` dessine
  Repasse en mode dessin classique : la tortue laisse un trait derrière elle en se déplaçant.

```
pour ex2
av 200 gomme attends 60
re 200 dessine td 6
ex2
fin
```

Il ne reste plus qu'à lancer ce programme. A chaque seconde le même motif recommence et le programme simule ainsi une trotteuse !

8.2 Avec la zone de texte

8.2.1 Un premier exemple :

La primitive `ecris, ec` permet d'afficher un texte dans la zone de texte. elle attend pour argument soit une liste soit un mot. Ex : `ec "bonjour ec [J'écris ce que je veux]` (Ne pas oublier la quote " lorsqu'on veut juste écrire un mot.)

```
pour ex3 :n
ecris :n
ex3 :n+1
fin
```

Lancer la commande `ex3 0` puis interrompre avec le bouton STOP
Faites les changements nécessaires dans ce programme pour que les chiffres apparaissent de deux en deux.

Je veux à présent afficher tous les chiffres supérieur à 100 qui sont dans la table de cinq. Il suffit alors de modifier le programme ainsi :

```
pour ex3 :n
ecris :n
ex3 :n+5
fin
```

et de lancer : `ex3 100`

8.2.2 Réaliser un test de sortie

Taper les commandes suivantes :

```
si 2+1=3 [ecris [ceci est vrai]]
si 2+1=4 [ecris [ceci est vrai]][ecris [le calcul est faux]]
si 2+5=7 [ec "vrai][ec "faux]
```

Si vous n'avez toujours pas compris la syntaxe de la primtive `si`, reportez-vous au manuel de référence de XLogo.

```
pour ex3 :n
si :n=100 [stop]
ecris :n
ex3 :n+1
fin
```

Lancer la commande `ex3 0`
Faites les changements nécessaires dans ce programme pour faire apparaître les chiffres compris entre et 55 et 350 qui sont dans la table de 11.

8.3 Un exemple de fractale : le flocon de Van Koch

Grâce à la récursivité, il est très facile de générer en Logo des objets que l'on appelle en mathématiques des fractales.

Voici les premières étapes permettant de créer la ligne brisée de Van Koch.

Entre chaque étape :

1. chacun des segments est partagé en trois parties égales.
2. on trace un triangle équilatéral sur le deuxième segment.
3. on efface ce deuxième segment.

Ce qu'il faut remarquer : Prenons le cas de la deuxième étape, on constate que cette ligne est formée de quatre motifs correspondant à l'étape précédente et dont la taille est divisée par 3. On vient de mettre en évidence la nature récursive de la fractale.

Appelons $L_{n,\ell}$ le motif de longueur ℓ, tracé à l'étape n.
Pour tracer ce motif voici le procédé :

1. On trace $L_{n-1,\ell/3}$
2. On tourne à gauche de 60 degrés
3. On trace $L_{n-1,\ell/3}$
4. On tourne à droite de 120 degrés
5. On trace $L_{n-1,\ell/3}$
6. On tourne à gauche de 60 degrés
7. On trace $L_{n-1,\ell/3}$

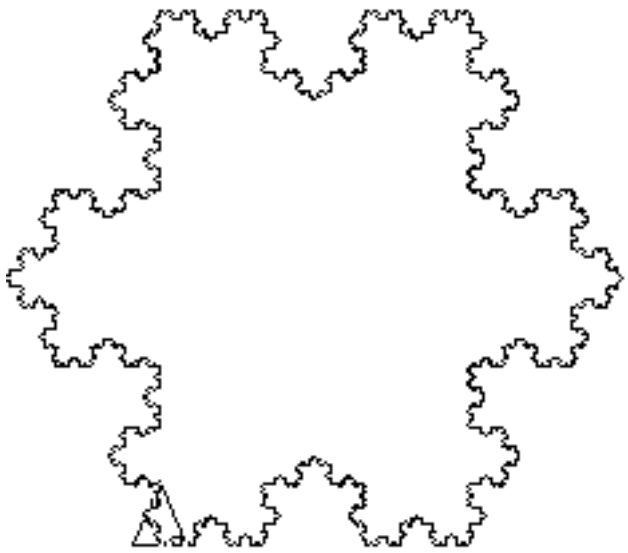
En Logo, cela donne tout simplement :

```
# :l longueur du motif
# :p étape
pour ligne :l :p
si :p=0 [av :l] [
  ligne :l/3 :p-1 tg 60 ligne :l/3 :p-1 td 120 ligne :l/3 :p-1 tg 60 ligne :l/3 :p-1
]
fin
```

Si l'on trace un triangle équilatéral composé de trois de ces lignes, on obtient un magnifique flocon de Van Koch

```
# :l longueur du côté
pour flocon :l :p
repete 3[ligne :l :p td 120]
fin
```

Puis en lançant : `flocon 200 6`



8.4 Recursivite sur les mots

Consulter la liste des primitives p.87 afin de comprendre le rôle des primitives `mot`, `dernier`, et `saufdernier`.

Voici une procédure récursive qui permet d'inverser l'ordre des lettres d'un mot

```
pour inversem :m
si vide? :m [retourne "]
retourne mot dernier :m inversem saufdernier :m
fin

ecris inversem "abcde
edcba
```

On dit qu'un mot est un palindrome si on peut le lire dans les deux sens (exemples : radar, laval ...).

```
# teste si le mot :m est un palindrome
pour palindrome :m
si  :m=inversem :m [retourne vrai] [retourne faux]
fin
```

Et enfin ce joli petit programme (Merci Olivier SC) :

```
pour palin :n
si palindrome :n [ecris :n stop]
ecris (liste :n "PLUS inversem :n "EGAL somme :n inversem :n)
palin :n + inversem :n
fin

palin 78
78 PLUS 87 EGAL 165
165 PLUS 561 EGAL 726
726 PLUS 627 EGAL 1353
1353 PLUS 3531 EGAL 4884
4884
```

8.5 Calculer un factorielle

On définit factorielle de 5, noté 5! par :

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

De manière générale pour n strictement positif, on remarque que : $n! = n \times (n-1)!$.
Cette relation explique la nature récursive de ce programme :

```
pour fac :n
si :n=0[ret 1][ret :n*fac :n-1]
fin

ec fac 5
120
ec fac 6
720
```

8.6 Une approximation de π

On peut obtenir une approximation du nombre π avec la formule :

$$\pi \approx 2^k \sqrt{2 - \sqrt{2 + \sqrt{2 + \ldots \sqrt{2 + \sqrt{2}}}}}$$

où k est le nombre de racines carrées. Plus k devient grand et plus cette expression se rapproche du nombre π.

La formule est constituée de l'expression $2 + \sqrt{2 + \ldots \sqrt{2 + \sqrt{2}}}$ qui est clairement récursive d'où le programme :

```
# k désigne le nombre de racines
pour approxpi :k
tape "Approximation:\  ec (puissance 2 :k) * racine (2- racine (calc :k-2))
ec "-------------------------
tape "Pi:\  ec pi
fin

pour calc :p
si :p=0 [ret 2][ret 2+racine calc :p-1]
fin

approxpi 10
Approximation: 3.141591421568446
-------------------------
Pi: 3.141592653589793
```

On a obtenu les 5 premières décimales ! Si l'on souhaite davantage, il faudra éliminer certaines erreurs de calculs dues aux racines carrées imbriquées. Pour cela nous allons augmenter le nombre de décimales avec la primitive `fixedecimales`.

```
fixedecimales 100
approxpi 100
Approximation: 3.14159265358979323846264338327950288419733930696701609758076843138804 68...
-------------------------
Pi: 3.14159265358979323846264338327950288419716939937510582097494459230781640 6....
```

Et on obtient à présent 39 décimales...

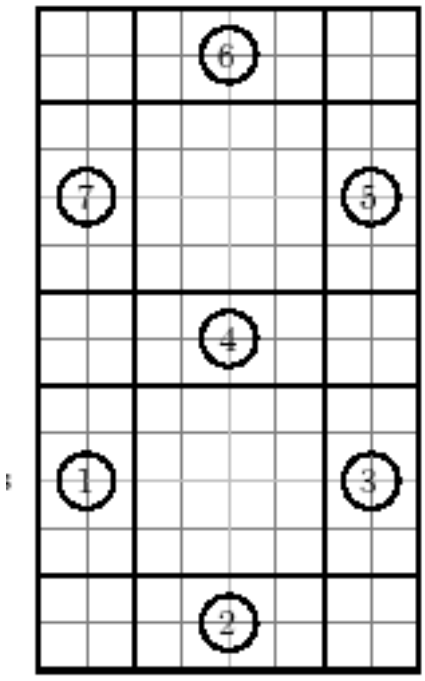
Chapitre 9

Créer une animation

Niveau : moyen

Ce chapitre propose deux thèmes assez différents dont l'objectif est de créer une animation à l'aide de XLogo.

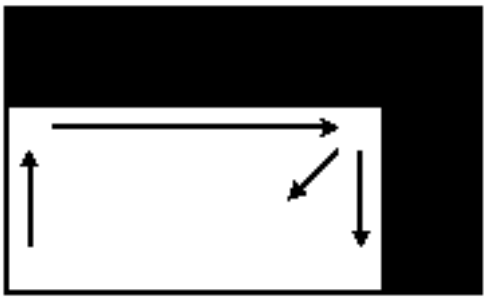
9.1 Les chiffres de calculatrice



Cette activité est basé sur le fait que tous les nombres de calculatrice peuvent être obtenus à l'aide du patron ci-contre :

- Par exemple, pour dessiner un « 4 », on allumera les rectangles 3,4,5,7.
- Pour dessiner un « 8 », on allumera les rectangles 1,2,3,4,5,6,7.
- Pour dessiner un « 3 », on allumera les rectangles 2,3,4,5,6.

9.1.1 Remplir un rectangle



Si l'on souhaite par exemple tracer un rectangle rempli de 100 sur 200, une première idée peut être de dessiner le rectangle de 100 sur 200 puis de tracer un rectangle de 99 sur 199 puis un rectangle de 98 sur 198 ... jusqu'à ce que le rectangle soit entièrement rempli.
Commençons par définir un rectangle de longueur et largeur dépendant de deux variables.

```
pour rec :lo :la
repete 2[av :lo td 90 av :la td 90]
fin
```

Pour remplir notre grand rectangle, on va donc exécuter :
`rec 100 200 rec 99 199 rec 98 198 ..... rec 1 101`

Définissons alors une procédure rectangle dédié à tracer ce rectangle rempli.

```
pour rectangle :lo :la
rec :lo :la
rectangle :lo-1 :la-1
fin
```

On teste `rectangle 100 200` et on s'aperçoit qu'il y a un problème : la procédure ne s'arrête pas lorsque le rectangle est rempli, elle continue de tracer des rectangles ! On va donc ajouter un test permettant de détecter si la longueur ou la largeur est égale à 0. A ce moment, on demande au programme de s'interrompre avec la commande `stop`.

```
pour rectangle :lo :la
si ou :lo=0 :la=0 [stop]
rec :lo :la
rectangle :lo-1 :la-1
fin
```

Note : à la place d'utiliser la primitive `ou`, on peut utiliser le symbole « | » : on obtiendrait :

```
si :lo=0 | :la=0 [stop]
```

9.1.2 Le programme

Nous avons besoin du rectangle rempli précédent :

```
pour rec :lo :la
si :lo=0 |:la=0[stop]
repete 2[av :lo td 90 av :la td 90]
rec :lo-1 :la-1
fin
```

Nous supposons ici que la tortue part du coin inférieur gauche. Nous allons définir une procédure appelée `chiffre` admettant 7 argument `:a`, `:b`, `:c`, `:d`, `:e`, `:f`, `:g`. Quand `:a` vaut 1, on dessine le rectangle 1. Si `:a` vaut 0, on ne le dessine pas. Voilà le principe.

On obtient la procédure suivante :

```
pour chiffre :a :b :c :d :e :f :g
# On dessine le rectangle 1
si :a=1 [rec 160 40]
# On dessine le rectangle 2
si :b=1 [rec 40 160]
lc td 90 av 120 tg 90 bc
# On dessine le rectangle 3
si :c=1 [rec 160 40]
lc av 120 bc
# On dessine le rectangle 5
si :e=1 [rec 160 40]
# On dessine le rectangle 4
tg 90 lc re 40 bc
si :d=1 [rec 160 40]
# On dessine le rectangle 6
td 90 lc av 120 tg 90 bc
si :f=1 [rec 160 40]
# On dessine le rectangle 7
lc av 120 tg 90 re 40 bc
si :g=1 [rec 160 40]
fin
```

9.1.3 Création d'une petite animation

Nous allons ici simuler un compte à rebours en faisant apparaitre succesivement les chiffres de 9 à 0 par ordre décroissant.

```
pour rebours
ve ct chiffre 0 1 1 1 1 1 1 attends 60
ve ct chiffre 1 1 1 1 1 1 1 attends 60
ve ct chiffre 0 0 1 0 1 1 0 attends 60
ve ct chiffre 1 1 1 1 0 1 1 attends 60
ve ct chiffre 0 1 1 1 0 1 1 attends 60
ve ct chiffre 0 0 1 1 1 0 1 attends 60
ve ct chiffre 0 1 1 1 1 1 0 attends 60
ve ct chiffre 1 1 0 1 1 1 0 attends 60
ve ct chiffre 0 0 1 0 1 0 0 attends 60
ve ct chiffre 1 1 1 0 1 1 1 attends 60
fin
```

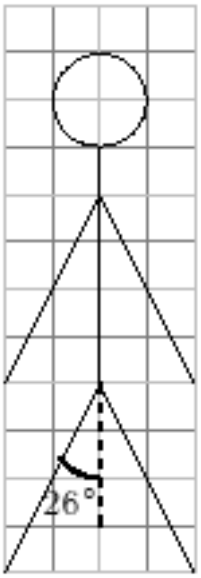
Petit problème : il y a un effet de clignotement désagréable pendant la création de chaque chiffre. Pour fluidifier cela on va utiliser les primitives `animation`, `stopanimation` et `rafraichis`.

- `animation` permet de basculer en mode « animation ». La tortue ne dessine plus à l'écran mais dans le cache, c'est à dire qu'elle effectue les changements en mémoire. Elle n'affichera l'image que lorsqu'on lui le demande à l'aide la primitive `rafraichis`.
- `stopanimation` permet de revenir à l'affichage classique.

On obtient ainsi le programme modifié :

```
pour rebours
# On passe en mode animation
animation
ve ct chiffre 0 1 1 1 1 1 1 rafraichis attends 60
ve ct chiffre 1 1 1 1 1 1 1 rafraichis attends 60
ve ct chiffre 0 0 1 0 1 1 0 rafraichis attends 60
ve ct chiffre 1 1 1 1 0 1 1 rafraichis attends 60
ve ct chiffre 0 1 1 1 0 1 1 rafraichis attends 60
ve ct chiffre 0 0 1 1 1 0 1 rafraichis attends 60
ve ct chiffre 0 1 1 1 1 1 0 rafraichis attends 60
ve ct chiffre 1 1 0 1 1 1 0 rafraichis attends 60
ve ct chiffre 0 0 1 0 1 0 0 rafraichis attends 60
ve ct chiffre 1 1 1 0 1 1 1 rafraichis attends 60
# On rebascule en mode dessin classique
stopanimation
fin
```

9.2 Une animation : le bonhomme qui grandit



Tout d'abord, nous allons définir une procédure `bon` qui trace le bonhomme ci-contre à la taille de notre choix.

```
pour bon :c
tg 154 av 44*:c re 44*:c
tg 52 av 44*:c re 44*:c
tg 154 av 40*:c
tg 154 av 44*:c re :c*44
tg 52 av 44*:c re :c*44
tg 154 av 10*:c
tg 90 repete 180[av :c/2 td 2] td 90
fin
```

Nous allons à présent créer une animation donnant l'illusion que le bonhomme grandit petit à petit. Pour cela, nous allons tracer `bon 0.1` puis `bon 0.2` `bon 0.3` ... jusqu'à `bon 5`. Entre chaque tracé, on effacera l'écran. On obtient les deux procédures suivantes :

```
pour bon :c
tg 154 av 44*:c re 44*:c
tg 52 av 44*:c re 44*:c
tg 154 av 40*:c
tg 154 av 44*:c re :c*44
tg 52 av 44*:c re :c*44
tg 154 av 10*:c
tg 90 repete 180[av :c/2 td 2] td 90
si :c=5[stop]
ve ct bon :c+0.1
fin

pour demarrer
ve ct
bon 0
fin
```

Enfin, pour fluidifier le tout, on va se servir du mode animation et de la primitive `rafraichis`.

```
pour bon :c
tg 154 av 44*:c re 44*:c
tg 52 av 44*:c re 44*:c
tg 154 av 40*:c
tg 154 av 44*:c re :c*44
tg 52 av 44*:c re :c*44
tg 154 av 10*:c
tg 90 repete 180[av :c/2 td 2] td 90
rafraichis
si :c=5[stop]
ve ct bon :c+0.1
fin

pour demarrer
ct animation
bon 0
stopanimation
fin
```

Remarque : Ici, la procédure `bon` est récursive, on aurait plus simplement utiliser la primitive `repetepour` afin de faire varier `:c` de 0,1 à 5. Voici le programme obtenu alors :

```
pour bon :c
ve ct tg 154 av 44*:c re 44*:c
tg 52 av 44*:c re 44*:c
tg 154 av 40*:c
tg 154 av 44*:c re :c*44
tg 52 av 44*:c re :c*44
tg 154 av 10*:c
tg 90 repete 180[av :c/2 td 2] td 90
rafraichis
fin

pour demarrer
ct animation
```

```
repetepour [c 0 5 0.1][bon :c]
stopanimation
fin
```

Chapitre 10

Programme interactif

Niveau : débutant

10.1 Communiquer avec l'utilisateur

Nous allons réaliser un petit programme qui demande à l'utilisateur son nom, son prénom et son age. A la fin du questionnaire, le programme répond par un récapitulatif su style :

```
Ton nom est:........
Ton prénom est: .......
Ton age est: .......
Tu es mineur ou majeur
```

Pour cela, nous allons utiliser les primitives suivantes :

- `lis` : `lis [Quel est ton age ? ] "a`
  Affiche une boîte de dialogue ayant pour titre le texte contenu dans la liste (ici, « Quel est ton age ? »). La réponse donnée par l'utilisateur est mémorisée sous forme d'un mot ou d'une liste (si l'utilisateur tape plusieurs mots) dans la variable `:a`.

- `donne` : `donne "a 30`

  Donne la valeur 30 à la variable `:a`

- `phrase, ph` : `phrase [30 k] "a`

  Rajoute une valeur dans une liste. Si cette valeur est une liste, assemble les deux listes.

  ```
  phrase [30 k] "a ---> [30 k a]
  phrase [1 2 3] 4 ---> [1 2 3 4]
  phrase [1 2 3] [4 5 6] ---> [1 2 3 4 5 6]
  ```

Nous obtenons le code suivant :

```
pour question
lis [Quel est ton age?] "age
lis [Quel est ton nom?] "nom
lis [Quel est ton prénom?] "prenom
ecris phrase [Ton nom est: ] :nom
ecris phrase [Ton prénom est: ] :prenom
ecris phrase [Ton age est: ] :age
si ou :age>18 :age=18 [ecris [Tu es majeur]] [ecris [Tu es mineur]]
fin
```

10.2 Programmer un petit jeu.

L'objectif de ce paragraphe est de créer le jeu suivant :

Le programme choisit un nombre au hasard entre 0 et 32 et le mémorise. Une boîte de dialogue s'ouvre et demande à l'utilisateur de rentrer un nombre. Si le nombre proposé est égal au nombre mémorisé, il affiche « gagné »dans la zone de texte. Dans le cas contraire, le programme indique si le nombre mémorisé est plus petit ou plus grand que le nombre proposé par l'utilisateur puis rouvre la boîte de dialogue. Le programme se termine quand l'utilisateur a trouvé le nombre mémorisé.

Vous aurez besoin d'utiliser la primitive suivante :
`hasard` : `hasard 8`
`hasard 20` rend donc un nombre choisi au hasard entre 0 et 19.
Rend un nombre au hasard compris entre 0 et 8 strictement.

Voici quelques règles à respecter pour réaliser ce petit jeu :

- Le nombre mémorisé par l'ordinateur sera mémorisé dans une variable nommée `nombre`.
- La boîte de dialogue aura pour titre : « Propose un nombre : ».
- Le nombre proposé par l'utilisateur sera enregistré dans une variable nommée `essai`.
- La procédure qui permet de lancer le jeu s'appellera `jeu`.

Quelques améliorations possibles :

- Afficher le nombre de coups.
- Le nombre recherché devra être compris entre 0 et 2000.
- Vérifier si ce que rentre l'utilisateur est réellement un nombre. Pour cela, utiliser la primitive `nombre ?`.
  Exemples : `nombre ? 8` est vrai.
  `nombre ? [5 6 7]` est faux. ([5 6 7] est une liste et non pas un nombre)
  `nombre ? "abcde` est faux. ("abcde est un mot et non pas un nombre)

Chapitre 11

Thème : Somme de deux dés

Niveau : Moyen

Lorsqu'on lance deux dés et qu'on fait le total des points de chacun des dés, on obtient un entier compris entre 2 et 12. Ici, nous allons voir dans cette activité la répartition des différents tirages et la représenter sous forme d'un petit graphique.

11.1 Simuler le lancer d'un dé.

Pour simuler le lancer d'un dé, nous allons utiliser la primitive `hasard`. Voici comment procéder.

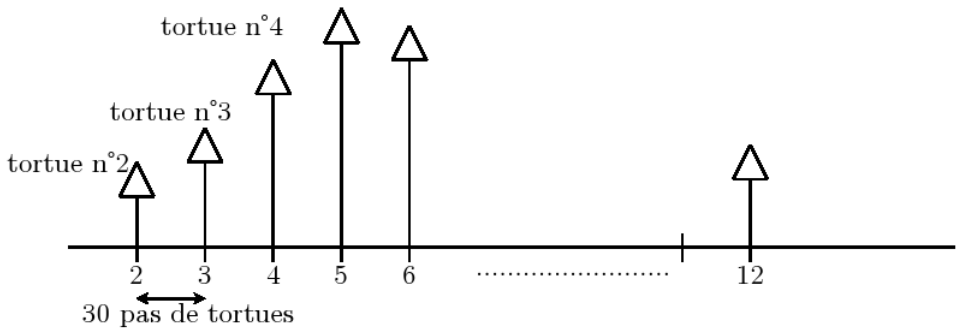
`hasard 6` $\longrightarrow$ renvoie un entier pris au hasard parmi 0, 1, 2, 3, 4, 5.
Par conséquent, `(hasard 6)+1` renvoie un entier pris au hasard parmi 1, 2, 3, 4, 5, 6. Noter bien les parenthèses, sinon l'interpréteur Logo comprendrait `hasard 7`. Pour éviter les parenthèses, on peut taper également `1+hasard 6`.

On définit ainsi la procédure `lancer` qui simule le lancer d'un dé.

```
pour lancer
  retourne 1+hasard 6
fin
```

11.2 Le programme

Nous allons utiliser le mode multi-tortues. Pour disposer ainsi de plusieurs tortues sur l'écran, on utilise la primitive `fixetortue` suivi du numéro de la tortue que l'on veut sélectionner.
Un bon schéma valant mieux que mille explications....



Sur le principe, chaque tortue numérotée de 2 à 12 avancera d'un pas de tortue, lorsque le tirage de la somme des deux dés sera identique à son numéro. Par exemple, si les dés ont pour somme 8, la tortue numéro 8

avancera d'un pas. Toutes les tortues sont espacées de 30 pas de tortues horizontalement.

On placera les tortues à l'aide des coordonnées.

- La tortue n°2 sera placée en $(-150; 0)$
- La tortue n°3 sera placée en $(-120; 0)$
- La tortue n°4 sera placée en $(-90; 0)$
- La tortue n°5 sera placée en $(-60; 0)$

$\vdots$

```
 fixetortue 2 fpos [-150 0]
 fixetortue 3 fpos [-120 0]
 fixetortue 4 fpos [-90 0]
 fixetortue 5 fpos [-60 0]
 fixetortue 6 fpos [-30 0]
.....
```

Plutôt que de taper 11 fois quasiment la même ligne de commande, on peut automatiser cela en utilisant la primitive `repetepour`. A l'aide de cette primitive, on peut affecter à une variable une succession de valeurs prises dans un intervalle à espaces réguliers. Ici, on veut que la variable `:i` prenne successivement les valeurs 2, 3, 4, ... , 12. On tapera :
`repetepour [i 2 12] [ liste des instructions à exécuter ]`

Pour placer les tortues, on crée donc la procédure `initialise`

```
pour initialise
  ve ct
  repetepour [i 2 12] [
      # On place la tortue
      fixetortue :i fpos liste -150+(:i-2)*30 0
      # On écrit le numéro de la tortue juste en dessous
      lc re 15 etiquette :i av 15 bc
  ]
fin
```

Bien comprendre la formule `-150+( :i-2)*30`. On part de -150, puis à chaque nouvelle tortue on rajoute 30. (Tester avec les différentes valeurs de `:i` si vous n'êtes pas convaincu)

Au final on obtient le programme suivant :

```
pour lancer
 retourne 1+hasard 6
fin

pour initialise
  ve ct
  repetepour [i 2 12] [
      # On place la tortue
      fixetortue :i fpos liste -150+(:i-2)*30 0
      # On écrit le numéro de la tortue juste en dessous
      lc re 15 etiquette :i av 15 bc
  ]
fin

pour demarrer
initialise
```

```
# On effectue 1000 tentatives
repete 1000 [
  donne "somme lancer+lancer
  fixetortue :somme av 1
]
# On affiche les fréquences de tirage
repetepour [i 2 12] [
  fixetortue :i
  # L'ordonnée de la tortue représente le nombre de tirages
  soit "effectif dernier pos
  lc av 10 tg 90 av 10 td 90 bc etiquette :effectif/1000*100
]
fin
```

Voici une généralisation de ce programme. Ici, on demandera à l'utilisateur le nombre de dés souhaités ainsi que le nombre de lancers à effectuer.

```
pour lancer
soit "somme 0
repete :des [
   soit "somme :somme+1 +hasard 6
 ]
retourne :somme
fin

pour initialise
  ve ct fixemaxtortues :max+1
  repetepour ph liste "i :min :max [
      # On place la tortue
      fixetortue :i fpos liste (:min-:max)/2*30+(:i-:min)*30 0
      # On écrit le numéro de la tortue juste en dessous
      lc re 15 etiquette :i av 15 bc
  ]
fin

pour demarrer
lis [Nombre de dés:] "des
si non nombre? :des [ec [Le nombre rentré n'est pas valide!] stop]
donne "min :des
donne "max 6*:des
lis [Nombre de lancers à effectuer] "tirages
si non nombre? :tirages [ec [Le nombre rentré n'est pas valide!] stop]
initialise
# On effectue 1000 tentatives
repete :tirages [
  fixetortue lancer av 1
]
# On affiche les fréquences de tirage
repetepour ph liste "i :min :max [
  fixetortue :i
  # L'ordonnée de la tortue représente le nombre de tirages
  soit "effectif dernier pos
  # On arrondit à 0,1
  lc av 10 tg 90 av 10 td 90 bc etiquette (arrondi :effectif/:tirages*1000)/10
]
fin
```

Chapitre 12

Thème : Approximation probabilistique de π

Niveau : Avancé

Avertissement : Quelques notions de mathématiques sont nécessaires pour bien appréhender ce chapitre.

12.1 Notion de pgcd (plus grand commun diviseur)

Etant donné deux nombres entiers, leur pgcd désigne leur plus grand commun diviseur.

- Par exemple, 42 et 28 ont pour pgcd 14 (c'est le plus grand nombre possible qui divise à la fois 28 et 42)
- 25 et 55 ont pour pgcd 5.
- 42 et 23 ont pour pgcd 1.

Lorsque deux nombres ont pour pgcd 1, on dit qu'ils sont premiers entre eux. Ainsi sur l'exemple précédent, 42 et 23 sont premiers entre eux. Cela signifie qu'ils n'ont aucun diviseur commun hormis 1 (bien sûr, il divise tout entier !).

12.2 Algorithme d'Euclide

Pour déterminer le pgcd de deux nombres, on peut utiliser une méthode appelée algorithme d'Euclide : (Ici, on ne démontrera pas la validité de cet algorithme)

Voici le principe : Étant donnés deux entiers positifs a et b, on commence par tester si b est nul. Si oui, alors le PGCD est égal à a. Sinon, on calcule r, le reste de la division de a par b. On remplace a par b, puis b par r, et on recommence le procédé.
Calculons par exemple, le pgcd de 2160 et 888 par cet algorithme avec les étapes suivantes :

a	b	r
2160	888	384
888	384	120
384	120	24
120	24	0
24	0	

Le pgcd de 2160 et 888 est donc 24. Il n'y pas de plus grand entier qui divise ces deux nombres. (En fait $2160 = 24 \times 90$ et $888 = 24 \times 37$)
Le pgcd est en fait le dernier reste non nul.

12.3 Calculer un pgcd en Logo

Un petit algorithme récursif permet de calculer le pgcd de deux nombres `:a` et `:b`

```
pour pgcd :a :b
si (reste :a :b)=0 [retourne :b][retourne pgcd :b reste :a :b]
fin

ecris pgcd 2160 888
24
```

Remarque : On est obligé de mettre des parenthèses sur `reste :a :b`, sinon l'interpréteur va chercher à évaluer `:b = 0`. Pour éviter ce problème de parenthésage, écrire : `si 0=reste :a :b`

12.4 Calculer une approximation de π

En fait, un résultat connu de théorie des nombres établit que la probabilité que deux entiers pris au hasard soient premiers entre eux est de $\frac{6}{\pi^2} \approx 0,6079$. Pour essayer de retrouver ce résultat, voilà ce que l'on va faire :

- Prendre deux nombres au hasard entre 0 et 1 000 000.
- Calculer leur pgcd
- Si leur pgcd vaut 1. Rajouter 1 à une variable compteur.
- Répéter l'expérience 1000 fois
- La fréquence des couples de nombres premiers entre eux s'obtiendra en divisant la variable compteur par 1000 (le nombre d'essais).

```
pour test
# On initialise la variable compteur à 0
donne "compteur 0
repete 1000 [
  si (pgcd hasard 1000000 hasard 1000000)=1 [donne "compteur :compteur+1]
]
ecris [frequence:]
ecris :compteur/1000
fin
```

Note : De même que précédemment, On est obligé de mettre des parenthèses sur `pgcd hasard 1000000 hasard 1000000`, sinon l'interpréteur va chercher à évaluer 1 000 000 = 1. Pour éviter ce problème de parenthésage, écrire : `si 1=pgcd hasard 1000000 hasard 1000000`

On lance le programme `test`.

```
test
0.609
test
0.626
test
0.597
```

On obtient des valeurs proches de la valeur théorique de 0,6097. Ce qui est remarquable est que cette fréquence est une valeur approchée de $\frac{6}{\pi^2}$.

Si je note f la fréquence trouvée, on a donc : $f \approx \frac{6}{\pi^2}$

Donc $\pi^2 \approx \frac{6}{f}$ et donc $\pi \approx \sqrt{\frac{6}{f}}$.

Je m'empresse de rajouter cette approximation dans mon programme, je transforme la fin de la procédure `test` :

```
pour test
# On initialise la variable compteur à 0
donne "compteur 0
repete 1000 [
  si 1=pgcd hasard 1000000 hasard 1000000 [donne "compteur :compteur+1]
]
# On calcule la frequence
donne "f :compteur/1000
# On affiche la valeur approchée de pi
ecris phrase [approximation de pi:] racine (6/:f)
fin
test
approximation de pi: 3.164916190172819
test
approximation de pi: 3.1675613357997525
test
approximation de pi: 3.1008683647302115
```

Bon, je modifie mon programme de tel sorte que quand je le lance, je précise le nombre d'essais souhaités. J'ai dans l'idée d'essayer avec 10000 essais, voilà ce que j'obtiens sur mes trois premières tentatives :

```
pour test :essais
# On initialise la variable compteur à 0
donne "compteur 0
repete :essais [
  si 1=pgcd hasard 1000000 hasard 1000000 [donne "compteur :compteur+1]
]
# On calcule la frequence
donne "f :compteur/:essais
# On affiche la valeur approchée de pi
ecris phrase [approximation de pi:] racine (6/:f)
fin

test 10000
approximation de pi: 3.1300987144363774
test 10000
approximation de pi: 3.1517891481565017
test 10000
approximation de pi: 3.1416626832299914
```

Pas mal, non ?

12.5 Compliquons encore un peu : π qui génère π.....

Qu'est-ce qu'un entier aléatoire ? Est-ce qu'un entier pris au hasard antre 1 et 1 000 000 est réellement représentatif d'un entier aléatoire quelconque ? On s'aperçoit très vite que notre modélisation ne fait qu'approcher le modèle idéal. Bien, c'est justement sur la façon de générer le nombre aléatoire que nous allons effectuer quelques changements... Nous n'allons plus utiliser la primitive `hasard` mais utiliser la séquence des décimales de π. Je m'explique : les décimales de π ont toujours intrigué les mathématiciens par leur manque d'irrégularité, les chiffres de 0 à 9 semblent apparaître en quantité à peu près égales et de manière aléatoire. On ne peut prédire la prochaine décimales à l'aide des précédentes. Nous allons voir ci-après comment générer un nombre alatoire à l'aide des décimales de π. Tout d'abord, il va vous falloir récupérer les premières décimales de pi (par exemple un million).

- Il existe des petits programmes qui font cela très bien. Je conseille PiFast pour Windows et ScnhellPi pour Linux.
- Récupérer ce fichier sur le site de XLogo :

`http ://downloads.tuxfamily.org/xlogo/common/millionpi.txt`

Pour créer nos nombres aléatoires, bous prendrons des paquets de 8 chiffres dans la suite des décimales de π. Explication, le fichier commence ainsi :

$\underbrace{3.1415926}_{\text{Premier nombre}}$ $\underbrace{53589793}_{\text{Deuxième nombre}}$ $\underbrace{23846264}_{\text{Troisième nombre}}$ 338327950288419716939 etc

J'enlève le « . »du 3.14 .... qui risque de nous ennuyer quand on extraiera les décimales. Bien, tout est en place, nous créons une nouvelle procédure appelée `hasardpi` et modifions légèrement la procédure `test`

```
pour pgcd :a :b
si (reste :a :b)=0 [retourne :b][retourne pgcd :b reste :a :b]
fin

pour test :essais
# On ouvre un flux repéré par le chiffre 1 vers le fichier millionpi.txt
# (ici, supposé être dand le répertoire courant
# sinon utiliser fixerepertoire et un chemin absolu)
ouvreflux 1 "millionpi.txt
# Affecte à la variable ligne la première ligne du fichier millionpi.txt
donne "ligne premier lisligneflux 1
# On initialise la variable compteur à 0
donne "compteur 0
repete :essais [
  si 1=pgcd hasardpi 7 hasardpi 7 [donne "compteur :compteur+1]
]
# On calcule la frequence
donne "f :compteur/:essais
# On affiche la valeur approchée de pi
ecris phrase [approximation de pi:] racine (6/:f)
fermeflux 1
fin
pour hasardpi :n
soit "nombre "
repete :n [
# S'il n'y plus de caractere sur la ligne
si 0=compte :ligne [donne "ligne premier lisligneflux 1]
# On donne à la variable caractere la valeur du premier caractere de la ligne
donne "caractere premier :ligne
# puis on enleve ce premier caractere de la ligne.
donne "ligne saufpremier :ligne
donne "nombre mot :nombre :caractere
]
retourne :nombre
fin

test 10
approximation de pi: 3.4641016151377544
test 100
approximation de pi: 3.1108550841912757
test 1000
approximation de pi: 3.081180112566604
```

```
test 10000
approximation de pi: 3.1403714651066386
test 70000
approximation de pi: 3.1361767950325627
```

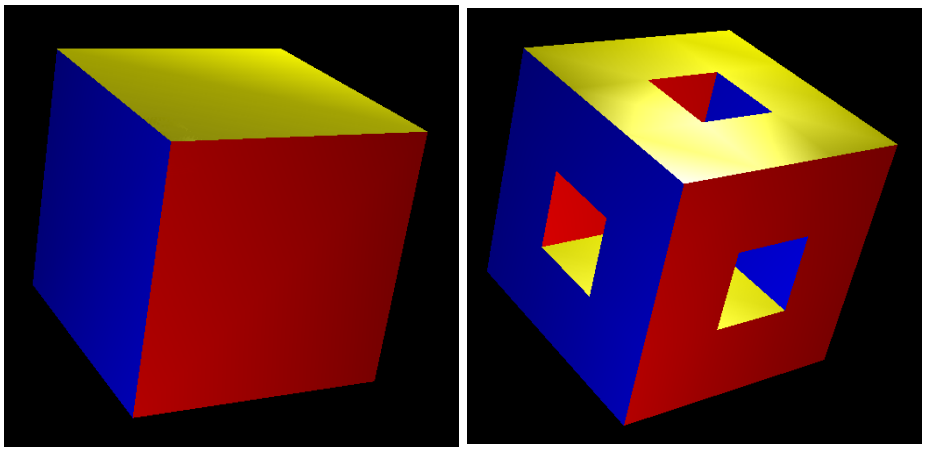
On retrouve donc une approximation du nombre π à l'aide de ses propres décimales !
Il est encore possible d'améliorer ce programme en indiquant par exemple le temps mis pour le calcul. On rajoute alors en première ligne de la procedure test :
`donne "debut temps`
On rajoute juste avant `fermeflux 1` :
`ecris phrase [Temps mis : ] temps - :debut`

Chapitre 13

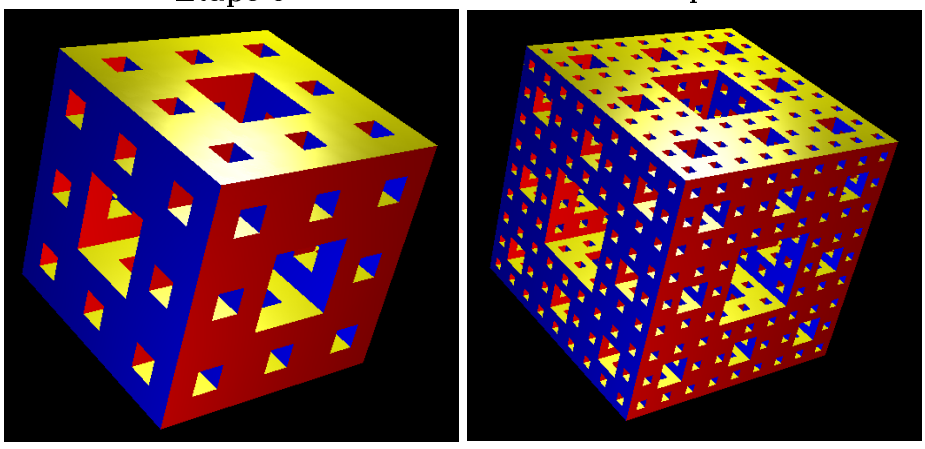
Thème : Eponge de Menger

Niveau : Avancé

Dans ce chapitre, nous allons construire un solide fractal appelé l'éponge de Menger. Voici les premières itérations pour contruire ce solide :



Etape 0 **Etape 1**



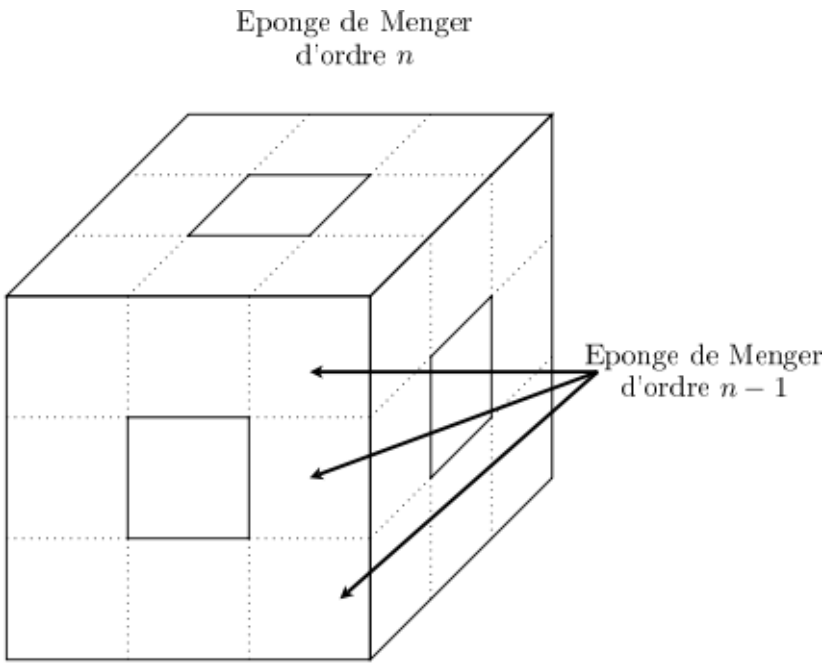
Etape 2 **Etape 3**

Le chapitre se décompose en deux parties :

- Tout d'abord, nous montrerons comment créer ce solide aisément en utilisant la récursivité.
- Dans un deuxième temps, on essaiera d'optimiser le tracé afin de tracer une éponge de Menger d'ordre 4.

13.1 En utilisant la récursivité

Considérons une éponge de menger d'ordre n dont le côté mesure L.



Le schéma montre bien que cette éponge est constitué en fait de 20 éponges de Menger d'ordre $n-1$ ayant chacune un côté de $\frac{L}{3}$. La structure récursive de l'éponge est ainsi mise en évidence.

Le programme :

```
# Commande principale: eponge 3
pour cube :l
si :compteur=10000 [vue3d]
# Couleur des faces latérales
soit "couleurs [jaune magenta cyan bleu]
# faces latérales
repete 4 [fcc exec item compteur :couleurs carre :l td 90 av  :l tg 90 rd 90]
# Dessous
fcc rouge pique 90 carre :l cabre 90
av :l pique 90 fcc vert carre :l cabre 90 re :l
fin

pour carre :c
donne "compteur :compteur+1
polydef
repete 4 [av :c td 90]
polyfin
fin

# Eponde de Menger
# p: profondeur de récursivité
# l: Longueur du grand cube.
pour menger :l :p
si :p=0 [cube :l] [
  soit "p :p-1
  soit "l :l/3
  #face avant
  repete 3 [menger :l :p av :l] re 3*:l
  td 90 av :l tg 90
  menger :l :p av 2*:l menger :l :p re 2*:l
```

```
  td 90 av :l tg 90
  repete 3 [menger :l :p av :l] re 3*:l
  #Côté droit
 pique 90 av :l cabre 90
  menger :l :p  av 2*:l menger :l :p re 2*:l
  pique 90 av :l cabre 90
  repete 3 [menger :l :p av :l] re 3*:l
  tg 90 av :l td 90
  menger :l :p  av 2*:l menger :l :p re 2*:l
  tg 90 av :l td 90
  repete 3 [menger :l :p av :l] re 3*:l
  pique 90 re :l cabre 90
  menger :l :p  av 2*:l menger :l :p re 2*:l
   pique 90 re :l cabre 90
]
fin

pour eponge :p
ve ct donne "compteur 0 perspective fcfg 0 menger 800 :p
tape [Nombre de polygone: ] ec :compteur
vue3d
fin
```

Ce programme comprend quatre procédures :

- `carre :c`
  Cette procédure trace un carré de côté `:c`. De plus, ce polygone est enregistré par le modeleur 3D. La variable `compteur` est chargée de dénombrer le nombre de polygones dessinés.
- `cube :l`
  Cette procédure trace un cube de côté `:l`. Elle utilise bien sûr la procédure `carre`
- `menger :l :p`
  Cette procédure est la clé du programme, elle dessine un motif de Menger d'ordre p et dont le côté mesure l. Ce motif est créé de manière récursive de manière tout à fait naturelle en exploitant le schéma précédent.
- `eponge :p`
  Cette procédure trace une eponge de Menger d'ordre p et de côté 800 puis l'affiche dans le modeleur 3D.

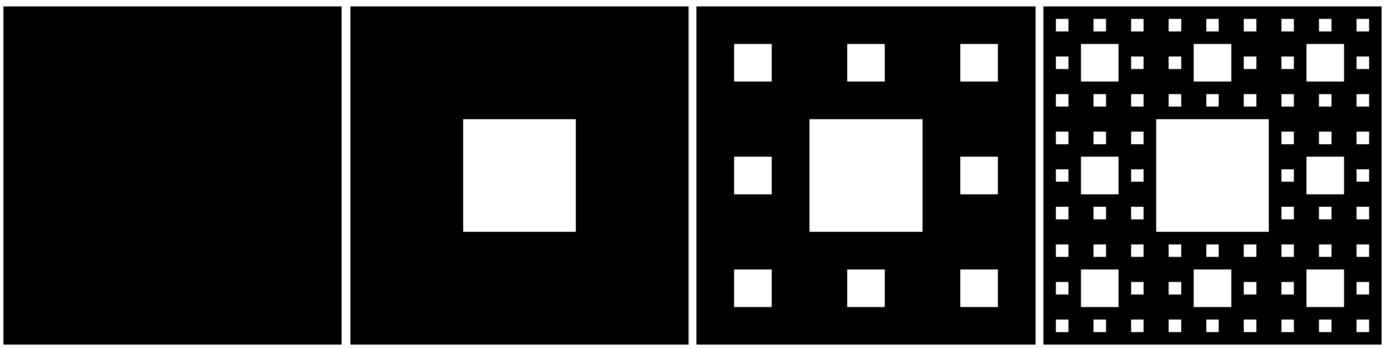
13.2 Deuxième approche : objectif solide d'ordre 4

Le programme précédent a pour principal avantage d'exploiter la structure naturellement récursive du solide fractal. On peut noter que cette même méthode peut être réemployer pour générer d'autres solides fractals, ou plus simplement, d'autres courbes fractales. En tout cas, La conséquence immédiate de l'approche récursive est un code court et simple à comprendre. Malheureusement, on s'aperçoit qu'une éponge d'ordre 3 nécessite déjà 48 000 polygônes. Il faut alors régler la mémoire allouée à XLogo à 256 Mo dans le panneau des préférences pour que le programme puisse s'exécuter entièrement.

Si l'on souhaite tracer une éponge de Menger d'ordre 4, on va vite être bloqué par un dépassement mémoire. Nous allons voir dans cette partie un programme basé sur un algorithme complètement différent, il permettra de créer une éponge de Menger d'ordre 0,1,2,3 ou 4.

13.2.1 Le tapis de Sierpinski

L'éponge de Menger est en fait la généralisation en 3 dimensions d'une figure du plan appelée le tapis de Sierpinski. Voici les premières itérations de cette figure :



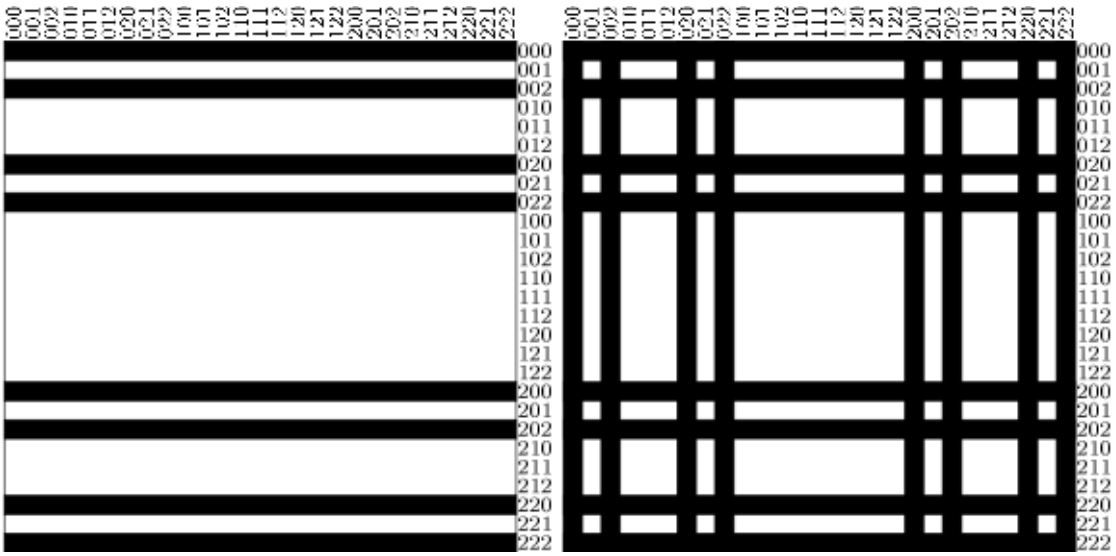
Etape 0 **Etape 1** **Etape 2** **Etape 3**

Le motif présent sur chacune des faces d'une éponge de Menger d'ordre p est un tapis de Sierpinski d'ordre p.

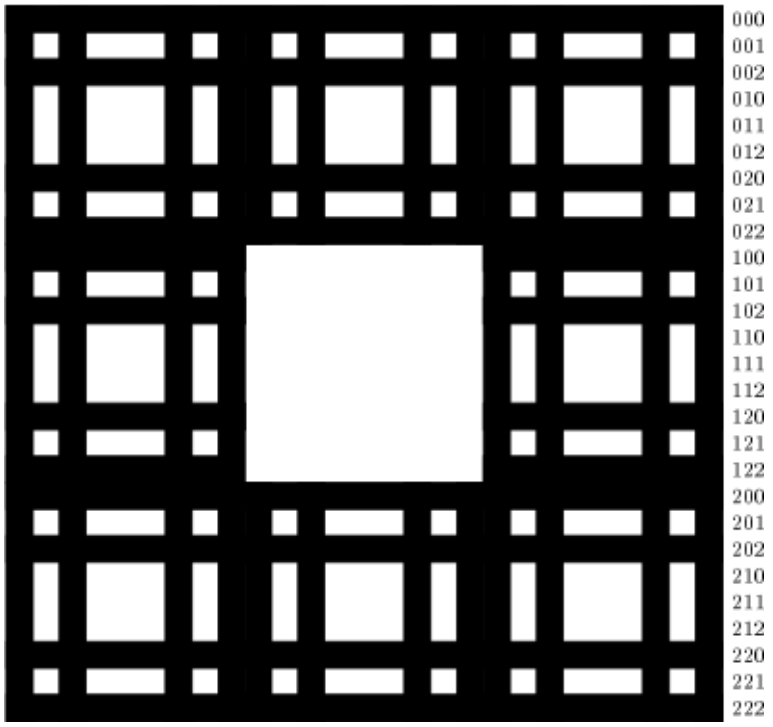
13.2.2 Tracer un tapis de Sierpinski d'ordre p

L'objectif est déjà de réussir à minimiser le nombre de polygônes nécessaires pour dessiner un tapis de Sierpinski. L'exemple suivant explique le procédé employé pour créer un tapis de Sierpinski d'ordre 3. Ici, le carré initial comporte donc $3^3 = 27$ lignes et 27 colonnes. On écrit en base 3 le numéro de chacune des lignes et de chacune des colonnes.

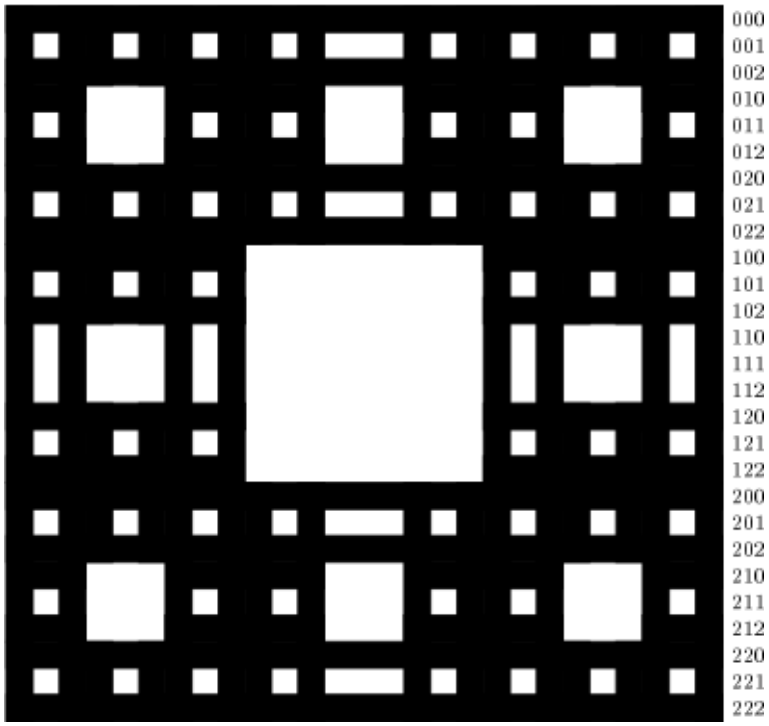
- **Première étape :** Pour toutes les lignes dont le numéro ne comporte aucun 1, on trace une ligne de 27 carreaux. Par symétrie, on effectue la même opération sur les colonnes.



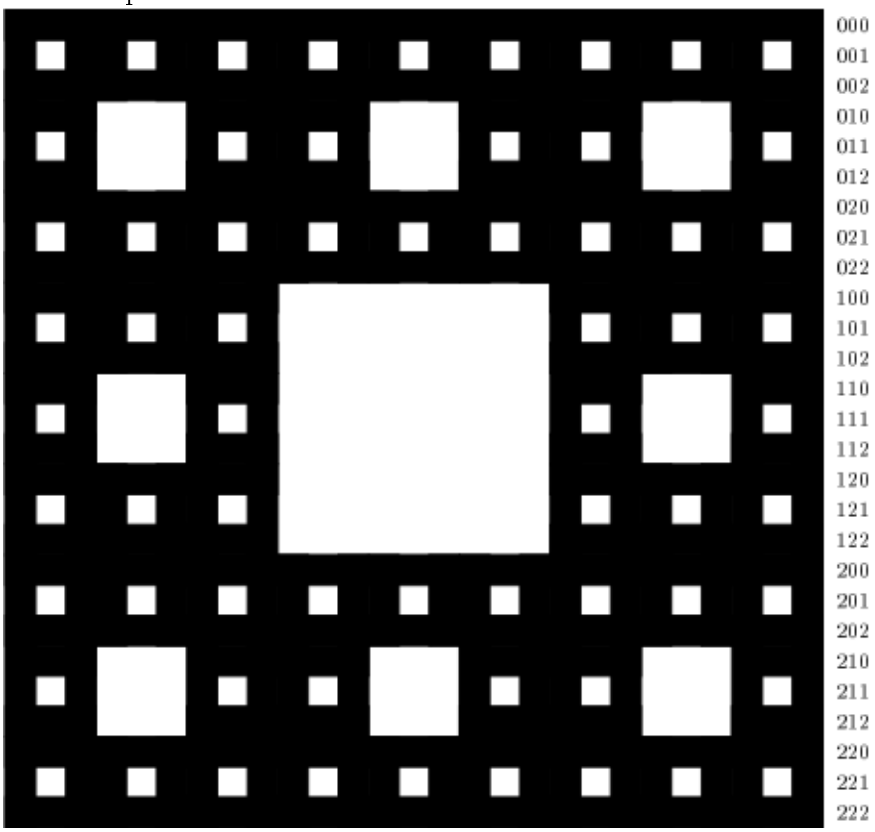
- **Deuxième étape :** On s'intéresse à présent aux lignes dont le numéro comporte un seul 1 en première position. On trace successivement par alternance des rectangles de longueur 9 carreaux. On reporte alors cette opération sur les colonnes par symétrie .



- **Troisième étape :** On s'intéresse à présent aux lignes dont le numéro comporte un seul 1 en deuxième position. On trace successivement par alternance des rectangles en suivant le schéma [3 3 6 3 6 3 3]. (3 carreaux tracés, 3 non tracés, 6 tracés etc...) On reporte alors par symétrie cette opération sur les colonnes.



- **Dernière étape :** On s'intéresse alors aux lignes dont le numéro comporte deux 1 placés aux premières positions. On trace successivement par alternance des rectangles en suivant le schéma [3 3 3 9 3 3 3]. On reporte ensuite cette opération sur les colonnes.



La construction du tapis de Sierpinski d'ordre 3 est alors terminée. Pour créer ce tapis, il a fallu utiliser en tout : $16 + 16 + 32 + 16 = 80$ polygones.

13.2.3 Différents schémas de colonnes possibles

Pour récapituler la construction précédente, voici les différents types de schémas de colonnes suivant leur numéro de lignes. (Le symbole * désigne le chiffre 0 ou le chiffre 2)

Numéro de ligne du type	Schéma à appliquer
***	27
1**	9 9 9
1	3 3 6 3 6 3 3
11*	3 3 3 9 3 3 3

Sur le même principe, pour créer un tapis d'ordre 4, on utilisera un carré avec $3^4 = 81$ carreaux. Les numéros de ligne et de colonne possèderont donc 4 chiffres dans leur décomposition en base 3. Pour chaque type de numéro de lignes, voici le schéma à appliquer (le symbole * désigne le chiffre 0 ou le chiffre 2) :

Numéro de ligne du type	Schéma à appliquer
****	81
1***	27 27 27
*1**	9 9 18 9 18 9 9
**1*	3 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 3
11	3 3 3 9 3 3 6 3 3 9 3 3 6 3 3 9 3 3 3
1*1*	3 3 6 3 6 3 3 27 3 3 6 3 6 3 3
11**	9 9 9 27 9 9 9
111*	3 3 3 9 3 3 3 27 3 3 3 9 3 3 3

496 polygônes sont alors nécessaires pour tracer un tapis de Sierpinski d'ordre 4.

Enfin, voici les chémas de constructions de colonnes pour les solides d'ordre 2 :

Numéro de ligne du type	Schéma à appliquer
**	9
1*	3 3 3

13.2.4 Le programme

```
 #trace un tapis de Sierpinski d'ordre :p et de taille :size
pour carpet :size :p
donne "unit :size/(puissance 3 :p)
si :p=0 [ rec :size :size stop]
si :p=1 [repete 4 [rec :size :unit av :size td 90 ] stop]
repetepour (liste "x 1 puissance 3 :p) [
  soit "cantorx cantor :x :p []
# On ne trace pas les éléments ayant un 1 en derniere position
si  non (1=dernier :cantorx)  [
  soit "nom evalue saufdernier :cantorx "
  drawColumn :x rprop "map :nom
  ]
]
fin

# Retourne la décomposition en base 3 du nombre x
# p indice de profondeur 3^p
# :list liste vide au démarrage

pour cantor :x :p :list
si :p=0 [retourne :list]
soit "a puissance 3 :p-1
si :x<= :a [
  retourne cantor  :x :p-1  phrase :list 0]
  [ si :x<=2*:a [retourne cantor  :x-:a :p-1  phrase :list 1]
  retourne cantor :x-2*:a :p-1 phrase :list 0]
```

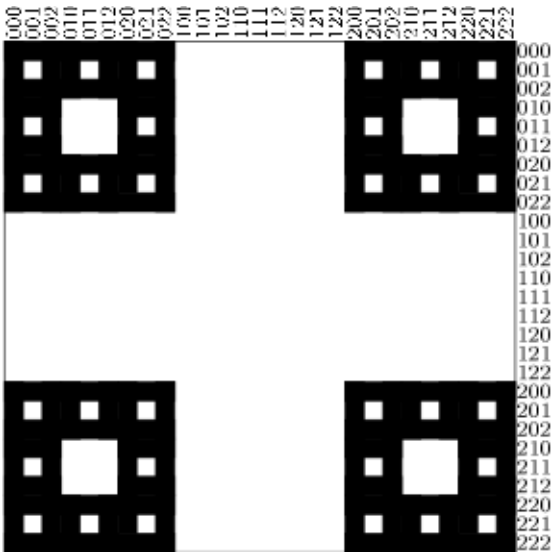
```
fin

# Trace la colonne numéro x en respectant le schéma de construction défini dans la liste
pour drawcolumn :x :list
  lc  td 90 av (:x-1)*:unit tg 90  bc des :list
  lc tg 90 av (:x-1)*:unit td 90 av :x*:unit td 90 bc des :list
lc tg 90 re :x*:unit bc
fin

# Trace un rectangle de dimensions données
# Le polygone est enregistrées par le viewer3d
pour rec :lo :la
donne "compteur :compteur+1
polydef
repete 2 [av :lo td 90 av :la td 90]
polyfin
fin

# Initialise les différentes colonnes possibles pour les tapis d'ordre 1 à 4
pour initmap
dprop "map 111 [3 3 3 9 3 3 3 27 3 3 3 9 3 3 3]
dprop "map 110 [9 9 9 27 9 9 9]
dprop "map 101 [3 3 6 3 6 3 3 27 3 3 6 3 6 3 3]
dprop "map 011 [3 3 3 9 3 3 6 3 3 9 3 3 6 3 3 9 3 3 3]
dprop "map 000 [81]
dprop "map 100 [27 27 27]
dprop "map 010 [9 9 18 9 18 9 9]
dprop "map 001 [3 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 3]
dprop "map 01 [3 3 6 3 6 3 3]
dprop "map 00 [27]
dprop "map 10 [9 9 9]
dprop "map 11 [3 3 3 9 3 3 3]
dprop "map 1 [3 3 3]
dprop "map 0 [9]
fin

# Si la decomposition est [1 0 1] --> retourne 101
pour evalue :list :mot
  si vide? :list [retourne :mot]
  [
  soit "mot mot :mot premier :list
  retourne evalue saufpremier :list :mot
]
fin
# Trace les blocs de rectangles de chaque colonne par alternance
pour des :list
soit "somme 0
repetepour (liste "i 1 compte :list) [
   soit "element item :i :list
    soit "somme :element+:somme
  si pair? :i [lc av :element*:unit bc ] [rec :element*:unit :unit av :element*:unit]
]
lc re  :somme * :unit bc
```

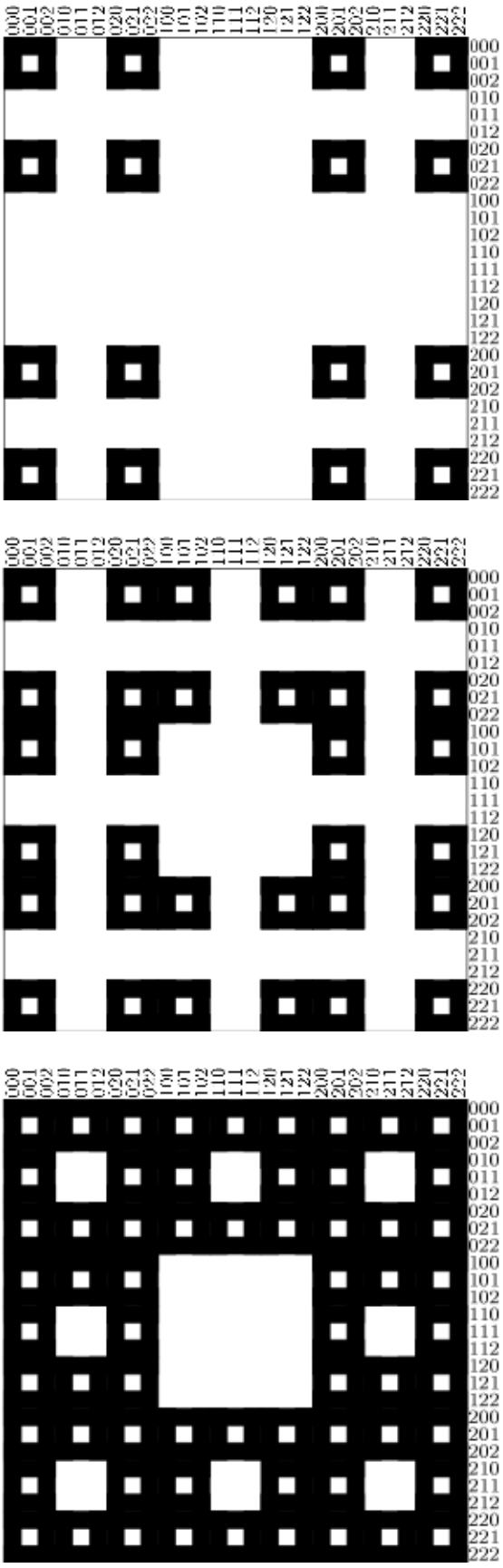
```
fin

# Teste si un nombre est pair
pour pair? :i
retourne 0=reste :i 2
fin

pour tapis :p
ve perspective ct initMap
donne "compteur 0
carpet 810 :p
tape "Nombre\ de\ polygones:\  ec :compteur
vue3d
fin
```

`tapis 3` dessine un tapis de Sierpinski d'ordre 3 de côté 810. Voilà, nous sommes prêts à passer à l'éponge de Menger !

13.2.5 L'éponge de Menger d'ordre 4

L'éponge de Menger possède de multiples propriétés de symétrie. Pour la générer nous allons tracer les différentes sections suivant le plan (xOy) puis reporter ces figures suivant (yOz) et (xOz). Pour bien expliquer ce qui se passe, restons sur l'exemple de l'éponge d'ordre 3 :
Lorsque l'on coupe l'éponge par un plan vertical, on peut obtenir quatre motifs différents :





Pour tracer une éponge d'ordre 3, nous allons parcourir les nombres de 1 à 27, c'est à dire de 001 à 222 en base 3. Pour chaque numéro, on appliquera la section adéquate que l'on reportera suivant les 3 directions (Ox), (Oy) et (Oz).

Le code

Le programme suivant permet de tracer les solides de Menger d'ordre 0,1,2,3,4. Le nombre de procédures est important donc j'apporterai quelques éclaircissements ensuite.

```
#trace un tapis de Sierpinski d'ordre :p et de taille :size
pour carpet :size :p
donne "unit :size/(puissance 3 :p)
si :p=0 [ rec :size :size stop]
si :p=1 [repete 4 [rec :size :unit av :size td 90 ] stop]
repetepour (liste "x 1 puissance 3 :p) [
  soit "cantorx cantor :x :p []
# On ne trace pas les éléments ayant un 1 en derniere position
si  non (1=dernier :cantorx)  [
  soit "nom evalue saufdernier :cantorx "
  drawColumn :x rprop "map :nom
  ]
]
fin

# Retourne la décomposition en base 3 du nombre x
# p indice de profondeur 3^p
# :list liste vide au démarrage

pour cantor :x :p :list
si :p=0 [retourne :list]
soit "a puissance 3 :p-1
si :x<= :a [
  retourne cantor  :x :p-1  phrase :list 0]
  [ si :x<=2*:a [retourne cantor  :x-:a :p-1  phrase :list 1]
  retourne cantor :x-2*:a :p-1 phrase :list 2]
fin

# Trace la colonne number x en respectant le schéma de construction défini dans la liste
pour drawcolumn :x :list
  lc  td 90 av (:x-1)*:unit tg 90  bc des :list
  lc tg 90 av (:x-1)*:unit td 90 av :x*:unit td 90 bc des :list
lc tg 90 re :x*:unit bc
fin

# Trace un rectangle de dimensions données
# Le polygone est enregistrées par le viewer3d
pour rec :lo :la
donne "compteur :compteur+1
polydef
repete 2 [av :lo td 90 av :la td 90]
polyfin
fin

# Initialise les différentes colonnes possibles pour les tapis d'ordre 1 à 4
pour initmap
dprop "map 111 [3 3 3 9 3 3 3 27 3 3 3 9 3 3 3]
dprop "map 110 [9 9 9 27 9 9 9]
dprop "map 101 [3 3 6 3 6 3 3 27 3 3 6 3 6 3 3]
dprop "map 011 [3 3 3 9 3 3 6 3 3 9 3 3 6 3 3 9 3 3 3]
```

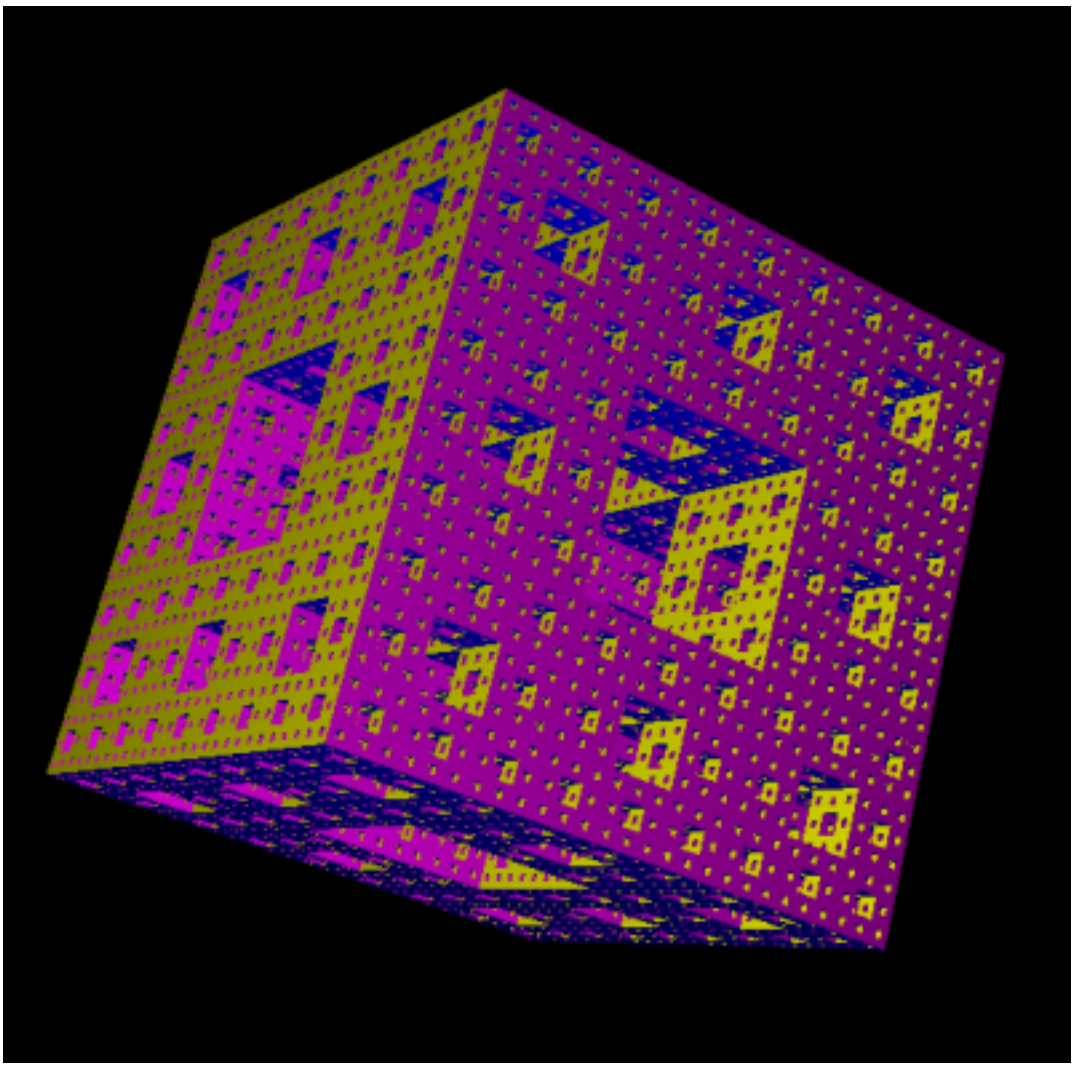
```
dprop "map 000 [81]
dprop "map 100 [27 27 27]
dprop "map 010 [9 9 18 9 18 9 9]
dprop "map 001 [3 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 3]
dprop "map 01 [3 3 6 3 6 3 3]
dprop "map 00 [27]
dprop "map 10 [9 9 9]
dprop "map 11 [3 3 3 9 3 3 3]
dprop "map 1 [3 3 3]
dprop "map 0 [9]
fin

# Si la decomposition est [1 0 1] --> retourne 101
# Si la decomposition est [1 0 2] --> retourne 100
#  Les éléments de la liste sont concaténés en un mot.
# De plus, es 2 sont remplacés par des zéros
pour evalue :list :mot
  si vide? :list [retourne :mot]
  [
  soit "first premier :list
  si :first=2 [soit "first 0]
 soit "mot mot :mot :first
  retourne evalue saufpremier :list :mot
]
fin
# Trace les blocs de rectangles de chaque colonne par alternance
pour des :list
soit "somme 0
repetepour (liste "i 1 compte :list) [
   soit "element item :i :list
    soit "somme :element+:somme
  si pair? :i [lc av :element*:unit bc ] [rec :element*:unit :unit av :element*:unit]
]
lc re  :somme * :unit bc
fin

# Teste si un nombre est pair
pour pair? :i
retourne 0=reste :i 2
fin

pour tapis :p
ve perspective ct initMap
donne "compteur 0
carpet 810 :p
tape "Nombre\ de\ polygones:\  ec :compteur
vue3d
fin

# Supprime le dernier 1 dans la liste :list
pour deletelastone :list
repetepour (liste "i compte :list 1 moins 1) [
  soit "element item :i :list
```

```
  si :element=1 [soit "list remplace :list :i 0 stop] [si :element=2 [stop]]
]
retourne :list
fin

# Eponge de Menger de taille donnée et de profondeur :p

pour menger :size :p
donne "unite :size/(puissance 3 :p)
repetepour (liste "z 1 puissance 3 :p) [
  soit "cantorz cantor :z :p []
  soit "last dernier :cantorz
  soit "cantorz saufdernier :cantorz
  si :last=0 [soit "order evalue deleteLastOne :cantorz "]  [soit "order evalue :cantorz "]
  soit "order mot "coupe :order
  draw3carpet :size :order :z
  lc cabre 90 av :unit pique 90 bc
]
draw3carpet :size :order (puissance 3 :p)+1
fin

# Trace les tapis de Sierpinski d'ordre :p
# suivant chaque axe (Ox), (Oy) et (Oz)
# à l'altitude :z
pour draw3carpet :size :order :z
lc origine
cabre 90 av (:z-1)*:unite pique 90 bc
fcc bleu exec :order :size
lc origine
rg 90 av (:z-1)*:unite pique 90  bc
fcc jaune exec :order :size
lc origine
cabre 90 av :size td 90 av (:z-1)*:unite pique 90 bc
fcc magenta exec :order :size
fin


# Procédure principale
# Trace une eponge de Menger de profondeur p
pour eponge :p
ve perspective ct
soit "temps temps
initMap
donne "compteur 0
si :p=0 [cube 405] [menger 405 :p]
# Affiche le temps mis et le nombre de polygone nécessaire à la construction
tape "Nombre\ de\ polygones:\  ec :compteur
tape "Temps\ mis:\  ec temps -:temps
vue3d
fin

# Section pour le Menger d'ordre 2
pour coupe1 :size
```

```
repete 4 [carpet :size/3 1 lc av :size td 90 bc]
fin

pour coupe0 :size
carpet :size 2
fin

# Section pour le Menger d'ordre 3

pour coupe10 :size
repete 4 [carpet :size/3 2 lc av :size td 90 bc]
fin

pour coupe01 :size
repete 4 [repete 2 [coupe1 :size/3 lc av :size/3 bc] av :size/3 td 90]
fin

pour coupe11 :size
repete 4 [coupe1 :size/3 lc av :size td 90 bc]
fin


pour coupe00 :size
carpet :size 3
fin

# Section pour le Menger d'ordre 4
pour coupe000 :size
carpet :size 4
fin

pour coupe100 :size
repete 4 [carpet :size/3 3 lc av :size td 90 bc]
fin

pour coupe010 :size
repete 4 [repete 2 [coupe10 :size/3 lc av :size/3 bc] av :size/3 td 90]
fin

pour coupe001 :size
repete 4 [repete 2 [coupe01 :size/3 lc av :size/3 bc] av :size/3 td 90]
fin

pour coupe110 :size
repete 4 [coupe10 :size/3 lc av :size bc td 90 ]
fin

pour coupe111 :size
repete 4 [coupe11 :size/3 lc av :size td 90 bc]
fin

pour coupe101 :size
repete 4 [coupe01 :size/3 lc av :size td 90 bc]
```

```
fin

pour coupe011 :size
repete 4 [repete 2 [coupe11 :size/3 lc av :size/3 bc] av :size/3 td 90]
fin

pour coupe :size
carpet :size 1
fin

pour cube :size
repete 2 [
fcc bleu rec :size :size lc av :size pique 90 bc
fcc jaune rec :size :size lc av :size pique 90  bc
]
fcc magenta
lc rg 90 tg 90 av :size td 90 bc rec :size :size
lc td 90 av :size tg 90 rd 90 td 90 av :size tg 90 rd 90 bc rec :size  :size
rg 90 tg 90 av :size td 90
fin

pour cubes
ve perspective ct
soit "temps temps
initMap
donne "compteur 0
repete 4 [si compteur=1 [cube 405] [menger 405 compteur-1] lc av 1000 td 90 bc ]
# Affiche le temps mis et le nombre de polygone nécessaire à la construction
tape "Nombre\ de\ polygones:\  ec :compteur
tape "Temps\ mis:\  ec temps -:temps
vue3d
fin
```

Ensuite, on règle la mémoire allouée à XLogo à 640 Mo : `eponge 4`



Chapitre 14

Thème : Système de Lindenmayer

Niveau : Avancé

Dans cette partie, certaines références sont issues :

- de la page Wikipedia sur les L-system : `http ://fr.wikipedia.org/wiki/L-System`.
- du livre « The Algorithmic Beauty of Plants »écrit par Przemyslaw Prusinkiewicz et Aristid Lindenmayer.

Cette partie va traiter de la notion de système de Lindenmayer ou L-system inventé en 1968 par le biologiste hongrois Aristid Lindenmayer. Un L-System est un ensemble de règles et de symboles qui modélisent un processus de croissance d'êtres vivants comme des plantes ou des cellules. Le concept central des L-Systems est la notion de réécriture. La réécriture est une technique pour construire des objets complexes en remplaçant des parties d'un objet initial simple en utilisant des règles de réécriture.

Pour ce faire, les cellules sont modélisées à l'aide de symboles. À chaque génération, les cellules se divisent, i.e. un symbole est remplacé par un ou plusieurs autres symboles formant un mot.

14.1 Définition formelle

Un L-System est une grammaire formelle qui comprend :

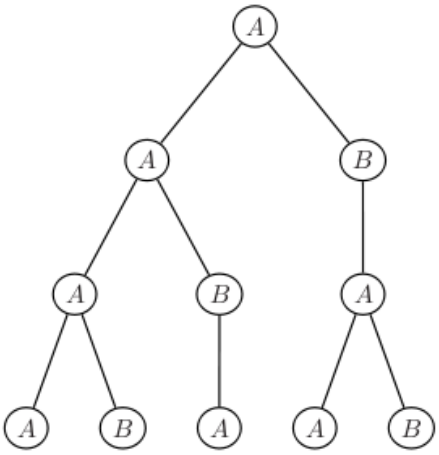
1. Un alphabet V : l'ensemble des variables du L-System. $V*$ est l'ensemble des "mots" que l'on peut construire avec les symboles de V, et $V+$ l'ensemble des mots contenant au moins un symbole.
2. Un ensemble de valeur constantes S. Certains de ces symboles sont communs à tous les L-System. (Notamment quand on utilisera la tortue).
3. Un axiome de départ ω choisi parmi $V+$, c'est à dire l'état initial.
4. Un ensemble de règles, noté P, de reproduction des symboles de V.

Un L-System est alors noté $\{V, S, \omega, P\}$.

Considérons le L-system suivant :

- Alphabet : $V = \{A, B\}$
- Constantes : $S = \{\emptyset\}$
- Axiome de départ : $\omega = A$
- Régles : $\boxed{\begin{array}{l} A \rightarrow AB \\ B \rightarrow A \end{array}}$

Les deux règles qui sont données sont les règles de réécriture du système. A chaque étape, A est remplacé par la séquence AB, et B est remplacé par A. Voici les premières itérations de ce système de Lindemayer :



- Itération 1 : A
- Itération 2 : AB
- Itération 3 : ABA
- Itération 4 : $ABAAB$

Bien, bien... et concrètement ? ? Lisons la suite !

14.2 Interprétation par la tortue

Ce premier exemple vous a permis d'appréhender la notion de système de Lindenmayer sans toutefois peut-être discerner comment nous allons utiliser cela concrètement avec la tortue.

C'est là que cela devient intéressant : Chacun des mots ainsi construits ne possède pas de signification particulière. On va alors attacher à chacune des lettres de la séquence, une commande à exécuter par la tortue et générer ainsi des dessins en 2D ou en 3D.

14.2.1 Symboles usuels

- F : Se déplacer d'un pas unitaire ($\in V$)
- $+$: Tourner à gauche d'angle α ($\in S$).
- $-$: Tourner à droite d'un angle α ($\in S$).
- & : Pivoter vers le bas d'un angle α ($\in S$).
- ^ : Pivoter vers le haut d'un angle α ($\in S$).
- \ : Roulez vers la gauche d'un angle α ($\in S$).
- / : Roulez vers la droite d'un angle α ($\in S$).
- | : Effectuer un demi-tour. Avec XLogo : `td 180`

Prenons par exemple $\alpha = 90$ et un déplacement unitaire de 10 pas de tortue, on obtient alors :

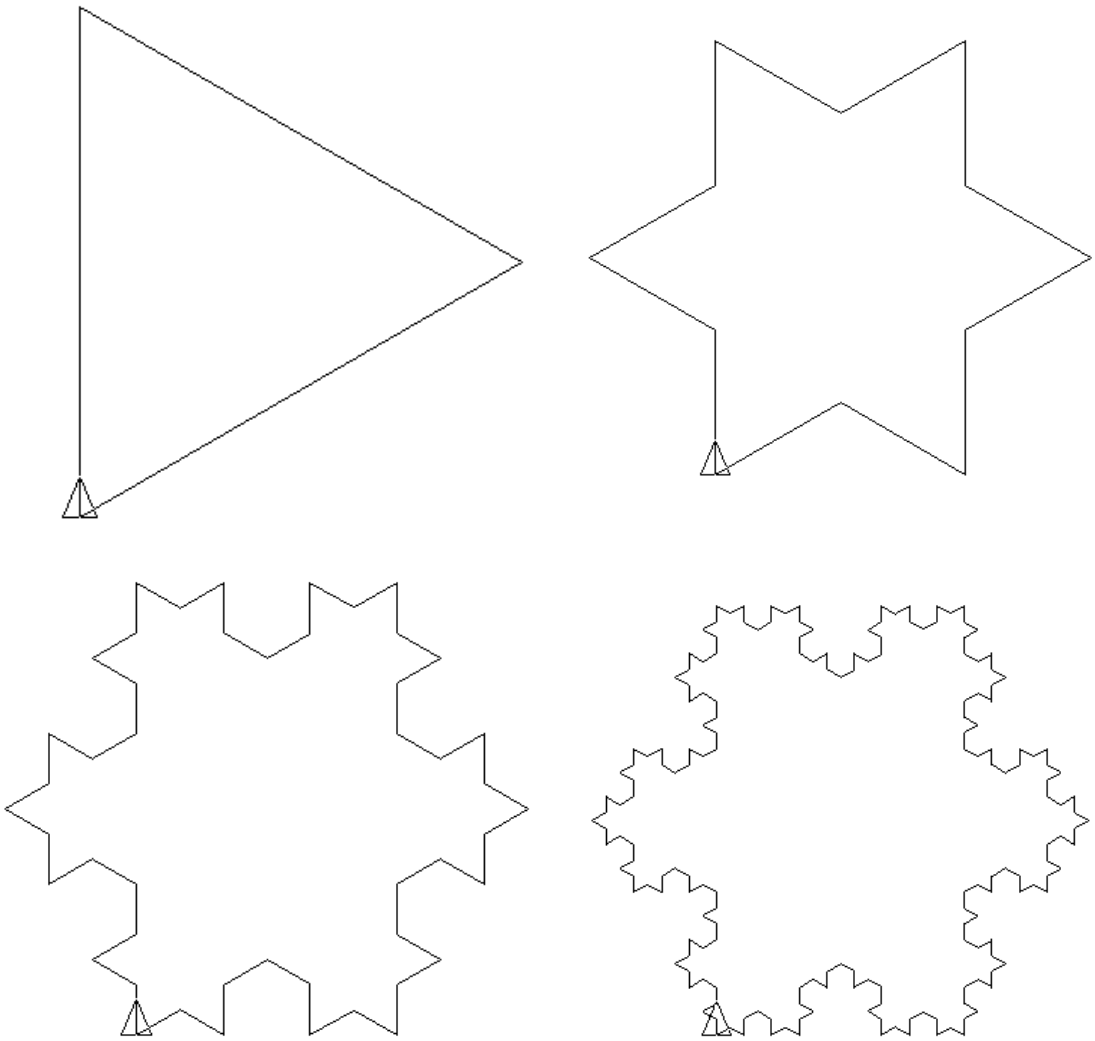
Symbole	F	$+$	$-$	&	^	\	/	|
Commande XLogo	`av 10`	`tg 90`	`td 90`	`pique 90`	`cabre 90`	`rg 90`	`rd 90`	`td 180`

14.2.2 Flocon de Koch

Considérons le L-system :

- Etat initial : $F--F--F--$
- Règle de production : $F \to F+F--F+F$
- Angle $\alpha = 60^\circ$, le pas unitaire est divisé par 3 entre chaque itération.

Premières itérations :



Programme en Logo :

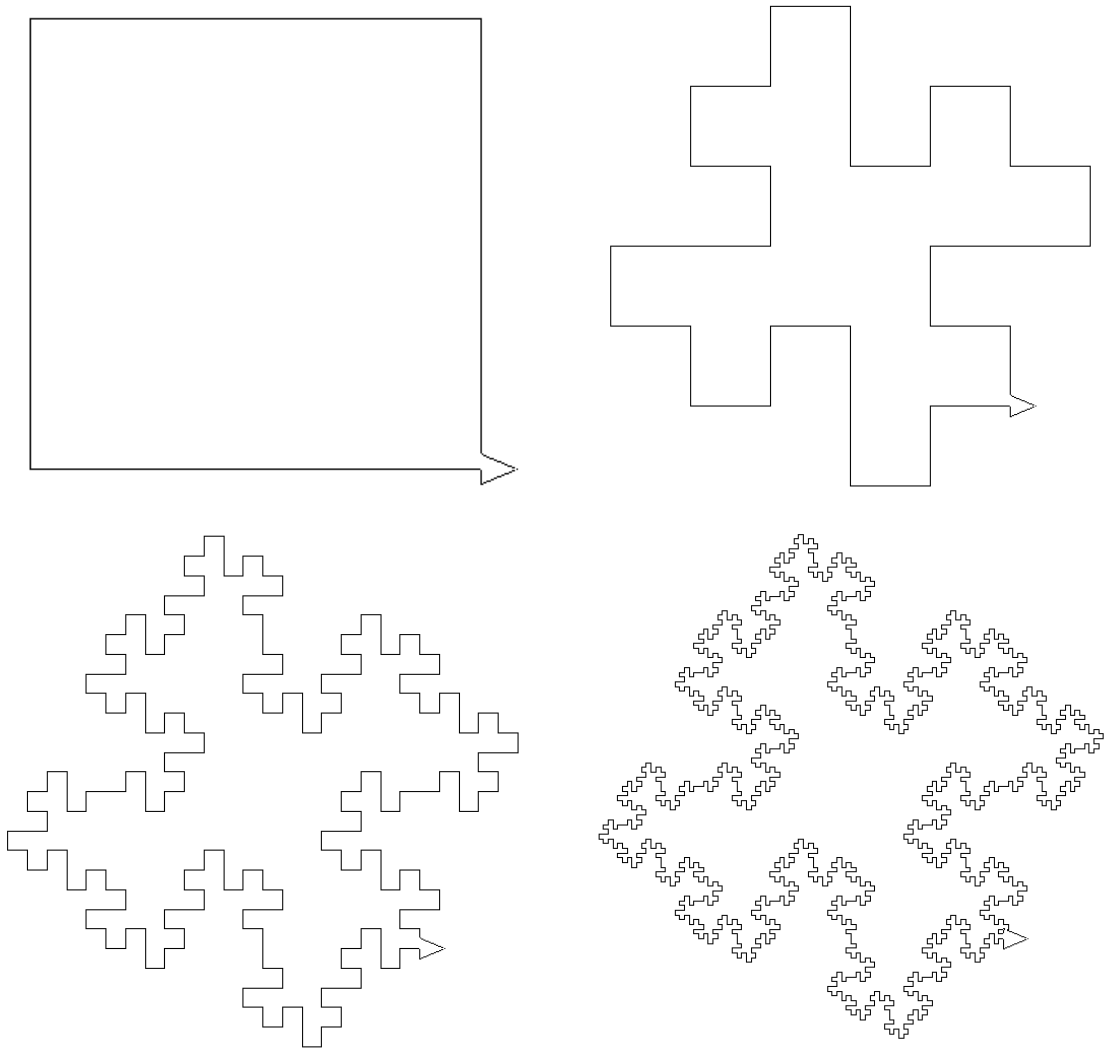
```
pour flocon :p
donne "unit 300/puissance 3 :p-1
repete 3 [F :p-1 td 120]
fin

pour f :p
si :p=0 [av :unit stop]
F :p-1 tg 60 F :p-1 td 120 F :p-1 tg 60
F :p-1
fin
```

14.2.3 Courbe de Koch d'ordre 2

Intéressons-nous le L-system suivant :

- Etat initial : $F - F - F - F$
- Règle de production : $F \to F - F + F + FF - F - F + F$

Voici les premières représentations en utilisant $\alpha = 90$ et en ajustant le pas unitaire de telle sorte que la figure fasse toujours la même taille :



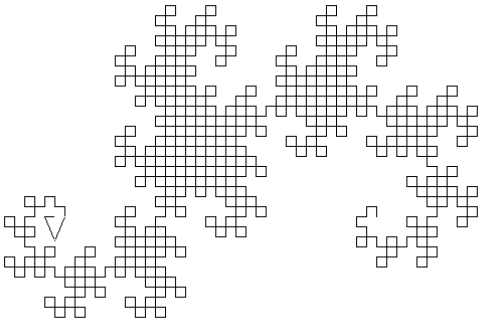
Il est alors très facile de créer le programme Logo permettant de générer ces dessins :

```
# p désigne l'itération
pour koch :p
# Entre chaque itération, la distance unitaire est divisée par 4
# Ici, la figure finale aura une taille de 600x600 au maximum
donne "unit 300/puissance 4 :p-1

repete 3 [F :p-1 tg 90] F :p-1
fin

# La chaine de réécriture
pour F :p
si :p=0 [av :unit stop]
F :p-1 tg 90 F :p-1 td 90 F :p-1 td 90
F :p-1 F :p-1 tg 90 F :p-1 tg 90 F :p-1 td 90 F :p-1
fin
```

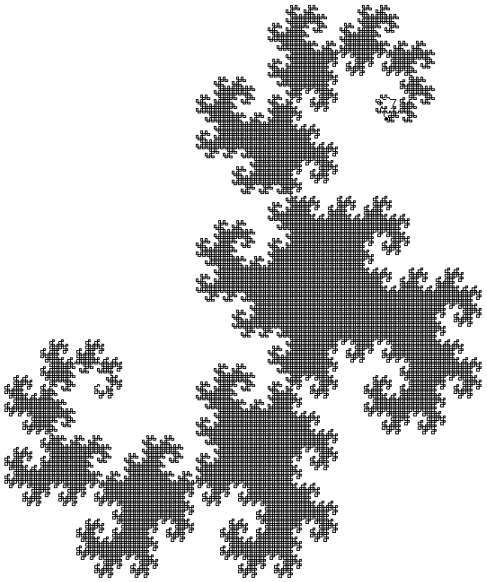
14.2.4 Courbe du dragon

- Etat initial : F
- Règle de production : $\boxed{\begin{array}{l} A \to A + B + \\ B \to -A - B \end{array}}$

```
pour a :p
si :p=0 [av :unit stop]
a :p-1 tg 90 b :p-1 tg 90
fin

pour b :p
si :p=0 [av :unit stop]
td 90 a :p-1 td 90 b :p-1

fin

pour dragon :p
donne "unit 300/8/ :p
a :p
fin
```



dragon 10



dragon 15

14.2.5 Courbe de Hilbert en 3D

L'exemple suivant traite de la courbe de Hilbert dans l'espace, c'est une courbe qui a la propriété de remplir parfaitement un cube quand on augmente le nombre d'itérations.

Voici le L-system associé :

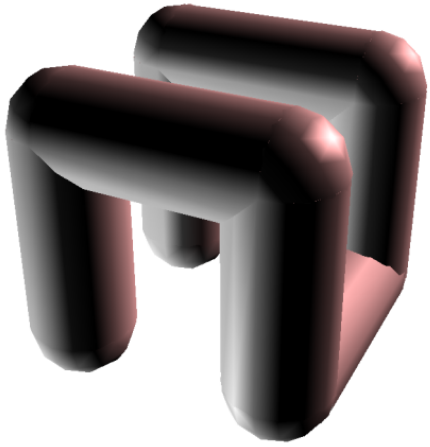
- Etat initial : A
- Angle $\alpha = 90^\circ$, on divise la longueur unitaire par deux à chaque itération.
- Règle de production :

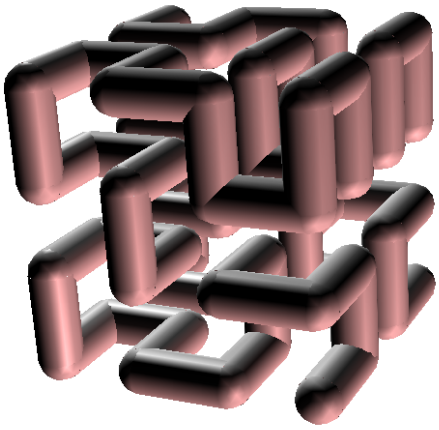
$$\begin{array}{|l|}\hline A \rightarrow B-F+CFC+F-D\&F\hat{}D-F+\&\&CFC+F+B// \\ B \rightarrow A\&F\hat{}CFB\hat{}F\hat{}D\hat{}\hat{}-F-D\hat{}|F\hat{}B|FC\hat{}F\hat{}A// \\ C \rightarrow |D\hat{}|F\hat{}B-F+C\hat{}F\hat{}A\&\&FA\&F\hat{}C+F+B\hat{}F\hat{}D// \\ D \rightarrow |CFB-F+B|FA\&F\hat{}A\&\&FB-F+B|FC// \\ \hline \end{array}$$

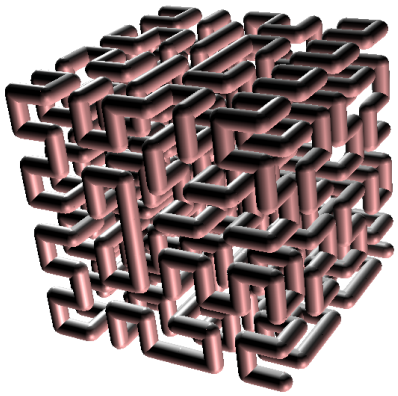
```
pour hilbert :p
ve perspective
donne "unit 400/puissance 2 :p
lignedef ftc :unit/2
a :p
```

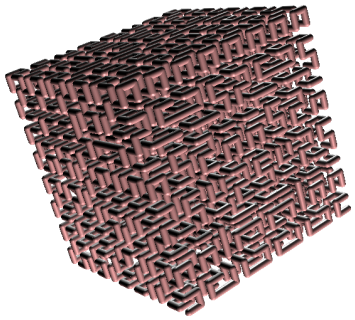
```
lignefin
vue3d
fin

pour a :p
si :p=0 [stop]
b :p-1 td 90 av :unit tg 90  c :p-1 av :unit c :p-1
tg 90 av :unit td 90 d :p-1 pique 90 av :unit cabre 90 d :p-1
td 90 av :unit tg 90 pique 180 c :p-1 av :unit c :p-1
tg 90 av :unit tg 90 b :p-1 rd 180
fin

pour b :p
si :p=0 [stop]
a :p-1 pique 90 av :unit cabre 90 c :p-1 av :unit b :p-1 cabre 90
av :unit cabre 90 d :p-1 cabre 180 td 90 av :unit td 90 d :p-1 cabre 90
td 180 av :unit cabre 90 b :p-1 td 180 av :unit c :p-1 cabre 90 av :unit
cabre 90 a :p-1 rd 180
fin

pour c :p
si :p=0 [stop]
td 180 d :p-1 cabre 90 td 180 av :unit cabre 90 b :p-1 td 90 av :unit tg 90
c :p-1 cabre 90 av :unit cabre 90 a :p-1 pique 180 av :unit a :p-1 pique 90
av :unit cabre 90 c :p-1 tg 90 av :unit tg 90 b :p-1 cabre 90 av :unit cabre 90
d :p-1 rd 180
fin

pour d :p
si :p=0 [stop]
td 180 c :p-1 av :unit b :p-1 td 90 av :unit tg 90 b :p-1 td 180
av :unit a :p-1 pique 90 av :unit cabre 90 a :p-1 pique 180 av :unit
b :p-1 td 90 av :unit tg 90 b :p-1 td 180 av :unit c :p-1 rd 180
fin
```

Et les premières itérations obtenues :









Annexe A

Liste des primitives

Comme il a été indiqué auparavant, le contrôle de la tortue s'effectue à l'aide de commandes internes appelées « primitives ». Voici une classification de ces primitives :

A.1 Déplacement de la tortue, gestion du crayon et des couleurs

A.1.1 Déplacement

Ce premier lot de primitives permet de déplacer la tortue.

avance, av *n*
Fait avancer de n pas la tortue suivant l'orientation courante.

recule, re *n*
Fait reculer de n pas la tortue suivant l'orientation courante.

tournedroite, td *n*
Fait tourner la tortue de n degrés vers la droite par rapport à son orientation actuelle.

tournegauche, tg *n*
Fait tourner la tortue de n degrés vers la gauche par rapport à son orientation actuelle.

cercle *R*
Trace un cercle de rayon R autour de la tortue.

arc *R cap1 cap2*
Trace un arc de cercle de rayon R autour de la tortue. Cette arc est compris entre les caps cap1 et cap2.

origine
Replace la tortue à sa position initiale, c'est à dire au point de coordonnées [0 0] et avec pour cap 0

fixeposition, fpos *liste*
Déplace la tortue au point de coordonnées spécifié à l'aide de la liste des deux nombres.(abscisse puis ordonnée)

fixex *x*
Déplace la tortue horizontalement jusqu'au point d'abscisse x

fixey *y*
Déplace la tortue verticalement jusqu'au point d'ordonnée y

fixexy *x y*
Analogue à fpos[x y]

fixecap *n*
Oriente la tortue au cap spécifié. 0 correspond à la position verticale vers le haut. On tourne ensuite dans le sens des aiguilles d'une montre.

etiquette *mot-liste*
Dessine le mot ou la liste spécifiée à l'endroit où se trouve la tortue et suivant son inclinaison.
Exemple : `etiquette [Salut à toi]` va écrire la phrase « Salut à toi »à l'endroit où est placé la tortue en respectant le cap de celle-ci.

point *liste*
Le point défini par les coordonnées de la liste s'allume (avec la couleur du crayon).

A.1.2 Propriétés de la tortue

Les primitives présentées ici permettent d'agir sur les propriétés de la tortue. Par exemple, faut-il que la tortue soit visible à l'écran ? De quelle couleur doit-elle écrire lorsqu'elle se déplace ?

montretortue, mt
Rend la tortue visible à l'écran.

cachetortue, ct
Rend la tortue invisible à l'écran.

videecran, ve
Efface la zone de dessin et réinitialise la tortue à sa position initiale.

nettoie
Efface la zone de dessin mais laisse la tortue au même endroit.

init
Efface la zone de dessin et initialise à leurs valeurs par défaut un certain nombre de paramètre :

- Couleur du crayon : Noir
- couleur de l'écran : Blanc
- mode Animation : désactivé
- police pour les zones graphique et d'historiquet : Dialog 12 pts
- forme du crayon : carré
- qualité du dessin : normal
- nombre maximum de tortues : 16
- mode trace : désactivé
- taille de l'écran : 1000x1000

baissecrayon, bc

La tortue écrit lorsqu'elle se déplace.

levecrayon,lc
La tortue n'écrit plus lors d'un déplacement.

gomme, go
La tortue efface tous les traits qu'elle rencontre.

inversecrayon, ic
Abaisse le crayon et met la tortue en mode d'inversion.

dessine, de
Abaisse le crayon et le met en mode dessin classique.

fixecouleurcrayon, fcc *entier-liste[r g b]*
Fixe la couleur du crayon. Voir p.93.

fixecouleurfond, fcfg *entier-liste[r g b]*
Fixe la couleur du fond d'écran. Voir p.93.

pos
Retourne la position courante de la tortue. Ex : `pos` retourne [10 -100]

x
Retourne l'abscisse de la position courante de la tortue.

y
Retourne l'ordonnée de la position courante de la tortue.

z
Retourne la cote de la position courante de la tortue (valable uniquement dans l'espace).

cap
Retourne le cap de la tortue (cf `fixecap`)

vers *liste*
La liste doit contenir deux nombres représentant des coordonnées. Rend le cap qu'il faut donner à la tortue pour aller vers le point défini par les coordonnées de la liste.

distance *liste*
La liste doit contenir deux nombres représentant des coordonnées. Rend le nombre de pas entre la position actuelle et le point défini par les coordonnées de la liste.

couleurcrayon, cc
Retourne la couleur actuelle du crayon. Cette couleur est déterminée à l'aide d'une liste [r g b] ou r est la composante rouge, b la bleue et g la verte.

couleurfond, cf
Retourne la couleur actuelle du fond. Cette couleur est déterminée à l'aide d'une liste [r g b] ou r est la composante rouge, b la bleue et g la verte.

enroule, enr
Configure le mode de fenêtrage. Si la tortue sort de la zone de dessin, elle réapparaît de l'autre côté !

fenetre, fen
Configure le mode de fenêtrage. La tortue est libre de sortir de la zone de dessin. Bien sûr, elle n'écrira pas en dehors de cette dernière.

clos
Configure le mode de fenêtrage. La tortue est confinée à la zone de dessin. Si elle s'apprête à sortir, un message d'erreur vous l'indiquera et vous donnera le nombre de pas maximum de la tortue avant sortie (à 1 ou 2 pas près ...).

perspective
Configure le mode de fenêtrage. La tortue peut à présent s'orienter dans l'espace. (Allez voir la section A.2 dédiée à ce mode). Pour sortir de ce mode, utiliser la primitive `fenetre`, `enroule` ou `clos`

trouvecouleur *liste*
Retourne la couleur du pixel de coordonnées a. Cette couleur est déterminée à l'aide d'une liste [r g b] ou r est la composante rouge, b la bleue et g la verte.

fixetaillecrayon, ftc *nombre*
Définit l'épaisseur de la pointe du crayon en pixel. Réglé sur 1 par défaut.

taillecrayon, tc
Renvoie l'épaisseur de la pointe du crayon en pixel.

ffc, fixeformecrayon *0-1*
Fixe la forme de la mine du crayon.

- 0→Carré.
- 1→Rond.

fc, formecrayon
Renvoie la forme de la mine du crayon. 0→Carré. 1→Rond.

fqd, fixequalitedessin *0-1-2*
Fixe la qualité du dessin.

- 0→normal.
- 1→haute.
- 2→basse.

qd, qualitedessin
Renvoie la qualité du dessin.

- 0→normal.
- 1→haute.

- 2→basse.

ftd, fixetailledessin *liste*
Fixe la taille de la zone de dessin.

tailledessin
Renvoie la taille de la zone de dessin.

fixeforme, fforme *entier*
Vous pouvez choisir de l'aspect de la tortue utilisée soit en allant dans Option-Préférences-Choix de la tortue soit à l'aide de cette primitive. Le nombre n doit être un entier compris entre 0 et 6. (0 désigne la forme triangulaire)

forme
Renvoie le numéro qui représente l'image actuelle de la tortue.

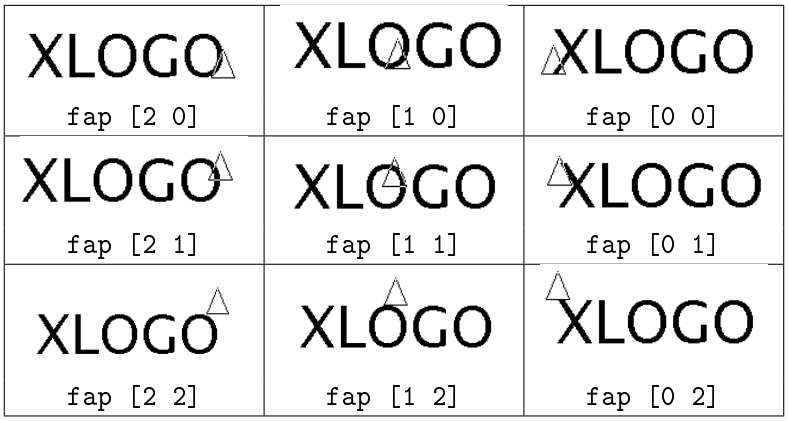
fixetaillepolice, ftp *entier*
Lorsqu'on écrit du texte sur l'écran à l'aide de la primitive `etiquette`, il est possible de modifier la taille de la police utilisée à l'aide de cette primitive. Par défaut, la taille de la police est réglée à 12.

taillepolice, tp
Renvoie la taille de la police actuellement utilisée lorsqu'on écrit avec la primitive `etiquette`.

fixealignementpolice, fap *liste*
Lorsqu'on écrit du texte sur l'écran à l'aide de la primitive `etiquette`, il est possible de spécifier la façon dont le texte est centré par rapport à la tortue. La liste est composée de deux nombres.

- Le premier représente l'alignement horizontal.
  - 0 : alignement horizontal à gauche.
  - 1 : alignement horizontal centré.
  - 2 : alignement horizontal à droite.
- Le deuxième représente l'alignement vertical.
  - 0 : alignement vertical sur le bas.
  - 1 : alignement vertical centré.
  - 2 : alignement vertical sur le haut.

Voici les différents cas possibles : `fixetaillepolice 50 etiquette "XLogo`



alignementpolice ap
Retourne la liste représentant le mode d'alignement du texte lorsqu'on écrit avec la primitive `etiquette`

fixenompolice, fnp *entier*
Fixe la police utilisée pour écrire à l'écran à l'aide de la primitive `etiquette`. Le numéro identifiant la police à utiliser est repérable dans Menu→Options→Préférences→Onglet Police.

nompolice, np
Renvoie une liste composée de deux éléments. Le premier est le numéro correspondant à la police utilisée pour écrire à l'aide de la primitive `etiquette`. Le second est une liste contenant le nom de cette même police.

fixeseparation, fsep *nombre*
Détermine le ratio entre la fenêtre graphique et la zone d'historique. Le nombre « `a` » doit être compris entre 0 et 1. Lorsqu'il vaut 1 la zone de dessin occupe toute la place, lorsqu'il vaut 0, la zone d'historique occupe toute la fenêtre etc

separation,sep
Renvoie le ratio actuel entre la zone de dessin et la zone d'historique.

grille *a b*
a et b sont des entiers. Trace une grille dont chaque carreau vaut a sur b.

stopgrille
Efface la grille.

fcg, fixecouleurgrille *couleur*
Permet de choisir la couleur de la grille. Ex : `fcg rouge`

couleurgrille
Retourne la couleur actuelle de la grille.

grille ?
Teste si la grille est tracée. Rend vrai ou faux selon les cas.

axes *n*
Trace les deux axes. Les graduations sont espacées de n pas de tortues.

axex *n*
Trace l'axe horizontal. Les graduations sont espacées de n pas de tortues.

axey *n*
Trace l'axe vertical. Les graduations sont espacées de 30 pas de tortues.

stopaxes
Efface les axes.

fca fixecouleuraxes *couleur*
Permet de choisir la couleur des axes. Ex : `fca [120 5 100]`

couleuraxes
Retourne la couleur actuelle des axes.

axex ?
Teste si l'axe horizontal est tracé. Rend vrai ou faux selon les cas.

axey ?
Teste si l'axe vertical est tracé. Rend vrai ou faux selon les cas.

fixezoom *n*
Effectue un zoom sur la zone de dessin. En fait le facteur `a` représente l'échelle par rapport à la taille de l'image fixée dans le panneau de préférence.

zoom
Retourne le facteur de zoom de la zone de dessin.

taillefenetre, tf
Renvoie une liste formée des coordonnées du coin supérieur gauche de la zone de dessin et du coin inférieur droit.

message, msg *liste*
Affiche un message d'information dans une boîte de dialogue, l'exécution du programme est stoppé en attente d'un click sur OK.

longueuretiquette, le *mot-liste*
Renvoie la longueur nécessaire pour écrire le mot ou la liste désirée sur la zone de dessin en utilisant la police sélectionnée. Cette longueur est exprimée en pas de tortue.

A.1.3 Un petit mot sur les couleurs

Les couleurs sont définies dans XLogo à l'aide de trois nombres compris entre 0 et 255. Ce système de codage s'appelle le codage « RGB »(Red, Green, Blue). Chaque nombre correspond respectivement à l'intensité du rouge, du vert et du bleu dans la couleur considérée. Etant donné que ce codage n'est pas très intuitif, XLogo vous propose également 16 couleurs prédéfinies accessibles soit par un numéro soit par une primitive.

Numéro	Primitives	[R G B]	Couleur
0	`noir`	[0 0 0]	
1	`rouge`	[255 0 0]	
2	`vert`	[0 255 0]	
3	`jaune`	[255 255 0]	
4	`bleu`	[0 0 255]	
5	`magenta`	[255 0 255]	

6	`cyan`	[0 255 255]	
7	`blanc`	[255 255 255]	
8	`gris`	[128 128 128]	
9	`grisclair`	[192 192 192]	
10	`rougefonce`	[128 0 0]	
11	`vertfonce`	[0 128 0]	
12	`bleufonce`	[0 0 128]	
13	`orange`	[255 128 0]	
14	`rose`	[255 175 175]	
15	`violet`	[128 0 255]	
16	`marron`	[153 102 0]	

```
# Ces trois commandes ont le même effet.
fcc orange
fcc 13
fcc [255 200 0]
```

A.1.4 Le mode animation

Il existe trois primitives : `animation`, `stopanimation` et `rafraichis` qui permettent d'exécuter des commandes sans que la tortue ne les affiche.

animation

On passe en mode animation. La tortue ne dessine plus à l'écran mais effectue le tracé en mémoire. Pour actualiser le dessin à l'écran, utiliser la primitive `rafraichis`. Très utile pour créer une animation ou effectuer un tracé plus rapidement.

stopanimation

Ceci termine le mode animation : On repasse en mode classique. On voit les déplacements de la tortue à l'écran.

rafraichis

En mode animation, rafraichit l'écran : l'image sur la zone de dessin est actualisée.
Pour indiquer le mode animation, une icone représentant une caméra apparait dans la zone d'historique. Si vous cliquez sur la caméra, cela stoppera l'animation, c'est à dire que ceci est équivalent à utiliser la primitive `stopanimation`.

A.1.5 Affichage du texte dans la zone d'historique

Ce tableau regroupe les primitives associées à la zone de texte d'historique. Toutes les primitives concernant la taille et la couleur de la police utilisée ne sont valables que pour le rendu de la primitive `ecris`.

vt, videtexte
Efface la zone contenant l'historique des commandes et des commentaires.

ec, ecris *arg1*
Affiche l'argument *arg1* dans la zone d'historique.

```
ecris "abcd --------> abcd
ec [1 2 3 4] ----> 1 2 3 4
ec 4 ------------> 4
```

tape *arg1*
Identique à la primitive `ecris` mais ne retourne pas à la ligne.

fixetaillepolicetexte, ftpt *n*
Définis la taille de la police dans la zone d'historique. Valable uniquement pour la primitive `ecris`

taillepolicetexte, tpt
Renvoie la taille de la police associée à la primitive `ecris`.

fixecouleurtexte, fct *couleur*
Définis la couleur de la police dans la zone d'historique. Valable uniquement pour la primitive `ecris`. Voir p.93.

couleurtexte, ctexte
Renvoie la couleur de la police associée à la primitive `ecris` dans la zone d'historique.

fixenompolicetexte, fnpt *n*
Fixe la police utilisée pour écrire dans l'historique à l'aide de la primitive `ecris`. Le numéro de la police est repérable dans Menu→Options→Préférences→Onglet Police.

nompolicetexte, npt
Renvoie une liste composée de deux éléments. Le premier élément est le numéro représentant la police utilisée pour écrire à l'écran à l'aide de la primitive `ecris`. Le second est une liste contenant le nom de cette même police.

fixestyle, fsty *arg1*
Fixe le style du rendu de la police utilisée par la primitive `ecris`. Les différents styles possibles sont `aucun, gras, italique, barre, indice, exposant, souligne`. Si vous souhaitez en utiliser plusieurs à la fois, les indiquer dans une liste.
Quelques exemples pour le formatage du texte avec la primitive `ecris` :

```
fixestyle [gras souligne] ecris "bonjour
bonjour
fsty "barre tape [texte rayé] fsty "italique tape "\ x fsty "exposant ecris 2
```

~~texte rayé~~ x^2

sty, style

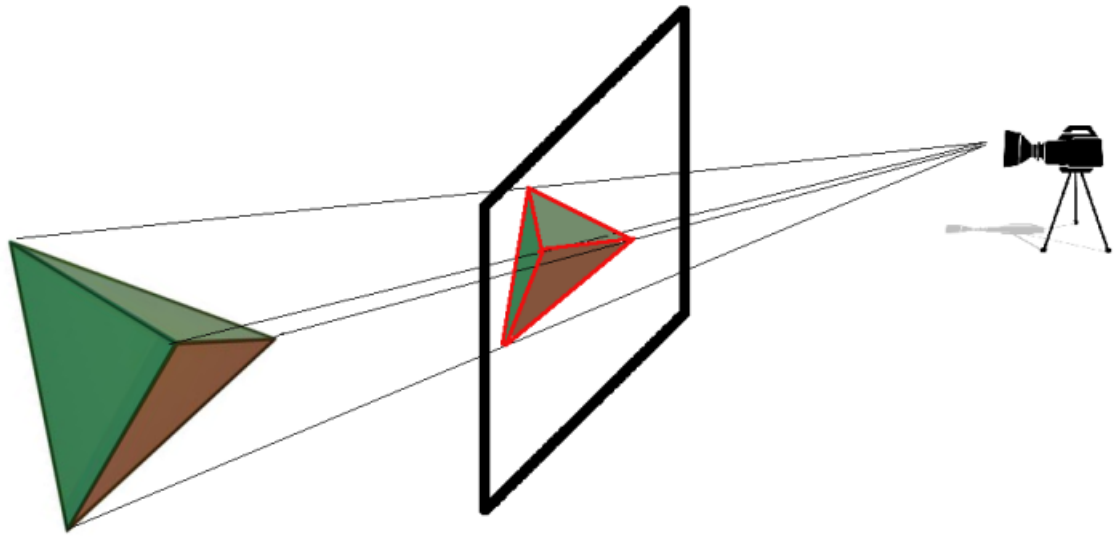
Renvoie une liste composée des différents styles actuellement utilisés pour le rendu de la primitive `ecris`.

A.2 La tortue dans l'espace

A partir de la version 0.9.92, la tortue peut s'échapper du plan pour se déplacer dans l'espace. Pour cela, on utilise la primitive `perspective`. Bienvenue dans le monde de la perspective 3D !

A.2.1 La technique de perspective

Pour représenter l'espace en trois dimensions dans un plan à deux dimensions uniquement, on utilise une perspective de projection. Une caméra observe la scène 3D et sa vision est projetée sur un plan intermédiaire. Voici un schéma illustrant cette technique.



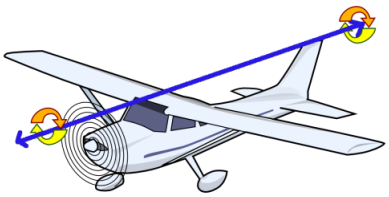
Certaines primitives vous permettent de positionner la caméra à votre guise, l'écran de projection étant situé à une distance égale à la moitié de la distance caméra-origine du repère.

A.2.2 Comprendre les déplacements dans l'espace

Dans le plan, la direction de la tortue était définie par son cap uniquement. Dans l'espace, l'orientation de la tortue est donnée par 3 valeurs d'angles :

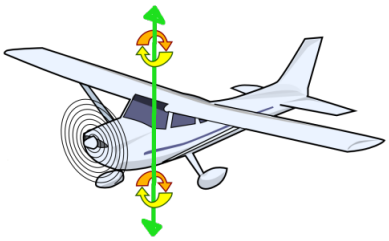
- Le roulis : Inclinaison de la tortue suivant l'axe (Oy)
- Le tangage : Inclinaison de la tortue suivant l'axe (Ox)
- Le cap : Inclinaison de la tortue suivant l'axe (Oz)

En fait, pour se déplacer dans l'espace la tortue, se comporte exactement comme un avion. Voici un petit schéma permettant de se représenter ces trois grandeurs :



Le roulis

Le tangage



Le cap

Cela peut paraitre compliqué de prime abord mais vous allez voir que beaucoup de choses se ramènent aux déplacements usuels du plan. Voici les primitives élémentaires de déplacement dans l'espace :

av, avance, re, recule n
Même comportement que dans le plan.

td, tournedroite, tg, tournegauche n
Meme comportement que dans le plan.

rd, roulisdroite n
La tortue pivote sur la droite suivant son axe longitudinal de n degrés.

rg, roulisgauche n
La tortue pivote sur la gauche suivant son axe longitudinal de n degrés.

cabre n
La tortue pivote vers le haut suivant son axe transversal de n degrés.

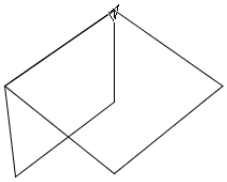
pique n
La tortue pivote vers le bas suivant son axe transversal de n degrés.

Dans le plan pour tracer un carré de côté 200 :

```
repete 4[av 200 td 90]
```

Ces instructions restent valables dans l'espace, et le carré est tracé en perspective. Si l'on fait « piquer » la tortue vers le bas de 90 degrés on peut tracer alors un nouveau carré.

```
ve
repete 4[av 200 td 90]
pique 90
repete 4[av 200 td 90]
```



Reste à s'entraîner pour appréhender toutes les orientations possibles !
Il faut toutefois bien comprendre que les trois primitives de rotation sont liées entre elles. Par exemple, testez la séquence suivante :

```
ve
roulisgauche 90 cabre 90 roulisdroite 90
```

Le déplacement effectué revient à avoir effectuer `tournegauche 90` (Tester en simulant la tortue avec votre main par exemple...)

A.2.3 Liste des autres primitives

L'ensemble des primitives suivantes est valable dans l'espace comme dans le plan. La seule différence est la nature des arguments attendus ou bien la nature des réponses. Par exemple, la primitive `fpos` ou `fixeposition` attend toujours une liste comme argument, mais maintenant, il faut que cette liste contiennent trois nombres $(x; y; z)$ représentant les coordonnées spatiaux du point désiré. Voici un récapitulatif de ces commandes :

Primitives compatibles dans le plan et dans l'espace

`cercle`	`arc`	`origine`	`vers`
`distance`	`fpos, fixeposition`	`fixex`	`fixey`
`fixecap`	`etiquette`	`longueuretiquette`	`point`
`pos, position`	`cap`		

Primitives valables uniquement en mode 3D

fixexyz *x y z*
Cette primitive déplace la tortue au point de coordonnées indiqués. Elle attend donc trois arguments, cette primitive est similaire à `fpos` mis à part que les coordonnées ne sont pas notés dans une liste.
Exemple, `fixexyz -100 200 50` : déplace la tortue au point de coordonnées $x = -100; y = 200; z = 50$

fixez *z*
Cette primitive déplace la tortue au point dont la cote z est égale à l'argument indiqué. Elle attend donc un nombre comme argument, cette primitive est comparable à `fixex` ou `fixey`.

fixeorientation *liste*
Positionne la tortue suivant l'inclinaison souhaitée. Cette primitive attend une liste contenant trois nombres, respectivement le roulis, le tangage et le cap.
Exemple, `fixeorientation [100 0 58]` : la tortue prend pour roulis 100 degrés, pour tangage 0 degré et pour cap 58 degrés,.

orientation
Retourne l'orientation de la tortue sous forme d'une liste contenant respectivement le roulis, le tangage et le cap. Attention à l'ordre de ces nombre, si par exemple, l'orientation est `[100 20 90]`, ceci signifie que pour obtenir la même orientation à partir de la position initiale (suite à avoir vider l'écran par exemple), il faudra taper :

```
roulisdroite 100 cabre 20 tournedroite 90
```

Si vous permutez l'ordre de ces instructions, vous n'obtiendrez pas l'orientation désirée !

fixeroulis *n*
Fait pivoter la tortue suivant son axe longitudinal de telle sorte qu'elle adopte l'angle de roulis indiqué.

roulis
Retourne la valeur actuelle de l'angle de roulis.

fixetangage *n*
Fait pivoter la tortue suivant son axe transversal de telle sorte qu'elle adopte l'angle de tangage indiqué.

tangage
Retourne la valeur actuelle de l'angle de tangage.

A.2.4 Le modeleur 3D

XLogoest également muni d'un modeleur 3D qui vous permet d'afficher votre tracé en 3 dimensions. Ce module utilise la bibliothèque JAVA3D qu'il est donc nécessaire d'installer si vous voulez profiter de cette fonctionnalité.

Voici les consignes d'utilisation du modeleur :
Au fur et à mesure de vos tracés sur la zone de dessin, il faut indiquer au modeleur les formes géométriques qu'il conservera pour un futur affichage. Il est possible d'enregistrer des polygones (surfaces), des lignes, des points ou encore du texte. Pour cela, on dispose des primitives suivantes :

polydef
Tous les prochains déplacements seront enregistrés en vue de créer un polygone.

polyfin
L'ensemble des sommets par lesquels est passé la tortue depuis l'appel de `polydef` matérialise un polygone dont la couleur est déterminée par l'ensemble des sommets. Cette primitive finalise la création du polygone.

lignedef
Tous les prochains déplacements seront enregistrés afin de créer une succession de segments.

lignefin
L'ensemble des sommets par lesquels est passé la tortue depuis l'appel de `lignedef` matérialise une ligne brisée dont l'écriture est ainsi finalisée.

pointdef
Tous les prochains déplacements seront enregistrés afin de créer un ensemble de points.

pointfin
L'ensemble des sommets par lesquels est passé la tortue depuis l'appel de `pointdef` sont enregistrés.

textedef
A chaque fois que l'utilisateur affichera un texte à l'aide de la primitive `etiquette`, celui-ci sera enregistré pour être ensuite confier au modeleur 3D.

textefin
Fin de l'enregistrement des textes affiché à l'écran.

vue3d polyaf
Lancement du modeleur 3D, tous les objets préalablement enregistrés sont affichés à l'écran.

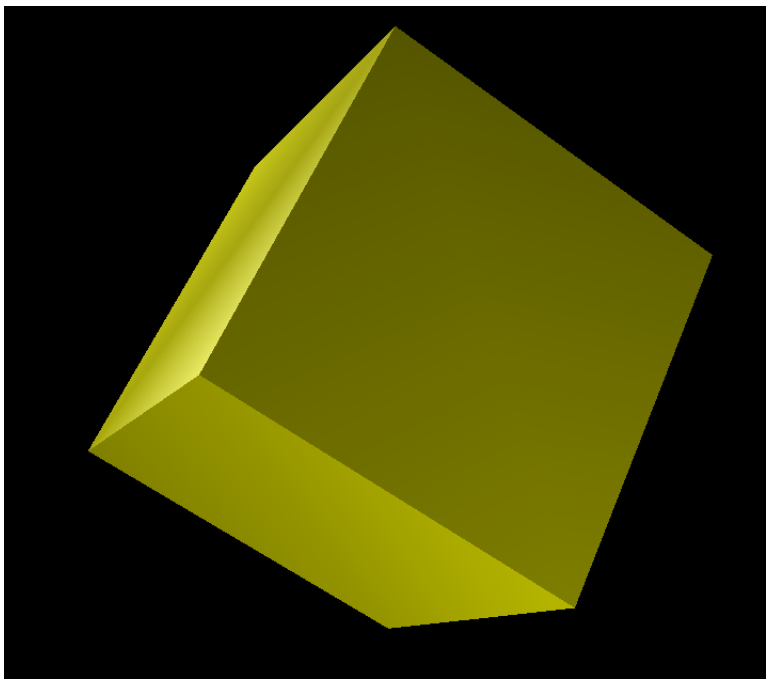
A.2.5 Création d'un cube

Chaque face est un carré de 400 pas de tortue de côté. Voici le programme

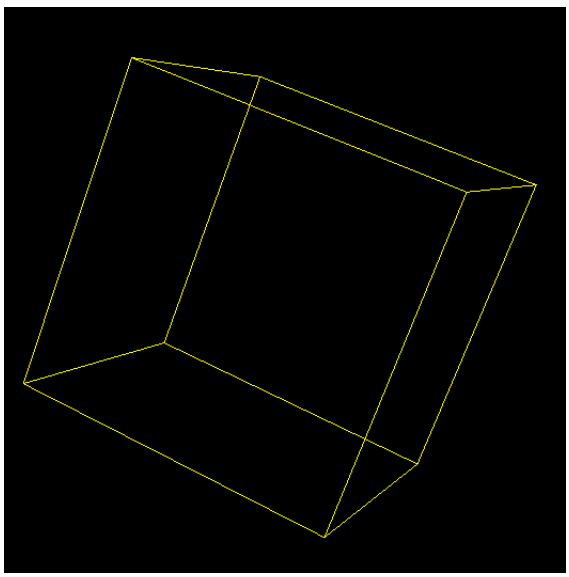
```
pour carre
# Les sommets du carre sont enregistres
polydef repete 4[av 400 td 90] polyfin
```

```
fin

pour cubeSimple
# Cube jaune
ve perspective fcc jaune
# faces laterales
repete 4[carre lc td 90 av 400 tg 90 rd 90 bc]
# face du dessous
pique 90 carre cabre 90
# face du dessus
av 400 pique 90 carre
# Visualisation
vue3d
fin
```

On lance la commande `cubeSimple` :



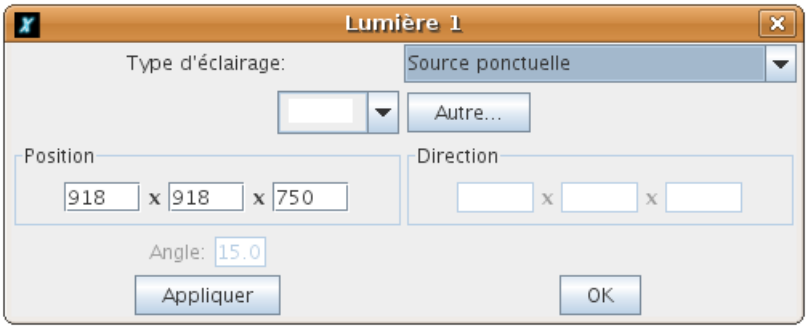
Puis en remplaçant dans la procédure `carre`, `polydef` par `lignedef` et `polyfin` par `lignefin`



Si on utilisait `pointdef` et `pointfin` au lieu de`lignedef` et `lignefin`, on aurait alors à l'écran uniquement les 8 sommets du cube. Ces deux primitives peuvent être tout particulièrement utilisées pour visualiser des nuages de points dans l'espace.

A.2.6 Gérer les lumières

Vous avez la possibilité pour éclairer vos scènes 3D d'utiliser quatre lumières. Par défaut, la scène est éclairée par deux lumières de type ponctuelle. Cliquer sur l'une des 4 ampoules dans le modeleur 3D, la boîte de dialogue ci-dessous apparaît alors :



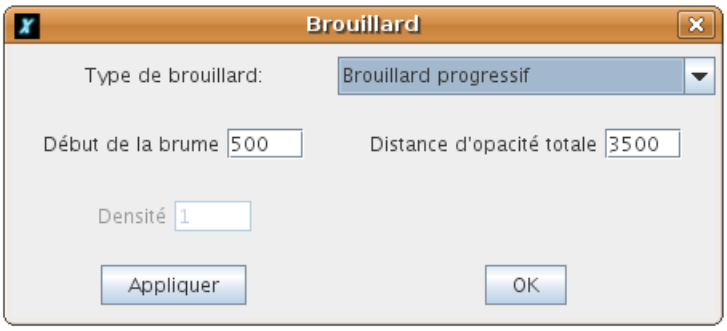
Plusieurs choix de lumières possible :

- Lumière ambiante : lumière uniforme, il est juste nécessaire de spécifier sa couleur
- Lumière unidirectionnelle : lumière éclairant suivant une direction constante. Cela correspond au cas d'une source ponctuelle située très loin, par exemple, le cas du soleil.
- Lumière ponctuelle : lumière dont on connait la position, on pourrait comparer cette lumière à un phare.
- Lumière de type « Spot » : c'est une lumière ponctuelle pour laquelle on peut restreindre l'éclairage au sein d'un cône de lumière dont on doit spécifier l'angle.

Le meilleur est tout simplement de les essayer afin de comprendre leur fonctionnement respectif !

Effet de brouillard

Vous avez la possibilité de rajouter un effet de brume sur votre scène 3D. Cliquer sur le bouton en forme de nuage dans le modeleur 3D, la boîte de dialogue ci-dessous apparaît.

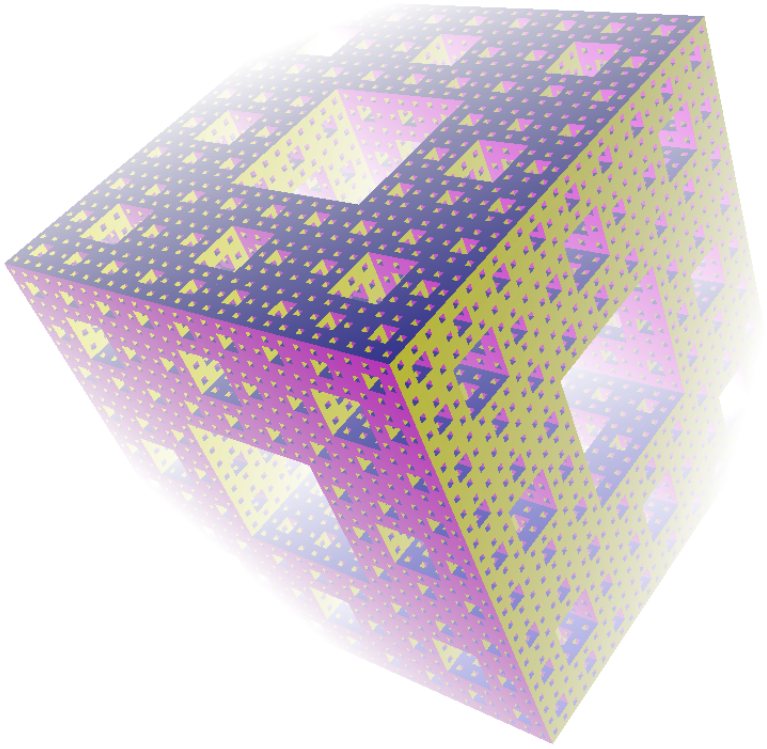


Deux types de brouillard au choix :

- Brouillard progressif : brouillard dont l'opacité devient de plus en plus importante. Vous devez spécifier deux paramètres :
  - La distance où le brouillard commence.
  - La distance où l'opacité du brouillard est totale

- Brouillard uniforme : brouillard s'appliquant de manière uniforme à l'ensemble de la scène. Vous devez juste spécifier ici le niveau de densité du brouillard.

Exemple avec un brouillard de type progressif :



A.3 Opérations arithmétiques et logiques

somme x y

Additionne les deux nombres x et y puis retourne le résultat.
Exemple : `somme` 40 60 retourne 100

difference x y

Retourne la soustraction $x - y$.
Exemple : `difference` 100 20 retourne 80

moins x

Retourne l'opposé de x.
Exemple : `moins` 5 retourne -5 .Voir la remarque à la suite de ce tableau

produit x y

Retourne le produit de x par y.

div, divise x y

Retourne le quotient de x par y
`divise 15 6` retourne 2.5

quotient x y

Retourne le quotient entier de x par y
`quotient 15 6` retourne 2

reste x y
Retourne le reste de la division de x par y.

modulo mod x y
Retourne x modulo y.

x	y	`reste x y`	`modulo x y`
14	5	4	4
−14	5	−4	1
14	−5	4	−1
−14	−5	−4	−4

Ce tableau montre la différence entre `modulo x y` et `reste x y`.

arrondi x
Retourne l'entier le plus proche du nombre x.
`arrondi 6.4` renvoie 6

tronque x
tronque à l'unité le nombre x.
`tronque 6.8` renvoie 6

puissance x n
Renvoie x élevé à la puissance n.
`puissance 3 2` renvoie 9

racine, rac n
Renvoie la racine carrée de n

log x
Renvoie le logarithme de x.

exp x
Renvoie l'exponentielle de x.

log10 x
Renvoie le logarithme décimal de x.

sinus, sin x
Renvoie le sinus du nombre x. (x est exprimé en degré)

cosinus, cos x
Renvoie le cosinus du nombre x. (x est exprimé en degré)

tangente, tan x
Renvoie le tangente du nombre x. (x est exprimé en degré)

arccosinus, acos x
Renvoie l'angle dont le cosinus vaut x. (l'angle est exprimé en degré)

arcsinus, asin *x*
Renvoie l'angle dont le sinus vaut x. (l'angle est exprimé en degré)

arctangente, atan *x*
Renvoie l'angle dont la tangente vaut x. (l'angle est exprimé en degré)

pi
Renvoie le nombre π (3.141592653589793)

hasard *n*
Renvoie un entier aléatoire compris entre 0 et n-1.

alea
Renvoie un nombre aléatoire compris entre 0 et 1.

absolue abs *x*
Renvoie la valeur absolue (distance à zéro) du nombre proposé.

fixedecimales *n*
Permet de fixer le nombre de décimales souhaités lors des calculs.
Cette primitive règle en fait le degré de précision des calculs. Quelques précisions :

- Par défaut, les calculs se font avec 16 décimales.
- Si n est négatif, le mode d'affichage par défaut est choisi.
- Si n est nul, les nombres affichés sont arrondis à l'unité.

Cette primitive est très utile pour effectuer des calculs nécessitant de garder beaucoup de décimales. Voir l'exemple avec le nombre π p.43.

decimales
Renvoie le nombre de décimales utilisé pour les calculs. Par défaut, cette valeur est fixée à -1.
Remarque : Attention aux primitives nécessitant deux paramètres !

Ex : `fixexy` a b Si b est négatif
Par exemple, `fixexy` 200 -10
L'interpréteur Logo va effectuer l'opération 200-10. Il va donc considérer qu'il n'y a qu'un paramètre (190) alors qu'il lui en faut deux d'où un message d'erreur. Pour éviter ce type de problème, utiliser la primitive « `moins` »indiquant l'opposé. `fixexy 200 moins 10`et là, il n'y a plus de problèmes !

Une autre possibilité peut être d'utiliser les parenthèses en tapant : `fixexy 200 (-10)`

Voici la liste des opérateurs logiques :

ou *b1 b2*
Renvoie vrai si b1 ou b2 est vrai, sinon renvoie faux

et *b1 b2*
Renvoie vrai si b1 et b2 sont égaux à vrai sinon renvoie faux

non *b*

Renvoie la négation du booléen b.

- Si b est vrai, renvoie faux.
- Si a est faux, renvoie vrai.

A.4 Opérations sur les listes et les mots

mot *mot1 mot2*

Concatène les deux mots mot1 et mot2.
Exemple : `ec mot "a 1` renvoie `a1`

liste *arg1 arg2*

Retourne une liste composée de *arg1* et *arg2*. Par exemple :
`liste 3 6` renvoie [3 6].
`liste` “une “liste renvoie [une liste]

phrase, ph *arg1 arg2*

Retourne une liste composée de *arg1* et *arg2*. Si *arg1* ou *arg2* est une liste, alors chacun des composants de *arg1* ou *arg2* devient élément de la liste créée (les crochets sont supprimés).
Ex : `ph [4 3] “bonjour` renvoie [4 3 bonjour]
`ph` [comment ça] “va renvoie [comment ça va]

metspremier, mp *arg1 liste2*

Insère *arg1* en première position de la liste.
Ex : `mp` “coucou [2] renvoie [coucou 2]

metsdernier, md *arg1 liste2*

Insère *arg1* en dernière position de la liste
Ex : `md` 5 [7 9 5] renvoie [7 9 5 5]

inverse *liste*

Inverse l'ordre des éléments de la liste.
`inverse` [1 2 3] renvoie [3 2 1]

choix *arg1*

- Si *arg1* est un mot, renvoie une des lettres de *arg1* prise au hasard.
- Si *arg1* est une liste, renvoie un des éléments de *arg1* pris au hasard.

enleve *arg1 liste*

Enlève l'élément *arg1* de la liste s'il apparaît dedans.
Ex : `enleve` 2 [1 2 3 4 2 6] renvoie [1 3 4 6]

item *n arg2*

- Si *arg2* est un mot renvoie la lettre *n* du mot (1 désigne la première lettre).
- Si *arg2* est une liste renvoie l'élément numéro *n* de la liste.

saufdernier, sd *arg*

- Si *arg* est une liste, renvoie toute la liste sauf le dernier élément.
- Si *arg* est un mot, renvoie le mot sans sa dernière lettre.

saufpremier, sp *arg*

- Si *arg* est une liste, renvoie toute la liste sauf le premier élément.
- Si *arg* est un mot, renvoie le mot sans sa première lettre.

dernier, der *arg*

- Si *arg* est une liste, renvoie le dernier élément de la liste.
- Si *arg* est un mot, renvoie la dernière lettre du mot.

premier, prem *arg*

- Si *arg* est une liste, renvoie le premier élément de la liste.
- Si *arg* est un mot, renvoie la première lettre du mot.

remplace *liste1 n arg*

Dans `liste1`, remplace l'élément numéro `n` par le mot ou la liste proposé.
`remplace [a b c] 2 8 --> [a 8 c]`

ajoute *liste1 n arg*

Dans `liste1`, ajoute en position numéro `n` le mot ou la liste proposé.
`ajoute [a b c] 2 8 --> [a 8 b c]`

compte *arg*

- Si *arg* est une liste, renvoie le nombre d'éléments de *arg*.
- Si *arg* est un mot, renvoie le nombre de lettres de *arg*.

unicode *mot1*

Renvoie la valeur unicode du caractère « *mot1* ».
`ec unicode "A` renvoie 65

caractere,car *n*

Renvoie le caractère dont la valeur unicode est *n*.
`ec caractere 65` renvoie `"A`

A.5 Booléens

Un booléen est une primitive qui renvoie le mot “vrai ou le mot “faux. Ces primitives se terminent par un point d'interrogation.

vrai

Renvoie "vrai

faux
Renvoie "faux

mot ? *arg1*
Renvoie vrai si *arg* est un mot, faux sinon.

nombre ? *arg1*
Renvoie vrai si *arg1* est un nombre, faux sinon.

entier ? *arg1*
Renvoie vrai si *arg1* est un entier, faux sinon.

liste ? *arg1*
Renvoie vrai si *arg1* est une liste, faux sinon.

vide ? *arg1*
Renvoie vrai si *arg1* est une liste vide ou un mot vide, faux sinon.

egal ? *arg1 arg2*
Renvoie vrai si *arg1* et *arg2* sont égaux, faux sinon.

precede ? *mot1 mot2*
Renvoie vrai si *mot1* est avant *mot2* dans l'ordre alphabétique, faux sinon.

membre ? *arg1 arg2*

- Si *arg2* est une liste, précise si *arg1* est élément de *arg2*.
- Si *arg2* est un mot, précise si *arg1* est un caractère de *arg2*.

membre *arg1 arg2*

- Si *arg2* est une liste, recherche `arg1` dans cette liste, deux cas possibles :
  - Si *arg1* est dans *arg2*, renvoie la sous-liste générée à partir de la première apparition de *arg1* dans *arg2*.
  - Si *arg1* n'est pas dans *arg2*, renvoie le mot faux.
- Si *arg2* est un mot, recherche le caractère *arg1* dans *arg2*, deux cas possibles :
  - Si *arg1* est dans *arg2* renvoie la fin du mot à partir de *arg1*.
  - Sinon, renvoie le mot faux.

`membre` “o “coucou renvoie oucou
`membre` 3 [1 2 3 4] renvoie [3 4]

baissecrayon ?, bc ?
Renvoie le mot vrai si le crayon est baissé, faux sinon.

visible ?
Renvoie le mot vrai si la tortue est visible, faux sinon.

primitive ?, prim ? *mot1*
Renvoie vrai si le mot est une primitive de XLogo, faux sinon.

procedure ?, proc ? *mot1*
Renvoie vrai si le mot est une procédure définie par l'utilisateur, faux sinon.

var ? variable ? *mot1*
Teste si *mot1* est une variable. Rend vrai ou faux selon les cas.

A.6 Effectuer un test à l'aide de la primitive `si`

Comme dans tout langage de programmation, le LOGOoffre la possibilité de vérifier si une condition donnée est vraie ou fausse afin d'exécuter le bout de code associé.

La primitive `si` permet de réaliser ces tests.

si *expression_ test liste1 liste2*

- Si *expression_ test* est vraie alors les commandes situées dans *liste1* sont exécutées.
- Si *expression_ test* est fausse alors ce sont les commandes de *liste2* qui sont exécutées.

La deuxième liste d'intructions est optionnelle.

Exemples d'utilisation :

- `si 1+2>=3[ecris "vrai][ecris "faux]`
- `si (premier "XLOGO)="Y [av 100 td 90] [ec [ XLOGO commence par un X !] ]`
- `si (3*4)=6+6 [ec 12]`

Remarque : Lorsque le résultat de la première expression est `faux`, la primitive `si`, cherche une deuxième liste, c'est à dire une expression commençant par un crochet ouvrant. Dans certains cas très particuliers, elle ne peut réaliser cette condition et il faut alors utiliser la primitive `sisinon` . Par exemple :

```
# On affecte deux listes aux variables a et b
 donne "a [ecris vrai]
 donne "b [ecris faux]

# premier test avec la primitive si--> La deuxième liste ne peut être évaluée.
 si 1=2 :a :b
Que faire de [ecris faux]?

# Deuxième test avec la primitive sisinon --> Effet escompté.
 sisinon 1=2 :a :b
 faux
```

A.7 L'espace de travail

L'espace de travail est composé de l'ensemble des éléments définis par l'utilisateur. Ceci comprend :

- Les procédures
- Les variables
- Les listes de propriétés.

A.7.1 Les procédures

Présentation

Les procédures sont des sortes de « programmes ». A l'appel de leur nom, les instructions comprises dans le corps de la procédure sont exécutées. On définit une procédure à l'aide du mot-clé **pour**.

```
pour nom_de_la_procédure :v1 :v2 :v3 .... [:v4 ....] [:v5 ....]
Corps de la procédure
fin
```

- nom_de_la_procédure est le nom donnée à la procédure.
- :v1 :v2 :v3 représentent les variables utilisées au sein de cette procédure (variables locales).
- [:v4 ...], [:v5 ...] sont les variables optionnelles que l'on peut rajouter à la procédure. (cf explication plus loin)
- Corps de la procédure représente les instructions à exécuter à l'appel de cette procédure.

Ex :

```
pour carre :c
repete 4[av :c td 90]
fin
```

La procédure se nomme `carre` et possède un paramètre s'appelant c. `carre` 100 produira donc un carre de côté 100. (Voir les exemples de procédures à la fin du manuel.)

Il est possible depuis la version 0.7c de rajouter des commentaires dans le code en les précédant du signe #.

```
pour carre :c
#cette procédure permet de tracer un carré de côté donné :c.
repete 4[av :c td 90] # pratique, non?
fin
```

Variables optionnelles

Il est à présent possible dans XLogo d'utiliser une « surcharge »d'arguments. Considérons la procédure suivante :

```
pour poly :n [:l 10]
repete :n [av :l td 360/:n]
fin

# Ceci trace un polygone régulier dont les 20
# côtés mesurent 10 pas de tortue
poly 20
```

A l'intreprétation, la variable `:l` est remplacée par sa valeur par défaut, c'est à dire 10. Si l'on souhaite changer cette valeur, on doit appeler la procédure `poly` entre parenthèses pour signaler à l'interpréteur que l'on va utiliser des paramètres optionnels.

```
# Ceci trace un polygone régulier dont les 20
# côtés mesurent à présent 5 pas de tortue
 (poly 20 5)
# Ceci trace un carré dont les
# côtés mesurent 100 pas de tortue
(poly 4 100)
```

La primitive 'trace'

Il est possible pour suivre le déroulement d'un programme de lui faire afficher les procédures en cours d'exécution. Ce mode permet d'afficher également si les procédures rendent des arguments à l'aide de la primitive `retourne`.

trace
Active le mode trace

stoptrace
Désative le mode trace

Un petit exemple avec le factorielle (voir p. 42).

```
trace ecris fac 4
fac 4
  fac 3
    fac 2
      fac 1
      fac retourne 1
    fac retourne 2
  fac retourne 6
fac retourne 24
24
```

A.7.2 Les variables

Il existe deux sortes de variables :

- Les variables globales : elles sont toujours disponibles à n'importe quel endroit du programme.
- Les variables locales : elles ne sont accessibles que dans la procédure où elles ont été définies.

Dans cette version de Logo, les variables locales ne sont pas accessibles dans les sous-procédures. A la sortie de la procédure, les variables locales sont éliminées.

donne *mot1 arg2*

- Si la variable locale *mot1* existe, lui affecte la valeur *arg2*
- Sinon, crée la variable globale *mot1* en lui affectant la valeur *arg2*.

Exemple : `donne` "a 100 affecte 100 à la variable a

locale *arg1*

- Si *arg1* est un mot, crée la variable locale nommée *arg1*.
- Si *arg1* est une liste, se sert de tous les éléments de la liste pour créer des variables locales.

Pour lui donner une valeur, voir `soit`.

donnelocale, soit *mot1 arg2*
Crée une nouvelle variable locale *mot1* et lui affecte la valeur *arg2*.

def, definis *mot1 liste1*
Définis une nouvelle procédure nommée *mot1*.
`liste1` contient une série de listes :

- La première de ces listes contient les variables de la procédure ainsi que le variables optionnelles.
- Chacune des listes suivantes représente une ligne de la procédure.

```
def "polygone [ [nb longueur] [repete :nb[av :longueur td 360/:nb]]]
```

Ceci définit une procédure nommée `polygone` avec deux variables (`:nb` et `:longueur`). Elle permet de tracer un polygône régulier dont on peut choisir le nombre de côtés et la longueur de chacun des côtés.

texte *mot1*
Renvoie toutes les information sur la procédure appelée `mot1`. Elle retourne une liste contenant plusieurs listes.
- La première de ces listes contient les variables de la procédure `mot1`.
- Les listes suivantes représentent chacune des lignes de la procédure.

Cette primitive est bien sûr associée à la primtive `definis`.

chose *mot1*
Renvoie la valeur de la variable *mot1*.
`chose "a` et `:a` sont deux notations équivalentes.

procs, procedures
Renvoie une liste contenant l'ensemble des procédures actuellement définies.

vars, variables
Renvoie une liste contenant l'ensemble des variables actuellement définies.

listesproprietes
Renvoie une liste contenant l'ensemble des listes de propriétés actuellement définies.

contenu
Renvoie une liste formées de trois autres listes. La première contient toutes les procédures définies, la seconde toutes les variables et la dernière toutes les listes de propriétés.

primitives
Renvoie une liste contenant toutes les primitives connues.

effaceprocedure, efp *arg1*
Efface la procédure s'appelant *arg1*, ou toutes les procédures contenues dans la liste *arg1*.

effacevariable, efv *arg1*
Efface la variable *arg1* ou toutes les variables contenues dans la liste *arg1*.

effacelisteproprietes, eflp *arg1*
Efface la liste de propriétés nommée *arg1* ou toutes les listes de propriétés contenues dans la liste *arg1*.

effacetout
Efface toutes les variables, listes de propriétés et procédures définies dans l'esapce de travail.

bye, aurevoir
Quitte XLogo.

execute, exec *liste1*
Exécute la liste d'instruction contenue dans *liste1*.

commandeexterne *liste1*
Permet d'exécuter une commande système depuis XLogo. *liste1* doit contenir plusieurs listes contenant chacune les mots constituant la commande. Quelques exemples :

`commandeexterne [[gedit][/home/xlogo/fichier.txt]]`
Lance l'application `gedit` et charge le fichier `/home/xlogo/fichier.txt` (Linux)

`commandeexterne [[notepad][C: /fichier.txt]]`
Lance l'application `notepad` et charge le fichier nommé `C: /fichier.txt` (Windows)

Cette syntaxe un peu particulière permet notamment l'utilisation d'espaces dans les chemins de fichiers.

A.7.3 Les listes de propriétés

A partir de la version 0.9.92, XLogo supporte les listes de propriétés. Chaque liste a un nom principal et est composée d'un ensemble de paires Clé-Valeur.

Par exemple, considérons une liste de propriétés nommée « voiture ». Elle peut contenir par exemple la clé « couleur » associée à la valeur « rouge », ou encore la clé « type » associée à la valeur « décapotable ».

Pour manipuler ces listes, nous disposons des primitives suivantes :

dprop donnepropriete *nom clé valeur*
Considère la liste de propriétés *nom* (Si elle n'existe pas, la crée). Elle y ajoute l'élémet *valeur* qui sera accessible à l'aide du mot *clé*.

rprop rendspropriete *nom clé*
Récupère dans la liste de propriétés `nom` la valeur associée à la clé désirée. Si la liste de propriétés n'existe pas ou si la clé n'existe pas, retourne une liste vide.

efprop effacepropriete *nom clé*
Dans la liste de propriétés `nom`, efface la valeur associée à la clé choisie.

lprop listepropriete *nom*
Affiche l'ensemble des couples clé-valeur contenue dans la liste de propriétés `nom`.

Reprenons l'exemple de la liste « voiture ».

```
# Remplissage de la liste
 dprop "voiture "couleur "rouge
 dprop "voiture "type "décapotable
 dprop "voiture "marque "Citroën

# Affichage d'une valeur particulière
 ecris rprop "voiture "couleur
 rouge

# Affichage de tous les éléments
 ecris lprop "voiture
 couleur rouge type décapotable marque Citroën
```

A.8 Gestion des Fichiers

chargeimage, ci *mot1*

Affiche le fichier image *mot1*. Son coin supérieur gauche sera placé où se trouve la tortue . Les formats supportés sont le png et le jpg.
Le chemin spécifié doit être relatif par rapport au répertoire courant. Ex : `chargeimage "tortue.jpg]`

sauveimage *mot liste*

Sauve la zone de dessin dans le fichier *mot* du répertoire courant.
Les formats supportés sont le « jpg » et le « png ». Si aucun des deux n'est spécifié, le format « png » est choisi par défaut. Il est également possible de spécifier une zone de sélection à l'aide de la deuxième liste qui doit contenir quatre nombres $[X_{min}Y_{min}X_{max}Y_{max}]$ qui permettent de spécifier les coordonnées des deux coins du rectangle de sélection. Un petit exemple :

```
## Dans l'éditeur
pour test
repete 20 [
  avance 30 tournedroite 18
  # sauve les images sous le nom 1.png, 2.png ... 20.png
  sauveimage mot compteur ".png  [-50 150 200 -150]
]
fin

## Sur la ligne de commande:
test
videecran cachetortue repete 20 [chargeimage mot compteur ".png]
```

Et vous avez créé une petite animation !

catalogue, cat

Liste le contenu du répertoire par défaut. (Equivalent de la commande `ls` pour linux et `dir` pour DOS)

fixerepertoire, frep *mot1*

Fixe le répertoire en cours. Le chemin doit être absolu et doit être spécifié à l'aide d'un mot.

changedossier, cd *mot1*

Permet de choisir le répertoire courant. Le chemin est relatif par rapport au répertoire courant actuel. On peut utiliser la notation « `..` » pour faire référence au répertoire parent.

repertoire, rep

Rend le répertoire en cours. Par défaut, il est fixé au répertoire utilisateur c'est à dire `/home/votre_login` pour les linuxiens, `C :\WINDOWS` pour les autres.

sauve *mot1 liste2*

Un bon exemple pour expliquer cela : `sauve "essai.lgo [proc1 proc2 proc3]` sauve dans le fichier essai.lgo du répertoire courant les procédures proc1, proc2 et proc3. Si l'extension .lgo est omise, elle est rajoutée par défaut. Le mot spécifié désigne un chemin relatif par rapport au répertoire courant. Cette commande ne fonctionne pas avec un chemin absolu.

sauved *mot1*

`sauved "test.lgo` sauve dans le fichier test.lgo du répertoire courant l'ensemble des procédures définies

actuellement. Si l'extension .lgo est omise, elle est rajoutée par défaut. Le mot spécifié désigne un chemin relatif par rapport au répertoire courant. Cette commande ne fonctionne pas avec un chemin absolu.

ed, edite *arg1*
Ouvre dans l'éditeur l'ensemble des procédures dont le nom est spécifié dans la liste `arg1` ou le mot *arg1*.

edtout, editetout
Ouvre dans l'éditeur l'ensemble des procédures définies actuellement.

ramene *mot1*
Ouvre et interprete le fichier *mot1*. Par exemple, pour effacer toutes les procédures définies et charger le fichier essai.lgo, on écrira : `efp procs ramene "essai.lgo`. Le mot spécifié désigne un chemin relatif par rapport au répertoire courant. Cette commande ne fonctionne pas avec un chemin absolu.

ouvreflux *if fich*
Lorsque l'on veut lire ou écrire dans un fichier, il faut au préalable ouvrir un flux vers ce fichier. L'argument `fich` doit être le nom du fichier considéré. On doit utiliser un mot pour indiquer le le nom du fichier dans le répertoire courant. L'argument `id` est le numéro que l'on affecte à ce flux afin de pouvoir l'identifier.

listeflux
Affiche la liste des différents flux ouverts avec leur identifiant.

lisligneflux *id*
Ouvre le flux dont l'identifiant est le numéro *id* puis lis une ligne dans ce fichier.

liscarflux *id*
Ouvre le flux dont le numéro d'identifiant est celui passé en argument puis lis un caractère dans ce fichier. Cette primitive renvoie un nombre représentant la valeur du caractère (semblable à `liscar`).

ecrisligneflux *id liste2*
Ecris la ligne de texte contenue dans la liste au début du fichier repéré par l'identifiant *id*. Attention, l'écriture n'est effective que lorsque l'on ferme le flux avec la primitive `fermeflux`.

ajouteligneflux *id liste2*
Ecris la ligne de texte contenue dans la liste à la fin du fichier repéré par l'identifiant *id*. Attention, l'écriture n'est effective que lorsque l'on ferme le flux avec la primitive `fermeflux`.

fermeflux *id*
Ferme le flux dont le numéro d'identifiant est celui passé en argument.

finflux ? *id*
Renvoie `"vrai` si on est arrivé à la fin du fichier. Renvoie `"faux` sinon.

Voici un exemple d'utilisation des primitives permettant de lire et écrire dans un fichier. Nous présenterons cet exemle pour une architecture de type Windows. Les autres utilisateurs sauront adapter l'exemple suivant.

L'objectif est de créer le fichier `c :\exemple` contenant les trois lignes :

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789

```
# On ouvre un flux vers le fichier désiré. Ce flux sera repéré par le numéro 2
fixerepertoire "c:\\
ouvreflux 2 "exemple
# On écrit les lignes désirées
ecrisligneflux 2 [ABCDEFGHIJKLMNOPQRSTUVWXYZ]
ecrisligneflux 2 [abcdefghijklmnopqrstuvwxyz]
ecrisligneflux 2 [0123456789]
# On ferme le flux pour achever l'écriture
fermeflux 2
```

A présent, on peut constater que l'écriture s'est bien passée :

```
# On ouvre un flux vers le fichier à lire. Ce flux sera repéré par le numéro 0
ouvreflux 0 "c:\\exemple
# On lit les lignes du fichiers successivement
ec lisligneflux 0
ec lisligneflux 0
ec lisligneflux 0
# On ferme le flux
fermeflux 0
```

Si on souhaite à présent rajouter la ligne « Formidable ! » :

```
fixerepertoire "C:\\
ouvreflux 1 "exemple]
ajouteligneflux 1 [Formidable !]
fermeflux 1
```

A.9 Fonction avancée de remplissage :

Il existe trois primitives permettant de colorier une forme : La primitive `remplis`, la primitive `rempliszone` et la primitive `remplispolygone`.

A.9.1 `remplis` et `rempliszone`

On peut apparenter ces primitives avec la fonction "pot de peinture" utilisée dans de nombreux logiciels de retouche d'images. On peut atteindre les bords de la zone de dessin. Il y a deux règles à respecter pour utiliser correctement ces primitives :

1. Le crayon doit être en position baissé (`bc`).
2. La tortue ne doit pas être située sur un pixel de la couleur dont on veut remplir la forme. (Si on veut colorier en rouge, ne pas se trouver soi-même sur du rouge...)

Voyons un exemple pour expliquer la différence entre `remplis` et `rempliszone` : Le pixel sous la tortue est actuellement de couleur blanche. La primitive `remplis` va colorier tous les pixels blancs voisins avec la couleur du crayon en cours. Si par exemple on tape : `fcc 1 remplis`
Revenons à présent au premier cas, Si la couleur du crayon de la tortue est le noir, la primitive `rempliszone`, colorie tous les pixels voisins jusqu'à rencontrer la couleur en cours (ici noire). Voici, un bel exemple d'utilisation de la primtive `remplis` :

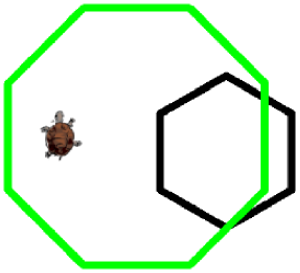


FIG. A.1 – Situation initiale

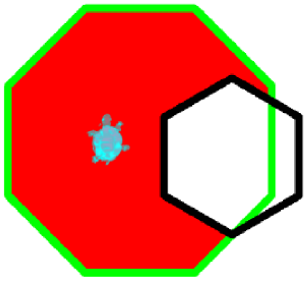


FIG. A.2 – Avec la primitive `remplis`

```
pour demice :c
# trace un demi-cercle de diamètre :c
repete 180 [av  :c*tan 0.5 td 1]
av :c*tan 0.5
td 90 av :c
fin

pour arcenciel :c
si :c<100 [stop]
demice :c td 180 av 20 tg 90
arcenciel :c-40
fin

pour dep
lc td 90 av 20 tg 90 bc
fin

pour demarrer
ct arcenciel 400 go tg 90 av 20 re 120 de lc td 90 av 20 bc
fcc 0 remplis dep
fcc 1 remplis dep
fcc 2 remplis dep
fcc 3 remplis dep
fcc 4 remplis dep
fcc 5 remplis dep
fcc 6 remplis dep
fin
```

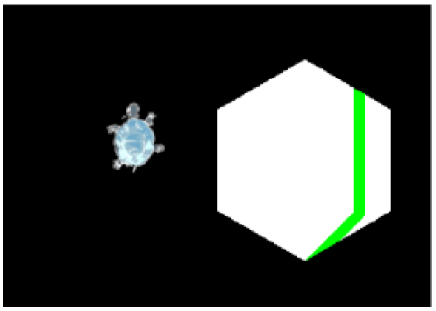


FIG. A.3 – Avec la primitive `rempliszone`, en tapant : `fcc 0 rempliszone`

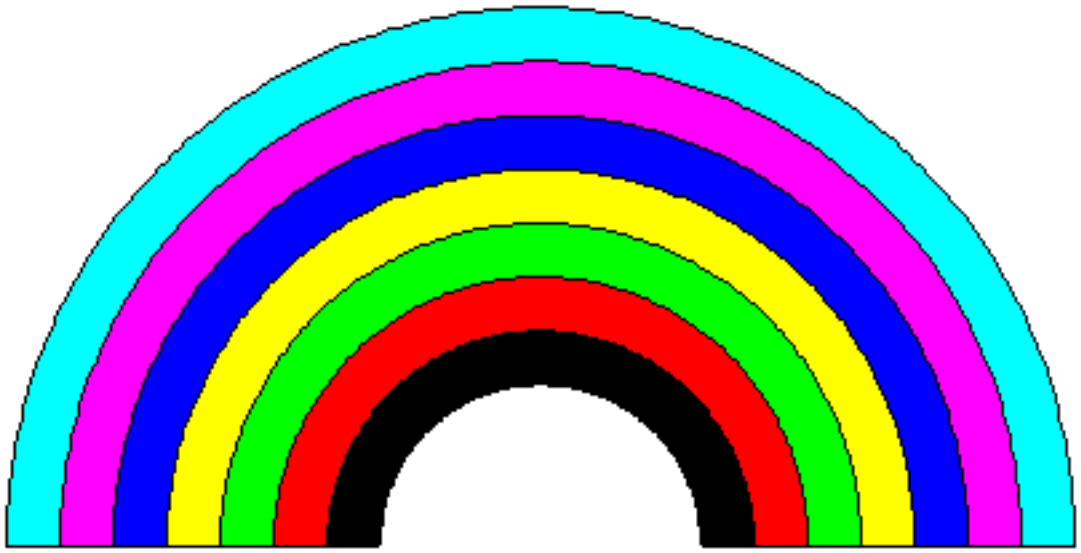


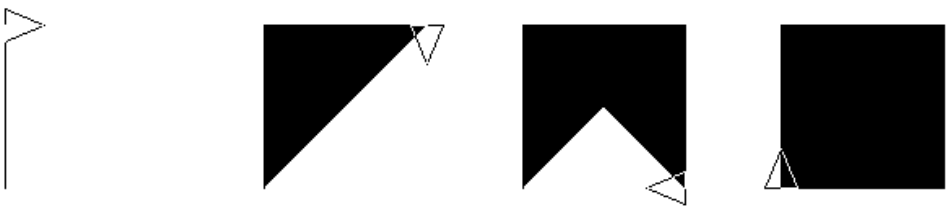
FIG. A.4 – Arc-en-LOGO

A.9.2 La primitive `remplispolygone`

remplispolygone *liste*

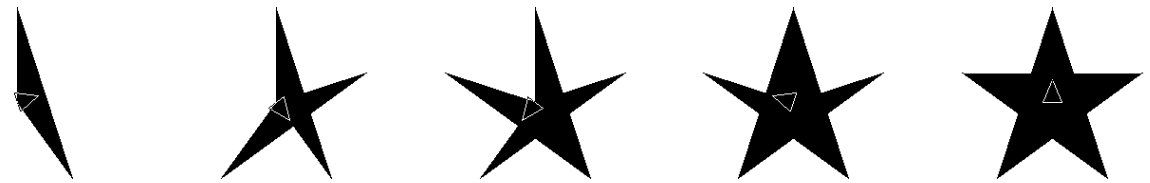
Cette primitive permet de remplir une forme en utilisant une série de triangles. A chaque nouvelle ligne tracée, un triangle est rempli. La liste passée en argument contient les instructions assurant le découpage en triangles de la forme désirée.

- Un premier exemple pour remplir un carré :
  `ve remplispolygone [repete 4 [avance 100 tournedroite 90]]`



Etape 1 Etape 2 Etape 3 Etape 4

- Un second exemple qui trace une étoile à 5 branches :
  `repete 5 [avance 100 remplispolygone [recule 100 td 144 avance 100 ] re 100 tg 72]`



Etape 1 Etape 2 Etape 3 Etape 4 Etape 5

A.10 Les instructions de saut

XLogo possède trois instructions de saut : `stop`, `stoptout` et `retourne`.

stop

`stop` peut avoir deux effets : s'il est inclus dans une boucle `repete` ou `tantque` : à ce moment là, on sort de la boucle. S'il est dans une procédure, on sort immédiatement de la procédure.

stoptout

`stoptout` interrompt définitivement l'exécution de toutes les procédures en cours.

retourne, ret *arg1*

`retourne` permet de sortir d'une procédure en retournant la valeur *arg1*.

A.11 Le mode multitortues

Il est possible de piloter à l'écran plusieurs tortues à la fois. Par défaut, lorsqu'on lance XLogo, une seule tortue est active. Elle est repérée par le numéro 0. Pour "créer" une nouvelle tortue à l'écran, on utilise la primitive `fixetortue` suivi du numéro de la tortue désirée. Pour évitement l'encombrement sur la fenêtre de dessin, celle-ci est créée à l'origine de la zone de dessin (coordonnées (0 ;0)) et est invisible, c'est à dire qu'il faudra se servir de la commande `mt` pour la faire apparaître. Ensuite cette nouvelle tortue obéit aux ordres classiques tant que l'on ne change pas de tortue avec `fixetortue`. Le nombre maximal de tortues disponibles se règles dans Options - Préférences - Onglet options.

Voici la liste des primitives concernant le mode multitortues :

fixetortue, ftortue n

Fixe le numéro de la tortue active. Par défaut, la première tortue active au lancement de XLogoa le numéro 0.

tortue

Renvoie le numéro de la tortue actuellement utilisée.

tortues

Renvoie une liste composée de tous les numéros des tortues actuellement utilisées.

effacetortue n

Elimine de l'écran la tortue numéro n.

fmt, fixemaxtortues n

Fixe le nombre mamximal de tortues sur l'écran en mode multitortues.

maxtortues
Renvoie le nombre mamximal de tortues sur l'écran en mode multitortues.

A.12 Jouer de la musique

A.12.1 Musique avec le synthériseur MIDI

sequence, seq *liste1*
Met en mémoire la séquence musicale située dans la liste. Pour apprendre à rédiger séquence musicale, voir les instructions après le tableau.

joue
Joue la séquence actuellement mise en mémoire.

instrument, instr
Renvoie le numéro correspondant à l'instrument actuellement sélectionné.

fixeinstrument, finstr *n*
Sélectionne comme instrument l'instrument numéro n. Vous pouvez voir la liste des instruments disponibles dans menu-options-préférence-onglet son.

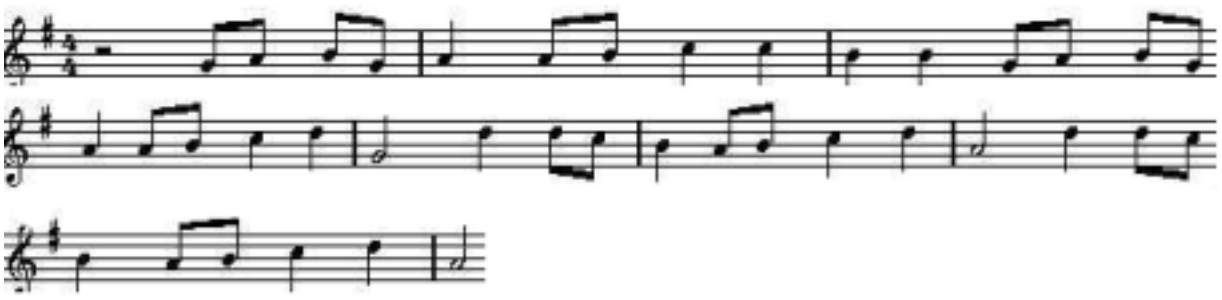
indexsequence, indseq
Renvoie à quel temps le curseur est situé dans la séquence en cours.

fixeindexsequence, findseq *n*
Déplace le curseur au temps numéro n dans la séquence musicale actuellement en mémooire.

effacesequence, efseq
Efface la séquence actuellement en mémoire.

Pour jouer de la musique, il faut mettre au préalable la composition désirée en mémoire dans ce que l'on appellera ici une séquence musicale. On crée la séquence à l'aide de la commande `seq` ou `sequence`. Voici les quelques règles à respecter pour écrire convenablement une séquence musicale :
`do re mi fa sol la si` : désignent les notes usuelles de la première octave.
Pour faire un ré dièse, on tapera `re +`
Pour faire un re bémol, on tapera `re -`
Si on veut changer d'octave, on utilise le symbole " :" suivit de + ou -. Par exemple, après avoir tapé :++, toutes les notes jouées seront augmentées de deux octaves (il y a deux "+").
Les notes sont par défaut jouées sur une durée de un temps. Si on veut changer la durée d'une série de notes, on l'indique par le nombre indiquant la durée désirée. Pour taper une série de croches($\frac{1}{2}$ temps), on tapera `seq [0.5 sol la si]`.
Un bon exemple valant mieux que mille explications :

```
pour tabac
# Met en mémoire la partition
sequence [0.5 sol la si sol 1 la 0.5 la si 1 :+ do do :- si si 0.5 sol la si sol
          1 la 0.5 la si 1 :+ do re 2 :- sol ]
sequence [:+ 1 re 0.5 re do 1 :- si 0.5 la si 1 :+ do re 2 :- la ]
sequence [:+ 1 re 0.5 re do 1 :- si 0.5 la si 1 :+ do re 2 :- la ]
```



```
sequence [0.5 sol la si sol 1 la 0.5 la si 1 :+ do do :- si si 0.5 sol la si sol
          1 la 0.5 la si 1 :+ do re 2 :- sol ]
fin
```

Pour lancer la musique, il ne nous reste plus qu'à taper : `tabac joue`
Voyons à présent une application intéressante de la primitive findseq. Taper les commandes suivantes :

```
efseq       # On efface la séquence actuellement en mémoire
tabac       # On recharge la musique précédente
findseq 2   # On replace le curseur au niveau du premier "la" noir de la 2nde mesure
tabac       # On recharge la même séquence mais décalée de deux temps.
joue        # Un magnifique canon !
```

Vous pouvez également changer d'instruments, soit à l'aide de la commande `finstr` soit dans le menu Options-Préférences-Onglet son. Vous trouverez la liste de tous les instruments disponibles avec leur numéro (C'est en anglais, mais ça permet de se donner une idée. Chez moi, 411 instruments disponibles !)

A.12.2 Jouer du MP3

jouemp3 *mot1*
Lit le fichier mp3 *mot1*. Ce fichier doit être situé dans le répertoire courant. Il est également possible d'indiquer un chemin réseau. Des exemples d'utilisation :
`jouemp3 fichier.mp3`
`jouemp3 http ://monsite.fr/fichier.mp3`

stopmp3
Interrompt la lecture du fichier mp3 en cours.

A.13 Boucles :

XLOGO dispose de sept primitives permettant d'effectuer des boucles : `repete`, `repetepour` et `tantque`, `pourchaque`, `repetetoujours`, `repetetantque` et `repetejusqua`.

A.13.1 Une boucle avec `repete`

repete *n liste_ d_ instruction*
n est un entier et `liste_d_instruction` est une liste contenant des instruction à exécuter. L'interpréteur LOGO va effectuer *n* fois les commandes contenues dans la liste : cela évite ainsi de recopier *n* fois la même instruction !
Ex :

```
repete 4 [avance 100 tournegauche 90]      # Un carré de côté 100
repete 6 [avance 100 tournegauche 60]      # Un hexagone de côté 100
repete 360 [avance 2 tournegauche 1]       # Un euh... 360-gone de côté 2
                                           # Bref, quasiment un cercle !
```

compteur

Au sein d'une boucle `repete`, est définie une variable interne `compteur`. Celle-ci désigne le numéro de l'itération en cours (la première itération étant numérotée 1).

```
repete 3 [ec compteur]
1
2
3
```

A.13.2 Une boucle avec `repetepour`

`repetepour` joue le rôle des boucles `for` dans d'autres langages de programmation.

repetepour *liste1 liste2*

Cette boucle consiste à affecter à une variable un certain nombre de valeurs comprises dans un intervalle donné suivant un incrément donné.
liste1 contient trois paramètres : le nom de la variable, la borne initiale, la borne finale.
On peut rajouter un quatrième argument optionnel désignant l'incrément (le pas avec lequel la variable défile). S'il est omis, il est par défaut de 1. Quelques exemples d'utilisation :

```
repetepour [i 1 4][ec :i*2]
2
4
6
8

# A présent on fait varier i entre 7 et 2 en descendant de 1.5 à chaque fois
# noter l'incrément négatif
# On affiche ensuite i son carré.

repetepour [i 7 2 -1.5 ][ec liste :i puissance :i 2]
7 49
5.5 30.25
4 16
2.5 6.25
```

A.13.3 Une boucle avec `tantque`

tantque *liste_ a_ tester liste_ d_ instruction*

liste_ a_ tester est une liste contenant une suite d'instruction rendant un booléen. *liste_ d_ instruction* est une liste contenant des instructions à exécuter. L'interpréteur LOGO exécutera continuellement *liste_ d_ instruction* tant que *liste_ a_ tester* rend vrai.
Ex :

```
tantque ["vrai] [td 1]                    # La tortue tourne sur elle-même
```

```
# Un exemple qui nous permet d'épeler l'alphabet à l'envers

donne "liste "abcdefghijklmnopqrstuvwxyz
tantque [non vide? :liste] [ec dernier :liste donne "liste saufdernier :liste]
```

A.13.4 Une boucle avec `pourchaque`

pourchaque *nom_variable liste_ou_mot commande*

Cette primitive permet de décrire tous les éléments d'une liste ou tous les caractères d'un mot puis exécute à chaque fois le contenu de la liste des commandes.

```
pourchaque "i "XLOGO [ecris :i]
X
L
O
G
O
pourchaque "i [a b c] [ecris :i]
a
b
c
```

A.13.5 Une boucle avec `repetetoujours, repetetjs`

repetetoujours,repetetjs *liste_instructions*

Répète indéfiniment une liste d'instructions.

```
repetetoujours [av 1 td 1]
```

Attention : cette primitive est à utiliser avec prudence du fait de la boucle infinie !

A.13.6 Une boucle avec `repetetantque`

repetetanque *liste1 liste2*

Répète une séquence d'instructions contenue dans *liste1* tant que *liste2* est vraie.
La principale différence avec la primitive `tantque` est qu'ici, la liste d'instructions est exécutée au moins une fois même si le test de sortie est faux.

```
donne "i 0
repetetantque [ec :i donne "i :i+1] [:i<4]
0
1
2
3
4
```

A.13.7 Une boucle avec `repetejusqua`

repetejusqua *liste1 liste2*

Répète une séquence d'instructions contenue dans *liste1* jusqu'à ce que *liste2* soit vraie.

```
donne "i 0
repetejusqua [ec :i donne "i :i+1] [:i>4]
0
1
2
3
4
```

A.14 Intercepter des actions de l'utilisateur

XLOGO peut interagir avec l'utilisateur pendant l'exécution d'un programme par l'intermédiaire du clavier et de la souris.

A.14.1 Interaction avec le clavier

On peut donc recevoir du texte de l'utilisateur pendant le programme à l'aide de 3 primitives : `touche ?`, `liscar` et `lis`.

touche ?
Rend vrai ou faux selon qu'une touche ait été pressée ou non depuis le début de l'exécution du programme.

liscar

- Si `touche ?` est faux, bloque le programme jusqu'à ce l'utilisateur appuie sur une touche.
- Si `touche ?` est vrai rend la valeur correspondant à la touche qui a été la dernière enfoncée.

A —> 65	B —> 66	C —> 67	etc ...	Z —> 90
← —> -37 ou -226 (NumPad)	↑ —> -38 ou -224	→ —> -39 ou -227	↓ —> -40 ou -225	
Echap —> 27	F1 —> -112	F2 —> -113	....	F12 —> -123
Shift —> -16	Espace —> 32	Ctrl —> -17	Enter —> 10	

TAB. A.2 – Quelques valeurs de touche

Si vous avez un doute par le mot retourné par une touche, il vous suffit de taper :
`ec liscar`. L'interpréteur va alors attendre que vous tapiez sur une touche puis vous donnera la valeur correspondante.

lis *liste1 mot2*
Affiche une boîte de dialogue dont le titre est *liste1*. L'utilisateur peut alors rentrer une réponse dans un champ de texte, la réponse sera stockée sous forme d'un mot ou d'une liste (si l'utilisateur tape plusieurs mots) dans la variable *mot2* lorqu'il validera ou cliquera sur le bouton OK.

A.14.2 Quelques exemples d'utilisation :

```
pour yeuv
lis [Quel est ton age?] "age
donne "age :age
si :age<18 [ec [tu es mineur]]
si ou :age=18 :age>18 [ec [tu es majeur]]
si :age>99 [ec [Respect !!]]
fin
pour rallye
si touche? [
```

```
donne "car liscar
si :car=-37 [tg 90]
si :car=-39 [td 90]
si :car=-38 [av 10]
si :car=-40 [re 10]
si :car=27 [stop]
]
rallye
fin
# On contrôle la tortue avec le clavier, on arrête avec Esc
```

A.14.3 Intercepter certains événements provenant de la souris

Pour cela, on dispose de trois primitives : `lissouris`, `souris ?` et `possouris`.

lissouris

Bloque le programme jusqu'à ce qu'un événement souris se produise : on entend par événement souris le fait de déplacer la souris ou de cliquer sur l'un de ses boutons. Une fois l'événement produit, `lissouris` rend un nombre permettant de caractériser l'événement. Voici les différents codes associés aux différents événements qu'ils représentent :

- 0→on a déplacé la souris.
- 1→on a appuyé sur le bouton 1 de la souris.
- 2→on a appuyé sur le bouton 2 de la souris.

Les boutons sont numérotés de la gauche vers la droite (en principe...)

possouris

Renvoie une liste contenant les coordonnées de la souris lors du dernier événement intercepté.

souris ?

Rend vrai ou faux selon que l'on ait agi ou non sur la souris depuis le début de l'exécution du programme.

A.14.4 Quelques exemples d'utilisation

Dans cette première procédure, la tortue suit la souris lorsqu'elle se déplace sur la zone de dessin.

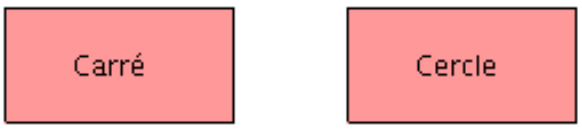
```
pour exemple
# Si on déplace la souris, se positionner à la nouvelle position
si lissouris=0 [fpos possouris]
exemple
fin
```

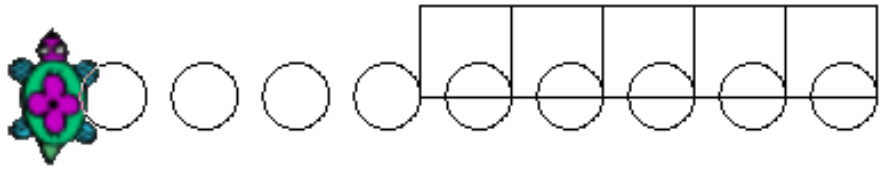
Dans cette deuxième procédure, c'est le même principe sauf qu'il faut cliquer avec le bouton gauche de la souris pour provoquer le déplacement de la tortue sur la zone de dessin.

```
pour exemple2
si lissouris=1 [fpos possouris]
exemple2
fin
```

Dans ce troisième exemple, nous allons créer deux boutons. Celui de gauche permettra de tracer un carré de 40 sur 40 vers la droite, celui de droite un petit cercle vers la gauche. Enfin, si l'on clique avec le troisième bouton de la souris sur le bouton de droite, cela provoquera l'arrêt du programme.

```
pour bouton
#crée un bouton rectangulaire de 50 sur 100 colorié en saumon
repete 2[av 50 td 90 av 100 td 90]
td 45 lc av 10 bc fcc [255 153 153]
remplis re 10 tg 45 bc fcc 0
fin

pour lance
ve bouton lc fpos [150 0] bc bouton
lc fpos [30 20] bc etiquette "Carré
lc fpos [180 20] bc etiquette "Cercle
lc fpos [0 -100] bc
souris
fin

pour souris
# On enregistre le résultat de lissouris dans la variable ev
donne "ev lissouris
# On enregistre la première coordonnée de la souris dans la variable x
donne "x item 1 possouris
# On enregistre la première coordonnée de la souris dans la variable y
donne "y item 2 possouris
# Si l'on clique sur le bouton de gauche
si :ev=1 & :x>0 & :x<100 & :y>0 & :y<50 [carre]
# Si l'on clique sur le bouton de droite
si  :x>150 & :x<250 & :y>0 & :y<50 [
          si :ev=1 [cercle]
          si :ev=3 [stop]
]
souris
fin

pour cercle
repete 90 [av 1 tg 4] tg 90 lc  av 40 td 90 bc
fin

pour carre
repete 4 [av 40 td 90] td 90 av 40 tg 90
fin
```





A.14.5 Utiliser des composants graphiques

XLogooffre également la possibilité de rajouter certains composants graphiques à la zone de dessin (Bouton, menu déroulant...). Ces composants étant associés aux interfaces graphiques, toutes les primitives se rapportant à ce sujet débute par le préfixe « gui »(Graphical User Interface).

Créer un composant

Pour manipuler ces objets graphiques, il est tout d'abord nécessaire de les créer, de leur adjoindre certaines propriétés puis de les afficher ensuite.

- Pour créer un bouton :

  guibouton *mot1 mot2*
  Cette commande crée un bouton dont le nom identifiant est *mot1* et sur lequel est écrit *mot2*
  Exemple : `guibouton "b "Cliquer`
- Pour créer un menu déroulant :

  guimenu *mot1 liste2*
  Cette commande crée un menu déroulant dont le nom est *mot1* contenant les entrées de la liste *liste2*
  Exemple : `guimenu "m [item1 item2 item3]`

Attribuer certaines propriétés à un composant

guiposition *mot1 liste2*
Permet de positionner l'élément graphique à l'endroit désiré sur la zone de dessin. Par exemple, pour positionner le bouton précédent au point de coordonnées $(20; 100)$, on écrit :

```
guiposition "b [20 100]
```

Si la position du composant n'est pas précisée, le composant est placé par défaut au coin supérieur gauche de la zone de dessin.

guienleve *mot1*
Supprime un élément graphique. Par exemple, pour supprimer le bouton précédent :

```
guienleve "b
```

guiaction *mot1 liste2*
Définit une action à réaliser lorsque l'utilisateur interagit avec l'élément graphique considéré.

```
# La tortue avance de 100 pas si l'on clique sur le bouton "b
 guiaction "b [av 100 ]

# Pour le menu déroulant, chaque item possède sa propre action
guiaction "m [[ecris "item1]  [ecris "item2] [ecris "item3]]
```

guidessine *mot1*
Affiche le composant graphique sur la zone de dessin. Par exemple, pour afficher le bouton :

```
guidessine "b
```

A.15 Gestion du temps

XLogodispose de plusieurs primitives permettant de connaître l'heure, la date ou encore de gérer des comptes à rebours (utiles pour répéter une tâche à des intervalles fixés).

attends n
Bloque le programme et donc la tortue pendant n $60^{\text{ème}}$ de secondes.

debuttemps n
Lance un compte à rebours de n secondes. On peut savoir si le compte à rebours est terminé à l'aide de la primitive `fintemps ?`

fintemps ?
Rend `"vrai` si aucun compte à rebours n'est actif. Rend `"faux` si le compte à rebours n'est pas terminé.

date
Renvoie une liste formé de trois entiers représentant la date. Le premier indique le jour, le second le mois et le dernier l'année. —> [jour mois année]

heure
Renvoie une liste de trois entiers représentant l'heure. Le premier entier représente les heures, le second les minutes et le dernier les secondes. —> [heure minute seconde]

temps
Renvoie le temps écoulé depuis le démarrage de XLogo. Ce temps est exprimé en secondes.

Voici une petite procédure exemple :

```
pour horloge
# affiche l'heure sous forme numérique
# (on actualise l'affichage toutes les 5 secondes)
si fintemps? [
ve
fixepolice 75 ct
donne "heu heure
donne "h premier :heu
donne "m item 2 :heu
#affichage à deux chiffres des minutes (on rajoute le 0)
si :m-10<0 [donne "m mot 0 :m]
donne "s dernier :heu
#affichage à deux chiffres des secondes
si :s-10<0 [donne "s mot 0 :s]
etiquette mot mot mot mot :h ": :m ": :s
debuttemps 5
]
horloge
fin
```

A.16 Se servir du réseau avec XLogo

A.16.1 Le réseau : comment ça marche ?

Tout d'abord, dans cette introduction, il est nécessaire de vous expliquer certains termes de vocabulaire afin de bien comprendre l'usage des différentes primitives. Deux ordinateurs peuvent commuiquer via le

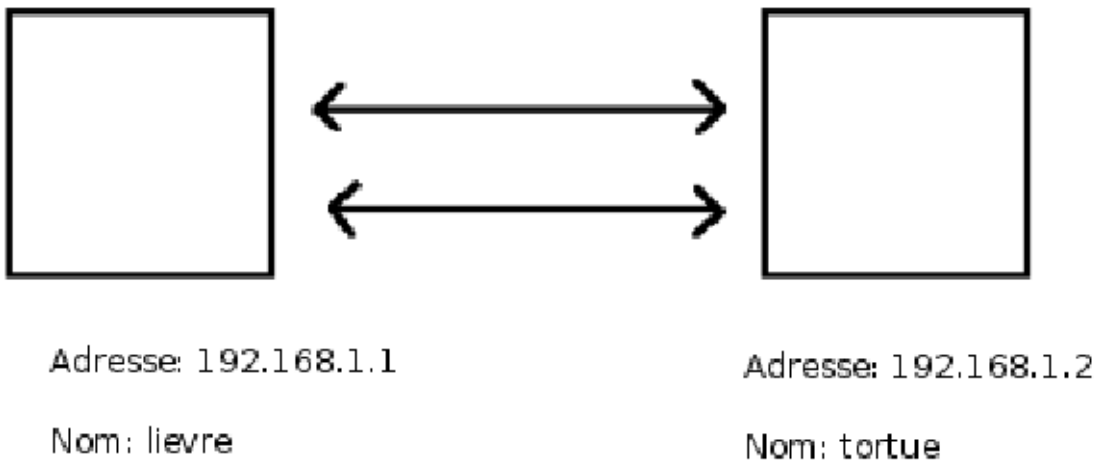


FIG. A.5 – Notion de réseau

réseau s'ils sont équipés de carte réseau (appelée aussi carte ethernet). Chaque ordinateur est alors repéré par une adresse personnelle : *Son adressse IP*. Cette adresse IP est composée de 4 entiers compris entre 0 et 255 séparés par des points. Par exemple, l'adresse IP du premier ordinateur du schéma précédent est 192.168.1.1.

Etant donné qu'il n'est pas facile de retenir ce genre d'adresse, il est également possible de faire correspondre à chaque adresse IP un nom usuel plus facile à retenir. Sur le schéma précédent, on peut ainsi s'adresser à l'ordinateur de droite soit en l'appelant par son adresse IP : 192.168.1.2, soit en l'appelant par son nom : `tortue`

Je ne m'étends pas davantage sur la signification de ces nombres. Je rajoute juste une chose qu'il est bon de savoir, l'ordinateur local sur lequel on travaille est repéré également par une adresse : 127.0.0.1. Le nom qui lui est associé est généralement `localhost`

A.16.2 Primitives orientées réseau

XLogo dispose de 4 primitives permettant de communiquer grâce au réseau : `ecoutetcp`, `executetcp`, `chattcp` et `envoietcp`. On prendra toujours dans les exemples qui suivent le cas des deux ordinateurs du schéma précédent.

ecoutetcp

Cette primitive `ecoutetcp` est la base de toute communication réseau. Elle n'attend aucun argument. Elle permet de mettre l'ordinateur qui l'exécute à l'écoute d'ordres donnés par d'autres ordinateurs du réseau.

executetcp *mot1 liste2*

Cette primitive permet d'exécuter des instructions sur un ordinateur du réseau.
mot1 désigne l'adresse IP ou le nom de l'ordinateur appelé, *liste2* contient les instructions à exécuter.

Exemple : Je suis sur l'ordinateur `lievre`, je souhaite tracer un carré de côté 100 sur l'autre ordinateur. Par conséquent, il faut que sur l'ordinateur `tortue`, je lance la commande `ecoutetcp`. Ensuite, sur l'ordinateur `lievre`, je lance :

```
executetcp "192.168.1.2 [repete 4[av 100 td 90]]
ou
executetcp "tortue [repete 4[av 100 td 90]]
```

chattcp *mot1 liste2*

Permet de dialoguer entre deux ordinateurs du réseau en affichant une fenêtre permettant la conversation. *mot1* désigne l'adresse IP ou le nom de l'ordinateur appelé, *liste2* contient la phrase à afficher.

Exemple : `lievre` veut discuter avec `tortue`.
`tortue` lance `ecoutetcp` pour se mettre en attente de requête d'ordinateurs du réseau. `lievre` lance alors : `chattcp "192.168.1.2 [bonjour]`.
Deux fenêtres permettant le dialogue s'ouvre alors sur chacun des ordinateurs.

envoietcp *mot1 liste2*

Envoie des données vers un ordinateur du réseau puis renvoie la réponse de l'autre ordinateur.
mot1 désigne l'adresse IP ou le nom de l'ordinateur appelé, *liste2* contient les données à envoyer. Si la communication se fait avec un autre ordinateur où XLogoest lancé, cet ordinateur répondra OK une fois l'opération terminée. Il est également possible de dialoguer avec un robot muni d'une interface réseau, la réponse pourra être différente à ce moment.

Exemple : `tortue` veut envoyer à `lievre` la séquence "3.14159 presque le nombre pi".
`lievre` lance `ecoutetcp` pour se mettre en attente de requête d'ordinateurs du réseau. `tortue` lance alors : `ecris envoietcp "lievre [3.14159 presque le nombre pi]`.

Une petite astuce : Lancer deux fois XLogo sur le même ordinateur.

- Dans la première fenêtre, lancer `ecoutetcp`.
- Dans la seconde, lancer `executetcp "127.0.0.1 [av 100 td 90]`

Vous avez ainsi déplacer la tortue sur l'autre fenêtre ! (éh oui, 127.0.0.1 désigne l'adresse locale donc l'ordinateur lui-même...)

Annexe B

Lancement de XLogo en ligne de comandes

Voici la syntaxe de la commande à taper pour lancer XLogo :

```
java -jar xlogo.jar [-a] [-lang fr] [-memory 64][fichier1.lgo fichier2.lgo ...]
```

Détails des différentes options disponibles :

- Attribut `-lang` : cet attribut permet de spécifier un langage particulier pour XLogo. Ce paramètre écrase celui contenu dans le fichier personnel de configuration nommé `.xlogo`. Les différentes langues sont accessibles suivant ce tableau :

Français	Anglais	Espagnol	Allemand	Arabe	Portugais	Espéranto	Galicien	Grec
fr	en	es	de	ar	pt	eo	gl	el

- Attribut `-a` : cet attribut permet d'exécuter dès l'ouverture de XLogo la commande principale contenue dans les fichiers chargés au démarrage.

- Attribut `-memory` : cet attribut permet de fixer la mémoire allouée à la machine virtuelle Java.

- fichier1.lgo, fichier2.lgo ... : ces fichiers au format `.lgo` sont chargés au démarrage de XLogo. Ces fichiers peuvent être locaux ou distants c'est à dire que leur adresse peut aussi bien désigner l'arborescence locale qu'une adresse Internet.

- Attribut `tcp_port` : cet attribut permet de sélectionner un numéro de port précis pour les communications réseaux. Le port par défaut est 1948. Voir p.128

Quelques exemples de commandes :

- `java -jar xlogo.jar -lang es prog.lgo` : Les fichiers `xlogo.jar` et `prog.lgo` sont contenus dans le répertoire courant. Cette commande lance XLogo configuré en espagnol et charge ensuite le fichier `prog.lgo` (Qui doit par conséquent être rédigé en espagnol...)

- `java -jar xlogo.jar -a -lang en http ://xlogo.tuxfamily.org/prog.lgo` :
  Cette commande lance XLogo configuré en anglais et charge le fichier nommé `http ://xlogo.tuxfamily.org/prog.lgo`. Pour finir, la commande principale définie dans ce fichier est exécutée au démarrage.

Annexe C

Lancer Xlogo depuis le WEB

Vous disposez d'un site web sur lequel vous parlez de XLogo. Mieux encore, vous souhaiteriez mettre à disposition certains des programmes que vous avez créer. Plutôt que de distribuer simplement les fichiers `.lgo`, il serait plus agréable pour l'utilisateur de pouvoir lancer XLogo en ligne afin de tester directement ces exemples. Voici la procédure à suivre :

Le lancement de Xlogo en ligne est assuré par la technologie JAVA WEB START. En fait, il suffit de fournir sur votre site un lien vers un fichier d'extension `.jnlp`, cela assurera l'exécution de XLogo.

Création du fichier d'extension `jnlp`

Voici ci-dessous un exemple d'un tel fichier. Ce fichier est en fait celui utilisé dans la rubrique « exemples » du site français. Il permet de charger le programme traçant le dé dans la section 3D. Les grandes lignes d'explication sont données par la suite.

```
 <?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.5+" codebase="http://downloads.tuxfamily.org/xlogo/common/webstart">
<information>
  <title>XLogo</title>
  <vendor>xlogo.tuxfamily.org</vendor>
  <homepage href="http://xlogo.tuxfamily.org"/>
  <description>Logo Programming Language</description>
  <offline-allowed/>
</information>

<security>
<all-permissions/>
</security>

<resources>
  <j2se version="1.4+"/>
  <jar href="xlogo.jar"/>
</resources>

<application-desc main-class="Lanceur">
  <argument>-lang</argument>
  <argument>fr</argument>
  <argument>-a</argument>
  <argument>http://xlogo.tuxfamily.org/fr/html/examples-fr/3d/de.lgo</argument>
</application-desc>
</jnlp>
```

Ce fichier est écrit en respectant le format XML. La partie importante est située à la fin du fichier notamment ces 4 lignes :

```
<argument>-lang</argument>
<argument>fr</argument>
<argument>-a</argument>
<argument>http://xlogo.tuxfamily.org/fr/html/examples-fr/3d/de.lgo</argument>
```

C'est ici que sont spécifiés les paramètres de lancement.

- Les deux premières lignes force l'utilisation de la langue française.
- La dernière ligne indique l'adresse du fichier à charger.
- La troisième ligne indique que la commande principale de ce fichier sera exécutée au démarrage de XLogo.

Dernière petite astuce : Si vous désirez ne pas surcharger le serveur de Tuxfamily, vous avez la possibilité de déposer le fichier `xlogo.jar` sur votre site. Pour lier le fichier `.jnlp` à ce fichier, il vous suffit de changer l'adresse contenue à la ligne 2 après `codebase=`

Annexe D

Corrigé des activités

D.1 Chapitre 2

```
pour carre
repete 4[av 150  td 90]
fin

pour tri
repete 3[av 150 td 120]
fin

pour porte
repete 2[av 70 td 90 av 50 td 90]
fin

pour che
av 55 td 90 av 20 td 90 av 20
fin

pour dep1
td 90 av 50  tg 90
fin

pour dep2
tg 90 av 50 td 90 av 150 td 30
fin

pour dep3
lc td 60 av 20 tg 90 av 35 bc
fin

pour ma
carre dep1 porte dep2 tri dep3 che
fin
```

D.2 Chapitre 3

```
pour supercube
ve lc fpos[ -30 150] bc fpos[-150 150]  fpos[-90 210] fpos[30 210] fpos[-30 150]
fpos[-30 -210] fpos[30 -150] fpos[30 -90] fpos[-30 -90] fpos[90 -90] fpos[90 30]
```

```
fpos[-270 30] fpos[-270 -90] fpos[-210 -90] fpos[-210 -30] fpos[-90 -30] fpos[-90 -150]
fpos[-210 -150] fpos[-210 -30] fpos[-150 30] fpos[-30 30] fpos[-90 -30] fpos[90 150]
fpos[30 150] fpos[30 210] fpos[30 90] fpos[90 90] fpos[90 150] fpos[90 90] fpos[150 90]
fpos[150 -30] fpos[90 -90] fpos[90 30] fpos[150 90] lc fpos[-150 30] bc fpos[-150 150]
fpos[-150 90] fpos[-210 90] fpos[-270 30] lc  fpos[-90 -150] bc fpos[-30 -90]
lc fpos[-150 -150] bc fpos[-150 -210] fpos[-30 -210]
fin
```

D.3 Chapitre 4

D.3.1 Le robot

Le premier dessin est composé exclusivement de motif élémentaire à base de rectangle, carré et triangle. Voici le code associé à ce dessin :

```
pour rec :lo :la
# trace un rectangle de longueur :lo et largeur :la
repete 2[av :lo td 90 av :la td 90]
fin

pour carre :c
# trace un carre de cote :c
repete 4[av :c td 90]
fin

pour tri :c
# trace un triangle equlateral de côté :c
repete 3[av :c td 120]
fin

pour patte :c
rec 2*:c 3*:c carre 2*:c
fin

pour antenne :c
av 3*:c tg 90 av :c td 90 carre 2*:c
lc re 3 *:c td 90 av :c tg 90 bc
fin

pour robot :c
ve ct
# Le corps
rec 4*:c 28* :c
# Les pattes
td 90 av 2*:c patte :c av 4* :c patte :c av 14*:c patte :c av 4*:c patte :c
# La queue
lc tg 90 av 4* :c bc td 45 av 11*:c re 11 * :c tg 135
# le cou et la tête
av 18 *:c carre :c av 3*:c carre :c td 90 av :c tg 90 av 2*:c td 90 carre 8* :c
# Oreilles
av 4*:c tg 60 tri 3*:c lc td 150 av 8 *:c tg 90 bc tri 3*:c
# Les antennes
av 4 *:c tg 90 av 2*:c td 90 antenne :c tg 90 av 4*:c td 90 antenne :c
# les yeux
```

```
lc re 3 *:c bc carre :c td 90 lc av 3*:c bc tg 90 carre :c
# La bouche
lc re 3*:c tg 90 av 3*:c td 90 bc rec :c 4*:c
fin
```

D.3.2 La grenouille

```
pour gre :c
ve ct
av 2 *:c td 90 av 5*:c tg 90 av 4*:c tg 90 av 7 *:c td 90 av 7*:c td 90
av 21 *:c td 90 av 2*:c tg 90 av 2*:c td 90 av 9*:c td 90 av 2*:c tg 90
av 2*:c td 90 av 9*:c td 90 av 2*:c td 90 av 7*:c re 5*:c tg 90 av 4*:c
td 90 av 4*:c re 4*:c tg 90 re 2*:c tg 90 av 5*:c tg 90 av 4*:c td 90 av 7*:c
td 90 lc av 9*:c bc repete 4[av 2*:c td 90]
fin
```

D.4 Chapitre 8 :

```
pour jeu
# On initialise le ,nombre recherché et le nombre de coups
donne "nombre hasard 32
donne "compteur 0
boucle
fin

pour boucle
lis [proposez un nombre] "essai
si nombre? :essai[
  # Si la valeur rentrée est bien un nombre
  si :nombre=:essai[ec ph ph [vous avez gagné en ] :compteur+1 [coup(s)]][
    si :essai>:nombre [ec [Plus petit]][ec [Plus grand]]
    donne "compteur :compteur+1
    boucle
  ]
]
[ecris [Vous devez rentrer un nombre valide!] boucle]
fin
```

Annexe E

Foire aux questions - Astuces - trucs à connaître

E.1 J'ai beau effacer une procédure dans l'éditeur, elle réapparaît tout le temps !

Lorsqu'on sort de l'éditeur, celui-ci se contente juste de sauver ou de mettre à jour le contenu de l'éditeur. Le seul moyen d'effacer une procédure dans XLogo est d'utiliser la primitive `effaceprocedure` ou `efp`. Exemple : `effaceprocedure "toto` ⟶ efface la procédure `toto`.

E.2 J'utilise la version en espéranto mais je ne peux écrire les caractères spéciaux !

Lorsque vous tapez dans la ligne de commande ou l'éditeur, si vous faites un clic avec le bouton droit de la souris, apparait un menu déroulant. Dans ce menu, figurent les traditionnels fonctions d'édition (copier, couper, coller) et les caractères spéciaux de l'espéranto lorsque ce langage est sélectionné.

E.3 Dans l'onglet Son de la boîte de dialogue Préférences, aucun instrument n'est disponible.

Parfois, la liste des instruments MIDI n'apparait pas dans `Outils/Préférences/Son` et on ne peut pas exploiter totalement les fonctionnnalités sonores de XLogo. Rendez-vous à cette adresse :

`http ://java.sun.com/products/java-media/sound/soundbanks.html`

Télécharger une des banques sonores (soundbank) proposées : minimal, midsize ou deluxe puis décompresser-la dans `C :\Program Files\Java\jre1.6.0_05\lib\audio\`.

- le dossier `jre1.6.0_05` correspond à votre version du JRE installée.
- si le dossier `audio` n'existe pas, il faudra le créer.
- il faudra renommer le fichier décompressé en : `soundbank.gm`

Ensuite relancez XLogo et allez donc voir dans `Outils/Préférences/Son`

E.4 Comment faire pour taper rapidement une commande déjà utilisée au préalable ?

- Première méthode : avec la souris, cliquer dans la zone d'historique sur la ligne désirée, elle réapparaîtra immédiatement dans la ligne de commande.
- Deuxième méthode : avec le clavier, les flèches Haut et Bas permettent de naviguer dans la liste des dernières commandes tapées(Très pratique).

E.5 Comment peut-on vous aider ?

- En reportant tout bug constaté. Si vous êtes capable de reproduire systématiquement un problème constaté, c'est encore mieux.
- Vos suggestions en vue de l'amélioration sont toujours les bienvenues.
- En aidant aux traductions : en particulier l'anglais...
- Un petit encouragement fait toujours du bien !

Remerciements

- Je tiens à remercier tout d'abord l'ensemble des traducteurs actifs de XLogo.

  - Anglais : Guy Walker
  - Espagnol : Marcelo Duschkin, Alvaro Valdes Menendez
  - Arabe : El Houcine Jarad
  - Portugais : Alexandre Soares
  - Allemand : Michael Malien
  - Espéranto : Michel Gaillard,Carlos Enrique Carleos Artime
  - Galicien : Justo freire
  - Grec : Anastasios Drakopoulos
  - Italien : Marco Bascietto
  - Catalan : David Arso
  - Hongrois : József Varga

- Je tiens également à remercier tout particulièrement Eitan Gurari pour sa patience, et pour l'extension LaTeX `tex4ht` qui me permet l'export des manuels vers différents formats

  `www.cse.ohio-state.edu/~gurari/TeX4ht`

- Plusieurs projets OpenSource permettant à XLogo d'exister :

  - Java3D : `https ://java3d.dev.java.net/`
  - JavaHelp : `http ://java.sun.com/javase/technologies/desktop/javahelp/`
  - Eclipse : `http ://www.eclipse.org/`
  - JLayer : `http ://www.javazoom.net/javalayer/javalayer.html`

- Enfin, un GRAND merci à Tuxfamily pour la qualité de l'hébergement fourni et à leur engagement envers le logiciel libre !

  `http ://www.tuxfamily.org`

Index