AMSTRAD

GUIDE DE LOGO

ANATOLE D'HARDANCOURT

SYBEX

AMSTRAD
GUIDE DE LOGO

Amstrad est une marque déposée de Amstrad Consumer Electronic Plc.
CP/M, CP/M plus et Dr Logo sont des marques déposées de Digital Research Inc.

Tous les efforts ont été faits pour fournir dans ce livre une information complète et exacte. Néanmoins, Sybex n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Copyright (c) 1986 Sybex

Tous droits résevés. Toute reproduction même partielle, par quelque procédé que ce soit, est interdite sans autorisation préalable. Une copie par xérographie, photographie, film, bande magnétique ou autre, constitue une contrefaçon passible des peines prévues par la loi sur la protection des droits d'auteurs.

ISBN 2-7361-0171-5

SOMMAIRE

CHAPITRE I

INSTALLATION

Branchement et mise en route

Après avoir déballé le matériel et vérifié son bon état, la première chose à faire est de l'installer correctement et de connecter les différents éléments.

L'installation des Amstrad CPC 464, 664 et 6128 se résume à la connexion des fiches d'alimentation 12 volts et 5 volts et à celle de la prise vidéo. (La prise 12 volts n'existe pas sur le 464.) Aucune confusion possible, la fiche 12 volts se connectant sur le moniteur et la fiche 5 volts sur l'unité centrale. La prise vidéo se connectant sur l'unité centrale est de type DIN et ne peut être confondue avec aucune des deux autres. Pour utiliser Logo, Le CPC 464 doit être connecté à au moins un lecteur de disquette externe. Celui-ci se connecte sur la face arrière de l'ordinateur.

L'installation des PCW 8256 et 8512 n'est pas plus compliquée, seul le clavier et l'imprimante étant à connecter.

Quelques précautions doivent être prises lors de l'installation du matériel. La disposition des différents éléments doit en général répondre aux critères de confort de l'utilisateur (clavier suffisamment bas pour éviter la fatigue des bras lors de la frappe, moniteur aussi loin de l'unité centrale que le permet la longueur des câbles de liaison pour la sécurité des yeux, éviter les éclairages de face ou de dos et préférer un éclairage latéral afin d'éviter les reflets sur l'écran). Une précaution supplémentaire est cependant à observer : l'unité de disques doit être aussi éloignée que possible de toute source de rayonnements électromagnétiques. On placera donc l'unité centrale des CPC 664 et 6128 plutôt décalée vers la droite par rapport au moniteur, celui-ci constituant une source de rayonnement intense. (Sur les PCW 8256 et 8512, l'unité de disques se trouve très près de l'écran, mais un blindage a été prévu à l'intérieur de l'appareil pour la protéger des rayonnements.) De

même, le lecteur externe du 464 sera placé le plus loin possible de l'écran. On évitera également la proximité des câbles d'alimentation ainsi que de toute ligne électrique, d'un téléphone, de tout appareil comportant un haut-parleur et, bien sûr, d'un aimant ou d'un électro-aimant. Les mêmes précautions s'appliquent à l'installation d'une unité de disques supplémentaire et à la manipulation des disques. En effet, les informations y sont enregistrées sous forme de minuscules champs magnétiques qui risquent d'être perturbés par toute influence magnétique extérieure. Les risques de détérioration des disques ne se limitent malheureusement pas aux influences magnétiques. Les disques craignent les chocs, la poussière, les liquides, la chaleur, le froid, le contact des doigts ou de tout objet sur leur surface magnétique, et bien d'autres choses encore. Deux types de précautions peuvent être pris pour la sécurité des disques et des informations qu'ils contiennent. (N'oubliez jamais que ce qui fait la valeur d'un disque, ce sont les informations qu'il contient. Imaginez la perte qui résulterait de la destruction d'un disque contenant le résultat d'un mois de travail !) Un premier type de précaution est d'éviter tout risque de détérioration en respectant les consignes d'utilisation et de stockage (en particulier, toujours remettre un disque dans sa boîte en plastique après usage et ne jamais laisser un disque dans le lecteur lorsque l'ordinateur n'est pas en fonctionnement). Le second type de précaution à prendre OBLIGATOIREMENT est de tout faire pour réduire au maximum les conséquences d'un accident. Pour cela, une seule solution : faire des copies de sauvegarde. La règle à respecter impérativement est qu'il doit toujours exister une copie de tout disque manipulé. Les opérations de copie de disques nécessitant la manipulation simultanée de deux disques (l'original et la copie en cours de réalisation), on constate qu'il doit toujours exister TROIS copies au moins de chaque disque. Ainsi, si un accident arrive pendant une copie et que l'original et la copie en cours sont détruits, tout n'est pas perdu. La seconde règle est que toute copie détruite doit être IMMEDIATEMENT reconstituée. Les copies sont destinées uniquement à la sécurité et ne doivent jamais être utilisées. De plus, si les disques contiennent des informations de grande valeur, il faut impérativement ranger les copies dans des endroits différents.

La première opération indispensable va donc être d'effectuer une copie des disques livrés avec l'ordinateur. En effet, ces disques contiennent tous les programmes nécessaires pour utiliser votre or-

dinateur (et en particulier Logo). En cas de destruction de ces disques, votre ordinateur ne pourrait donc plus fonctionner. Il faudrait alors en commander de nouveaux au constructeur de l'appareil, ce qui risquerait d'immobiliser la machine à un moment où vous pourriez en avoir besoin.

Mise en route de l'ordinateur

Avant de mettre en route l'Amstrad, vérifiez toujours que le lecteur ne contient pas de disque. Allumer ou éteindre l'ordinateur avec un disque dans le lecteur a de grandes chances d'entraîner un effacement des informations qui s'y trouvent enregistrées. Vous devrez donc également toujours retirer le disque du lecteur avant d'éteindre l'ordinateur.

Après avoir vérifié qu'aucun disque ne se trouve dans le lecteur, actionnez l'interrupteur à poussoir se trouvant en bas à droite (464, 664 et 6128) ou à gauche (8256 et 8512) de la face avant du moniteur, ce qui a pour effet de mettre celui-ci sous tension. Sur les 464, 664 et 6128, placez ensuite l'interrupteur se trouvant sur la face latérale droite de l'ordinateur (464 et 664) ou sur la face arrière (6128) sur la position ON. Dans le cas des 464, 664 et 6128, l'écran affiche :

```
Amstrad xxK Microcomputer  (vx)
(c)1985 Amstrad Consumer Electronics plc
            and Locomotive Software Ltd.

BASIC 1.1

Ready
```

Ce message indique que l'ordinateur se trouve sous BASIC et est prêt à travailler. Le rectangle de couleur se trouvant sous le message `Ready` est le curseur. Il indique à quel endroit sera affiché le prochain caractère tapé. Sur l'écran du 8256 et du 8512, rien n'est affiché.

Lancement de CP/M

Insérez maintenant le disque système dans le lecteur, étiquette vers vous et face 1 vers le haut dans le cas des 464, 664 et 6128, vers la gauche s'il s'agit du 8256 ou du 8512. Le disque doit s'enclencher en faisant ressortir le poussoir d'éjection. Sur les 8256 et 8512, le chargement de CP/M commence automatiquement. Avec un autre modèle, tapez :

```
|cpm
```

Le signe | s'obtient en maintenant la touche SHIFT (majuscule) enfoncée et en tapant la touche @/|, à droite de la touche P. Pressez alors la touche RETURN.

Le lecteur de disques se met en marche. Le fonctionnement du lecteur est indiqué par l'allumage du voyant rouge, sur sa face avant. Vous ne devez JAMAIS toucher au disque se trouvant dans le lecteur lorsque celui-ci est en fonctionnement (voyant rouge allumé). L'écran affiche maintenant :

```
CP/M 2.2 Amstrad Consumer Electronics plc

A>
```

ou

```
CP/M Plus  Amstrad Consumer Electronics plc

v 1.0, 61K TPA, 1 disc drive

A>
```

Le fond de l'écran du CPC 6128 couleur est devenu bleu et les caractères blancs. Avec un 464 ou un 664, on obtient des caractères noirs sur fond bleu ciel. Notez que ceux-ci sont plus étroits que précédemment. L'ordinateur passe en effet automatiquement en mode d'affichage 80 colonnes lors de l'appel de CP/M. L'écran des PCW 8256 et 8512 a toujours une largeur de 90 caractères.

Copie des disques

Pour effectuer la copie des disques fournis avec l'ordinateur, nous allons utiliser la commande DISCKIT sur le 8256 et le 8512, DISCKIT3 sur le 6128 et DISCCOPY sur le 464 et le 664. Ces commandes permettent d'effectuer une copie de disques sur un système à un seul lecteur.

PCW 8256 et 8512

Placez le disque système dans le lecteur (face 2 à gauche) et tapez DISCKIT suivi de la touche RETURN. Le message suivant est affiché à l'écran :

```
                DISC KIT 1.1
              PCW8256 & CP/M Plus
(c) 1985 Amstrad Consumer Electronic plc et Locomotive
                                                Software Ltd.

Un drive en ligne

                                  f6
                                       Copier
                                  f5
                                  f4
                                       Formater
                                  f3
                                  f2
Quitter le programme   EXIT           Vérifier
                                  f1
```

Tapez la touche f5. L'écran affiche maintenant :

```
0   Copier disque CF2


          Toute autre touche pour quitter le menu
```

Placez un des deux disques à copier dans le lecteur et tapez O (la lettre O) pour commencer la copie (toute autre touche vous ramène au menu précédent). Un message est affiché indiquant le format du disque. Un nouveau message demande l'introduction dans l'unité du disque devant recevoir la copie :

```
Insérer disque à ECRIRE
Appuyez sur une touche pour continuer
```

Introduisez un disque vierge dans le lecteur puis tapez une touche quelconque. L'écran affiche :

```
Disque n'est pas formaté
Formatage pendant la copie
Disque sera au format CF2
```

Après quelques instants, l'ordinateur affiche un nouveau message :

```
Insérer disque à LIRE
Appuyez sur une touche pour continuer
```

Placez dans le lecteur le disque à copier et tapez une touche quelconque. L'opération est répétée plusieurs fois jusqu'à ce que la copie soit terminée, puis le message suivant est affiché :

```
Copie terminée
Retirer disque
Appuyer sur une touche pour continuer
```

Retirez la copie de l'unité de disques et tapez une touche quelconque. Vous pouvez alors copier le second disque en tapant O. Notez que les deux faces d'un disque étant indépendantes, elles doivent être copiées séparément. Vous devez donc effectuer quatre opérations de copie pour copier les deux disques livrés avec l'ordinateur.

CPC 6128

Placez le disque système dans le lecteur (face 1 vers le haut) et tapez DISCKIT3 suivi de la touche RETURN. Le message suivant est affiché :

```
                DISC KIT 1.0
              CPC6128 & CP/M Plus
(c) 1985 Amstrad Consumer Electronic plc
         and Locomotive Software Ltd.

One drive found


                Copy                7
                Format              4
                Verify              1
                Exit from program   0
```

Tapez la touche f7. L'écran affiche maintenant :

```
Y           Copy
            Any other key to exit to menu
```

Laissez le disque système dans le lecteur et tapez Y pour commencer la copie (toute autre touche vous ramène au menu précédent). Un message est affiché indiquant le format du disque. Un second message demande l'introduction dans l'unité du disque devant recevoir la copie :

```
Insert disc to WRITE
Press any key to continue
```

Introduisez un disque vierge dans le lecteur puis tapez une touche quelconque. Après quelques instants, l'ordinateur affiche un nouveau message :

```
Insert disc to READ
Press any key to continue
```

Placez dans l'unité le disque à copier et tapez une touche quelconque. L'opération est répétée plusieurs fois jusqu'à ce que la copie soit terminée, puis le message suivant est affiché :

```
Copy completed
Remove disc
Press any key to continue
```

Retirez la copie de l'unité de disques et tapez une touche quelconque. Vous pouvez alors copier un autre disque en tapant Y ou retourner au menu général en tapant une autre touche. Les deux faces de chaque disque étant indépendantes, vous devez effectuer quatre opérations de copie (une par face) pour copier les deux disques livrés avec l'ordinateur.

464 et 664

Le disque système se trouvant dans le lecteur, face 1 vers le haut, tapez :

```
A>DISCCOPY <ENTER>
```

L'écran affiche :

```
DISCCOPY V2.0

Please insert source disc into drive A then press any key:
```

Ce message vous demande de placer le disque à copier (disque source) dans le lecteur A puis de taper une touche quelconque. (Sur le CPC 664, le lecteur A est le lecteur intégré. Sur le 464, le lecteur A est celui qui est le plus éloigné de l'ordinateur.) Si vous n'obtenez pas ce message, deux causes sont possibles. Vous avez fait une erreur de frappe (recommencez) ou vous avez oublié de placer le disque système face 1 vers le haut (retournez-le et recommencez).

Le disque à copier se trouvant dans le lecteur A, appuyez simplement sur une touche quelconque (la barre d'espacement par

exemple). Le lecteur de disques se met en marche pendant que l'écran affiche divers messages puis :

```
Copying started

Please insert destination disc into drive A then press any key:
```

L'ordinateur indique que la copie a commencé et vous demande d'introduire le disque qui contiendra la copie (disque destination) dans le lecteur. Retirez le disque système du lecteur et placez-y un disque vierge. Une fois cette opération effectuée, tapez une touche quelconque : le lecteur se remet en marche et copie les huit premières pistes du disque avant de demander à nouveau l'introduction du disque source. Cette opération est répétée cinq fois pour copier entièrement le disque, puis l'ordinateur affiche le message :

```
Copying complete

Do you want to copy another disc (Y/N)
```

La copie de la première face est terminée. Pour copier la seconde face, répondez Y à cette question. Lorsque l'ordinateur demande de placer le disque source ou le disque destination dans le lecteur, insérez-les avec la face 2 vers le haut. Lorsque la copie de la seconde face est terminée, répondez N à la question précédente. L'écran affiche :

```
Please insert a CP/M system disc into drive A then press any key
```

Tapez une touche quelconque pour retourner sous CP/M.

Les disques système sont "protégés en écriture", c'est-à-dire qu'il est impossible d'écrire sur ces disques. Il est souhaitable de posséder deux jeux de copies des disques système. Pour plus de sécurité, vous pourrez protéger en écriture un des jeux de copies. Pour protéger un disque, reportez-vous à la Figure 1.1

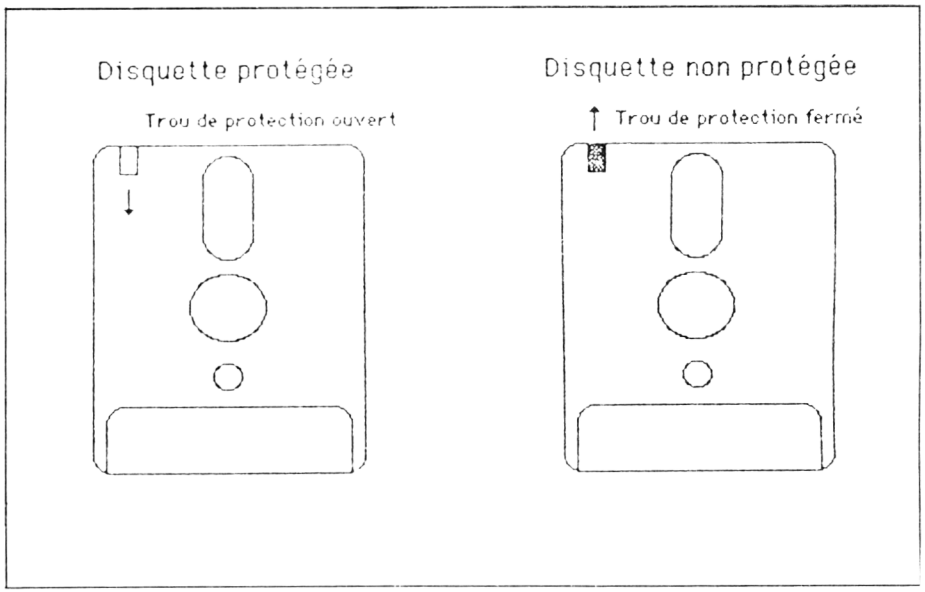


Figure 1.1 : Protection d'un disque.

Préparation d'un disque de travail Logo

Pour programmer en Logo, vous aurez besoin d'un disque de travail contenant l'interpréteur Logo et quelques utilitaires, et sur lequel vous pourrez enregistrer les programmes que vous allez écrire.

464 et 664

Copiez sur un disque vierge la face 2 du disque système livré avec l'ordinateur en utilisant la commande DISCCOPY décrite précédemment. Pour tout renseignement supplémentaire concernant la commande DISCCOPY ou l'utilisation de CP/M, reportez-vous à l'ouvrage *Amstrad, CP/M 2.2*, Sybex.

6128

Formatez un disque vierge à l'aide de la commande DISCKIT3 en demandant le format système. Copiez sur ce disque les fichiers C10CPM3.EMS (face 1), LOGO3.COM, LOGO3.SUB, KEYS.DRL, SUBMIT.COM, SETKEYS.COM (face 3) en utilisant la commande PIP. Pour tout renseignement concernant les commandes DISCKIT3 et PIP, reportez-vous à l'ouvrage Amstrad, CP/M plus, Sybex, 1985.

8256 et 8512

Formatez un disque vierge à l'aide de la commande DISCKIT. Copiez sur ce disque les fichiers J12FCPM3.EMS, SETKEYS.COM, LANGUAGE.COM et SUBMIT.COM (face 2), Logo.COM, Logo.SUB, KEYS.DRL, SUBMIT.COM, (face 4) en utilisant la commande PIP. Pour tout renseignement concernant les commandes DISCKIT et PIP, reportez-vous à l'ouvrage *Amstrad, CP/M plus*, SYBEX, 1985

Vous êtes maintenant en possession d'un disque système contenant le langage Logo. Vous pouvez démarrer l'ordinateur directement à l'aide de ce disque, puis charger Logo en tapant :

```
Logo <ENTER>
```

sur un 464 ou un 664,

```
SUBMIT LOGO3 <RETURN>
```

sur un 6128 et

```
LANGUAGE 0 <RETURN>
SUBMIT Logo <RETURN>
```

sur un 8256 ou un 8512.

Les 464 et 664 possèdent une touche marquée ENTER. Les 8256 et 8512 possèdent une touche marquée RETURN. Le 6128 possède à la fois la touche RETURN et la touche ENTER. Ces deux touches

sont équivalentes. Dans cet ouvrage, nous ne parlerons que de la touche RETURN. Les utilisateurs de 464 et de 664 traduiront par ENTER.

CHAPITRE II

PREMIERS PROGRAMMES

Une fois la commande de lancement de Logo tapée, l'écran affiche un court instant un message indiquant le copyright et le numéro de version, puis il est entièrement effacé pour ne plus laisser apparaître qu'un point d'interrogation dans le coin supérieur gauche. Ce point d'interrogation est l'indicatif de Logo. Il signifie que Logo est prêt et attend que vous tapiez quelque chose. Tapez par exemple :

```
?BONJOUR
```

(vous ne devez pas taper le point d'interrogation, qui est affiché par Logo). Il ne se passe absolument rien. En fait, tant que vous ne tapez pas la touche RETURN, Logo ignore totalement ce qui a été tapé. Ce n'est que lorsque la touche RETURN est tapée que Logo essaie d'interpréter la ligne affichée. Tapez la touche RETURN. L'écran affiche :

```
?BONJOUR
I don't know to BONJOUR
?
```

Ce message en anglais signifie "Je ne sais pas BONJOUR". En fait, Logo attend de vous que vous lui demandiez d'exécuter une action désignée par un verbe. Chaque fois que Logo ne comprend pas ce qui lui est demandé, il se contente de répéter la demande précédée de "I don't know to" (Je ne sais pas). Nous verrons en fait plus tard que ce message à une signification bien précise relative aux "propriétés" de l'objet BONJOUR.

Nous allons commencer l'étude du langage par l'apprentissage du maniement de la "tortue". Il serait dommage de réduire Logo au seul maniement de la tortue, mais celle-ci est un instrument pédagogique remarquable.

L'écran de l'Amstrad peut être consacré exclusivement au texte ou exclusivement au graphisme. Il peut également être partagé en deux

zones distinctes : la partie supérieure pour le graphisme et la partie inférieure pour le texte. Au chargement de Logo, l'écran est en mode Texte. Nous allons mettre l'écran en mode mixte texte et graphisme. L'indicatif de Logo (?) se trouve pour l'instant en haut de l'écran. Tapez :

```
?ss <Return>
```

(<Return> signifie que vous devez taper la touche portant l'inscription RETURN (ou ENTER) et non les lettres R E T U R et N. Le point d'interrogation est affiché par Logo et ne doit pas être tapé.) L'indicatif s'est déplacé vers le tiers inférieur de l'écran. Les deux tiers supérieurs sont maintenant réservés au graphisme (ss signifie Split Screen - écran partagé).

Pour l'instant, la tortue n'est pas visible. Pour rendre la tortue visible, nous utiliserons la commande **st** (Show Turtle - montrer tortue). Tapez :

```
?st <Return>
```

Notez bien que ces commandes doivent être tapées en minuscules. En effet, Logo fait la distinction entre les majuscules et les minuscules et ST n'est pas du tout équivalent à **st**. A partir de maintenant, nous n'indiquerons plus la touche RETURN après chaque commande. Il est bien entendu que la touche RETURN doit être tapée à la fin de chaque ligne, sauf indication contraire.

Malgré l'utilisation de la commande **st**, la tortue n'est toujours pas visible ! En fait, la commande **st** fait passer la tortue en mode visible. Mais si la tortue n'est pas là, bien qu'elle soit en mode visible, on ne la voit pas. Pour appeler la tortue sur l'écran, utilisez la commande d'initialisation de l'écran graphique **cs** (Clear Screen - effacer écran). Tapez :

```
?cs
```

Vous voyez maintenant la tortue au centre de l'écran. Celle-ci est représentée par une flèche dont la pointe indique la tête et donc le sens normal de déplacement (en marche avant).

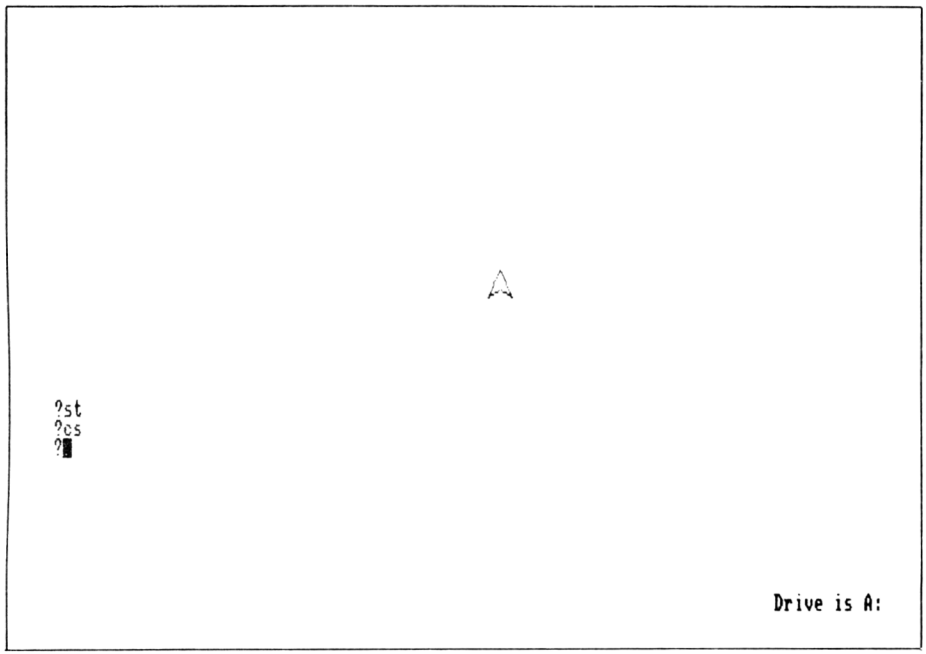


Figure 2.1 : La tortue.

Il existe deux catégories principales de commandes de la tortue : les déplacements et les rotations.

Les rotations peuvent se faire à droite ou à gauche grâce aux commandes **rt** (Right Turn - virage à droite) et **lt** (Left Turn - virage à gauche). Ces deux commandes prennent pour paramètre un angle en degrés indiquant la valeur de la rotation. Essayez de faire tourner la tortue à droite de 90 degrés en tapant la commande :

```
?rt 90
```

La tortue est maintenant horizontale, la tête dirigée vers la droite. Essayez plusieurs valeurs de rotation vers la droite et vers la gauche. Notez que tourner à droite d'un angle négatif donne le même résultat que tourner à gauche d'un angle positif.

Quelle que soit la position de la tortue, il est facile de la replacer dans sa position d'origine en utilisant la commande **home**. Tapez :

```
?home
```

La tortue se trouve de nouveau dans sa position d'origine.

Jusqu'ici, nous n'avons fait effectuer à la tortue que des mouvements relatifs. Il est également possible de placer la tortue dans une position absolue, l'origine des angles étant la position verticale pointe en haut. La commande **seth** permet d'obtenir cet effet. Tapez

```
?seth 90
```

La tortue se trouve maintenant horizontale, dirigée vers la droite. Tapez :

```
?seth 270
```

La tortue se trouve maintenant dirigée vers la gauche. La commande **seth 0** permet comme la commande **home** de replacer la tortue en position verticale, tête en haut. Nous verrons cependant un peu plus loin que la commande **home** a également pour effet de ramener la tortue au centre de l'écran.

Il est également possible avec la commande **seth** d'utiliser des valeurs d'angle négatives. Ainsi :

```
?seth 270
```

est équivalent à

```
?seth -90
```

Toutes les valeurs sont prises modulo 360. Ainsi :

```
?seth 450
```

est équivalent à

```
?seth 90
```

Nous allons maintenant faire faire à la tortue ses premiers pas. L'instruction permettant de déplacer la tortue est l'instruction **fd**. Cette instruction doit être suivie d'un nombre indiquant le nombre de pas que la tortue doit effectuer. Placez la tortue dans sa position initiale en tapant :

```
?home
```

puis faites-la avancer de 40 pas en tapant :

```
?fd 40
```

La tortue s'est déplacée vers le haut de l'écran en laissant une trace. Faites maintenant tourner la tortue à angle droit vers la gauche puis faites-la avancer de 40 pas :

```
?lt 90
?fd 40
```

La tortue à tracé un nouveau trait de même longueur et perpendiculaire au premier.

Les deux instructions auraient pu être placées sur la même ligne. Nous allons le vérifier en terminant le tracé d'un carré. Tapez :

```
?lt 90 fd 40 lt 90 fd 40 lt 90
```

Il est important de noter que les instructions et les paramètres doivent être séparés par un espace. Un carré est maintenant dessiné sur l'écran. Nous avons terminé le tracé par l'instruction **lt 90** pour remettre la tortue dans la position où nous l'avions trouvée.

Il est plus satisfaisant de terminer un carré par un quatrième angle à 90 degrés. Cette notion est très importante. Si nous n'avions pas terminé par un angle à 90 degrés, l'effet aurait été fondamentalement différent. Comparez les Figures 2.3 et 2.4.

Paradoxalement, il y a plus de différence entre la position initiale et la Figure 4 qu'entre la position initiale et la Figure 3 bien qu'il ait fallu une instruction de plus pour la Figure 3. La Figure 3 peut se définir par rapport à la position initiale comme un carré. La

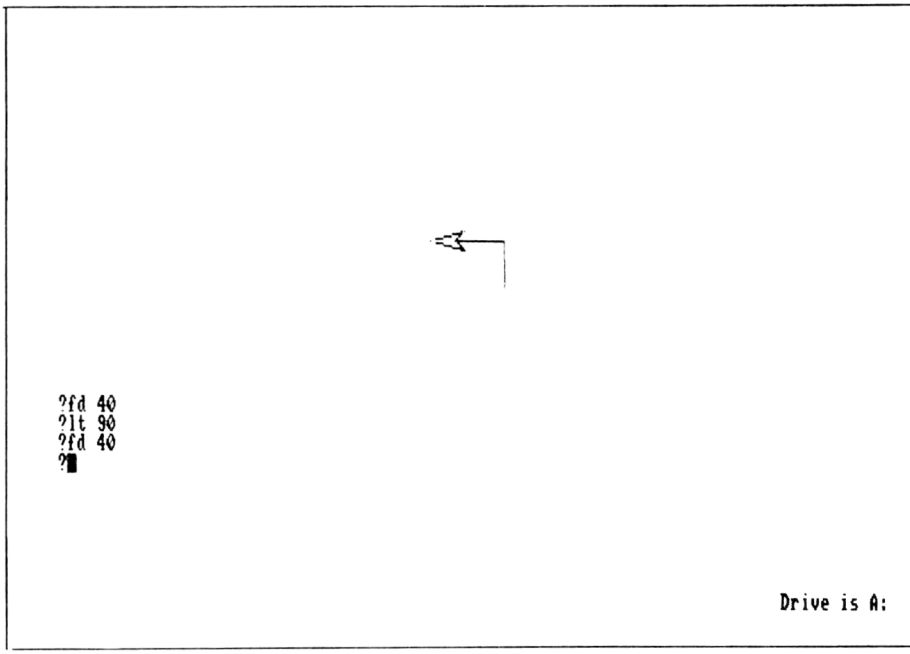


Figure 2.2 : La trace de la tortue.

Figure 4 se définit comme un carré plus une rotation à droite de 90 degrés. Dans le premier cas nous avons effectué quatre fois le couple d'opérations :

```
fd 40 lt 90
```

ce qui correspond bien à la définition d'un carré : quatre côtés égaux et quatre angles droits. Dans le second cas, nous aurions effectué trois fois le couple :

```
fd 40 lt 90
```

et une fois l'opération :

```
fd 40
```

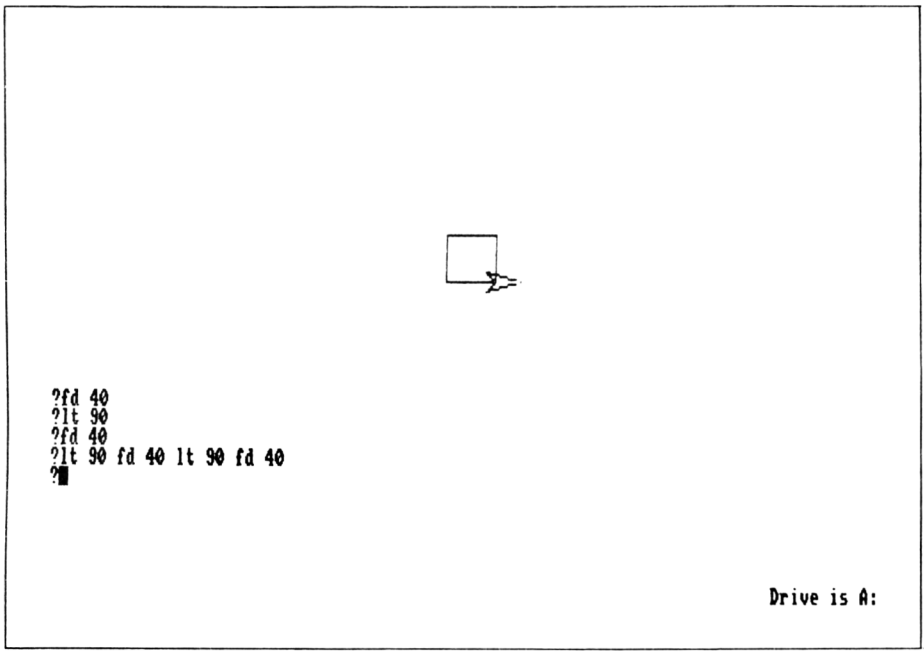


Figure 2.3 : Un carré.

Nous pouvons profiter de cette remarque pour constater qu'il aurait alors été plus simple de demander à la tortue de répéter quatre fois le couple d'instructions :

```
fd 40 lt 90
```

ce qui se fait très simplement en tapant :

```
?cs
?repeat 4 [fd 40 lt 90]
```

Sur l'Amstrad PCW 8256 et 8512, les crochets carrés [et] sont obtenus (à condition que la commande **language 0** ait été utilisée avant de charger Logo) à l'aide des touches "degré" (SHIFT + parenthèse fermante) et "paragraphe" (touche 6).

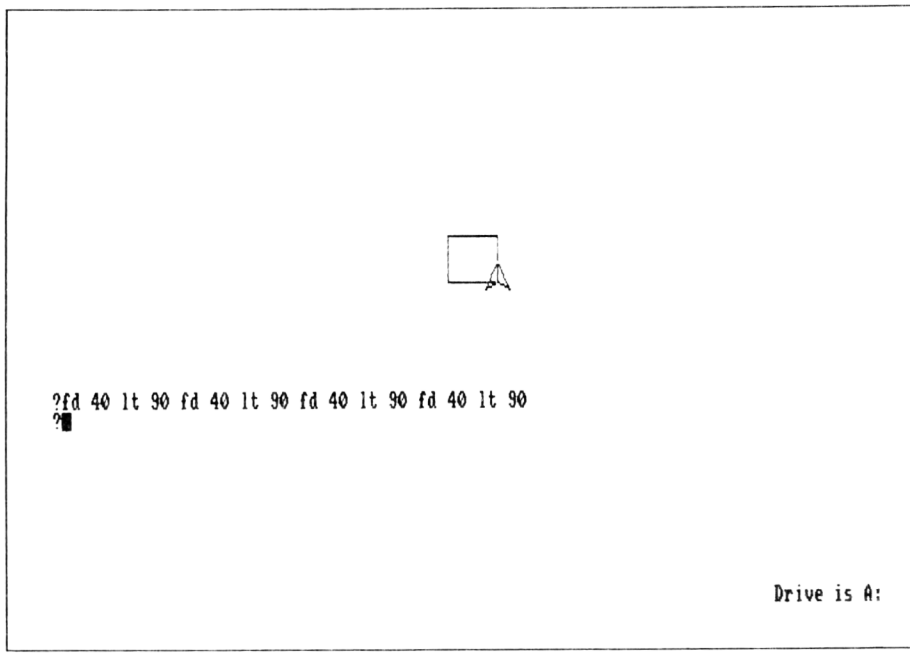


Figure 2.4 : Un carré plus une rotation.

L'instruction cs permet d'abord d'effacer l'écran. La seconde ligne demande à la tortue de répéter **(repeat)** quatre fois la séquence d'instructions placée entre crochets. L'instruction **repeat** prend donc deux paramètres ou arguments : un nombre et une liste. Une liste est pour Logo un objet composé d'autres objets ou éléments et placés entre crochets carrés. Nous verrons plus tard que les éléments d'une liste peuvent être des objets simples ou des listes.

Un nouveau carré identique au premier est maintenant tracé sur l'écran.

Nous pouvons dessiner un autre carré deux fois plus grand en tapant :

```
?cs
?repeat 4 [fd 80 lt 90]
```

On obtient alors le dessin de la Figure 5.

On constate qu'un seul élément de la commande a changé : le nombre 40 (la longueur du côté du carré) a été remplacé par la valeur 80. Il vient immédiatement à l'esprit qu'il serait intéressant de ne pas avoir à retaper toute la commande pour changer la taille du carré. Logo permet de définir une procédure capable de dessiner n'importe quel carré :

```
?to carre :cote
>repeat 4 [fd :cote lt 90]
>end
```

La définition d'une procédure Logo commence par le mot **to** et finis par le mot **end.** Le mot **to** est suivi du nom que l'on veut donner à la procédure, ici **carre**. La procédure **carre**, lorsqu'elle sera utilisée, devra être accompagnée d'un paramètre indiquant la longueur du côté du carré. Cette valeur sera associée à un objet Logo dont le nom doit figurer après le nom de la procédure et précédé du signe deuxpoints (:). Lorsque la première ligne est tapée (terminée par la touche RETURN) Logo affiche le signe > à la ligne suivante. Ce signe indique que Logo attend le corps de la procédure. Celui-ci nous est maintenant familier. La seule différence avec les commandes utilisées précédemment est que la longueur du côté a été remplacé par :cote. A la ligne suivante, nous taperons simplement **end** pour indiquer que la définition de la procédure est terminée. Nous pouvons maintenant tracer un carré en tapant simplement.

```
?carre 50
```

Que se passe-t-il lorsque nous tapons la touche RETURN à la fin de cette commande ? Logo connaît maintenant le mot **carre** que nous avons défini précédemment. Il sait que c'est une procédure. Cette procédure prend un paramètre. Ici, la valeur 50 est affectée à :cote. Logo exécute donc la commande :

```
repeat 4 [fd :cote lt 90]
```

en remplaçant :cote par sa valeur c'estàdire 50. La commande exécutée est donc :

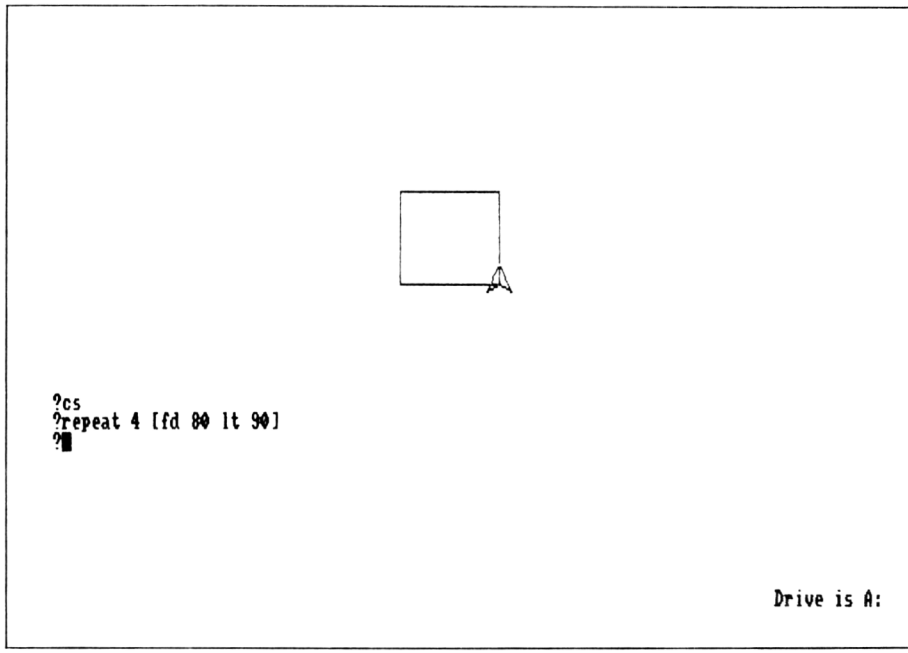


Figure 2.5 : Un carré deux fois plus grand.

```
repeat 4 [fd 40 lt 90]
```

On pourrait souhaiter modifier cette procédure pour qu'elle puisse servir à tracer d'autres figures que des carrés. Il suffirait pour cela de remplacer 4 (le nombre de côtés) par une valeur quelconque que nous appellerons :ncote pour "nombre de côtés") et 90 par 360/nombre de côtés. Ceux pour qui cette formule n'est pas évidente peuvent facilement la vérifier à l'aide d'une figure et éventuellement d'un livre de géométrie. Logo accepte deux notations pour les opérations arithmétiques. La notation traditionnelle appelée "infixée", par exemple :

```
?3 + 4
7
```

et la notation "préfixée" :

```
?+ 3 4
7
```

Nous emploierons le plus souvent la notation préfixée car elle est beaucoup plus cohérente avec le fonctionnement de Logo. En effet, dans l'exemple ci-dessus, + est une procédure fournissant la somme de ses paramètres. Il est donc logique que, comme pour les autres procédures Logo, le nom de la procédure précède les paramètres. Notons que l'espace n'est pas obligatoire entre les signes et les nombres. Ainsi :

```
?+3 4
```

est équivalent à :

```
+ 3 4
```

mais

```
+34
```

donne une erreur. Il manque ici un paramètre car les chiffres 3 et 4 non séparés par un espace sont interprétés comme le nombre 34.

Nous disposons maintenant des éléments nécessaires pour modifier notre procédure. Il existe en Logo une procédure permettant de modifier la définition d'une autre procédure. Tapez :

```
?ed "carre
```

Le texte de la définition de la procédure **carre** s'affiche en haut de l'écran. Vous pouvez maintenant, à l'aide des touches du curseur, la modifier suivant l'exemple suivant :

```
to polygone :ncote :cote
repeat :ncote [fd :cote lt (/ 360 :ncote)]
end
```

Lorsque la modification est terminée, vous pouvez revenir au mode normal (et quitter le mode Edition) en tapant ^C (Maintenez la touche CTRL enfoncée et tapez la touche C. Sur les PCW 8256 et 8512, la touche CTRL est remplacée par la touche ALT.) Dans cet ouvrage, les caractères de contrôle seront toujours notés précédés d'une flèche dirigée vers le haut. Les caractères de contrôle disponibles sont souvent doublés par une touche spéciale ayant la même fonction. En fait, le déplacement du curseur vers la gauche, par exemple, se fait à l'aide de ^B. La flèche dirigée vers la gauche donne le même résultat. L'Annexe 1 donne la liste des caractères de contrôle et des touches spéciales disponibles sur les CPC 464/664, CPC 6128 et PCW 8256 et 8512.

Pour quitter le mode Edition, tapez ^C. Les modifications apportées à la procédure sont enregistrées mais, le nom de la procédure ayant été modifié, la procédure originale **(carre)** existe toujours. Notez que dans cette nouvelle procédure, nous avons mis la liste (/ 360 :ncote) entre parenthèses. Ces parenthèses ne sont pas obligatoires mais facilitent la lecture. En effet, cette liste constitue un objet composé qui est l'argument de la procédure **lt**. Nous utiliserons des parenthèses chaque fois que cela pourra clarifier une expression. (Si vous désirez quitter le mode Edition sans enregistrer les modifications effectuées, vous pouvez tout simplement interrompre la procédure **ed** en tapant ^G.) Vous pouvez essayer la nouvelle procédure en tapant par exemple :

```
?cs
?polygone 11 50
```

Vous voyez maintenant se dessiner sur l'écran un polygone régulier à onze côtés longs de cinquante pas de tortue. Si on augmente le nombre de côtés, on finira par obtenir approximativement un cercle. Essayez par exemple :

```
?cs
?polygone 40 10
```

Il est possible d'accélérer le tracé en utilisant la commande **ht** qui supprime l'affichage de la tortue. Essayez l'exemple suivant et comparez les vitesses d'exécution :

```
?cs ht polygone 40 10
```

Si on augmente la longueur des côtés du polygone, un problème se pose. Essayez la commande suivante :

```
?cs polygone 40 40
```

Un grand cercle est dessiné sur l'écran, si grand qu'il déborde largement en haut, à gauche et en bas. En fait, l'écran peut être placé dans trois modes différents. Jusqu'ici, nous avons travaillé en mode Window, dans lequel l'écran est une fenêtre sur un écran virtuel beaucoup plus large. Si la tortue sort de l'écran, elle continue son chemin hors de notre vue. Nous pouvons par contre sélectionner le mode Wrap dans lequel lorsque la tortue sort d'un côté de l'écran, elle réapparaît de l'autre côté. Essayez la commande suivante :

```
?cs wrap polygone 40 40
```

Vous pouvez constater que la tortue commence son tracé vers le haut, puis, lorsqu'elle atteint le haut de l'écran, elle réapparaît en bas, puis disparaît de nouveau en bas pour réapparaître en haut et ainsi de suite. Le troisième mode disponible est le mode Fence. Essayez la commande :

```
?cs fence polygone 40 40
```

La tortue commence son tracé. Lorsqu'elle atteint le haut de l'écran, un message d'erreur s'affiche indiquant que les limites de l'écran ont été dépassées. En mode Fence, la tortue ne doit pas sortir des limites de l'écran. Notez que le message d'erreur indique le nom de la procédure (**polygone**) et la ligne à laquelle s'est produite l'erreur. De plus, si après une erreur vous tapez la commande **ed** sans argument, le mode Edition est activé avec la procédure en cause, le curseur se trouvant sur l'instruction qui a causé l'erreur.

Pour repasser en mode Window, tapez la commande :

```
?window
```

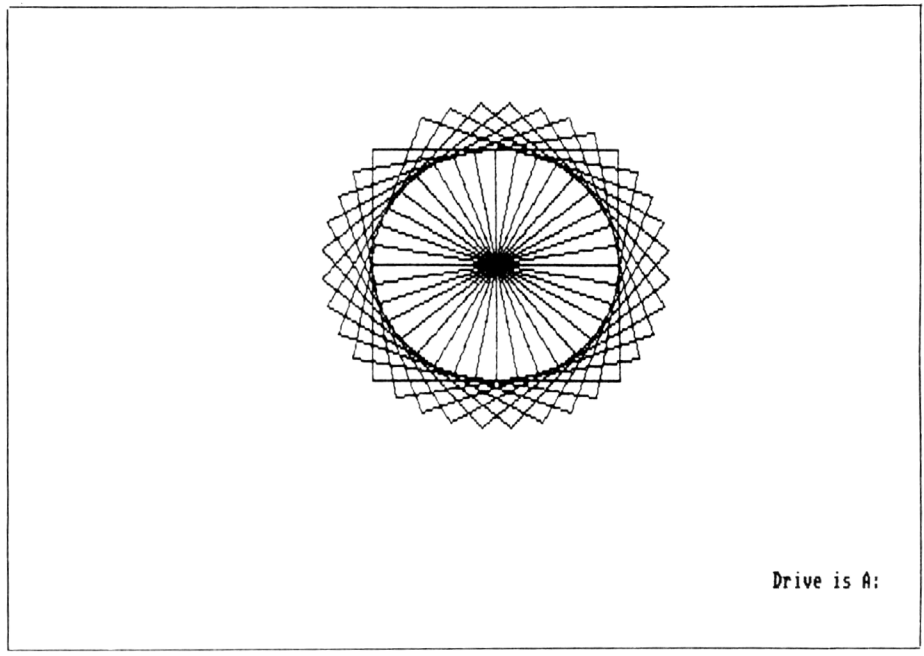


Figure 2.6 : Premier exemple.

Il est d'usage dans les manuels Logo d'"émerveiller" le lecteur avec une multitude de dessins tous plus artistiques les uns que les autres. Nous en donnerons pour notre part deux exemples simples, laissant au lecteur le soin d'expérimenter à loisir dans ce domaine qui est certainement le moins intéressant de ce langage.

Le premier exemple est une utilisation de la procédure **carre** répétée un certain nombre de fois avec un décalage de quelques degrés :

```
?fs cs ht repeat 36 [carre 100 lt 10]
```

Lorsque le dessin est terminé, l'écran est en mode graphique. Il suffit de taper :

```
ss
```

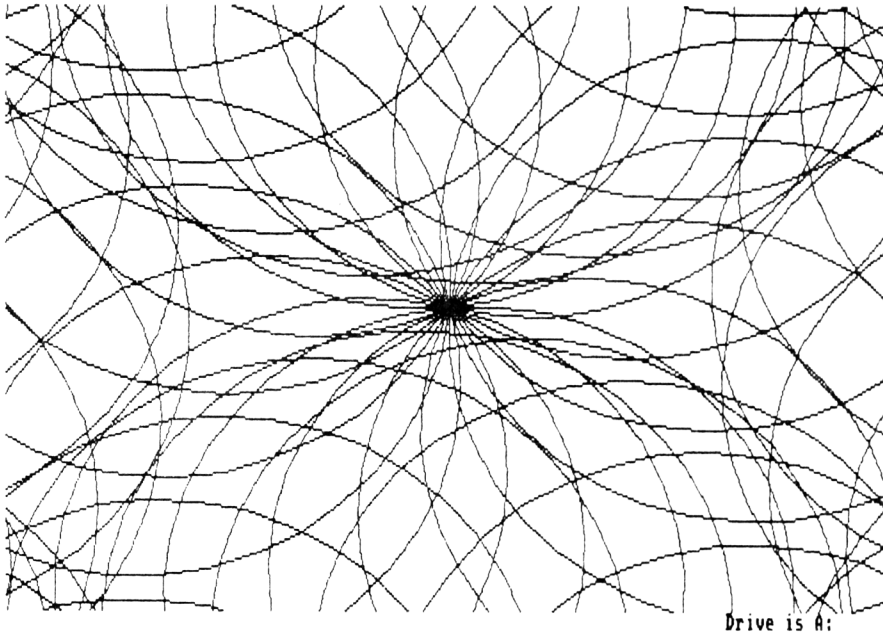


Figure 2.7 : Deuxième exemple.

pour revenir en mode mixte.

Le second exemple utilise la procédure **polygone** :

```
?fs cs ht wrap repeat 18 [polygone 40 40 lt 20]
```

De nombreuses autres procédures graphiques sont à la disposition du programmeur. Nous en donnons la liste complète au Chapitre 8.

CHAPITRE III

FONCTIONNEMENT DE LOGO

Nous avons jusqu'ici appris à écrire quelques procédures sans vraiment comprendre le fonctionnement interne de Logo. Chaque fois que l'on soumet à Logo un objet simple (un mot) ou un objet composé (une liste), nous obtenons une réaction. Certains mots déclenchent une action telle que déplacer la tortue, effacer l'écran, etc. D'autres mots déclenchent l'apparition d'un message d'erreur. D'autres provoquent le réaffichage du mot lui-même. Pour comprendre les raisons de ces différences, il faut essayer de comprendre le fonctionnement de Logo.

Logo manipule des objets : symboles (mots) ou listes. On peut répartir les objets en deux catégories : ceux qui sont connus de Logo et ceux qui lui sont inconnus. Nous employons ici le mot Logo pour désigner un certain interpréteur Logo à un moment déterminé. Pour illustrer cette précision, nous allons tout d'abord consulter la liste des objets connus de Logo. Chargez Logo à partir de CP/M. (Si Logo est déjà chargé, quittez-le en tapant bye puis rechargez-le. Ceci est très important pour la suite des opérations.) Tapez maintenant :

```
?.contents
```

Nous voyons s'afficher sur l'écran une liste contenant tous les objets connus de Logo. Tapez maintenant :

```
?napoleon
```

Logo répond :

```
I don't know to napoleon
```

Nous avons déjà rencontré ce message qui semble signifier que Logo ne connaît pas le mot napoleon. Consultons à nouveau la liste des objets connus de Logo en tapant :

```
?.contents
```

Nous voyons maintenant figurer dans cette liste le mot napoleon. Logo connaît donc cet objet. Pourtant, si nous tapons à nouveau :

```
?napoleon
```

nous obtenons le même message d'erreur. Ce message ne signifie donc pas que Logo ne connait pas ce mot. En fait, lorsque nous soumettons à Logo un objet, Logo cherche à exécuter la procédure associée à cet objet. Le message d'erreur que nous venons de rencontrer signifie donc qu'aucune procédure n'est associée à cet objet.

Les procédures font partie des *propriétés* pouvant être associées à un objet. Un objet peut avoir d'autres propriétés. Un objet simple (par opposition à une liste) est défini par son nom et la liste de ses propriétés. Si l'on soumet à Logo un objet "quoté", c'estàdire précédé d'un guillemet, Logo renvoie le nom de l'objet. Si nous tapons par exemple :

```
?"a
```

Logo affiche :

```
a
```

c'est-à-dire le nom de l'objet qui lui a été soumis. Nous avons vu que si un objet est soumis à Logo non "quoté", Logo essaie d'exécuter la procédure qui lui est associée. Une autre propriété peut être associée à un objet : sa valeur. La plupart des objets connus de Logo n'ont pas de valeur associée. Nous pouvons associer une valeur à un objet à l'aide de la procédure **make**. Nous allons associer à l'objet *a* la valeur *b* en tapant :

```
?make "a "b
```

Pourquoi "a et "b et non *a* et *b* ? Simplement parce que nous associons au *nom* de l'objet la *valeur* de l'objet. Si nous avions écrit :

```
?make a b
```

Logo aurait d'abord essayé d'exécuter les procédures *b* et *a* avant d'appliquer la procédure **make** au résultat obtenu. On dit que Logo

évalue d'abord *b* et *a* avant d'évaluer **make**. Aucune procédure n'étant associée aux noms *b* et *a*, Logo aurait affiché un message d'erreur. En fait, le caractère " peut être considéré de deux façons : soit une procédure qui s'évalue comme le nom de son argument, soit un caractère spécial qui empêche l'évaluation de l'objet qui le suit.

Prenons par exemple la procédure **cs** qui efface l'écran texte. Si nous tapons :

```
?cs
```

l'écran texte est effacé. L'objet **cs** à été évalué. L'évaluation ne donne aucun résultat mais une conséquence de l'évaluation est l'effacement de l'écran. (Nous reviendrons plus loin sur cette distinction.)

Si maintenant nous tapons :

```
?"cs
```

Logo affiche :

```
cs
```

Nous pouvons considérer deux interprétations : Logo a évalué l'objet " et le résultat est le nom de l'argument qui le suit ou bien le caractère " a empêché l'évaluation de l'objet **cs** et Logo en a affiché le nom.

Dans l'exemple :

```
?make "a "b
```

Logo affecte pour valeur au nom *a* le nom *b*. Nous pouvons le vérifier en tapant :

```
?:a
```

ce qui entraîne l'affichage de :

```
b
```

Le caractère : est utilisé pour désigner la valeur d'un objet. Nous pouvons vérifier que *b* n'a pas été modifié par cette opération en tapant :

```
?:b
```

ce qui produit l'affichage du message :

```
b has no value
```

qui signifie que *b* n'a pas de valeur.

Nous pouvons résumer les choses de la façon suivante :

- :a désigne la valeur associée à *a*
- "a désigne le nom de *a*
- a désigne la procédure associée à *a*.

la valeur et la procédure associées à un objet sont ses propriétés. Logo offre un moyen de savoir quels sont les objets qui ont une valeur. Tapez :

```
?glist ".APV
```

Logo affiche :

```
[a]
```

Si nous avions tapé cette commande avant d'affecter une valeur à *a*, la réponse aurait été :

```
[]
```

c'est-à-dire la liste vide indiquant qu'aucun objet n'avait de valeur. .APV désigne la propriété "valeur". La propriété "procédure" est dé-

signée de la même façon par .DEF. Nous pouvons donc taper la commande :

```
?glist ".DEF
```

ce qui produit l'affichage de :

```
[]
```

c'est-à-dire la liste vide puisque nous n'avons défini aucune procédure. Nous allons définir la procédure suivante :

```
?to a
>pr "c
>end
```

La procédure **pr** affiche simplement son argument sur l'écran. Ici, la procédure *a* affichera le nom de *c*, c'est-à-dire la lettre *c*. Si nous demandons maintenant la liste des objets ayant la propriété d'être associés à une procédure en tapant :

```
?glist ".DEF
```

nous obtenons l'affichage de :

```
[a]
```

Nous pouvons également demander la liste des propriétés de *a* en utilisant la commande :

```
?plist "a
```

Nous obtenons le résultat suivant :

```
[.DEF [[] [pr "c]] .APV b]
```

Ce résultat nous indique que l'objet *a* a deux propriétés. Il a pour propriété .APV (valeur) la valeur *b* et pour propriété .DEF (procédure associée) la liste [[] [pr "c]]. Cette liste est composée de deux éléments dont le premier est la liste vide désignée par [] et le

second une liste composée de deux éléments dont le premier est **pr** et le second "c.

La procédure **gprop** permet d'afficher une propriété d'un objet :

```
?gprop "a ".APV
```

affiche

```
b
```

qui est la valeur (propriété .APV) de *a*. De même, la commande :

```
?gprop "a ".DEF
```

affiche :

```
[[] [pr "c]
```

qui correspond à la définition de la procédure associée à *a*. La liste vide figurant comme premier élément de cette définition est la liste des variables utilisées dans la procédure. Ici, la procédure *a* n'utilise pas de variable. Nous pouvons définir une procédure que nous appellerons double et qui fournira le double de son argument :

```
?to double :a
>pr (+ :a :a)
>end
```

Il faut noter que les parenthèses ne sont pas obligatoires mais sont utilisées pour clarifier l'écriture. Lorsque cette procédure est utilisée, :a prend pour valeur l'argument de la procédure et le résultat de l'opération + :a :a est affiché.

Si nous demandons maintenant l'affichage de la propriété .DEF de l'objet double en tapant :

```
?gprop "double ".DEF
```

nous obtenons :

```
[[a] [pr (+ :a :a)]]
```

indiquant que la procédure utilise la variable locale *a* et que sa définition est **pr** (+ :a :a).

Il faut noter que l'utilisation de la variable *a* est purement locale et n'affecte pas la valeur de *a* en dehors de la procédure. Nous pouvons le constater en modifiant la procédure double de la façon suivante :

```
to double :a
pr :a
pr (+ :a :a)
end
```

Si nous utilisons cette procédure avec pour argument 2, nous obtenons l'affichage suivant :

```
?double 2
2
4
?
```

qui nous prouve que pendant l'exécution de la procédure, *a* avait bien pour valeur 2. Mais si nous tapons maintenant :

```
?:a
```

nous constatons que la valeur de *a* n'a pas été modifiée et est toujours *b*. En fait, pendant l'exécution de la procédure, la valeur de *a* a été stockée en mémoire, puis restaurée une fois la procédure terminée.

Il existe en Logo une troisième propriété. Appelée .PRM, elle indique l'adresse en mémoire des primitives du langage. (Les primitives sont les procédures prédéfinies de Logo.)Si nous tapons

```
?glist ".PRM
```

nous obtenons la liste des primitives de Logo. Cette liste est semblable à celle obtenue avec la procédure **.contents** juste après le chargement de Logo, c'estàdire avant que de nouveaux objets n'y soient rajoutés. Logo ne connaît à ce moment que ses primitives. Si nous tapons :

```
?gprop "rt ".PRM
```

nous apprenons que la primitive **rt** commence à l'adresse 7258 (cette valeur est obtenue sur un PCW 8256 et peut être différente sur un autre modèle).

Nous avons précédemment affecté à *a* la valeur *b* en tapant :

```
?make "a "b
```

Si maintenant nous affectons à *b* la valeur *d* à l'aide de la commande :

```
?make "b "d
```

nous obtiendrons les résultats suivants :

```
?"a
a
?:a
b
?"b
b
:b
d
```

Tout cela est très logique. Contrairement à ce que l'on aurait pu penser, si *a* vaut *b* et si *b* vaut *d*, *a* ne vaut pas *d* pour autant. En fait la valeur de *a* est le nom de *b*. la valeur de *b* est le nom de *d*. Pour obtenir un effet de transitivité, il faut écrire :

```
?make "a :b
```

ce qui signifie "donner pour valeur à *a* la valeur de *b*" (ce qui est très différent de "donner pour valeur à *a* le nom de *b*"). On peut vérifier la différence en tapant :

```
?:a
d
```

Il est très important d'avoir bien compris la différence entre le nom d'un objet, sa valeur et la procédure qui lui est associée. Il est nécessaire d'expérimenter ces notions à l'aide d'exemples personnels avant de passer aux chapitres suivants.

CHAPITRE IV

LES PROCEDURES DE MANIPULATION DE LISTES

Nous avons vu dans les chapitres précédents que lorsqu'un objet est soumis à Logo, celui-ci tente d'exécuter la procédure qui lui est associée. Pour éviter cela, on peut faire précéder le nom de l'objet du caractère ". Dans ce cas, Logo affiche simplement le nom de l'objet. Si par contre on fait précéder le nom de l'objet du caractère deux points (:), Logo affiche sa valeur.

Logo manipule deux sortes d'objets : les objets simples ou mots, et les objets composés ou listes. Les listes sont composées d'objets simples ou composés : une liste peut avoir pour éléments d'autres listes. Les listes sont délimitées par des crochets carrés [et]. Lorsqu'une liste est soumise à Logo, elle est simplement réaffichée. Si nous tapons par exemple :

```
?[pomme orange poire]
```

Logo affiche

```
[pomme orange poire]
```

Les listes peuvent servir à de nombreuses applications. La liste ci-dessous pourrait servir à classer des renseignements sur une personne dans une base de données :

```
[[taille 178] [poids 72] [yeux marron] [cheveux bruns]]
```

En Logo, un certain nombre de procédures sont prédéfinies. On les appelle des primitives. Plusieurs de ces primitives servent à la manipulation des listes. Les opérations que l'on peut être amené à pratiquer sur des listes sont multiples. Les plus importantes sont l'extraction d'un élément d'une liste et l'ajout d'un élément à une liste. Cinq primitives sont particulièrement importantes pour la manipulation des listes, car toutes les autres opérations peuvent être réalisées à l'aide d'une combinaison de ces cinq-là.

Extraction du premier élément d'une liste

La procédure permettant d'extraire le premier élément d'une liste est **first**. Pour extraire le premier élément de la liste :

```
[pomme orange poire]
```

il suffit de taper :

```
?first [pomme orange poire]
```

Logo répond en affichant :

```
pomme
```

Obtention d'une liste privée de son premier élément

La procédure **bf** fournit la liste moins le premier élément. Tapez :

```
?bf [pomme orange poire]
[orange poire]
```

Il est important de noter que la procédure **first** donne un mot alors que le résultat de la procédure **bf** est une liste. Il est possible de combiner ces deux procédures pour obtenir le deuxième élément d'une liste. Exemple :

```
?first (bf [pomme orange poire])
orange
```

Notez que l'on aurait tout aussi bien pu écrire :

```
?first bf [pomme orange poire]
```

sans parenthèses. L'emploi des parenthèses facilite la lecture en indiquant au premier coup d'oeil que la procédure **first** prend un seul argument et qu'elle est appliquée au résultat de la procédure **bf**

qui prend elle aussi un seul argument et est appliquée à la liste [pomme orange poire]. L'absence de parenthèses pourrait faire croire au premier abord que la procédure **first** prend deux arguments : le mot **bf** et la liste [pomme orange poire].

On peut de la même façon extraire le troisième élément d'une liste :

```
?first (bf (bf [pomme orange poire]))
poire
```

Si l'on considère maintenant le second exemple de liste donné précédemment :

```
[[taille 178] [poids 72] [yeux marron] [cheveux bruns]]
```

et si l'on veut isoler le poids, nous devons considérer qu'il s'agit du second élément du second élément de la liste. En effet, le deuxième élément de la liste est la liste [poids 72]. 72 est le second élément de ce deuxième élément. Pour isoler le poids, nous devons donc taper :

```
?first (bf (first (bf [[taille 178] [poids 72] [yeux marron] [cheveux
                                                        bruns]])))
72
```

Si nous essayons d'appliquer la procédure **bf** à la liste [pomme] qui ne comporte qu'un seul élément, nous obtenons :

```
?bf [pomme]
[]
```

[] désigne la liste vide. En effet, le résultat de la procédure **bf** ne peut être qu'une liste, même si celle-ci ne comporte aucun élément. Nous verrons plus tard que la liste vide a une importance capitale pour la programmation en Logo.

Les deux procédures précédentes permettent d'extraire un élément ou une sous-liste d'une liste. L'inverse est tout aussi important.

Construction d'une liste

La procédure **fput** permet de rajouter un élément en tête d'une liste. Si nous voulons placer le mot raisin en tête de la liste [pomme orange poire], nous pouvons taper :

```
?fput "raisin [pomme orange poire]
[raisin pomme orange poire]
```

Le guillemet est indispensable devant le mot raisin pour empêcher son évaluation. S'il est omis, Logo répond :

```
I don't know to raisin
```

indiquant qu'il a recherché une procédure associée au mot raisin pour l'exécuter mais ne l'a pas trouvée.

Pour mettre en application les éléments étudiés jusqu'ici, essayez à titre d'exercice d'obtenir la liste [a y z] à partir des listes [a b c] et [x y z].

Si vous ne parvenez pas à trouver la solution, vérifiez que celle donnée ci-dessous fonctionne et essayez d'en comprendre le mécanisme :

```
?fput (first [a b c]) (bf [x y z])
[a y z]
```

On voit facilement que la liste [a y z] est obtenue par l'ajout du premier élément de la liste [a b c] à la liste [x y z] privée de son premier élément.

Note : il est possible d'appliquer les procédures **first** et **bf** à des mots. Les mots sont alors traités comme des listes de caractères. **first** donne le premier caractère du mot et **bf** donne le mot privé de son premier caractère.

wordp

La procédure **wordp** donne le résultat TRUE (vrai) si son argument est un mot. Elle donne le résultat FALSE (faux) dans le cas contraire. Exemples :

```
?wordp "pomme
TRUE
?wordp [pomme orange poire]
FALSE
?wordp (first [pomme orange poire])
TRUE
?wordp (bf [pomme orange poire])
FALSE
```

La procédure =

La procédure = prend deux arguments. Elle donne la valeur TRUE si ses deux arguments sont égaux et la valeur FALSE dans le cas contraire. Exemples :

```
?= "pomme "pomme
TRUE
?= "pomme "poire
FALSE
?= [pomme poire] [pomme poire]
TRUE
?= [pomme poire] [pomme orange]
FALSE
?= (first [pomme poire]) "pomme
TRUE
```

A titre d'exercice, essayez de vérifier si la personne décrite par la liste :

```
[[taille 178] [poids 72] [yeux marron] [cheveux bruns]]
```

les yeux marron.

La solution est obtenue facilement en tapant :

```
?= (first (bf (first (bf (bf [[taille 178] [poids 72] [yeux marron]
                                        [cheveux bruns]]))))) "marron
TRUE
```

Définition de nouvelles procédures

Une procédure définie par l'utilisateur a la même syntaxe qu'une primitive, c'est-à-dire :

```
procédure argument1 argument2 ... argumentn
```

Une procédure est définie avec des arguments formels. Elle est appelée avec des arguments réels. Nous avons déjà mis cette caractéristique en pratique en définissant la procédure **carre** :

```
to carre :cote
repeat 4 [fd :cote lt 90]
end
```

Dans cette définition, :cote est un argument formel. Lorsque la procédure est appelée avec pour argument 50 :

```
?carre 50
```

50 est l'argument réel. Avant d'exécuter une procédure, Logo remplace les arguments formels par les arguments réels. Le nombre d'arguments réels doit donc correspondre au nombre d'arguments formels. Dans l'exemple précédent, la procédure réellement exécutée est donc :

```
repeat 4 [fd 50 lt 90]
```

Nous pouvons donc récapituler en disant qu'une procédure se définit de la façon suivante :

```
To nom :arg1 :arg2 ... :argn
action1
action2
 ....
actionn
end
```

:arg1, :arg2 ... :argn sont les arguments formels. action1, action2 ... actionn constituent le corps de la procédure. On peut définir de façon très simple une procédure réalisant la même chose que la procédure **first**, mais sous un autre nom :

```
?to premier :liste
>op (first :liste)
>end
?premier [pomme orange poire]
pomme
?
```

Notez l'utilisation de la procédure **op** pour afficher le résultat de la procédure **first**. En effet, le but de la procédure **first** n'est pas d'afficher son résultat mais de le fournir éventuellement à une autre procédure. Par exception, lorsqu'une primitive est utilisée seule, ou quand son résultat ne peut être fourni à aucune autre procédure, celui-ci est affiché. C'est le cas pour la première d'une série de primitives. Imaginez l'effet qui serait obtenu si le résultat de chacune des procédures était affiché dans une expression comme :

```
?= (first (bf (first (bf (bf [[taille 178] [poids 72] [yeux marron]
                                                [cheveux bruns]]))))))
"marrons
```

Heureusement, seul le résultat de la première procédure (c'est-à-dire le résultat final) est affiché. Cependant, dans une procédure définie par l'utilisateur, le résultat n'est pas affiché implicitement. Dans la procédure **premier**, si l'on avait omis la primitive **op**, Logo aurait répondu :

```
I don't know what to do with pomme in premier: first :liste
```

indiquant que le résultat (pomme) a bien été trouvé mais que Logo ne sait pas quoi en faire. Grâce à l'emploi de la procédure **op**, le résultat est soit fourni à la procédure qui précède, soit affiché s'il s'agit du résultat final.

Il est ainsi possible de redéfinir toutes les primitives de Logo en français (au prix d'une diminution de la vitesse d'exécution).

La procédure nulle

Nous allons maintenant définir une procédure qui donne la valeur TRUE si son argument est la liste vide et FALSE dans le cas contraire. Logo possède la primitive **emptyp** qui fournit à peu près ce résultat. Nous nous appliquerons pour le moment à n'utiliser que le strict nécessaire parmi les primitives Logo. La définition de la procédure **nulle** est très simple :

```
?to nulle :liste
>op = :liste []
>end
```

Nous pouvons facilement vérifier son fonctionnement :

```
?nulle []
TRUE
?nulle "pomme
FALSE
?nulle [pomme poire]
FALSE
?nulle (bf [pomme poire])
FALSE
?nulle (bf (bf [pomme poire]))
TRUE
```

Structures de décision

Nous allons maintenant essayer de définir une procédure **liste** qui devra fournir la valeur TRUE si son argument est une liste et FALSE dans le cas contraire. La définition est la suivante :

```
?to liste :a
>if wordp :a [op "FALSE] [op "TRUE]
>end
```

Cette définition utilise la structure de décision **if** dont la syntaxe est la suivante :

```
if condition [séquence1] [séquence2]
```

Si la condition est remplie, la séquence [séquence1] est exécutée. Dans le cas contraire, la séquence [séquence2] est exécutée. Ici séquence1 fournit la valeur FALSE et séquence2 fournit la valeur TRUE, c'estàdire l'inverse de la condition. Nous aurions pu également définir une procédure **non** inversant la valeur logique de son argument et l'appliquer à la procédure **wordp**. La procédure **non** se définirait ainsi :

```
?to non :a
>if = :a "TRUE [op "FALSE] [op "TRUE]
>end
```

Nous pouvons la tester en tapant :

```
?non "TRUE
FALSE
?non "FALSE
TRUE
?non wordp "a
FALSE
?non liste "a
TRUE
```

La procédure **liste** pourrait maintenant être réécrite sous la forme :

```
?to liste2 :a
>op non wordp :a
>end
```

Cette procédure **non** nous sera très utile pour inverser des valeurs logiques. (Cette procédure existe évidemment sous forme de primitive et s'appelle **not**. Nous répétons cependant que, dans un but pédagogique, il est excellent de se contenter des seules primitives indispensables.)

La procédure element

Définissons maintenant une procédure permettant de déterminer si un objet est élément d'une liste. Cette procédure se définit simplement de la manière suivante :

```
?to element :a :liste
>if nulle :liste [op "FALSE]
>if = :a (first :liste) [op "TRUE]
>if element :a (bf :liste) [op "TRUE] [op "FALSE]
>end
```

Nous pouvons vérifier son fonctionnement en tapant :

```
?element "poire [pomme poire orange]
TRUE
?element "citron [pomme poire orange]
FALSE
?element [citron vert] [pomme poire [citron vert] orange]
TRUE
```

Il est intéressant d'étudier de plus près le fonctionnement de cette procédure, car elle présente pour la première fois un des aspects essentiels de Logo, la récursivité. La définition de cette procédure est récursive car elle utilise la procédure en cours de définition. Détaillons son fonctionnement.

Si le paramètre :liste est la liste vide, la procédure donne la valeur FALSE (l'objet :a ne peut évidemment pas être élément d'une liste vide). Si le premier élément de la liste est égal à :a, :a est élément de la liste. Si aucune de ces conditions n'est remplie, il ne reste plus qu'à regarder si :a est élément du reste de la liste débarrassée de son premier élément. C'est ce que fait la troisième ligne du corps de la procédure. On se rend compte ici de la considérable simplification apportée à l'écriture de ce type de programme par la récursivité. La même procédure écrite de façon itérative ressemblerait à :

```
  compter le nombre d'éléments dans la liste
  donner au compteur d'éléments I la valeur 1
1 si :a est égal au Ième élément, donner la valeur TRUE et arrêter
  ajouter 1 à I
  si I <= nombre d'éléments, recommencer en 1
  sinon donner la valeur FALSE et arrêter
```

Il faut cependant noter que la définition itérative ne donne aucun avantage en vitesse d'exécution. Cette méthode est même souvent plus lente, surtout lorsqu'elle est mal utilisée.

La procédure concat

Si on applique la procédure **fput** aux listes [a b c] et [d e f], on obtient la liste [[a b c] d e f]. Nous allons maintenant définir la procédure **concat** qui, appliquée à ces deux listes, donnera la liste [a b c d e f]. Voici la définition de cette procédure :

```
?to concat :liste1 :liste2
>if nulle :liste1 [op fput (first :liste1) concat (bf :liste1)
                                                    :liste2]
>end
```

Cette fois encore, nous avons utilisé la récursivité. Nous pouvons tester cette procédure en tapant :

```
?concat [a b c] [d e f]
[a b c d e f]
```

La procédure retire

La procédure **retire** aura pour but de supprimer la première occurrence d'un élément dans une liste. Voici sa définition

```
to retire :a :liste
if nulle :liste [op :liste]
if = :a (first :liste) [op bf :liste]
op fput (first :liste) retire :a (bf :liste)
end
```

La première ligne du corps de la procédure (deuxième ligne de la définition) teste si la liste est vide. Si c'est le cas, la procédure fournit :liste comme valeur, c'est-à-dire la liste vide (on ne peut pas retirer un élément d'une liste vide). A la deuxième ligne, si le premier élément de la liste est égal à l'élément à retirer, la procédure fournit comme valeur la liste débarrassée de son premier élément (bf :liste). Dans le cas contraire, la troisième ligne enclenche le processus de récursivité. La valeur fournie est la liste formée par le premier élément de :liste et l'application de la procédure à :liste sans son premier élément. Il est important de bien comprendre ce processus qui est le fondement des langages récursifs tels que Logo. Testons cette procédure :

```
?retire "c [a b c d e]
[a b d e]
?retire "c [a c b c d e]
[a b c d e]
```

Nous constatons que la procédure ne retire bien que la première occurrence de l'élément dans la liste.

La procédure retiretous

Nous allons modifier la procédure **retire** pour créer la procédure **retiretous** qui retirera toutes les occurrences d'un élément dans une liste.

```
to retiretous :a :liste
if nulle :liste [op :liste]
if = :a (first :liste) [op retiretous :a (bf :liste)]
op fput (first :liste) retiretous :a (bf :liste)
end
```

Nous pouvons tester cette nouvelle version :

```
?retiretous "a [a b a c a d a e a f]
[b c d e f]
```

La procédure renverse

La procédure **renverse** doit permettre de renverser l'ordre d'une liste, c'est-à-dire transformer [a b c d] en [d c b a]. Voici sa définition :

```
to renverse :liste
if nulle :liste [op :liste]
op concat (renverse (bf :liste)) (fput (first :liste) [])
end
```

Nous vérifions facilement que son fonctionnement est correct :

```
?renverse [a b c d e]
[e d c b a]
```

Les procédures logiques

Nous avons déjà défini la procédure **non** donnant la négation logique. Nous aurons besoin pour les exemples suivants des procédures **et** et **ou**. Leur définition est extrêmement simple :

```
to et :a :b
if = "FALSE :a [op "FALSE]
```

```
if = "FALSE :b [op "FALSE]
op "TRUE
end

to ou :a :b
if = "TRUE :a [op "TRUE]
if = "TRUE :b [op "TRUE]
op "FALSE
end
```

Si les définitions de ces deux procédures sont extrêmement simples et se passent d'explications, il est cependant intéressant de noter l'extrême similitude entre la définition de **et** et de **ou**. Il suffit en fait de remplacer TRUE par FALSE et FALSE par TRUE pour passer de l'une à l'autre.

Nous avons vu dans ce chapitre comment nous pouvons, à l'aide des cinq primitives **first**, **bf**, **fput**, **=** et **,wordp**, définir de nombreuses procédures de manipulation de listes. Nous avons eu besoin également de la procédure **op** bien que celle-ci n'apporte absolument rien à la structure des procédures définies. Avant de continuer à construire des procédures de plus en plus puissantes (en utilisant les procédures définies dans ce chapitre), vous devez vous entraîner à manipuler les notions acquises. Essayez de redéfinir toutes les procédures étudiées, mais cette fois sans l'aide du livre. Lorsque vous maîtriserez parfaitement ces notions, faites les exercices suivants :

- Définissez la procédure **renversetout** qui, à partir de la liste [a [b c] [d e] f] donne la liste [f [e d] [c b] a].
- Définissez la procédure **union** qui réalise l'union de deux listes (l'union de [a b c d e f] et de [b d f g h] donne la liste [a b c d e f g h]).
- Définissez la procédure **inter** qui réalise l'intersection de deux listes (l'intersection des deux listes précédentes donne [b d f])
- Définissez la procédure **remplace :a :b :liste** qui remplace dans :liste tous les éléments :a par les éléments :b.

Sauvegarde sur disque

Lorsque vous aurez défini toutes ces procédures, vous voudrez les sauvegarder sur disque ainsi que celles que nous avons définies pour pouvoir les réutiliser plus tard. Vous pouvez sauvegarder la totalité de votre travail en utilisant la procédure **save**. Tapez :

```
?save "travail
```

La totalité de l'espace de travail de Logo est sauvegardée sur le disque dans le fichier "travail" (vous pouvez bien sûr donner n'importe quel nom à ce fichier. Lorsque vous rechargerez Logo lors de votre prochaine séance, il vous suffira de taper :

```
?load "travail
```

pour retrouver exactement les mêmes conditions qu'au moment de la sauvegarde.

CHAPITRE V

LES OBJETS LOGO

Nous avons vu que la procédure **.contents** affiche la liste des objets connus de Logo. Pour effectuer les manipulations décrites dans ce chapitre, vous devez faire "oublier" à Logo tous les nouveaux objets que vous lui avez fait connaître au chapitre précédent. Si Logo se trouve en mémoire, quittezle en tapant :

```
?bye
```

puis rechargez-le. (Sauvegardez d'abord votre travail car nous utiliserons les procédures que nous avons définies dans un prochain chapitre.)

Juste après le chargement de Logo, la procédure **.contents** n'affiche que les objets prédéfinis, que l'on appelle des primitives. Si nous utilisons un nouvel objet, il devient automatiquement connu de Logo et figurera ultérieurement dans la liste affichée par **.contents.** Il sera effacé de cette liste lorsque l'on quittera Logo.

Chaque objet simple ou mot (par opposition aux objets composés ou listes) est caractérisé au minimum par son nom. Il peut en outre être caractérisé par une ou plusieurs propriétés :

- une valeur
- la définition d'une procédure
- le fait qu'il soit ou non une primitive

Logo utilise pour nommer les propriétés les termes suivants :

- .APV (valeur)
- .DEF (définition d'une procédure)
- .PRM (primitive)

La procédure glist

La procédure **glist** est une primitive qui prend pour argument le nom d'une propriété. Exemple :

```
?glist ".APV
[]
```

Cette procédure donne la liste de tous les objets qui ont reçu une valeur. Pour l'instant, aucun objet n'a reçu de valeur. La procédure donne donc la liste vide. Si nous tapons :

```
?glist ".DEF
[]
```

nous obtenons la liste de tous les objets ayant reçu la définition d'une procédure. Pour l'instant, cette liste est vide.

```
?glist ".PRM
```

Cette fois la procédure donne la liste de toutes les primitives Logo. Cette liste est légèrement différente de la liste donnée par la procédure **.contents.** Tous les objets prédéfinis de Logo ne sont donc pas des primitives. En fait, les objets dont le nom apparaît en majuscules dans la liste fournie par **.contents** sont des variables système.

La procédure make

Cette procédure que nous avons déjà rencontrée affecte une valeur à un objet. Si nous soumettons à Logo un objet qu'il ne connaît pas, en tapant par exemple :

```
?a
```

Logo répond :

```
I don't know to a
```

A partir de ce moment, l'objet *a* est connu de Logo. On peut le vérifier à l'aide de la commande **.contents**. Cependant, si l'on tape :

```
?glist ".APV
[]
```

on obtient toujours la liste vide. L'objet *a* n'a pas de valeur. La procédure **make** peut être utilisée pour donner une valeur à *a* :

```
?make "a "b
```

Si l'on tape maintenant :

```
?glist ".APV
[a]
```

on obtient une liste contenant un élément : *a*.

La procédure plist

La procédure **plist** est une primitive Logo qui peut être utilisée pour fournir la liste des propriétés d'un objet.

```
?plist "a
[.APV b]
```

Cet exemple nous montre que l'objet *a* a la propriété valeur (.APV) *b*. Si nous affectons à *a* une définition de procédure, par exemple :

```
?to a :c
>pr :c
>end
```

la procédure **plist** appliquée à *a* donnera le résultat suivant :

```
?plist "a
[.DEF [[c] [pr :c]] .APV b]
```

Le résultat affiché est une liste qui comporte quatre éléments. Le premier est la propriété .DEF. Le deuxième est la définition de la procédure. Le troisième est la propriété .APV et le quatrième est la valeur. Le deuxième élément (la définition de la procédure) est lui-même une liste comportant deux éléments qui sont tous les deux des listes. La première est la liste des arguments formels et la seconde est le corps de la définition.

Si nous avions d'abord défini la procédure puis ensuite seulement attribué une valeur à *a*, nous aurions obtenu le résultat suivant :

```
?plist "a
[.APV b [.DEF [[c] [pr :c]]]
```

La procédure gprop

La procédure **gprop** est une primitive qui permet d'obtenir une propriété d'un objet :

```
?gprop "a ".DEF
[[c] [pr :c]]
?gprop "a ".APV
b
```

La procédure **gprop** peut être utilisée pour définir la procédure valeur qui fournit la valeur d'un objet :

```
?to valeur :arg
>op gprop :arg ".APV
>end
```

Cette procédure existe sous forme de primitive sur l'Amstrad PCW 8256 et 8512 sous le nom de "thing". Elle a le même effet que les deux points (:) placés devant un nom d'objet :

```
?:a
b
?valeur "a
b
```

Son avantage est que l'on peut la combiner pour obtenir la valeur d'une valeur. Si l'on affecte à *b* la valeur *c* :

```
?make "b "c
```

on ne peut obtenir *c* à partir de *a* en écrivant :

```
?::a
```

ce qui affiche seulement :

```
:a has no value
```

On peut par contre obtenir le résultat souhaité en tapant :

```
?valeur valeur "a
c
```

La procédure op

Logo ne possède pas de fonctions (contrairement à LISP dont il dérive et qui, lui, ne possède pas de procédures). Grâce à la procédure **op**, il est possible de transformer une procédure en fonction. La procédure **op** fournit le résultat de la procédure qu'elle a pour argument à la procédure dont elle est argument. Dans l'exemple de la procédure **valeur**, si l'on omet **op**, Logo répond :

```
?valeur "a
I don't know what to do with b in valeur: gprop :arg ".APV
```

ce qui indique que le résultat de la procédure **gprop** est bien *b* mais que Logo ne sait pas quoi en faire. La présence de la procédure **op**

devant **grop** indique à Logo de passer le résultat de **grop** à la procédure dont **op** est l'argument. Dans le cas présent, **op** n'est apparemment l'argument d'aucune procédure. En fait, l'interface utilisateur de Logo (la partie qui permet à l'utilisateur de soumettre à Logo des objets) est une procédure. Sa fonction est de servir d'intermédiaire (interface) entre Logo et l'utilisateur. C'est elle qui lit les caractères tapés par l'utilisateur au clavier et qui affiche les réponses de Logo. Quand le résultat d'une procédure est fourni à cette procédure de niveau supérieur (TOPLEVEL), ce résultat est simplement affiché.

La procédure **plist** peut être appliquée à une primitive Logo :

```
?plist "lt
[.PRM 7241]
```

Le résultat indique que **lt** a la propriété .PRM (primitive). La valeur 7241 est l'adresse à laquelle commence la définition de cette primitive en mémoire. De même on peut utiliser pour obtenir cette adresse la procédure **gprop** :

```
?gprop "lt ".PRM
7241
```

La procédure pops

La procédure **pops** fournit la liste des procédures qui ont été définies. Cette procédure est une primitive du Logo de l'Amstrad PCW 8256 et 8512 mais n'existe pas sur les 464 et 664. Nous pouvons la définir de la façon suivante :

```
to pops
po glist ".DEF
end
```

Au stade où nous en sommes, cette procédure doit donner le résultat suivant :

```
?pops
to a :c
pr :c
end
to valeur :arg
gprop :arg ".APV
end
?
```

(Sur un 464 ou un 664, la définition de la procédure **pops** s'ajoute à cette liste.)

La procédure edall

Cette procédure permet d'éditer toutes les procédures qui ont été définies ainsi que toutes les affectations de valeurs. Cette procédure n'existe pas dans toutes les versions de Logo. Si votre version ne possède pas cette primitive, voici comment la définir :

```
to edall
ed glist ".DEF
end
```

Essayez-la en tapant :

```
?edall
```

Le mode Edition est activé et l'écran affiche toutes les procédures qui ont été définies et toutes les affectations de valeurs.

La procédure er

Cette primitive permet d'effacer de la mémoire une définition de procédure. Tapez :

```
?er "a
```

et vérifiez en utilisant la procédure **edall** que la définition de la procédure *a* a bien été supprimée.

D'autres primitives permettant de manipuler des procédures peuvent exister dans certaines version de Logo :

- **po** "a affiche la définition de la procédure *a*
- **poall** affiche les définitions de toutes les procédures et de toutes les variables.
- **pots** affiche les noms et les arguments virtuels de toutes les procédures.
- **text** "a affiche le corps de la procédure *a*
- **erall** efface toutes les variables et toutes les procédures de l'espace de travail.
- **ern** [liste] détruit toutes les variables se trouvant dans la liste utilisée comme argument.
- **pons** affiche les noms et les valeurs de tous les objets qui ont été définis.
- **thing** "a affiche la valeur de *a*
- **nodes** indique le nombre de "noeuds" (unité de mémoire pour Logo) disponibles.
- **recycle** déclenche l'opération de "garbage collection" qui consiste à récupérer les espaces inutilisés. Logo exécute automatiquement cette procédure lorsque la mémoire commence à être saturée. Pour vérifier son fonctionnememt, utilisez la procédure **nodes** avant et après la procédure **recycle**.

Les procédures liées à l'utilisation des disques

Logo possède 5 primitives pour gérer les fichiers sur disque :

- **dir** permet d'afficher la liste des fichiers se trouvant sur le disque. Seuls les fichiers Logo obtenus par la sauvegarde de

l'espace de travail à l'aide de la procédure **save** sont affichés. La procédure **dir** peut prendre comme argument un nom de fichier qui peut éventuellement comporter les caractères génériques * et ?. (Pour tout renseignement concernant les caractères génériques et les noms de fichiers, voir *Amstrad, CP/M 2.2* et *Amstrad, CP/M plus*, Sybex.)

- **save** "nom_de_fichier permet de sauvegarder l'espace de travail dans le fichier dont le nom est fourni comme argument.
- **load** "nom_de_fichier permet de charger dans l'espace de travail le contenu du fichier argument. Les définitions de procédures et de variables contenues dans le fichier sont ajoutées à celles se trouvant déjà dans l'espace de travail.
- **erasefile** "nom_de_fichier sert à effacer un fichier sur le disque.
- **changef** "ancien_nom "nouveau_nom permet de changer le nom d'un fichier sur disque.

CHAPITRE VI

PROCEDURES RECURSIVES ET PROCEDURES ITERATIVES

Dans ce chapitre, nous allons continuer à développer de nouvelles procédures, toujours en partant des cinq procédures de bases : **first, bf, fput, = et wordp**

Logo possède une primitive permettant d'extraire le dernier élément d'une liste. Nous allons écrire notre propre définition. Deux approches sont possibles. Logo permettant la récursivité, nous allons commencer par définir la procédure récursive **dernier** :

```
?to dernier :liste
>if nulle :liste [op :liste]
>if nulle bf :liste [op first :liste]
>op dernier bf :liste
>end
```

Le fonctionnement de cette procédure est facile à comprendre. La procédure prend pour argument la liste dont il faut extraire le dernier élément. Si cette liste est vide, la procédure renvoie son argument, c'est-à-dire la liste vide. Si le reste de la liste moins le premier élément est vide (c'est-à-dire si la liste ne comporte qu'un seul élément), la procédure renvoie le premier élément de la liste (c'est évidemment aussi le dernier). Si aucune de ces deux conditions n'est remplie, la procédure est de nouveau appliquée au reste de la liste moins le premier élément. On s'aperçoit facilement qu'au prochain tour (si la liste comporte plus de deux éléments), la procédure s'appliquera à la liste moins les deux premiers éléments, et ainsi de suite jusqu'à ce qu'il ne reste plus qu'un élément. Détaillons ci-dessous le traitement de la liste [a b c] par la procédure **dernier.**

L'appel de la procédure se fait de la façon suivante :

```
?dernier [a b c]
```

L'argument réel remplace tout d'abord l'argument formel. La définition de la procédure devient :

```
to dernier [a b c]
if nulle [a b c] [op [a b c]]
if nulle bf [a b c] [op first [a b c]]
op dernier bf [a b c]
end
```

La deuxième ligne de la procédure teste [a b c] pour voir s'il s'agit de la liste vide. Ce n'est pas le cas, donc la ligne suivante est exécutée.

La troisième ligne teste **bf** [a b c] pour voir s'il s'agit de la liste vide. **bf** [a b c] est exécutée et donne [b c]. La condition n'est donc pas remplie et la ligne suivante est exécutée.

La quatrième ligne comporte un appel à la procédure **dernier.** La procédure **dernier** s'appelle donc elle-même. Ce procédé constitue la récursivité. La procédure **dernier** est cette fois appelée avec l'argument réel **bf** [a b c] qui est évalué et donne [b c]. Cet argument réel prend la place de l'argument formel de la procédure, qui va maintenant être exécuté sous la forme :

```
to dernier [b c]
if nulle [b c] [op [b c]]
if nulle bf [b c] [op first [b c]]
op dernier bf [b c]
end
```

La deuxième ligne de la procédure teste [b c] pour voir s'il s'agit de la liste vide. Ce n'est pas le cas, donc la ligne suivante est exécutée.

La troisième ligne teste **bf** [b c] pour voir s'il s'agit de la liste vide. **bf** [b c] est exécutée et donne [c]. La condition n'est donc pas remplie et la ligne suivante est exécutée.

La quatrième ligne comporte à nouveau un appel à la procédure **dernier**. La procédure **dernier** est cette fois appelée avec l'argument réel **bf** [b c] qui est évalué et donne [c]. Cet argument réel prend la place de l'argument formel de la procédure, qui va maintenant être exécuté sous la forme :

```
to dernier [c]
if nulle [c] [op [c]]
if nulle bf [c] [op first [c]]
op dernier bf [c]
end
```

La deuxième ligne de la procédure teste [c] pour voir s'il s'agit de la liste vide. Ce n'est pas le cas, donc la ligne suivante est exécutée.

La troisième ligne teste **bf** [c] pour voir s'il s'agit de la liste vide. **bf** [c] est exécutée et donne [] (la liste vide). Cette fois, la condition est remplie. La fin de la ligne est exécutée : [op first [c]]. **first** [c] est évaluée et donne *c* qui est la valeur renvoyée par la procédure.

On voit que la deuxième ligne n'a pas été exécutée. En fait cette ligne n'est là que pour traiter le cas où l'argument réel est la liste vide. Nous avons choisi dans ce cas de faire renvoyer par la procédure la valeur [] (liste vide). C'est une question de convention. La primitive Logo exécutant la même opération **(last)** ne traite pas la liste vide de la même façon et produit un message d'erreur signalant que le paramètre est incorrect. Nous pensons pour notre part qu'il est préférable d'utiliser une version plus intelligente de cette procédure, capable de traiter tous les cas. Si vous souhaitez plus d'homogénéité (la procédure **first** produisant elle aussi un message d'erreur si son argument est la liste vide), vous pouvez réécrire la procédure **dernier** pour qu'elle produise le même message d'erreur que la procédure **last** :

```
to dernier :liste
if nulle :liste [pr [dernier n'aime pas l'argument [] stop]
if nulle bf :liste [op first :liste]
op dernier bf :liste
end
```

Notez au passage la présence de la primitive **stop** à la fin de la deuxième ligne. Dans la version précédente, la primitive **op** stoppait le déroulement de la procédure. Dans cette version, **stop** est néces-

saire pour empêcher les lignes suivantes d'être exécutées après l'affichage du message d'erreur.

Vous pouvez également utiliser en remplacement de la procédure **first** une version amémiorée de la procédure **premier** que nous avons définie précédemment :

```
to premier :liste
if nulle :liste [op :liste]
op first :liste
end
```

Cette procédure renvoie la liste vide si son argument est la liste vide.

Nous allons maintenant définir une version itérative de la procédure **dernier**, que nous appellerons dernier2 :

```
to dernier2 :liste
if nulle :liste [op :liste]
label "a
if nulle bf :liste [op first :liste]
make "liste bf :liste
go "a
end
```

La première différence évidente est que cette procédure ne comporte pas d'appel à elle-même : elle n'est pas récursive. Nous voyons d'autre part apparaître une nouvelle structure constituée des primitives **label** et **go**. Lorsque Logo rencontre la primitive **go** suivie d'un nom d'objet, l'exécution de la procédure se poursuit à l'instruction suivant la procédure **label** précédant le même nom. Ici, lorsque Logo rencontre la ligne :

```
go "a
```

l'exécution se poursuit à la procédure suivant la ligne :

```
label "a
```

Les deux premières lignes de la procédure dernier2 sont identiques à celles de la version récursive. La boucle label "a ..... go "a est ensuite exécutée jusqu'à ce que Logo rencontre une condition de sortie. Détaillons le traitement de la liste [a b c] :

La procédure est d'abord exécutée avec pour argument réel la liste [a b c] :

```
to dernier2 [a b c]
if nulle [a b c] [op [a b c]]
label "a
if nulle bf [a b c] [op first [a b c]]
make "liste bf [a b c]
go "a
end
```

La deuxième ligne teste si [a b c] est la liste vide. Ce n'est pas le cas. L'exécution se poursuit donc à la ligne suivante qui n'a aucun effet.

La quatrième ligne teste si **bf** [a b c] est la liste vide, c'estàdire si la liste ne comporte plus qu'un seul élément. Ce n'est pas le cas. L'exécution se poursuit à la cinquième ligne qui attribue à "liste la valeur **bf** [a b c], c'estàdire [b c]. Ceci est très important car, de ce fait, toutes les occurrences de :liste seront remplacées par [b c]. La boucle de la procédure est maintenant équivalente à :

```
label "a
if nulle bf [b c] [op first [b c]]
make "liste bf [b c]
go "a
```

La procédure **go** "a renvoie ensuite l'exécution à la ligne label "a qui une fois de plus ne produit aucun effet.

La deuxième ligne de la boucle teste si **bf** [b c] est la liste vide, c'estàdire si la liste ne comporte plus qu'un seul élément. Ce n'est pas le cas. L'exécution se poursuit à la ligne suivante qui attribue à "liste la valeur **bf** [b c], c'estàdire [c]. Ainsi, toutes les occurrences de :liste seront maintenant remplacées par [c]. La boucle de la procédure est maintenant équivalente à :

```
label "a
if nulle bf [c] [op first [c]]
make "liste bf [c]
go "a
```

La procédure go" a renvoie ensuite l'exécution à la ligne label "a. Cette fois, la condition nulle **bf** [c] est remplie et la procédure renvoie la valeur **first** [c] qui est bien le dernier élément de la liste [a b c]. La procédure **op** induit de plus une condition de sortie de la boucle.

On constate tout de suite une différence importante par rapport à la version récursive. Logo n'a pas besoin d'appeler plusieurs fois la procédure **dernier** pour exécuter la répétition des opérations. La boucle label-go produit le même effet en évitant de répéter la première ligne de la procédure qui se trouve hors de la boucle. C'est une des raisons de la vitesse d'exécution supérieure de la version itérative.

On pourra de la même façon définir une version itérative de la procédure **element** que nous appellerons element2. Cette procédure prendra deux arguments, un objet quelconque :a et une liste :liste. Pour savoir si :a est élément de liste, nous chercherons tout d'abord à savoir si :liste est égale à la liste vide. Si c'est le cas, la procédure s'arrêtera en renvoyant la valeur FALSE. Dans le cas contraire, nous regarderons si a: est égal au premier élément de :liste. Si oui, la procédure s'arrêtera en renvoyant la valeur TRUE. Dans le cas contraire, nous enlèverons le premier élément de :liste et nous recommencerons les opérations. Essayez de définir la procédure element2.

Voici une solution :

```
to element2 :a :liste
label "b
if nulle :liste [op "FALSE]
if = :a first :liste [op "TRUE]
make "liste bf :liste
go "b
end
```

Comparons cette définition avec celle de la version récursive :

```
to element :a :liste
if nulle :liste [op "FALSE]
if = :a first :liste [op "TRUE]
op element :a bf :liste
end
```

La compacité de la version récursive apparaît ici de façon flagrante, une ligne remplaçant trois lignes de la version itérative :

Version itérative :

```
label "b
...
...
make "liste bf :liste
go "b
```

Version récursive :

```
op element :a bf :liste
```

Tout le principe de la récursivité se trouve dans cette ligne. Nous avons dit précédemment que **op** renvoyait une valeur en arrêtant le déroulement de la procédure. On pourrait croire a priori que la procédure **element** va s'arrêter à cette ligne. Comment alors se poursuivent les opérations ? Les choses seront plus faciles à comprendre si on écrit cette ligne de la façon suivante :

```
op (element :a bf :liste)
```

Tout ce qui se trouve entre parenthèses constitue l'argument de la procédure **op**. Pour pouvoir renvoyer cet argument, celui-ci doit être évalué. la procédure **element** va donc être de nouveau exécutée à l'intérieur de la procédure **op** qui ellemême fait partie de la procédure **element**. Pour mieux comprendre ce phénomène, nous allons comparer les déroulements des procédures **element** et element2 avec pour argument le mot "c et la liste [b c].

Version itérative :

```
to element2 "c [b c]          :a et :liste remplacés par "c
                              et [b c]
label "b
if nulle [b c]                condition non remplie
if = "c first [b c]           condition non remplie
make "liste bf [b c]          :liste vaut [c]
go "b                         l'exécution reprend à
label "b                      la ligne label "b
if nulle [c]                  condition non remplie
if = "c first [c]             condition remplie
op "TRUE                      le programme s'arrête en
                              renvoyant la valeur TRUE
```

Version récursive :

```
to element "c [b c]           :a et :liste remplacés par "c
                              et [b c]
if nulle [b c]                condition non remplie
if = "c first [b c]           condition non remplie
op (element "c bf [b c])      :a et :liste remplacée
   (                    )     par "c et bf [b c]
   (if nulle [c]        )     condition non remplie
   (if = "c first [c]   )     condition remplie
   (op "TRUE            )     la procédure op renvoie la
                              valeur TRUE à la procédure op
                              précédente qui la renvoie
                              comme résultat de la procédure
                              element
```

Une différence extrêmement importante apparaît ici. Dans la version itérative, tout le programme se déroule sur le même plan. Dans la version récursive, la procédure est exécutée à l'intérieur d'elle-même. En fait la procédure **element** se termine après l'évaluation de l'argument de la procédure **op**. La série de procédures composant **element** n'est donc exécutée sur un même plan qu'une fois seulement. La deuxième exécution se fait à l'intérieur de la première, en profondeur. Si on avait remplacé [b c] par [a b c], il y aurait eu trois niveaux d'imbrication.

Nous allons maintenant définir une nouvelle procédure itérative appelée **nieme**. Cette procédure, si on l'appelle avec pour arguments

une liste et un nombre n, doit renvoyer la fin de la liste à partir du $n^{ième}$ élément. Exemple :

```
nieme [a b c d e] 3
```

doit donner :

```
[c d e]
```

Si le nombre est supérieur au nombre d'éléments dans la liste, la procédure doit renvoyer la liste vide. Essayez de définir vousmême la procédure **nieme** avant de regarder la solution ci-dessous (souvenezvous de la procédure **ou** que nous avons définie au chapitre précédent).

```
to nieme :liste :a
label "x
if ou = 1 :a nulle :liste [op :liste]
make "a (- :a 1)
make "liste bf :liste
go "x
end
```

Si vous avez du mal à saisir l'utilisation de la procédure **ou**, remplacez la troisième ligne par les deux lignes suivantes :

```
if = 1 :a [op :liste]
if nulle :liste [op :liste]
```

Note : les parenthèses de la quatrième ligne sont indispensables à cause de l'utilisation de la notation préfixée. Leur omission provoque l'affichage d'un message d'erreur.

Nous allons maintenant utiliser la procédure **nieme** pour définir la procédure récursive **index** qui prendra pour argument deux listes dont la seconde sera une liste de nombres. La procédure **index** fournira une liste contenant les éléments de la première liste de rang correspondant aux éléments de la seconde. Exemple :

```
index [a b c d e f g] [3 1 5]
```

donnera:

```
[c a e]
```

c'estàdire une liste contenant le troisième, le premier et le cinquième éléments de la liste constituant le premier argument. Si la seconde liste est la liste vide, la procédure devra renvoyer la liste vide.

Essayez de définir la procédure **index** sans regarder la solution proposée ci-dessous.

```
to index :liste1 :liste2
if nulle :liste2 [op liste2]
op fput first nieme :liste1 first :liste2 index :liste1 bf :liste2
end
```

Souvenezvous que le seul critère important pour juger la valeur d'une procédure est d'ordre fonctionnel. Que votre solution diffère de celle proposée n'a aucune importance du moment qu'elle produit bien le résultat escompté.

La plupart des procédures que nous avons définies jusqu'ici existent en tant que primitives Logo. Nous les avons redéfinies uniquement pour montrer que Logo peut être presque entièrement redéfini en Logo. Bien sûr, les primitives sont beaucoup plus rapides et vous devrez les utiliser si vous voulez écrire des programmes performants. La liste des primitives Logo disponibles sur chaque version de l'Amstrad est donnée au Chapitre 8.

Nous avons rencontré dans les pages précédentes un exemple dans lequel l'emploi de parenthèses était rendu obligatoire par l'utilisation de la notation préfixée. Il existe un autre cas où les parenthèses sont obligatoires. L'emploi de parenthèses est obligatoire chaque fois qu'une procédure peut prendre un nombre indéterminé d'arguments. Il en est ainsi de la procédure **list** qui prend pour argument un nombre indéterminé d'objets et renvoie une liste ayant tous ces objets pour éléments. Exemple :

```
?(list "a "b "c "d "e "f)
[a b c d e f]
```

La procédure **list** fait partie, avec les procédures **fput** et **se** qui est la primitive équivalente de la procédure **concat** que nous avons définie, des procédures de construction par opposition avec les primitives **first, bf** et **last** (équivalent de notre procédure **dernier**) qui sont des procédures d'extraction.

Il est important de bien comprendre les différences entre **list, se** et **fput**. Ces différences apparaissent clairement en considérant les trois exemples suivants :

```
?list [a b] [c d]
[[a b] [c d]]

?se [a b] [c d]
[a b c d]

?fput [a b] [c d]
[[a b] c d]
```

Note : en ce qui concerne la procédure **list**, les parenthèses ne sont pas obligatoires s'il n'y a que deux arguments.

La procédure **list** produit une liste ayant autant d'éléments qu'elle a d'arguments. La procédure **se** produit une liste ayant autant d'éléments que la somme des nombres d'éléments de ses arguments. La procédure **fput** produit une liste ayant un élément de plus que son deuxième argument.

Nous allons maintenant définir une version itérative de la procédure **renverse** précédemment définie de manière récursive :

```
to renverse2 :liste1
local "liste2
label "a
if nulle :liste1 [op :liste2]
make "liste2 fput first :liste1 :liste2
make :liste1 bf :liste2
go "a
end
```

La définition de cette procédure nécessite l'emploi d'une variable que nous avons appelée liste2. Comme son nom l'indique, une variable locale n'est connue qu'à l'intérieur de la procédure où elle est déclarée. Cette variable locale est utilisée pour stocker un par un les éléments de l'argument liste1. Les lignes 5 et 6 effectuent cette opération. La ligne 5 place en tête de la liste liste2 le premier élément de la liste liste1. La ligne 6 supprime le premier élément de la liste liste1. A chaque passage de la boucle label "a....go "a, la liste liste1 est testée pour voir si elle est vide. Si ce n'est pas le cas, une nouvelle boucle est effectuée. Sinon, le renversement de liste1 dans liste2 est terminé et la procédure s'arrête à la ligne 4 en renvoyant la liste liste2.

Lorsque l'exécution d'une procédure se termine, Logo remet tous les objets qui ont été déclarés à l'intérieur de la procédure dans l'état dans lequel ils se trouvaient avant l'appel de la procédure. Si la définition de la procédure comporte l'argument formel "a, la valeur de l'objet "a est mémorisée au début de la procédure et restituée lorsque celle-ci se termine. Il est cependant possible de modifier dans une procédure un objet déclaré à l'extérieur, à condition que celui-ci ne figure pas dans la liste des arguments de la procédure ou comme argument d'une déclaration locale. Exemple :

```
to abc :a :liste
local "c
make "c "a
make "liste fput :a :liste
make "a "b
make "b "c
end
```

Dans cet exemple, si les objets "a, "b, "c et "liste existaient avant l'appel de la procédure, seule la valeur de "b est modifiée par la procédure. Les valeurs de "a, "c et "liste sont restituées lorsque la procédure se termine.

Une fois de plus, la version itérative de la procédure **renverse** est plus rapide que la version récursive, surtout pour le traitement de listes longues. La définition de la version récursive est plus courte, et donc plus économique en espace mémoire. Lorsque l'on doit choisir entre une procédure récursive et une procédure itérative, il

est bon de tenir compte du nombre d'utilisations de cette procédure. Les procédures devant être utilisées souvent (en particulier dans des boucles) doivent être rapides. On sera donc amené à choisir dans ce cas une version itérative. En ce qui concerne les procédures peu utilisées, on pourra préférer économiser la mémoire en choisissant la version récursive.

Il faut noter que si la définition d'une procédure récursive est économique en mémoire, son utilisation en consomme beaucoup. Si l'utilisation d'une procédure récursive produit une erreur due à l'épuisement de la mémoire, la solution sera souvent de la remplacer par un version itérative.

CHAPITRE VII

PROCEDURES DE CALCUL NUMERIQUE

Dans ce chapitre, nous allons développer un certain nombre de procédures de calcul numérique. Nous allons tout d'abord voir comment Logo traite les nombres.

Tapez :

```
?"1
1
```

Nous voyons que Logo considère "1 comme un nom. Tapez maintenant :

```
?:1
1 has no value
```

Nous constatons qu,e contrairement à ce à quoi nous aurions pu nous attendre, "1 est le nom d'un objet ordinaire qui n'a pas de valeur prédéfinie. Nous pouvons même lui attribuer une valeur quelconque, par exemple :

```
?make "1 2
?:1
2
```

Nous avons attribué à "1 la valeur 2. Nous constatons cependant que Logo a traité 2 d'une façon particulière. En effet, nous n'avons pas écrit "2 qui aurait été un nom d'objet, ou :2 qui aurait représenté une valeur, mais 2 sans aucun signe devant. Normalement, Logo devrait alors considérer 2 comme un nom de procédure et produire le message d'erreur :

```
I don't know to 2
```

Cependant, rien de tel ne s'est produit. En fait, Logo traite les nombres comme des procédures prédéfinies (des primitives) qui renvoient leur propre valeur.

Tapez maintenant :

```
?make "a 12
?plist "a
[.APV 12]
```

Nous voyons que l'objet "a a reçu la propriété "valeur 12". Si nous tapons maintenant :

```
?:a
12
```

Logo renvoie la valeur 12. Essayons maintenant l'exemple suivant :

```
?make "a (/ 1 3)
?plist "a
[.APV 0.333333333333333]
```

Comme nous l'avons vu dans les chapitres précédents, Logo exécute d'abord la procédure / puis le résultat est affecté à "a. Logo stocke les nombres fractionnaires avec 15 chiffres après le point décimal.

Continuons nos expériences sur les nombres en tapant :

```
 ?"111111111111111111111111111111111111
1111111111111111111111111111111111111
```

Ici, Logo a affiché le nom de l'objet que nous avions fait précéder d'un guillemet.

```
?


1111111111111111111111111111111111111
1.1111111111111e+34
```

Logo affiche cette fois une valeur ramenée au format standard. Essayez maintenant :

```
?"11111111111111111111111111111/1
1.11111111111111e+29
```

Logo a cette fois interprété "1111111111111111111111111111111 comme un nombre pour pouvoir le diviser par 1. Il résulte de ce fait que si l'on tape :

```
?ab/cd
/ doesn't like ab as input
```

on obtient un message d'erreur car Logo interprète ceci comme *ab* divisé par *cb*. Si l'on veut employer dans un nom d'objet un caractère utilisé comme nom de procédure, on doit le faire précéder de \ ou de ^Q. (Côntrole Q ou Alt Q). Ainsi, pour obtenir le résultat souhaité, il faudra taper :

```
?ab\/cd
ab/cd
```

Nous avons vu que Logo admet à la fois la notation courante infixée et la notation fonctionnelle préfixée, c'est-à-dire que les deux exemples ci-dessous sont équivalents :

```
?/ 4 2
2
?4 / 2
2
```

Il existe cependant certaines différences. La notation infixée permet de supprimer les espaces, l'opérateur servant de délimiteur :

```
?4/2
2
```

Par contre, la notation préfixée offre deux avantages considérables. Elle est cohérente avec la notion de procédure de Logo. De plus, elle permet avec certains opérateurs d'utiliser un nombre quelconque d'arguments :

```
?(+ 1 2 3 4 5 6 7)
28
?(* 1 2 3 4 5 6 7)
5040
```

Une conséquence de ce fait est l'obligation d'utiliser des parenthèses. En l'absence de parenthèses, Logo considère ces exemples comme la suite d'instructions :

```
?+ 1 2
3
4
5
6
7
```

et fournit donc le résultat suivant :

```
3
3
4
5
6
7
```

Le premier 3 est le résultat de + 1 2. Les autres chiffres sont simplement répétés par Logo. La notation préfixée permet de ne pas tenir compte de la priorité des opérateurs. Ainsi, en notation infixée :

```
2*3+4
```

est équivalent de :

```
(2*3)+4
```

à cause de la priorité de la multiplication sur l'addition. On ne peut pas écrire :

```
2*(3+4)
```

sans utiliser de parenthèses. Avec la notation préfixée, pas de problème. Il suffit de se souvenir que Logo commence toujours la résolution des procédures "en profondeur". Ainsi :

```
+ * 2 3 4
```

est interprété par Logo comme :

```
+ (* 2 3) 4
```

et donne 10.

```
* + 2 3 4
```

est interprété comme :

```
* (+ 2 3) 4
```

et donne 20. Cependant, dans certains cas, la notation préfixée exige des parenthèses. C'est le cas dans l'exemple suivant :

```
?make "a * 2 3
```

qui est interprété par Logo comme :

```
?make ("a * 2) 3
```

et fournit le message d'erreur :

```
* doesn't like a as input
```

(un nom ne pouvant être utilisé comme opérande dans une opération arithmétique). Pour obtenir le résultat souhaité, il faut écrire :

```
?make "a (* 2 3)
```

La procédure factorielle

La fonction mathématique factorielle notée ! est définie de la façon suivante :

0! = 1

n! = n * (n-1)! si n est différent de 0

Essayez de définir une procédure factorielle récursive.

La solution qui vient immédiatement à l'esprit est d'appliquer simplement la définition :

```
to ! :n
if =0 :n [op 1]
op * :n ! - :n 1
end
```

Essayez cette procédure avec diverses valeurs. Si vous essayez avec un argument supérieur à 40, Logo affiche le message :

```
Out of Logo Stack space
```

Pour contourner cet inconvénient, nous pouvons définir une version itérative que nous appellerons !2 :

```
to !2 :n
local "a
make "a 1
label "b
if = 0 :n [op :a]
make "a (* :n :a)
make "n (- :n 1)
go "b
end
```

Cette version est beaucoup moins compacte. (Notons au passage les parenthèses obligatoires ici pour indiquer que les opérateurs sont utilisés en notation préfixée avec les deux opérandes qui les suivent et non en notation infixée.)

Nous pouvons maintenant obtenir factorielle 60 sans problème. Cependant, si vous essayez !2 100 et !2 1000 vous pouvez constater que Logo fournit le même résultat dans les deux cas, la capacité maximum de représentation des nombres étant dépassée. Il faut toujours rester vigilant à ce sujet car Logo ne fournit pas dans ce cas de message d'erreur.

La procédure Fibonacci

La suite de Fibonacci est une suite de nombres entiers tels que :

F(0) = 1

F(1) = 1

F(n) = F(n-1)+F(n-2) si n>1

Essayez de définir une procédure récursive **fib** donnant les nombres de Fibonacci. L'exemple le plus simple reprend tout simplement la définition :

```
to fib :n
if ou = :n 0 = :n 1 [op 1]
op + fib - :n 1 fib - :n 2
end
```

Cette procédure fonctionne correctement, mais elle est d'une lenteur extrême. La raison en est que pour chaque valeur de n, on calcule f(n-1) et f(n-2). Il en résulte que chaque valeur est calculée deux fois :

pour n : f(n-1) et f(n-2)

pour n-1 : f(n-2) et f(n-3)

pour n-2 : f(n-3) et f(n-4)

Ce défaut n'est cependant pas lié à la récursivité. Nous allons maintenant définir la procédure fib2 qui, au lieu de commencer par n et de progresser vers 0, partira de 0 et progressera vers n en accumulant les résultats dans deux variables. Essayez de définir une

telle procédure récursive. Pour vous aider, sachez que la procédure fib2 prendra trois arguments et non un seul.

Voici notre solution à ce problème :

```
to fib2 :n :f1 :f2
if = 0 :n [op :f1]
op fib2 (- :n 1 (+ :f1 :f2) :f1
end
```

Cette fonction est appelée avec trois arguments, le deuxième valant 1 et le troisième 0. Exemple :

```
?fib2 8 1 0
34
```

Si l'on veut absolument une procédure ne prenant qu'un seul argument, on peut très bien définir maintenant une procédure servant d'interface que nous appellerons **fibo** :

```
to fibo :n
op fib2 :n 1 0
end
```

L'appel de la procédure **fibo** se fait avec un seul argument :

```
?fibo 8
34
```

Comparez les vitesses d'exécution des procédures **fib** et **fibo**.

La procédure puissance

Construisons maintenant une procédure donnant le résultat de l'élévation d'un nombre n à la puissance m et que nous appellerons **puissance**.

```
to puissance :n :m
local "resultat
make "resultat 1
label "a
if = 0 :m [op :resultat]
make "resultat (* :n resultat)
make "m (- :m 1)
go "a
end
```

Le fonctionnement de cette procédure est très simple. La variable **resultat** est déclarée locale et reçoit la valeur 1. Si :m vaut 0, **resultat** est renvoyé par la procédure, ce qui correspond bien à la définition de la fonction **puissance** car X puissance 0 vaut 1.

Si :m est différent de 0, **resultat** est multiplié par :n et :m est décrémenté, puis l'exécution reprend avec le test de la valeur de :m. Il est clair que lorsque :m vaudra 0, :n aura été multiplié :m fois par lui-même dans **resultat.**

On peut définir une autre procédure utilisant la primitive **run.** Cette primitive prend pour argument une liste d'objet qui doit respecter la syntaxe d'une ligne de commande Logo et l'exécute. Ainsi :

```
?run [2 * 2 * 2 * 2]
16
```

Nous pouvons donc réécrire la procédure **puissance** sous la forme suivante :

```
to p :n :m
local "a
make "a []
label "x
if = 0 :m [op run bf :a]
make "a fput :n :a
make "a fput "\* :a
make "m (- :m 1)
go "x
end
```

La variable "a est déclarée locale et reçoit dans une boucle :m fois [* :n] puis le premier * est retiré et la liste est soumise à la procédure **run** qui l'exécute. Cette version ne présente pas d'intérêt par rapport à la version précédente.

Nous pouvons également définir une version récursive de cette procédure en remarquant les faits suivants :

- si m est pair, n à la puissance m est égal à n*n à la puissance M/2.
- si m est impair, n à la puissance m est égal à n * (n à la puissance n-1).

Pour définir cette procédure, nous aurons besoin d'une procédure **pair** qui devra donner TRUE si son argument est pair et FALSE dans le cas contraire. Définissez **pair** en utilisant la procédure INT qui est la primitive Logo donnant la valeur entière de son argument.

En voici une définition :

```
to pair :n
if = int / :n 2 /:n 2 [op "TRUE]
op "FALSE
end
```

Notez que pair 0 donne un message d'erreur (division par 0). On peut traiter le cas de 0 en ajoutant entre la première et la deuxième lignes :

```
if = 0 :n [op "TRUE]
```

si l'on veut considérer 0 comme pair. Cette caractéristique n'est pas utile ici car **pair** n'est pas faite pour être utilisée seule mais pour être appelée par la procédure puissance2 qui ne le fera jamais avec 0 pour argument.

Nous pouvons maintenant définir la procédure puissance2 :

```
to puissance2 :n :m
if = 0 : m [op 1]
if pair :m [op puissance2 (* :n :m) (/ :m 2)]
```

```
op * puissance2 :n (- :m 1)
end
```

Procédure donnant le PGCD

Nous aurons besoin pour la suite de notre propos de la fonction **reste** qui donnera le reste de la division et que nous pouvons définir ainsi :

```
to reste :n :m
op - :n (* int / :n :m :m)
end
```

A l'aide de la procédure **reste**, nous allons maintenant pouvoir définir la procédure PGCD donnant le plus grand commun diviseur de ses deux arguments. L'algorithme utilisé est le suivant (on cherche le PGCD de n et m) :

- si n = 0 PGCD (n, m) = m
- sinon PGCD (n, m) = pgcd ((reste m, n),n)

Logo permet de définir la procédure PGCD d'une manière très proche de l'algorithme utilisé :

```
to pgcd :n :m
if = 0 :n [op :m]
op pgcd reste :m :n :n
end
```

Les lecteurs qui connaissent le BASIC apprécieront la compacité de cette définition.

Le tri

Nous allons enfin écrire une procédure permettant de trier une liste de nombres par ordre croissant. Nous utiliserons pour cela la méthode du tri par insertion. Nous avons donc tout d'abord besoin de la procédure **insert** permettant d'insérer un nombre à sa place dans une liste triée. En voici un exemple récursif :

```
to insert :n :liste
if nulle :liste [op fput :n :liste]
if < :n first :liste [op fput :n :liste]
op fput first :liste insert :n bf :liste
end
```

Le fonctionnement de cette procédure est simple. Si la liste est vide, la procédure renvoie une liste contenant uniquement le nombre à insérer. Sinon, le nombre est comparé au premier élément de la liste. S'il est plus petit, il est inséré en début de liste. Sinon, la procédure renvoie la liste constitue du premier élément de la liste argument et du nombre inséré dans la liste sans le premier élément. Cette procédure est évidemment récursive.

Vérifiez son fonctionnement à l'aide d'un exemple :

```
?insert 4 [1 2 3 5 6 7]
[1 2 3 4 5 6 7]
```

Nous pouvons maintenant définir la procédure **tri**. Son fonctionnement sera très simple. Il suffira en fait d'insérer le premier élément de la liste dans le reste de la liste triée. Cette définition récursive est donnée ci-dessous :

```
to tri :liste
if nulle :liste [op :liste]
op insert first :liste tri bf :liste
end
```

Nous pouvons tester cette procédure :

```
?tri [3 5 2 7 6 4 1 9 8 0]
[0 1 2 3 4 5 6 7 8 9]
```

On remarquera encore une fois l'extrême compacité de cette définition. Bien sûr, cette procédure n'est pas très performante. Il ne tient qu'à vous de l'améliorer.

CHAPITRE VIII

REPERTOIRE DES PRIMITIVES LOGO

Nous donnerons dans ce chapitre une description détaillée des principales primitives Logo disponibles sur les ordinateurs Amstrad. Celles-ci apparaissent ici par ordre alphabétique. Une liste des primitives groupées par types de fonctions est donnée en annexe.

and

Objet : Donne la valeur TRUE (vrai) si tous les arguments sont vrais et la valeur FALSE (faux) si un des arguments au moins est faux.

Syntaxe : and *argument1 argument2*
(and *arg1 arg2 ... argn*)

Commentaire : and peut prendre plus de deux arguments à condition que l'expression soit placée entre parenthèses.

.APV

Objet : Propriété correspondant à la valeur d'une variable globale.

Commentaire : .APV peut être utilisée pour émuler la procédure thing :

```
?to thing :nom
>op gprop :nom ".APV
>end
```

Cette procédure ne fonctionnera que pour des variables globales. .APV utilisée avec une variable locale donnera une valeur nulle.

arctan

Objet : Donne l'arctangente de l'angle argument.

Syntaxe : arctan *angle*

Commentaire : L'*angle* doit être exprimé en degrés.

ascii

Objet : Donne le code ASCII du premier caractère du mot argument.

Syntaxe : ascii *mot*

Commentaire : Le *mot* argument doit contenir au moins un caractère. La liste des codes ascii est donnée en annexe.

bf

Objet : Donne son argument privé de son premier élément.

Syntaxe : bf *liste*
bf *mot*

Commentaire : Si l'argument est une liste, bf donne la liste privée de son premier élément. Si l'argument est un mot, bf donne le mot privé de son premier caractère.

Exemple :
```
?bf [a b c d]
[b c d]
?bf "amstrad
mstrad
```

bk

Objet : Déplace la tortue vers l'arrière.

Syntaxe : bf *distance*

Commentaire : distance est une valeur numérique indiquant le nombre de pas dont la tortue doit se déplacer. bf ne modifie pas l'orientation de la tortue.

bl

Objet : Donne son argument privé de son dernier élément.

Syntaxe : bl *liste*
bl *mot*

Commentaire : Si l'argument est une liste, bl donne la liste privée de son dernier élément. Si l'argument est un mot, bl donne le mot privé de son dernier caractère.

Exemple :
```
?bf [a b c d]
[a b c]
?bf "amstrad
amstra
```

buttonp

Objet : Donne la valeur TRUE si le bouton de la manette de jeu spécifiée comme argument est enfoncé.

Syntaxe : buttonp *numéro-de-manette*

Commentaire : Les valeurs possibles de *numéro-de-manette* sont 0 (première manette de jeu) et 1 (deuxième manette de jeu). Si la manette correspondant au paramètre n'est pas connectée, la valeur retournée par buttonp est FALSE.

bye

Objet : Quitte Dr Logo et retourne sous CP/M.

Syntaxe : bye

Commentaire : Lorsque bye est utilisée, toutes les définitions non sauvegardées sont perdues.

catch

Objet : Détecte les erreurs et certaines conditions particulières.

Syntaxe : catch *nom liste-d'instructions*

Commentaire : catch, utilisée conjointement avec throw, permet d'intercepter certaines erreurs et conditions particulières et d'empêcher l'affichage des messages d'erreur correspondants pour les remplacer par vos propres messages. Pour lier une procédure catch à une procédure throw, leurs arguments *nom* doivent être identiques.

Exemple :

```
?to essai
>catch "bouton [action]
>pr [le bouton est enfoncé]
>end
essai defined
?to action
>if buttonp 0 [throw "bouton]
>action
>end
action defined
```

Dans cet exemple, si la procédure essai est exécutée, le contrôle est passé à la procédure action qui est exécutée jusqu'à ce que le bouton de la première manette de jeu soit enfoncé. A ce moment, le contrôle est rendu à la ligne suivante de la

procédure essai et le message "le bouton est enfoncé" est affiché.

changef

Objet : Change le nom d'un fichier.

Syntaxe : changef *nouveau-nom ancien-nom*

Commentaire : Change le nom du fichier *ancien-nom* en *nouveau-nom*. Le nouveau nom peut être précédé de l'indication d'une unité de disques si le fichier ne se trouve pas dans l'unité par défaut.

char

Objet : Affiche le caractère dont le code ASCII est donné comme argument.

Syntaxe : char *code*

Commentaire : Le code ASCII est donné en annexe.

clean

Objet : Efface l'écran graphique sans affecter la position de la tortue.

Syntaxe : clean

co

Objet : Termine une pause due à une expression pause ou à la frappe de Ctrl-Z.

Syntaxe : co

Commentaire : Permet de relancer un programme interrompu par une procédure pause, la frappe de Ctrl-Z ou une erreur si la variable système ERRACT a la valeur TRUE. Pendant la pause, Logo affiche le message "Pausing...".

.contents

Objet : Affiche le contenu de l'espace de travail de Logo.

Syntaxe : .contents

Commentaire : .contents affiche la liste de tous les mots connus de Logo, c'estàdire les noms des primitives et les noms des procédures qui ont été définies depuis le chargement de Logo.

copyoff

Objet : Supprime l'écho sur l'imprimante.

Syntaxe : copyoff

Commentaire : copyoff est utilisée pour stopper l'envoi à l'imprimante du texte affiché sur l'écran, déclenché par copyon.

copyon

Objet : Déclenche l'écho sur l'imprimante.

Syntaxe : copyon

Commentaire : Si la procédure copyon est utilisée, tout le texte affiché ensuite sur l'écran est également envoyé à l'imprimante.

cos

Objet : Donne le cosinus de son argument.

Syntaxe : cos *angle*

Commentaire : L'argument de cos doit être un angle exprimé en degrés. cos donne un résultat compris entre 0 et 1.

count

Objet : Donne le nombre d'éléments contenus dans son argument.

Syntaxe : count *mot*
count *liste*

Commentaire : Si l'argument est une liste, count donne le nombre d'éléments dans cette liste. Si l'argument est un mot, count donne le nombre de caractères qu'il contient.

Exemple :
```
?count [a b c d e]
5
?count [[a [b c]] [d e]]
2
?count "amstrad
7
```

cs

Objet : Efface l'écran graphique et replace la tortue dans sa position d'origine.

Syntaxe : cs

Commentaire : La position d'origine de la tortue est au centre de l'écran, pointant vers le haut.

ct

Objet : Efface l'écran texte.

Syntaxe : ct

Commentaire : En mode texte, la totalité de l'écran est effacée et le curseur est placé dans l'angle supérieur gauche. En mode split screen, seule la fenêtre du bas (fenêtre texte) est effacée.

cursor

Objet : Donne la position du curseur sur l'écran.

Syntaxe : cursor

Commentaire : cursor donne une liste contenant les numéros de colonne et de ligne où se trouve le curseur. Le premier élément de la liste est le numéro de colonne et le second est le numéro de ligne. Le numéro de ligne est compris entre 0 et 24. Le numéro de colonne est compris entre 0 et 79.

.DEF

Objet : Propriété correspondant à la définition d'une procédure.

Commentaire : Logo garde la trace des procédures définies par l'utilisateur en associant à leur nom une liste contenant leur définition. Cette liste correspond à la propriété .DEF du nom de la procédure.

Exemple : Soit la définition suivante :

```
?to valeur :nom
>op gprop :nom ".APV
>end
```

On peut afficher la valeur de .DEF pour le mot "thing de la façon suivante :

```
?gprop "valeur ".DEF
[[] [op gprop :nom ".APV]]
```

Notons que lorsqu'une procédure précédemment définie est effacée, son nom n'est pas retiré de l'espace de travail de Logo. Seule sa propriété .DEF est supprimée.

defaultd

Objet : Donne le nom de l'unité de disques par défaut.

Syntaxe : defaultd

define

Objet : Définit une procédure.

Syntaxe : define *procédure liste*

Commentaire : define permet à une procédure d'en définir une autre. define prend deux arguments : un nom de procédure et une liste correspondant à la définition de la procédure. L'exemple suivant définit la procédure valeur décrite pour la propriété .DEF (voir ce mot) :

```
define "valeur [[] [op gprop :nom ".APV]]
```

.deposit

Objet : Place une valeur à une adresse.

Syntaxe : .deposit *adresse valeur*

Commentaire : *valeur* doit être comprise entre 0 et 256. La procédure .examine permet de consulter le contenu d'une adresse.

dir

Objet : Affiche le contenu d'un disque.

Syntaxe : dir *nom-de-fichier*

Commentaire : Seuls les fichiers possédant l'extension .LOG sont listés, mais l'extension n'est pas affichée.

dirpic

Objet : Affiche la liste des fichiers images contenus sur un disque.

Syntaxe : dirpic *nom-de-fichier*

Commentaire : Seuls les fichiers .PIC sont listés (voir la procédure savepic).

dot

Objet : Affiche le point dont les coordonnées sont données par la liste argument.

Syntaxe : dot [*abcisse ordonnée*]

Commentaire : Cette procédure n'affecte ni la position ni la direction de la tortue.

ed

Objet : Modification de la définition d'une procédure.

Syntaxe : ed *procédure*
ed *liste-de-procédures*
ed

Commentaire : ed sans argument fait entrer dans l'éditeur avec un écran vierge.

ed *procédure* fait entrer dans l'éditeur avec la définition de la procédure affichée à l'écran.

ed *liste-de-procédures* fait entrer dans l'éditeur avec les définitions des procédures figurant dans la liste affichée à l'écran.

Une fois dans l'éditeur, il est possible de modifier les définitions des procédures. La touche COPY permet de quitter l'éditeur en enregistrant les modifications. La touche ESC permet de quitter l'éditeur sans enregistrer les modifications.

edall

Objet : Permet d'éditer l'ensemble des procédures et des variables définies.

Syntaxe : edall

Commentaire : edall fait entrer dans l'éditeur en affichant toutes les définitions enregistrées depuis le début de la session de travail.

emptyp

Objet : Donne la valeur TRUE si l'argument est un mot ou une liste vide.

Syntaxe : emptyp *objet*

Commentaire : Cette procédure est généralement utilisée pour savoir si tous les éléments d'une liste ont été traités.

end

Objet : Indique la fin de la définition d'une procédure.

Syntaxe : end

Commentaire : L'éditeur ajoute automatiquement le mot end si vous terminez l'édition sans l'avoir placé à la fin de la procédure éditée. Il en est de même pour la procédure define. end n'est pas une primitive Logo.

ent

Objet : Modifie l'enveloppe de tonalité d'un son.

Syntaxe : ent *liste*

Commentaire : Cette primitive est équivalente de la commande BASIC ENT et prend les mêmes arguments.

env

Objet : Modifie l'enveloppe de volume d'un son.

Syntaxe : env *liste*

Commentaire : Voir ent.

er

Objet : Efface une ou plusieurs procédures.

Syntaxe : er *procédure*
er *liste*

Commentaire : Si l'argument est un mot, er efface la procédure correspondante. Si l'argument est une liste, toutes les procédures figurant dans la liste sont effacées. Si l'une des procédures figurant dans la liste n'existe pas, la procédure er n'a aucun effet.

erall

Objet : Efface toutes les procédures définies dans l'espace de travail.

Syntaxe : erall

Commentaire : erall n'efface pas le nom des procédures de l'espace de travail, mais seulement les définitions associées (voir remprop).

erasefile

Objet : Efface un fichier sur disque.

Syntaxe : erasefile *nom-de-fichier*

Commentaire : Un nom d'unité peut précéder le nom de fichier si celui-ci ne se trouve pas sur l'unité par défaut. erasefile affiche un message demandant confirmation avant d'effacer le fichier.

erasepic

Objet : Efface un fichier image.

Syntaxe : erasepic *nom-de-fichier*

ern

Objet : Efface une ou plusieurs variables.

Syntaxe : ern *variable*
ern [*variable1 variable2* ...]

Commentaire : ern efface les variables figurant dans la liste argument. L'espace de travail correspondant est récupéré pour un usage ultérieur. Si une des variables n'existe pas, la procédure s'interrompt sans qu'aucune variable soit effacée.

ERRACT

Objet : Utilisée pour modifier le traitement des erreurs.

Commentaire : La valeur normale de ERRACT est FALSE. Dans ce cas, si une erreur est rencontrée, un message est affiché et la procédure est interrompue définitivement. Si la valeur de ERRACT est modifiée de la façon suivante :

```
make "ERRACT "TRUE
```

le comportement en cas d'erreur est modifié. La procédure est interrompue temporairement et le message "pausing..." est affiché, suivi de l'indication de l'endroit où l'erreur a été rencontrée. Après correction de l'erreur, l'exécution peut être relancée en utilisant la procédure co.

error

Objet : Description d'une erreur.

Syntaxe : error

Commentaire : error affiche une liste décrivant la dernière erreur qui s'est produite. Une liste de description d'erreur non vide comporte six éléments :

1. Un numéro identifiant l'erreur. La liste des messages d'erreur est donnée en annexe.
2. Un message expliquant l'erreur. Ce message est identique à celui qui est normalement affiché sur l'écran.
3. Le nom de la procédure dans laquelle l'erreur s'est produite. Si l'erreur s'est produite au niveau TOPLEVEL, il s'agit d'une liste vide.
4. La ligne dans laquelle l'erreur s'est produite.
5. Le nom de la partie de la procédure où s'est produite l'erreur.
6. L'entrée correspondant à l'expression erronée s'il y a lieu.

En cas d'erreur, si ERRACT a la valeur FALSE, Logo exécute une procédure throw "error. Si aucune procédure catch "error n'a été exécutée, l'exécution s'arrête avec un message d'erreur. Dans le cas contraire, l'exécution se poursuit à la ligne suivant catch "error.

Si ERRACT a la valeur TRUE, Logo exécute une pause pendant laquelle l'erreur peut être corrigée. L'exécution peut alors être relancée à l'aide de la procédure co.

.examine

Objet : Donne le contenu d'une adresse.

Syntaxe : .examine *adresse*

Commentaire : .examine donne le contenu de l'*adresse* argument. (Voir .deposit.)

FALSE

Objet : Valeur logique signifiant "faux".

Commentaire : Le contraire de FALSE est TRUE.

fd

Objet : Déplace la tortue vers l'avant.

Syntaxe : fd *nombre-de-pas*

Commentaire : fd fait avancer la tortue dans la direction vers laquelle elle pointe, d'une distance correspondant à l'argument *nombre-de-pas*.

fence

Objet : Limite le déplacement de la tortue.

Syntaxe : fence

Commentaire : Lorsque fence est utilisée, le déplacement de la tortue est limité par les bords de l'écran. Si une procédure tente de déplacer la tortue hors des limites de l'écran, Logo affiche le message d'erreur "Turtle out of bounds". Voir également window et wrap.

first

Objet : Donne le premier élément de son argument.

Syntaxe : first *mot*
first *liste*

Commentaire : Si l'argument est une liste, first en donne le premier élément. Si l'argument est un mot, first en donne le premier caractère. Si l'argument est vide, Logo affiche un message d'erreur.

fput

Objet : Ajoute son premier argument en tête de son second.

Syntaxe : fput *mot1 mot2*
fput *objet liste*

Commentaire : Si le second argument est un mot, fput donne un mot composé de son premier argument concaténé avec le second. Si le second argument est une liste, fput ajoute son premier argument en tête de cette liste.

Exemple :
```
?fput "a "mstrad
amstrad
?fput "a [b c d e f]
[a b c d e f]
```

```
?fput [a [b]] [[[c] d] e]
[[a [b]] [[c] d] e]
```

fs

Objet : Consacre la totalité de l'écran au graphisme.

Syntaxe : fs

glist

Objet : Affiche tous les objets ayant une certaine propriété.

Syntaxe : glist *propriété*

Commentaire : glist donne la liste de tous les objets se trouvant dans l'espace de travail et possédant la propriété citée comme argument.

Exemple :

```
glist ".DEF
[essai carre]
```

go

Objet : Interrompt l'exécution séquentielle d'une procédure.

Syntaxe : go *étiquette*

Commentaire : go exécute la ligne suivant immédiatement l'étiquette argument.

Exemple :

```
?to dernier2 :liste
>if nulle :liste [op :liste]
>label "a
>if nulle bf :liste [op first :liste]
```

```
>make "liste bf :liste
>go "a
>end
```

gprop

Objet : Donne une propriété d'un objet.

Syntaxe : gprop *nom propriété*

Commentaire : gprop donne une des propriétés de son premier argument. Cette propriété est déterminée par le second argument.

Exemple :
```
?gprop "dernier2 ".DEF
[[:liste] [if nulle :liste [op :liste]] [label "a] [if nulle
bf :liste [op first :liste]] [make "liste bf :liste] [go
"a]]
```

home

Objet : Replace la tortue à sa position d'origine.

Syntaxe : home

Commentaire : La position d'origine de la tortue est le centre de l'écran, pointant vers le haut. home n'affecte ni l'écran ni la position du crayon (pen).

ht

Objet : Rend la tortue invisible.

Syntaxe : ht

Commentaire : ht rend la tortue invisible mais n'affecte ni sa position, ni sa direction, ni l'état du crayon, ni l'écran.

if

Objet : Test.

Syntaxe : if *condition liste1 liste2*

Commentaire : Si la valeur du premier argument est TRUE, la première liste d'instructions est exécutée. Si la valeur de la condition est FALSE, la seconde liste d'instructions est exécutée.

int

Objet : Donne la partie entière d'un nombre.

Syntaxe : int *nombre*

Commentaire : int donne la partie entière d'un nombre par troncature. Pour obtenir une valeur arrondie, utiliser round.

item

Objet : Donne le $n^{ième}$ élément d'un liste ou d'un mot.

Syntaxe : item *n mot*
item *n liste*

Commentaire : Le premier argument est une valeur entière. Si le second argument est un mot, item donne le $n^{ième}$ caractère de ce mot. Si le second argument est une liste, item donne le $n^{ième}$ élément de cette liste.

Exemple :

```
?item 3 "amstrad
s
?item 3 [a b c d e]
c
?item 3 [[a [b]] c [d] [e f]]
[d]
```

keyp

Objet : Indique si un caractère a été tapé au clavier.

Syntaxe : keyp

Commentaire : Donne la valeur TRUE si un caractère a été tapé au clavier et est prêt à être lu. Ce caractère peut alors être lu par la procédure rc.

label

Objet : Déclare une étiquette.

Syntaxe : label *étiquette*

Commentaire : label identifie une ligne qui sera exécutée après la procédure go *étiquette* correspondante.

last

Objet : Donne le dernier élément d'un objet.

Syntaxe : last *mot*
last *liste*

Commentaire : Si l'argument est une liste, last donne le dernier élément de cette liste. Si l'argument est un mot, last en donne le dernier caractère.

Exemple :
```
?last "amstrad
d
?last [a b c d e f]
f
```

lc

Objet : Conversion en minuscule.

Syntaxe : lc *mot*

Commentaire : lc convertit son argument (qui doit être un mot) en minuscules. lc est utilisée principalement dans les tests de réponse d'un utilisateur.

Exemple :
```
?to question
>pr [Aimez vous les ordinateurs ?]
>if "o = lc first rq
>  [pr [Moi aussi !!!]]
>  [pr [Moi non plus !!!]]
>end
question defined
?question
Aimez vous les ordinateurs ?
OUI
Moi aussi !!!
?question
Aimez vous les ordinateurs ?
pas vraiment
Moi non plus !!!
```

list

Objet : Donne une liste composée de ses arguments.

Syntaxe : list *arg1 arg2 ..... argn*

Commentaire : Si list a un ou plus de deux arguments, list et ses arguments doivent être mis entre parenthèses.

Exemple :
```
?list "a "b
[a b]
?(list "a "b "c
[a b c]
```

listp

Objet : Donne la valeur logique TRUE si son argument est une liste.

Syntaxe : listp *argument*

Exemple :
```
?listp "amstrad
FALSE
?listp [a b c d e f]
TRUE
```

load

Objet : Charge un programme.

Syntaxe : load *nom-de-fichier*

Commentaire : L'argument doit être le nom d'un fichier ASCII CP/M présent sur l'unité de disques. L'indication de l'unité de disques peut précéder le nom du fichier s'il ne s'agit pas de l'unité par défaut. Le fichier doit posséder l'extension .LOG. Les procédures contenues dans le fichier sont définies au moment du chargement et remplacent éventuellement les procédures de même nom se trouvant précédemment en mémoire.

local

Objet : Rend la variable argument accessible uniquement dans la procédure où figure la déclaration locale.

Syntaxe : local *variable*
(local *variable1 variable2 ... variablen*)

Commentaire : Si local a plusieurs arguments, l'expression entière doit être entre parenthèses. Une variable déclarée locale n'est accessible que dans la procédure où figure la déclaration. Les variables locales peuvent porter le même nom que des variables globales sans affecter celles-ci.

lput

Objet : Place le premier argument à la fin du second.

Syntaxe : lput *mot1 mot2*
lput *mot liste*

Commentaire : Si le second argument est une liste, lput donne une liste constituée de son second argument auquel a été ajouté son premier argument comme dernier élément. Si le second argument est un mot, lput donne un mot constitué de son second argument suivi de son premier argument.

Exemple :
```
?lput "d "amstra
amstrad
?lput "e [a b c d]
[a b c d e]
```

lt

Objet : Fait tourner la tortue vers la gauche.

Syntaxe : lt *angle*

Commentaire : lt fait tourner la tortue vers la gauche de la valeur de son argument exprimé en degrés. lt n'affecte ni la position de la tortue, ni l'état du crayon, ni le contenu de l'écran.

make

Objet : Donne une valeur à une variable.

Syntaxe : make *variable objet*

Commentaire : make fait de son second argument la valeur de son premier argument. Le premier argument doit être un mot. Le second argument peut être une liste ou un mot. Il est important de noter que si le second argument est un mot, celui-ci peut avoir lui même une valeur.

Exemple :

```
?make "a [a b c d e]
?:a
[a b c d e]
?make "b 2
?:b
2
?make "a "b
?:b
2
?:a
b
```

menberp

Objet : Donne la valeur logique TRUE si son premier argument est élément de son second argument.

Syntaxe : memberp *objet1 objet2*

Commentaire : Le second élément peut être une liste ou un mot. S'il s'agit d'un mot, le premier élément doit lui-même être un mot. Dans le cas contraire, un message d'erreur est affiché.

Exemple :
```
?memberp "stra "amstrad
TRUE
?memberp "c [a b c d e]
TRUE
?memberp "b [a [b c] d e]
FALSE
?memberp [b c] [a [b c] d e]
TRUE
?memberp [b c] [a b c d e]
FALSE
?memberp [a] "amstrad
memberp does'nt like "amstrad as input
```

namep

Objet : Donne la valeur logique TRUE si son argument est une variable définie.

Syntaxe : namep *mot*

Commentaire : L'argument de namep doit obligatoirement être un mot.

nodes

Objet : Affiche le nombre de noeuds libres dans l'espace de travail.

Syntaxe : nodes

Commentaire : Un noeud correspond à cinq octets. node peut être utilisée régulièrement pour vérifier que la mé-

moire n'est pas saturée. L'espace libéré peut être réutilisé grâce à la procédure recycle.

noformat

Objet : Retire les commentaires de l'espace de travail.

Syntaxe : noformat

Commentaire : Les définitions de procédures peuvent être accompagnées de commentaires commençant par un point-virgule et se terminant par une fin de ligne. Ces commentaires sont utiles dans les fichiers de programmes mais consomment inutilement de l'espace mémoire. La procédure noformat permet de supprimer ces commentaires en mémoire. Les commentaires figurant dans le fichier programme ne sont pas affectés.

not

Objet : Négation logique.

Syntaxe : not *expression-logique*

Commentaire : Si la valeur de l'expression logique argument est TRUE, not donne la valeur FALSE. Si la valeur de l'argument est FALSE, not donne la valeur TRUE

notrace

Objet : Désactive le mode de mise au point trace.

Syntaxe : notrace

Commentaire : notrace supprime l'effet de la procédure trace.

nowatch

Objet : Désactive le mode de mise au point watch.

Syntaxe : nowatch

Commentaire : nowatch supprime l'effet de la procédure watch.

numberp

Objet : Donne la valeur logique TRUE si son argument est un nombre.

Syntaxe : numberp *objet*

op

Objet : Fait de son argument le résultat de la procédure.

Syntaxe : op *objet*

Commentaire : op transmet le résultat d'une procédure à la procédure appelante (éventuellement à la procédure TOPLEVEL). L'exécution d'une procédure s'interrompt dès qu'un résultat a été transmis à la procédure appelante.

or

Objet : Donne la valeur logique TRUE si au moins un de ses arguments a la valeur TRUE.

Syntaxe : or *arg1 arg2*
(or *arg1 arg2 ... argn*)

Commentaire : Si or compte un ou plus de deux arguments, l'ensemble doit être placé entre parenthèses.

paddle

Objet : Donne la position des manettes de jeu.

Syntaxe : paddle *numéro-de-manette*

Commentaire : Le numéro de manette peut être 0 (première manette) ou 1 (deuxième manette). La valeur retournée est 255 si la manette est verticale. Dans le cas contraire, l'angle d'orientation de la manette est obtenu en multipliant le résultat de la procédure paddle par 45 (en degrés).

pal

Objet : Donne une liste représentant la palette des couleurs.

Syntaxe : pal *n*

Commentaire : Voir setpal

pause

Objet : Interrompt l'exécution du programme pour permettre la mise au point.

Syntaxe : pause

Commentaire : Lorsque Logo rencontre une procédure pause, il interrompt l'exécution du programme et affiche le message "Pausing...". L'exécution peut être reprise grâce à la procédure co.

pd

Objet : Abaisse le crayon de la tortue.

Syntaxe : pd

Commentaire : pd abaisse le crayon de la tortue de façon que ses déplacements ultérieurs laissent une trace sur l'écran.

pe

Objet : Donne au crayon la couleur du fond.

Syntaxe : pe

Commentaire : Après exécution de pe, le crayon de la tortue écrit dans la couleur du fond et efface donc sur son passage ce qui se trouve affiché sur l'écran.

piece

Objet : Donne une partie d'un objet.

Syntaxe : piece *nombre1 nombre2 objet*

Commentaire : piece donne une liste composée des éléments *nombre1* à *nombre2* de son troisième argument si celui-ci est une liste. Si le troisième argument est un mot, piece donne un mot composé des caractères *nombre1* à *nombre2*.

Exemple :
```
?piece 3 6 "amstrad
stra
?piece 3 6 [a m s t r a d]
[s t r a]
```

plist

Objet : Donne la liste des propriétés de son argument.

Syntaxe : plist *mot*

Exemple :
```
?make "a 2
?plist "a
[.APV 2]
```

po

Objet : Affiche la définition de son argument s'il s'agit d'un nom de procédure.

Syntaxe : po *procédure*
po *liste-de-procédures*

Commentaire : po permet d'afficher la définition d'une procédure ou d'une liste de procédures.

poall

Objet : Affiche les définitions de toutes les procédures se trouvant dans l'espace de travail.

Syntaxe : poall

pops

Objet : Affiche les noms et définitions de toutes les procédures se trouvant dans l'espace de travail.

Syntaxe : pops

pots

Objet : Affiche la liste des noms de procédures définies dans l'espace de travail.

Syntaxe : pots

Commentaire : pots affiche la première ligne de chaque définition et permet donc de consulter les noms de variables utilisés.

pprop

Objet : Attribue une valeur pour une propriété à un objet.

Syntaxe : pprop *nom propriété objet*

Commentaire : pprop permet d'attribuer une valeur à une des propriétés standard de Logo :

```
?pprop "a ".APV 2
?:a
2
```

L'exemple ci-dessus est identique à la forme suivante :

```
?make "a 2
```

pprop permet également d'ajouter de nouvelles propriétés à la liste de propriétés standard. Pour supprimer une propriété ajoutée à la liste standard, utiliser la procédure remprop.

pps

Objet : Donne la liste des propriétés non standard.

Syntaxe : pps

Commentaire : pps donne la liste, pour tous les objets se trouvant dans l'espace de travail, des propriétés non standard et de leurs valeurs.

pr

Objet : Affiche son ou ses arguments.

Syntaxe : pr *objet*
(pr *liste-d'objets*)

Commentaire : Si pr a plusieurs arguments, l'ensemble doit être placé entre parenthèses.

Exemple :
```
?make "variable "trois
?(pr [Vous avez gagné] :variable "francs)
Vous avez gagné trois francs
```

.PRM

Objet : Propriété standard correspondant à l'adresse d'implantation en mémoire d'une primitive.

Exemple :
```
gprop "make ".PRM
glist ".PRM
```

Commentaire : Le premier exemple ci-dessus permet d'afficher l'adresse à laquelle commence le code correspondant à la primitive make. Le second exemple affiche la liste de toutes les primitives Logo.

pu

Objet : Lève le crayon de la tortue.

Syntaxe : pu

Commentaire : pu lève le crayon de sorte que la tortue puisse être déplacée sans laisser de trace sur l'écran. pu n'affecte ni l'orientation ni la position de la tortue, ni ce qui est affiché sur l'écran.

px

Objet : Fait passer le crayon en mode inverse.

Syntaxe : px

Commentaire : Lorsque la procédure px est utilisée, la tortue laisse un trace là où rien n'est affiché et efface ce qui est affiché.

quotient

Objet : Donne le quotient de ses deux arguments.

Syntaxe : quotient *dividende diviseur*

Commentaire : Le résultat est entier (obtenu par troncature).

Exemple :

```
?quotient 10 4
2
?10/4
2.5
?quotient 5 0
Can't divide by zero
?quotient 5 .9
Can't divide by zero
```

random

Objet : Donne un nombre aléatoire.

Syntaxe : random *argument*

Commentaire : random donne un nombre positif ou nul strictement inférieur à l'argument. L'argument doit être une valeur numérique.

rc

Objet : Lit un caractère au clavier.

Syntaxe : rc

Commentaire : La procédure rc donne le premier caractère tapé au clavier. Tout les caractères sont passés normalement à l'exception de Ctrl-Z qui cause une pause et Ctrl-G qui arrête l'exécution. Si aucun caractère n'est prêt, rc attend qu'un caractère soit tapé. Le caractère n'est pas affiché sur l'écran (voir keyp).

recycle

Objet : Réorganise l'espace de travail.

Syntaxe : recycle

Commentaire : recycle permet de récupérer l'espace de travail libéré grâce à une opération connue sous le nom évocateur de "garbage collection". L'espace regagné peut être contrôlé grâce à la procédure nodes.

Exemple :

```
?nodes
512
?recycle
?nodes
3458
```

REDEFP +

Objet : Permet de redéfinir des primitives.

Commentaire : REDEPF est un objet Logo qui peut prendre les valeurs TRUE ou FALSE. Lorsque sa valeur est TRUE, il est possible de redéfinir les primitives de Logo. Ceci est une manipulation dangereuse et il est conseillé de sauvegarder l'espace de travail avant de s'y aventurer. Une fois qu'une primitive a été redéfinie, il n'existe aucun moyen de retrouver la définition d'origine.

release

Objet : Libère les canaux sonores.

Syntaxe : release *n*

Commentaire : *n* indique les canaux de la façon suivante :

bit 1 = canal A
bit 2 = canal B
bit 3 = canal C

5 désigne donc par exemple les canaux A et C.

remainder

Objet : Donne le reste d'une division.

Syntaxe : remainder *dividende diviseur*

Commentaire : remainder donne le reste de la division de son premier argument par le second.

remprop

Objet : Supprime une propriété d'un objet.

Syntaxe : remprop *nom propriété*

Commentaire : remprop permet de supprimer une propriété ayant été placée dans la liste de propriétés d'un objet à l'aide de la procédure pprop. Si toutes les propriétés d'un objet sont supprimées, cet objet est lui-même retiré de l'espace de travail.

repeat

Objet : Répète une liste d'instructions.

Syntaxe : repeat *nombre liste-d'instruction*

Commentaire : Le premier argument doit être une valeur numérique positive. S'il ne s'agit pas d'un entier, l'argument est tronqué.

rerandom

Objet : Réinitialise le générateur de nombres aléatoires.

Syntaxe : rerandom

Commentaire : rerandom permet de reproduire plusieurs fois la même séquence de nombres pseudo-aléatoires.

rl

Objet : Lit une liste au clavier.

Syntaxe : rl

Commentaire : rl donne une ligne tapée au clavier dès que la touche Return est frappée. La ligne est affichée sur l'écran. Si aucun caractère n'a été frappé, rl attend jusqu'à ce que la touche Return soit frappée. La procédure keyp peut être utilisée pour déterminer si quelque chose a été tapé.

rq

Objet : Lit un mot au clavier.

Syntaxe : rq

Commentaire : rq donne sous forme d'un mot une ligne tapée au clavier dès que la touche Return est frappée. La ligne est affichée sur l'écran. Si aucun caractère n'a été frappé, rq attend jusqu'à ce que la touche Return soit frappée. La procédure keyp peut être utilisée pour déterminer si quelque chose a été tapé. Si la ligne comporte des espaces, ceux-ci sont automatiquement précédés du caractère \ pour indiquer qu'ils doivent être traités comme des caractères et non comme des séparateurs.

round

Objet : Donne son argument arrondi.

Syntaxe : round *n*

Commentaire : round arrondit son argument à la valeur entière la plus proche. Si la partie fractionnaire est supérieure ou égale à 0.5, round donne la valeur entière immédiatement supérieure. Dans le cas contraire, round donne la valeur entière immédiatement inférieure.

rt

Objet : Fait tourner la tortue vers la droite.

Syntaxe : rt *angle*

Commentaire : rt fait tourner la tortue vers la droite d'un nombre de degrés égal à la valeur de son argument. Si son argument est négatif, la tortue tourne vers la gauche.

run

Objet : Exécute une liste d'instructions.

Syntaxe : run *liste*

Commentaire : run exécute la liste d'instructions fournie comme paramètre. Si cette liste d'instructions donne une valeur, run donne cette valeur.

save

Objet : sauvegarde un fichier.

Syntaxe : save *nom-de-fichier*

Commentaire : save écrit le contenu de l'espace de travail dans un fichier sur disque. Si le fichier existe, le message "File already exists." est affiché. Il faut alors supprimer le fichier existant avec la procédure erasefile pour pouvoir sauvegarder le nouveau.

savepic

Objet : Sauvegarde une image.

Syntaxe : savepic *nom-de-fichier*

Commentaire : savepic permet de sauvegarder une image écran qui peut ensuite être rechargée par loadpic. Le fichier ne doit pas déjà exister. Les fichiers images sont sauvegardés avec l'extension .PIC.

se

Objet : Donne une liste composée de ses arguments.

Syntaxe : se *objet1 objet2*
(se *liste-d'objets*)

Commentaire : se est identique à list à la différence que se retire les crochets intérieurs. Si se a un ou plus de deux arguments, l'expression entière doit être placée entre parenthèses.

Exemple :
```
?se [a b c d] [e f g h]
[a b c d e f g h]
?se [a [b [c [d]]]] [[[[e] f] g] h]
[a [b [c [d]]] [[[e] f] g] h]
```

setcursor

Objet : Déplace le curseur.

Syntaxe : setcursor [*colonne ligne*]

Commentaire : Le numéro de colonne doit être compris entre 0 et 79. Le numéro de ligne doit être compris entre 0 et 24.

setd

Objet : Change l'unité par défaut.

Syntaxe : setd *unité:*

Commentaire : L'unité spécifiée par le paramètre devient l'unité par défaut.

seth

Objet : Oriente la tortue.

Syntaxe : seth *direction*

Commentaire : seth oriente la tortue en fonction de la valeur du paramètre. Le paramètre doit être une valeur numérique en degrés. 0 correspond à la tortue dirigée vers le haut. Une valeur positive fait tourner la tortue vers la droite. Une valeur négative la fait tourner vers la gauche.

setpal

Objet : Modifie les couleurs des crayons..

Syntaxe : setpal *numéro-de-crayon liste*

Commentaire : setpal prend deux paramètres. Le premier est le numéro de crayon auquel s'applique la modification. Le second paramètre est une liste de trois valeurs numériques. Ces trois valeurs doivent être comprises entre 0 et 2 et indiquent respectivement les quantités de rouge, de vert et de bleu qui doivent composer la couleur.

setpc

Objet : Change le crayon utilisé par la tortue.

Syntaxe : setpc *numéro*

Commentaire : setpc indique le crayon qui sera utilisé par la tortue. Il existe quatre crayons numérotés de 0 à 3. Le crayon 0 correspond à la couleur du fond.

setpos

Objet : Déplace la tortue en coordonnées absolues.

Syntaxe : setpos [*x y*]

Commentaire : La tortue va se placer aux coordonnées x,y. Si le crayon est baissé, une ligne est tracée de la position de départ à la position d'arrivée. x doit être compris entre -150 et +149. y doit être compris entre -99 et +100.

setsplit

Objet : Fixe le nombre de lignes de texte en mode splitscreen.

Syntaxe : setsplit *n*

Commentaire : *n* indique le nombre de lignes que doit contenir la fenêtre de texte en mode splitscreen.

setx

Objet : Déplace la tortue horizontalement.

Syntaxe : setx *n*

Commentaire : *n* indique la nouvelle position horizontale de la tortue et doit être compris entre -150 et +149. La direction de la tortue n'est pas modifiée.

sety

Objet : Déplace la tortue verticalement.

Syntaxe : sety *n*

Commentaire : *n* indique la nouvelle postion verticale de la tortue et doit être compris entre -99 et +100

sf

Objet : Affiche des informations sur l'état de l'écran.

Syntaxe : sf

Commentaire : sf fournit une liste de cinq éléments. Le premier indique la couleur du fond et vaut toujours 0. Le deuxième indique l'état de l'écran qui peut être TS, SS ou FS. Le troisième indique le nombre de lignes de texte. Le quatrième indique le type de limites de l'écran : WINDOW, WRAP ou FENCE. Le cinquième indique le rapport entre la hauteur et la largeur de l'écran et vaut toujours 1.

show

Objet : Affiche un objet sur l'écran.

Syntaxe : show *objet*
(show *objet1 objet2 ... objetn)*

Commentaire : show affiche son argument sur l'écran. Si show a plusieurs arguments, l'expression entière doit être placée entre parenthèses.

shuffle

Objet : Place les éléments d'une liste en ordre aléatoire.

Syntaxe : shuffle *liste*

Commentaire : shuffle donne une liste dont les éléments sont les mêmes que ceux de la liste argument mais sont placés en ordre aléatoire.

sin

Objet : Donne le sinus d'un angle.

Syntaxe : sin *angle*

Commentaire : L'angle doit être exprimé en degrés. sin donne une valeur décimale comprise entre 0 et 1.

ss

Objet : Fait passer en mode splitscreen.

Syntaxe : ss

Commentaire : ss partage l'écran en deux parties : la partie du haut pour le graphisme et la partie du bas pour le texte. La procédure setsplit peut être utilisée pour fixer le nombre de lignes affichées dans la fenêtre de texte.

st

Objet : Fait apparaître la tortue.

Syntaxe : st

Commentaire : Lorsque la procédure st est utilisée, la tortue devient visible sur l'écran. L'exécution des graphiques est ralentie. La procédure ht est utilisée pour cacher la tortue.

stop

Objet : Interrompt une procédure.

Syntaxe : stop

Commentaire : stop interrompt la procédure en cours d'exécution et repasse le contrôle à la procédure TOPLEVEL ou à la procédure appelante.

text

Objet : Donne la définition d'une procédure.

Syntaxe : text *nom-de-procédure.*

tf

Objet : Donne des informations sur l'état de la tortue.

Syntaxe : tf

Commentaire : tf donne une liste de six éléments indiquant l'état de la tortue : (1) la coordonnée x ; (2) la coordonnée y ; (3) la direction de la tortue ; (4) l'état du crayon (PD =bas et PU = haut) ; (5) le numéro de crayon utilisé ; (6) TRUE si la tortue est visible et FALSE dans le cas contraire.

thing

Objet : Donne le contenu d'une variable.

Syntaxe : thing *nom-de-variable*

Commentaire : thing *nom-de-variable* est équivalent à *:nom-de-variable.*

throw

Voir catch

to

Objet : Indique le début d'une définition de procédure.

Syntaxe : to *nom-de-procédure <définition>*

Commentaire : to n'est pas une procédure mais un mot réservé indiquant le début d'une définition de procédure. to ne fait pas partie de la définition de la procédure, pas plus que end qui signale la fin de la définition.

TOPLEVEL

Commentaire : TOPLEVEL est un mot réservé qui peut être utilisé avec throw (throw TOPLEVEL) pour repasser le contrôle à Logo dans certaines conditions. (Voir catch.)

towards

Objet : Donne la direction que doit prendre la tortue pour pointer vers un point donné.

Syntaxe : towards [*x y*]

Commentaire : x et y sont les coordonnées du point vers lequel la tortue doit pointer.

trace

Objet : Active le mode "mise au point".

Syntaxe : trace *liste-de-procédures*

Commentaire : Lorsque la procédure trace est utilisée, les noms des procédures qui doivent être tracées sont affichés lorsqu'elles sont exécutées, ainsi que les noms et valeurs de toutes les variables qu'elles utilisent. Le niveau et le nombre de procédures exécutées depuis le début sont également affichés.

TRUE

Objet : Valeur logique signifiant "vrai".

ts

Objet : Sélectionne l'écran texte.

Syntaxe : ts

Commentaire : Lorsque cette procédure est exécutée, la totalité de l'écran est consacrée à l'affichage du texte.

tt

Objet : Affiche du texte sur l'écran graphique.

Syntaxe : tt *objet*
(tt *objet1 objet2 ... objetn*)

Commentaire : tt affiche son argument sur l'écran graphique à la position de la tortue. Si l'argument est une liste, les crochets extérieurs sont supprimés. Si la procédure a plusieurs arguments, l'expression entière doit être entre parenthèses.

type

Objet : Affichage sans passage à la ligne.

Syntaxe : tt *objet*
(tt *objet1 objet2 ... objetn*)

Commentaire : tt permet d'afficher un objet sur l'écran sans exécuter un passage à la ligne. Si tt a plusieurs arguments, l'expression complète doit être placée entre parenthèses. Dans ce cas, les arguments sont affichés sur la même ligne.

uc

Objet : Conversion en majuscules.

Syntaxe : uc *mot*

Commentaire : La procédure uc effectue l'opération inverse de celle réalisée par la procédure lc. Les caractères accentués ne sont pas convertis.

wait

Objet : Attente.

Syntaxe : wait *durée*

Commentaire : wait permet d'interrompre le déroulement d'un programme pour une durée spécifiée par l'argument. L'argument doit être une valeur numérique indiquant la durée de l'attente en 1/60 de seconde.

watch

Objet : Mise au point de programmes.

Syntaxe : watch *liste-de-procédures*

Commentaire : Lorsque la procédure watch est utilisée, le nom de chaque procédure est affiché avant qu'elle soit exécutée. Logo observe alors une pause pendant laquelle il est possible d'examiner le contenu des variables. L'exécution reprend lorsque la touche Return est frappée.

where

Objet : Détermine la position d'un élément dans une liste.

Syntaxe : where

Commentaire : where ne prend pas d'argument. Cette procédure indique la position du premier argument de la dernière procédure memberp exécutée, dans son deuxième argument.

Exemple :
```
?memberp "c [a b c d e f]
TRUE
```

```
?where
3
```

window

Objet : Permet à la tortue de sortir des limites de l'écran.

Syntaxe : window

Commentaire : En mode Window, la tortue peut sortir des limites de l'écran sans produire de message d'erreur.

word

Objet : Concaténation de mots.

Syntaxe : word *mot1 mot2*
(word *mot1 mot2 ... motn*)

Commentaire : Si word a plus de deux arguments, l'expression complète doit être placée entre parenthèses. (Si aucune autre expression ne suit sur la même ligne, la parenthèse droite peut être omise.)

Exemple :

```
?(word "am "str "ad
amstrad
```

wordp

Objet : Indique si son argument est un mot.

Syntaxe : wordp *objet*

Commentaire : wordp donne la valeur TRUE si son argument est un mot et la valeur FALSE dans le cas contraire.

wrap

Objet : Modifie le mode d'affichage graphique.

Syntaxe : wrap

Commentaire : En mode Wrap, lorsque la tortue disparaît d'un côté de l'écran, elle réapparaît du côté opposé.

+

Objet : Opérateur d'addition.

Syntaxe : valeur1 + valeur2
(+ valeur1 valeur2 ... valeurn)

Commentaire : L'opérateur + peut être utilisé en notation infixée ou en notation préfixée. La notation préfixée permet d'utiliser un nombre quelconque d'arguments. Dans ce cas, l'expression doit être placée entre parenthèses.

Objet : Opérateur de soustraction.

Syntaxe : valeur1 - valeur2
- valeur1 valeur2

Commentaire : L'opérateur - peut être utilisé en notation infixée ou en notation préfixée.

*

Objet : Opérateur de multiplication.

Syntaxe : valeur1 * valeur2
(* valeur1 valeur2 ... valeurn)

Commentaire : L'opérateur * peut être utilisé en notation infixée ou en notation préfixée. La notation préfixée permet d'utiliser un nombre quelconque d'arguments. Dans ce cas, l'expression doit être placée entre parenthèses.

Objet : Opérateur de division.

Syntaxe : valeur1 / valeur2
/ valeur1 valeur2

Commentaire : L'opérateur + peut être utilisé en notation infixée ou en notation préfixée.

=

Objet : Opérateur relationnel "égal".

Syntaxe : objet1 = objet2
= objet1 objet2

Commentaire : Cet opérateur peut être utilisé en notation infixée ou en notation préfixée.

<

Objet : Opérateur relationnel "plus petit que"

Syntaxe : objet1 < objet2
< objet1 objet2

Commentaire : Cet opérateur peut être utilisé en notation infixée ou en notation préfixée.

>

Objet : Opérateur relationnel "plus grand que".

Syntaxe : objet1 > objet2
> objet1 objet2

Commentaire : Cet opérateur peut être utilisé en notation infixée ou en notation préfixée.

ANNEXE A

Liste des primitives par type de fonction

Traitement de mots et de listes

ascii
bf
bl
char
count
emptyp
first
fput
item
last
list
listp
lc
lput
memberp
numberp
piece
se
shuffle
uc
where
word
wordp

Opérations logiques

and
not
or
=
<
>

Opérations arithmétiques

arctn
cos
int
quotient
random
remainder
rerandom
round
sin
+
-
*
/

Variables

local
make
namep
thing

Définition de procédures

define
end
text
to

Edition

ed
edall

Ecran texte

ct
cursor
pr
setcursor
show
ts
type

Ecran graphique

clean
cs
dot
fence
fs
pal
pd
pe
pu
px
setpal
setpc
setsplit
sf
ss
tf
tt
window
wrap

Tortue

bk
fd
ht
home
lt
rt
seth
setpos
setx
sety
st
towards

Gestion de l'espace de travail

.contents
.deposit
er
erall
ern
.examine
nodes
noformat
po
poall
pops
pots
recycle

Listes de propriétés

glist
gprop
plist
pprop
pps
remprop

Disques et fichiers

changef
defaultd
dir
dirpic
erasefile
erasepic
load
loadpic
save
savepic
setd

Clavier

keyp
rc
rl
rq

Imprimante

copyoff
copyon

Manettes de jeu et son

buttonp
ent
env
paddle
release

Contrôle de l'exécution

bye
co
go
if
label
repeat
run
stop
wait

Gestion d'erreur et mise au point

catch
error
notrace
nowatch
pause
throw
trace
watch

ANNEXE B

LES MESSAGES D'ERREUR

2 Number too big

Le nombre utilisé est trop grand.

6 (symbole) is a primitive

Le symbole utilisé est une primitive (mot réservé).

7 Can't find label (symbole)

L'étiquette utilisée n'a pu être trouvée.

8 Can't (symbole) from the editor

La procédure ne peut être exécutée à partir de l'éditeur.

9 (symbole) is undefined

symbole non défini.

11 I'm having trouble with the disk

Problème d'accès disque.

12 Disk full

Disque plein.

13 Can't divide by zero

Division par zéro impossible.

15 File already exists

Le fichier spécifié existe déjà.

17 File not found

Fichier non trouvé.

21 Can't find catch for (symbole)

Pas d'instruction catch correspondant au symbole utilisé.

23 Out of space

Espace mémoire insuffisant.

25 (symbole) is not true or false

Le symbole utilisé n'a pas une valeur logique.

29 Not enough inputs to (procédure)

Nombre d'entrées insuffisant pour la procédure utilisée.

30 Too many inputs to (procédure)

Entrées trop nombreuses pour la procédure utilisée.

32 Too few items in (liste)

Nombre d'éléments dans la liste insuffisant.

34 Turtle out of bounds

Tortue hors limites.

35 I don't know how to (symbole)

Le symbole utilisé n'est pas une procédure.

36 (symbole) has no value

Le symbole utilisé n'a pas de valeur.

37)without(

Une parenthèse droite ne correspond à aucune parenthèse gauche.

38 I don't know what to do with (symbol)

Une procédure a donné un résultat non traité.

40 Disk is write protected

Le disque est protégé.

41 (procédure) doesn't like (symbol) as input

L'entrée ne convient pas à la procédure utilisée.

42 (procédure) didn't output

Un résultat était attendu d'une procédure qui n'en a pas fourni.

44 !!! LOGO system bug !!!

Erreur système de Logo.

45 The word is too long

Le mot utilisé est trop long.

46 I don't have enough buffer space

Espace insuffisant pour le tampon.

47 If wants [ ]'s around instruction list

Crochet manquant autour des listes d'instructions dans une instruction IF.

48 *Variable selon le type d'erreur disque.*

49 (symbole) isn't a parameter

Le symbole utilisé n'est pas un paramètre.

51 The file is write protected

Le fichier est protégé.

52 I can't find the disk drive

L'unité de disques est inaccessible.

ANNEXE C

LES CARACTERES DE CONTROLE

Ctrl-A

Place le curseur au début de la ligne.

Ctrl-B

Déplace le curseur d'un caractère vers la gauche.

Ctrl-C

Sort de l'éditeur.

Ctrl-D

Efface un caractère à la position du curseur.

Ctrl-E

Place le curseur à la fin de la ligne.

Ctrl-F

Déplace le curseur d'un caractère vers la droite.

Ctrl-G

Interrompt l'exécution ou l'édition.

Ctrl-H

Efface un caractère à gauche du curseur.

Ctrl-I

Tabulation.

Ctrl-K

Efface de la position du curseur jusqu'à la fin de la ligne. Le texte effacé est placé dans un tampon.

Ctrl-L

Place le curseur au milieu de l'écran.

Ctrl-M

Equivalent de la touche Return.

Ctrl-N

Déplace le curseur d'une ligne vers le bas.

Ctrl-O

Le curseur reste sur la même ligne. La fin de la ligne est décalée d'une ligne vers le bas.

Ctrl-P

Déplace le curseur d'une ligne vers le haut.

Ctrl-Q

Equivalent de \. Le caractère suivant est pris litérallement.

Ctrl-R

Place le curseur au début du texte.

Ctrl-S

Passe en mode Splitscreen.

Ctrl-T

Passe en mode Texte.

Ctrl-U

Déplace le curseur d'un écran vers le haut.

Ctrl-V

Déplace le curseur d'un écran vers le bas.

Ctrl-W

Arrête le défilement du texte. Le défilement reprend après la frappe d'une touche quelconque.

Ctrl-X

Place le curseur à la fin du texte.

Ctrl-Y

Fait réapparaître la dernière ligne effacée ou la dernière ligne tapée.

Ctrl-Z

Arrête la procédure en cours d'exécution.

ANNEXE D

LE CODE ASCII

Caractère ASCII	Valeur décimale	Valeur hexadécimale	Caractère de contrôle	Signification
NUL	0	00	^@	Caractère nul
SOH	1	01	^A	Curseur en début de ligne
STX	2	02	^B	Curseur vers la gauche
ETX	3	03	^C	Sortie de l'éditeur
EOT	4	04	^D	Effacement d'un caractère à la position du curseur
ENQ	5	05	^E	Curseur en fin de ligne
ACK	6	06	^F	Curseur vers la droite
BEL	7	07	^G	Interruption d'exécution ou d'édition
BS	8	08	^H	Effacement d'un caractère à gauche du curseur
HT	9	09	^I	Tabulation
LF	10	0A	^J	
VT	11	0B	^K	Effacement de fin de ligne
FF	12	0C	^L	Curseur au milieu de l'écran
CR	13	0D	^M	Retour chariot
SO	14	0E	^N	Curseur vers le bas
SI	15	0F	^O	Décalage du texte d'une ligne vers le bas
DLE	16	10	^P	Curseur vers le haut
DC1	17	11	^Q	Equivalent de \
DC2	18	12	^R	Curseur en début de texte
DC3	19	13	^S	Passage en mode Splitscreen
DC4	20	14	^T	Passage en mode Texte
NAK	21	15	^U	Défilement vers le haut
SYN	22	16	^V	Défilement vers le bas
ETB	23	17	^W	Arrêt du défilement
CAN	24	18	^X	Curseur en fin de texte

Caractère ASCII	Valeur décimale	Valeur hexadécimale	Caractère de contrôle	Signification
EM	25	19	^Y	Réinsère le texte effacé
SUB	26	1A	^Z	Arrêt d'une procédure
ESC	27	1B	^[	
FS	28	1C	^\	
GS	29	1D	^]	
RS	30	1E	^^	
US	31	1F	^_	
SP	32	20		Espace
!	33	21		
"	34	22		
#	35	23		
$	36	24		
%	37	25		
&	38	26		
'	39	27		Apostrophe
(	40	28		
)	41	29		
*	42	2A		
+	43	2B		
,	44	2C		Virgule
-	45	2D		Signe moins
.	46	2E		Point
/	47	2F		
0	48	30		
1	49	31		
2	50	32		
3	51	33		
4	52	34		
5	53	35		
6	54	36		
7	55	37		
8	56	38		
9	57	39		
:	58	3A		
;	59	3B		
<	60	3C		

Caractère ASCII	Valeur décimale	Valeur hexadécimale	Caractère de contrôle	Signification
=	61	3D		
>	62	3E		
?	63	3F		
@	64	40		
A	65	41		
B	66	42		
C	67	43		
D	68	44		
E	69	45		
F	70	46		
G	71	47		
H	72	48		
I	73	49		
J	74	4A		
K	75	4B		
L	76	4C		
M	77	4D		
N	78	4E		
O	79	4F		
P	80	50		
Q	81	51		
R	82	52		
S	83	53		
T	84	54		
U	85	55		
V	86	56		
W	87	57		
X	88	58		
Y	89	59		
Z	90	5A		
[	91	5B		
\	92	5C		
]	93	5D		
^	94	5E		
_	95	5F		Souligné
`	96	60		Accent grave

Caractère ASCII	Valeur décimale	Valeur hexadécimale	Caractère de contrôle	Signification
a	97	61		
b	98	62		
c	99	63		
d	100	64		
e	101	65		
f	102	66		
g	103	67		
h	104	68		
i	105	69		
j	106	6A		
k	107	6B		
l	108	6C		
m	109	6D		
n	110	6E		
o	111	6F		
p	112	70		
q	113	71		
r	114	72		
s	115	73		
t	116	74		
u	117	75		
v	118	76		
w	119	77		
x	120	78		
y	121	79		
z	122	7A		
{	123	7B		
|	124	7C		
}	125	7D		
~	126	7E		
DEL	127	7F		

Achevé d'imprimer le 18 novembre 1986 sur les presses de l'Imprimerie «La Source d'Or»
63200 Marsat - Dépôt légal : 4e trimestre 1986 - Imprimeur n° 2156

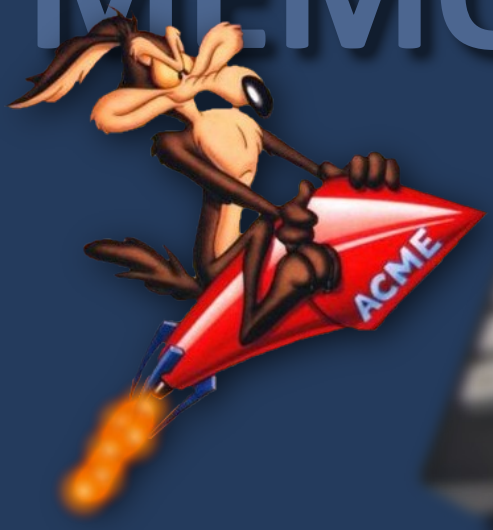
Dérivé de Lisp, le langage le plus utilisé pour l'intelligence artificielle, Logo a été créé pour faciliter l'apprentissage de la programmation. De ce fait, il a été souvent considéré comme un langage pour enfant, ceci étant encore accentué par l'existence de nombreuses instructions graphiques et de la fameuse tortue. La plupart des livres publiés sur Logo sont des recueils de procédures graphiques qui font peut-être la joie des enfants (?) mais sont vite lassantes pour qui veut progresser dans l'apprentissage de la programmation. Nous avons donc choisi une approche totalement différente en essayant de montrer les aspects les plus fondamentaux de Logo. Nous avons essayé de faire comprendre au lecteur la structure de Logo plutôt que de donner de trop nombreux exemples qui tendent à stimuler seulement les facultés d'imitation du lecteur. On trouvera au Chapitre 8 une liste détaillée des primitives Logo disponibles sur les Amstrad, ce qui permettra au lecteur d'aller plus loin dans la programmation en palliant l'absence de toute documentation livrée avec la machine.

0171 1186 128 F
9 782736 101718

Document numérisé avec amour par :

AMSTRAD CPC

MÉMOIRE ÉCRITE



https://acpc.me/