Manuel de référence MSWLogo et UCBLogo

Procédures primitives principales en anglais et en français

Francis Leboutte

Première parution: mars 2001
Dernière mise à jour: 10 novembre 2003
Dernière mise à jour mineure: 18 octobre 2004
www.algo.be/logo.html

IDDN.BE.010.0093308.000.R.C.2001.035.42000

Table des matières

1 Introduction

1.1 Terminologie

Note : ce paragraphe fait suite au document ***Introduction à la programmation en Logo et MSWLogo*** (voir www.algo.be/logo1/logo-primer-fr.html).

`avance 50` est une ***instruction,*** c'est-à-dire quelque chose que le Logo est capable d'interpréter comme un ordre, en l'occurrence *fais avancer la tortue de 50 pas*.

Les mots qui comme `avance, droite` et `somme` font partie du vocabulaire du langage Logo sont des ***noms de procédures***. Une procédure est comme une recette décrivant une tâche à accomplir. Par exemple la procédure qui a pour nom `avance` a pour tâche de faire avancer la tortue. Une procédure ***primitive*** est une procédure qui fait partie du langage Logo tel qu'il est mis initialement à votre disposition par un environnement de programmation Logo comme ***MSWLogo*** (dans MSWLogo, il y a environ 200 primitives). Les procédures *non* primitives sont celles que vous définissez vous-même.

Toute procédure Logo, qu'elle soit primitive ou non, est soit une ***opération***, c'est-à-dire une procédure qui calcule et retourne une valeur à exploiter (par exemple **`somme`**) , soit une ***commande***, c'est-à-dire une procédure qui ne retourne pas de valeur (par exemple **`avance`**).

1.2 Syntaxe

En Logo il n'y a qu'**une seule règle syntaxique générale** décrivant comment appliquer (utiliser) une procédure dans une instruction : c'est celle de la notation ***préfixe***, c'est-à-dire une notation où le nom de la procédure apparaît en premier suivi ensuite de ses arguments. Cette règle est valable pour toutes les procédures, pour les procédures primitives comme pour celles que vous définissez vous-même. Exemples:

```
somme 1 3.6
(somme 1 2 3 4 5)
liste 100 200
premier [1 2]
baissecrayon
```

Notez que cette notation s'accommode d'un nombre quelconque d'arguments, zéro, un ou plusieurs arguments.

Il y a une exception à cette règle : lorsqu'on veut utiliser les abréviations des procédures arithmétiques (+, - , * , /), on doit utiliser la notation ***infixe***, où la procédure est mise entre **deux** arguments; par exemple:

```
1.4 + 3.6
```

Notez les différences:

- Le ***nombre d'espaces*** autour du signe + ne joue pas (il peut être zéro ou plus)
- Après le mot somme, il faut ***au moins un espace***, sinon, comme par exemple dans *(somme2 3 4 5)* , l'évaluateur Logo considère que *somme2* est un nom de procédure.

Dans `(somme 1 2 3 4 5)`, il *faut* mettre des ***parenthèses*** car le nombre d'arguments n'est pas celui requis par défaut (2 dans le cas de *somme* qui est une procédure qui accepte un nombre quelconque d'arguments) . Il est *permis* de mettre des parenthèses autour de toute application de

procédure, c'est-à-dire autour d'un nom de procédure et de ses arguments: en général, ceci est fait afin d'améliorer la lisibilité d'une instruction.

1.3 Conventions

Dans ce document, quelques symboles et mots ont une signification particulière:

n désigne un nombre réel sauf mention particulière (exemple : -20.5)

truc désigne un mot ou une liste

True.false désigne un mot qui doit être le mot *true* (vrai) ou le mot *false* (faux).

faire : est un raccourci pour dire *écrire ce qui suit (une instruction) dans la ligne de commande et l'exécuter*, c'est-à-dire, *donner ce qui suit au Logo pour évaluation.*

-> Dans un exemple d'application d'une procédure opération, sépare ce qui est donné à l'évaluateur Logo, c'est-à-dire une instruction (à gauche de la flèche), du résultat de l'exécution de l'opération, c'est-à-dire la valeur retournée par l'opération (à droite de la flèche). Si l'instruction est tapée dans la ligne de commande de MSWLogo (voir le chapitre *Présentation de MSWLogo*), ne pas oublier de faire précéder l'instruction de la commande *montre* (*show* en anglais; celle-ci demande au Logo d'afficher le résultat dans la fenêtre texte). Au lieu de `montre` vous pouvez aussi utiliser la commande `écris` (`print` en anglais).
Exemple :

```
somme 1.4 11  ->  12.4
```

Dans la ligne de commande de MSWLogo, écrivez :

```
montre somme 1.4 11
```

Ensuite, poussez le bouton *Execute* pour exécuter l'instruction et voir le résultat (*12.4*) affiché dans l'historique.

>> Permet de représenter une interaction avec le Logo via la ligne de commande (voir le chapitre *Présentation de MSWLogo*). Le symbole >> est utilisé pour indiquer le résultat de l'exécution d'une instruction, quand il y en a un (une valeur ou un message).
Exemple:

```
montre somme 1.4 11
>> 12.4
```

***** indique une procédure quasi-primitive, c'est-à-dire une extension du langage Logo définie via le fichier de démarrage startup.lgo (voir www.algo.be/logo.html)

1.4 Traduction du vocabulaire MSWLogo et UCBLogo en français

A chaque nom de primitive MSWLogo (UCBLogo) en français décrite dans ce manuel, il correspond l'original en anglais (en italique). Par exemple:

avance n	**AV**
forward n	*FO*

n : nombre de pas de la tortue…

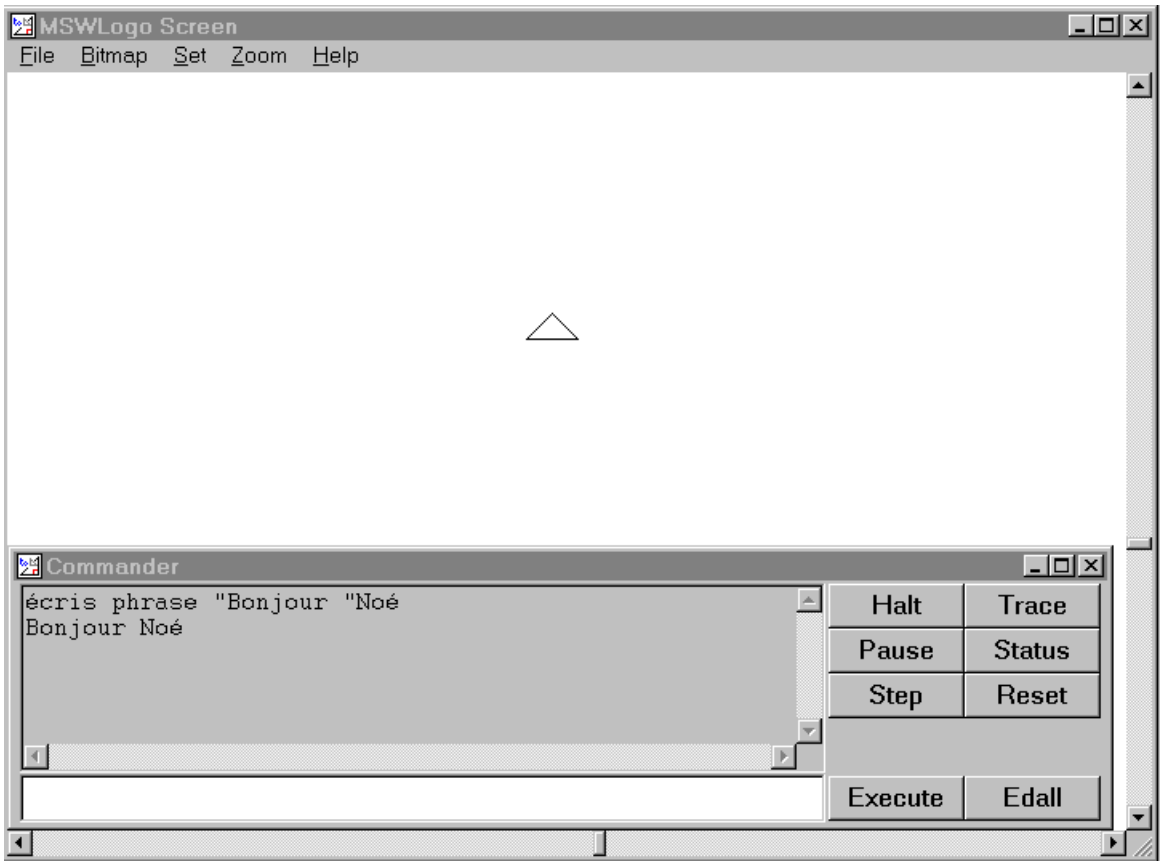
La traduction de *forward* est donc *avance*, son abréviation *av* (l'abréviation anglaise étant *fo*).

Pour ***installer la traduction française*** de MSWLogo et UCBLogo , voir www.algo.be/logo.html

1.5 Présentation de MSWLogo

MSWLogo est un logiciel libre écrit par George Mills et distribué par Softronix, dont l'adresse est www.softronix.com. Le cœur de MSWLogo a été développé par Brian Harvey et ses étudiants de l'université de Californie à Berkeley (Berkeley Logo ou UCBLogo). Ce qui fait que le contenu de ***ce document s'applique aussi à UCBLogo***, à l'exception de ce qui est spécifique de Microsoft Windows (la description de l'interface de MSWLogo de ce chapitre).

Voici comment se présente l'interface de MSWLogo:



Il y a deux fenêtres principales: la **fenêtre graphique** (ou **écran** - *MSWLogo Screen*) et la **fenêtre texte** *(Commander)*.

La fenêtre graphique est celle où se déplace la tortue (représentée par un triangle).

La fenêtre texte comprend plusieurs éléments:

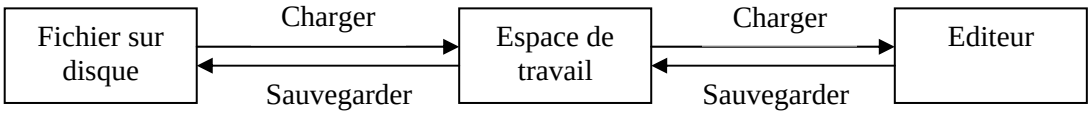
- la **ligne de commande** (ou ligne de saisie - c'est la boîte dont le fond est de couleur blanche) : c'est là que vous tapez une instruction comme par exemple `forward 50`
- l'**historique**, qui comme son nom l'indique contient la trace de toute votre activité. Dans la figure ci-dessus, il contient 2 lignes correspondant à l'exécution d'une instruction et au résultat de cette exécution:
  écris phrase "Bonjour "Noé : l'instruction qui a été tapée dans la ligne de commande
  Bonjour Noé : le résultat de l'exécution de cette instruction
- Différents boutons:
  - **Halt** : pour arrêter l'exécution d'un programme
  - **Pause** : pour interrompre momentanément l'exécution d'un programme
  - **Step** et **Trace** : pour la mise au point d'un programme

- **Reset** : pousser ce bouton est équivalent à l'exécution de la commande `nettoietout` (ou `nt` en abrégé, `clearscreen` en anglais)
- **Execute** : permet d'exécuter l'instruction qui a été tapée dans la ligne de commande
- **Edall** : pour faire apparaître l'**éditeur Logo**.

Trucs de la ligne de commande :

- Pousser la touche du clavier *Entrer* est équivalent à pousser le bouton *Execute.*
- Pour rappeler une instruction présente dans l'historique, cliquez dans l'historique sur la ligne qui contient l'instruction en question : l'instruction réapparaît dans la ligne de commande où vous pouvez la modifier avant de la réexécuter.
- Pour réexécuter une instruction présente dans l'historique, cliquez rapidement *deux fois* dans l'historique sur la ligne qui contient l'instruction en question.
- Cliquer via le bouton de droite de la souris dans la ligne de commande fait apparaître un menu d'édition (copier, coller, etc.)

L'éditeur de MSWLogo et l'espace de travail



L'éditeur Logo est activé en poussant le **bouton Edall (***Edit all procedures*, éditer toutes les procédures) ou en tapant la **commande Edall** dans la ligne de commande. Il est constitué d'une seule fenêtre dans laquelle vous éditez toutes vos propres définitions de procédure. C'est un éditeur de texte semblable au *bloc-notes* (un éditeur de texte pur).

Lorsque vous avez terminer votre travail d'édition et désirez que vos modifications soient prises en compte par le Logo, allez dans le menu *File* (*Fichier*) de l'*éditeur* et choisissez une des 3 commandes :

- *Save and exit* ce qui signifie : sauvegarder (mettre à jour l'espace de travail) et quitter l'éditeur)
- *Save to workspace* (mettre à jour l'espace de travail – sans fermer l'éditeur). Pour pouvoir utiliser cette commande, MSWLogo doit être démarré en mode expert à l'aide de l'option de démarrage ***–e*** . Le champ cible du raccourci MSWLogo sera alors quelque chose comme : *c:\mswlogo\logo32.exe -e -l startup.lgo*
- *Exit* (quitter l'éditeur) ; ensuite répondez *Yes* au dialogue suivant (*Contents have changed. Save to workspace?* - ce qui veut dire : le contenu a changé. Mettre à jour l'espace de travail ?).

Notez que ces commandes ne sauvent pas votre travail sur le disque dur : elles ne font que garder en mémoire votre travail et transmettre l'information à l'interpréteur Logo. L'***espace de travail*** (***workspace***) désigne l'espace mémoire dans lequel sont stockés vos procédures Logo. **L'éditeur Logo permet d'éditer l'espace de travail.**

Si vous désirez que vos modifications ne soient pas prises en compte par le Logo et définitivement écartées, allez dans le menu *File* de l'*éditeur*, choisissez *Exit* et ensuite répondez *No* au dialogue *Contents have changed. Save to workspace?*

Note à propos de la syntaxe du Logo

- Il est permis de mettre plusieurs instructions sur une même ligne

- Une instruction peut être écrite sur plusieurs lignes si :
  - elle apparaît dans une liste d'instructions (entre des crochets)
  - elle est mise entre parenthèses
  - le caractère de continuation (le tilde ~) est utilisé

Sauvegarder dans un fichier sur le disque dur

Pour sauvegarder votre travail (vos définitions de procédure) dans un fichier sur le disque dur, aller dans le menu *File* de la fenêtre graphique (*MSWLogo Screen*), choisissez la commande *Save*. Lors d'une prochaine session MSWLogo vous pourrez alors charger vos procédures via la commande *Load* du même menu *File* de la fenêtre graphique. La commande *Load* permet d'ajouter les procédures d'un fichier dans l'espace de travail (chargement ou restauration de procédures).

Messages d'erreur

> Note : une traduction des messages d'erreurs en français est disponible dans UCBLogo (pas dans MSWLogo à ce jour), voir www.algo.be/logo.html

Pour rappel, le symbole >> est utilisé pour indiquer le résultat de l'exécution d'une instruction, quand il y en a un (ce résultat est une valeur ou un message) ; dans MSWLogo, ce résultat est affiché par le Logo dans l'historique de la fenêtre texte.

```
somme 1 2
>> You don't say what to do with 3
```

Signifie *Vous ne dites pas quoi faire avec 3* . Dans l'instruction, il manque la commande `écris` pour indiquer à l'évaluateur Logo qu'il doit donner *3* (la valeur retournée par l'exécution de la procédure `somme`) à la commande `écris` (pour que 3 soit affiché dans l'historique).
Il faut toujours spécifier quoi faire avec une valeur retournée par une procédure du type opération :

```
écris somme 1 2
>> 3


écris somme 1
>> not enough inputs to somme
```

Signifie *somme n'a pas assez d'arguments*. Normalement `somme` reçoit 2 arguments
Il faut toujours donner à une procédure le nombre d'arguments requis. Sinon, si la procédure permet un nombre d'argument variable, utilisez des parenthèses :

```
écris (somme 1)
>> 1


écris somme 1 2 6
>> 3
>> You don't say what to do with 6
```

Dans ce cas, un argument de trop a été donné à la procédure `somme`. Cependant, dans son processus d'évaluation de l'instruction, le Logo lit l'instruction de gauche à droite: il commence par appliquer `somme` aux 2 arguments 1 et 2; ensuite il transmet le résultat 3 à la procédure

`écris` et exécute `écris 3`, ce qui fait que 3 est affiché dans l'historique. Ensuite le Logo tombe sur la valeur 6 orpheline, d'où le message d'erreur similaire au premier exemple.

```
écris bonjour
>> I don't know how  to bonjour
```

Signifie *Je ne sais pas comment faire bonjour*. Le Logo interprète `bonjour` comme le nom d'une procédure (qu'il ne connaît pas). Dans ce cas pour dire au Logo qu'il doit considérer le mot `bonjour` comme un simple mot et non pas comme le nom d'une procédure, il faut le faire précéder du caractère " (guillemet anglais - *quotation mark* ou *quote* en anglais) :

```
écris "bonjour
>> bonjour


montre :longueur
>> longueur has no value
```

Signifie que la variable *longeur* n'a pas de valeur.

2 Structures de données

Les 2 principaux types de donnée Logo sont les mots[1] et les listes[2]. Les nombres ne sont qu'un cas particulier des mots. Par exemple *bonjour* dans cet exemple :

```
écris "bonjour
>> bonjour
```

Notez qu'il faut faire précéder *bonjour* du caractère " (guillemet anglais - *quotation mark* ou *quote* en anglais) sinon le Logo évalue *bonjour* comme le nom d'une procédure (qu'il ne connaît pas), au lieu d'un mot à prendre pour lui-même. On dit que le caractère " empêche le mot *bonjour* d'être évalué. Sinon, on a une erreur de procédure inconnue :

```
écris bonjour
>> I don't know how  to bonjour
```

Pour imprimer plusieurs mots en même temps, on peut les inclure dans une liste, en les mettant entre des crochets :

```
écris [bonjour Noé]
>> bonjour Noé

écris [bonjour Noé, comment vas-tu ?]
>> bonjour Noé, comment vas-tu ?
```

Deux mots jouent un rôle particulier, *true* et *false* (vrai et faux)[3]. Ils représentent les deux valeurs de vérité vrai et faux, voir les chapitres *Prédicats*, *Opérations logiques* et *Structures de contrôle*.

Note : pour plus d'information sur l'évaluation, voir le *Fonctionnement de l'évaluateur Logo* dans Une Introduction à la programmation en Logo

[1] N'est pas équivalent au type *chaîne de caractères* d'autres langages
[2] Il y a aussi les tableaux (*array*)
[3] Ils ne sont pas traduits dans l'adaptation française de MSWLogo, mais bien dans celle d'UCBLogo

2.1 Procédures de création

liste truc1 truc2
(liste truc1 truc2 truc3 …)
list truc1 truc2
(list truc1 truc2 truc3 …)
Retourne une liste faite des éléments *truc1* et *truc2* (ou *truc1 truc2 truc3* …) ; *truc* est un mot ou une liste.

```
liste 10 11                -> [10 11]
liste "Léonie "Waha        -> [Léonie Waha]
(liste "Léonie "de "Waha)  -> [Léonie de Waha]
liste "Léonie [de Waha]    -> [Léonie [de Waha]]
```

phrase truc1 truc2 **PH**
(phrase truc1 truc2 truc3 …)
sentence truc1 truc2 *SE*
(sentence truc1 truc2 truc3 …)
Retourne une liste dont les éléments sont les arguments *truc1* et *truc2* (ou *truc1 truc2 truc3* …) si ces arguments sont des mots (tout argument peut être un mot ou une liste). Si un argument est une liste, chacun des éléments de la liste devient un élément de la liste résultat.
Dans les 3 premiers exemples *sentence* se comporte exactement comme *list* étant donné que les arguments sont des mots. Par contre observez la différence pour le 4ème.

```
phrase 10 11                -> [10 11]
phrase "Léonie "Waha        -> [Léonie Waha]
(phrase "Léonie "de "Waha)  -> [Léonie de Waha]
phrase "Léonie [de Waha]    -> [Léonie de Waha]
```

metspremier truc liste **MP**
fput truc list
Retourne une liste faite du second argument *liste* plus le premier argument (*truc*) ajouté en premier.

```
metspremier 0 [1 2 3]  ->  [0 1 2 3]
```

metsdernier truc liste **MD**
lput truc list
Retourne une liste faite du second argument *liste* plus le premier argument ajouté en dernier.

```
metsdernier 4 [1 2 3]  ->  [1 2 3 4]
```

inverse liste
reverse list
Retourne une liste dont les éléments sont ceux de l'argument dans l'ordre inverse

```
inverse [1 2 3]  ->  [3 2 1]
```

2.2 Procédures d'accès

premier truc
first truc
Si *truc* est une liste, retourne le premier élément de la liste.
Si *truc* est un mot, retourne le premier caractère du mot.

```
premier 1 2 3]  ->  1
premier "first   ->  f
```

dernier truc
last truc
Si *truc* est une liste, retourne le dernier élément de la liste.
Si *truc* est un mot, retourne le dernier caractère du mot.

```
dernier [1 2 3]  ->  3
dernier "first   ->  t
```

saufpremier truc **SP**
butfirst truc *BF*
Si *truc* est une liste, retourne une liste identique moins le premier élément de la liste.
Si *truc* est un mot, retourne un mot identique moins le premier caractère du mot.

```
saufpremier [1 2 3]  ->  [2 3]
saufpremier "first   ->  irst
```

saufdernier truc **SD**
butlast truc *BL*
Si *truc* est une liste, retourne une liste identique moins le dernier élément de la liste.
Si *truc* est un mot, retourne un mot identique moins le dernier caractère du mot.

```
saufdernier [1 2 3]  ->  [1 2]
saufdernier "first ->  firs
```

item index truc
Si *truc* est une liste, retourne l'élément qui occupe la *index* ème place dans la liste.
Si *truc* est un mot, retourne le caractère qui occupe la *index* ème place dans le mot.
La numérotation commence à 1 : par exemple, dans le cas d'une liste, le premier élément correspond à l'index 1, le dernier élément à un index égal à la longueur de la liste (c'est-à-dire le nombre d'éléments dans la liste).

```
item 4 [11 12 13 14 15]  ->  14
item 1 "Léonie ->  L
item 6 "Léonie ->  e
item 1 + hasard 6 "Léonie  ->  …   (un des caractères au hasard)
```

enlève truc list
remove truc list
Retourne une liste faite du second argument *list* moins tous les éléments identiques au premier argument *truc*.

```
enlève 0 [0 1 2 0 3 0 0]  ->  [1 2 3]
```

choix truc
pick truc
Retourne un élément de *truc* pris au hasard; *truc* est un mot ou une liste.

```
choix "waha         ->  …   (une des lettres au hasard)
choix [1 2 3 4 5]   ->  …   (un des chiffres au hasard)
```

2.3 Prédicats

Un prédicat est une procédure qui retourne toujours un des deux mots *true* (vrai) ou *false* (faux).

mot? truc
wordp truc
word? truc
Retourne *true* si l'argument est un mot, *false* sinon

```
mot? "Waha  ->  true
```

nombre? truc
numberp truc
number? truc
Retourne *true* si l'argument est un nombre, *false* sinon
```
nombre? 1.23  ->  true
```

liste? truc
listp truc
list? truc

Retourne *true* si l'argument est un mot, *false* sinon
```
liste? [1 2]  ->  true
liste? "waha  ->  false
```

vide? truc
emptyp truc
empty? truc

Retourne *true* si l'argument est la liste ou le mot vide
```
vide? "   ->  true
vide? []  ->  true
```

égal? truc1 truc2
equalp truc1 truc2 =
equal? truc1 truc2
Retourne *true* si les 2 arguments sont égaux
```
égal? [1 2] [1 2]  ->  true
[1 2] = [1 2]       ->  true
égal? [1 2] [1]    ->  false
1 = 1               ->  true
1.1 = 1.1           ->  true
1.1 = 1             ->  false
"Léonie = "Léonie   ->  true
"Léonie = "Léoni    ->  false
```

précède? mot1 mot2
beforep mot1 mot2
before? mot1 mot2
Retourne *true* si *mot1* précède *mot2* dans l'ordre alphabétique. La casse ne joue pas.
```
précède? "aaa "abc  ->  true
précède? "aaa "aaa  ->  false
```

membre? truc1 truc2
memberp truc1 truc2
member? truc1 truc2
Si *truc2* est une liste retourne *true* si *truc1* est une élément de *truc2*, *false* sinon.
Si *truc2* est un mot retourne *true* si *truc1* est un mot d'un caractère égal à un des caractères de *truc2*, *false* sinon. La casse ne joue pas.

```
membre? "l [l é o n i e]  ->  true
membre? "l "Léonie        ->  true
```

2.4 Divers

compte truc
count truc
Retourne le nombre d'éléments dans l'argument si celui-ci est une liste. Si c'est un mot retourne le nombre de caractères.

```
compte "Léonie        ->  6
compte [l é o n i e]  ->  6
```

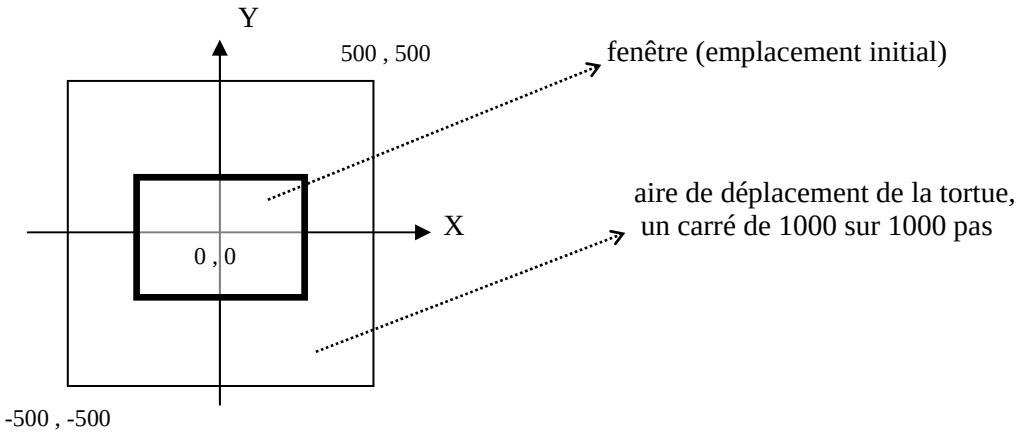
membre truc1 truc2
member truc1 truc2
Si t*ruc2* est une liste et si *truc1* est un élément de *truc2*, retourne une liste égale à la sous-liste de *truc2* à partir de la 1ère occurrence de *truc1*. Sinon retourne *false*.
Si t*ruc2* est un mot, si *truc1* est un mot d'un caractère et si ce caractère est présent dans *truc2*, retourne un mot égal au sous-mot de *truc2* à partir de la 1ère occurrence de *truc1*. Sinon retourne *false*.

```
membre "n [l é o n i e]  ->  [n i e]
membre "n "Léonie        ->  nie
```

3 Graphisme

3.1 Déplacement de la tortue

Voir aussi le chapitre ci-dessous, *Information à propos de la tortue*.



Note : si un déplacement de la tortue la fait atteindre la limite de l'aire de déplacement, elle « disparaît » et continue son déplacement en réapparaissant à l'opposé, en gardant son cap.

avance n **AV**
forward n *FD*
n : nombre de pas de la tortue
Fait avancer la tortue de n pas, selon l'orientation courante

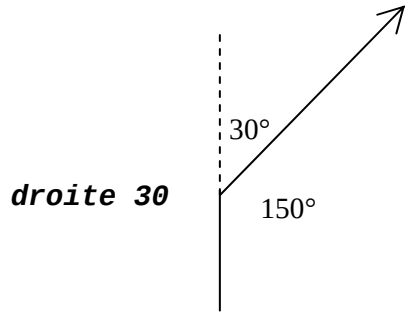
```
avance 90
```

recule n **RE**
back n *BK*
n : nombre de pas de la tortue
Fait reculer la tortue de n pas

```
recule 90
```

droite n **DR**
right n *RT*
n : nombre de degrés (1/360 d'un cercle)
Fait tourner la tortue de n degrés vers la droite (dans le sens des aiguilles d'une montre)

```
droite 30
```



gauche n **GA**
left n *LT*
n : nombre de degrés

Fait tourner la tortue de n degrés vers la gauche

```
gauche 10
```

origine
home
Remet la tortue en son état d'origine: place la tortue au centre de l'écran et réoriente la tortue vers le haut, c'est-à-dire avec un cap de 0 degré.
Equivalent à *fixepos [0 0]* suivi de *fixecap 0*

fixepos pos
setpos pos
Déplace la tortue à la position spécifiée. Une position est une liste de 2 nombres, les coordonnées X et Y

```
fixepos [100 300]
```

fixex n
setx n
Déplace la tortue horizontalement à la nouvelle coordonnée n sur l'axe des X (axe horizontal)

```
fixex 100
```

fixey n
fixey n
Déplace la tortue verticalement à la nouvelle coordonnée n sur l'axe des Y (axe vertical)

```
fixey 300
```

fixexy n1 n2
setxy n1 n2
Déplace la tortue à la position [n1 n2]

```
fixexy 100 300
```

fixecap n
setheading n *SETH*
Oriente la tortue au cap spécifié. L'argument n (le cap) est un nombre de degrés compté à partir de l'axe des Y, dans le sens des aiguilles d'une montre

```
fixecap 60
```

3.2 Dessin de courbes

cercle r
circle r
Dessine un cercle de rayon *r*, dont le centre est la position courante de la tortue. La tortue ne bouge pas.

cercle2 r
circle2 r
Dessine un cercle de rayon *r*, en commançant là où est la tortue et selon son cap, dans le sens des aiguilles d'une montre. La tortue ne bouge pas.

arc angle r
Dessine un arc de rayon *r* centré sur la position courante de la tortue. L'arc commence à l'arrière de la tortue et balaye un angle *angle* dans le sens des aiguilles d'une montre. La tortue ne bouge pas.

arc2 angle r
Dessine un arc de rayon *r* et d'amplitude *angle*, en commançant là où est la tortue et selon son cap, dans le sens des aiguilles d'une montre. La tortue parcourt l'arc.

ellipse mini maxi
Dessine une ellipse selon le cap de la tortue. La position courante de la tortue est le centre de la tortue. *Mini* est la distance minimum entre le centre et l'ellipse. *Maxi* est la distance maximum entre le centre et l'ellipse. La tortue ne bouge pas.

ellipse2 mini maxi
Comme *ellipse*, mais en commançant là où est la tortue.

3.3 Contrôle de la tortue et de l'écran

montretortue **MTO**
showturtle *ST*
rend la tortue visible

cachetortue **CTO**
hideturtle *HT*
rend la tortue invisible

nettoie
clean
efface l'écran - la tortue ne bouge pas

nettoietout **NT**
clearscreen *CS*
Efface l'écran, déplace la tortue au centre de l'écran et réoriente la tortue vers le haut. Equivalent à *nettoie* suivi de *origine* ; ou à pousser le bouton *Reset* dans MSWLogo.

étiquette truc
label truc
Ecrit truc dans la fenêtre graphique, là où se trouve la tortue et selon son orientation. *truc* peut-être un mot ou une liste

```
étiquette 111
étiquette "waha
étiquette [sons et couleurs [rouge vert bleu]]
```

3.4 Information à propos de la tortue

pos
En Logo, une position est une liste de 2 nombres, les coordonnées x et y. Elle représente les coordonnées rectangulaires (ou cartésiennes) de la position de la tortue (x étant l'abscisse et y l'ordonnée).
L'origine des axes de coordonnées est le centre de l'écran graphique (là ou se trouve la tortue au départ ou après une commande *origine*).
Voir aussi plus haut, le diagramme du chapitre *Déplacement de la tortue*.

```
nettoietout
montre pos -> [0 0]
avance 100
montre pos -> [0 100]
droite 90
avance 100
montre pos -> [100 100]
recule –200
montre pos -> [-100 100]
```

cap
heading
Retourne le cap courant de la tortue. Compté en degrés à partir de l'axe des Y, dans le sens des aiguilles d'une montre

```
nt
cap  ->  0
droite 30
montre cap -> 30
dr 30
montre cap -> 60
dr 30
montre cap -> 90
```

3.5 Crayon et couleurs

Dans MSWLogo une couleur est représentée par une liste des 3 intensités des couleurs de base (RGB : red , green, blue) ou *triplet* RGB. Une intensité étant un entier compris entre 0 et 255. Par exemple, la couleur rouge est le triplet [255 0 0] , la couleur verte le triplet [0 255 0] , etc

baissecrayon **BC**
pendown *PD*
Abaisse le crayon de la tortue

lèvecrayon **LC**
penup *PU*
Lève le crayon

gomme **GO**
penerase *PE*
abaisse le crayon et le met en mode effacement

dessine **DE**
penpaint *PPT*
abaisse le crayon et le met en mode dessin

inversecrayon **IC**
penreverse *PX*
abaisse le crayon et le met en mode inverse

couleurcrayon **CC**
pencolor *PC*
Retourne la couleur du crayon sous forme d'une liste des 3 intensités des couleurs de base (RGB : red , green, blue) ou *triplet* RGB. Une intensité étant un entier compris entre 0 et 255. Par exemple, si la couleur courante est vert pur :

```
couleurcrayon -> [0 255 0]
```

fixecouleurcrayon couleur **FCC**
setpencolor color *SETPC*
Change la couleur du crayon. L'argument couleur est une liste des 3 intensités des couleurs de base (RGB : red , green, blue) ou *triplet* RGB. Une intensité étant un entier compris entre 0 et 255. Par exemple pour la couleur rouge :

```
fixecouleurcrayon [255 0 0]
```

crayon
pen
Retourne l'état du crayon. Voir exemple de fixecrayon

fixecrayon état.crayon
setpen pen.state
Fixe l'état du crayon.

```
pour étoile5double :distance
  donnelocale "état.crayon.initial crayon
  fixetaillecrayon  5
  étoile5 :distance
  fixetaillecrayon  2
  gomme
  étoile5 :distance
  fixecrayon :état.crayon.initial
end

pour étoile5 :distance
 ; 2 tours complets (2 fois 360 degrés), en 5 fois (5 branches)
 répète 5 [avance :distance droite 144]
end
```

* opérateurs de couleur

Un certain nombre de couleurs sont définies comme des opérateurs via le fichier de démarrage (startup.lgo) ou *couleurs prédéfinies* : **rouge, vert, bleu, brun, jaune, orange, cyan, magenta, gris, noir, blanc**...

```
montre rouge
>> [255 0 0]
```

Ce qui permet d'écrire :

```
fixecouleurcrayon rouge
```

Pour voir la liste complète des couleurs prédéfinies faire **montre.couleurs**

Les couleurs sont de plus classées par catégories (**rouges, verts, bleus, noirs, blancs, grisés, cyans, bruns, oranges** et **jaunes**) auxquelles correspondent autant d'opérations (la procédure *rouges* retourne tous les triplets RGB correspondant aux couleurs rouges prédéfinies, etc.) ; par exemple,

```
montre rouges
[[255 0 0] [255 0 255] [255 105 180]]
```

Pour fixer la couleur du crayon à une couleur prise au hasard parmi les rouges :

```
fixecouleurcrayon choix rouges
```

Autres opérateurs définis :
couleurs : retournes la liste de tous les triplets RGB des couleurs prédéfinies
couleurs.vraies : comme *couleurs* sauf les noirs et blancs

Autres manières de fixer une couleur du crayon au hasard :
parmi les couleurs prédéfinies :

```
fcc choix couleurs
```

parmi toutes les couleurs possibles :

```
fcc (liste hasard 256  hasard 256  hasard 256)
```

fixecouleurécran couleur **FCÉ**
SetScreenColor color *setSC*
Fixe la couleur du fond de l'écran à *couleur*.

couleurécran
ScreenColor
Retourne la couleur du fond de l'écran sous forme d'une liste des 3 intensités des couleurs de base (*triplet* RGB).

*** fixetaillecrayon** **FTC**
** setpenwidth n* *SETPW*
Change l'épaisseur du trait

```
setpenwidth 2.5
```

remplis
fill
Si la tortue est dans une surface fermée, la remplit de la couleur de remplissage courante (voir *fixecouleurremplissage*). Sinon remplit tout l'écran. Peu importe que le crayon soit baissé ou levé.

```
pour carré.plein :taille
 répète 4 [avance :taille droite 90]
 droite 45
 lèvecrayon
 avance :taille / 2
 remplis
 recule :taille / 2
 baissecrayon
 gauche 45
end
```

Note : *remplis* accepte un argument optionnel pour le choix du mode de remplissage : *(fill "false)* ou *(fill "true)*

fixecouleurremplissage **FCR**
setfloodcolor *SETFC*

Fixe la couleur de remplissage (voir *remplis*).

```
pour testremplissage
 fixecouleurremplissage noir
 carré.plein 50
 droite 90
```

```
 fixecouleurremplissage jaune
 carré.plein 100
 droite 90
 fixecouleurremplissage rouge
 carré.plein 150
end
```

4 Arithmétique

somme n1 n2 **+**
(somme n1 n2 n3 …)
sum n1 n2 +
(sum n1 n2 n3 …)
Retourne la somme des arguments

```
somme 1.4 3.6      ->  5
(somme 1 2 3 4 5)  -> 15
1.4 + 3.6          ->  5
1.4+3.6            ->  5
```

En Logo il n'y a qu'une seule règle syntaxique générale décrivant comment appliquer (utiliser) une procédure: c'est la notation ***préfixe***, c'est-à-dire une notation où la procédure apparaît en premier suivi de ses arguments, comme par exemple dans :

```
somme 1.4 3.6
(somme 1 2 3 4 5)
list 100 200
first [1 2]
pendown
```

Notez que cette notation permet **zéro, un ou plusieurs** arguments.

Il y a une exception à cette règle : lorsqu'on veut utiliser les abréviations des procédures arithmétiques (+, - , * , /), on doit utliser la notation ***infixe***, où la procédure est mise entre **deux** arguments; par exemple :

```
1.4 + 3.6
```

Notez les différences:

- Le ***nombre d'espaces*** autour du signe + ne joue pas (zéro ou plus)
- Après le mot sum, il faut ***au moins un espace***, sinon, par exemple dans *(sum2 3 4 5)* , l'évaluateur Logo considère que *sum2* est un nom de procédure.

Les ***parenthèses*** sont obligatoires si le nombre d'arguments n'est pas celui requis (2 dans le cas de *sum*) . Il est toujours permis de mettre des parenthèses autour de l'application d'une procédure.

différence n1 n1 **-**
difference n1 n2 -
Retourne la différence des arguments

```
difference 10 5  ->  5
```

moins **-**
minus n -
Retourne l'argument changé de signe

```
moins 3  ->  -3
-3  ->  -3
```

produit n1 n2 *****
(produit n1 n2 n3 …)
product n1 n2 *
(product n1 n2 n3 …)
Retourne le produit des arguments

```
produit 3 2  ->  6
(produit 1 2 3 4 5)  ->  120
```

quotient n1 n2 /

Retourne le quotient des arguments

```
quotient 3 5  ->  0.6
```

reste

remainder n1 n2

Retourne le reste de la division de n1 par n2 . n1 et n2 doivent être des entiers

```
reste 5 3  ->  2
```

arrondi n

round n

Retourne l'entier le plus proche du nombre n (arrondi)

```
arrondi 3.6  ->  4
```

racine n

sqrt n

Retourne la racine carrée de n

```
racine 9  ->  3
```

puissance n1 n2

power n1 n2

Retourne *n1* à la puissance *n2*

```
puissance 2 4  ->  16
```

log10 n

Retourne le logarithme de n

```
log10 1000  ->  3
```

sin n

Retourne le sinus de n (degrés)

```
sin 90  ->  1
```

cos n

Retourne le cosinus de n (degrés)

```
cos 0  ->  1
```

5 Opérations logiques

et true.false1 true.false2
(et true.false1 true.false2 true.false3 …)
and true.false1 true.false2
(and true.false1 true.false2 true.false3 …)
Retourne *true* si tous les arguments sont *true,* sinon retourne *false*
Chacun des arguments doit être *true* ou *false*

ou true.false1 true.false2
(ou true.false1 true.false2 true.false3 …)
or true.false1 true.false2
(or true.false1 true.false2 true.false3 …)
Retourne *true* si un ou plus d'un argument est *true,* sinon retourne *false*
Chacun des arguments doit être *true* ou *false*

non true.false
not true.false
Retourne *true* si l' argument est *false*, et inversément
L'argument doit être *true* ou *false*

6 Espace de travail

6.1 Définition de procédure

pour nom.procédure :argument1 :argument2 :argument3 …
corps de la procédure
end
to nom.procédure :argument1 :argument2 :argument3 …
corps de la procédure
end

La commande spéciale **pour** permet de définir un procédure. Son premier argument est le nom de la procédure à définir. A droite du nom de la procédure, dans la 1ère ligne, vient la spécification d'un certain nombre de variables correspondant aux arguments à utiliser lors de l'application de la procédure (zéro, un ou plusieurs arguments). Le corps de la procédure est constitué d'un ensemble d'instructions en une ou plusieurs lignes. **end** est un mot spécial qui marque la fin de la définition de la procédure. Par exemple :

```
pour carré50
 répète 4 [avance 50 droite 90]
end

pour carré70
 répète 4 [avance 70 droite 90]
end

pour carré :taille
 répète 4 [avance :taille droite 90]
end

pour bonjour
 écris [Bonjour]
 écris [Comment vas-tu?]
end
```

Note : le deux points utilisé dans la définition du nom des variables d'argument (à droite du nom de la procédure) n'a pas la signification du deux points décrite dans le chapitre suivant (abréviation pour *chose "*). Ici le deux points n'a qu'une valeur mnémonique : par exemple dans *pour carré :taille*, le deux points rappelle qu'une variable de nom *taille* est définie.

MSWLogo et UCBLogo permettent de définir une procédure avec un ou plusieurs arguments optionnels. Par exemple, la procédure qui suit permet de faire dessiner un polygone par la tortue. Cette procédure a un argument optionnel, la longueur du côté d'un polygone (dont la valeur est par défaut 100 pas de tortue).

```
pour polygone :n [:longueur 100]
 répète :n [avance :longueur droite (quotient 360 :n)]
end
```

Noter que les parenthèses autour de *quotient 360 :n* ne sont pas nécessaires (*quotient* requiert 2 arguments), elles ajoutent à la lisibilité en montrant bien que *quotient* s'applique aux 2 arguments *360* et *:n*.
Pour dessiner un pentagone (de côté 100), faire :

```
polygone 5
```

Pour dessiner un pentagone de côté 80, faire :
```
(polygone 5 80)
```

Notez que dans *(polygone 5 80)* les parenthèses sont obligatoires parce que le nombre d'arguments n'est pas égal au nombre d'arguments requis (un dans ce cas).

Pour définir la procédure polygone de telle sorte que le nombre de côtés soit aussi optionel, commencer la définition par :
```
pour polygone [:n 5] [:longueur 100]
```

Dans ce cas, pour dessiner un pentagone de côté 100, faire :
```
polygone
```

Un exemple de procédure récursive :
```
pour spirale :taille :angle [:incr 2] [:max 200]
 si :taille > :max [stop]
 avance :taille
 droite :angle
 (spirale (somme :taille :incr) :angle :incr :max)
end
```

Exemples d'applications de la procédure *spirale* :
```
spirale 2 90
(spirale 2 90 4)
(spiral 1 91 1 400)
```

6.2 Définition de variable

donne nom.variable valeur
make nom.variable valeur
Soit définit une nouvelle variable globale et lui assigne une valeur,
soit assigne une nouvelle valeur à une variable existante (globale ou locale).
Le nom de la variable est un mot.

```
donne "var.a 1
donne "var.a 2
```

La première instruction définit la variable globale de nom *var.a* et lui assigne la valeur 1. La 2ème intsruction assigne la valeur 2 à la variable *var.a* .
Une variable ***globale*** est une variable accessible par toutes les procédures.

Note : l'utilisation du guillemet anglais (" ou *quote*) est nécessaire pour faire en sorte que le mot *var.a* soit évalué à lui-même et non comme le nom d'une procédure.

locale nom.variable
(locale nom.variable.1 nom.variable.2 …)
local nom.variable
(local nom.variable.1 nom.variable.2 …)
Une commande qui accepte un ou plusieurs mots en argument pour définir autant de variables locales à une procédure (ceci veut dire que la commande *locale* ne peut s'employer que dans la définition d'une procédure). Les variables locales créées par *locale* n'ont pas de valeur initiale. Il faut assigner une valeur à une variable locale (voir *donne*) avant que la procédure ne tente de lire sa valeur.

Cette commande est surtout intéressante quand il faut définir plusieurs variables locales, sinon il est plus simple d'utiliser *donnelocale*.

```
pour spirale.carrée :n
 locale "distance
 donne "distance 2
 répète :n [avance :distance droite 90
            donne "distance (somme :distance 2)]
end
```

donnelocale nom.variable valeur
localmake nom.variable valeur

Une commande qui définit une nouvelle variable locale et lui assigne une valeur (ceci veut dire que la commande *donnelocale* ne peut s'employer que dans la définition d'une procédure). Une variable ***locale*** à une procédure est une variable accessible dans cette procédure et toutes les sous-procédures appelées par cette procédure (on dit qu'en Logo les variables ont une **portée dynamique**)

```
pour spirale.carrée :n
 donnelocale "distance 2
 répète :n [avance :distance dr 90
            donne "distance somme :distance 2]
end
```

Bien entendu, cette fonction pourrait s'écrire comme ceci, sans variable locale mais avec un argument optionnel :

```
pour spirale.carrée :n [:distance 2]
 répète :n [av :distance dr 90
            donne "distance (somme :distance 2)]
end
```

Voir aussi dans le chapitre de définition d'une procédure, la fonction *spirale*, récursive et plus générale.

chose nom.variable
thing nom.variable
Une opération qui retourne la valeur d'une variable.

```
Soit la variable var.a initialisée à 2 :
    donne "var.a 2
alors
    montre chose "var.a
    >>  2
    donne "var.a 1
    montre chose "var.a
    >> 1
```

Il y a une abréviation pour **chose "** (*thing-quote*) : c'est le caractère deux points accolé au nom de la variable ; donc :

```
    montre :var.a 1
    >> 1
    donne "var.a 2
    montre :var.a
    >>  2
```

7 Structures de contrôle

7.1 Divers

si test liste.instructions
if test liste.instructions
Exécute la liste d'instructions si le premier argument est le mot *true* (vrai) ; dit autrement: exécute la liste d'instructions si l'évaluation de l'expression *test* retourne *true*.
Ne fait rien si le premier argument est le mot *false* (faux).
Si le premier argument n'est ni *true*, ni *false*, donne une erreur.
En pratique le premier argument sera la valeur retournée par un prédicat comme *equalp* dans l'exemple ci-dessous.

Ceci a pour effet l'affichage du mot *ouais :*
```
if equalp 2 1+1 [écris "ouais]
```

Ceci n'a aucun effet:
```
if equalp 2 2+2 [écris "non]
```

sinon test liste.instructions.1 liste.instructions.2
ifelse test liste.instructions.1 liste.instructions.2
Synonyme : *sisinon*
Si le premier argument est le mot *true* (vrai), exécute la première liste d'instructions, si le premier argument est le mot *false* (faux) exécute la deuxième liste d'instructions. Si le premier argument n'est ni *true*, ni *false*, donne une erreur. En pratique le premier argument sera la valeur retournée par un prédicat comme *equalp* dans l'exemple ci-dessous.

Ceci a pour effet l'affichage du mot *ouais :*
```
ifelse equalp 2 1+1 [écris "ouais][écris "non]
```

Ceci a pour effet l'affichage du mot *non :*
```
ifelse equalp 2 2+2 [écris "ouais][écris "non]
```

retourne valeur
output valeur
Synonyme : *rapporte*
Termine l'exécution d'une procédure là où cette commande apparaît. La ***procédure*** retourne la valeur spécifiée. *Output* est une commande.
```
output 1
```

stoppe
stop
Termine l'exécution d'une procédure là où cette commande apparaît. La procédure ne retourne aucune valeur.

attends
wait n
Provoque une pause de *n* $60^{ème}$ de seconde.

7.2 Itération

répète n liste.instructions
repeat n liste.instructions
Exécute la liste d'instructions autant de fois que spécifié (n)

```
répète 10 [écris "waha]
répète  4 [avance 100 gauche 90 étiquette "waha]
```

compteur.r
repcount
Cette opération ne peut être utilisée que dans la liste d'instructions d'une commande *répète*. Elle retourne le numéro de la répétition courante (1 pour le première passage dans la liste d'instruction, etc.).
Par exemple,

```
répète 3 [montre compteur.r]
```

aura pour pour effet d"afficher 1 , 2 et 3 dans la fenêtre texte.

répètepour directive liste.instructions
for directive liste.instructions
Le 1er argument (la *directive*) doit être une liste de 3 ou 4 éléments:

1. un mot qui sera utilisé comme variable locale (*variable de contrôle*)
2. un mot ou une liste dont l'évaluation doit donner un nombre qui sera la valeur initiale de la variable
3. un mot ou une liste dont l'évaluation doit donner un nombre qui sera la valeur limite de la variable
4. un mot ou une liste optionnel dont l'évaluation doit donner un nombre qui sera le pas d'incrémentation de la variable (si ce 4ème élément n'est pas présent, il vaut 1 ou -1 selon que la valeur limite est plus grande ou plus petite que la valeur initiale

L'effet de *for* est de répéter l'exécution de la liste d'instructions (2ème argument) en affectant chaque fois une nouvelle valeur à la variable de contrôle (en commençant par la valeur initiale, ensuite en l'incrémentant du pas). Dès que la valeur de la variable dépasse la valeur limite, l'action de *for* se termine. Notez qu'à chaque boucle, *for* commence par l'affectation, ensuite le test et éventuellement l'exécution de la liste d'instructions: cela veut dire que l'exécution de la liste d'instructions peut très bien ne jamais se produire.

```
for [i 1 3] [écris :i]
```

a pour effet d'afficher 1 , 2 et 3 . Le pas est de 1 par défaut : cela reviendrait au même d'écrire :

```
for [i 1 3 1] [écris :i]

for [i 1 3 3] [écris :i]
```

a pour effet d'afficher 1

tantque liste.test.fin liste.instructions
while liste.test.fin liste.instructions
Répète l'évaluation de la liste d'instructions tant que l'évaluation du 1er argument donne *true*. Commence par l'évaluation du 1er argument, ce qui veut dire que l'exécution de la liste d'instructions peut très bien ne jamais se produire.

Dans l'exemple, ci-dessous il y a répétition (ou boucle) tant que le nombre tiré au hasard est plus petit que 6. Cette procédure est exactement équivalente à la procédure *démo.until* (voir la définition de la primitive *until*) ; notez les différences dans le test de fin de boucle.

```
to démo.while
 localmake "nombre 2
 écris (sentence "démo, "valeur "initiale :nombre)
 while [less? :nombre 6] ~
  [make "nombre hasard 10
   écris :nombre]
 écris [fin démo]
end
```

dèsque liste.test.fin liste.instructions
until liste.test.fin liste.instructions
Répète l'évaluation de la liste d'instructions jusqu'au moment où l'évaluation du 1er argument donne *true* (autrement dit *dès que la valeur du test est true, interrompre la boucle*). Commence par l'évaluation du 1er argument, ce qui veut dire que l'exécution de la liste d'instructions peut très bien ne jamais se produire.

Dans l'exemple ci-dessous, la répétition (ou boucle) s'arrête dès que le nombre tiré au hasard est plus grand que 5. Cette procédure est exactement équivalente à la procédure *démo.while* (voir la définition de la primitive *while*) ; notez les différences dans le test de fin de boucle.

```
to démo.until
 localmake "nombre 2
 print (sentence "démo, "valeur "initiale :nombre)
 until [greater? :nombre 5] ~
  [make "nombre hasard 10
   print :nombre]
 print [fin démo]
end
```

7.3 Itération selon un modèle

map modèle donnée
(map modèle donnée1 donnée2 …)
Retourne une donnée de même type et de même longueur que *donnée* (ou *donnée1*), qui est un mot ou une liste. Si il y a plusieurs données (*donnée1, etc.)* en argument, elles doivent avoir même longueur. Chaque élément de la donnée retournée est le résultat de l'évaluation du modèle où les emplacements (ou marqueurs) sont remplis avec les éléments correspondants dans les données d'entrée (*donnée* ou *donnée1 donnée2* …).

Un premier exemple trivial:

```
map [?] [1 2 3]  ->  [1 2 3]
```

Le marqueur est le point d'interrogation, tour à tour remplacé par 1 , 2 et 3 …

```
map [? * ?] [1 2 3]  ->  [1 4 9]
```

Le marqueur est tour à tour remplacé par 1 , 2 et 3 . L'évaluation du modèle donne tour à tour le résultat des multiplications 1*1 , 2*2 et 3*3

```
(map [product ?1 ?2] [1 2 3] [2 3 4])  ->  [2 6 12]
```

Lorsque il y a deux arguments *donnée* après le *modèle*, on utilise 2 marqueurs ?1 et ?2 qui seront tour à tour liés aux éléments correspondant dans les deux données. L'évaluation du modèle donne tour à tour le résultat des multiplications 1*2 , 2*3 et 3*4.
Notez:

- qu'il est obligatoire d'utiliser des parenthèses autour de la procédure *map* vu que *map* est appliqué à un nombre d'arguments différent (3) de celui requis par défaut (2).
- au lieu de *product ?1 ?2* on aurait pu écrire *?1 * ?2*

Même principe, si il y a plus de deux arguments données:

```
(map [(product ?1 ?2 ?3)] [1 2 3] [2 3 4] [3 4 5])  ->  [6 24
60]
```
Notez les parenthèses obligatoire pour la procédure *product* vu qu'elle reçoit 3 arguments au lieu des 2 requis par défaut.

Un exemple avec des mots:

```
map "first [Noé Leboutte"] -> [N L]
```

filtre modèle.prédicat donnée

filter modèle.prédicat donnée

Retourne une donnée de même type que *donnée*, qui est un mot ou une liste. La valeur retournée est un sous-ensemble de *donnée*. Le *modèle.prédicat* est évalué tour à tour pour chaque élément ou caractère de *donnée* : si cette évaluation donne *true,* alors l'élément (ou caractère) correspondant est gardé pour accumulation dans le résultat ; si elle donne *false* alors l'élément (ou caractère) correspondant est écarté.

```
filtre "number? [5 pommes, 3 poires, 2 navets]  -> [5 3 2]
```

8 Communication

montre truc
(montre truc1 truc2 …)
show truc
(show truc1 truc2 …)
Ecrit le ou les arguments dans la fenêtre texte. Chaque argument peut-être un mot ou une liste.

```
montre 2001
montre "waha
montre [waha 1PAa 1PAb]
(montre 2001 "waha [waha 1PAa 1PAb])
```

écris truc **ÉC**
(écris truc1 truc2 …)
print truc *PR*
(print truc1 truc2 …)
Ecrit le ou les arguments dans l'historique de la fenêtre de commande. Chaque argument peut-être un mot ou une liste. Si un argument est une liste, n'écrit pas les crochets de la liste.

```
écris 2001
```

9 Divers

hasard n
random n
n doit être un entier positif. Retourne un entier positif au hasard plus petit que *n*.

```
hasard 256  ->  129
```

son sons
sound sons
L'argument *sons* est une liste de sons. Chaque son est une paire de nombres, le premier étant la fréquence du son (hertz), le second sa durée (en 1/60 de seconde).

```
son [1000 100]
son [1000 100 2000 300]
```

REMARQUE: dans UCBLogo la procédure *son* (*sound*) n'est disponible que si le fichier de démarrage misc.lgo a été chargé.

10 Index

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W