Leçons et exemples de programmation en Logo

Francis Leboutte

Le 18 mai 2005[1]
www.algo.be/logo.html

IDDN.BE.010.0093308.000.R.C.2001.035.42000

[1] 1ère édition en juin 2003

Table des matières

Table des matières *2*

1 Déplacements de la tortue *3*
Messages d'erreurs 3

2 Les angles, la position et le cap *5*
Angle externe et angle interne 5

3 Terminologie et syntaxe *7*

4 répète *8*

5 Crayon et couleurs *9*

6 Définition de procédures *10*
Procédures primitives ou procédures prédéfinies 10

6.1 Définir une procédure **10**

6.2 Editeur Logo (éditeur de l'espace de travail). Sauvegarde dans fichier. **12**

7 Procédures avec arguments *13*

8 Polygones réguliers *15*

8.1 Syntaxe des opérations arithmétiques et de leurs abréviations **15**

8.2 Règles de précédence **16**

9 Variables *18*

9.1 Variables globales et locales **20**

10 Division d'un problème en plusieurs problèmes plus simples - Abstraction *22*

11 Autres exemples *24*

11.1 Conversion d'un montant en FB en euros, en arrondissant aux centimes **24**

11.2 Calcul de angles d'un polygone régulier **26**

11.3 Cubage d'un arbre **26**

11.4 Nuances de couleur **26**

11.5 Etoile à 5 branches **27**

11.6 Ciel **28**

11.7 L'ordre à partir du chaos **29**

11.8 Poly-polygones **29**
6 hexagones en un tour complet 29
10 décagones en un tour complet 29
Généralisation, n polygones à n côtés en un tour complet 30
Suite de poly.polygones 30

11.9 Le robot de Thomas **30**

12 Récursivité *31*

Remarque

- Le code Logo des exemples de ce document est disponible sous forme de fichiers LGO, voir Guides Logo.

1 Déplacements de la tortue

Voir dans le manuel de référence les procédures suivantes :

- **avance**, **recule**, **droite** et **gauche** et leurs abréviations **av, re, dr** et **ga** (chapitre *Graphisme, Déplacement de la tortue*)

Notes :

- ces 4 procédures primitives (mots du langage) sont des commandes
- elles ont un (seul) argument
- *avance* et *recule* modifient la position de la tortue (sans modifier son cap !)
- *droite* et *gauche* modifient son cap (sans modifier sa position !)

Voir dans le manuel de référence :

- **origine** (*Graphisme, Déplacement de la tortue*)
- **nettoie**, **nettoietout** (abréviation : **nt**). Chapitre *Graphisme, Contrôle de la tortue et de l'écran*.

Ces 3 commandes primitives n'ont pas d'argument.

Messages d'erreurs

Voir le manuel de référence pour les messages d'erreur communs (chapitre *Introduction*). Dans les exemples suivants, le symbole `>>` est utilisé pour indiquer le résultat d'une interaction avec le Logo (via la ligne de commande), tel qu'il peut apparaître dans l'historique de la fenêtre texte de MSWLogo.

```
recule
>> not enough inputs to recule

recul 111
>> I don't know how  to recul

recule111
>> I don't know how  to recule111
```

Activités

Essayer toutes ces commandes.
Remarquer que *recule* fait reculer la tortue tout en conservant le même cap.

Sur papier, dessiner le parcours de la tortue dans le cas des 3 séries d'instructions suivantes :

```
nt
av 30
dr 90
av 40

nt
av 20
dr 30
re 60
dr 180
```

```
nt
dr 45
av 80
re 80
ga 90
av 80
re 80
```

Exécuter les instructions suivantes :

```
nt
av 75
re 75
dr 30
av 75
re 75
dr 30
```

Quel est le motif qui est reproduit ? Continuer la séquence d'instruction jusqu'à obtenir un dessin qui vous plaît.
Suggestion : ensuite, faire *dr 15* (ou *ga 15*) et reproduire le dessin (éventuellement, commencer par changer la couleur du crayon via le menu *Set* | *Pencolor*).

Dessiner :

- un L
- un carré
- un rectangle

Amener la tortue :

- près du haut de l'écran
- dans un coin
- de la ramener au centre de l'écran (en utilisant les 4 premières commandes)

Si la tortue disparaît, faire *nt* ou déplacer la position de la fenêtre à l'aide de la barre de défilement.

Solutions

Carré

Dans la ligne de commande, introduire successivement :

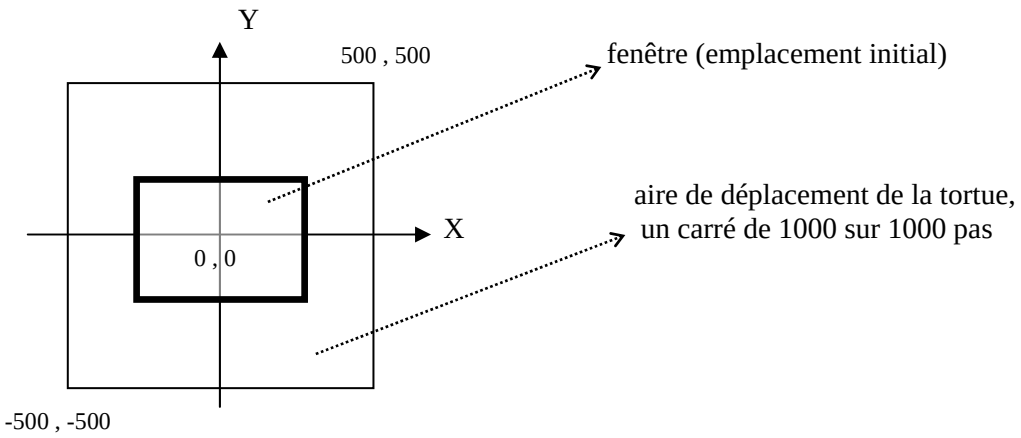
```
av 50
dr 90
av 50
dr 90
av 50
dr 90
av 50
dr 90
```

ou, dans la ligne de commande, introduire 4 fois :

```
av 50 dr 90
```

Note : voir dans le manuel de référence, les ***Trucs de la ligne de commande*** du chapitre *Introduction, Présentation de MSWLogo*.

2 Les angles, la position et le cap



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.

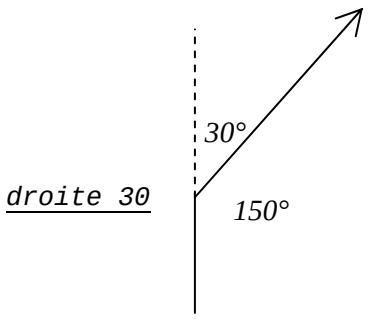
Voir dans le manuel de référence :

- **pos**, **cap** (*Graphisme, Information à propos de la tortue*)
- **montre** (*Communication*)

Note

- Toute procédure Logo est soit une **opération**, c'est-à-dire une procédure qui retourne une valeur (par exemple, *pos* et *cap*) , soit une **commande**, c'est-à-dire une procédure qui ne retourne pas de valeur (par exemple *avance*).

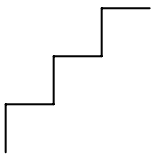
Angle externe et angle interne



- L'angle de rotation de la tortue est de 30° (angle **externe**)
- L'angle **interne** est de 150°
- La somme des 2 angles fait 180° (ces deux angles sont **supplémentaires**).

Activités

Dessiner 3 marches d'un escalier :



Vérifier la position et le cap de la tortue à chaque étape en faisant

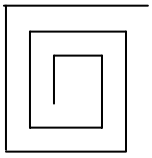
```
montre cap
montre pos
```

Même exercice, mais en mettant à profit la possibilité de taper plusieurs instructions à la suite (rechercher la séquence d'instructions à répéter pour dessiner une marche complète).

Dessinez vos initiales. Voir dans le manuel les commandes ***lèvecrayon*** (*lc*) et ***baissecrayon*** (*bc*). Commencez par faire un plan sur papier.

Dessiner un triangle équilatéral (les 3 angles valent 60 degrés).
Attention à la notion d'angle externe et d'angle interne.
Question : à la fin du dessin, de combien de degrés la tortue a-t-elle tourné au total ?

Dessiner une spirale à angles droits :



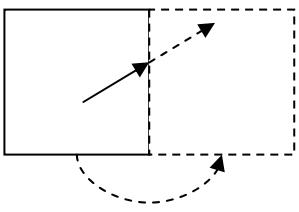
Notes

- Commencer par le centre (pourquoi ?)
- Que faut-il faire pour que la spirale soit régulière ?

Explorer les limites de l'aire de déplacement de la tortue : que se passe-t-il quand la tortue atteint la limite ? Essayer ce qui suit et observer ce qui se passe (attention à utiliser la barre de défilement pour visualiser les limites de l'aire de déplacement de la tortue) :

- `nt` et et plusieurs fois `av 100`
- `nt` et `av 1000` (*mille* exactement)
- De même, positionner la tortue en un endroit quelconque, orientée verticalement ou horizontalement (c'est-à-dire selon un des 4 caps 0, 90, 180 ou 270°).
  Ensuite faire `av 1000` (*mille* exactement).
  Note : en partant de la position origine, si le cap est un des 4 caps à 45° des 2 axes de coordonnées (45, 135, 225 ou 315°), la tortue revient exactement sur sa trace : elle parcourt la diagonale de l'aire de déplacement qui est un carré.
- `nt dr 10` et plusieurs fois `av 100`
- ```
  nt
  dr 1 av 16000
  ga 2 av 16000
  ```

  Pour mieux comprendre, tracer une ligne verticale et une ligne transversale passant par le centre d'une feuille de papier ; rouler la feuille en un cylindre. Ou imaginer que, lorsqu'un déplacement de la tortue la fait atteindre un des côtés de l'aire de déplacement, l'aire de déplacement est aussitôt repositionnée à l'endroit de ce côté :


  ```

Solutions

Escalier

Dans la ligne de commande, pour dessiner une marche tapez :

```
av 50 dr 90 av 50 ga 90
```

Deux marches de plus :

```
av 50 dr 90 av 50 ga 90
av 50 dr 90 av 50 ga 90
```

Notes
- *ga 90* remet la tortue au cap initial
- essayer aussi

```
av 50 dr 90 av 50 ga 90 montre cap montre pos
```

ou encore

```
- av 50  dr 90  av 50  montre cap  montre pos  ga 90
```

Initiales, par exemple ML :

D'abord le M en 2 séries d'instructions (après avoir fait un *nt*). Attention de spécifier les angles externes pour les commandes *dr* et *ga* ; par exemple 135º pour le 1er angle de rotation (et non pas 45º, l'angle interne). Pas de problème pour la rotation suivante où les angles externe et interne sont égaux (90º) :

```
av 100 dr 135 av 50 ga 90
av 50 dr 135 av 100
```

Ensuite, le déplacement à l'horizontal vers la droite (cap 90º), crayon levé :

```
lc ga 90 av 40 bc
```

et le dessin du L :

```
av 50 re 50 ga 90 av 100
```

Triangle équilatéral

La première instruction n'est pas nécessaire, elle n'est là que pour dessiner un triangle avec une base horizontale.

```
ga 90
av 100 dr 120
av 100 dr 120
av 100 dr 120
```

Spirale carrée. A **chaque rotation de 90 degrés**, on augmente le nombre de pas de 10 :

```
av 10 dr 90
av 20 dr 90
av 30 dr 90
…
```

3 Terminologie et syntaxe

Voir le manuel de référence (*Introduction*).

4 répète

Lorsqu'on veut exécuter plus d'une fois le même ensemble d'instructions, une possibilité est d'utiliser la commande *répète*. Voir dans le manuel de référence :

- **répète** (*Structures de contrôle*, *Itération*)

La commande *répète* prend 2 arguments, le nombre de répétitions et une liste d'instructions

```
répète 10 [avance 10]
répète 10 [avance 10 droite 9]
```

Un carré, c'est 4 fois les 2 instructions `avance 100 droite 90,` donc `:`
```
répète 4 [avance 100 droite 90]
```

Un carré, agrémenté d'un *bonjour* à chaque sommet :
```
répète 4 [avance 100 droite 90 étiquette "bonjour]
```

Activités

Faire des carrés de tailles différentes

Quel est l'effet de `répète 4 [av 100 re 30 dr 90]` ?

Faire un triangle équilatéral avec `répète`

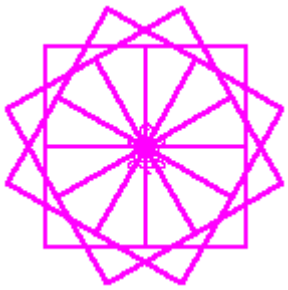
Faire un rectangle avec `répète`

Dessiner 2 carrés inscrits l'un dans l'autre, un de 50 et un de 100 pas de côté.

Faire un terrain de 4 carrés égaux et juxtaposés, en imbriquant 2 `répète`

Même exercice, mais avec des triangles équilatéraux (6 triangles juxtaposés par leurs sommets)

Dessiner une série de 12 carrés comme celle-ci



Même chose avec plus de carrés (30, ..., 360).

Ensuite après le dessin de chaque carré, changer la couleur du crayon de façon aléatoire en insérant l'instruction `fixecouleurcrayon choix couleurs` (ou `fcc choix couleurs`)

Solutions

Faire un rectangle avec `répète :`
```
répète 2 [av 100 dr 90 av 40 dr 90]
```

Dessiner 2 carrés inscrits l'un dans l'autre, un de 50 et un de 100 pas de côté :
```
  répète 4 [av 50 dr 90]
  lc dr 180 av 25 dr 90 av 25 dr 90 bc
  répète 4 [av 100 dr 90]
```

Faire un terrain de 4 carrés égaux et juxtaposés :
```
  répète 4 [répète 4 [av 100 dr 90] dr 90]
```

6 triangles équilatéraux égaux et juxtaposés :
```
  répète 6 [répète 3 [av 100 dr 120] dr 60]
```

Série de 12 carrés :
```
  répète 12 [répète 4 [av 100 dr 90] dr 30]
```

Il est très facile d'expérimenter en Logo. Pour voir de plus près la progression du dessin ci-dessus, taper l'instruction :
```
  répète 3 [carré droite 30]
```

ceci correspond au premier quart du dessin. Pour terminer le dessin, taper l'instruction :
```
  répète 9 [carré droite 30]
```

Note : 12 * 30° = 360°

Série de 12 carrés en changeant la couleur de façon aléatoire :
```
  répète 12 [répète 4 [av 100 dr 90] dr 30 fcc choix couleurs]
```

Autres séries :
```
  nt répète 36 [répète 4 [av 100 dr 90] ga 10]
```

5 Crayon et couleurs

Voir dans le manuel de référence, dans le chapitre *Graphisme*, *Crayon et couleurs* :

- **fixecouleurcrayon**, **rouge**, **vert**, ..., **montre.couleurs**, **rouges**, **verts**, ...
- **baissecrayon**, **lèvecrayon**
- **gomme**, **dessine**
- **fixetaillecrayon**

Voir aussi dans le manuel de référence l'opération **choix (***Structures de données, Procédures d'accès*).

Activités

Essayer les exemples du manuel de référence à propos des couleurs.

Refaire les activités du chapitre précédent en insérant des instructions de changement de couleur comme décrit dans le manuel de référence.

Essayer les instructions suivantes :

```
arbre.coloré 80
(arbre.coloré 80 bleu)
(arbre.coloré 60 vert)
```

Ensuite :

```
(arbre.coloré 70 choix verts)
(arbre.coloré 70 choix couleurs)
```

Explorer :

```
nt répète 36 [répète 4 [av 100 dr 90] dr 10 fcc choix couleurs]
nt répète 360 [répète 4 [av 100 dr 90] ga 1 fcc choix couleurs]
nt répète 1000 [répète 4 [av 100 dr 90] dr 0.36 fcc choix couleurs]
répète 10 [(arbre.coloré 70 choix couleurs) gauche 36]
```

6 Définition de procédures

Procédures primitives ou procédures prédéfinies

Jusqu'à présent nous n'avons utilisé que des procédures primitives, comme la procédure *avance.* Comme toute procédure, *avance* répond à ces caractéristiques :

1. elle a un **nom** (*avance*)
2. elle accomplit une **tâche** particulière (dans ce cas, celui de faire avancer la tortue d'un certain nombre de pas)
3. son application nécessite **un nombre standard d'arguments** (un seul argument dans ce cas, le nombre de pas).

L'application d'une procédure à des arguments constitue une **instruction**, exécutable par le Logo, comme on peut le faire dans la ligne de commande :

```
avance 100

montre somme 1 2
>> 3
```

somme est une **opération** . Elle calcule une valeur et la retourne. Le Logo impose que toute valeur retournée par une opération soit donnée à une autre procédure (*montre* dans l'exemple).
avance une **commande** . Elle a un certain effet mais ne retourne rien.

6.1 Définir une procédure

Voir dans le manuel de référence, la commande **pour** (*Espace de travail, Définition de procédure*).

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

La commande **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
```

La procédure ainsi définie a pour nom *carré50*. Son application ne nécessite pas d'argument. Le corps de cette procédure est fait d'une seule instruction faisant appel à 3 procédures primitives (une instruction *répète* dont la liste d'instructions est constituée elle-même de 2 instructions). Le tout permet d'accomplir le dessin d'un carré de 50 pas de côté.
Après que l'expression ci-dessus ait été évaluée par le Logo, la procédure *carré50* est disponible pour utilisation comme n'importe quelle procédure primitive. Dans la ligne de commande on peut maintenant introduire *carré50* pour dessiner un carré de 50 pas :

```
carré50
```

Même chose pour définir d'autres procédures dessinant des carrés de taille différente ; par exemple, la définition de *carré80* :

```
pour carré80
 répète 4 [avance 80 droite 90]
end
```

et son application dans la ligne de commande :

```
carré80
```

Comme n'importe quelle procédure primitive, ces nouvelles procédures peuvent aussi être utilisées dans d'autres instructions. Par exemple, pour faire un dessin de 100 carrés décalés d'un angle de 3.6° (la tortue faisant un tour complet et revenant à son état initial - mêmes position et cap) ; dans la ligne de commande :

```
répète 100 [carré50 droite 3.6]
```

Au lieu d'écrire *répète 100 [carré50 droite 3.6]* dans la ligne de commande, on pourrait définir une nouvelle procédure dont le nom serait *100.carrés* et qui ferait ce dessin de 100 carrés :

```
pour 100.carrés
 répète 100 [carré50 droite 3.6]
end
```

Ensuite, il suffirait de l'appliquer dans la ligne de commande :

```
100.carrés
```

Dans ce dernier cas on dit que *carré50* est une **sous-procédure** de *100.carrés*.

Un exemple de définition de procédure où le corps contient 2 instructions :

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

Notes

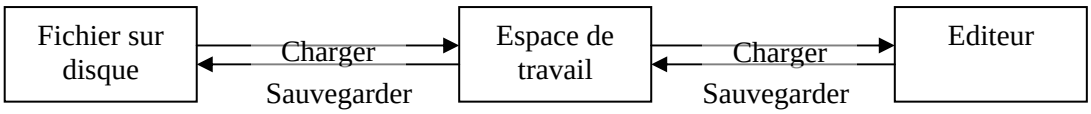
- La commande *pour* permet donc d'étendre le vocabulaire du langage Logo.

- Ce n'est pas le nom de la procédure qui fait ce que la procédure accomplit comme tâche. *carré50* n'est que le nom de la procédure. Au lieu de *carré50*, on aurait pu choisir *carré.50* ou *carré_50* . Ou encore *carré*, *triangle* ou *machin* (ce qui, dans les 3 cas, ne serait pas une bonne idée).
- Certains caractères sont interdits dans le nom d'une procédure, comme l'espace et le tiret.
- Une nouvelle procédure peut se définir via la ligne de commande, en tapant *pour carré* suivi de la touche *Entrée*, etc. En pratique, il est plus simple de se servir de l'éditeur Logo.

6.2 Editeur Logo (éditeur de l'espace de travail). Sauvegarde dans fichier.

Pour définir une ou plusieurs nouvelles procédures, on utilise l'éditeur Logo qui est invoqué via la commande *edall* ou le bouton **Edall** (*Edit all procedures* : éditer toutes les procédures). Quand l'édition est finie et que des procédures ont été écrites ou modifiées, il reste à les faire connaître au Logo via la commande de l'éditeur**, File | Save and exit** (Fichier | Sauvegarder et quitter). Ce faisant, il met à jour l'**espace de travail**. L'espace de travail est la partie de la mémoire de l'ordinateur dans laquelle le Logo met les procédures définies à l'aide de la commande *pour*. L'éditeur Logo permet d'éditer le contenu de l'espace de travail, c'est-à-dire l'ensemble des procédures courantes définies à l'aide de la commande *pour*.

Avant de quitter le Logo, il **faut sauvegarder l'espace de travail dans un fichier** sur le disque dur, afin de pouvoir restaurer les procédures lors d'une prochaine session MSWLogo. Les commandes concernées sont **MSWLogo | File | Save** (sauvegarde) et **MSWLogo | File | Load** . *Load* ajoute les procédures d'un fichier dans l'espace de travail (chargement ou restauration de procédures).



Pour plus d'information, voir le chapitre *Présentation de MSWLogo* dans le manuel de référence.

Activités

- Refaire les exemples ci-dessus.
- Définir des procédures pour dessiner des carrés de taille différentes : carré100 , carré30 ...
- Définir des procédures similaires à 100.carrés avec des nombres de tournants différents (tout en conservant une rotation totale de 360 degrés).
- Sauvegarder l'espace de travail. Quitter le Logo et restaurer l'espace de travail. Vérifer le contenu de l'espace de travail.
- Définir des procédures correspondantes aux exercices précédents.

Notes
Bien distinguer la phase de définition d'une procédure de la phase d'utilisation (application dans la ligne de commande ou utilisation dans la définition d'une autre procédure).
Par exemple, dans l'éditeur, définition de la procédure dont le nom est carré50 :

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

Application de cette procédure dans le la ligne de commande pour dessiner un carré de 50 pas :

```
carré50
```

Solutions et compléments

```
pour carré80
 répète 4 [avance 80 droite 90]
end

pour 360.carrés
 répète 360 [carré50 droite 1]
end

pour 2.100.carrés
 ; fcc : fixecouleurcrayon
 fcc bleu
 100.carrés
 lc av 150 bc            ; lc : lèvecrayon  -  bc : baissecrayon
 fcc vert
 100.carrés
 fcc noir                ; rétablit la couleur noire
end

pour 3.100.carrés
  répète 3 [fcc choix couleurs
            100.carrés
            lc av 150 dr 120 bc]
  fcc noir
end

pour 9.100.carrés
 répète 9 [fcc choix couleurs
           100.carrés
           lc av 150 dr 40 bc]
 fcc noir
end
```

Remarque

- une ligne précédée d'un ou plusieurs point-virgules est une ligne de commentaire, c'est-à-dire qu'elle n'est pas considérée comme une instruction à exécuter lors de l'application de la procédure où elle apparaît.

7 Procédures avec arguments

Les procédures *carré30, carré50, carré100*, etc. sont pratiques, mais il serait plus intéressant de définir une seule procédure **générale** pour dessiner un carré de taille quelconque ; son nom serait *carré* et elle aurait **un argument** (comme la procédure primitive *avance*), qui serait la taille du côté du carré. De la sorte, il suffirait de taper *carré 24* dans la ligne de commande pour obtenir un carré de 24 pas, *carré 33* pour un carré de 33 pas, etc. Voilà :

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

- A la droite du nom de la procédure, il y a le mot :*taille* qui indique au Logo que la procédure nécessite un argument lors de son utilisation (ou application).
- :*taille* indique la présence d'une variable, quelque chose qui sera remplacé par une valeur particulière lorsque la procédure sera appliquée (exécutée).

- Une variable peut être vue comme une boîte qui a un nom (*taille* dans ce cas) et qui contient une valeur. Dans le corps d'une procédure, on accède à la valeur qui est dans une boîte via le nom de la boîte (*taille*), en faisant précéder le nom de la boîte d'un deux-points (:*taille).*

Lorsqu'on écrit *carré 33* dans la ligne de commande, on dit au Logo que la valeur de la variable *taille* à utiliser est 33 . Lorsqu'on écrit *carré 50* (ou *carré 70, carré 80, etc.),* la valeur de la variable *taille* à utiliser est 50 (ou 70, 80, etc.) La valeur d'une variable dépend donc du contexte ; lors de l'exécution de tous ces exemples, la variable *taille* contient la valeur qui a été spécifiée comme argument lors de l'application de la procédure *carré*.

Comme le nom d'une procédure, le nom d'une variable n'a pas d'importance en soi. Au lieu de *taille*, on aurait pu par exemple utiliser *xx* :

```
pour carré :xx
 répète 4 [avance :xx droite 90]
end
```

Activités
Définir la procédure *carré* dans l'éditeur et l'appliquer.
Définir la procédure *rectangle* dans l'éditeur et l'appliquer (procédure à 2 arguments).
Définir une procédure *multi.carré* qui généralise la procédure *100.carrés*. Nécessite l'utilisation de la procédure *quotient*, voir le manuel.

Solutions

```
pour rectangle :b :h
  répète 2 [av :h dr 90 av :b dr 90]
end

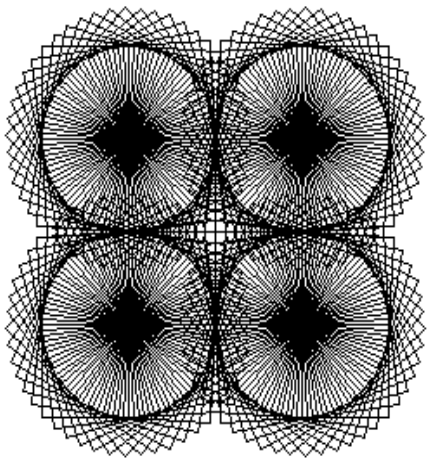
pour multi.carré :n
 répète :n [carré 50 droite (quotient 360 :n)]
end
```

La même, plus générale avec 2 arguments :

```
pour multi.carré :n :taille
 répète :n [carré :taille droite (quotient 360 :n)]
end
```

Exemple :

```
répète 4 [multi.carré 50 50 lc av 100 dr 90 bc]
```



8 Polygones réguliers

Polygone régulier : polygone dont tous les côtés ont même taille et dont les angles sont égaux.

A la fin du tracé d'un polygone régulier, la tortue aura tourné de 360 degrés, donc à chaque sommet elle tourne d'un angle de 360°/n (angle externe, supplément de l'angle interne ou angle du polygone). A noter : lorsque la tortue a fait un tour complet en dessinant un polygone, elle revient dans son état initial (position et cap).

```
pour triangle :taille
   répète 3 [av :taille ga 120]
end

pour pentagone :taille
   répète 5 [av :taille dr 72]
end

…
```

8.1 Syntaxe des opérations arithmétiques et de leurs abréviations

En Logo, il n'y a qu'**une seule règle syntaxique générale** décrivant comment appliquer (utiliser) une procédure dans une instruction : elle fait appel à la notation ***préfixe***, c'est-à-dire une notation où 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
- cette notation s'accommode d'un **nombre quelconque d'arguments**, zéro, un ou plusieurs arguments.

```
montre quotient 360 10
>> 36

montre somme 1 2
>> 3

montre (somme 1 2 3)
>> 6

montre (somme 1 2 3 4)
>> 10
```

On voit que *somme* est une procédure qui accepte un nombre quelconque d'arguments. Cependant, dans l'instruction *(somme 1 2 3)* , il est obligatoire de mettre des ***parenthèses*** car le nombre d'arguments n'est pas celui requis par défaut : le nombre *standard* d'arguments de *somme* est 2.

```
montre somme 1 2 3
>> erreur
```

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.

```
montre (somme 1 2)
>> 3
```

/ est l'abréviation de l'opération quotient ; elle nécessite la notation ***infixe*** (la procédure est mise entre **deux** arguments). De même pour les autres opérations arithmétiques de base : *somme* (+), *différence* (-) et *produit* (*).

```
montre 360/10
>> 36

montre 360 / 10
>> 36

montre 1 + 2 + 3
>> 6

montre 1+2+3
>> 6

pour polygone :n :distance
    ; 3 instructions équivalentes :
    ; répète :n [avance :distance droite (quotient 360 :n)]
    ; répète :n [avance :distance droite quotient 360 :n]
    répète :n [avance :distance droite 360/:n]
end
```

8.2 Règles de précédence

Dans le cas de la notation infixe, il est nécessaire de définir l'ordre dans lequel les opérateurs binaires[2] sont appliqués, pour le cas où une expression auraient plusieurs opérateurs (règles de précédence). Par exemple, en Logo, la division est prioritaire par rapport à l'addition :

```
montre 1+2+3/2
>> 4.5
```

Il est conseillé d'utiliser les parenthèses pour séparer les sous-expressions, car les règles de précédence varient selon les langages. D'autre part, la lisibilité est améliorée :

```
montre 1+2+(3/2)
>> 4.5
```

Activités

- Appliquer la procédure polygone
- Explorer :

```
pour hexa.bizarre :taille
   polygone 6 :taille
   attends 11
   hexa.bizarre somme :taille 10
end
```

- Approximation du cercle à partir de la procédure polygone

[2] à 2 arguments

Solutions et compléments

```
pour arbres
 répète 10 [arbre.coloré 80 droite 36]
end

pour 100.carrés :taille
 répète 100 [carré :taille droite 3.6]
end

pour 2.100.carrés :taille
 fcc bleu
 100.carrés  :taille
 lc av 150 bc
 fcc vert
 100.carrés  :taille
end

pour n.100.carrés :n :taille
 ; dans la ligne de commande tester l'effet de :
 ; liste (somme -400 hasard 800) hasard 400
 ; consulter le manuel pour : liste, choix , somme et hasard
 répète :n [fcc choix couleurs
            100.carrés :taille
            lc
            fixepos liste (somme -400 hasard 800) hasard 400
            bc]
end
```

Approximation du cercle à partir de la procédure polygone : il suffit de spécifier un polygone ayant de nombreux côtés suffisamment petits, comme dans *polygone 300 1*

Comment la procédure *cercle* pourrait s'écrire (note : *cercle* et *cercle2* sont des primitives), sachant que :

```
montre pi
>> 3.14159265358979

pour cercle3 :diam
 lc av :diam/2 dr 90 bc
 répète 360 [av pi*:diam/360 dr 1]
 lc dr 90 av :diam/2 dr 180 bc
end

pour multi.polygone :nbre.pg :n :taille
 répète :nbre.pg [polygone :n :taille
                  droite 360 / :nbre.pg]
end
```

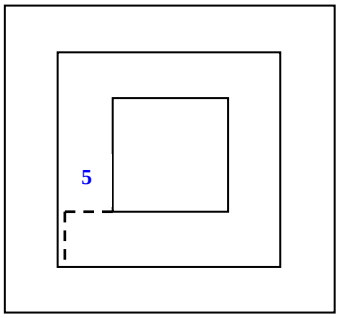
9 Variables

Une variable est comme une boîte qui a un nom et contient une valeur. On peut changer le contenu de la boîte (la valeur) à volonté.

Voir dans le manuel de référence, les procédures :

- *donne* (*make* en anglais) : chapitre *Structures de contrôle, Définition de variable*
- *compteur.r* : chapitre *Structures de contrôle, Itération*

Procédure dessinant une suite de carrés circonscrits et distants de **5** pas (un argument, le nombre de carrés à dessiner).

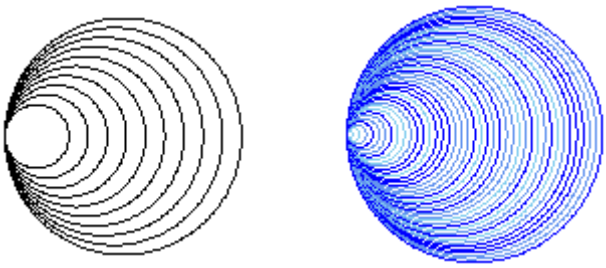



```
pour poly.carré :n
    ; initialisation de la variable taille à 22 (taille du 1er carré)
    donne "taille 22
    répète :n [carré :taille
               lc
               gauche 90
               avance 5
               gauche 90
               avance 5
               gauche 180
               ; donne à taille la valeur courante de taille + 10
               donne "taille somme :taille 10
               bc]
end
```

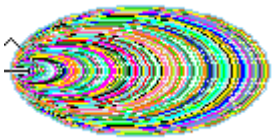
Activités

- Appliquer poly.carré
- Ajouter l'instruction *montre :taille* dans la définition de la procédure pour voir l'évolution de la variable *taille*.
- Ajouter l'instruction (*montre compteur.r :n :taille)* dans ...
- Que faire si on veut changer la valeur de la distance entre les carrés successifs ? Ne pourrait-on faire mieux ? (solution dans la variante ci-dessous)

- De la manière similaire, et en utilisant la procédure *polygone* pour dessiner un cercle, écrire les procédures *cercles* et *cercles.bleus* réalisant les 2 dessins ci-dessus. Ces procédures ont un seul argument, le nombre de cercles à dessiner.



- De même, faire le dessin suivant en utilisant la procédure primitive ellipse2



- Faire le parallèle entre la notion de variable et la notion de nom de cellule et de référence absolue dans un tableur

Variante avec une variable pour la distance entre 2 carrés successifs

```
pour poly.carré :n
  donne "taille 10
  donne "d2c 5       ; Distance entre 2 Carrés
  répète :n [carré :taille
             lc
             gauche 90
             avance :d2c
             gauche 90
             avance :d2c
             gauche 180
             donne "taille somme :taille produit 2 :d2c
             bc]
end
```

Evolution de la valeur des variables au cours de l'exécution de *poly.carré 3*, à chaque itération (au début et à la fin de la liste d'instructions du *répète* en ce qui concerne *taille*) :

compteur.r (numéro de l'itération)	**taille**		**n**	**d2c**
1	début	10	3	5
	fin	20		
2	début	20	3	5
	fin	30		
3	début	30	3	5
	fin	40		

Ce qu'on peut vérifier en insérant deux instructions *montre* dans la procédure :

```
pour poly.carré.b :n
  donne "d2c 5
  donne "taille 10
  répète :n [(montre "début compteur.r :taille :n :d2c)
             carré :taille
             lc
             gauche 90
             avance :d2c
             gauche 90
             avance :d2c
             gauche 180
             donne "taille somme :taille produit 2 :d2c
             bc
             (montre "fin compteur.r :taille :n :d2c)
             ]
 end
```

Solutions

```
pour cercles :n
 donne "n.côtés 100
 répète :n [polygone :n.côtés 1
            donne "n.côtés somme :n.côtés 30]
end

pour cercles.bleus :n
 donne "n.côtés 10
 répète :n [fixecouleurcrayon choix bleus
            polygone :n.côtés 1
            donne "n.côtés somme :n.côtés 10]
 fcc noir
end
```

9.1 Variables globales et locales

Après l'exécution de *poly.carré 3*, on constate que le variable *taille* est accessible via la ligne de commande et vaut 40 :

```
poly.carré 3

montre :taille
>> 40
```

De même pour :d2c :

```
montre :d2c
>> 5

(montre :taille :d2c)
>> 40 5

pour tester
 montre :taille
end
```

```
tester
>> 40
```

Les variables *taille* et *d2c* sont **globales**, c'est-à-dire qu'elles sont visibles de partout, comme le montre les exemples ci-dessus : elles sont visibles au niveau de la ligne de commande ou dans une autre procédure. C'est pourquoi, logiquement, des instructions *make* apparaissent dans l'éditeur de l'espace de travail, par exemple : *make "d2c 5*

Pour éviter de créer ainsi des variables globales, on peut les rendre locales de telle sorte qu'elles ne soient plus visibles que dans la procédure où elles sont définies :

```
pour poly.carré :n
  ; définition et initialisation de 2 variables locales
  donnelocale "long.côté 10
  donnelocale "dc 5
  répète :n [carré :long.côté
             lc
             gauche 90
             avance :dc
             gauche 90
             avance :dc
             gauche 180
             donne "long.côté somme :long.côté produit 2 :dc
             bc]
 end


montre :long.côté
>> long.côté has no value
```

La variable :*longueur* n'est donc plus visible de partout. On vérifiera aussi dans l'éditeur que des instructions *make* n'apparaissent pas pour les variables *long.côté* et *dc*

Autre exemple :

```
pour spirale.carrée :n
 ;; définition d'une variable locale dont le nom est distance et dont la valeur initiale est 3
 donnelocale "distance 3
 répète :n [avance :distance
            droite 90
            ; L'instruction qui suit augmente la valeur courante de la variable distance de 5
            donne "distance (somme :distance 5) ]
end
```

Activités

1. Essayer poly.carré et spirale.carrée
2. Insérer l'instruction (montre compteur.r :n :taille :inc) dans la procédure pour voir comment évolue les variables au cours de l'exécution de la procédure.
3. Transformer poly.carré pour que la distance entre 2 carrés soit un argument et non plus une constante.
4. Idem pour spirale.carrée.
5. Essayer de modifier les valeurs numériques de la définition de spirale.carrée, par exemple :
   - remplacer le 5 de (somme :distance 5) par 2 ou 1.
   - remplacer le 90 de droite 90 par 91
   - essayer de transformer la spirale carrée en une spirale arrondie.

6. Essayer et comprendre toutes les autres fonctions
7. Apprenez à vous servir de Edit | Copy (ou CTRL-C) dans l'éditeur et du coller dans la ligne de commande via le menu du bouton droit de la souris ou CTRL-V.

10 Division d'un problème en plusieurs problèmes plus simples - Abstraction

Voir dans le manuel de référence : *liste*, *pos*, *fixepos* et *hasard*

Activités

Expérimenter à partir de ce qui suit :

```
liste 100 100
montre liste 100 100
montre (liste 100 100)
montre pos
fixepos list 100 100
fixepos liste hasard 400 hasard 400
fixepos (liste hasard 400 hasard 400)
fixepos liste (hasard 400) (hasard 400)
fixepos (liste (hasard 400) (hasard 400))
(fixepos (liste (hasard 400) (hasard 400)))
répète 5 [fixepos list (hasard 400) (hasard 400)
          montre pos]
répète 5 [fixepos liste (hasard 400) (hasard 400)
          montre (liste compteur.r pos)]
répète 5 [fixepos liste (hasard 400) (hasard 400)
          écris (liste compteur.r pos)]
```

Flocon de neige

La procédure *vé* dessine un élément d'une branche d'un flocon :

```
pour vé
 ga 60
 av 10 re 10
 dr 120
 av 10
 ;; 1. retour à la position initiale
 re 10
 ;; 2. retour au cap initial !
 ga 60
 ;; 1. et 2. font que l'état de la tortue est préservé (position et cap)
 ;; (les états avant et après le dessin du v sont identiques)
end
```

La procédure *branche* dessine une branche d'un flocon :

```
pour branche
 répète 4 [av 10 vé]
 re 40              ; pour revenir à l'état initial
end
```

Un flocon a toujours **6** branches :

```
pour flocon
 répète 6 [branche droite 60]
end
```

neige, dessine un ensemble de flocons :

```
pour neige
 répète 100 [flocon
             lc
              fixepos liste hasard 400 hasard 400
              bc]

end
```

Activités

Reproduire les exemples ci-dessus. Expérimenter en modifiant la forme du vé et de la branche. Par exemple, modifier la 1ère instruction de branche en :

```
répète 4 [av 8 vé av 2 vé]
```

Conseil : avant de modifier la procédure vé, copier là et renommer la vé.1

Compléments

```
pour vé :taille :angle
 ga :angle
 av :taille re :taille
 dr 2 * :angle
 av :taille re :taille
 ; retour au cap initial => L'état de la tortue est préservé (position et cap)
 ga :angle
end

pour branche
 ; cette variante de branche restaure l'état initial de la tortue ce qui simplifie l'expérimentation
 donnelocale "pos.initiale pos
 donnelocale "cap.initial cap
 répète 3 [av 6 vé 10 60 av 4 vé 6 60]
 lc
 fixepos :pos.initiale
 fixecap :cap.initial
 bc
end

pour neige
 ; flocons blancs sur fond noir
```

```
 fixecouleurécran noir fcc blanc
 répète 50 [flocon
           lc
           fixepos liste (-400 + hasard 800) hasard 400
           bc]
 attends 60
 fcé blanc fcc noir
end
```

11 Autres exemples

11.1 Conversion d'un montant en FB en euros, en arrondissant aux centimes

A titre de préparation, commencez par simplement convertir 50 FB en tapant quelques instructions dans la fenêtre texte. Un euro valant 40.3399 FB, la conversion se fait en divisant par 40.3399 à l'aide de la procédure quotient. La procédure quotient retourne une valeur qui est le résultat de la division de son premier argument par son second.

```
montre quotient 50 40.3399
>> 1.2394676238662
```

En arrondissant aux centimes, vous obtiendriez 1.24 euro. Ce qui en programmation peut se faire selon les 3 étapes suivantes:

```
montre produit 100 1.2394676238662
>> 123.94676238662

montre arrondi 123.94676238662
>> 124

montre quotient 124 100
>> 1.24
```

Donc pour convertir un montant en FB en euros et arrondir aux centimes, il faut effectuer les 4 étapes suivantes:

1. diviser par 40.3399
2. multiplier par 100
3. arrondir à l'entier
4. diviser par 100

La première étape sera réalisée par la procédure *FB.euro* ci-dessous. La procédure primitive *retourne* ne s'emploie que dans la définition d'une procédure : elle termine l'exécution de la procédure là où elle apparaît de telle sorte que cette procédure retourne la valeur spécifiée en argument de *retourne*. La procédure *retourne* n'accepte qu'un argument.

```
pour FB.euro :n
 retourne quotient :n 40.3399
end
```

La procédure finale, *FB.euro.arrondi*, se définit comme suit :

```
pour FB.euro.arrondi :n
 retourne quotient arrondi produit 100 FB.euro :n 100
end
```

Afin d'augmenter la lisibilité d'une expression, il toujours permis d'ajouter des parenthèses autour de l'application d'une procédure. Dans la 2ème version de *FB.euro.arrondi* ci-dessous, les parenthèses permettent de mieux visualiser les 2 argument de quotient qui sont :

```
    arrondi produit 100 FB.euro :n
```
et
```
    100
```

ce qui donne:

```
pour FB.euro.arrondi :n
 retourne quotient (arrondi produit 100 FB.euro :n) 100
end
```

Vous pourriez utiliser plus de parenthèses et par exemple écrire le premier argument de quotient comme ceci :

```
(arrondi produit 100 (FB.euro :n))
```

Exemples de calcul :

```
montre FB.euro 100
>> 2.47893524773239

montre FB.euro.arrondi 100
>> 2.48
```

Compléments

Ecrire l'opération inverse (euro.FB.arrondi).

Opération arrondi générale :

```
pour arrondi_2 :n
 retourne (quotient (arrondi (produit 100
                                      :n))
                    100)
end

pour arrondi_3 :n
 retourne (quotient (arrondi (produit 1000
                                      :n))
                    1000)
end

pour arrondi_n :n_chiffres :n
 donnelocale "m puissance 10 :n_chiffres
 retourne (quotient (arrondi (produit :m :n))
                    :m)
end
```

Pour définir avec une opération de conversion d'euros en FB avec un chiffre après le point décimal, on pourrait écrire :

```
pour euroFB :n
 retourne produit :n 40.3399
end

pour euroFB_arrondi :n
 retourne (arrondi_n 1 (euroFB :n))
end
```

11.2 Calcul de angles d'un polygone régulier

```
pour angle.polygone :n
 retourne différence 180 (quotient 360 :n)
end

pour somme.angles.polygone :n
 retourne produit angle.polygone :n :n
end

pour angles.polygone :n
 (écris [ Angles d'un polygone régulier à ] :n [côtés :])
 (écris [- angle d'un sommet : ] angle.polygone :n)
 (écris [- somme des angles : ] somme.angles.polygone :n)
 (écris [- somme des angles externes : ] 360)
end
```

11.3 Cubage d'un arbre

```
pour cubage :h :cm
 retourne 0.08*:cm*:cm*:h
end
```

11.4 Nuances de couleur

Voir dans le manuel de référence : *cercle* (Dessin de courbes), le chapitre *Crayon et couleurs*, *sisinon* et *tantque* (Structures de contrôle).

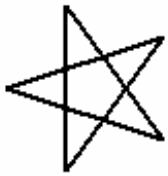
```
pour nuances.rouge
 locale "intensité
 fixetaillecrayon 2
 ; 256 cercles emboîtés, en 256 intensités de rouge, de 255 à 0
 répète 256 [
    donne "intensité (différence 256 compteur.r)
    fixecouleurcrayon (liste :intensité 0 0)
    circle 1.2 * :intensité]
 fixetaillecrayon 1
end

pour nuances.couleur :rouge? :vert? :bleu?
 locale "intensité
 fixetaillecrayon 2
 répète 256 [
    donne "intensité (différence 256 compteur.r)
    fixecouleurcrayon (list
                       sisinon :rouge? [:intensité] [0]
                       sisinon :vert? [:intensité] [0]
                       sisinon :bleu? [:intensité] [0])
    circle 1.2 * :intensité]
 fixetaillecrayon 1
end

pour nuances
 ; ne s'arrête jamais
 tantque ["true] [nt
          nuances.couleur "true "false "false
          nt
          nuances.couleur "false "true "false
          nt
          nuances.couleur "false "false "true
          nt
          nuances.couleur  "true "true "false
          nt
          nuances.couleur  "true "false "true
          nt
          nuances.couleur "false "true "true
          nt
          nuances.couleur "true "true "true]
end
```

11.5 Etoile à 5 branches

Définir une procédure *étoile5* qui dessine une étoile à 5 branches (la procédure a un argument, la taille d'une branche).



2 tours complet (2 fois 360 degrés) , en 5 fois (5 branches – tourner de 144°)

Définir une procédure *étoile5double* qui dessine une étoile à 5 branches en trait double (utiliser la précédente et le mode gomme)

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

11.6 Ciel

A partir de la procédure *étoile5* on peut dessiner un «ciel» avec une étoile baladeuse :

```
to ciel
 nt cto
 fixecouleurécran noir
 fcc blanc
 répète 20 [; déplace la tortue dans une zone particulière, au hasard et crayon levé
            lc
            fixepos (liste (somme -300 hasard 601)
                           hasard 201)
            bc
            étoile5 100 attends 40
            ; effacement
            fcc noir étoile5 100 attends 10
            ; au lieu de fcc choix couleurs :
            fcc (liste hasard 256  hasard 256  hasard 256)
            ]
 fixecouleurécran blanc fcc noir mto
end

pour étoile5double :distance
  donnelocale "état.crayon.initial crayon
  ;; crayon en gros trait
  fixetaillecrayon  5
  étoile5 :distance
  ;; crayon en trait plus fin
  fixetaillecrayon  2
  ;; crayon en mode gomme
  gomme
  étoile5 :distance
  fixecrayon :état.crayon.initial
end

pour étoile5double.b :distance
  donnelocale "état.crayon.initial crayon
  ;; crayon en gros trait
  fixetaillecrayon  5
  étoile5 :distance
  ;; crayon en trait plus fin
  fixetaillecrayon  2
  ;; crayon couleur écran (équivalent à gommer)
  fcc couleurécran
  étoile5 :distance
  fixecrayon :état.crayon.initial
end

pour étoile5multi [:distance 100] [:n 10]
 ;; essayez en tapant simplement étoile5multi (les arguments sont optionnels)
 ;; Sinon avec des arguments : (étoile5multi 80 11)
 recule 150
 répète :n [;; crayon dans une couleur au hasard
            fixecouleurcrayon choix couleurs
            étoile5double :distance
            donne "distance somme :distance 60]
 fixecouleurcrayon noir
end
```

11.7 L'ordre à partir du chaos

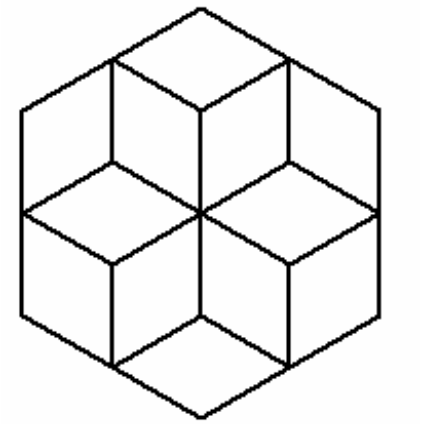
```
pour multi.av.dr
   ; suite quelconque d'appels successifs à avance et à droite
   av 100 dr 150 av 120 dr 100 av 60 dr 40
   ; av 10 dr 150 av 120 dr 100 av 60 dr 55
   ; attends 30
end

pour n.multi.av.dr :n
 répète :n [multi.av.dr]
end
```

Insérer *fcc choix rouges* devant l'appel à *multi.av.dr*

11.8 Poly-polygones

6 hexagones en un tour complet



```
pour 6hexagones
 ;; pour illustrer , essayer en remplaçant le nombre de répétition par 1 , 2 ou 3
 ;;  ou en insérant l'instruction attends 60 dans la liste du répète
 répète 6 [polygone 6 50
           droite 60]
end
```

10 décagones en un tour complet

```
pour 10décagones
 répète 10 [polygone 10 50
            fixecouleurcrayon choix couleurs
            droite 36]
end
```

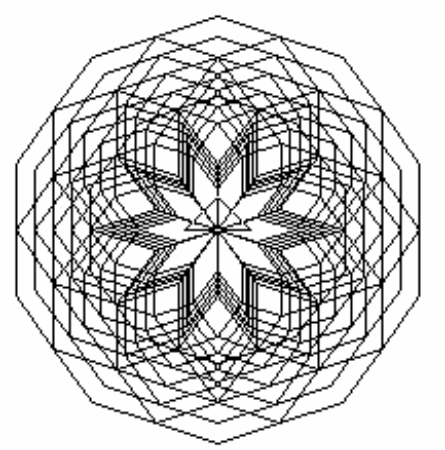
Généralisation, n polygones à n côtés en un tour complet

```
pour poly.polygones :n :taille
 répète :n [polygone :n :taille
            droite 360 / :n]
end
```

Suite de poly.polygones

Applications successives de *poly.polygones* pour des tailles différentes

```
n.poly.polygones 5 10 3
```



```
pour n.poly.polygones :n.poly :n :incr
 donnelocale "taille 20
 répète :n.poly [poly.polygones :n :taille
                 donne "taille somme :taille :incr]
end
```

11.9 Le robot de Thomas

```
pour démo
 cachetortue
 nettoietout
 répète 2 [fixecouleurcrayon choix couleurs.vraies
           dessiner.robot
           crie.robot
           attends 20 effacer.robot attends 10]
 répète 6 [avance.robot 100 effacer.robot
           droite 60
           fixecouleurcrayon choix couleurs.vraies
           crie.robot]
 dessiner.robot
end

pour avance.robot :d
 répète :d [micro.avance.robot]
end
```

```
pour crie.robot
 son [1000 100 2000 300]
end

pour dessiner.robot [:taille 10]
 donnelocale "double produit :taille 2
 donnelocale "demi quotient :taille 2
 polygone 4 :double
 avance :double droite 180
 polygone 3 :taille
 droite 270 avance :demi gauche 90
 polygone 4 :taille
 avance :taille droite 30
 polygone 3 :taille
 gauche 30 recule :taille droite 90 recule :demi avance :double droite 30
 polygone 3 :taille
 droite 60 avance :double droite 90 avance :double droite 90
end

pour effacer.robot
 gomme
 dessiner.robot
 dessine
end

pour micro.avance.robot
 effacer.robot
 lèvecrayon
 avance 1
 baissecrayon
 dessiner.robot
 attends 1
end

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

12 Récursivité **(chapitre à compléter)**

Une procédure qui fait appel à elle-même comme sous-procédure est une procédure récursive. La procéduer *spirale.a* ne s'arrête jamais, il faut utiliser le bouton Halt pour l'arrêter.

```
pour spirale.a :distance
 avance :distance
 droite 90
 ;; attente de 5/60 ème de seconde, sinon ça va trop vite…
 attends 5
 spirale.a somme :distance 3
end
```

La procédure *spirale.b* a une condition d'arrêt :

```
pour spirale.b :distance
 si :distance > 200 [stop]
 avance :distance
 droite 90
 attends 5
 spirale.b somme :distance 3
end
```

Essayez *poly.spirale 5 120*

```
pour poly.spirale :taille :angle
 si :taille > 205 [stop]
 avance :taille
 droite :angle
 poly.spirale somme :taille 5 somme :angle 0.12
end
```