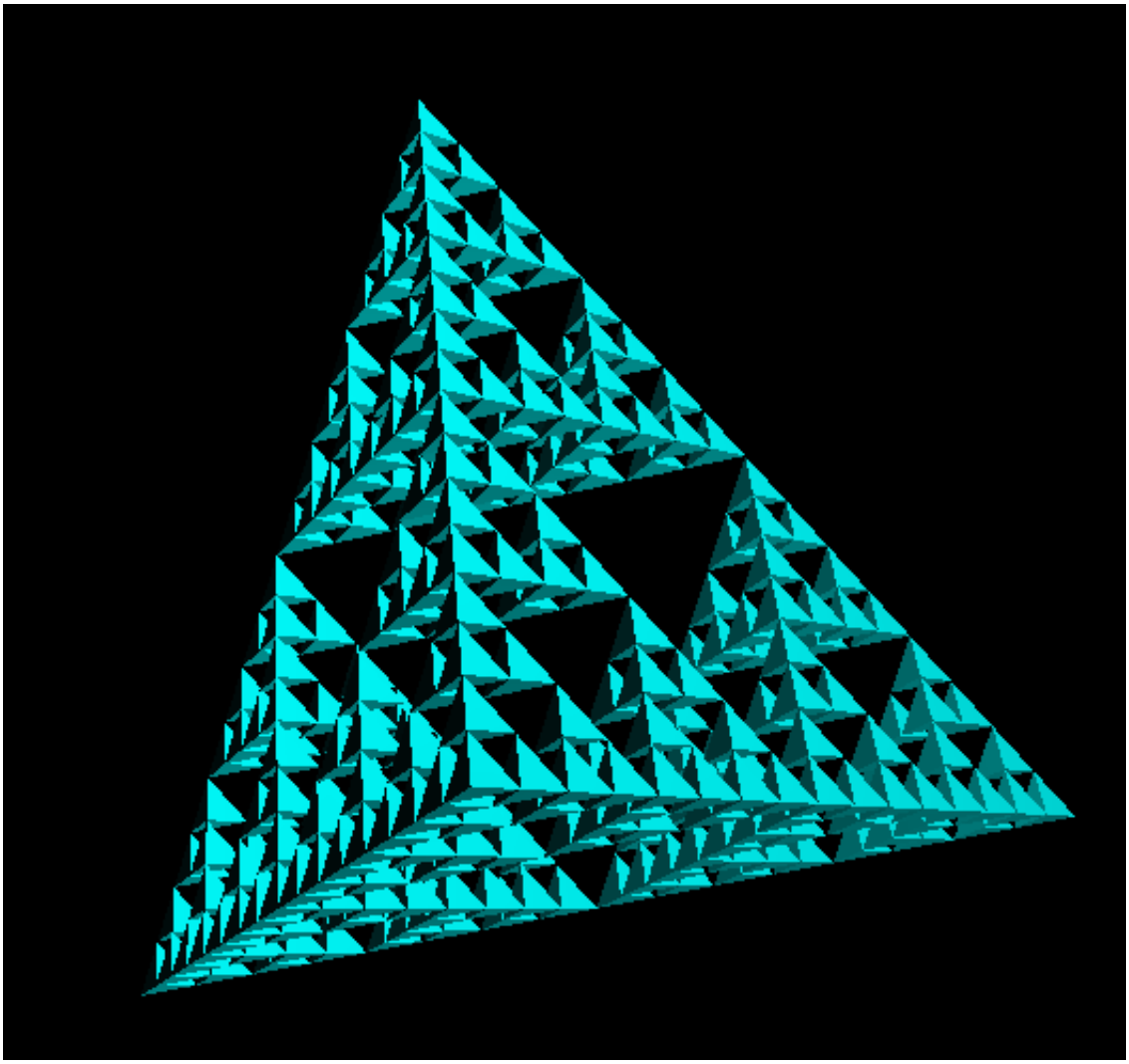
XLogo: Reference Manual

Loïc Le Coq

Translators: Guy Walker

September 18, 2008



http://xlogo.tuxfamily.org

Chapter 1

Introduction

Logo is a programming language developed in the 1960's by Seymour Papert. Papert was the developer of an original and highly influential theory on learning called "constructionism" which could be summarised with the expression: "learning by doing".

Logo is a really good language to develop mathematical and logic skills. It is an excellent language to begin learning with, as it teaches the basics of things like loops, tests, procedures, etc. The user moves an object called a "turtle" around the screen using commands as simple as forward, back, right, and so on. As it moves, the turtle leaves a trail behind it, and so it is therefore possible to create drawings. The fact that the user can give the turtle orders in a very natural language makes Logo very easy to learn. More advanced usage is possible too with operations on lists, words or files.

XLogo is a Logo interpreter, it means that the user's instructions are executed directly. The user can see their errors on screen immedately. This very intuitive graphical approach makes Logo an ideal language for beginners, especially children!

The main adress for the XLogowebsite is

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

Here you can download both the documentation and the software. You can also find many examples created with XLogo and you will be able to judge XLogo's capacity.

XLogo now supports ten languages (arabic, asturian, english, esperanto, french, galician, greek, german, portuguese and spanish) and is written in Java - a programming language with the benefit of being cross-platform. Therefore XLogo will run on Linux, Windows or MacOS machines without problems.

XLogo is licensed under the GPL: Hence, it is free software and users have four freedoms:

- Freedom 1: The freedom to run the program for any purpose.
- Freedom 2: The freedom to study and modify the program.
- Freedom 3: The freedom to copy the program so you can help your neighbour.
- Freedom 4: The freedom to improve the program, and release your improvements to the public, so that the whole community benefits.

Manual structure:

This manual will help you to discover XLogo.

- In the first part, different menus and interface options are explained.

- Then, some chapters presenting the most important instructions to begin using XLogo. The first are very easy and then, complexity grows. Sometimes, at the end of a chapter, some exercices are presented. Their solutions can be found in appendix D.
- Then, a sequence of different themes is offered for advanced users.
- In appendix A, you'll find a complete description of all XLogo's primitives.

This manual exists under several formats:

- PDF: http://downloads.tuxfamily.org/xlogo/downloads-en/manual-en.pdf
- Zipped HTML: http://downloads.tuxfamily.org/xlogo/downloads-en/manual-html-en.zip
- LaTeX2$_\varepsilon$: Manual Source: http://downloads.tuxfamily.org/xlogo/downloads-fr/manual-src-en.zip
- JavaHelp: Menu Help-Online Manual in XLogo

Contents

Chapter 2

Install XLogo

- First of all, you will have to install the Java Runtime Environment on your computer. Go to this page:

  `http://java.sun.com/javase/downloads/index.jsp`

  Download the JRE (Java Runtime Environment) which corresponds to your operating system (Windows, Linux ...), and install it.

- You have to download the file `xlogo.jar` at the address:

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

  Otherwise, you can go on the XLogo website, at the address `http://xlogo.tuxfamily.org`, choose english language and then click on the downloads menu.

2.1 XLogo Configuration

2.1.1 Linux Environment

Under Ubuntu 8.04:

1. To install JAVA:
   - System -> Administration -> Synaptic Package Manager
   - Install the package `sun-java6-jre`
2. To open the file `xlogo.jar` double-clicking:
   - Right click on `xlogo.jar`, Properties
   - Tab "Open With": Choose Sun Java 6 Runtime
3. To associate extensions `lgo` to XLogo:
   - Right click on `xlogo.jar`, Properties
   - Tab "Open With"
   - Button "Add"
   - Field "Use a custom command", write:

     `java -jar path_to_xlogo.jar`

Note: XLogo is included in distribution OpenSuse.

2.1.2 Windows Environment

In theory, if you double-click on the XLogo icon, the program should launch. If this is the case, go on to the next section. If not, and another application is launched instead (something like winzip, perhaps), this is because .jar files are in fact .zip files, and these are themselves executable (ie a program can be launched by clicking on them). If your computer opens a program like winzip, it is because from its point of view files with a .jar extension can only be opened with that program. You therefore have to deactivate the association of that program with .jar files. To do that, follow these steps for Windows XP (some paths may differ depending on the flavour of Windows you are running, and you will have to adjust them):

1. Start -> Control Panel -> Switch to Classic View -> Folder options
2. Click on the tab File Types (the third tab)
3. Find in the list of registered file types those connected with jar files (jar files, executable jar files, jar archive, etc)
4. Click the file type, and then click Advanced...
5. A new window will appear: click on Open, and then Edit...
6. Click on Browse... and navigate to javaw.exe; this is usually

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

7. The path "c:\Program Files\java\j2re1.4.1\bin\javaw.exe" will then appear in the field *Application used to perform action:*. You need to make an addition to the end of this, so that it reads:

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

   (note that there is a space on either side of -jar).
8. Finally, close all the dialogue windows. Now all you should have to do is to double-click on the file icon to launch XLogo!

If that still doesn't work, there is a second possibility. Open an MSDOS box (on XP: Start -> All Programs -> Accessories -> Command Prompt), and then type in the following command:

```
java -jar \path\to\XLogo

For example : java -jar c:\windows\office\xlogo.jar
```

(if `xlogo.jar` is located in this folder).

If you find it annoying to have to keep typing this command, type it into a text file and save it as (say) xlogo.bat. You can then just double-click on xlogo.bat to launch XLogo.

Associating files with the extension .lgo with XLogo

Files with the extension .lgo will not usually be recognised by your computer, and when you double-click on them, a dialogue box will appear asking you which application should be used to open files with the .lgo extension. Select `other` and then give the path to `javaw.exe`

Usually, this will be: `C:\Program Files\java\j2re1.4.1\bin\javaw.exe`

You will have to input a name to designate files with an `.lgo` extension.
For example: `Logo Files`
To set this up as a default on Windows XP, follow the steps below:

1. Start -> Control Panel -> Switch to Classic View -> Folder options
2. Click on the tab File Types (the third tab)
3. Find in the list of registered file types those connected with jar files (jar files, executable jar files, jar archive, etc)
4. Click the file type, and then click New
5. Type the extension .lgo into the File Extension box, and click OK
6. Click on the newly-added LGO entry in the list of Registered file types, and then click Advanced...
7. A new window will appear: click on New...
8. Under Action, enter "open", and then click on Browse... to navigate to javaw.exe; this is usually

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

9. Click on Open to add the path to the Actions box of the Edit File Type dialogue.
10. Click on open, and then Edit...
11. The path "c:\Program Files\java\j2re1.4.1\bin\javaw.exe" will be in the field *Application used to perform action:*. You need to make an addition to the end of this, so that it reads:

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

12. Finally, close all the dialogue windows. Now all you should have to do is to double-click on the file icon to launch XLogo!

2.2 XLogo Updates

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

To update XLogo, you just have to replace the file `xlogo.jar` with its new version. If you want to receive an alert whenever a new version is published, you can subscribe to XLogo's RSS feed. Its address is

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

Several softwares can manage RSS feeds, if you're not familiar with this technique, the easiest way is to use Mozilla Thunderbird:

- Menu Edit - Account Settings...
- Button "Add Account"
- "RSS News & Blogs", Next
- Account Name: "RSS Feeds" for example
- Buttons "Next" and "Finish"
- In the main window "Account Settings", Select "RSS Feeds" on the left menu and click on button "Manage Subscriptions".
- Button "Add"
  - Feed URL: `http://xlogo.tuxfamily.org/rss.xml`
  - Check item "Show the article summary instead of loading the web page"

It's done, with the button "Get Mail", you can receive XLogo News in the same way you receive mails.

2.3 Uninstall

To uninstall XLogo, all that needs to be done is to delete the file `xlogo.jar` and the configuration file `.xlogo`, which is located in your home directory (`/home/votre_login` for Linux users, or `c:\windows\.xlogo` for Windows users).

Chapter 3

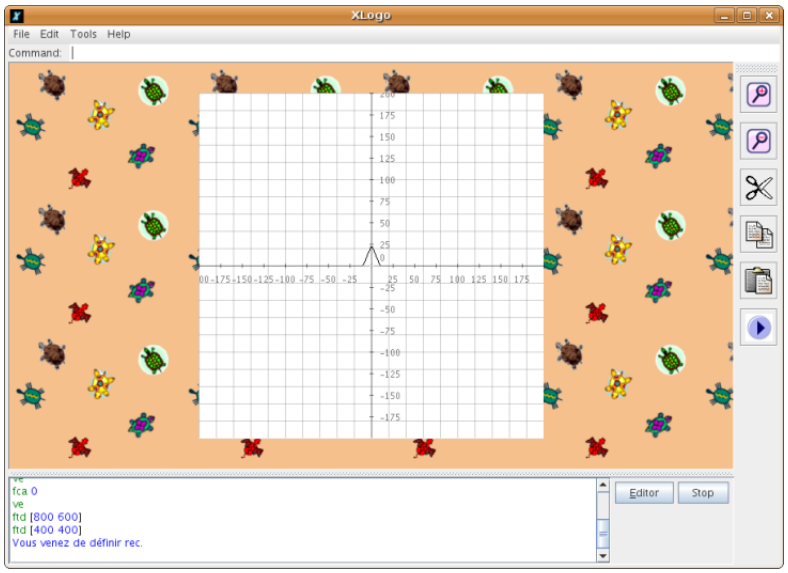
Interface features:

3.1 First run

The first time you launch XLogo (or if you have deleted the file `.xlogo` - Read Section 2.3), a dialog box will ask you for your language.

Then, you could modify the language with the Preferences Dialog Box (Read Section 4.3).

3.2 The main window

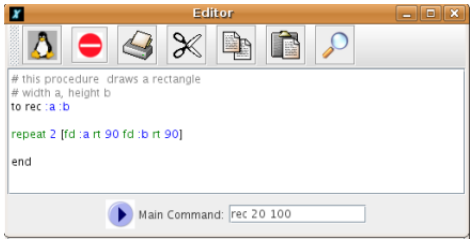


- Along the top, there are the usual menus **File Edit Options and Help**
- Just below this is **the command line**, which allows the logo instructions to be applied.
- In the middle of the screen is **the drawing area**.
- On the right of the drawing area, a **tool bar** allows the user to do several actions:
  - Zoom in/out.

  - Edit (cut/copy/paste)
  - The "play" button launches the main command defined in the editor.

- At the bottom is the **command history**, which shows every command entered, and the associated response. To quickly recall a command which has already been entered, there are two options: you can either click on the old command in the history, or you can click several times on the upper scroll-arrow until the desired command appears. The upper and lower scroll-arrows in fact allow you to navigate through all the commands that you have already entered (very practical).
- To the right of the history are two buttons: **STOP and EDITOR**.
  - Button STOP interrupts the execution of the program.
  - Button EDITOR allows the procedure editor to be opened.

3.3 The procedure editor



There are three ways to open the editor:

- Enter `ed` on the command line at the top of the screen. The editor will then open to show all the procedures already defined. If you only want to edit specific procedures, enter:
  `ed [procedure_1 procedure_2 ...]`
- Press the Editor button on the main screen.
- Use the keyboard shortcut Alt+E.

These are the different buttons that you will find in the editor:

Save the changes made in the editor and then close it. It is this button that you have to press each time you want to apply newly entered operations. If you prefer, you can use the keyboard shortcut ALT+Q.

Close the editor without saving any of the changes made there. You can also use the shortcut ALT+C.

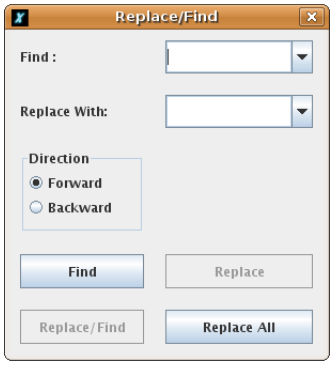
Print the contents of the editor.

Copy the selected text to the clipboard.

Cut the selected text to the clipboard.

Paste the selected text from the clipboard.

 Open a Replace/Find Dialog Box for the procedure Editor.



 At the bottom of the editor, a text field allows the user to define a main command. This command is the general instruction that launches the program. It can be accessed with the "play" button from the main window's tool bar. This command is saved and then restored when the editor and all its content are recorded in a Logo format file (`.lgo`)

IMPORTANT:

- Note that clicking on the close button (x) in the window titlebar will have no effect! Only the two main buttons will allow you to quit the editor.
- To delete one or more unwanted procedures, use the primitives `er` and `erall`, or use in the menu bar, Tools→ Procedure Eraser.

3.4 Quit XLogo

To quit XLogo, you can choose in the menu bar, **File - Quit** or click on the close button in the window titlebar. A dialog box wil ask you if you really want to quit.

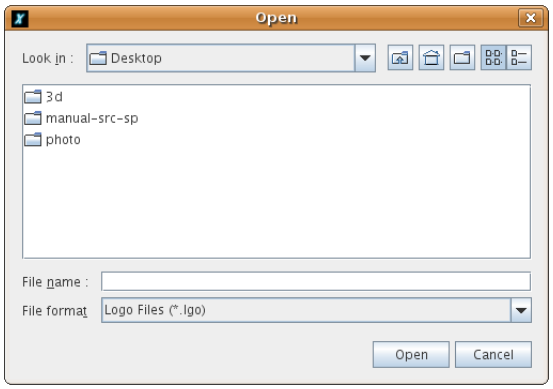
Chapter 4

Menu options:

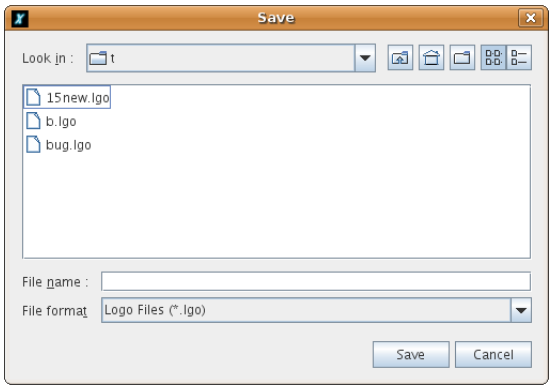
4.1 "File" Menu

- **File→New**: delete all procedures and variables. You create a new workspace.

- **File→Open**: open a previously saved logo file.



- **File→Save as...** save the current procedures under a different name.



- **File→Save**: save the procedures in the current file.

- **File→Capture image→Save image as...** : allow the image to be saved in jpg or png format. If you wish to select only a part of the image, you can define a bounding box by dragging the mouse on the drawing area.

- **File→Capture image→Print image**: allows the image to be printed. In the same way as above, you can select an area to print.

- **File→Capture image→Copy image into the clipboard**: put the image into the system clipboard. Just as for printing and recording, you can select an area of the image. This functionality works very well under the Windows environments. On the other hand, it does not work under Linux (the clipboard has a different behaviour).

- **File→Text→Save As... (RTF)** : save the command history in RTF format (color and text format are preserved).

- **File→Quit**: quit the XLogo application.

4.2 "Edit" Menu

- **Edit→Copy**: copy the selected text to the clipboard.

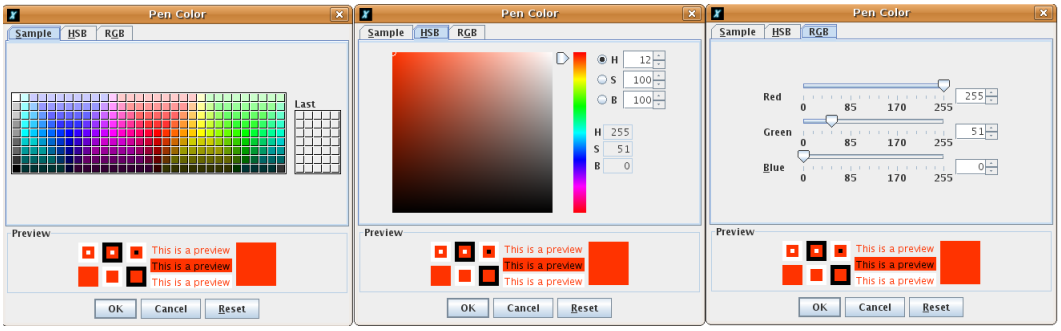
- **Edit→Cut**: cut the selected text to the clipboard.

- **Edit→Paste**: paste the text contained in the clipboard into the command line.

- **Edit→Select All**: select all the text in the command line.

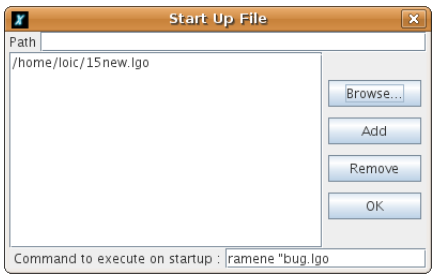
4.3 "Tools" Menu

- **Tools→Pen Color**: allows the colour with which the turtle will draw to be chosen from a palette of colours. Also accessible via the command `setpc`.

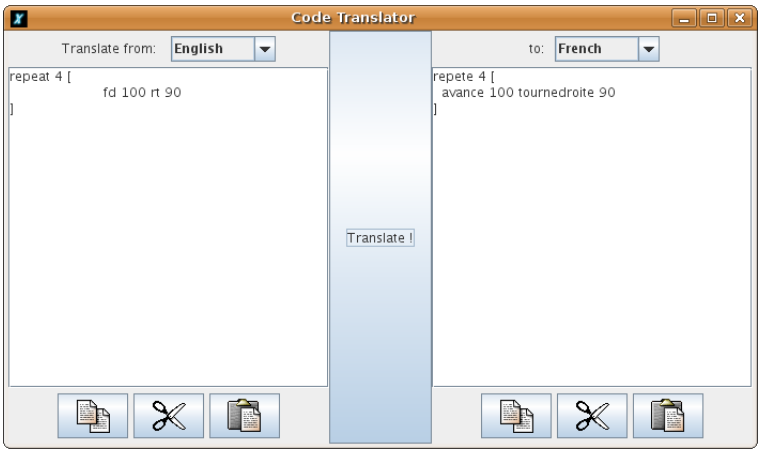


- **Tools→Screen Color**: set the color of the screen background. Accessible via the primitive `setscreencolor`.

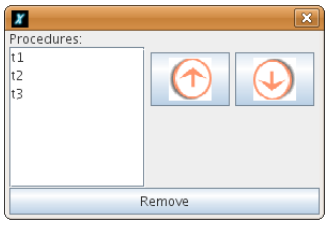
- **Tools→Start Up File**: allows the path to "start-up" files to be defined. Any procedures contained in these *.lgo format files will then become "pseudo-primitives" in the XLogo language. They cannot be edited or changed by the user. You can thus define personalised primitives.



- **Tools→Code Translator**: allows code translation from one language to another. In fact, very useful when you want to use an XLogo example written in another language.

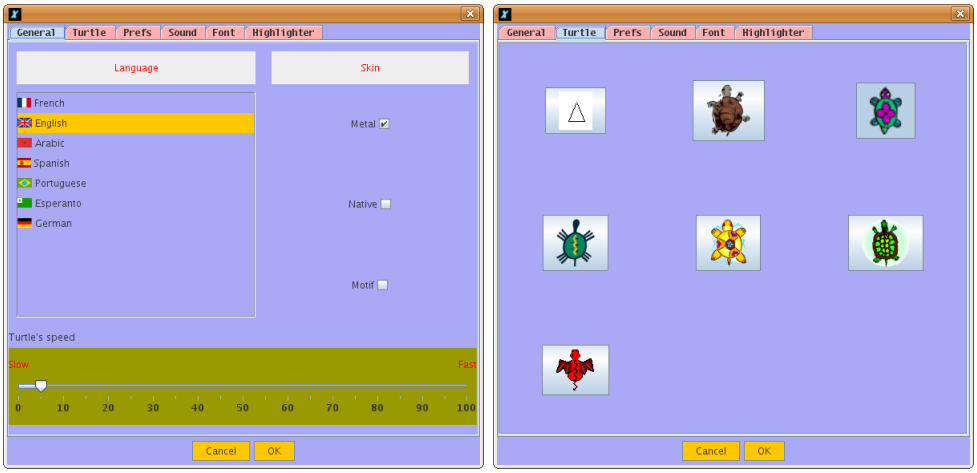


- **Tools→Procedure Eraser**: with this dialog box, you can delete some procedures. You can also modify the order of appearance of the procedures in the editor.

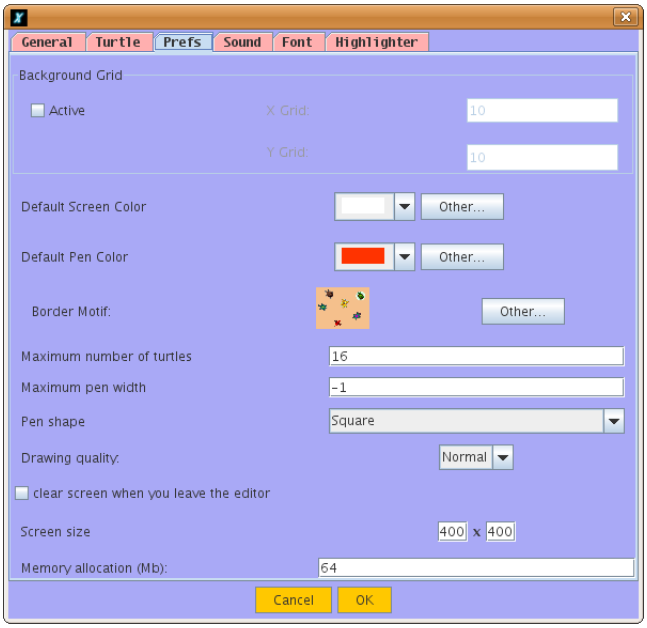


- **Tools→Preferences**: opens a dialog box in which you can configure several things:
  - **General Tab:**
    - → **Language** : allows language to be chosen. Note that the primitives differ in each language.
    - → **Look**: allows the "look" or skin of XLogo to be defined. Metal, Native Java and Motif styles are available.
    - → **Drawing speed**: if you prefer to see all the turtle's movements, you can slow it down by using the slider bar on the first tab.

.



– **Turtle Tab**: On the second tab, you can choose your preferred turtle.



– **Options Tab**: On the third tab, many options:

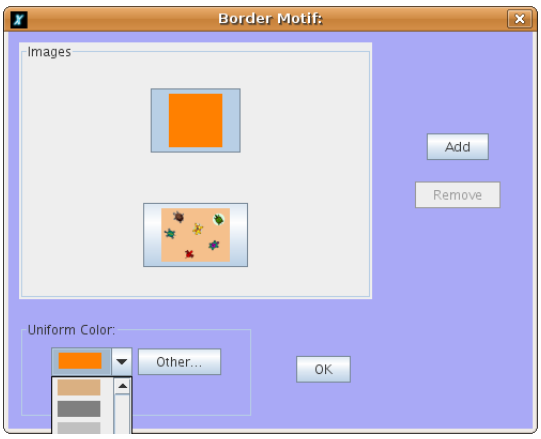
→ **Background grid**: You can choose to draw a grid on the background drawing screen. You can define the width and the height of a square of the grid and the grid color.

→ **Background axis**: You can choose to draw horizontal axis or vertical axis on the background drawing screen. You can define the distance between two divisions and the axis color.

→ **Default screen color**: You can define a default screen color.

→ **Default pen color**: You can define a default pen color.

→ **Border motif**: You can choose your own motif for the drawing area's border (an image or a uniform color).



→ **Maximum pen width**: You can choose the maximum pen width allowed. If you don't want to use this option, put -1.

→ **Pen shape**: You can choose the shape of the pen, round or square.

→ **Drawing quality**: Finally, you can choose the accuracy of the drawing line. In high quality, pen edges will be smoothed. But remember that by increasing the quality you will lose some execution speed.

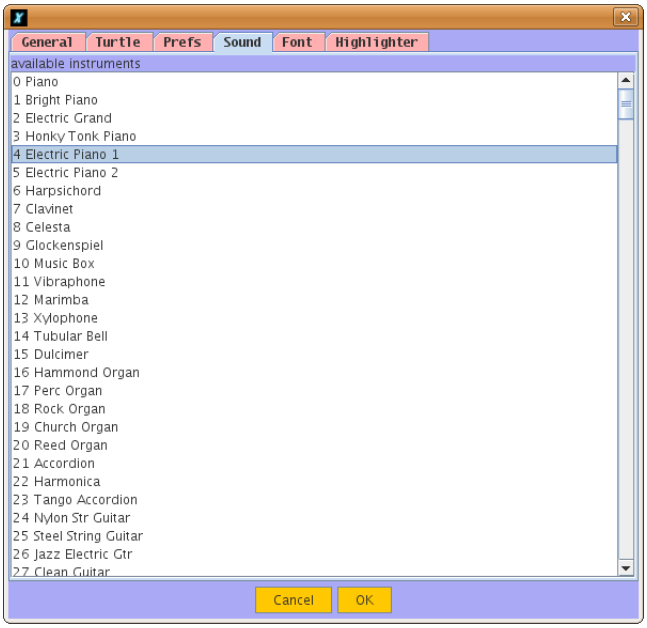
→ **Maximum number of turtles**: You can choose the maximum number of turtles available in mode multiturtle.

→ **Clear screen when closing the editor**: You can choose if you want to clear the screen when you leave the editor.

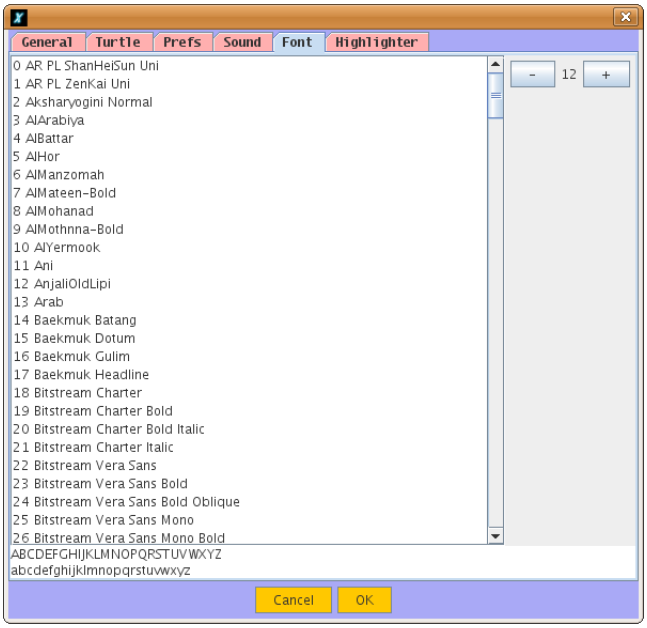
→ **Screen size**: You can choose a personal size for the drawing area. Default size is 1000 by 1000 pixels. Be careful, when you increase the size of the drawing area, you may need to increase the memory size of XLogo, (an error message will pop up).

→ **Memory allocation:** You can change the memory space allocated to XLogo. Default size is 64MB. You might have to increase this if you want to work on a bigger drawing area. If you modify this parameter, you must restart XLogo so that the change takes place. Be careful, do not over increase this parameter since it could considerably slow your system down.

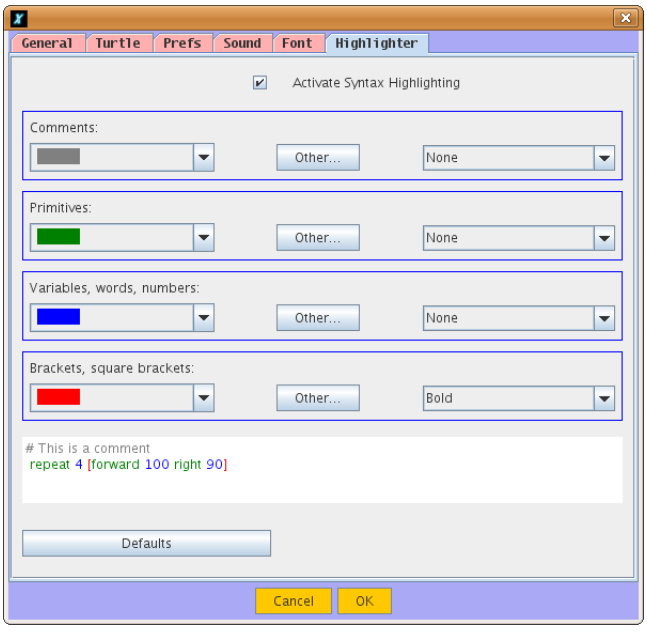
→ **TCP port number**: You can modify the default TCP port used for networking. (See p.125)



– **Sound Tab:** On the fourth tab, you can choose an instrument for your MIDI interface. If you experience some detection problems, and the list is empty, have a look at the FAQs at the end of the manual. This function can be accessed with the primitive `setinstrument`.



– **Font Tab**: On the fifth tab, you can choose the font for the interface.

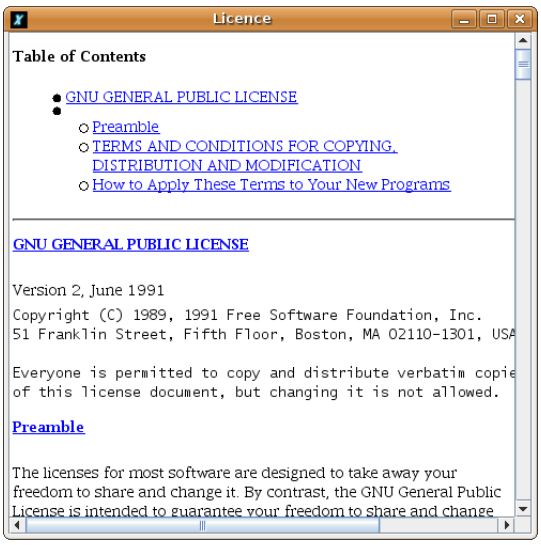


– **Highlighter Tab**: You can choose active syntax highlighting and define your own highlight colors.

4.4 "Help" Menu

- **Help→Online Manual**: Displays the reference manual of XLogo, accessible only by internet.

- **Help→Licence**: shows the GPL license under which this software is distributed.



- **Help→Translation**: shows a translation of the above license. This translation has no official standing - this belongs only to the English version, and the translation is provided here only as an aid to understanding.

- **Help→Translate XLogo**: this dialog box allows to consult/modify/complete XLogo transaltions for any language (messages and primitives).

Otherwise, You can create a new translation for a new language if you want to. In every case, send me the generated file at `loic@xlogo.tuxfamily.org`

- **Menu – > About**: The standard thing .... and xlogo.tuxfamily.org for your bookmarks !! o:)


About XLogo
XLogo version: 0.9.91pre
Web Site: http://xlogo.tuxfamily.org
You can contact me at:
loic@xlogo.tuxfamily.org
OK

Chapter 5

Conventions adopted by XLogo

This section sets out some key points about the LOGO language itself, and about XLogo specifically.

5.1 Commands and their interpretation

The LOGO language allows certain events to be triggered by internal commands - these commands are called *primitives*. Each primitive may have a certain number of parameters which are called *arguments*. For example, the primitive `cs`, which clears the screen, takes no arguments, while the primitive `sum` takes two arguments.

`print sum 2 3` will return 5.

LOGO arguments are of three kinds:

- **Numbers:** some primitives expect numbers as an argument: `fd 100` is an example.
- **Words:** Words are marked by an initial ". An example of a primitive which can take a word argument is `print`.

  ```
  print "hello
  ```

  This command displays `hello`. If you forget the ", the interpreter will return an error message. In effect, `print` expects an argument, or for the interpreter, `hello` does not represent anything, since it is not a number, a word, a list, or an already defined procedure.
- **Lists:** these are defined between brackets.

Note: Numbers are treated in some instances as a numeric value (eg: `fd 100`), and in others as a word (eg: `print first 12` writes `1`).

Several primitives have a general form, it means they could be used with an undefined number of arguments. All those primitives are on the table below:

`print`	`sum`	`product`	`or`
`and`	`list`	`sentence`	`word`

To notify the interpreter that these primitives will be used in their general form, we have to write our command into parenthesis, look at those examples below:

```
 print (sum 1 2 3 4 5)
15

(list [a b] 1 [c d])
I don't know what to do with [[a b] 1 [c d]]?

if (and 1=1 2=2 8=5+3) [fd 100 rt 90]
```

5.2 Procedures

In addition to these primitives, you can define your own commands. These are called *procedures*. Procedures are introduced by the word `to` and conclude with the word `end`. They can be created using the internal XLogo procedure editor. Here is a short example:

```
to square
repeat 4[forward 100 right 90]
end
```

These procedures can take advantage of arguments as well. To do that, variables are used. A variable is a word to which a value can be assigned. Here is a very simple example:

```
to total :a :b
print sum :a :b
end

total 2 3
5
```

5.3 Specific character \

The specific character \ (backslash) allows the creation of words containing blank or line feed symbols. If \n is used, the phrase skips to the following line, and \␣ followed by a blank means a blank in a word. Example:

```
pr "xlogo\ xlogo
xlogo xlogo
pr "xlogo\nxlogo
xlogo
xlogo
```

You can therefore only write the \ symbol by typing \\.

Similar behaviour, characters ( ) [ ] # are specific delimiters of Logo. If you want to use them in a word, you just have to add the character \before.

All \ only symbols are ignored. This remark is especially important for the use of files.

To set your current directory path to `c:\My Documents`:

```
setdir "c:\\My\ Documents.
```

Please note the use of \␣ to notify the space between My and Documents. If, you forget the double backslash, the path that will be defined will then be `c:My Documents` and the interpretor will send you an error message.

5.4 Case-sensitivity

XLogo makes no distinctions on case as regards procedure names and primitives. Thus, with the procedure `square` as defined earlier, whether you type `SQUARE` or `sQuaRe`, the command interpreter will translate it correctly and execute `square`. On the other hand, XLogo is case-sensitive on lists and words:

```
print "Hello ----> "Hello (the initial capital H is retained)
```

5.5 Operators and syntax

There are two ways to write certain commands. For example, to add 4 and 7, there are two possibilities: you can either use the primitive `sum` which expects two arguments: `sum 4 7`, or you can use the operator +: `4+7`. Both have the same effect.
This table shows the relationship between operators and primitives:

sum	difference	product	quotient
+	-	*	/
or	and	equal?	
| (ALT GR+6)	&	=	

There are two other operators with no associated primitive:

- Operator "Less than or equal to": `<=`
- Opérator "Greater than or equal to": `>=`

Note: The two operators | and & are specific to XLogo. They do not exist in traditional versions of LOGO. Here are some examples of usage:

```
pr 3+4=7-1 ----> false
pr 3=4 | 7<=49/7 ----> true
pr 3=4 & 7=49/7 ----> false
```

Chapter 6

Basic primitives

Level: newbie

To move the turtle on the drawing area, we use predefined commands called "primitives". In this chapter, we're going to discover the basic primitives allowing us to pilot the turtle on the drawing area.

6.1 New primitives

- `forward number` `fd 50`
  Moves the turtle forward number of steps in the direction it is currently facing.

- `back number` `bk 100`
  Moves the turtle backwards number of steps in the direction it is currently facing.

- `right number` `rt 90`
  Turns the turtle towards the right in relation to the direction it is currently facing.

- `left number` `lt 45`
  Turns the turtle towards the left in relation to the direction it is currently facing.

- `clearscreen` `cs`
  Erases the drawing area.

- `showturtle` `st`
  The turtle is visible on screen.

- `hideturtle` `ht`
  The turtle is invisible (drawing is faster).

- `penup` `pu`
  The turtle won't draw a line when it moves.

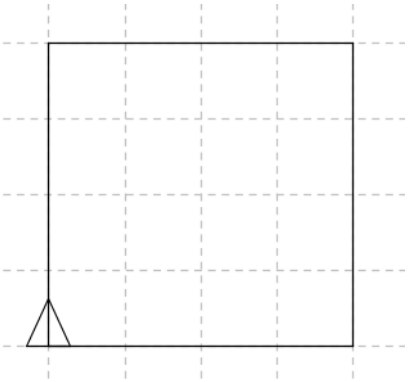
- `pendown` `pd`
  The turtle will draw a line when it moves.

- `repeat integer list` `repeat 5[fd 50 rt 45]`
  Repeat instructions contained in the list.

6.2 Drawing a regular polygon

In this part, we'll learn to draw a square, equilateral triangle, and a regular polygon in general....

6.2.1 Square



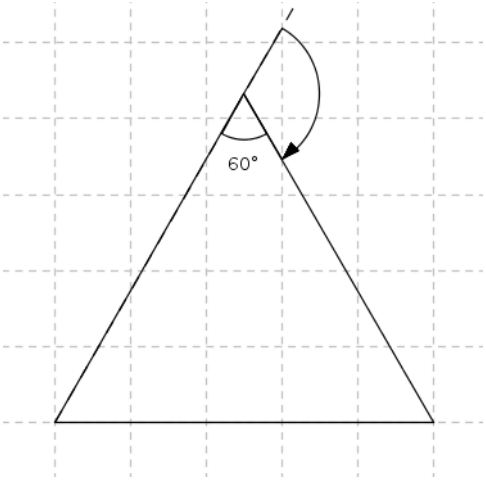
To draw this square, we're going to write:

```
fd 200 rt 90 fd 200 rt 90 fd 200 rt 90 fd 200 rt 90
```

We can see that we repeat 4 times the same instructions. Therefore, a better syntax:

```
repeat 4[fd 200 rt 90]
```

6.2.2 Equilatéral triangle



Here, we'll learn how to draw this equilatéral triangle (all three sides have 150 steps lengths).

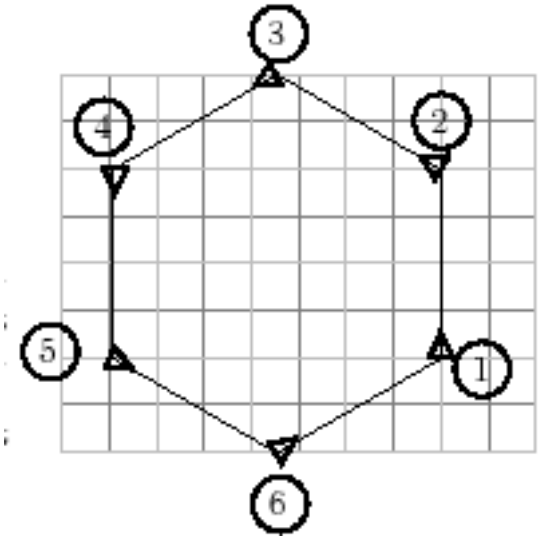
The command will have this form:

```
repeat 3[fd 150 rt ....]
```

We must determinate the angle. In an equilateral triangle, all three internal angles are equal to each other and so are each 60 degrees. The turtles turns outside the triangle. Hence, the value for this angle is 180-60=120 degrees. The valid command is:

```
repeat 3[fd 150 rt 120]
```

6.2.3 Hexagon




```
repeat 6[fd 80 rt ....]
```

When the turtle has finished its moves, it has made one tour from its initial position to its final position. This is done using 6 steps. Therefore, each angle value is equal to $\frac{360}{6} = 60^\circ$.

The valid command is: `repeat 6[fd 80 rt 60]`

6.2.4 Drawing a regual polygon in general

In fact, the reasoning makes us think that to draw a polygon with n sides, the turtle will have to turn from an angle whose value is 360 divided by n. For example:

- To draw a regular pentagon with side length 100:

  ```
  repeat 5[fd 100 rt 72]    (360:5=72)
  ```

- To draw a nine sided polygon with side length 20:

  ```
  repeat 9[fd 20 rt 40]    (360:9=40)
  ```

- To draw hum... a regular 360-gon with side length 2:

  ```
  repeat 360[fd 2 rt 1]
  ```

  This form is really near a circle!

- To draw an heptagon with side length 120:

  ```
  repeat 7[fd 120 rt 360/7]
  ```

6.3 Saving a procedure

Because we do not wish to rewrite each time the same instructions to draw a square, a triangle... it's better to save these instructions into a "procedure". To define a procedure, open the editor. A procedure starts with the keyword `to` and finishes with the keyword `end`. To define a square procedure:

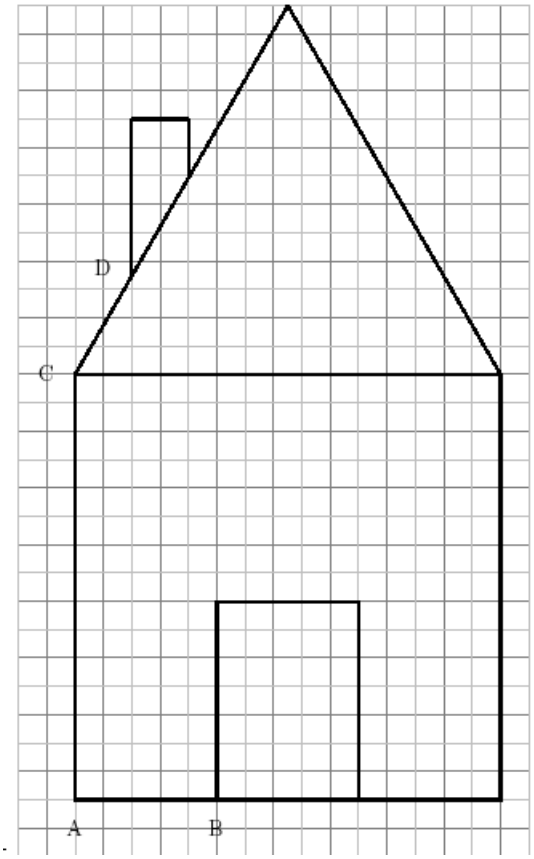
```
to square
repeat 4[fd 100 rt 90]
end
```

then we close the editor by clicking on the turtle button. It will save the editor contents. Now, when we write `square` in the command line, a square appears on screen!

6.4 Exercice ...

Each square is 10 steps wide. Try to draw this image defining eight procedures:

- A procedure called `square` that draws the main square of the house.
- A procedure called `triangle` that draws the roof as an equilateral triangle.
- A procedure called `door` that draws the rectangular door.
- A procedure called `chimney` that draws the chimney.
- A procedure called `move1` that allows the turtle to move from point A to point B.
- A procedure called `move2` that allows the turtle to move from point B to point C.
- A procedure called `move3` that allows the turtle to move from point C to point D. (Warning: you'll have to pen up!)
- A general procedure called `house` that draws the whole house using all previous procedures.



Chapter 7

Using coordinates

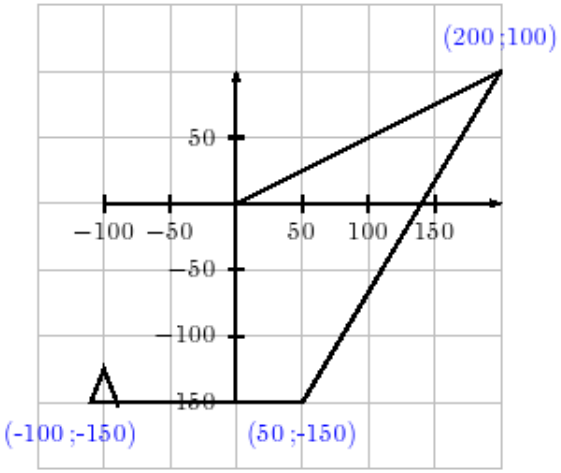
Level: Newbie

7.1 Presentation

In this chapter, we're going to discover the primitive `setposition, setpos`. The drawing area has two axis that allows to determine each point using the cartesian coordinate system. The origin is the center of the drawing area.

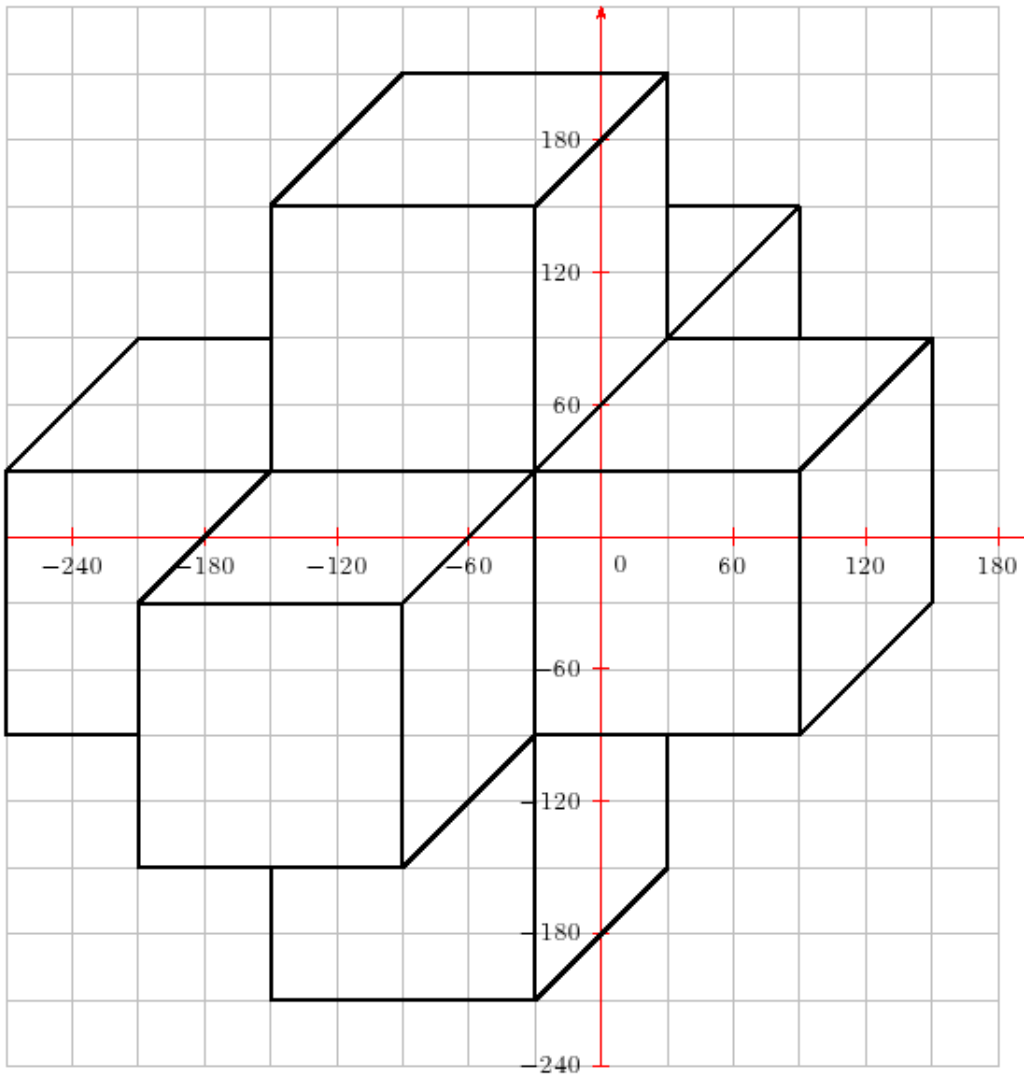
`seposition list` `setpos [100 -250]`
Moves the turtle to the co-ordinates specified by the two numbers in the list

A little exemple:
`cs setpos [200 100] setpos [50 -150] setpos [-100 -150]`



7.2 Exercice:

Try to draw this picture using only the following primitives: `setpos`, `cs`, `pu`, `pd`.



Chapter 8

Variables

Level: Newbie

Sometimes, it's necessary to draw a figure on a different scale. For example, if we want to draw a square with side length 100, a square with side length 200 and a square with side length 50, we need actually three different procedures for each square.

```
to square1
repeat 4 [forward 100 right 90]
end
to square2
repeat 4 [forward 200 right 90]
end
to square3
repeat 4 [forward 50 right 90]
end
```

We can see immediately that it would be easier to define a single procedure waiting for an argument: the side length. For example, `square 200` should draw a square with side length 200, `square 100` should draw a square with side length 100 ... It's time to introduce the variable notion!

8.1 Examples

To draw a square with side length 100, we write in the editor:

```
to square
repeat 4 [forward 100 right 90]
end
```

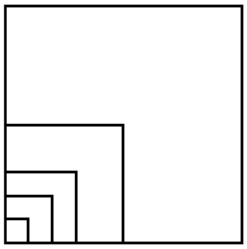
We just have to modify this procedure by doing:

- We add `:c` at the end of the definition line. This indicates that the procedure is now waiting for an argument called `:c`
- We replace the side length 100 by the variable name `:c`

We obtain

```
to square :c
repeat 4 [forward :c right 90]
end
```

Therefore, if we write: `square 100 square 50 square 30 square 20 square 10`



8.2 Drawing a rectangle with chosen dimension

We define here a procedure called `rec` depending on two variables representing the dimensions of the rectangle. Hence `rec 200 100` will draw a rectangle with height 200 and width 100.

```
to rec :lo :la
repeat 2 [forward :lo right 90 forward :la right 90]
end
```

Make some examples:

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

If we give to the procedure `rec` only one number, the interpreter will send an error message indicating that the procedure is waiting for a second argument.

8.3 Drawing at different scales

We saw how to draw a square, and a rectangle with different sides. Now, we return to our house example p. 32 and we're going to modify the code to draw this house at any chosen scale.

The objective is to send an argument to the procedure `house` and according to this parameter, the house will be smaller or bigger.

- `house 1` will draw the house in real size.
- `house 0.5` will draw the house at scale 0.5.
- `house 2` will draw the house with double proportion.

In real size, the procedure square was:

```
to square
repeat 4 [forward 150 right 90]
end
```

All the initial dimensions are multiplied by the scale. Hence, the procedure `square` becomes:

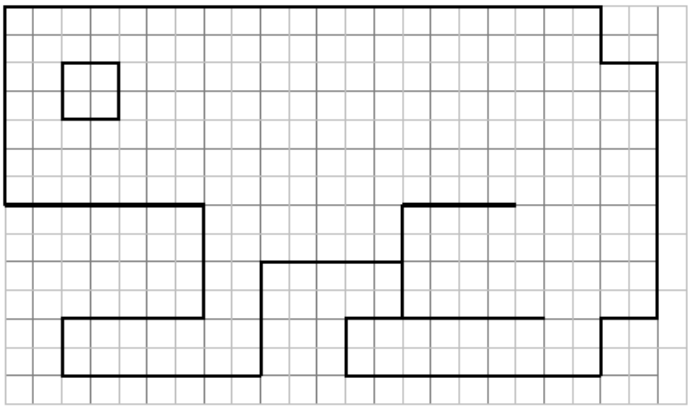
```
to square :c
repeat 4 [forward 150*:c right 90]
end
```

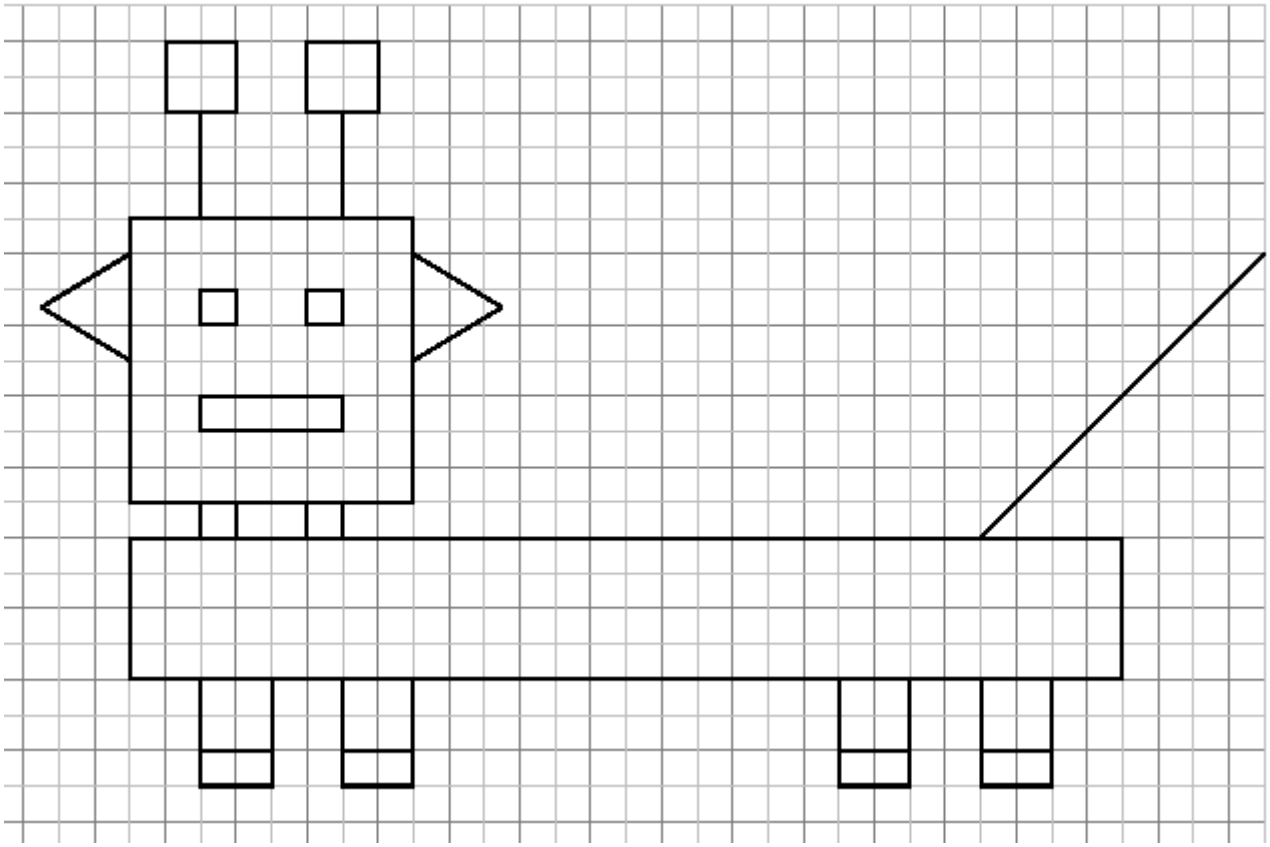
Therefore, when we'll write `square 2`, the square will have for side length $150 \times 2 = 300$. Proportions are well respected! In fact, we can see that we just need to modify all procedures replacing the length according to this rule:
`fd 70` becomes `fd 70*:c`
`fd 45` becomes `fd 45*:c`

```
to carre :c
repeat 4[forward 150*:c  right 90]
end

to tri :c
repeat 3[forward 150*:c right 120]
end

to porte :c
repeat 2[forward 70*:c right 90 forward 50*:c right 90]
end

to che :c
forward 55*:c right 90 forward 20*:c right 90 forward 20*:c
end

to dep1 :c
right 90 forward 50*:c left 90
end

to dep2 :c
left 90 forward 50*:c right 90 forward 150*:c right 30
end

to dep3 :c
penup right 60 forward 20*:c left 90 forward 35*:c pendown
end

to ma :c
carre :c dep1 :c porte :c dep2 :c tri :c dep3 :c che :c
end
```

8.4 Exercice:

Try to generate the following drawings at different scales.





Chapter 9

Recursion

Level: Medium

Logo programming often uses a technic called recursion. In this chapter, first, we'll explore recursion with some simple examples. Then, we'll go further with the drawing of a fractal curve called the Van Koch snowflake". First of all:

A procedure is recursive if it calls itself.

9.1 With drawing area.

9.1.1 First example:

```
to ex1
rt 1
ex1
end
```

This procedure is recursive because the procedure `ex1` is called on the last line. While executing, we can see that the turtle turns on itself forever. To break the program, we must click on the STOP button.

9.1.2 Second example:

Here are three new primitives:

- `wait number` wait 60
  Pause the program during the number of 60^{th} seconds.
  For example, `wait 120` will pause the program for two seconds.
- `penerase` penerase
  When the turtle moves, it erases all it encounters instead of drawing.
- `penpaint` penpaint
  Returns to classic mode. The turtle draws lines when it moves.

```
to ex2
fd 200 penerase wait 60
bk 200 penpaint rt 6
ex2
end
```

Now, we can execute the program. On each second, the same procedure is repeated. We obtain the seconds of a clock!

9.2 With the text zone

9.2.1 First example:

The primitive `print, pr` displays text in the text zone. `print` is waiting for an argument: a list or a word. Eg: `pr "hello pr [I write what I want]` (Don't forget the quote " when you want to write only a word.)

```
to ex3 :n
print :n
ex3 :n+1
end
```

Run the command: `ex3 0` and stop the program with the STOP button.
Modify the program to dispay the numbers with an interval of 2.

We now want to display all integers greater than 100 which are divisable by 5. We just have to modify the program:

```
to ex3 :n
print :n
ex3 :n+5
end
```

and then run: `ex3 100`

9.2.2 Breakout test

Try the following lines:

```
if 2+1=3 [print [it is true]]
if 2+1=4 [print [it is true]][print [it is false]]
if 2+5=7 [print "true][print "false]
```

If you doesn't understand yet the syntax of the primitive `if`, refer to the annex.

```
to ex3 :n
if :n=100 [stop]
print :n
ex3 :n+1
end
```

Then run the command `ex3 0`
Modify the program to display integers between 55 and 350 which are divisable by 11.

9.3 A fractal example: Van Koch snowflake

Using recursion, it's very easy to generate in LOGO some special curves called fractals in mathematics.

Here are the first steps to create the Van Koch broken line:

Between two steps:

1. each segment is divided into three equal part.
2. an equilateral triangle is drawn on the middle segment.
3. finally, this middle segment is erased.

What is important: Let's have a look at step 2, we can see that the broken lines contains four identical motifs corresponding to precedent step with a 3 lesser size. Here we have found the recursive structure of the fractal.

Let's call $L_{n,\ell}$ the motif of size ℓ, corresponding to step n.
To draw this motif:

1. We draw $L_{n-1,\ell/3}$
2. We turn left 60 degrees
3. We draw $L_{n-1,\ell/3}$
4. We trun right 120 degrees
5. We draw $L_{n-1,\ell/3}$
6. We trun left 60 degrees
7. We draw $L_{n-1,\ell/3}$

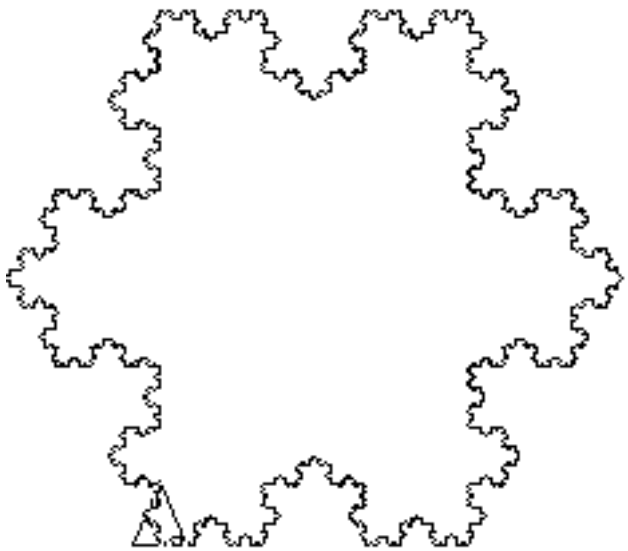
With Logo, it's very easy to write:

```
# :l motif size
# :p step
to line :l :p
if :p=0 [fd :l] [
  line :l/3 :p-1 lt 60 line :l/3 :p-1 rt 120 line :l/3 :p-1 lt 60 line :l/3 :p-1
]
end
```

If we draw an equilateral triangle with three Van Koch lines, we obtain a beautiful Van Koch snowflake.

```
# :l side length
to snowflake :l :p
repeat 3[line :l :p rt 120]
end
```

Then run: `snowflake 200 6`



9.4 Recursion with words

Read p.87 to understand how to use the primitives `word`, `last`, and `butlast`.

Here is a recursive procedure that inverts the characters of a word.

```
to invertword :m
if emptyp :m [output "]
output word last :m invertword butlast :m
end

print invertword "abcde
edcba
```

A palindrome is a word, or a phrase that can be read in both sense (Examples: A man, a plan, a canal: Panama ...).

```
# test if the word :m is a palindrome
to palindrom :m
if  :m=invertword :m [output true] [output false]
end
```

Finally, this little kind program, (Thanks Olivier SC):

```
to palin :n
if palindrom :n [print :n stop]
print (list :n "PLUS invertword :n "EQUAL sum :n invertword :n)
palin :n + invertword :n
end

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

9.5 Calculate a factorial

Factorial of the integer 5 is defined by:

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

For n positive integer, we can note that: $n! = n \times (n-1)!$.
This relation explains the recursive nature of the program:

```
to fac :n
if :n=0[output 1][output :n*fac :n-1]
ent

pr fac 5
120
pr fac 6
720
```

9.6 π Approximation

approximation We can approximate the number π using the formula:

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

with k the number of squareroots. The greater is k, the better is the π .

The formula contains the recursive expression $2 + \sqrt{2 + \ldots \sqrt{2 + \sqrt{2}}}$, so let's code:

```
# k is the number of squareroots
to approxpi :k
write "approximation:\  print (power 2 :k) * squareroot (2- squareroot (calc :k-2))
print "-------------------------
write "pi:\  print pi
end

to calc :p
if :p=0 [output 2][output 2+squareroot calc :p-1]
end

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

We found the first 5 digits! If we're looking for more π digits, we have to allow a better precision with a higher number of digits while computing. Thus, we're going to use the primitive `setdigits`.

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

And now, we have 39 digits...

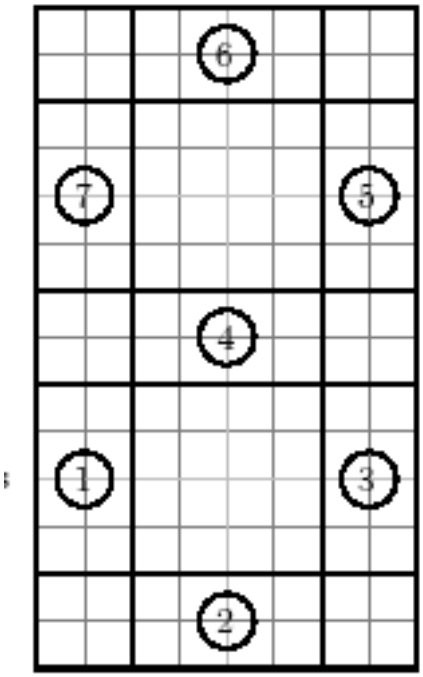
Chapter 10

Create an animation

Level: Medium

This chapter presents two different themes with the goal of creating animation in XLogo.

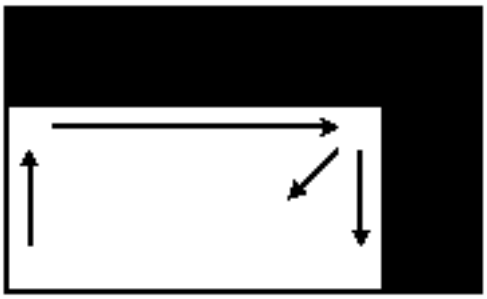
10.1 Calculator's numbers



This theme is based on the fact that every calculator's number could be drawn with the above schema:

- For example, to draw digit 4, we light rectangles 3,4,5,7.
- To draw digit 8, we light rectangles 1,2,3,4,5,6,7.
- To draw digit 3, we light rectangles 2,3,4,5,6.

10.1.1 Filling a rectangular



If we want to draw a filled rectangle with dimensions 100 by 200, a first idea could be to draw a rectangle 100 by 200 then to draw a rectangle 99 by 199, then a rectangle 98 by 198 ... until the rectangle is fully filled.
Let's begin by defining a rectangle with two variables corresponding to width and height

```
to rec :h :w
repeat 2[fd :h rt 90 fd :w rt 90]
end
```

To fill our rectangle, we have to run:
`rec 100 200 rec 99 199 rec 98 198 .....  rec 1 101`

Let's define a procedure for this filled rectangle.

```
to rectangular :h :w
rec :h :w
rectangular :h-1 :w-1
end
```

We test `rectangular 100 200` and we can see there is a problem: The procedure doesn't stop when the rectangle has been filled, it continues infinitely! We must add a breakout test that will detect if width or height is equal to 0. When this condition is realized, we'll ask the program to stop with the primitive `stop`.

```
to rectangular :h :w
if or :h=0 :w=0 [stop]
rec :h :w
rectangular :h-1 :w-1
end
```

Note: Instead of using the primitive `or`, it's possible to use the symbol |, the line becomes:

```
if :h=0 | :w=0 [stop]
```

10.1.2 The program

We must reuse the precedent filled rectangle:

```
to rectangular :h :w
if or :h=0 :w=0 [stop]
rec :h :w
rectangular :h-1 :w-1
end
```

We suppose that the turtle starts from the bottom left corner. We're going to define a procedure called `number` depending on 7 arguments `:a`, `:b`, `:c`, `:d`, `:e`, `:f`, `:g`. When `:a` is equal to 1, we draw the rectangle 1. If `:a` is equal to 0, we don't draw this rectangle. Here is the main idea.

The code:

```
to number :a :b :c :d :e :f :g
# we draw the rectangular 1
if :a=1 [rectangular 160 40]
# we draw the rectangular 2
if :b=1 [rectangular 40 160]
penup right 90 forward 120 left 90 pendown
# we draw the rectangular 3
if :c=1 [rectangular 160 40]
penup forward 120 pendown
# we draw the rectangular 5
if :e=1 [rectangular 160 40]
# we draw the rectangular 4
left 90 penup back 40 pendown
if :d=1 [rectangular 160 40]
# we draw the rectangular 6
right 90 penup forward 120 left 90 pendown
if :f=1 [rectangular 160 40]
# we draw the rectangular 7
penup forward 120 left 90 back 40 pendown
if :g=1 [rectangular 160 40]
end
```

10.1.3 Creating an animation

In this part, we'll define a countdown from 9 to 0.

```
to countd
clearscreen hideturtle number 0 1 1 1 1 1 1 wait 60
clearscreen hideturtle number 1 1 1 1 1 1 1 wait 60
clearscreen hideturtle number 0 0 1 0 1 1 0 wait 60
clearscreen hideturtle number 1 1 1 1 0 1 1 wait 60
clearscreen hideturtle number 0 1 1 1 0 1 1 wait 60
clearscreen hideturtle number 0 0 1 1 1 0 1 wait 60
clearscreen hideturtle number 0 1 1 1 1 1 0 wait 60
clearscreen hideturtle number 1 1 0 1 1 1 0 wait 60
clearscreen hideturtle number 0 0 1 0 1 0 0 wait 60
clearscreen hideturtle number 1 1 1 0 1 1 1 wait 60
end
```

Little problem: There is a flickering effect during each number drawing. To make the animation fluid, we're going to use the three primitives `animation`, `stopanimation` and `repaint`.

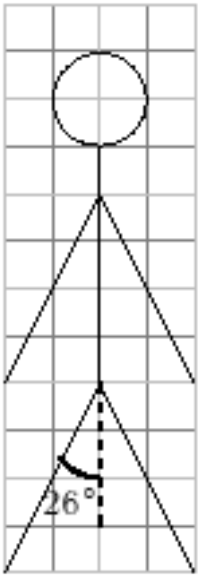
- `animation` enables the mode "animation". The turtle stops drawing on the screen but remembers all changes in cache. To display the image, it's necessary to use the primtive `repaint`.
- `stopanimation` returns to the classic drawing mode.

Here is the new code for this procedure:

```
to countd
# Enables animation mode
animation
clearscreen hideturtle number 0 1 1 1 1 1 1 repaint wait 60
clearscreen hideturtle number 1 1 1 1 1 1 1 repaint wait 60
```

```
clearscreen hideturtle number 0 0 1 0 1 1 0 repaint wait 60
clearscreen hideturtle number 1 1 1 1 0 1 1 repaint wait 60
clearscreen hideturtle number 0 1 1 1 0 1 1 repaint wait 60
clearscreen hideturtle number 0 0 1 1 1 0 1 repaint wait 60
clearscreen hideturtle number 0 1 1 1 1 1 0 repaint wait 60
clearscreen hideturtle number 1 1 0 1 1 1 0 repaint wait 60
clearscreen hideturtle number 0 0 1 0 1 0 0 repaint wait 60
clearscreen hideturtle number 1 1 1 0 1 1 1 repaint wait 60
# back to classic mode
stopanimation
end
```

10.2 Second animation: The growing man



First, we'll define a procedure `man` that draws the above schema. We use a variable to reproduce it at different scales

```
to man :c
left 154 forward 44*:c back 44*:c
left 52 forward 44*:c back 44*:c
left 154 forward 40*:c
left 154 forward 44*:c back :c*44
left 52 forward 44*:c back :c*44
left 154 forward 10*:c
left 90 repeat 180[forward :c/2 right 2] right 90
end
```

Now, we'll create an animation that will make the man grow. To realize this, we'll draw `man 0.1`, then `man 0.2` `man 0.3` ... until `man 5`. Between each man, we'll erase the screen. We obtain two different procedures:

```
to man :c
left 154 forward 44*:c back 44*:c
left 52 forward 44*:c back 44*:c
left 154 forward 40*:c
left 154 forward 44*:c back :c*44
left 52 forward 44*:c back :c*44
```

```
left 154 forward 10*:c
left 90 repeat 180[forward :c/2 right 2] right 90
if :c=5[stop]
cs ht man :c+0.1
end

to go
cs ht
man 0
end
```

Finally to make the animation fluid, we'll use animation mode and the primitive `repaint`.

```
to man :c
left 154 forward 44*:c back 44*:c
left 52 forward 44*:c back 44*:c
left 154 forward 40*:c
left 154 forward 44*:c back :c*44
left 52 forward 44*:c back :c*44
left 154 forward 10*:c
left 90 repeat 180[forward :c/2 right 2] right 90
repaint
if :c=5[stop]
cs ht man :c+0.1
end

to go
cs ht animation
man 0
stopanimation
end
```

Note: Here, the procedure `man` is recursive. In aother way, we could use the primitive `for` to make the variable `:c` from 0.1 to 5. Here is the program:

```
to man :c
cs left 154 forward 44*:c back 44*:c
left 52 forward 44*:c back 44*:c
left 154 forward 40*:c
left 154 forward 44*:c back :c*44
left 52 forward 44*:c back :c*44
left 154 forward 10*:c
left 90 repeat 180[forward :c/2 right 2] right 90
repaint
end

to go
ht animation
for [c 0 5 0.1][man :c]
stopanimation
end
```

Chapter 11

Interact with the user

Level: Newbie

11.1 Question-answer

The program that we're going to create in this chapter will ask the user his first name, his name and his age. At the end, the program will make a synthesis!

```
Your first name is: .......
Your name is: .......
Your age is: .......
You're over 20/less than 20
```

Here are the primitives we're going to use:

- `read`: `read [ ] "a`
  Displays a dialog box whose title is the text from the list (here, "How are you?"). The answer given by the user is stored in a word or in a list (in case of several words) in the variable `:a`.

- `make`: `make "a 30`
  Gives the value 30 to the variable `:a`

- `sentence, se`: `sentence [30 k] "a`
  Adds a value in a list. If this value is a list, removes square brackets.

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

This is the code program:

```
to question
read [How old are you?] "age
read [What's your first name?] "fname
read [What's your name?] "name
print sentence [Your name is: ] :name
print sentence [Your first name is: ] :fname
print sentence [Your age is: ] :age
if or :age>20 :age=20 [print [You're over 20]] [print [You're less than 20]]
end
```

11.2 Programming a little game.

HERE IS THE GAME WE WANT TO PROGRAM:

The program chooses an integer between 0 and 32 and memorizes it. Then, a dialog box opens and asks the user to enter an integer. If this integer is equal to the saved integer, it displays "WIN" in the text zone. Otherwise, the program indicates if the saved integer in greater or lesser than the user's integer and reopens the dialog box. The program will end when the user has found the correct integer.

We need to use the primitive `random`:
For example, `random 20` returns an integer randomly between 0 and 19.

HERE ARE THE RULES TO CREATE THIS GAME:

- The number choosen by the computer will be stored in a variable called `number`.
- The dialog box will be named "Give an integer please"
- The number choosen by the user will be stored in a variable called `try`.
- The main procedure will be named `game`.

SOME POSSIBLE IMPROVEMENTS:

- Displays the number of tries.
- The computer's number will be between 0 and 2000.
- Check that the user enters a valid number. You can use the primitive `number?`.
  Examples: `number? 8` returns true.
  `number? [5 6 7]` returns false.
  `number? "abcde` returns false

Chapter 12

Topic: Two dice sum

Niveau: Medium

By rolling two dice, the sum of the scores on the two dice is an integer between 2 and 12. Here, we're going to see the different probabilities for each integer to appear and represent this with a graphical diagram.

12.1 Simulating rolling one die.

To simulate rolling a die, we're going to use the primitive `random`. Here's how it works:.

`random 6` $\longrightarrow$ returns a randomly choosen integer among 0, 1, 2, 3, 4, 5.
Hence, `(random 6)+1` returns a randomly choosen integer from 1, 2, 3, 4, 5, 6. We need the parenthesis, otherwise, the Logo interpreter should understand `random 7`. To avoid parenthesis, we can write `1+random 6` too.

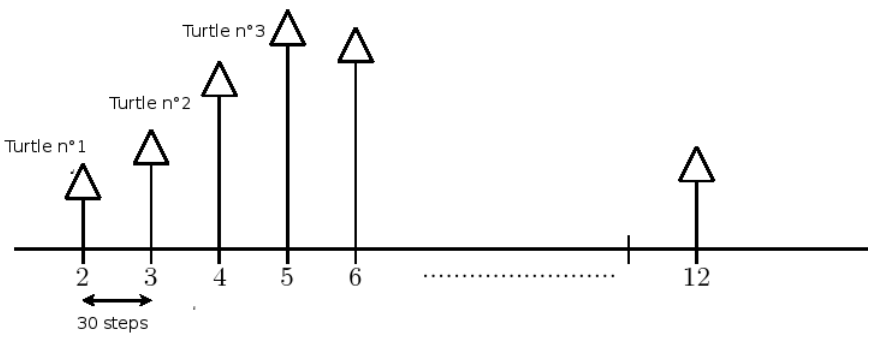
We define the primitive called `die` which simulates rolling one die.

```
to die
   output 1+random 6
 end
```

12.2 The program

We're going to use the multiturtle mode and the primitive `setturtle`. `setturtle` followed by an integer allows us to select the turtle whose identifier is the integer.

A good schema is better than a thousand explanations....



Each turtle whose integer is from 2 o 12 will forward one step, when the corresponding dice sum appears. For example, if the two dice scores is 8, then turtle number 8 will forward one step. Between two turtle, there are 30 steps horizontally.

We set all turtles using coordinates.

- The turtne n ° 2 has coordinates $(-150; 0)$
- The turtne n ° 3 has coordinates $(-120; 0)$
- The turtne n ° 4 has coordinates $(-90; 0)$
- The turtne n ° 5 has coordinates $(-60; 0)$

$$\vdots$$

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

Better than writing 11 times the same command line, we're going to use the primitive `for`. With this primitive, we can give a variable a sequence of values. Here, we want that the variable `:i` to have succesive values 2, 3, 4, ... , 12. We write:
`for [i 2 12] [ list of instructions]`

To set up all the turtles we create the procedure `initialize`

```
to initialize
  cs ht pu
  for [i 2 12] [
      # Set up the turtle
      setturtle :i setpos list -150+(:i-2)*30 0
      # We write turtle number 15 steps under
      pu bk 15 label :i fd 15 pd
  ]
end
```

You must understand the expression `-150+(:i-2)*30`. We're beginning from -150, and for each new turtle we add 30. (Test with the different values for `:i` if you're sceptic)

Finally this is the program:

```
to die
   output 1+random 6
 end

to initialize
  cs ht pu
  for [i 2 12] [
      # Set up the turtle
      setturtle :i setpos list -150+(:i-2)*30 0
      # We write turtle number 15 steps under
      pu bk 15 label :i fd 15 pd
  ]
end
```

```
to go
initialize
# We make 1000 tries
repeat 1000 [
  make "sum die+die
  setturtle :sum fd 1
]
# We display frequencies
for [i 2 12] [
  setturtle :i
  # The Y-coordinates of each turle represents the number of times
  # a dice scores has appeared
  localmake "number last pos
  pu fd 10 lt 90 fd 10 rt 90 pd label :number/1000*100
]
end
```

Here is a more general program. We'll ask the user the number of dice he wants and the number of tries to make.

```
to initialize
  cs ht pu setturtlesmax :max+1
  for sentence list "i :min :max [
      # Set up the turtle
      setturtle :i setpos list -150+(:i-2)*30 0
      # We write turtle number 15 steps under
      pu bk 15 label :i fd 15 pd
  ]
end
to die
localmake "somme 0
repeat :dice [
   localmake "somme :somme+1 +random 6
 ]
output :somme
end

to go
read [Number of dice:] "dice
if not numberp :dice [print [Not a valid number!] stop]
globalmake "min :dice
globalmake "max 6*:dice
read [Number of tries to make: ] "tries
if not numberp :tries [print [labellength nombre rentré n'est pas valide!] stop]
initialize
# We make tries
repeat :tries [
  setturtle die forward 1
]
# Display frequencies
for sentence list "i :min :max [
  setturtle :i
  # Y-Axis coordinates represent the number of times a score has appeared
```

```
  localmake "effectif last position
  # Round to 0.1
  penup forward 10 left 90 forward 10 right 90 pendown label (round :effectif/:tries*1000)/10
]
end
```

Chapter 13

Topic: Probabilistic approximation of π

Level: Advanced

NOTE: Some elementary mathematical knowledge needed for this chapter.

13.1 GCD (Greatest Common Divisor)

The GCD of two integers is the largest positive integer that divides both numbers without remainder.

- For example, the GCD of 42 and 28 is 14. (It's the largest integer that divide both 42 and 28)
- The GCD of 25 and 55 is 5.
- The GCD of 42 and 23 is 1.

The integers a and b are said to be **coprime** or **relatively prime** if they have no common factor other than 1 or, equivalently, if their greatest common divisor is 1. With the precedent example, 42 and 23 are relatively prime.

13.2 Euclidean algorithm

Calculate the GCD of two inetgers efficiently can be done with the Euclidean algorithm. (Here, we don't show that this algorithm is valid)

Description of the algorithm:
Given two positive integers a and b, we check first if b is equal to 0. If it's the case, then the GCD is equal to a. Otherwise, we calculate r, the remainder after dividing a by b. Then, we replace a by b, and b by r, and we restart this method.
For example, let's calculate the GCD of 2160 and 888 with the Euclidean algorithm:

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

Hence, the GCD of 2160 and 888 is 24. There's no largest integer that divide both numbers. (In fact $2160 = 24 \times 90$ and $888 = 24 \times 37$)
GCD is the last remainder not equal to 0.

13.3 Calculate a GCD in Logo programming

A little recursive procedure will calculate the GCD of two integers `:a` and `:b`

```
to gcd :a :b
if (modulo :a :b)=0 [output :b][output gcd :b modulo :a :b]
end

print gcd 2160 888
24
```

Note: It's important to put parenthesis around `modulo :a :b`. Otherwise, the interpreter would try to evaluate `:b = 0`. If you don't want to use parenthesis, write: `if 0=remainder :a :b`

13.4 Calculating π-approximation

In fact, a famous result in numbers theory says that the probability for two randomly chosen integers to be coprime is $\frac{6}{\pi^2} \approx 0,6079$. To exhibit this result, we're going to:

- Choose randomly two integers between 0 and 1 000 000.
- Calculate their GCD.
- If the GCD value is 1, increment the counter variable.
- Repeat this experience 1000 times.
- The frequence for the couple of coprime integers can be calculated dividing the variable counter by 1000 (tries number).

```
to test
# We set the variable counter to 0
globalmake "counter 0
repeat 1000 [
  if (gcd random 1000000 random 1000000)=1 [globalmake "counter :counter+1]
]
print [frequence:]
print :counter/1000
end
```

In a similar way as the precevious note, notice the parenthesis around `gcd random 1000000 random 1000000`. Otherwise, the interpreter will try to evaluate $1\ 000\ 000 = 1$. You can write in other way: `if 1=gcd random 1000000 random 1000000`

We execute the program `test`.

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

We obtain some values close to the theorical probability: 0,6097. This frequency is an approximation of $\frac{6}{\pi^2}$.
Thus, if I note f the frequency, we have: $f \approx \frac{6}{\pi^2}$
Hence, $\pi^2 \approx \frac{6}{f}$ and $\pi \approx \sqrt{\frac{6}{f}}$.
I append to my program a line that gives this π approximation in procedure `test`:

```
to test
# We set the variable counter to 0
globalmake "counter 0
repeat 1000 [
  if (gcd random 1000000 random 1000000)=1 [globalmake "counter :counter+1]
]
# We calculate te frequency
make "f :counter/1000
# we dispaly the pi approximation
print sentence  [ pi approximation:] sqrt (6/:f)
end

test
pi approximation: 3.1033560252704917
test
pi approximation: 3.1835726998350666
test
pi approximation: 3.146583877637763
```

Now, we modify the program because I want to set the number of tries. I want to try with 10 000 and perhaps more tries.

```
to test :tries
# We set the variable counter to 0
globalmake "counter 0
repeat :tries [
  if (gcd random 1000000 random 1000000)=1 [globalmake "counter :counter+1]
]
# We calculate te frequency
make "f :counter/:tries
# we dispaly the pi approximation
print sentence  [ pi approximation:] sqrt (6/:f)
end

test 10000
pi approximation: 3.1426968052735447
test 10000
pi approximation: 3.1478827771265787
test 10000
pi approximation: 3.146583877637763
test 10000
```

Quite interesting, isn't it?

13.5 More complex: π generating π.....

What is a random integer? Is an integer choosen randomly between 1 and 1000000 really representative for all integers choosen randomly? We can see that our experience is only an approximation of an ideal model. Here, we're going to modify the method for generating random integers... We won't use the primitive `random`, we're going to generate random integers with the π digits sequence.
π digits have always interested mathematicians:

- The numbers 0 to 9, do some appear more often than others?
- Is there some sequence of integers that appear frequently?

In reality, it **seems** that the π digit sequence is a really randomly sequence. (Result not demonstrated yet). It's not possible to predict the following digit after the others, there's no period.

Here is the method we're going to use to generate integers randomly choosen:

- First, we need the first digit of π (For example, one billion)
  1. First way: some programs calculate the π digits. For example, PiFast in Windows environment and SchnellPi for Linux.
  2. Second way: you can download this file from XLogo website:

     `http://downloads.tuxfamily.org/xlogo/common/millionpi.txt`
- To generate the integers, we're going to read the digits sequence in packet of 7 digits:
  3.1415926 53589793 23846264 33832795028841971 6939 etc
  First number Second number Third number
  I remove the point ". " in 3.14 ....that will cause problem when we're going to extract the digits

Let's create now a new procedure called `randompi` and let's modify the procedure `test`

```
to gcd :a :b
if (modulo :a :b)=0 [output :b][output gcd :b modulo :a :b]
end

to test :tries
# We open a flow whose identifier is 1 towards the file  millionpi.txt
# Here we suppose that millionpi.txt is in the current directory
# Otherwise, fix it with changedirectory
openflow 1 "millionpi.txt
# Set the variable line to the first line of the file millionpi.text
globalmake "line first readlineflow 1
# Set the variable counter to  0
globalmake "counter 0
repeat :tries [
  if 1=gcd randompi 7 randompi 7 [globalmake "counter :counter+1]
]
# Calculate frequency
globalmake "f :counter/:tries
# Display th pi approximation
print sentence [ pi approximation:] squareroot (6/:f)
closeflow 1
end

to randompi :n
localmake "number "
```

```
repeat :n [
# If there's no char yet on the line
if 0=count :line [globalmake "line first readlineflow 1]
# Set the variable char to the first character of the line
globalmake "char first :line
# Then remove  first character from the line.
globalmake "line butfirst :line
globalmake "number word :number :char
]
output :number
end

test 10
approximation de pi: 3.4641016151377544
test 100
approximation de pi: 3.1108550841912757
test 1000
approximation de pi: 3.081180112566604
test 10000
approximation de pi: 3.1403714651066386
test 70000
approximation de pi: 3.1361767950325627
```

We find a correct approximation of π with its own digits!

It's still possible to imrove the program by indicating the time for the computation. We add on the first line of the procedure `test`:

```
globalmake "begin pasttime
```

Then we append before `closeflow`:

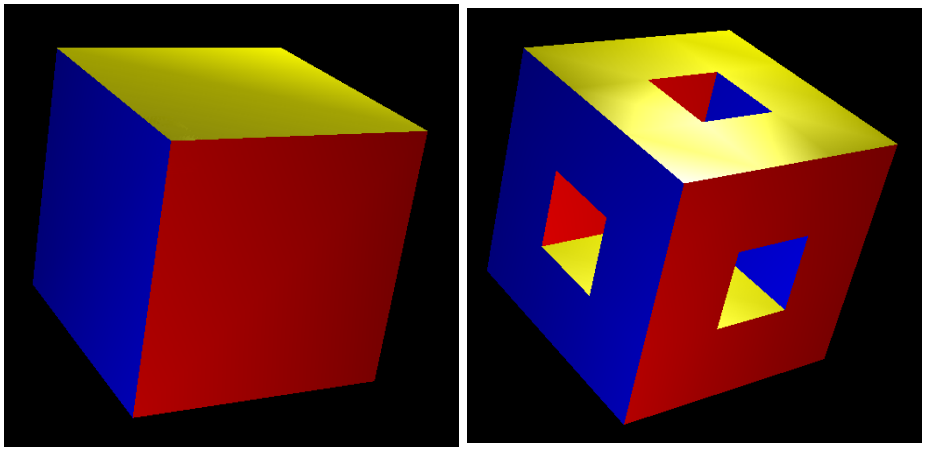
```
print sentence [pasttime mis:  ]  pasttime - :begin
```

Chapter 14

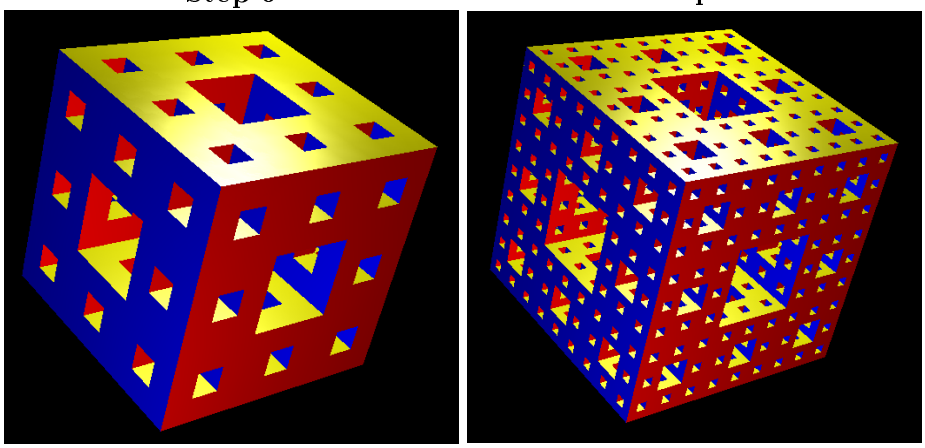
Topic: Menger's sponge

Level: Advanced

In this chapter, we're going to build a fractal solid called Menger's sponge. Here are the first steps to create this solid:



Step 0 **Step 1**



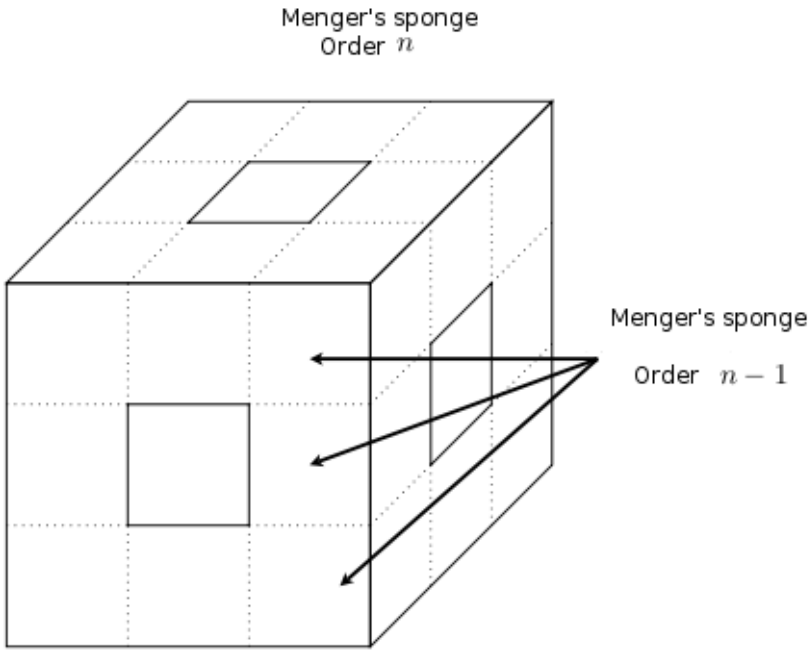
Step 2 **Step 3**

This chapter contains two sections:

- First, we'll show how to create this solid using recursion.
- Finally, we'll try to generate a Menger sponge of order 4.

14.1 Using recursion

Let's consider a Menger sponge of order n which side length is L.



On the schema, we can see that this sponge contains 20 Menger sponges of order $n-1$ and with a side length $\frac{L}{3}$. The recursive structure of the sponge is well shown.

The program:

```
to cube :l
if :counter=10000 [view3d]
# faces colors
localmake "colors [yellow magenta cyan blue]
# lateral faces
repeat 4 [setpencolor run item repcount :colors square :l right 90 forward  :l left 90 rightroll 9
# bottom
setpencolor red downpitch 90 square :l uppitch 90
forward :l downpitch 90 setpencolor green square :l uppitch 90 back :l
end

to square :c
globalmake "counter :counter+1
polystart
repeat 4 [forward :c right 90]
polyend
end

# Menger's sponge
# p: recursion order
# l: side length of the cube
to menger :l :p
if :p=0 [cube :l] [
  localmake "p :p-1
  localmake "l :l/3
  #front face
```

```
  repeat 3 [menger :l :p forward :l] back 3*:l
  right 90 forward :l left 90
  menger :l :p forward 2*:l menger :l :p back 2*:l
  right 90 forward :l left 90
  repeat 3 [menger :l :p forward :l] back 3*:l
  #right face
 downpitch 90 forward :l uppitch 90
  menger :l :p  forward 2*:l menger :l :p back 2*:l
  downpitch 90 forward :l uppitch 90
  repeat 3 [menger :l :p forward :l] back 3*:l
  left 90 forward :l right 90
  menger :l :p  forward 2*:l menger :l :p back 2*:l
  left 90 forward :l right 90
  repeat 3 [menger :l :p forward :l] back 3*:l
  downpitch 90 back :l uppitch 90
  menger :l :p  forward 2*:l menger :l :p back 2*:l
   downpitch 90 back :l uppitch 90
]
end

to sponge :p
clearscreen hideturtle globalmake "counter 0 3d setscreencolor 0 menger 800 :p
write [nombre penpaint polygone: ] print :counter
view3d
end
```

This program has four procedures:

- `square :c`
  This procedure draws a square which has side length `:c`. This polygon is stored in the 3D Viewer. The variable `counter` counts the number of drawn polygons.

- `cube :l`
  This procedure draws a cube which has side length `:l`. Of course, it uses procedure `square`

- `menger :l :p`
  This is the most important procedure of the program, it draws a Menger motif of order p and with a side length equal to l. This motif is created using recursion as we have seen before on the schema.

- `sponge :p`
  This procedure creates a Menger sponge, order p with a side length equal to 800 and draws it in the Viewer 3D.

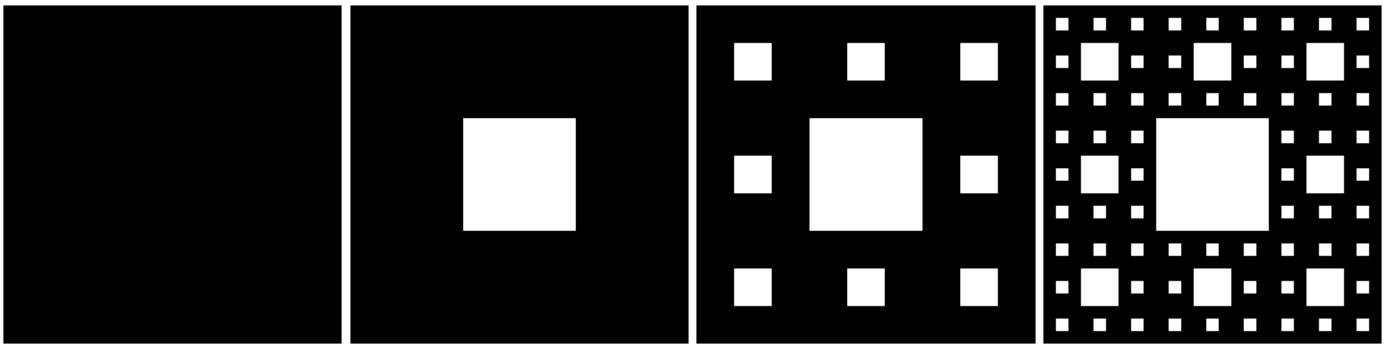
14.2 Second approach: Drawing a Menger sponge, order 4

The main advantage of the previous program is to exploit the recursive structure of the solid. This method is quite similar to the one we used to draw the Van Koch snowflake on p.40. The main advantage of using recursion is a quite natural short program code. The disadvantage of the recursive approach is the number of created polygons: for example, a sponge of order 3 needs 48 000 polygons. XLogo requires in this case an internal memory set to 256 Mb in the Preferences panel to prevent from memory overflow.

If we want to draw a Menger sponge, order 4, we have to rethink the program and to forget recursion. We're going to create in this section a program that will draw the Menger solid of order 0,1,2,3 or 4.

14.2.1 Sierpinski carpet

Menger's sponge is the generalization in 3 dimensions of a plane figure called "the Sierpinski carpet". Here are the first steps to generate this figure:



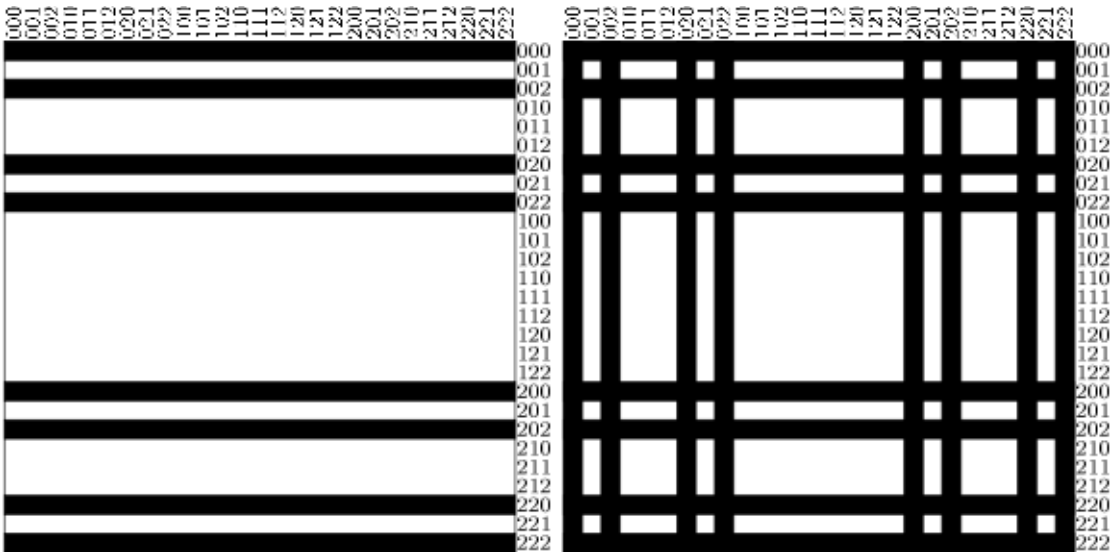
Step 0 **Step 1** **Step 2** **Step 3**

Each face of a Menger sponge of order p is a Sierpinski carpet of order p.

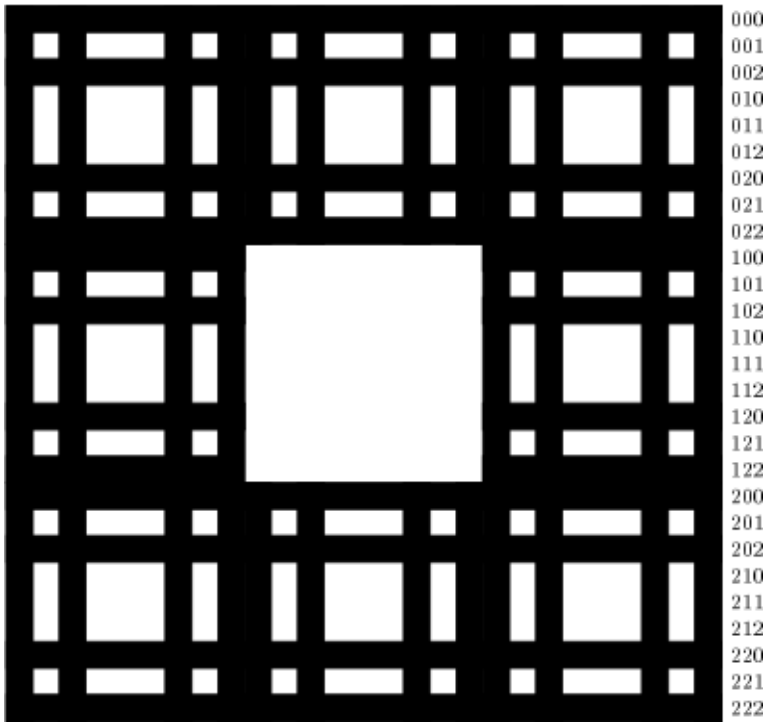
14.2.2 Drawing a Sierpinski carpet of order p

The objective is to set minimal the number of polygon to draw a Sierpinski carpet. The following example explains how to draw a Sierpinski carpet of order 3. Here, the first square has $3^3 = 27$ lines and 27 columns. We write in 3-basis each line number and each column number.

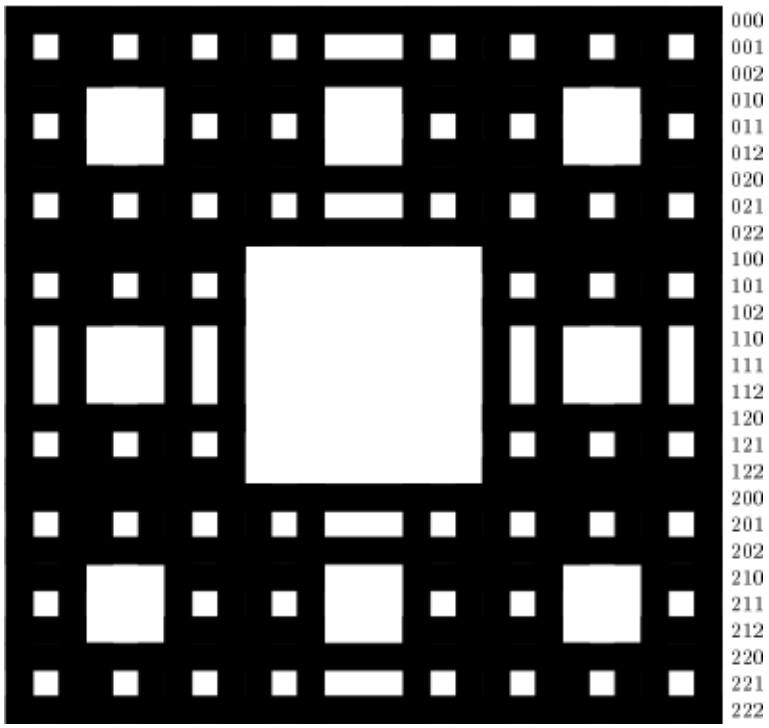
- **First step:** For each line whose number doesn't contain any 1, we draw a 27 units line. Using symmetry, we repeat the same operation on columns.



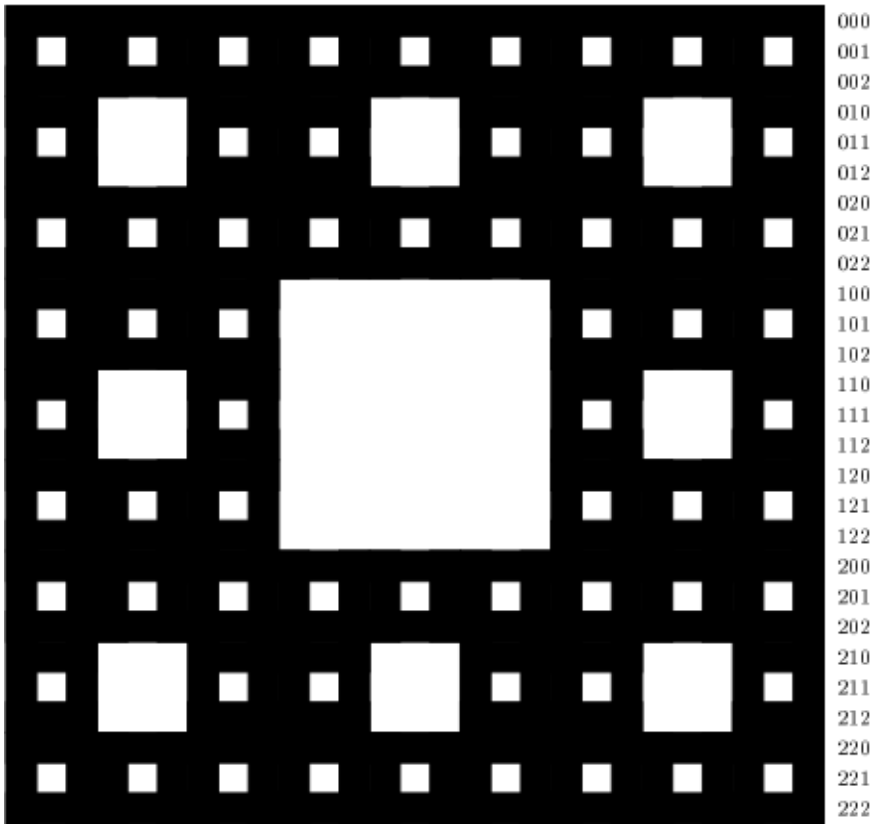
- **Second step:** Now, we're looking at lines whose numbers have a single 1 in first place. We draw rectangles of 9 units length by alternation.



- **Third step:** Now we're looking at lines whose number contains a single 1 in second place. We draw rectangles following the schema [3 3 6 3 6 3 3]. (It means 3 units pen down, 3 units pen up, 6 units pen down etc...). Using symmetry, we repeat this operation on columns.



- **Final step:** Now we're looking at lines whose number contains a double 1 in the first two positions. We draw rectangles alternating following the schema [3 3 3 9 3 3 3]. We repeat this operation on columns.



Now, we have built a Sierpnski carpet of order 3. To draw such a carpet, we need: $16 + 16 + 32 + 16 = 80$ polygons.

14.2.3 All Different possible schemas for columns

To recapitulate, here are the different column schemas according to the line numbers. (The symbol * represents 0 or 2)

Number of line	Schema to apply
***	27
1**	9 9 9
1	3 3 6 3 6 3 3
11*	3 3 3 9 3 3 3

In the same way, to build a carpet of order 4, we need a square with $3^4 = 81$ units. The line and column numbers will have 4 numbers in their writing in 3-basis. For each line number, here is the schema to apply (the symbol * represents 0 or 2):

Line number	Schema to apply
****	81
1***	27 27 27
*1**	9 9 18 9 18 9 9
**1*	3 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 3
11	3 3 3 9 3 3 6 3 3 9 3 3 6 3 3 9 3 3 3
1*1*	3 3 6 3 6 3 3 27 3 3 6 3 6 3 3
11**	9 9 9 27 9 9 9
111*	3 3 3 9 3 3 3 27 3 3 3 9 3 3 3

496 polygons are necessary to draw the a Sierpinski carpet of order 4.

Finally, here are the schema to apply for solid of order 2:

Line numbers	Schema to apply
**	9
1*	3 3 3

14.2.4 The program

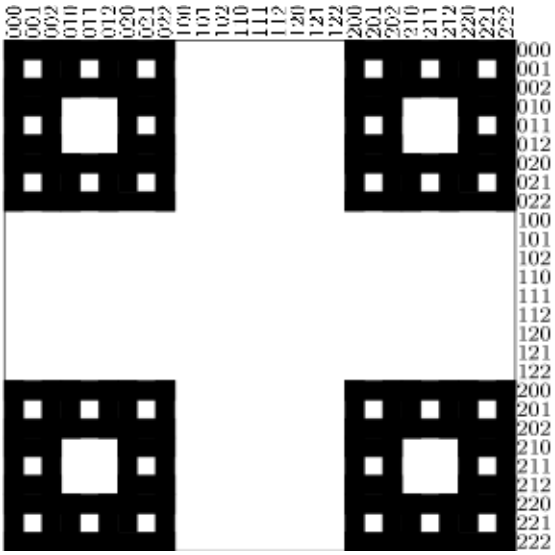
```
 # Draws a Sierpinski carpet of order :p and size :size
to carpet :size :p
globalmake "unit :size/(power 3 :p)
if :p=0 [ rec :size :size stop]
if :p=1 [repeat 4 [rec :size :unit forward :size right 90 ] stop]
for (list "x 1 power 3 :p) [
  localmake "cantorx cantor :x :p []
# We didn't draw elements with a 1 in last position
if  not (1=last :cantorx)  [
  localmake "nom evalue butlast :cantorx "
  drawcolumn :x getproperty "map :nom
  ]
]
end

# output the writing in 3-basis of number x
# p order of the carpet (3^p units)
# :list empty list

to cantor :x :p :list
if :p=0 [output :list]
localmake "a power 3 :p-1
if :x<= :a [
  output cantor  :x :p-1  sentence :list 0]
  [ if :x<=2*:a [output cantor  :x-:a :p-1  sentence :list 1]
  output cantor :x-2*:a :p-1 sentence :list 0]
end

# Draw the column number x respecting the schema in list :list
to drawcolumn :x :list
  penup  right 90 forward (:x-1)*:unit left 90  pendown des :list
  penup left 90 forward (:x-1)*:unit right 90 forward :x*:unit right 90 pendown des :list
penup left 90 back :x*:unit pendown
end

# Draws a rectangle with choosen dimensions
# It is stored in 3D viewer
to rec :lo :la
globalmake "compteur :compteur+1
polystart
repeat 2 [forward :lo right 90 forward :la right 90]
polyend
end

# Inits the different possible columns for carpet order 0 to 4
to initmap
putproperty "map 111 [3 3 3 9 3 3 3 27 3 3 3 9 3 3 3]
putproperty "map 110 [9 9 9 27 9 9 9]
```

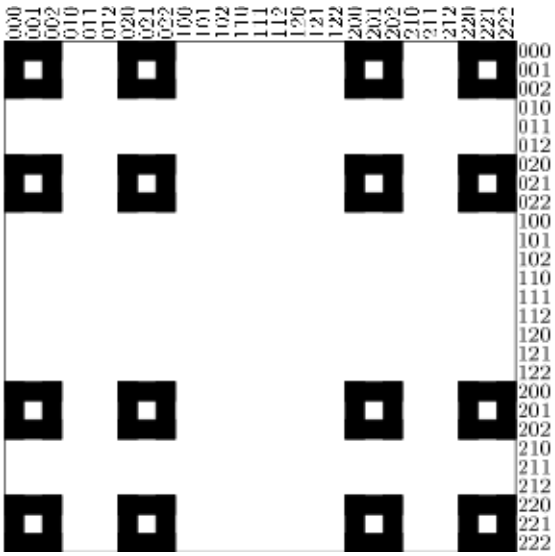
```
putproperty "map 101 [3 3 6 3 6 3 3 27 3 3 6 3 6 3 3]
putproperty "map 011 [3 3 3 9 3 3 6 3 3 9 3 3 6 3 3 9 3 3 3]
putproperty "map 000 [81]
putproperty "map 100 [27 27 27]
putproperty "map 010 [9 9 18 9 18 9 9]
putproperty "map 001 [3 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 3]
putproperty "map 01 [3 3 6 3 6 3 3]
putproperty "map 00 [27]
putproperty "map 10 [9 9 9]
putproperty "map 11 [3 3 3 9 3 3 3]
putproperty "map 1 [3 3 3]
putproperty "map 0 [9]
end

# if the 3-basis writing is  [1 0 1] --> output 101
to evalue :list :mot
  if emptyp :list [output :mot]
  [
  localmake "mot word :mot first :list
  output evalue butfirst :list :mot
]
end
# Draws the block of rectangles alternanting
to des :list
localmake "somme 0
for (list "i 1 count :list) [
   localmake "element item :i :list
    localmake "somme :element+:somme
  if even? :i [penup forward :element*:unit pendown ] [rec :element*:unit :unit forward :element*:
]
penup back  :somme * :unit pendown
end

# Is this number even?
to pair? :i
output 0=reste :i 2
end

# Draws the carpet order :p
to tapis :p
clearscreen 3d hideturtle initmap
globalmake "compteur 0
carpet 810 :p
write "nombre\ de\ polygones:\  print :compteur
view3d
end

# Is this number even?
to even? :i
output 0=modulo :i 2
end
```

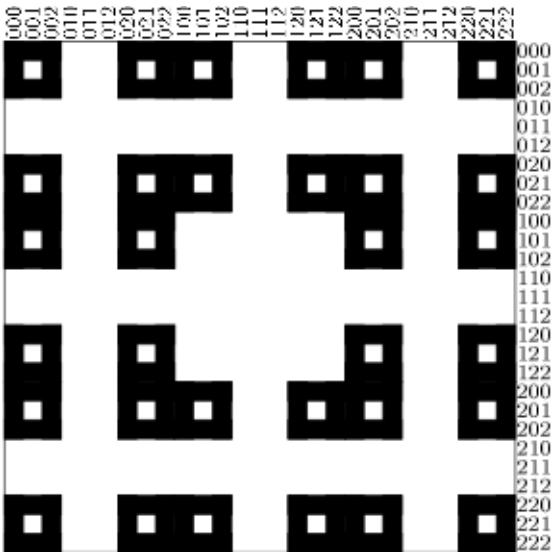
`tapis 3` draws a Sierpinski carpet of order 3 with a side length equal to 810. Here we are! Now we can come back to the Menger's sponge!

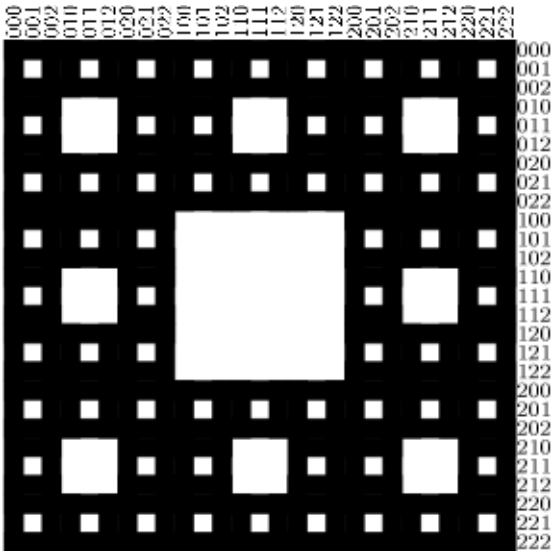
14.2.5 Menger's sponge order 4

The Menger sponge has a lot of symmetries. To build the sponge, we're going to draw the different sections along the plane (xOy) and then repeat those figures along the planes (yOz) and (xOz). To explain what happens, let's have a look at the sponge of order 2:
When we cut with a vertical plane, we can obtain four different motifs:









To draw a sponge of order 3, we're going to browse the number from 1 to 27, it means from 001 to 222 in 3 basis. For each number, we'll apply the valid section and we'll report this figure along (Ox), (Oy) and (Oz).

The code

With this program, we can draw Menger's sponge of order 0,1,2,3 and 4.

```
 # Draws a Sierpinski carpet of order :p and size :size
to carpet :size :p
globalmake "unit :size/(power 3 :p)
if :p=0 [ rec :size :size stop]
if :p=1 [repeat 4 [rec :size :unit forward :size right 90 ] stop]
for (list "x 1 power 3 :p) [
  localmake "cantorx cantor :x :p []
# We didn't draw elements with a 1 in last position
if  not (1=last :cantorx)  [
  localmake "nom evalue butlast :cantorx "
  drawcolumn :x getproperty "map :nom
  ]
]
end
```

```
# output the writing in 3-basis of number x
# p order of the carpet (3^p units)
# :list empty list

to cantor :x :p :list
if :p=0 [output :list]
localmake "a power 3 :p-1
if :x<= :a [
  output cantor  :x :p-1  sentence :list 0]
  [ if :x<=2*:a [output cantor  :x-:a :p-1  sentence :list 1]
  output cantor :x-2*:a :p-1 sentence :list 2]
end

# Draw the column number x respecting the schema in list :list
to drawcolumn :x :list
  penup  right 90 forward (:x-1)*:unit left 90  pendown des :list
  penup left 90 forward (:x-1)*:unit right 90 forward :x*:unit right 90 pendown des :list
penup left 90 back :x*:unit pendown
end

# Draws a rectange with choosen dimensions
# It is stored in 3D viewer
to rec :lo :la
globalmake "counter :counter+1
polystart
repeat 2 [forward :lo right 90 forward :la right 90]
polyend
end

# Inits the different possible columns for carpet order 0 to 4
to initmap
putproperty "map 111 [3 3 3 9 3 3 3 27 3 3 3 9 3 3 3]
putproperty "map 110 [9 9 9 27 9 9 9]
putproperty "map 101 [3 3 6 3 6 3 3 27 3 3 6 3 6 3 3]
putproperty "map 011 [3 3 3 9 3 3 6 3 3 9 3 3 6 3 3 9 3 3 3]
putproperty "map 000 [81]
putproperty "map 100 [27 27 27]
putproperty "map 010 [9 9 18 9 18 9 9]
putproperty "map 001 [3 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 3]
putproperty "map 01 [3 3 6 3 6 3 3]
putproperty "map 00 [27]
putproperty "map 10 [9 9 9]
putproperty "map 11 [3 3 3 9 3 3 3]
putproperty "map 1 [3 3 3]
putproperty "map 0 [9]
end

# if the 3-basis writing is  [1 0 1] --> output 101
# if the 3-basis writing is [1 0 2] --> output 100
#  Element from the list are translated into a word.
# 2 are replaced by 0
```

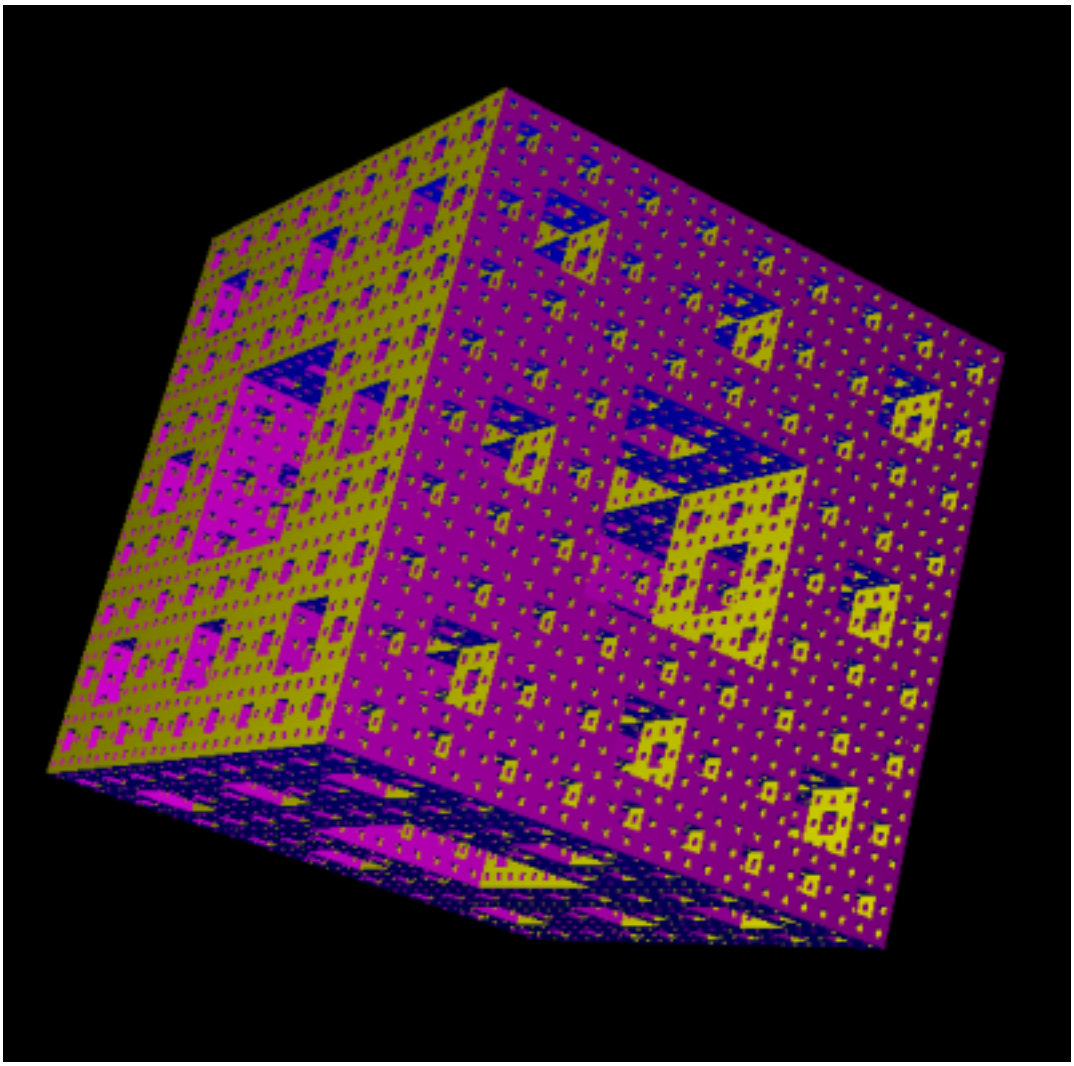
```
to evalue :list :mot
  if emptyp :list [output :mot]
  [
  localmake "first first :list
  if :first=2 [localmake "first 0]
 localmake "mot word :mot :first
  output evalue butfirst :list :mot
]
end
# Draws the block of rectangular alternanting
to des :list
localmake "somme 0
for (list "i 1 count :list) [
   localmake "element item :i :list
    localmake "somme :element+:somme
  if even? :i [penup forward :element*:unit pendown ]
      [rec :element*:unit :unit forward :element*:unit]
]
penup back  :somme * :unit pendown
end

# Draws the carpet order :p
to tapis :p
clearscreen 3d hideturtle initmap
globalmake "compteur 0
carpet 810 :p
write "nombre\ de\ polygones:\  print :compteur
view3d
end

# Is this number even?
to even? :i
output 0=modulo :i 2
end


# Remove the last 1 from :list
to deletelastone :list
for (list "i count :list 1 minus 1) [
  localmake "element item :i :list
  if :element=1 [localmake "list replace :list :i 0 stop] [if :element=2 [stop]]
]
output :list
end

# Draws the Serpinski carpet
# along axis (ox), (oy) and (oz)
to draw3carpet :size :order :z
penup home
uppitch 90 forward (:z-1)*:unite downpitch 90 pendown
setpencolor blue run :order :size
penup home
leftroll 90 forward (:z-1)*:unite downpitch 90  pendown
```

```
setpencolor yellow run :order :size
penup home
uppitch 90 forward :size right 90 forward (:z-1)*:unite downpitch 90 pendown
setpencolor magenta run :order :size
end

# Menger's sponge order :p and size :size

to menger :size :p
globalmake "unite :size/(power 3 :p)
for (list "z 1 power 3 :p) [
  localmake "cantorz cantor :z :p []
  localmake "last last :cantorz
  localmake "cantorz butlast :cantorz
  if :last=0 [localmake "order evalue deletelastone :cantorz "]
           [localmake "order evalue :cantorz "]
  localmake "order word "coupe :order
  draw3carpet :size :order :z
  penup uppitch 90 forward :unit downpitch 90 pendown
]
draw3carpet :size :order (power 3 :p)+1
end


# Main procedure
# Draws a sponge order :p with side length 405
to sponge :p
clearscreen setsc 0 3d hideturtle
localmake "time pasttime
initmap
globalmake "counter 0
if :p=0 [cube 405] [menger 405 :p]
# Displays the time to build the sponge
write "Polygons\ number:\  print :counter
write "Time:\  print pasttime -:time
view3d
end

# Different sections for menger order 2
to coupe1 :size
repeat 4 [carpet :size/3 1 penup forward :size right 90 pendown]
end

to coupe0 :size
carpet :size 2
end

# Different sections for Menger order 3

to coupe10 :size
repeat 4 [carpet :size/3 2 penup forward :size right 90 pendown]
end
```

```
to coupe01 :size
repeat 4 [repeat 2 [coupe1 :size/3 penup forward :size/3 pendown] forward :size/3 right 90]
end

to coupe11 :size
repeat 4 [coupe1 :size/3 penup forward :size right 90 pendown]
end


to coupe00 :size
carpet :size 3
end

# Different sections for Menger order 4
to coupe000 :size
carpet :size 4
end

to coupe100 :size
repeat 4 [carpet :size/3 3 penup forward :size right 90 pendown]
end

to coupe010 :size
repeat 4 [repeat 2 [coupe10 :size/3 penup forward :size/3 pendown] forward :size/3 right 90]
end

to coupe001 :size
repeat 4 [repeat 2 [coupe01 :size/3 penup forward :size/3 pendown] forward :size/3 right 90]
end

to coupe110 :size
repeat 4 [coupe10 :size/3 penup forward :size pendown right 90 ]
end

to coupe111 :size
repeat 4 [coupe11 :size/3 penup forward :size right 90 pendown]
end

to coupe101 :size
repeat 4 [coupe01 :size/3 penup forward :size right 90 pendown]
end

to coupe011 :size
repeat 4 [repeat 2 [coupe11 :size/3 penup forward :size/3 pendown] forward :size/3 right 90]
end

to coupe :size
carpet :size 1
end

to cube :size
repeat 2 [
setpencolor blue rec :size :size penup forward :size downpitch 90 pendown
```

```
setpencolor yellow rec :size :size penup forward :size downpitch 90  pendown
]
setpencolor magenta
penup leftroll 90 left 90 forward :size right 90 pendown rec :size :size
penup right 90 forward :size left 90 rightroll 90 right 90 forward :size left 90 rightroll 90 pend
leftroll 90 left 90 forward :size right 90
end
```

Then, we set memory allocated to XLogo to 640 Mb: `sponge 4`



Chapter 15

Topic: Lindenmayer system

Level: Advanced

In this part, I take reference from:

- the english Wikipedia page about L-systems: `http://en.wikipedia.org/wiki/L-System`.
- the book "The Algorithmic Beauty of Plants" written by Przemyslaw Prusinkiewicz and Aristid Lindenmayer.

This section will deal with the Lindemayer systems or L-system introduced and developed in 1968 by the Hungarian theoretical biologist Lindenmayer. A L-System is a set of rules and symbols used to to model the growth processes of plant development, but also able to model the morphology of a variety of organisms. The main concept in L-Systems is "rewriting rules". This technic is used to replace some initial condition using some rules to do the replacement.

15.1 Formal definition

A L-System is a formal grammar with :

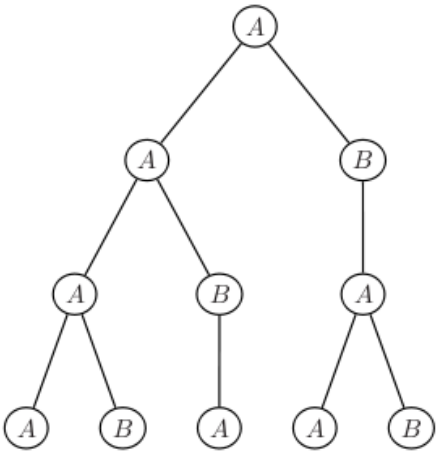
1. An alphabet V : The set of the variables of the L-System. $V*$ stands for the set of the "words" we could generate with any symbols taken from alphabet V, and $V+$ the set of "words" with at least one symbol.
2. A set of constant values S. Some of this symbol are common to all L-System. (in particular with the turtle!).
3. A start awiom ω taken from $V+$, it is the initial state.
4. A set of prodution rules P of the V symbols.

Such a L-System is defined as a tuple $\{V, S, \omega, P\}$.

Let's consider the following L-system:

- Alphabet : $V = \{A, B\}$
- Constants : $S = \{\emptyset\}$
- Start Axiom: $\omega = A$
- Rules : $\boxed{\begin{array}{l} A \rightarrow AB \\ B \rightarrow A \end{array}}$

The two production rules are rewriting rules. On each step, the symbol A is replaced by the séequence AB, and the symbol B is replaced by A. Here are the first iterations of this Lindemayer system:



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

Ok, ok but concretely? Let's read next section!

15.2 Turtle interpretation

This first example helps to understand what is a Lindenmayer system but we can't see for now the rapport with our turtle and Logo..

Here it comes interesting: every word we built before has no meaning. We're going to define for each letter of the sequence an action to execute with the turtle, and draw with this method 2D or 3D drawing.

15.2.1 Usual Symbols

- F : Forward one unit step ($\in V$)
- $+$: Turns left angle α ($\in S$).
- $-$: Turns right angle α ($\in S$).
- & : Go down angle α ($\in S$).
- ^: Go up angle α ($\in S$).
- \: Roll left angle α ($\in S$).
- /: Roll right angle α ($\in S$).
- |: Half-tour. In XLogo: `rt 180`

For example, if $\alpha = 90$ with a unit step of 10 turtle steps, we have:

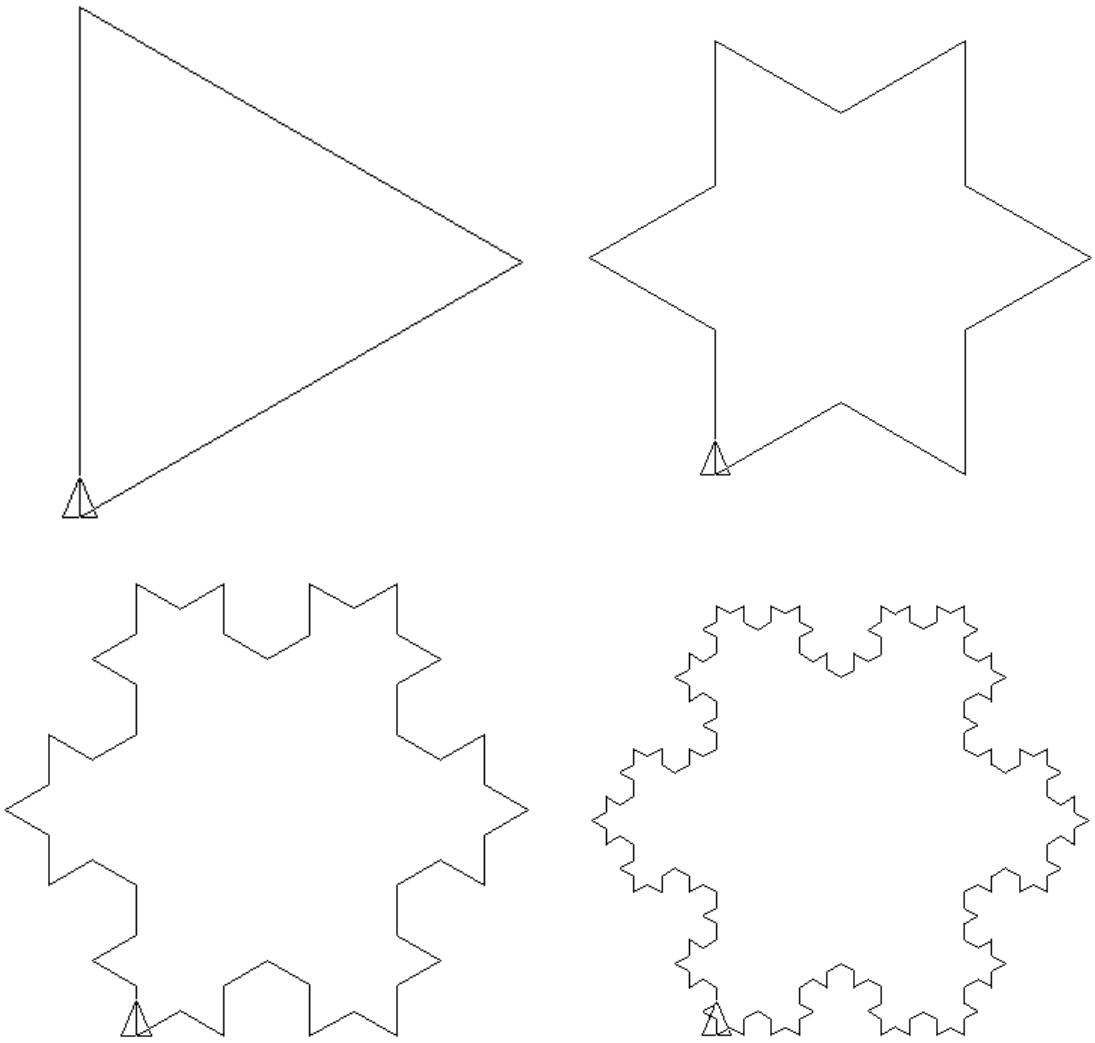
Symbol	F	$+$	$-$	&	^	\	/	|
XLogo Command	`fd 10`	`lt 90`	`rt 90`	`down 90`	`up 90`	`lr 90`	`rr 90`	`rt 180`

15.2.2 Van Snowflake

Let's consider the L-system:

- Initial state: $F--F--F--$
- Production rules: $F \rightarrow F+F--F+F$
- Angle $\alpha = 60^\circ$, Unit step is divided by 3 between each iteration.

First iterations:



XLogoProgram:

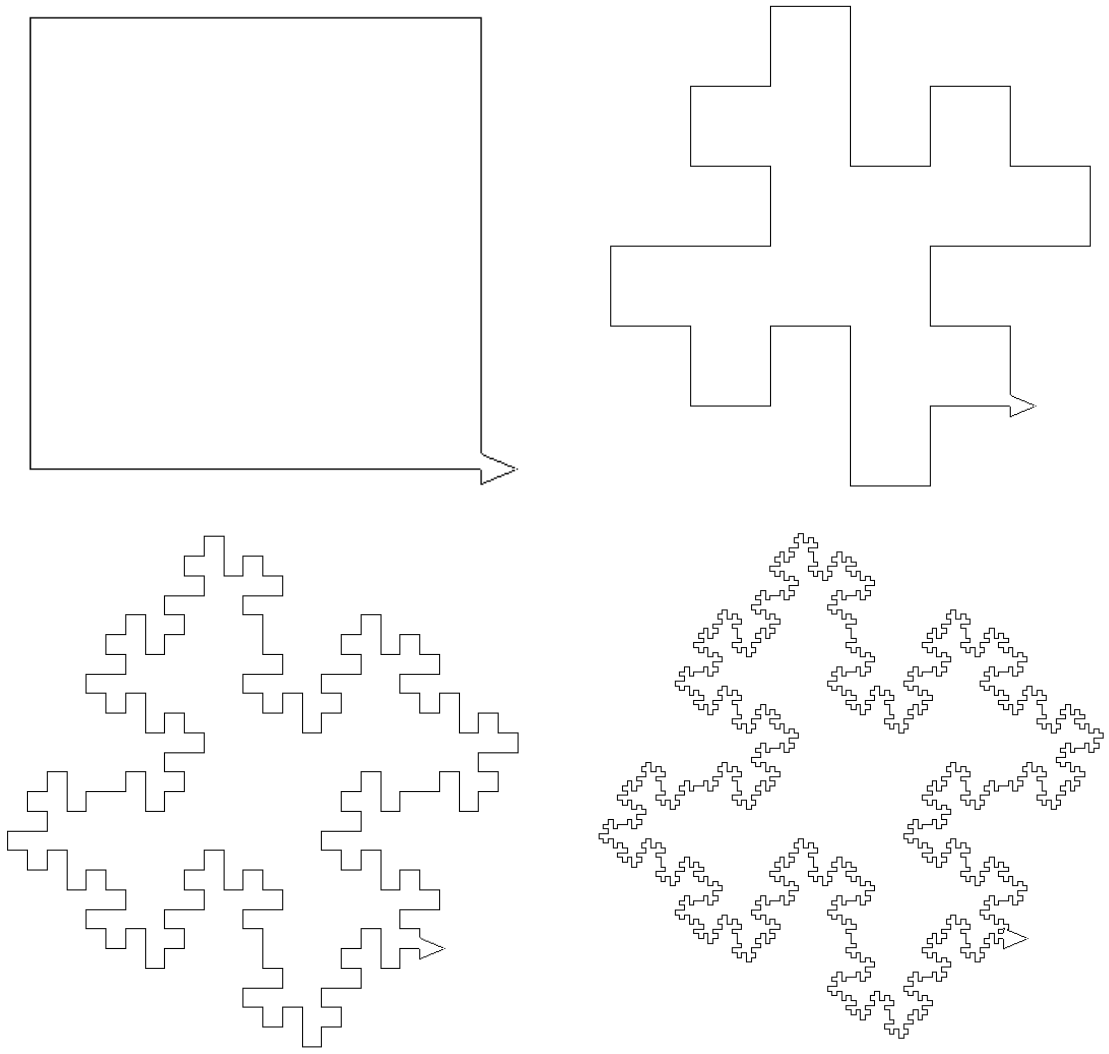
```
to snowflake :p
globalmake "unit 300/power 3 :p-1
repeat 3 [f :p-1 right 120]
end

to f :p
if :p=0 [forward :unit stop]
f :p-1 left 60 f :p-1 right 120 f :p-1 left 60
f :p-1
end
```

15.2.3 Quadratic Van Koch curve

Given this new L-system:

- Initial state: $F - F - F - F$
- Production rules: $F \rightarrow F - F + F + FF - F - F + F$

Here are the first representations using $\alpha = 90$, we adjust the unit step for the figure has a constant size.



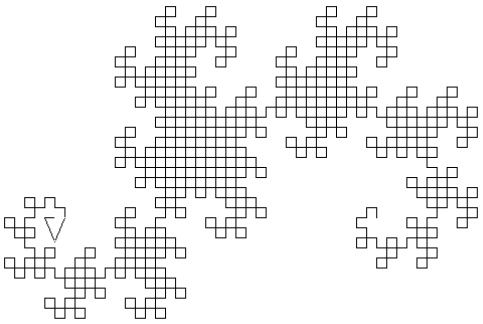
Then it is very easy to create a Logo program to generate these drawings:

```
# p represent the order
to koch :p
# Between two iteration, the unit step is divided by  4
# The final figure will have a maximal size of 600x600
globalmake "unit 300/power 4 :p-1

repeat 3 [f :p-1 left 90] f :p-1
end

# Rewriting rules
to f :p
if :p=0 [forward :unit stop]
f :p-1 left 90 f :p-1 right 90 f :p-1 right 90
f :p-1 f :p-1 left 90 f :p-1 left 90 f :p-1 right 90 f :p-1
end
```

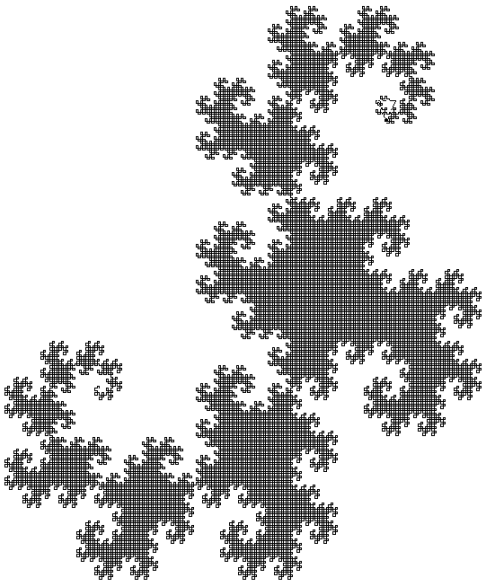
15.2.4 Dragon curve

- Initial state: F
- Production rules: $\boxed{\begin{array}{l} A \rightarrow A + B + \\ B \rightarrow -A - B \end{array}}$

```
to a :p
if :p=0 [forward :unit stop]
a :p-1 left 90 b :p-1 left 90
end

to b :p
if :p=0 [forward :unit stop]
right 90 a :p-1 right 90 b :p-1

end

to dragon :p
globalmake "unit 300/8/ :p
a :p
end
```



dragon 10



dragon 15

15.2.5 Hilbert 3D curve

The following example will generate a 3D Hilbert curve. This curve is singular because it fills perfectlty a cube when we increase iterations.

Here is the L-system to consider:

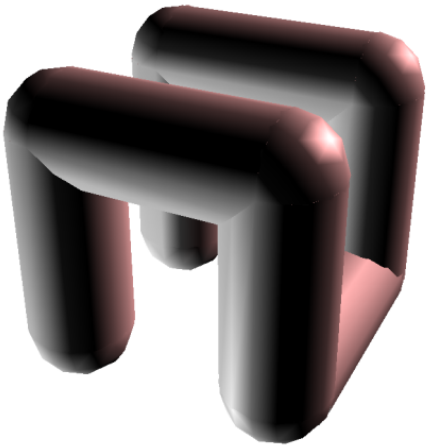
- Initial state: A
- Angle $\alpha = 90^\circ$, Unit step is divided by 2 between two iterations.

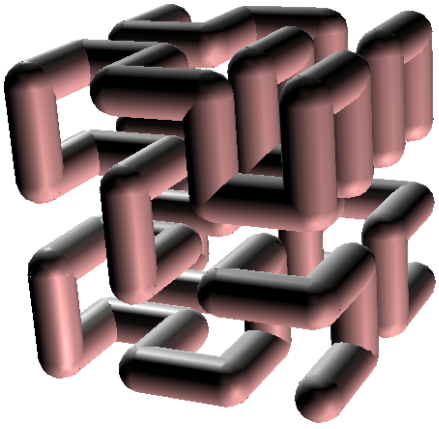
- Production rule:

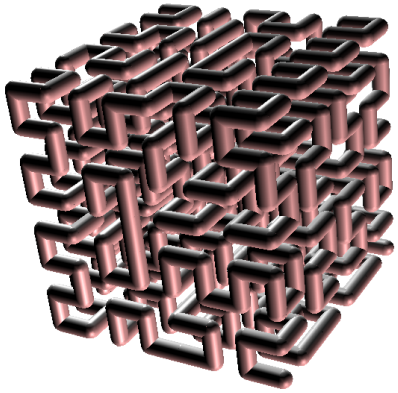
$$
\begin{array}{|l|}
\hline
A \rightarrow B-F+CFC+F-D\&F\hat{}D-F+\&\&CFC+F+B// \\
B \rightarrow A\&F\hat{}CFB\hat{}F\hat{}D\hat{}\hat{}-F-D\hat{}|F\hat{}B|FC\hat{}F\hat{}A// \\
C \rightarrow |D\hat{}|F\hat{}B-F+C\hat{}F\hat{}A\&\&FA\&F\hat{}C+F+B\hat{}F\hat{}D// \\
D \rightarrow |CFB-F+B|FA\&F\hat{}A\&\&FB-F+B|FC//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

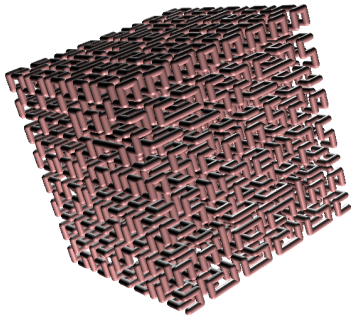
```
to hilbert :p
clearscreen 3d
globalmake "unit 400/power 2 :p
linestart setpenwidth :unit/2
a :p
lineend
view3d
end

to a :p
if :p=0 [stop]
b :p-1 right 90 forward :unit left 90  c :p-1 forward :unit c :p-1
left 90 forward :unit right 90 d :p-1 downpitch 90 forward :unit uppitch 90 d :p-1
right 90 forward :unit left 90 downpitch 180 c :p-1 forward :unit c :p-1
left 90 forward :unit left 90 b :p-1 rightroll 180
end

to b :p
if :p=0 [stop]
a :p-1 downpitch 90 forward :unit uppitch 90 c :p-1 forward :unit b :p-1 uppitch 90
forward :unit uppitch 90 d :p-1 uppitch 180 right 90 forward :unit right 90 d :p-1
uppitch 90 right 180 forward :unit uppitch 90 b :p-1 right 180 forward :unit c :p-1
uppitch 90 forward :unit uppitch 90 a :p-1 rightroll 180
end

to c :p
if :p=0 [stop]
right 180 d :p-1 uppitch 90 right 180 forward :unit uppitch 90 b :p-1 right 90
forward :unit left 90 c :p-1 uppitch 90 forward :unit uppitch 90 a :p-1 downpitch 180
 forward :unit a :p-1 downpitch 90 forward :unit uppitch 90 c :p-1 left 90 forward :unit
left 90 b :p-1 uppitch 90 forward :unit uppitch 90 d :p-1 rightroll 180
end

to d :p
if :p=0 [stop]
right 180 c :p-1 forward :unit b :p-1 right 90 forward :unit left 90 b :p-1 right 180
forward :unit a :p-1 downpitch 90 forward :unit uppitch 90 a :p-1 downpitch 180 forward :unit
b :p-1 right 90 forward :unit left 90 b :p-1 right 180 forward :unit c :p-1 rightroll 180
end
```

And the first iterations:









Nice, isn't it?

Appendix A

List of primitives

The turtle is controlled by means of internal commands called 'primitives' . The following sections set out these primitives:

A.1 Movement of the turtle; pen and color settings

These first primitives govern the movement of the turtle.

forward, fd *n*
Moves the turtle forward n steps in the direction it is currently facing.

back, bk *n*
Moves the turtle backwards n steps in the direction it is currently facing.

right, rt *n*
Turns the turtle n degrees towards the right in relation to the direction it is currently facing.

left, lt *n*
Turns the turtle n degrees towards the left in relation to the direction it is currently facing.

circle *R*
Draws a circle of R radius around the turtle.

arc *R cap1 cap2*
Draws an arc of R radius around the turtle. This arc is inscribed between the caps cap1 and cap2.

home
Returns the turtle to its initial position, that is, the co-ordinates [0 0] with a heading of 0 degrees.

setpos, setposition *list*
Moves the turtle to the co-ordinates specified by the two numbers in the list (x specifies the x-axis and y the y-axis)

setx *x*
Moves the turtle horizontally to the point x on the x-axis

sety *y*

Moves the turtle vertically to the point y on the y-axis

setxy *x y*
Identical to setpos [x y]

setheading, seth *n*
Orients the turtle in the specified direction. 0 corresponds to a position facing vertically upwards. The heading when the turtle is rotated is then based on compass bearings.

label *arg*
Draw the specified word or list at the turtle's location, and following the direction it is facing.
Eg: `label [Hello there!]` will write the sentence "Hello there!" wherever the turtle is, and corresponding to its bearing or heading.

dot *list*
The point defined by the co-ordinates in the list will be highlighted (in the pen colour).

This second group sets out the primitives which allow the properties of the turtle to be adjusted. For example, should the turtle be visible on screen? What colour should it draw when it moves?

showturtle, st
Makes the turtle visible on the screen.

hideturtle, ht
Makes the turtle invisible on the screen.

clearscreen, cs
Empties the drawing area.

wash
Erases the drawing area but leaves the turtle in the same place.

resetall
Initialize the XLogo interface to standard values.

- PenColor: black
- ScreenColor:white
- Animation mode: disabled
- Text and Graphics Font: Dialog 12 pts
- Pen shape: square
- Drawing quality: normal
- Turtles allowed: 16
- Mode trace: disabled
- Screen size: 1000x1000

and empties the drawing area.

pendown, pd
The turtle will draw a line when it moves.

penup, pu
The turtle will not draw a line when it moves.

penerase, pe
The turtle will rub out any marks that it meets.

penreverse, px
Lower the pen and put the turtle in inverted mode.

penpaint, ppt
Lower the pen and put the turtle in classic drawing mode.

setpencolor, setpc *color*
Sets the pen color. See p.93.

setscreencolor, setsc *color*
Sets the screen color. See p.93.

pos, position
Gives the current position of the turtle.Eg: `pos` returns [10 -100]

heading
Gives the bearing or heading of the turtle (cf `setheading`)

towards *list*
The list must contain two numbers representing co-ordinates. Gives the heading which the turtle must follow to go towards the point defined by the co-ordinates in the list.

distance *list*
The list must contain two numbers representing co-ordinates. Gives the number of steps between the current position and the point defined by the co-ordinates in the list.

pencolor, pc
Gives the current colour of the pen. This colour is specified by a list [r g b] where r is the red component, b the blue and g the green.

screencolor, sc
Gives the current colour of the screen (background). This colour is specified by a list [r g b] where r is the red component, b the blue and g the green.

window
Window configuration: the turtle can travel outside the drawing area (but of course, it cannot draw there).

wrap
Window configuration: if the turtle leaves the drawing area, it will reappear on the opposite side!

fence
Window configuration: the turtle is confined to the drawing area. If it is about to go outside, an error message will let you know, and give you the maximum number of steps the turtle can move before the exit point is reached (to within 1 or 2 steps ...).

perspective
Window configuration: the turtle can move through 3d Space. (See Special Section A.2 for this mode). To quit this mode, use one of these primitives `window`, `wrap` or `fence`

findcolor, fc *list*
Returns the colour of the *list* coordinates pixel. This color is determined by a [r g b] list where r is red, g is green and b is blue.

setpenwidth, setpw *n*
Defines the thickness of the pen nib in pixels. The default is 1. The pen has a square or round nib. (Other shapes will be provided in future versions.)

penwidth, pw
Returns the thickness of the pen nib in pixels.

setPenShape, setps *0-1*
Set the pen shape.

- 0→square.
- 1→round.

PenShape, ps
Returns the pen shape.

- 0→square.
- 1→round.

setDrawingQuality, setdq *0-1-2*
Set the drawing Quality.

- 0→normal.
- 1→high.
- 2→low.

DrawingQuality, dq
Returns the drawing Quality.

- 0→normal.

- 1→high.
- 2→low.

setscreensize *list*
Set the screen size to the dimension contained in the list. `setscreensize [1000 1000]`

screensize
Returns the current screen size in a list. `setscreensize [1000 1000]`

setshape *n*
You can choose your preferred turtle with the second tab of menu Options-Preferences.... But you can choose your favourite turtle with `setshape`. The number n goes from 0 to 6. (0 is the triangular shape).

shape
Returns the number that represents the shape of the turtle.

setfontsize, setfs *n*
When you write on the screen with the primitive `label`, it's possible to modify the size of the font with `setfontsize`. The size of the font is 12 by default.

fontsize
Returns the size of the font when you write on the screen with the primitive `label`.

setfontname, setfn *n*
Select the font number *n* when you write on the screen with the primitive `label`. You can find the link between number and font in Menu→OptionsvPreferences→Tab Font.

fontname
Returns a list with two elements. The first is the number corresponding to the font used when you write on the screen with the primitive `label`. The last element is a list which contains the name of the font.

setseparation, setsep *n*
Determines the ratio between the graphic screen and the history zone. The number *n* must be included between 0 and 1. When *n* equals 1 the drawing zone uses all the space, when *n* equals 0, the history zone uses all the window.

separation,sep
Provides the current ratio between the drawing zone and the history zone.

grid *a b*
Draw a grid. Each square has dimension *a* and *b*.

stopgrid
Erase grid.

setgridcolor *color*
Allow the user to choose a custom color for the grid. Eg: `setgridcolor red`

gridcolor
Returns current grid color.

grid?
Return true if the grid is drawn, else return false.

axis *n*
Draw horizontal and vertical axis. The distance between two divisions is *n* steps.

xaxis *n*
Draw only horizontal axis. The distance between two divisions is *n* steps.

yaxis *n*
Draw only vertical axis. The distance between two divisions is *n* steps.

stopaxis
Erase both axis.

setaxiscolor, sac *color*
Allow the user to choose a custom color for the axis. Eg: `setaxiscolor green`

axiscolor
Returns current axis color.

xaxis?
Return true if the horizontal axis is drawn, else return false.

yaxis?
Return true if the vertical axis is drawn, else return false.

zoom *a*
Zoom on the drawing screen. In fact, the number *a* represents the scale regarding to the original image size fixed in the preference panel.

labellength *arg*
Returns the length that needs the word or the list to be displayed on the screen with the primitive `label` using the current font.

zonesize
Returns a list which contains four numbers. These integers are the coordinates of the left upper corner of the drawing zone and the coordinates for the right bottom corner.

message, msg *list*
Shows the message in list in a dialog box, the program stops until the user has clicked the button "OK"

A.1.1 A word on colors

Colors are defined in XLogo with a list of three numbers `[r g b]` between 0 and 255. The number `r` is the red component, `b` the blue and `g` the green. Xlogo has 16 predefined colours: you can access with their rgb list, with a number, or with a primitive. look at this table:

Number	Primitives	[R G B]	Color
0	`black`	[0 0 0]	
1	`red`	[255 0 0]	
2	`green`	[0 255 0]	
3	`yellow`	[255 255 0]	
4	`blue`	[0 0 255]	
5	`magenta`	[255 0 255]	
6	`cyan`	[0 255 255]	
7	`white`	[255 255 255]	
8	`gray`	[128 128 128]	
9	`lightgray`	[192 192 192]	
10	`darkred`	[128 0 0]	
11	`darkgreen`	[0 128 0]	
12	`darkblue`	[0 0 128]	
13	`orange`	[255 200 0]	
14	`pink`	[255 175 175]	
15	`purple`	[128 0 255]	
16	`brown`	[153 102 0]	

```
# These three instructions are the same
setsc orange
setsc 13
setsc [255 200 0]
```

A.1.2 Animation Mode

There are two primitives which allow execution of commands witout the turtle displaying them: `animation` and `stopanimation`

anim, animation

You go into animation mode. The turtle does not draw on the screen anymore but follows the stored line.

To update the drawing on the screen, use the primitive `repaint`. It is very useful to create an animation or to draw a line faster.

stopanim, stopanimation
Animation mode is finished: you switch back to classical mode. You can see the turtle's moves on screen.

repaint
In animation mode, updates the screen: the image on the drawing area is updated.

To identify animation mode, a camera icon appears in the history window. If you click on the icon, the animation mode will stop. It's equivalent to the primitive `stopanimation`.

A.1.3 Writing in the text area with the primitive `print` or `write`

This table sets out the primitives which allow the properties of the text area to be adjusted. Primitive that control the color and the size of the history area, are available only for the primitives `print` or `write`

cleartext, ct
Empties the area containing the command and comment history.

pr, print *arg*
Shows the argument specified in the history zone.

```
print "abcd --------> abcd
pr [1 2 3 4] ----> 1 2 3 4
pr 4 ------------> 4
```

write *arg1*
The same as for the `print` primitive but doesn't go back to the start of the line.

setTextSize, setTS *n*
Define the size of the font in the command history. Only valid with the primtive `print`

textsize, ts
Returns the size of the font with primitive `print`.

setTextColor, setTC *color*
Define the color of the font in command history. Valid only with the primitive . See p.93.

TextColor, tc
Returns the color of the font with the primitive `print` in the command history.

setTextName, setTN *n*
Select the font number n when you write on the the command history with the primitive `print`. You can find the link between number and font in Menu→Options→Preferences→Tab Font.

TextName, tn

Returns a list with two elements. The first is the number corresponding to the font used when you write on the command history with the primitive `print`. The last element is a list which contains the name of the font.

setstyle, setsty *arg*

Set the format of the text in the text area. You can choose between seven styles: `none, bold, italic, strike, underline, superscript, subscript`. If you want several styles together, write them in a list. A few examples for formatting text:

```
setstyle [bold underline] print "hello
```
hello
```
ssty "strike write [strike] ssty "italic write "\ x ssty "superscript print 2
```
~~strike~~ x^2

sty, style

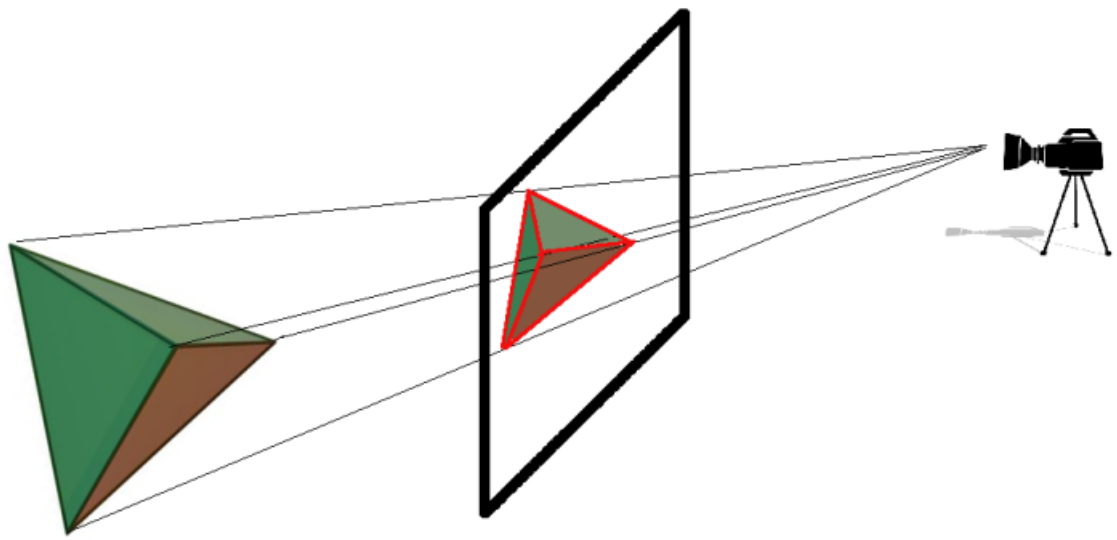
Returns a list which contains the differents styles used for the primitive `print`.

A.2 Turtle and 3D

>From version 0.9.92, our turtle can leave its plane and move into 3D space. To switch to this mode, we use the primitive `perspective`. Welcome to a 3D world!

A.2.1 The perspective projection

To represent a 3D space on a 2D plane, XLogo uses a projection perspective. A camera looks at the 3D scene, where the image from the projection screen is displaying. Here is a little scheme to explain this:



Some primitives allow us to set the camera position. The screen projection is half the distance from the camera.

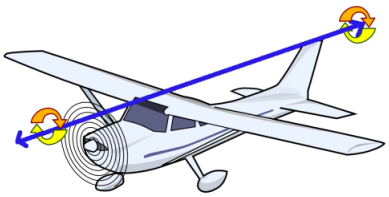
A.2.2 Understanding orientation in a 3D World

In a 2D plane, the turtle's orientation was only defined by its heading. In a 3D world, the turtle's orientation is given by 3 angles:

- Roll: The turtle's angle around axis (Oy)

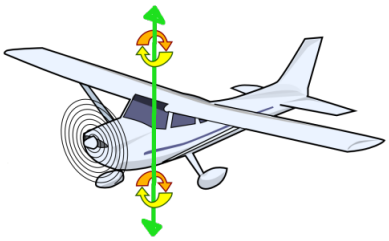
- Pitch: The turtle's angle around axis (Ox)
- Heading: The turtle's angle around axis (Oz)

In fact, to move itself in the 3D World, the turtle is very similar to an aircraft. Here is a little illustration which represents these 3 values:



Roll

Pitch



Heading

It seems quite complex at first, but you will see that a lot of things stay very similar to moving in a 2D plane. Here are the basic primitives for moving in the 3D world:

forward, fd, back, bk n
Same behaviour as in 2D plane.

right, rt, left, lt n
Same behaviour as in 2D plane.

rr, rightroll n
The turtle turns n degrees to the right around its longitudinal axis.

lr, leftroll n
The turtle turns n degrees to the left around its longitudinal axis.

up, uppitch n
The turtle goes n degrees up around its transversal axis.

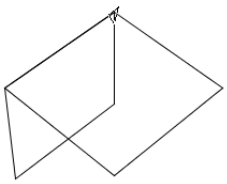
down, downpitch n
The turtle goes n degrees down around its transversal axis.

In the 2D plane, when we want to draw a square of side 200 steps, we write:

```
repeat 4[fd 200 rt 90]
```

These instructions are still available in the 3D world, where the square is drawn in perspective mode. If the turtle goes down 90 degrees, we can draw another square and we obtain:

```
cs
repeat 4[fd 200 rt 90]
down 90
repeat 4[fd 200 rt 90]
```



You just have to try some examples to understand these orientations and become an expert!
You must understand that the 3 rotation primitives are linked together, for example try this:

```
cs
leftroll 90 up 90 rightroll 90
```

The turtles movement is equivalent to `left 90` (You can try with your hand simulating the turtle if you don't understand)

A.2.3 Primitives available in 2D mode and 3D mode

The following primitives are available in 2D plane or in 3D world. The only difference is the arguments received by the primitives. For example, the primitive `setpos` or `setposition` is still waiting for a list as an argument but now, the list must contain three numbers $(x; y; z)$ which represent the three point coordinates. Here are all those primitives:

`circle`	`arc`	`home`	`towards`
`distance`	`setpos, setposition`	`setx`	`sety`
`setheading`	`label`	`labellength`	`dot`
`pos, position`	`heading`		

Primitives only available in 3D mode

setxyz *x y z*
This primitive moves the turtle to the chosen point. This primitive is waiting for three arguments representing the point's coordinates. `setxyz` is very similar to `setpos` but the coordinates are not written into a list. Example, `setxyz -100 200 50`: move the turtle to the point $x = -100; y = 200; z = 50$

setz *z*
This primitive moves the turtle to the point with the valid value z. `setz` is waiting for one number as an argument. This primitive is comparable to `setx` or `sety`.

setorientation *list*
Set the turtle's orientation. This primitive waits for a list which contains 3 numbers, the roll, the pitch and the heading.
Example, `setorientation [100 0 58]`: the turtle has roll: 100 degrees, pitch: 0 degree and heading: 58 degrees.

orientation
Returns the turtle's orientation in a list which contains: `[ roll pitch heading ]`. Note the number order, if for example, the orientation value is `[100 20 90]`, this means that if you want the same orientation starting from the origin position (after a `clearscreen` instruction), you'll have to write the following sequence:

```
rightroll 100 up 20 right 90
```

If you inverse the instruction's order, you don't obtain the valid orientation!

setroll *n*
The turtle turns around its longitudinal axis to the chosen roll angle.

roll
Returns the current roll value.

setpitch n
The turtle turns around its transversal axis to the chosen pitch angle.

pitch
Returns the current pitch value.

A.2.4 3D Viewer

A 3D Viewer is included in XLogo, it allows you to visualize your drawing in 3D. This module uses the JAVA3D library, so it's necessary to have java3D fully installed.

Here are the rules to use the 3D Viewer:
When we create a geometric figure on the drawing area, we have to indicate to the 3D Viewer which shapes we want to record for future visualization. It's possible to record polygons (surfaces), lines, points or text. To use this feature, here are the primitives:

polystart
The following turtle's moves are saved to create a polygon.

polyend
Since the last `polystart` call, the turtle has gone through several vertices. This new polygon is recorded, its color is defined by all vertices color. This primitive finalizes the polygon.

linestart
The following turtle's moves are saved to create a strip line.

lineend
Since the last `linestart` call, the turtle has gone through several vertices. This new line is recorded, its color is defined by all vertices color. This primitive finalizes the strip line.

pointstart
The following turtle's moves are saved to create a point set.

pointend
This primitive finalizes the point set.

textstart
Each time the user displays text on the drawing area with the primitive `label`, it will be recorded and then displayed by the 3D Viewer.

textend
End of text recording.

view3d polyview
Launch the 3D viewer, all recorded objects are drawn on this new window. You have control of the camera scene:

- You can rotate the scene by clicking on the mouse's left button.
- You can translate the scene by clicking on the mouse's right button.

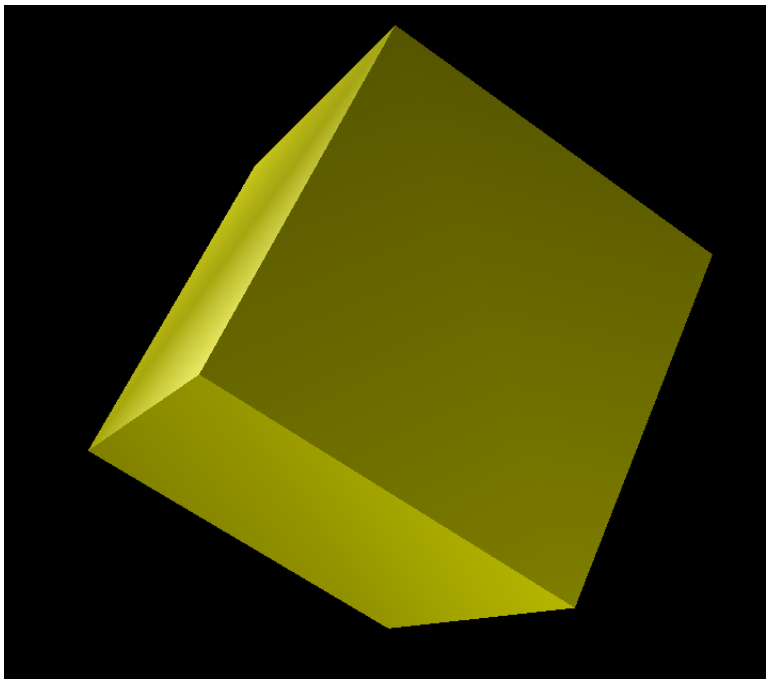
- You can zoom the scene with the mouse's wheel button.

A.2.5 Drawing a cube

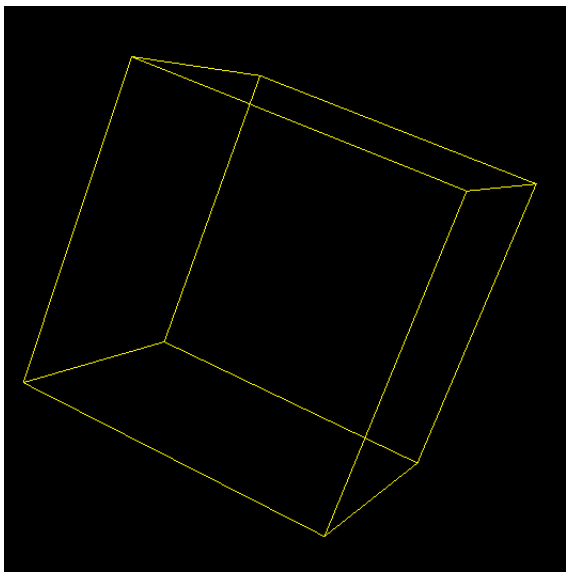
All faces are 400 steps square. Here is the program:

```
to square
# we record the vertice square
polystart repeat 4[forward 400 right 90] polyend
end

to simpleCube
# yellow cube
clearscreen perspective setpencolor yellow
# lateral faces
repeat 4[square penup right 90 forward 400 left 90 rightroll 90 pendown]
# bottom face
downpitch 90 square uppitch 90
# upper face
forward 400 downpitch 90 square
# visualization
view3d
end
```

We launch with the command: `simpleCube`:



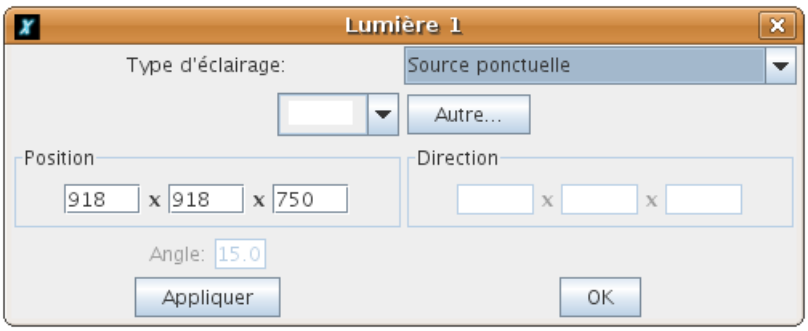
When we replace in the procedure `square`, `polystart` with `linestart` and `polyend` with `lineend`



If we had used `pointstart` and `pointend` instead of `linestart` and `lineend`, we would see on screen only the eight cube vertices. These primitives are very useful to display the point set in 3D Space.

A.2.6 Lighting the scene

You can specify four lights in your 3D scene. By default, the main 3D scene has only two ponctual lights enabled. Click on one of the 4 button lights in the 3D modeler, and this dialog box appears:



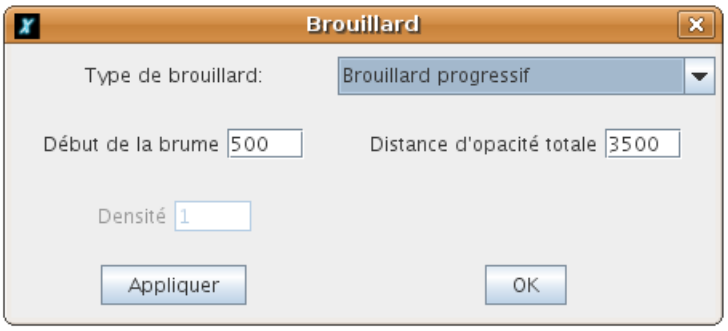
Several light type are available:

- Ambient light: uniform light, you just have to specify its color.
- Unidirectional light: diffuses according to a constant direction. It's the same case as a ponctual light when the source is very very far from the observer. For example, the case of sun.
- Ponctual light: This light has a specified position. This light is similar to a headlight.
- Spot Light: it is a ponctual light but the light is only displayed in a light cone. You have to specify a value angle for this cone.

The best thing is to play with those lights to understand how they work!

A.2.7 Fog effect

You can add a fog effect on the main 3d scene. Click on the cloud button in the 3D scene and this dialog box appears.

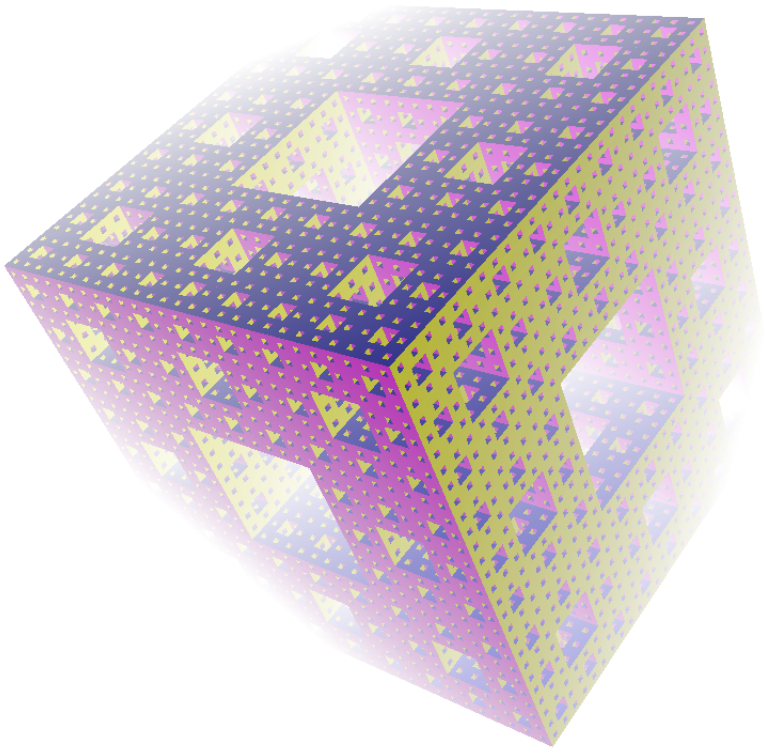


Two fogs are available:

- Progressive fog: this fog's opactity is progressive. You have to specify two parameters:
  - The distance from which fog begins.
  - The disatnce from which opacity is full

- uniform fog: This fog is uniform on the whole scene. You just have to specify the fog's density.

Example with a progressive fog:



A.3 Arithmetical and logical operations

This is a list of number-related commands:

sum x y
Adds the two numbers x and y, and returns the result
Eg: `sum` 40 60 returns 100

difference x y
Returns $x - y$.
Eg: `difference` 100 20 returns 80

minus x
Returns the negative of x.
Eg: `minus` 5 returns -5. See the note at the end of this table.

product x y
Returns the result of multiplying x by y.

div, divide x y
Returns the result of dividing x by y
`div` 3 6 returns 0.5

quotient x y
Returns quotient x by y
`quotient` 15 6 returns 2

mod, modulo x y
Returns the remainder after dividing x by y.

round, rnd x
Returns the nearest whole number to the number x.
`round` 6.4 returns 6

integer, int x
Returns the integer part of the number x. `integer 8.9` returns 8
`integer 6.8` returns 6

power x n
Returns x raised to the power of n.
`power` 3 2 returns 9

squareroot, sqrt x
Returns the square root.

log x
Returns the logarithm of x.

exp x
Returns the exponential of x.

log10 x
Returns the decimal logarithm of x.

sine, sin x

Returns the sine of x. (x is expressed in degrees)

cosine, cos x
Returns the cosine of x. (x is expressed in degrees)

tangent, tan x
Returns the tangent of x. (x is expressed in degrees)

arccosine, acos x
Returns the angle in range [0-180] which cosine is x.

arcsine, asin x
Returns the angle which sine is x.

arctangent, atan x
Returns the angle which tangent is x.

pi
Returns the number π (3.141592653589793)

random, ran n
Returns a random integer between 0 and $n-1$.

absolute, abs x
Returns the absolute value (its numerical value without regard to its sign) of a number.

setdigits n
Sets the number of digits, it sets the precision while calculating. Some more informations:

- By default, 16 digits are allowed.
- If n is negative, the default mode is choosen.
- If n is esual to 0, all numbers are rounded to the unit.

This primitive is useful when you want to calculate with a high precision. Have a look at the example with number π p.43.

digits
Returns the number of digits allowed while calculating. By default, this value is -1.
Important : Be careful with those primitives which require two parameters!
`setxy` a b If b is negative
Eg: For example, `setxy` 200 -10

The Logo interpreter will carry out the operation 200-10 (ie it will subtract 10 from 200). It will therefore conclude that there is only one parameter (190) when it requires two, and will generate an error message. To avoid this type of problem, use the primitive "`minus`" to specify the negative number - `setxy` 200 `minus` 10.
This is a list of logical operators:

or *b1 b2*
Returns true if *b1* or *b2* is true, otherwise returns false

and *b1 b2*
Returns true if *b1* and *b2* is true, otherwise returns false

not *b1*
Returns the negation of *b1*.

- If *b1* is true, returns false.
- If *b1* is false, returns true.

A.4 Operations on lists

word *word1 word2*
Concatenates the two words *word1* and *word2*.
Eg: `pr word` "a 1 returns a1

list *arg1 arg2*
Returns a list composed of *arg1* and *arg2*.
For example, `list 3 6` returns [3 6].
`list` “a “list returns [a list]

sentence, se *arg1 arg2*
Returns a list composed of *arg1* and *arg2*. If *arg1* or *arg2* is a list, then each element of *arg1* and *arg2* will become an element of the resulting list (square brackets are deleted).
Eg: `se` [4 3] “hello returns [4 3 hello]
`se` [how are] “things returns [how are things]

fput *arg1 list2*
Insert *arg1* in the first slot in *list2*.
Eg : `fput` “cocoa [2] returns [cocoa 2]

lput *arg1 list2*
Insert *arg1* in the last slot of *list2*.
Eg: `lput` 5 [7 9 5] returns [7 9 5 5]

reverse *list*
Reverse the order of elements in *list*.
`reverse` [1 2 3] returns [3 2 1]

pick *arg1*

- If *arg1* is a word, returns one of the letters of *arg1* at random.
- If *arg1* is a list, returns one of the elements of *arg1* at random.

remove *arg1 list2*

Remove element *arg1* from list *list2* if it occurs there.
Eg: `remove 2 [1 2 3 4 2 6 ]` returns [1 3 4 6]

item *n arg2*

- If *arg2* is a word, returns the letter numbered *n* from the word (1 represents the first letter).
- If *arg2* is a list, returns the element numbered *n* from the list.

butlast, bl *arg1*

- If *arg1* is a list, returns the whole list except for its last element.
- If *arg1* is a word, returns the word minus its last letter.

butfirst, bf *arg1*

- If *arg1* is a list, returns the whole list except for its first element.
- If *arg1* is a word, returns the word minus its first letter.

last *arg1*

- If *arg1* is a list, returns the last element of the list.
- If *arg1* is a word,returns the last letter of the word.

first *arg1*

- If *arg1* is a list, returns the first element of the list.
- If *arg1* is a word,returns the first letter of the word.

setitem, replace *list1 n arg3*

Replace the element number `n` in the list `list1`, by the word or the list `arg3`.

```
replace [a b c] 2 8 --> [a 8 c]
```

additem *list1 n arg3*

Adds at the position `n` in the list `list1` the word or the list `arg3`

```
additem [a b c] 2 8 --> [a 8 b c]
```

count *arg1*

- If *arg1* is a word, returns the number of letters in *arg1*.

- If *arg1* is a list, returns the number of elements in *arg1*.

unicode *word1*
returns the Unicode value of the character *word1*.
`pr unicode "A` returns 65

character,char *n*
Returns the character which Unicode value is *n*.
`pr character 65` returns `"A`

A.5 Booleans

A boolean is a primitive which returns the word “true or the word “false. These primitives terminate in a question-mark.

true
Returns "true.

false
Returns "false.

word? *arg1*
Returns true if *arg1* is a word, false otherwise.

number? *arg1*
Returns true if *arg1* is a number, false otherwise.

integer? *arg1*
returns true if *arg1* is a whole number, false otherwise.

list? *arg1*
Returns true if *arg1* is a list, false otherwise.

empty? *arg1*
Returns true if *arg1* is an empty word or an empty list, false otherwise.

equal? *arg1 arg2*
Returns true if *arg1* and *arg2* are equal, false otherwise.

before? *word1 word2*
Returns true if *word1* is before *word2* in terms of alphabetical order, false otherwise.

member? *word1 arg2*

- If *arg2* is a list, specifies if *word1* is an element of *arg2*.
- If *arg2* is a word, specifies if *word1* is a letter in *arg2*.

member *word1 arg2*

- If *arg2* is a list, look for the element *word1* in this list. There are two possible outcomes:
  - If *word1* is in *arg2*, returns a sublist containing all list elements from the first instance of *word1* in *arg2*.
  - If *word1* is not in *arg2*, returns the word false.
- If *arg2* is a word, look for the character *word1* in this word. There are two possibilities:
  - If *word1* is in *arg2*, returns the latter part of the word, starting from *word1*.
  - Otherwise, return the word false.

`member` “o “cocoa return ocoa
`member` 3 [1 2 3 4] returns [3 4]

pendown?, pd
Returns the word true is the pen is down, false otherwise.

visible?
Returns the word true if the turtle is visible, false otherwise.

primitive?, prim? *word1*
Returns true if the word is an XLogoprimitive, false otherwise.

procedure?, proc? *word1*
Returns true if the word is a procedure defined by the user, false otherwise.

var? variable? *word1*
Return true if :a is a variable, false otherwise.

A.6 Testing an expression with the primitive `if`

As in all programming languages, Logo allows you to check if a condition is satisfied and then to execute the desired code if it’s true or false.
With the primitive `if` you can realize those tests.

if *expression_ test list1 list2*
if `expression_test` is true, the instructions included in `list1` are executed. Else, if `expression_test` is false, the instructions in `list2` are executed. This second list is optional.

Examples:

- `if 1+2=3[print "true][print "false]`
- `if (first "XLOGO)="Y [fd 100 rt 90] [pr [ XLOGO starts with a X!] ]`
- `if (3*4)=6+6 [pr 12]`

Important: When the result of the first predicate is equal to `false`, the primitive `if` looks for a second list, I mean an expression starting with a square bracket. In some very rare cases, it can't be done, and you'll have to use the primitive `ifelse` . For example:

```
# We  affect two lists in variables a and b
 make "a [print true]
 make "b [print false]

# First test with the primitive if--> The second list can't be evaluated.
 if 1=2 :a :b
 What to do with [print false]?

# Second test with the primitive ifelse --> success.
 ifelse 1=2 :a :b
 false
```

A.7 The workspace

The workspace contains all the elements defined by the user. These are:

- The procedures
- The variables
- The property lists

A.7.1 Procedures

Procedures are a kind of "program". When a procedure is called, the instructions in the body of the procedure are executed. A procedure is defined with the keyword **to**.

```
to name_of_procedure :v1 :v2 :v3 .... [:v4 ....] [:v5 ....]
Body of the procedure
end
```

- name_of_procedure is the name given to the procedure.
- :v1 :v2 :v3 stand for the variables used internally in this procedure. (local variables).
- [:v4 ...], [:v5 ...] are optional variables that we could add into the procedure (go below for more explanations).
- Body of the procedure represents the commands to be executed when this procedure is called.

Eg:

```
to square :s
repeat 4[fd :s rt 90]
end
```

The procedure is called `square` and takes a parameter called s. `square` 100 will therefore produce a square, the length of whose sides is 100. (See the examples of procedures at the end of this manual.)

Since version 0.7c, it is possible to insert comments in the code preceded by #.

```
to square :s
#this procedure allows a square to be drawn whose side equals :s.
repeat 4[fd :s rt 90] # handy, isn't it?
end
```

Optional variables

It's now possible to add optional arguments to XLogo's procedure. Look at the example below:

```
to poly :n [:l 10]
repeat :n [fd :l rt 360/:n]
end

# this command will draw a regular polygon with
# 20 sides of 10
poly 20
```

During the interpretation, the variable `:l` has been replaced by its default value, I mean 10. If I we want to modify this value, we have to call the procedure `poly` between parenthesis to notice to the interpreter that we're going to use optional arguments.

```
# This command will draw a regular polygon with 20
# sides. Every side is 5 long.
 (poly 20 5)
# This is a square with each side equal to 100
(poly 4 100)
```

Primitive 'trace'

It is possible to follow the working of a program or have it show the procedures which are working. This mode can also show if the procedures provide arguments thanks to the primitive `output`.

trace

Activate `trace` mode

stoptrace

`stoptrace` will disactivate the `trace` mode.

A small example with the factorial (see page 42).

```
trace pr fac 4
fac 4
  fac 3
    fac 2
      fac 1
      fac returns 1
    fac returns 2
  fac returns 6
fac returns 24
24
```

A.7.2 Concept of variables

There are two kinds of variables:

- Global variables: these are always accessible from any location in the program.

- Local variables: these are only accessible in the procedure where they are defined.

In this version of Logo, local variables are not accessible in sub-procedures. At the end of the procedure, the local variables are deleted.

make *word1 arg2*
If the local variable *word1* exists, assigns it the value *arg2*. If not, creates a global variable *word1* and assigns it the value *arg2*.
Eg: `make` "a 100 assigns the value 100 to the variable a

local *arg1*
Creates a variable called *arg1*. If *arg1* is a list, creates all variables contained in the list. Note, this is not initialised. To assign it a value, see `make`.

localmake *word1 arg2*
Creates a new local variable and assigns it the value *arg2*.

define, def *word1 list1*
Define a new procedure called *word1*.
`list1` contains several lists:

- The first list contains all variables and optional variables.
- Then, each following list represents a procedure line.

```
def "polygon [ [nb length] [repeat :nb[fd :length rt 360/:nb] ]
```

—-> this command defines a procedure called `polygon` with two variables (`:nb` and `:length`). This procedure draws a regular polygon, we can choose the number of sides and their length.

text *word1*
Returns all information about the procedure called `word1`. It gives a list which contains several lists.

- The first list contains all variables and optional variables about the procedure `word1`.
- Then, each following list represents a procedure line.

This primitive is of course associated to `define`.

thing *word1*
returns the value of the variable *word1*. `thing "a` is similar to `:a`

eraseprocedure, erp *arg1*
Deletes the procedure calling *arg1* or all procedures that contains the list *arg1*.

erasevariable, erv *arg1*
Deletes the variable *arg1* or all variables that contains the list *arg1*.

erasepropertylist, erpl *arg1*
Deletes the property list *arg1* or all property lists that contains the list *arg1*.

erall, eraseall

Deletes all the variables, property lists and procedures currently running.

bye
Quit XLogo.

procs, procedures
Returns a list which contains all the procedures currently defined.

variables vars
Returns a list which contains all the defined variables.

pls, propertylists
Returns a list which contains all the property lists currently defined.

primitives
Enumerates all primitives for the current language.

contents
Returns a list which contains 3 lists. The first list contains all procedures names, the second all variables names and the last all propert lists names.

run *list1*
Executes the list of instructions contained in *list1*

A.7.3 Property Lists

Now, you can define property lists with XLogo. Each list has a specific name and contains some key-value couples.

For example, we can consider a property list named "car". It should contain the key "color" associated to the value "red", or the key "type" and the value "4x4".

To handle these lists, we can use the following primitives:

pprop putproperty *listname key value*
Adds a property to the property list named *listname*. *value* will be accessible with the key *key*. If no property list named *listname* exists, then it will be created.

gprop getproperty *listname key*
Returns the value associated with the key *key* in the property list called *listname*. If this property list doesn't exist or if there isn't any valid key, it returns an empty list.

rprop removeproperty *listname key*
Removes the corresponding couple *key-value* in the property list *listname*.

plist propertylist *listname*
Displays all key-value couples contained in the property list called *listname*.

Let's return to the property list called "car".

```
# Filling property list
pprop "car "color "red
pprop "car "type "4x4
```

```
pprop "car "vendor "Citroën

# Display one value
print gprop "car "color
red

# Display all elements
print plist "car
color red type 4x4 vendor Citroën
```

files
By default, lists the contents of the directory. (Equivalent to the ls command for Linux users and the dir command for DOS users)

loadimage, li *list*
Load the image file contained in the list. Its upper left corner will be placed at the turtle's location. The only supported formats are .png and .jpg. The path specified must be relative to the current folder. Eg: `setdir "C:\\my_images_dir loadimage "turtle.jpg`

setdirectory, setdir *word1*
Specifies the current directory. The path must be absolute. The directory must be specified with a word.

changedirectory, cd *word1*
Allows to choose the current directory. The path is related to the current directory. You can use the '`..`' notation to refer to the parent directory.

directory, dir
Gives the current directory. The default is the user's home directory, ie `/home/your_login` for Linux users, `C:\WINDOWS` for Windows users.

save *word1 list2*
A good example to explain this: `save "test.lgo [proc1 proc2 proc3]` saves in the file `test.lgo` in the current directory the procedues `proc1, proc2 et proc3`. If the extension .lgo is omitted, it is added by default. *word1* gives a relative path starting from the current directory. This command will not work with an absolute path.

saved *word1*
`saved "test.lgo` saves in the file `test.lgo` in the current directory the collection of procedures currently defined. If the extension .lgo is omitted, it is added by default. *word1* gives a relative path starting from the current directory. This command will not work with an absolute path.

ed, edit *arg1*
Open the editor with all the procedures specified in the list *arg1* or in the word *arg1*.

edall, editall
Open the editor with all the currently defined procedures.

load *word1*
Opens and reads the file *word1*. For example, to delete all the defined procedures and load the file `test.lgo`,

you would use: `efns load "test.lgo`. *word1* gives a relative path starting from the current directory. This command will not work with an absolute path.

openflow *id file*
When you want to read or write in a file, you must first open a flow toward this file. The argument *file* must be the name of the file you want. You must use a phrase to show the name of the file in the current directory. The *id* argument is the number given to this flux so as to identify it.

listflow
Shows the list of the various open fluxes with their identifiers.

readlineflow *id*
Opens the flow which identifier corresponds to the number used as argument and then reads a line in this file.

readcharflow *id*
Opens the flux which identifier corresponds to the number used as argument and then reads a character in this file. This primitive sends back a number representing the value of the character (similar to `readchar`).

writelineflow *id list*
Writes the text line included in *list* at the beginning of the file identified thanks to the identifier *id*. Be careful, the writing is effective only after the flow has been closed by the primitive `closeflow`.

appendlineflow *id list*
Writes the text line included in *list* at the end of the file identified thanks to the identifier *id*. Be careful, the writing is effective only after the flux has been closed by the primitive `closeflow`.

closeflow *id*
Closes the flux when its identifier number is written as argument.

endflow? *id*
Sends back `"true` if it is the end of the file. Otherwise sends back `"false`.

Here is an example of the use of primitives allowing to read and write in a file. I will give this example in a Windows-type framework. Other users should be able to adapt the following example.

The aim of this file is to create the file `c:\example` containing the following three lines:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
Abcdefghijklmnopqrstuvwxyz
0123456789

```
# You open a flow towards the desired file. This flow is given the number 2
setdirectory "c:\\
openflow 2 "example
# You type the desired lines
writelineflow 2 [abcdefghijklmnopqrstuvwxyz]
writelineflow 2 [abcdefghijklmnopqrstuvwxyz]
writelineflow 2 [0123456789]
# You close the flux to end the writing
closeflow 2
```

Now, you can see the writing procedure went alright:

```
# You open a flow towards the file you want to read. This flow is given the number 0
openflow 0 "c:\\example
# You read the one after the other the different lines from the file
pr readlineflow 0
pr readlineflow 0
pr readlineflow 0
# You close the flow
closeflow 0
```

if you wish to add the line 'Great !':

```
setdirectory "c:\\
openflow 1 "example]
appendlineflow 1 [Great!]
closeflow 1
```

A.8 Advanced fill function:

Two primitives allow to colour a figure. The primitive `fill` and `fillzone`. These primitives allow a shape to be coloured in. These primitives can be compared with the 'fill' feature available in many image-retouching programs. This feature can extend to the margins of the design area. There are two rules that must be adhered to in order to use this primitive correctly:

1. The pen must be lowered (`pd`).
2. The turtle must not be located on a pixel of the colour with which the shape is to be filled. (If you want to colour things red, it can't be sitting on red...)

Let's take a look at an example to see the difference between `fill` and `fillzone`:
The pixel under the turtle is white right now. The primitive `fill` will colour all the neighbouring white

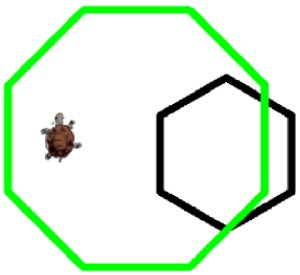


Figure A.1: At the beginning

pixels with the current pen colour. If for example, you type: `setpc 1 fill`. Let's now go back to the first case, if the pen colour of the turtle is black, the primitive `fillzone` colours all pixels until it encounters the current colour (here black).

This is a good example of the use of this primitive:

```
to halfcirc :c
# draw a half-circle of diameter :c
repeat 180 [fd  :c*tan 0.5 rt 1]
fd :c*tan 0.5
rt 90 fd :c
```

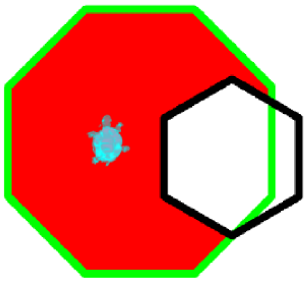


Figure A.2: With the primitive `fill`

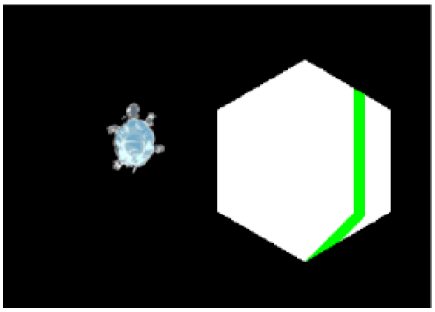


Figure A.3: With the primitive `fillzone`, if you type: `setpc 0 fillzone`

```
end

to tan :angle
# renders the tangent of the angle
output (sin :angle)/cos :angle
end

to rainbow :c
if :c<100 [stop]
halfcirc :c rt 180 fd 20 lt 90
rainbow :c-40
end

to dep
pu rt 90 fd 20 lt 90 pd
end

to arc
ht rainbow 400 pe lt 90 fd 20 bk 120 ppt pu rt 90 fd 20 pd
setpc 0 fill dep
setpc 1 fill dep
setpc 2 fill dep
setpc 3 fill dep
setpc 4 fill dep
setpc 5 fill dep
setpc 6 fill dep
end
```

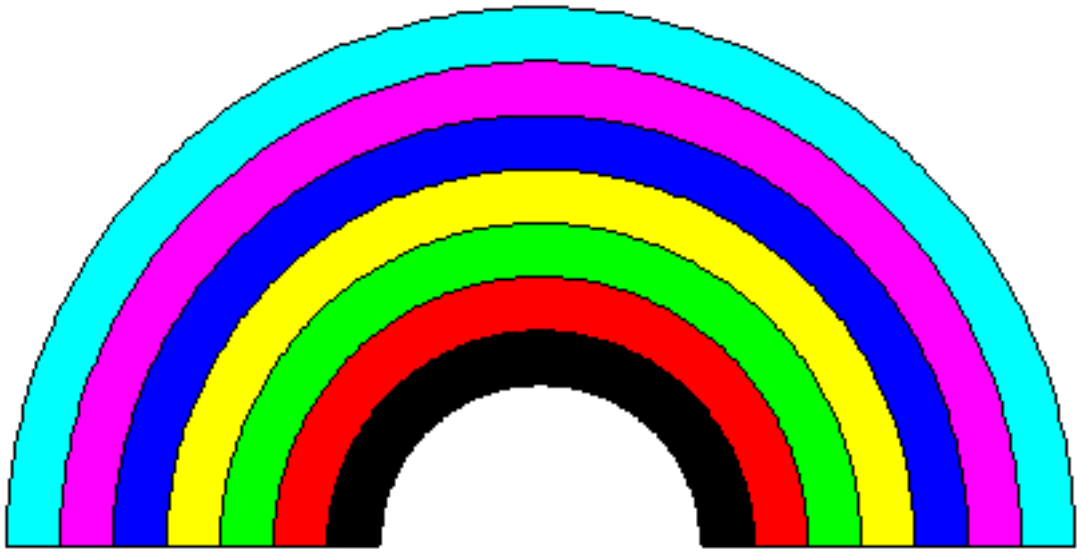


Figure A.4: Arc-in-LOGO

A.9 Break commands

XLogo has three break commands: `stop`, `stopall` and `op, output`.

stop

`stop` can have two results.

- If it is included in a `repeat` or `while` loop, the program jumps out of the loop there and then.
- If it occurs in a procedure, the program breaks out of the procedure immediately.

stopall

The program breaks out of all procedures immediately and stops.

output, op

`output, op` allows breaking out of a procedure with a value to be returned.

A.10 Multiturtle Mode

It's possible to have several active turtles on the screen. By default, on Xlogo startup, only one turtle is available. Its number is 0. If you want to "create" a new turtle, you can use the primitive `setturtle` followed by the number of the turtle. To prevent obstruction, the turtle is created on the origin and is invisible (you must use `showturtle` to show it). Then, the new turtle is the active turtle, it obeys all classic primitives while you don't change the active turtle with `setturtle`. The maximum number of available turtles can be set in menu Options - Preferences - Tab options.

Here are the primitives for the multiturtle mode:

setturtle, sturtle n

The turtle numero n is now the active turtle. By default on Xlogo startup the active turtle is the number 0.

turtle

Returns the number of the active turtle.

turtles
Returns a list which contains all the numero af the turtles actually on the screen.

eraseturtle, ert *n*
Delete the turtle number *n*

setTM, setturtlesmax *n*
Set the maximum number of turtles for multiturtle mode.

tm, turtlesmax
Returns the maximum number of turtles for multiturtle mode.

A.11 Play music

sequence, seq *list*
Put in memory the sequence in the list. Read after this table how to write a sequence.

play
Play the sequence in memory.

instrument, instr
Returns the number that corresponds to the selected instrument.

setinstrument, sinstr *n*
The selected instrument is now the instrument number *n*. You can see the list of all available instruments in menu Options-Preferences-Tab Sound.

indexsequence, indseq
Returns where the cursor is located in the current sequence.

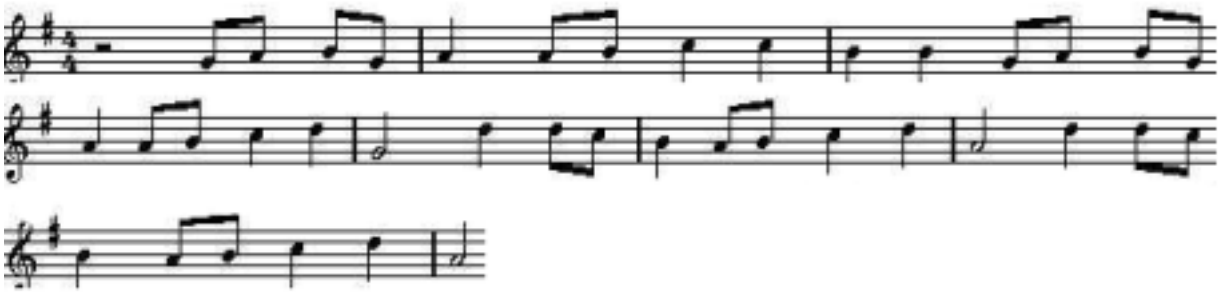
setindexsequence, sindseq *n*
Put the cursor to index *n* in the current sequence in memory.

deletesequence, delseq
Delete the current sequence in memory.

If you want to play music, you must put the notes in memory in a list called sequence. To create the sequence, you can use the primitive `seq` or `sequence`.

These are the rules to follow to create a valid sequence:
`do re mi fa sol la si` : the usual notes of the first octave.
To make a sharp re, we note `re +`
To make a flat re, we note `re -`
If you want to go up or down and octave, we use symbol ":" followed by + or -. E.g. After :++ in the sequence, all the notes will be played two octaves up (two ++) .
By default, notes are played for a duration of one. If you want to increase or decrease, you write the number

that corresponds to the duration of notes. E.g. `seq [sol 0.5 la si]`. will play sol with a duration 1 and la si with a duration 0.5 (twic as fast).
If you want to play this example:



```
to tabac
# create the sequence of notes
seq [0.5 sol la si sol 1 la 0.5 la si 1 :+ do do :- si si 0.5 sol la si sol
          1 la 0.5 la si 1 :+ do re 2 :- sol ]
seq [:+ 1 re 0.5 re do 1 :- si 0.5 la si 1 :+ do re 2 :- la ]
seq [:+ 1 re 0.5 re do 1 :- si 0.5 la si 1 :+ do re 2 :- la ]
seq [0.5 sol la si sol 1 la 0.5 la si 1 :+ do do :- si si 0.5 sol la si sol
          1 la 0.5 la si 1 :+ do re 2 :- sol ]
end
```

To hear music, launch the command: `tabac play`
Now, we can see/hear an interesting application of the primitive sindseq. Write those commands:

```
delseq       # Delete the sequence in memory
tabac       # Put in memory the notes
sindseq 2   # Put the cursor on the second "la".
tabac       # Put in memory the same sequence but translated of 2.
play        # Great!
```

You can choose your instrument with the primitive `sinstr` or with the menu Options-Preferences-Tab sound. You will find the list of all available instruments with their associated number.

A.12 Loops:

XLOGO has three primitives which allow the construction of loops: `repeat`, `for` and `while`.

A.12.1 A loop with `repeat`

repeat *n list_ of_ commands*
n is a whole number and `list_of_commands` is a list containing the commands to execute. The LOGO interpreter will implement the commands in the list n times: that avoids the need to copy the same command n times!
Eg:

```
repeat 4 [forward 100 left 90]      # A square of side 100
repeat 6 [forward 100 left 60]      # A hexagon of side 100
repeat 360 [forward 2 left 1]       # A uh... 360-gon of side 2
                                           # In short, almost a circle!
```

repcount
Included in a `repeat` loop. Its an internal variable. Returns the number of the running iteration. (The first iteration is number 1).

```
repeat 3 [pr repcount]
1
2
3
```

A.12.2 A loop with `for`

for *list1 list2*
`for` assigns to a variable some successive values in a fixed range with a choosen increment.
list1 contains three arguments: the variable name, the start value, the end value.
A fourth argument is optionnal representing the increment (the step between two successive values). Default value is 1. Here are a few examples:

```
for [i 1 4][pr :i*2]
2
4
6
8

# Now, i is going from 7 to 2 falling down of 1.5 each times
# Look at the negative increment
# Then, Displays its square.

for [i 7 2 -1.5 ][pr list :i power :i 2]

7 49
5.5 30.25
4 16
2.5 6.25
```

A.12.3 A loop with `while`

while *list_ to_ evaluate list_ of_ commands*
list_ to_ evaluate is a list containing an instruction set which can be evaluated as a boolean. *list_ of_ commands* is a list containing the commands to execute. The LOGO interpreter will continue implementing the *list_ of_ commands* so long as the *list_ to_ evaluate* is returned as true.
Eg:

```
while ["true] [rt 1]                    # The turtle will turn around

# An example which allows us to spell the alphabet in reverse

make "list "abcdefghijklmnopqrstuvwxyz
while [not empty? :list] [pr last :list make "list butlast :list]
```

A.12.4 A loop with `foreach`

foreach *variable_ name arg1 instructions*
The variable has for successive value the item from a list, or the character from a word. The instructions are repeated for each value of the variable.

```
foreach "i "XLOGO [print :i]
X
L
O
G
O
foreach "i [a b c] [print :i]
a
b
c

make "sum 0 foreach "i 12345 [make "sum :sum+:i] print :sum
15
```

A.12.5 A loop with `forever`

forever *instructions_ list*
Repeat forever a block of instructions waiting for a command to stop the loop.

`Eg: forever [fd 1 rt 1]`

Be careful when you use this primitive because of the infinite loop!

A.13 Receiving input from the user

A.13.1 Interact with the keyboard

Currently, text can be accepted from the user during program execution mainly via 3 primitives: `key?`, `readchar` and `read`.

key?
Is read as true or false according to whether a key has been pressed or not since the start of program execution.

readchar

- If `key?` is false, the program is paused until the user presses a key.
- If `key?` is true, it gives the key which was pressed last.

These are the values given for particular keys:
If you are uncertain about the value returned by a key, you can type:
`pr readchar`. The interpreter will then wait for you to type on a key before giving you the corresponding value.

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

Table A.2: Values for particular keys

read *list1 word2*

Presents a dialogue box whose title is *list1*. The user can then input a response in a text field, and the response will be stored in the form of a word or a list (if the user wrote several words) in the variable *word2*, and will be evaluated when the OK button is pressed.

A.13.2 Some examples of usage:

```
to vintage
read [What is your age?] "age
make "age :age
if :age<18 [pr [you are a minor]]
if or :age=18 :age>18 [pr [you are an adult]]
if :age>99 [pr [Respect is due!!]]
end

to rallye
if key? [
make "car readchar
if :car=-37 [lt 90]
if :car=-39 [rt 90]
if :car=-38 [fd 10]
if :car=-40 [bk 10]
if :car=27 [stop]
]
rallye
end
# You can control the turtle with the keyboard, and stop with Esc
```

A.13.3 Interact with the mouse

Currently, mouse events can be accepted from the user during program execution via three primitives: `readmouse`, `mousepos` and `mouse?`.

readmouse

The program is paused until the user presses the mouse. Then, it returns a number that represents the event. These are the differents values:

- 0→The mouse has moved.
- 1→The button 1 has been pressed.
- 2→The button 2 has been pressed.

The button 1 is the left button , the button 2 is the next on the right ...

mousepos, mouseposition
Returns a list that contains the position of the mouse.

mouse?
Returns `true` if we touch the mouse since the program begins. Returns `false` otherwise.

A.13.4 Some examples of usage:

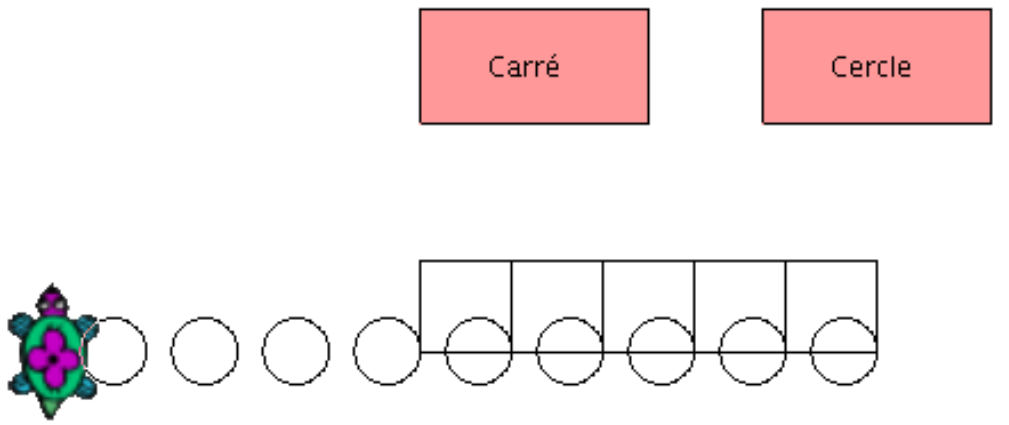
In this first procedure, the turtle follows the mouse when it moves on the screen.

```
to example
# when the mouse moves, go to the next position
if readmouse=0 [setpos mousepos]
example
end
```

In this second procedure, it's the same but you must click with the left button of the mouse if you want the turtle to move.

```
to example2
if readmouse=1 [setpos mousepos]
example2
end
```

In this third example, we create two pink buttons. If we left-click on the left button, we draw a square with a side of 40. if we left-click on the right button, we draw a little circle. Last, if we right-click on the right button, it stops the program.




```
to button
#create a pink rectangular button (height 50 - width 100)
repeat 2[fd 50 rt 90 fd 100 rt 90]
rt 45 pu fd 10 pd setpc [255 153 153]
fill bk 10 lt 45 pd setpc 0
end

to lance
cs button pu setpos [150 0] pd button
pu setpos [30 20] pd label "Square
pu setpos [180 20] pd label "Circle
pu setpos [0 -100] pd
mouse
```

```
end

to mouse
# we put the value of readmouse in the variable ev
make "ev readmouse
# we put the first coordinate of the mouse in variable x
make "x item 1 mousepos
# we put the second coordinate of the mouse in variable y
make "y item 2 mousepos
# When we click on the left button
if :ev=1 & :x>0 & :x<100 & :y>0 & :y<50 [square]
# When we click on the right button
if  :x>150 & :x<250 & :y>0 & :y<50 [
          if :ev=1 [circle]
          if :ev=3 [stop]
]
mouse
end

to circle
repeat 90 [fd 1 lt 4] lt 90 pu  fd 40 rt 90 pd
end

to square
repeat 4 [fd 40 rt 90] rt 90 fd 40 lt 90
end
```

A.13.5 Graphical components

With XLogo, you can add several graphical components on the drawing area (Button or Menu). All the primitives allowing the user to manipulate those components start with the prefix GUI (for Graphical User Interface).

Create a component

First, you need to create those graphical objects, then you can modify some of their properties and last, you can display them on the drawing area

- To create a button:

  guibutton *word1 word2*

  Create a button whose title is *word2*. The button name is *word1*

- To create a combo Menu:

  guimenu *word1 list2*

  This command creates a combo menu which name is *word1* and which contains items from *list2*

  ```
  guimenu "myMenu [item1 item2 item3]
  ```

Modify some properties of graphical components

guiposition *word1 list2*

Locates the graphical element *word1* on a specific place with its coordinate. For example, if you want to put the button at the point with coordinates (20; 100), you will write:

```
guiposition "b [20 100]
```

If you don't specify a location for the component, it will be placed by default on the upper left corner of the drawing area.

guiremove *word1*
Remove a graphical component. For example, to delete the button:

```
guiremove "b
```

guiaction *word1 list2*
Defines an action for the component when the user interacts with it.

```
# the turtle forwards of 100 if we click on the button "b
 guiaction "b [fd 100 ]

# For the combo menu, each  item has its own action
guiaction "m [[print "item1]  [print "item2] [print "item3]]
```

guidraw *word1*
Displays the graphical component on the drawing area. For example, to display the button:

```
guidraw "b
```

A.14 Time and date

XLogo has several primitives for date, time or generating countdown.

wait n
Halts the program, and therefore the turtle, for $\frac{n}{60}$ seconds.

countdown n
Starts a countdown of n seconds. We know if this countdown has finished with the primitive `endcountdown?`

endcountdown?
Returns `"true` if there's no active countdown. Returns `"false` if the countdown is active.

date
Returns a list wich contains three integers representing the date. The first integer indicates the day, the second the month and the last the year. —> [day month year]

time
Returns a list of three integers representing the time. The first integer indicates the hour, the second the minutes and the last the seconds. —> [hour minute seconde]

pasttime
Returns the past time in seconds since XLogohas started.

Difference between `wait` and `countdown` is that `countdown` doesn't halt the program.

Here is an example:

```
to clock
# shows time in numerical format
# we refresh the time each five seconds
if endcountdown? [
cs
sfont 75 ht
make "heu time
make "h first :heu
make "m item 2 :heu
# We shows two number for seconds and minutes. (we must add a 0)
if :m-10<0 [make "m word 0 :m]
make "s last :heu
# We shows two number for seconds and minutes. (we must add a 0)
if :s-10<0 [make "s word 0 :s]
label word word word word :h ": :m ": :s
countdown 5
]
clock
end
```

A.15 Using a network with XLogo

A.15.1 The network How to

First, we have to introduce the basis for network communication before we can use the XLogo primitives. Two computers (or more) can communicate through a network if they both have ethernet cards. Each com-

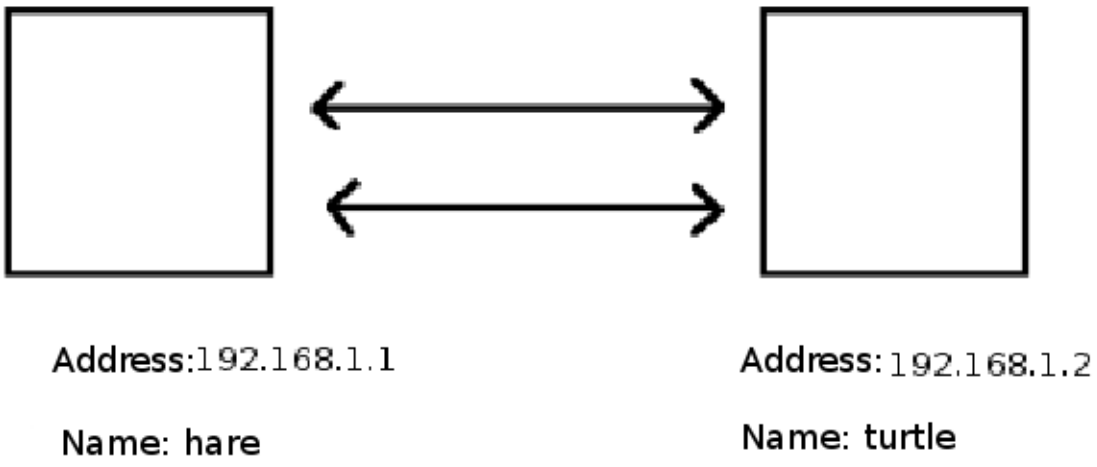


Figure A.5: A simple network

puter is identified by a personal address called an *IP address*. This IP address consists of four integers, each between 0 and 255 and separated by a dot. For example, The IP address of the first computer in the

illustration is 192.168.1.1

Because it's not easy to remember these numbers, it's also possible to identify each computer by a more usual name. As can be seen in the illustration, we can communicate to the right computer with its IP address: 192.168.1.2, or with its name: `turtle`

For the moment, I'll add just one more thing. The local computer on which you are working is located by the address: 127.0.0.1. Its general name is `localhost`. We will see this later in practice.

A.15.2 Primitives for networking

XLogo has 4 primitives that allow it to communicate over a network: `listentcp`, `executetcp`, `chattcp` and `send`. In all future examples, we will take the case of the two computers in the previous figure.

listentcp

This primitive `listentcp` is the basis for all network communication. It doesn't need an argument. When you execute this primitive on a computer, the computer will listen for instructions sent from other computers on the network.

executetcp *word1 list2*

this primitive allows execution of instructions by a computer on the network.
word1 is the called IP address or computer name, the *list2* contains instructions to execute.

Example: I'm on computer `hare`, I want to draw a square with a side of 100 on the other computer. Thus, on the computer `turtle`, I have to launch the command `listentcp`. Then, on the computer `hare`, I write:

```
executetcp "192.168.1.2 [repeat 4[fd 100 rt 90]]
or
executetcp "turtle [repeat 4[fd 100 rt 90]]
```

chattcp *word1 list2*

Allows chat between two computers on a network. On each computer, it displays a chat window.
word1 is the called IP address or computer name, *list2* contains the sentence to display.

Example: `hare` wants to talk with `turtle`.
First `turtle` executes `listentcp` so it is waiting for instructions from network computers. Then `hare` writes: `chattcp "192.168.1.2 [hello turtle]`.
Chat windows will open on both computers, allowing them to talk with each other.

sendtcp *word1 list2*

Send data towards a computer on the network and return his answer.

word1 is the called IP address or computer name, *list2* contains the data to send. When Xlogo is launched on the other computer, il will answer OK. It is possible with this primitive to communicate with a robot through its network interface. Then, the answer of the robot could be different.

Example: `turtle` wants to send to `hare` the sentence "3.14159 is quite pi".
First `hare` executes `listentcp` so it is waiting for the other computer to communicate. Then, `turtle` writes: `print sendtcp "hare [3.14159 is quite pi]`.

A little hint: Launch two instances of XLogo on the same computer.

- In the first window, execute `listentcp`.
- In the second one, write `executetcp "127.0.0.1 [fd 100 rt 90]`

You can move the turtle in the other window! (heh, heh, it's possible because 127.0.0.1 designates your local address, so it's your own computer...)

Appendix B

Launching XLogo with command line

To execute XLogo, here is the syntax of the command:

```
java -jar xlogo.jar [-a] [-lang en] [-memory 64] [file1.lgo file2.lgo ...]
```

List of available options:

- Attribute `-lang`: this attribute specifies a language for XLogo. This parameter overwrites the one from the config file called `.xlogo`. Have a look at the following table which shows all available languages:

French	English	Spanish	german	Arabic	Portuguese	Espéranto	Galician	Greek
fr	en	es	de	ar	pt	eo	gla el	

- Attribute `-a`: this attribute indicates execution of the main command, contained in the loaded files on startup, after XLogo's window has opened.

- Attribute `-memory`: this attribute changes the corresponding memory space allocated to XLogo.

- file1.lgo, file2.lgo ...: these files in format `.lgo` are loaded on XLogo startup. These files could be local or distant. Hence, you can specify a local address or a web address.

- Attribute `-tcp_port`: this attribute allows to modify the default TCP port used for networking (See p.125). By default, its value is 1948.

A few examples:

- `java -jar xlogo.jar -lang es prog.lgo`:
  Files `xlogo.jar` and `prog.lgo` are in the current directory. This command executes XLogo, with language configured to spanish. Then, it loads the file `prog.lgo` (Thus, this file is written in spanish...)

- `java -jar xlogo.jar -a -lang en http://xlogo.tuxfamily.org/prog.lgo`:
  This command executes XLogo in english. It loads the file `http://xlogo.tuxfamily.org/prog.lgo`. Finally, the main command from this file is executed on startup.

Appendix C

Executing Xlogo from the WEB

C.1 The problem

You're managing a web site. On this site, you're talking about XLogo and you want to provide some of the programs you have created with XLogo. You could distribute the Logo file in format `.lgo`, but it would be better if the user could launch Xlogo on line and directly test your program.

To launch XLogo from a web site, we'll use the technology JAVA WEB START. In fact, we just need to put on our site a link towards a file with extension `.jnlp`. It will execute XLogo on line.

C.2 How to create the `jnlp` file

Here is an example of such a file. In fact, the following example is the one used on the french site in the section called "exemples". This file allows loading of the program that draws a dice in the 3D section. Explanation of the file's contents will be given after the code.

```
 <?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.5+" codebase="http://downloads.tuxfamily.org/xlogo/common/webstart">
<information>
  <title>XLogo</title>
  <vendor>xlogo.tuxfamily.org</vendor>
  <homepage href="http://xlogo.tuxfamily.org"/>
  <description>Logo Programming Language</description>
  <offline-allowed/>
</information>

<security>
<all-permissions/>
</security>

<resources>
  <j2se version="1.4+"/>
  <jar href="xlogo.jar"/>
</resources>

<application-desc main-class="Lanceur">
  <argument>-lang</argument>
  <argument>fr</argument>
  <argument>-a</argument>
  <argument>http://xlogo.tuxfamily.org/fr/html/examples-fr/3d/de.lgo</argument>
```

```
</application-desc>
</jnlp>
```

This file is written in format XML. The most important part are these four lines:

```
<argument>-lang</argument>
<argument>fr</argument>
<argument>-a</argument>
<argument>http://xlogo.tuxfamily.org/fr/html/examples-fr/3d/de.lgo</argument>
```

These lines specify the parameters for XLogo on startup

- Line 1 and line 2 force the language to be french.
- The last line indicates the file address to load.
- Line 3 indicates that the main command from this file will be executed on XLogo start up.

A last hint: Because Tuxfamily's server can't accept all connections, it's better to put the file `xlogo.jar` on your site. To link this file with the `.jnlp` file, you just have to modify the address on line 2 after `codebase=`

Appendix D

Solutions

D.1 Chapitre 5

```
to square
repeat 4[forward 150  right 90]
end

to tri
repeat 3[forward 150 right 120]
end

to door
repeat 2[forward 70 right 90 forward 50 right 90]
end

to chimney
forward 55 right 90 forward 20 right 90 forward 20
end

to move1
right 90 forward 50  left 90
end

to move2
left 90 forward 50 right 90 forward 150 right 30
end

to move3
penup right 60 forward 20 left 90 forward 35 pendown
end

to house
square move1 door move2 tri move3 chimney
end
```

D.2 Chapter 6

```
to supercube
clearscreen penup setpos[ -30 150] pendown setpos[-150 150]  setpos[-90 210] setpos[30 210] setpos
setpos[-30 -210] setpos[30 -150] setpos[30 -90] setpos[-30 -90] setpos[90 -90] setpos[90 30]
```

```
setpos[-270 30] setpos[-270 -90] setpos[-210 -90] setpos[-210 -30] setpos[-90 -30] setpos[-90 -150
setpos[-210 -150] setpos[-210 -30] setpos[-150 30] setpos[-30 30] setpos[-90 -30] setpos[90 150]
setpos[30 150] setpos[30 210] setpos[30 90] setpos[90 90] setpos[90 150] setpos[90 90] setpos[150
setpos[150 -30] setpos[90 -90] setpos[90 30] setpos[150 90] penup setpos[-150 30] pendown setpos[-
setpos[-150 90] setpos[-210 90] setpos[-270 30] penup  setpos[-90 -150] pendown setpos[-30 -90]
penup setpos[-150 -150] pendown setpos[-150 -210] setpos[-30 -210]
end
```

D.3 Chapter 7

D.3.1 the robot

The first drawing is formed only with elementar motifs: rectangulars, squares and triangle. Here is the associated code:

```
to rec :lo :la
# draws a rectangular with choosen dimansion
repeat 2[forward :lo right 90 forward :la right 90]
end

to square :c
# Draws a square with side length :c
repeat 4[forward :c right 90]
end

to tri :c
# Draws an equilateral triangle with side length :c
repeat 3[forward :c right 120]
end

to leg :c
rec 20*:c 30*:c square 20*:c
end

to antenna :c
forward 30*:c left 90 forward 10*:c right 90 square 20*:c
penup back 30 *:c right 90 forward 10*:c left 90 pendown
end

to robot :c
clearscreen hideturtle
# body
rec 40*:c 280* :c
# legs
right 90 forward 20*:c leg :c forward 40* :c leg :c forward 140*:c leg :c forward 40*:c leg :c
# queue
penup left 90 forward 40* :c pendown right 45 forward 110*:c back 110* :c left 135
# head
forward 180 *:c square 10*:c forward 30*:c square 10*:c right 90 forward 10*:c left 90
 forward 20*:c right 90 square 80* :c
# ears
forward 40*:c left 60 tri 30*:c penup right 150 forward 80 *:c left 90 pendown tri 30*:c
#  antennas
forward 40 *:c left 90 forward 20*:c right 90 antenna :c left 90 forward 40*:c right 90 antenna :c
```

```
# eyes
penup back 30 *:c pendown square 10*:c right 90 penup forward 30*:c pendown left 90 square 10*:c
# mouth
penup back 30*:c left 90 forward 30*:c right 90 pendown rec 10*:c 40*:c
end
```

D.3.2 The frog

```
to frog :c
clearscreen hideturtle
forward 20 *:c right 90 forward 50*:c left 90 forward 40*:c
left 90 forward 70 *:c right 90 forward 70*:c right 90
forward 210 *:c right 90 forward 20*:c left 90 forward 20*:c
right 90 forward 90*:c right 90 forward 20*:c left 90
forward 20*:c right 90 forward 90*:c right 90 forward 20*:c
right 90 forward 70*:c back 50*:c left 90 forward 40*:c
right 90 forward 40*:c back 40*:c left 90 back 20*:c left 90
forward 50*:c left 90 forward 40*:c right 90 forward 70*:c
right 90 penup forward 90*:c pendown repeat 4[forward 20*:c right 90]
end
```

D.4 Chapter 9:

```
to game
# Initailaize number and counter
globalmake "number random 32
globalmake "counter 0
loop
end

to loop
read [choose a number] "try
if numberp :try[
  # if the value is a number
  if :number=:try[print sentence sentence [You win in ] :counter+1 [tries]][
    if :try>:number [print [Lesser]][print [Greater]]
    globalmake "counter :counter+1
    loop
  ]
]
[print [Enter a Valid number!] loop]
end
```

Appendix E

FAQ - Tricks Things to know

E.1 Though I erase a procedure from the editor, it keeps on popping back!

When you leave the editor, it just saves or updates whatever the editor contains. The only way to erase a procedure in XLogo is to use the primitive `eraseprocedure` or `erp`.
Exemple: `erp "toto` ⟶ erases the procedure `toto`.

E.2 I'm using the version in Esperanto but I can't write with the special characters!

When you type in the command line or the editor, if you click with the right button, a rolling screen appears. In this menu, you can find the traditional editing functions (cut, copy, paste) and the esperanto special characters when this language is selected.

E.3 In the Sound tab from the Preferences dialogue box, no instrument can be found.

Sometimes, the instruments list doesn't appear in `Tools/Preferences/Sound`. Go here:

`http://java.sun.com/products/java-media/sound/soundbanks.html`

Download on eof the sound banks: minimal, midsize ou deluxe and uncompress it in `C:\Program Files\Java\jre1.6.`

- The folder `jre1.6.0_05` corresponds to your installed JRE version.
- If the folder `audio` doesn't exist, create it.
- You have to rename the uncompressed file as: `soundbank.gm`

Then, restart XLogo and have a look at `Tools/Preferences/Sound`

E.4 How to quickly retype a command used previously?

- First method: with the mouse, click on the line in the history area, it will reappear immediately on the control line.
- Second method: with the keyboard, the Up and Down arrows allow navigation of the list of previous commands that have been typed, (very practical).

E.5 How can you help?

- By reporting any observed bug. If you are able to reproduce systematically an observed problem, it is even better.
- Any suggestion to improve the program is welcome.
- By helping to translate.
- A little moral support is always welcome!

Thanks

- I want to thank all my active translators for their work in XLogo.
  - English: Guy Walker
  - Spanish: Marcelo Duschkin, Alvaro Valdes Menendez
  - Arabic: El Houcine Jarad
  - Portuguese: Alexandre Soares
  - German: Michael Malien
  - Esperanto: Michel Gaillard
  - Galician: Justo freire
  - Greek: Anastasios Drakopoulos
- I want to thank Eitan Gurari for his patience, and for his great LaTeX extension called `tex4ht` which allow to convert my manuals into different formats

  `www.cse.ohio-state.edu/~gurari/TeX4ht`
- Several OpenSource projects very important for XLogo:
  - Java3D: `https://java3d.dev.java.net/`
  - JavaHelp: `http://java.sun.com/javase/technologies/desktop/javahelp/`
  - Eclipse: `www.eclipse.org/`
- Finally, a big THANKS to Tuxfamily for the quality of their website hosting and their important contribution in free software!

  `http://www.tuxfamily.org`

Index