XLOGO User Manual

Author: Le Coq Loïc

Translated by:

Walker Guy
Donnelly Kevin
Roch Etienne

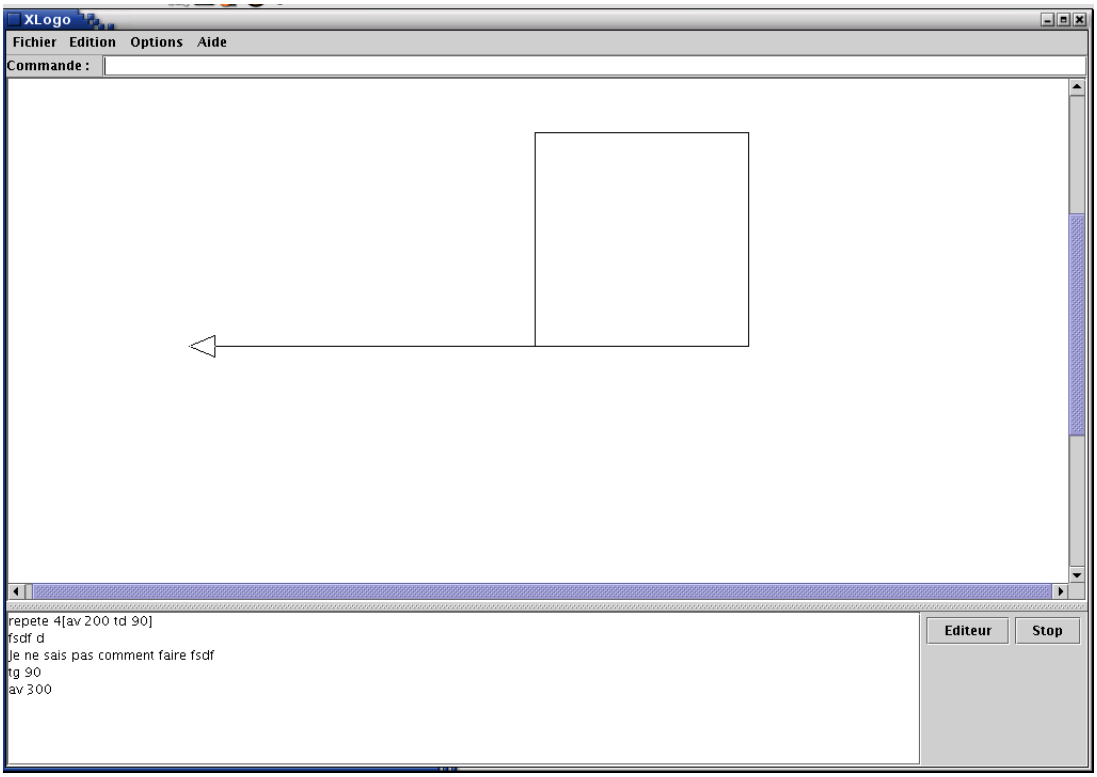
April 19, 2007

Contents

Chapter 1

Interface features:

1.1 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.
- 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. STOP interrupts the execution of the program and EDITOR allows the procedure editor to be opened.

1.2 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.

Quit 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

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`.

Chapter 2

Menu options:

2.1 "File" Menu

- File –> New: Delete all procedures and variables. You create a new workspace.
- File –> Open: open a previously saved logo file.
- File –> Save: save the procedures in the current file.
- File –> Save as ...: save the current procedures under a specific name.
- File –> Quit: quit the XLOGO application.
- File –> Capture image –> Save image as... : allow the image to be saved in the jpg or png format. If you wish to select only a part of the image, you can define a bounding box by cicking twice in succession to define the two corners of a diagonal through the bounding box.
- 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 prinitng and recording, you can select a zone of the image. This functionality works very well under the Window environments. On the other hand, it does not work under Linux (the clipboard have a different behaviour). Untested under Mac.

2.2 "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.

2.3 "Options" Menu

- Options –> Choose the pen colour: allows the colour with which the turtle will write to be chosen from a palette of colours. Also accessible via the command fcc
- Options –> Choose the background colour: set the colour of the screen background. Accessible via the primitive fcfg.

- Options –> Define start-up files: 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.
- Options –> Translate source...: Allows to translate source from a language to another. In fact, very usefull when you want to use for example downloaded Logo source written in another language.
- Options –> Preferences: opens a dialog box in which you can configure several things:
  - Language : allows French,Spanish or English to be chosen. Note that the primitives differ in each language.
  - Look: allows the "look" of the XLogo window to be defined. The Windows, native Java and Motif styles are available.
  - Choose the 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.
  - On the second tab, you can choose your preferred turtle.
  - On the third tab, many options:
    * You can choose the maximal pen width allowed. If you don't want to use this option, put -1.
    * You can choose the shape of the pen: round or square.
    * You can choose the maximal number of turtle in mode multiturtle.
    * You can choose if you want to clear screen when you leave the editor.
    * You can choose a personal size for the drawing zone. Otherwise XLogo opens in 1000 on 1000 pixels zone. Be careful, when you increase the size of the image, you might have to increase the memory size of XLogo. An error message will pop up.
    * Consequently, you can also change the corresponding memory space allocated to XLogo. Otherwise it will be a 64 Mo size. You might have to increase it if you want to work on a bigger drawing zone. When 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.
    * Finally, you can choose the accuracy of the drawing line. In high quality definition, you will especially not have the square effect . Yet do not forget that by increasing the quality you will lose some execution speed.
  - On the fourth tab, you can choose an instrument for your MIDI interface. Some problem of detection can appear, sorry... This function could be accessed with the primitive `setinstrument`.
  - On the fifth tab, You can choose the font for the interface.

2.4 "Help" Menu

- Menu –> Licence: shows the GPL license under which this software is distributed.

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

Chapter 3

Conventions adopted by XLOGO

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

3.1 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` returns `hello`. Note that if you forget the ", the interpreter will return an error message. In effect, `print` expects the 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.

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

3.2 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 small 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 twice :word
print :word
print :word
end

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

See the various examples of procedures at the end of the manual.

3.3 Specific character \

The specific character \ (backlash) especially 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 \\.

Same 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 backclash, the path that will be defined will then be `c:My Documents` and the interpretor will send you an error message.

3.4 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)
```

3.5 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 `somme` which expects two arguments: `somme 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)	&	=	

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

3.6 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]
```

Chapter 4

List of primitives

As noted above, the turtle is controlled by means of internal commands called 'primitives' . The following sections set out these primitives:

4.1 Movement of the turtle; pen and colour settings

This first table sets out the primitives which govern the movement of the turtle.

Primitives	Arguments	Utilisation
`forward`, `fd`	n : number of steps	Moves the turtle forward n steps in the direction it is currently facing.
`back`, `bk`	n: number of steps	Moves the turtle backwards n steps in the direction it is currently facing.
`right`, `rt`	n: angle	Turns the turtle n degrees towards the right in relation to the direction it is currently facing.
`left`, `lt`	n: angle	Turns the turtle n degrees towards the right in relation to the direction it is currently facing.
`circle`	R: number	Draws a circle of R radius around the turtle.
`arc`	R cap1 cap2: numbers	Draws an arc of R radius around the turtle. This arc is inscribed between the caps cap1 and cap2.
`home`	any	Returns the turtle to its initial position, that is, the co-ordinates [0 0] with a heading of 0 degrees.
`setpos`	[x y]: list of two nunbers	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: x-axis	Moves the turtle horizontally to the point x on the x-axis
`sety`	y: y-axis	Moves the turtle vertically to the point y on the y-axis
`setxy`	x y: x-co-ordinate followed by y-co-ordinate	Identical to setpos [x y]

Primitives	Arguments	Utilisation
`setheading`	n: heading or bearing	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`	a: word or list	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.
`labellength`	a: word or list	Returns the length that needs the word or the list to be displayed on the screen with the primitive `label` using the current font.
`dot`	a: list	The point defined by the co-ordinates in the list will be highlighted (in the pen colour).

This second table 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?

Primitives	Arguments	Utilisation
`showturtle`, `st`	none	Makes the turtle visible on the screen.
`hideturtle`, `ht`	none	Makes the turtle invisible on the screen.
`clearscreen`, `cs`	none	Empties the drawing area.
`wash`	none	Erases the drawing area but leaves the turtle in the same place.
`pendown`, `pd`	none	The turtle will draw a line when it moves.
`penup`, `pu`	none	The turtle will not draw a line when it moves.
`penerase`, `pe`	none	The turtle will rub out any marks that it meets.
`penreverse`, `px`	none	Lower the pen and put the turtle in inverted mode.
`penpaint`, `ppt`	none	Lower the pen and put the turtle in classic drawing mode.
`animation`	true or false	• if the argument is `true`: 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 `update`. It is very useful to create an animation or to draw a line faster. • If the argument is `false`: you switch back to classical mode. You can see the turtle's moves on screen.

Primitives	Arguments	Utilisation
`repaint`	none	In animation mode, updates the screen: the image on the drawing area is updated.
`setpencolor`, `setpc`	a: whole number or list [r g b]	Sets the pen color. See p.11.
`setscreencolor`, `setsc`	a: whole number or list [r g b]	Sets the screen color. See p.11.
`pos`	none	Gives the current position of the turtle.Eg: `pos` returns [10 -100]
`heading`	none	Gives the bearing or heading of the turtle (cf `setheading`)
`towards`	a: 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`	a: 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`	a: list	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`	a: list	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`	none	The turtle can travel outside the drawing area (but of course, it cannot draw there).
`wrap`	none	If the turtle leaves the drawing area, it will reappear on the opposite side!
`close`	none	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 ...).
`findcolor,fc`	a: list	Returns to the colour of the `a` coordinates pixel. This color is determined thanks to a [r g b] list where r is red, g is green and b is blue.
`setpenwidth`, `setpw`	n: number	Defines the thickness of the pen nib in pixels. The default is 1. The pen has a square nib. (Other shapes will be provided in future versions.)
`setshape`	n: number	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).

Primitives	Arguments	Utilisation
`shape`	none	Returns the number that represents the shape of the turtle.
`setfontsize`, `setfs`	n: number	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`	none	Returns the size of the font when you write on the screen with the primitive `label`.
`setfn`, `setfontname`	n: number	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 –> Options –> Preferences –> Tab Font.
`fontname`	none	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`	a: number	Determines the ratio between the graphic screen and the history zone. The number a must be included between 0 and 1. When a equals 1 the drawing zone uses all the space, when a equals 0, the history zone uses all the window.
`separation`, `sep`	none	Provides the current ratio between the drawing zone and the history zone.
`zonesize`	none	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`	a: list	Shows the message in list in a dialog box, the program stops until the user has clicked the button "OK"

4.2 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. Those primitive that control the color and the size of the history area, are available only for the primitives `print` or `write`

Primitives	Arguments	Utilisation
`ct, cleartext`	none	Empties the area containing the command and comment history.

`pr, print`	word, list or number	shows the argument specified in the history zone.
		
		`pr "abcd --------> abcd`
		`pr [1 2 3 4] ----> 1 2 3 4`
		`pr 4 ------------> 4`

`write`	word, list or number	The same as for the print primitive but doesn't go back to the line.
`sft, setfonttext`	a: number	Define the size of the font in the command history. Only valid with the primtive `print`
`ftext, fonttext`	none	Returns the size of the font with primitive `print`.
`sct, setcolortext`	a:number or list	Define the color of the font in command history. Valid only with the primitive . See p.11.
`ctext, colortext`	none	Returns the color of the font with the primitive `print` in the command history.
`setfnt, setfontnametext`	n: number	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.
`fnt, fontnametext`	none	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, ssty`	list or word	Set the format of the police 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. Look at examples after this table.
`style, sty`	none	Returns a list which contains the differents styles used for the primitive `print`.

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

4.3 Arithmetical and logical operations

Primitives	Parameters	Usage
`sum`	a b: numbers to add	Adds the two numbers a and b and returns the result Eg: `sum` 40 60 returns 100
`difference`	a b: numbers	Returns a-b. Eg: `difference` 100 20 returns 80
`minus`	a : number	Returns the negative of a. Eg: `minus` 5 returns -5. See the note at the end of this table.
`product`	a b : numbers	Returns the result of multiplying a by b.
`div, divise`	a b: numbers	Returns the result of dividing a by b `div` 3 6 returns 0.5
`quotient`	a b: numbers	Returns quotient a by b `quotient` 15 6 returns 2
`remainder`	a b: whole numbers	Returns the remainder after dividing a by b.
`round, rnd`	a: number	Returns the nearest whole number to the number a. `round` 6.4 returns 6
`integer`	a: number	returns the integer part of the number. `integer 8.9` returns 8 `integer 6.8` returns 6
`power`	a b: numbers	Returns a raised to the power of b. `power` 3 2 returns 9
`sqrt , sqrt`	n : number	returns the square root.
`log10`	n : number	Returns the decimal logarithm of n
`sin, sine`	a: number	Returns the sine of a. (a is expressed in degrees)
`cos, cosine`	a: number	Returns the cosine of a. (a is expressed in degrees)
`tan, tangent`	a: number	Returns the tangent of a. (a is expressed in degrees)
`acos, arccosine`	a: number	Returns the angle in range [0-180] which cosine is a.
`asin, arcsine`	a: number	Returns the angle which sine is a.
`atan, arctangent`	a: number	Returns the angle which tangent is a.
`pi`	aucun	Returns the number π (3.141592653589793)
`random, ran`	n: whole number	Returns a random number between 0 and $n-1$.
`absolute, abs`	n: nombre	Returns the absolute value (its numerical value without regard to its sign) of a number.

Important : Be careful with those primitives which require two parameters!

Eg:

`setxy` a b If b is negative

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:

Primitives	Parameters	Usage
`or`	b: booleans	Returns true if a or b is true, otherwise returns false
`and`	b: booleans	Returns true if a and b is true, otherwise returns false
`not`	a :boolean	Returns the negation of a. If a is true, returns false. If a is false, returns true.

4.4 Operations on lists

Primitives	Parameters	Usage
`word`	a b	Concatenates the two words a and b. Eg: `pr word` "a 1 returns a1
`list`	a b	Returns a list composed of a and b. For example, `list` 3 6 returns [3 6]. `list` "a "list returns [a list]
`sentence, se`	a b	Returns a list composed of a and b. If a or b is a list, then each element of a and b 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`	a b: a anything, b list	Insert a in the first slot in list b. Eg : `fput` "cocoa [2] returns [cocoa 2]
`lput`	a b: a anything, b list	Insert a in the last slot of list b Eg: `lput` 5 [7 9 5] returns [7 9 5 5]
`reverse`	a : list	Reverse the order of elements in list a `reverse` [1 2 3] returns [3 2 1]
`pick`	a : a word or list	If a is a word, returns one of the letters of a at random. If a is a list, returns one of the elements of a at random.
`remove`	a b: a anything, b list	Remove element a from list b if it occurs there. Eg: `remove` 2 [1 2 3 4 2 6] returns [1 3 4 6]
`item`	a b: a whole number, b list or word	If b is a word, returns the letter numbered a from the word (1 represents the first letter). If b is a list, returns the element numbered a from the list.
`butlast,bl`	a: list or word	If a is a list, returns the whole list except for its last element. If a is a word, returns the word minus its last letter.
`butfirst, bf`	a: list or word	If a is a list, returns the whole list except for its first element. If a is a word, returns the word minus its first letter.
`last`	a: list or word	If a is a list, returns the last element of the list. If a is a word, returns the last letter of the word.
`first`	a: list or word	If a is a list, returns the first element of the list. If a is a word, returns the first letter of the word.
`setitem, replace`	li1 n li2: li1 list, n integer, li2 word or list	Replace the element number `n` in the list `li`, by the word or the list `li2`. `replace [a b c] 2 8 --> [a 8 c]`
`additem`	li1 n li2: li1 list, n integer, li2 word or list	Adds at the position `n` in the list `li` the word or the list `li2` `additem [a b c] 2 8 --> [a 8 b c]`

`count`	a: list or word	If a is a word, returns the number of letters in a. If a is a list, returns the number of elements in a.
`unicode`	a: word	returns the Unicode value of the character `"a`. `pr unicode "A` returns 65
`character`, `char`	a: number	returns the character which Unicode value is `"a`. `pr character 65` returns `"A`

4.5 Booleans

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

Primitives	Parameters	Usage
`true`	none	Returns ”true.
`false`	none	Returns ”false.
`word?`	a	Returns true if a is a word, false otherwise.
`number?`	a	Returns true if a is a number, false otherwise.
`integer?`	a	returns true if a is a whole number, false otherwise.
`list?`	a	Returns true if a is a list, false otherwise.
`empty?`	a	Returns true if a is an empty word or an empty list, false otherwise.
`equal?`	a b	Returns true if a and b are equal, false otherwise.
`before?`	a b : words	Returns true if a is before b in terms of alphabetical order, false otherwise.
`member?`	a b	If b is a list, specifies if a is an element of b. If b is a word, specifies if a is a letter in b.
`member`	a b	If b is a list, look for the element a in this list. There are two possible outcomes: -If a is in b, returns a sublist containing all list elements from the first instance of a in b. -If a is not in b, returns the word false. If b is a word, look for the character a in this word. There are two possibilities: - If a is in b, return the latter part of the word, starting from a. -Otherwise, return the word false. `member` “o “cocoa return ocoa `member` 3 [1 2 3 4] returns [3 4]
`pd?, pendown?`	anything	Returns the word true is the pen is down, false otherwise.
`visible?`	anything	Returns the word true if the turtle is visible, false otherwise.
`prim?, primitive?`	a: word	Returns true if the word is an XLOGO primitive, false otherwise.
`proc?, procedure?`	a: word	Returns true if the word is a procedure defined by the user, false otherwise.

4.6 Testing an expression with the primitive `if`

As in all programming language, 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. Here is the syntax:

```
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]`

4.7 Dealing with procedures and variables

4.7.1 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 ....
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).
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
```

IMPORTANT: There must be no comments on the `to` or `end` lines.

4.7.2 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.

4.7.3 Primitive 'trace'

It is possible so as to follow the working of a program to have it show the procedures which are working. This mode equally allows to show if the procedures provide arguments thanks to the primitive `output`. To operate this mode, you type:

```
trace true
```

Of course, trace false will disactivate the `"trace"` mode. A small example with the factorial (see page 44).

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

4.7.4 Other primitives

Primitives	Arguments	Usage
`make`	a b: a word, b anything	If the local variable a exists, assigns it the value b. If not, creates a global variable a and assigns it the value b. Eg: `make` "a 100 assigns the value 100 to the variable a
`local`	a: word	Creates a variable called a. Note, this is not initialised. To assign it a value, see `make`.
`localmake`	a b: a word, b anything	Creates a new local variable and assigns it the value b.

`def, define`	word1 list2 list3	Define a new procedure called word1, which requires the variables in list2. The procedure's instructions are contained in list3.
		`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.

`thing`	a: word	returns the value of the variable :a. `thing "a` is similar to `:a`
`er, erase`	a: word	Deletes the procedure calling a.
`kill`	a: word	Deletes the variable a.
`erall`	none	Deletes all the variables and procedures currently running.
`poall`	none	Enumerates all the procedures currently defined.
`run`	a :list	Executes the list of instructions contained in list a.
`listvariables, lvars`	none	Returns a list which contains all the defined variables.

4.8 File handling

Primitives	Arguments	Usage
`ls, listfiles`	none	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`	a: 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`
`setdir, setdirectory`	l: list	Specifies the current directory. The path must be absolute. The directory must be specified with a word.
`changedirectory, cd`	m: word	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.
`dir, directory`	aucun	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`	w: word l:list	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. The specified word gives a relative path starting from the current directory. This command will not work with an absolute path.
`saved`	w: word	`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. The specified word gives a relative path starting from the current directory. This command will not work with an absolute path.
`load`	w: word	Opens and reads the file w. For example, to delete all the defined procedures and load the file test.lgo, you would use: `efns load "test.lgo`. The specified word 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.

Primitives	Arguments	Usage
`listflow`	none	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 the 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 the 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
```

4.9 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

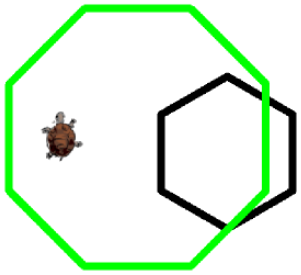


Figure 4.1: At the beginning

white pixels with the current pen colour. If for example, you type: `setpc 1 fill`. Let's now go back

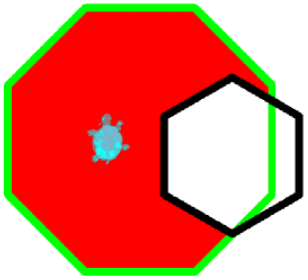


Figure 4.2: With the primitive `fill`

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
end
```

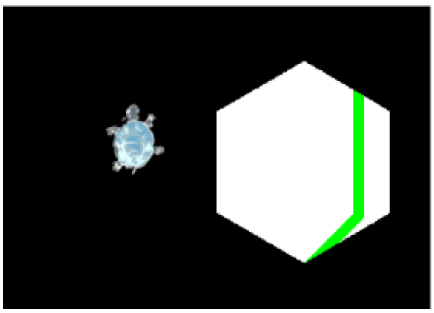


Figure 4.3: With the primitive `fillzone`, if you type: `setpc 0 fillzone`

```
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
```

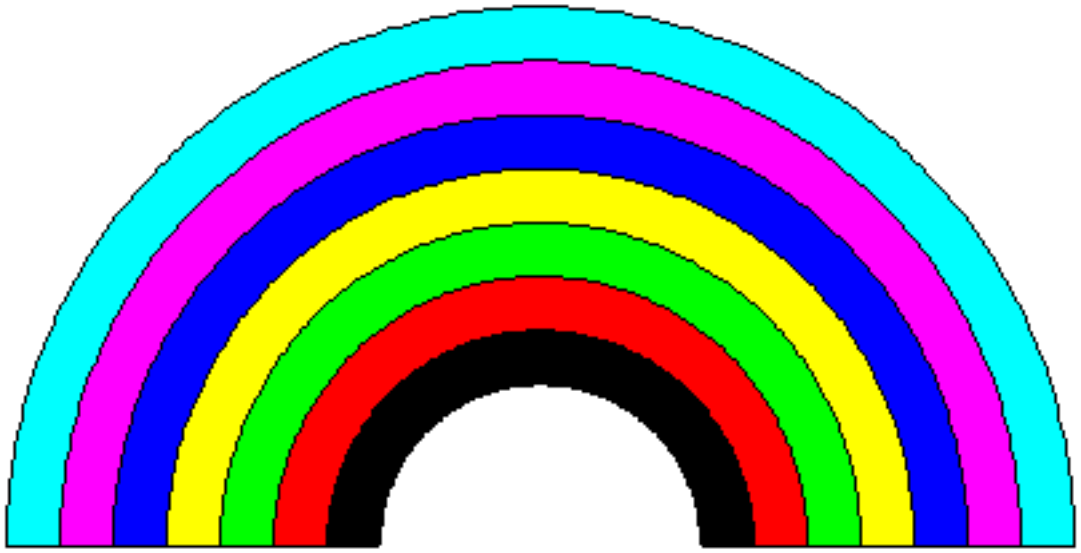


Figure 4.4: Arc-in-LOGO

4.10 Break commands

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

- `stop` can have two results. If it is included in a `repeat` or `while` loop, the program jumps out of the loop then and there. If it occurs in a procedure, the program breaks out of the procedure immediately.
- `stopall` the program breaks out of all the procedure immediately and stops.
- `output` allows breaking out of a procedure with a value to be returned.

See the numerous instances of usage of these two primitives in the examples at the end of this manual.

4.11 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 the 0. If you want to "create" a new turtle, you can use the primitive `setturtle` followed by the number of the turtle. To prevent from 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 to all classic primitives while you don't change active turtle with `setturtle`. The maximum number of available turtle can be set in menu Options - Preferences - Tab options.

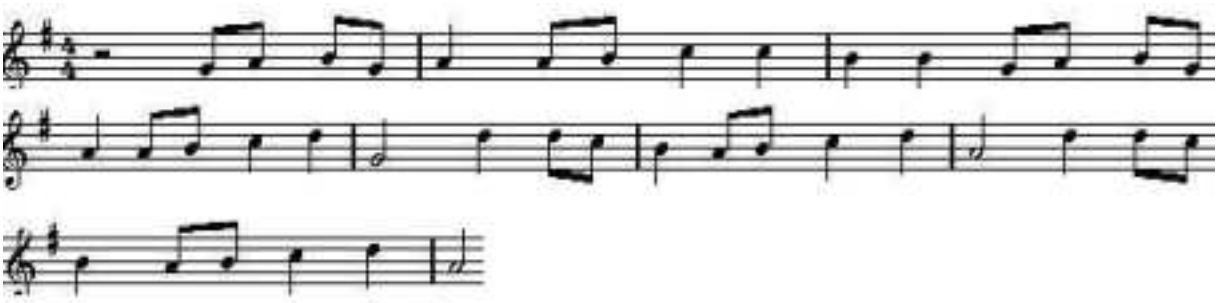
Here are the primitives for the multiturtle mode:

Primitives	Argumentss	Utilisation
`sturtle, setturtle`	a: number	The turtle numero a is now the active turtle. By default on Xlogo startup the active turtle is the numero 0.
`turtle`	none	Returns the numero of the active turtle.
`turtles`	none	Returns a list which contains all the numero af the turtles actually on the screen.
`killturtle`	a: number	Kill the turtle number a

4.12 Play music

Primitives	Arguments	Utilisation
`seq, sequence`	a: list	Put in memory the sequence in the list. Read after this table to learn how to write a sequence.
`play`	none	Play the sequence in memory.
`instr, instrument`	none	Returns the number that corresponds to the selected instrument.
`sinstr, setinstrument`	a: number	The selected instrument is now the instrument number a. You can see the list of all available instruments in menu Options-Preferences-Tab Sound.
`indseq, indexsequence`	none	Returns where the cursor is located in the current sequence.
`sindseq, setindexsequence`	a: number	Put the cursor to index a in the current sequence in memory.
`delseq, deletesequence`	none	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 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 (twice faster).
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 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 you 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.

4.13 Loops:

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

4.13.1 A loop with `repeat`

This is the syntax for `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!
```

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
```

4.13.2 A loop with `for`

`for` assign to a variable some successive values in a fixed range with a choosen increment.here is the syntax:

```
for list1 list2
```

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
```

4.13.3 A loop with `while`

This is the syntax for `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]
```

4.14 Receiving input from the user

4.14.1 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:

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 4.1: Values 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.

`read list_title word`: Presents a dialogue box whose title is `list_title`. The user can then input a response in a text field, and the response will be stored in the form of a list in the variable :word, and will be evaluated when the OK button is pressed.

4.14.2 Some examples of usage:

```
to vintage
read [What is your age?] "age
make "age first :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
```

4.14.3 Interact with the mouse

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

`posmouse`: 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.

4.14.4 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 posmouse]
example
end
```

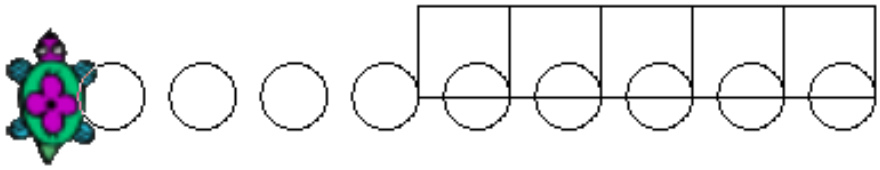
In this second procedure, it's the same but you must click with the left button of the mouse if you want the turtle to move.

```
to example2
if readmouse=1 [setpos posmouse]
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.

Carré

Cercle



```
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 posmouse
# we put the second coordinate of the mouse in variable y
make "y item 2 posmouse
# 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
```

4.15 Time and date

XLogo has several primitives for date, time or generating countdown.

Primitives	Arguments	Usage
`wait`	n: whole number	Halts the program, and therefore the turtle, for $\frac{n}{60}$ seconds.
`chrono`, `chronometre`	n: integer	Starts a countdown of n seconds. We know if this countdown has finished with the primitive `endcountdown?`
`endcountdown?`	none	Returns `"true` if there's no active countdown. Returns `"false` if the countdown is active.
`date`	none	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`	aucun	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`	none	Returns the past time in seconds since XLOGO has 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
```

4.16 Using a network with XLogo

4.16.1 The network Howto

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.

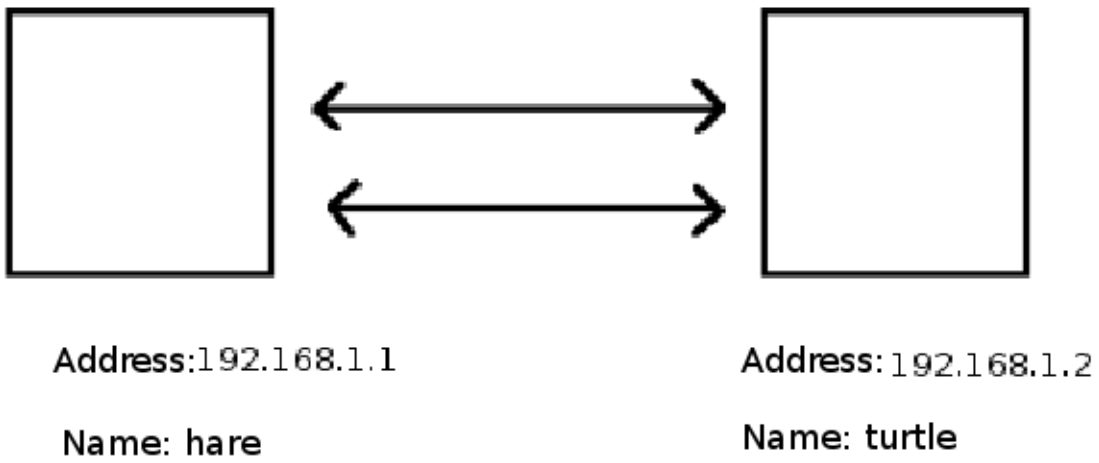


Figure 4.5: A simple network

Each computer 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.

4.16.2 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`: this primitive allows execution of instructions by a computer on the network. Syntax: `executetcp word list` $\longrightarrow$ The word is the called IP address or computer name, the list 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`: Allows chat between two computers on a network. On each computer, it displays a chat window.
  Syntax: `chattcp word list` $\longrightarrow$ The word is the called IP address or computer name, the list 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`: Send data towards a computer on the network and return his answer.

  Syntax: `sendtcp word list` $\longrightarrow$ The word is the called IP address or computer name, the list 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...)

Chapter 5

Program examples

5.1 Draw houses

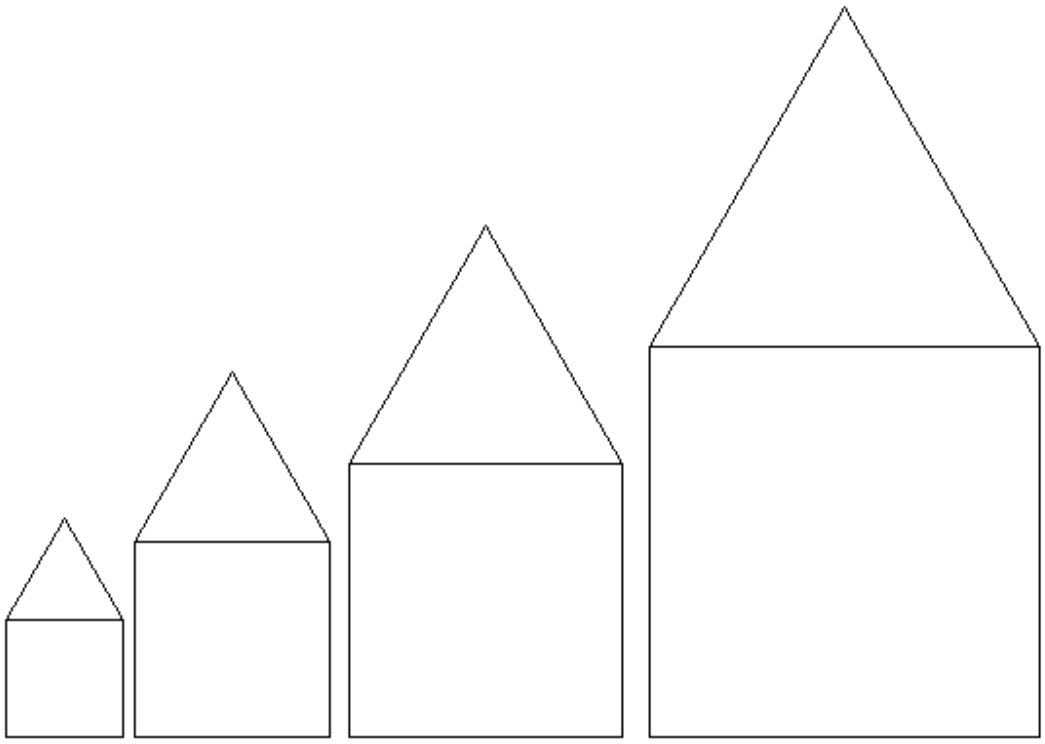


Figure 5.1: Houses

```
to house :c
repeat 4[fd 20*:c rt 90]
fd 20*:c rt 30
repeat 3[fd 20*:c rt 120]
end

to place :c
pu lt 30 bk :c*20 rt 90 fd :c*22 lt 90 pd
```

```
end

to hut
cs pu lt 90 fd 200 rt 90 pd ht
house 3 place 3 house 5 place 5 house 7 place 7 house 10
end
```

Figure 5.2: Rectangle

5.2 Draw a whole rectangle

```
to rect :lo :la
if :lo=0|:la=0 [stop]
repeat 2[fd :lo rt 90 fd :la rt 90]
rect :lo -1 :la -1
end
```

5.3 Factorial

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

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

por fac 5
120.0
pr fac 6
720.0
```

5.4 The snowflake (with thanks to Georges Noël)

```
to koch :order :len
if :order < 1 | :len <1 [forward :len stop]
koch :order-1 :len/3
lt 60
koch :order-1 :len/3
rt 120
koch :order-1 :len/3
lt 60
koch :order-1 :len/3
end
```

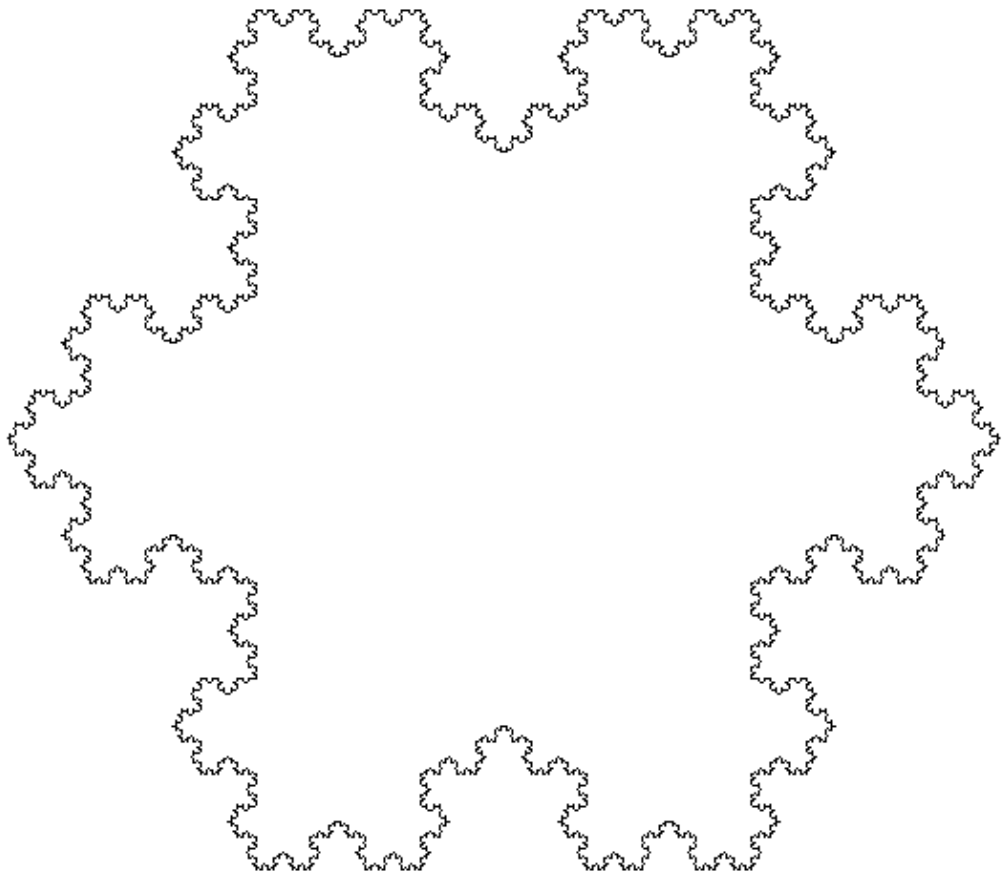


Figure 5.3: The snowflake

```
to kochflake :order :len
repeat 3 [rt 120  koch :order :len]
end

kochflake 5 450
```

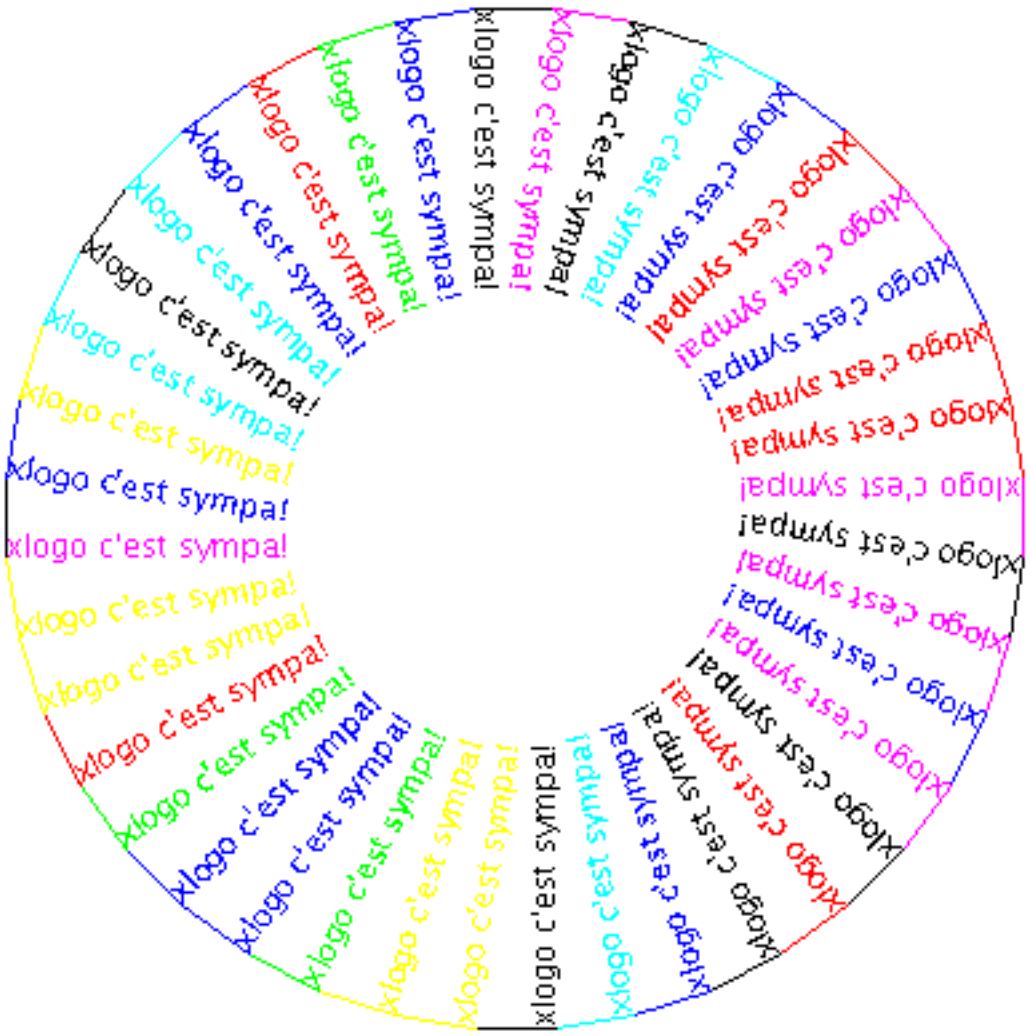


Figure 5.4: Xlogo c'est sympa!

5.5 A little bit of writing...

```
to write
ht repeat 40[fd 30 rt 9 setpc random 7 label [xlogo is cool!]]
end
```

5.6 And conjugation...

5.7 First version

```
to Fr_future :word
pr se "je word :word "ai
pr se "tu word :word "as
pr se "il word :word "a
pr se "nous word :word "ons
pr se "vous word :word "ez
pr se "elles word :word "ont
end

Fr_future "parler

je parlerai
tu parleras
il parlera
nous parlerons
vous parlerez
elles parleront
```

5.8 Second go

```
to fut :word
make "pronouns [je tu il nous vous elles]
make "endings [ai as a ons ez ont]
make "i 0
repeat 6[make "i :i+1 pr se item :i :pronouns word :word item :i :endings]
end

fut "parler

je parlerai
tu parleras
il parlera
nous parlerons
vous parlerez
elles parleront
```

5.9 Or even: A little recursion !

```
to fu :verb
make "pronouns [je tu il nous vous elles]
make "endings [ai as a ons ez ont]
conjugate :verb :pronouns :endings
end
```

```
to conjugate :verb :pronouns :endings
if empty? :pronouns [stop]
pr se first :pronouns word :verb first :endings
conjugate :verb bf :pronouns bf :endings
end

fu "parler

je parlerai
tu parleras
il parlera
nous parlerons
vous parlerez
elles parleront
```

5.10 All about colours

5.11 Introduction

Fist, a few explanations: you will note that the command `setpc` can take either a list or a number as a parameter. Here, we are interested in coding RGB values.
Every colour in XLOGO is coded using three values: red, green and blue, whence the name RGB encoding. The three numbers in the list parameter to the primitive `setpc` therefore represent respectively the red, blue and green components of a colour. This encoding is not really intuitive, and you can get an idea of the colour which will be given by the encoding by using the dialogue box Options—-> Choose the pen colour.
Using this encoding, it is very easy to transform an image. For example, if you want to turn a colour photo into a greyscale image, you can change the colour of each pixel of the image to the average value of the three components [r g b]. Imagine that the colour of one dot of the image is given by the encoding [0 100 80]. You calculate the average of these three numbers: $\frac{0+100+80}{3} = 60$, and then assign the colour [60 60 60] to this pixel. This operation has to be carried out on each pixel of the image.

5.12 Let's get practical!

We are going to transform a 100 x 100-pixel image to a greyscale. This means that there are therefore$100 \times 100 = 10000$ pixels to modify. You can access the image used in this example at the following address:
http://xlogo.tuxfamily.org/images/transfo.png
This is how we are going to proceed: first, we will refer to the top left corner of the image as [0 0]. Then the turtle will examine the first 100 pixels of the first line, followed by the first 100 of the second line, and so on. Each time, the colour of the pixel will be retrieved with `findcolor`, and the colour will then be changed to the average of the three [r g b] values. Here is the relevant code:
(Don't forget to change the filepath in the transform procedure!)

```
to pixel :list
# return the average of three numbers [r g b]

make "r first :list
make "list bf :list
```

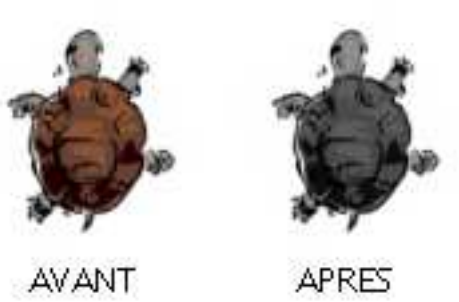


Figure 5.5: XLOGO fait de la retouche d'images....

```
make "g first :list
make "list bf :list
make "b first :list
make "b round (:r+:g+:b)/3
output se list :b :b :b
end

to greyscale :c
if :y=-100 [stop]
if :c=100 [make "c 0 make "y  :y-1]
# We assign the "average" colour of the following pixel to the pen
setpc pixel fc liste :c :y
# We turn the dot to greyscale
dot list :c :y
greyscale :c+1
end

to transform
# You must change the path to the image transfo.png
# Eg: setdir "c:\\my_images loadimage "transfo.png]

cs ht setdir "/home/loic loadimage "transfo.png make "y 0 greyscale 0
end
```

5.13 And if you want a negative??

To change an image to a negative, you can use the same process, except that instead of averaging the numbers r g b, you replace them by the number you get when you subtract them from 255. Eg: If a pixel has the colour [2 100 200], you replace it with the colour [253 155 55]

Only the pixel procedure needs to be changed in the following program:

```
to greyscale :c
if :y=-100 [stop]
if :c=100 [make "c 0 make "y  :y-1]
setpc pixel fc list :c :y
```

Figure 5.6: XLOGO pretending to be the GIMP ...(pretentious? :-))

```
dot list :c :y
greyscale :c+1
end

to transform
# You must change the path to the image transfo.png
# Eg: setdir "c:\\my_images loadimage "transfo.png]
ht cs setdir "/home/loic loadimage "transfo.png make "y 0 greyscale 0
end

to pixel :list
make "r first :list
make "list bf :list
make "g first :list
make "liste bf :list
make "b first :list
output se list 255-:r 255-:g 255-:b
end
```

5.14 A good example of using lists (with thanks to Olivier SC)

I hope you will appreciate this wonderful program:

```
to reversew :w
if empty? :w [output "]
output word last :w reversew bl :w
end

to palindrome :w
if equal? :w reversew :w [output "true] [output "false]
end
```

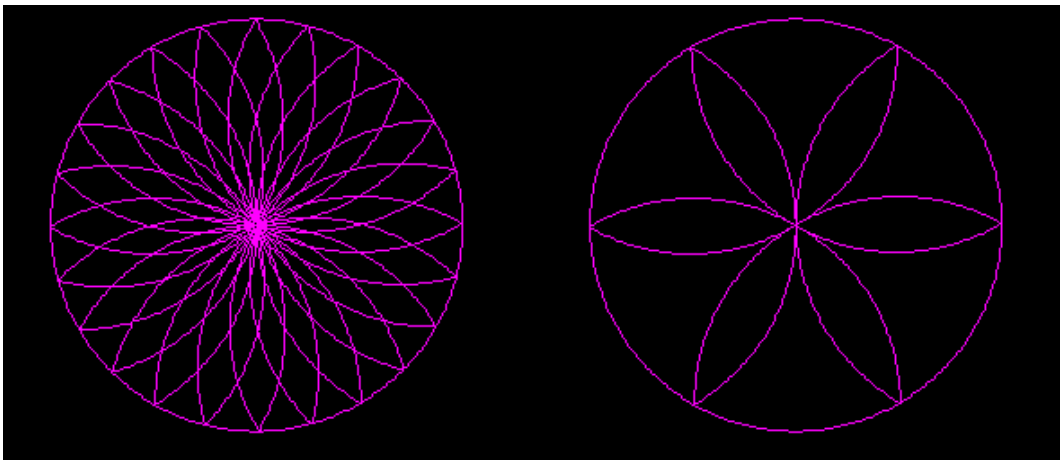


Figure 5.7: Better than using a compass!

```
to palin :n
if palindrome :n [print :n stop]
print se se se se :n "more reversew :n "equal sum :n reversew :n
palin :n + reversew :n
end

palin 78
78 more 87 equal 165
165 more 561 equal 726
726 more 627 equal 1353
1353 more 3531 equal 4884
4884
```

5.15 A pretty rosette

```
to rosette
repeat 6[ repeat 60[fd 2 rt 1] rt 60 repeat 120 [fd 2 rt 1] rt 60]
end

to pretty_rosette
rosette
repeat 30[fd 2 rt 1]
rosette
repeat 15[fd 2 rt 1]
rosette
repeat 30[fd 2 rt 1]
rosette
end

setsc 0 cs setpc 5 ht rosette pu setpos [-300 0] pd setheading 0 pretty_rosette
```

Chapter 6

Uninstall and bookmark

6.1 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/your_login) for Linux users, or c:\windows\.xlogo for Windows users).

6.2 Bookmark

For the latest version and bug-fixes, visit the XLogo site now and again - http://xlogo.tuxfamily.org. Feel free to contact the author is you have a problem with installation or use. All suggestions are welcome.

Chapter 7

FAQ - Tricks Things to know

7.1 Though I erase a procedure from the editor, it keeps on popping back!

When you go out of the editor, it just saves or updates whet the editor contains. The only way to erase a procedure in XLogo is to use the primitive `erase` or `er`.
Exemple: `erase "toto` ⟶ erase the procedure `toto`.

7.2 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.

7.3 In the Sound tab from the Preferences dialogue box, no instrument can be found.

Sorry, this problem has already been spotted. It is because of the java virtual machine. This problem is totally random. For example, in my case, I have a computer which works with Linux and Win 98. With Win 98, the list doesn't appear and with Linux, it does!! And I'm using the same JRE and have no material problem. This may change from a JRE version to another.

7.4 I have screen updating problems when the turtle is drawing.

This is also a known problem of the JRE. I will try to deal with it in the future, I might be able to do something about it. For now, there are two solutions:

- to minimize the window and increase its size again.
- To use the JRE 1.4.1_07 proposed on the website. With JRE $>$ 1.5, it seems better.

7.5 How to type quickly a control used previously?

- First method: with the mouse, click on the line in the historic area, it will reappear immediately on the control line.

- Second method: with the keyboard, the UP and Down arrows allow to navigate through the list of the last controls that have been typed. (very practical).

7.6 How can you be helped?

- 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: in English especially...
- A little moral support is always welcome!

Index