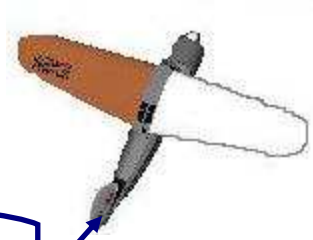
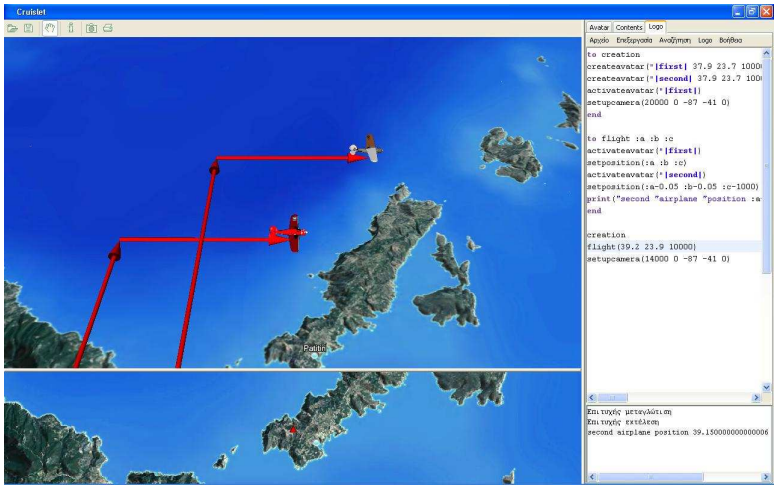
Educational Technology Lab

University of Athens



Cruislet Logo



Efi Alexopoulou

Introduction

Cruislet Logo is based upon Turtletracks language created by Daniel Azuma. You can find more about Turtletracks in http://turtletracks.sourceforge.net/.

The structure of this document, that describes Logo commands includes:

- Logo syntax of commands and functions.
- Illustrative comments about the way each command or function operates.
- Examples of use, where the Logo code is marked with grey shading and the output of code's execution is placed on a table underneath it.

Note: To execute Logo commands in Cruislet environment, use Insert in the keyboard.

Symbols in Logo

There are two symbols / characters used in Logo: quotation marks «"» and colon «:».
Quotation marks character declares that the following symbol must be considered as a symbol and there is no computation on this, implemented by Logo. For example, if we want to write a word, we must start with quotation marks followed by the word and then leave a white space. If we want to use more than one word or string separated by white spaces, we must use the vertical bar (|), in order to determine the boundaries.

```
print("Cruislet)
print("|Hello world|)
```

- *Cruislet*
- *Hello world*

Colon character is used to access a variable's value (see make and localmake commands). In the following example the colon character allow us to access variable's values, that their names are the contents of other variables.

```
make "y 10
make "x "y
make "z "x
print(:z)
print(::z)
print(:::z)
```

- *x*
- *y*
- *10*

In order to insert comments in the Logo Editor, we must use the semicolon character «;» before editing our comments.

Navigation Commands

In Cruislet environment, the user can navigate in the 3D geographical space, by using an object. The object (called an avatar), can be either an airplane plane or a helicopter.

Create / Delete

The user must firstly create the avatar.

createavatar(lat long height model-name)

Model-name can be either "|Plane 1|, "|Plane 2|, or "|Helicopter|. Lat and long values depends on geographical boundaries of Greece.

```
createavatar(37.4 26.2 10000 "|Plane 1|)
```

If the user want to refer to this plane in other commands it's better to name it, using the following command to create the avatar.

createavatar (name lat long height model_name)

```
createavatar("|MyAirplane| 36 25 5000 "|Helicopter|)
```

Note: 'createavatar' shortcut is 'creav'.

removeavatar()

Removes the active avatar. If more than one avatar is used, we must name the avatar we would like to remove, by using the following command.

```
removeavatar("|MyAirplane|)
```

Note: 'removeavatar' shortcut is 'remav'.

Activate

In order to navigate in geographical space using an avatar, we must activate the avatar we created.

activateavatar(name)

```
activateavatar("|MyAirplane|)
```

Note: 'activateavatar' shortcut is 'actav'

Μετατόπιση

Navigation in 3D geographical space is possible using two different frames of reference:

1. Geographical coordinates

setposition(lat long height)

Sets the avatar in the position specified by the input, that is the latitude, the longitude and the height correspondingly.

```
setposition(37 25 2500)
```

Note: 'setposition' shortcut is 'setpos'

outputposition()
Returns the current position of the avatar in the form of a three-element list, where the first element is the latitude, the second element is the longitude and the third element is the height.
Note: 'outputposition' shortcut is 'oppos'

2. Spherical coordinates

setdirection(theta fi r)
The avatar turns according to theta and fi angles and moves r meters.

```
setdirection(45 90 5500)
```

Note: 'setdirection' shortcut is 'setdir'

outputdirection()

Returns the current direction of the avatar in the form of a three-element list, where the first element is the theta angle, the second element is the fi angle and the third element is the r distance (in meters).
Note: 'outputdirection' shortcut is 'opdir'

stopavatar()
Stops the active avatar. If more than one avatar is used, we must name the avatar we would like to stop, by using the following command.

```
stopavatar("|MyAirplane|)
```

Note: 'stopavatar' shortcut is 'stopav'.

The avatar leaves a trace as it moves. This trace is a single, selectable, three-dimensional object (a thin cylinder ending on an arrow). It is actually a 3d vector representation. If the we want to delete the trail that has been created by avatar's displacement, we must use the following command.

cleartrail
Note: 'claertrail' shortcut is 'cltr'.

Camera

setcamera(distance azim pospitch orientpitch orientyaw)
Sets camera's properties.

```
setupcamera(15000 0 -87 -41 0)
```

Note: 'setcamera' shortcut is 'cam'.
If we want to change only one of camera's properties, we must specify it by using one of the following.

camdist setcameradistance(distance)

setcameraazimuth(azim)
Shortcut: camazim

setcamerapospitch(pospitch)
Shortcut: camppitch

setcameraorientationpitch(orientpitch)
Shortcut: camopitch

setcameraorientationyaw(orientyaw)
Shortcut: camoyaw

Variable operations

Variables' names or constants are words (strings) that always start with quotation marks (").

constant <ConstantName> value
Declares a constant variable named ConstantName and initializes the value of the constant. Constants don't change their value during the execution of the program.

```
constant "c 300000000
print(c)
```

- *300000000*

localmake <VariableName> value
Declares a variable as local to the current procedure, and initializes it at the same time. The variable is valid within the current procedure and it is not valid outside the procedure or in other procedures. In the following example, variable a, (which is accessed as :a), gives null as a result outside the procedure.

```
to try ;defines the procedure
localmake "a 123
print(:a)
end
try ;execute the procedure
print(:a)
```

- *123*
- *null*

make <VariableName> value
Declares a variable as global. Variable's value remains constant in all the procedures of the program.

```
make "myVariable 12
print(:myVariable)
```

- *12*

Data Structures

Three types of Logo data structures exist: lists, words and arrays.
A *list* is an ordered collection of objects or other data structures. The length of a list can be changed in a dynamic way.
A Logo *word* is represented by a string of letters or numbers and always starts with quotation marks (").
An *array* is a data structure which length is fixed and is declared automatically with the construction of the array.

Lists and Words

count(list), count(word)
Returns the length of the given argument. If the argument is a word, returns the number of characters it contains. If the argument is a list, returns the number of elements it contains.

```
print(count(["a "b "c "d]))
print(count("Cruislet))
```

- *4*
- *8*

length(list), length(word)
Returns the length of the given argument. If the argument is a word, returns the number of characters it contains. If the argument is a list, returns the number of elements it contains.

```
print(length(["a "b "c "d]))
print(length("Cruislet))
```

- *4*
- *8*

butfirst(list), butfirst(word)
Returns the given argument, with the first argument of the list or word removed.

```
print(butfirst("Cruislet))
print(butfirst([1 2 3 5 6]))
```

- *ruislet*
- *[2, 3, 5, 6]*

butlast(list), butlast(word)
Returns the given argument, with the last argument of the list or word removed.

```
print(butlast("Cruislet))
print(butlast([1 2 3 5 6]))
```

- *Cruisle*
- *[1, 2, 3, 5]*

first(list), first(word)

Returns the first element of the given argument, if it is a list, or the first character, if it is a word.

```
print(first("Cruislet))
print(first([1 2 3 5 6]))
```

- *C*
- *1*

last(list), last(word)

Returns the last element of the given argument, if it is a list, or the last character, if it is a word.

```
print(last("Cruislet))
print(last([1 2 3 5 6]))
```

- *t*
- *6*

item(number list), item(number word)

Returns the element of the list or the character of the word that is defined by the number.

```
print(item(4 "Cruislet))
print(item(4 [1 2 3 5 6]))
```

- *i*
- *5*

islist(list ή word ή number)

Returns true if the argument is a list. In all other cases, returns false.

```
print(islist("Cruislet))
print(islist([1 2 3 5 6]))
```

- *false*
- *true*

isarray(expr)

Returns true if the argument is an array. In all other cases, returns false.

```
make "a array(4)
print(isarray(:a))
```

- *true*

fput(number list), fput(word list), fput(list list)

Creates a new list in which the first argument is the first element and the members of the second argument are the remaining elements.

```
make "lst ["a 1]
print(fput(12 :lst))
print(fput("b :lst))
print(fput([2 3] :lst))
```

- *[12, a, 1]*
- *[b, a, 1]*
- *[[2, 3], a, 1]*

lput(number list), lput(word list), lput(list list)

Creates a new list in which the first argument is the last element and the numbers of the second argument are the remaining elements.

```
make "lst ["a 1]
print(lput(12 :lst))
print(lput("b :lst))
print(lput([2 3] :lst))
```

- *[a, 1, 12]*
- *[a, 1, b]*
- *[a, 1, [2, 3]]*

itemput(number wordOrlist1 wordOrlist2)

Returns the list wordOrlist1, having replaced the element that is defined by the number, with wordOrlist2.

```
print(itemput(2 "LO "OGO))
```

- *LOGO*

```
print(itemput(3 [1 2 3 4] [3 4 5]))
```

- *[1, 2, [3, 4, 5], 4]*

Arrays

An array in Cruislet Logo is one-dimensional. The length of the array (in cells) is defined when the array is constructed. In order to use and manipulate arrays, we use the following procedures.

array(number)

Creates a one-dimensional array and at the same time defines the number of the cells. The following command creates an array named "b" with 10 cells.

```
make "b array(10)
```

arraycount(array)

Returns the number of the array's cells.

```
print(arraycount(:b))
```

- *10*

arrayput(array number object)

Sets the value 'object' in place 'number' of the array. In the following example the array "b" gets the numbers 1 to 10. The cells of the array are filled from zero to N-1 where N is the number of the cells.

```
repeat 10 [
make "b arrayput(:b repcount-1 repcount-1)]
```

arrayget(array number)
Returns the element of the array that is placed in position defined by 'number'. The following example prints the contents of the cells of array "b".

```
make "i 0
while (:i<arraycount(:b)) [
type(arrayget(:b :i))
make "i :i+1
```

➢ *0123456789*

Flow Control

#include(filename)
With this command we can use files defined by the argument. In this way we can include library files.

```
#include "|file_name.lgo|
```

to <ProcedureName> <Param1> <Param2>

…

end
With 'to' and 'end' we define a chain of commands, that is a procedure. With the arguments following procedure's name we can use parameters in procedures.

```
to sum :a :b ;defines the procedure
print(:a+:b)
end
sum(3 4) ;executes the procedure
```

➢ *7*

if (condition) [Commands]
Evaluates the given condition. If the condition is true, the commands are executed, otherwise they are ignored. The condition must always be either true or false.

ifelse (condition) [Commands 1][Commands 2]
Evaluates the given condition. If the condition is true, 'Commands 1' are executed, otherwise 'Commands 2' are executed.

output(word ή list ή number)
Stops the execution of the procedure and returns the argument that could be a list or a word.

```
make "mes "|A hundred|
make "i 1
to count100
while (true) [
print(:i)
if (:i=100) [output(:mes)]
make "i :i+1
]
end
print(count100)
```

- ..
- 97
- 98
- 99
- 100
- A hundred

repeat

A loop structure where a list of commands are executed for a given number of times. It is edited as:
Repeat number [Commands]
Commands in the brackets are executed as many times as it is defined by the 'number' declares.

repcount

Evaluates to the current iteration number in the innermost repeat loop.

```
Repeat 10 [type(repcount)]
```

- 1234567890

stop

Stops the execution of the procedure.

until expr [commands]

The list of the commands is executed repeatedly until the expr (expression) evaluates to true. The expr is evaluated first (if it is true), so it is possible that the commands will never execute.

```
make "a 1
until :a>10 [type(:a)
make "a :a+1]
```

- 1234567890

while expr [commands]

Expr is evaluated and has to result to a Boolean value, that is true or false. As long as the expr evaluates to true, the given commands are repeatedly executed.

```
make "a 1
while :a<=10 [print(:a)
make "a :a+1]
```

- *1234567890*

wait(number)

Causes the current thread to pause for the given amount of time defined by 'number', measured in milliseconds.

do.until [commands] (condition)

A loop structure where a set of commands are executed repeatedly, until the condition evaluates to true. In the following example, the commands in the brackets are executed repeatedly until 'a' values more than 40.

```
make "a 34
do.until [print(:a) make "a :a+1] (:a>40)
```

- *34*
- *35*
- *36*
- *37*
- *38*
- *39*
- *40*

do.while [commands] (condition)

A loop structure, where a set of commands are executed repeatedly while the condition is true. In the following example, the commands in the brackets are executed repeatedly while 'a' values less than 10.

```
make "a 1
do.while [type(:a) make "a :a+1] (:a<10)
```

- *123456789*

Input – Output

print (expr)

Sends a string representation of the argument to the current writer, followed by an end-of-line. If the argument is a list, the outermost enclosing brackets are not output. If more than one argument is given, prints them on the same line, separated by spaces.

type(expr)

Operates in a similar way as 'print', but it is not followed by an end-of-line. For example, two 'type' in a row results to print the arguments in the same line.

.

Mathematical operations

integer(number)

Returns the integer portion of the argument.

```
print(integer(5.6))
```

- *5*

int(number)
Operates as 'integer' command.

chr(number)
Returns the character corresponding to the argument, according to the ASCII code.

```
print(integer(5.6))
```

- *A*

abs(number)
Returns the absolute value if the argument.

```
print(abs(-2))
print(abs(2))
```

- *2*
- *2*

sin(number)
Returns the sine of the argument (given in degrees).

```
print(sin(90))
```

- *1.0*

cos(number)
Returns the cosine of the argument (given in degrees).

```
print(cos(60))
```

- *0.5000000000000001*

tan(number)
Returns the tangent of the argument (given in degrees).

```
print(tan(45))
```

- *0.9999999999999999*

arcsin(number)
Returns the inverse sine of the argument in degrees. The returned value will be between -90 and 90.

```
print(arcsin(1))
```

- *90.0*

arcos(number)
Returns the inverse cosine of the argument in degrees. The returned value will be between 0 and 180.

```
print(arccos(1))
```

- *0.0*

arctan(number)

Returns the inverse tangent of the argument in degrees. The returned value will be between -90 and 90.

```
print(arctan(1))
```

- *45.0*

arctan2(number_y number_x)

Returns the inverse tangent of number_y number_x, in degrees, by finding the angle formed by the points (x, y), the origin and (1,0). The returned value will be between -180 and 180.

```
print(arctan2(1 1))
```

- *45.0*

not(Boolean)

Returns the logical opposite of the argument.

```
print(not(false))
```

- *true*

xor(boolean1 boolean2)

Returns the logical or of the arguments. The returned value will be Boolean, either True or False.

```
print(xor(false true))
```

- *true*

and(boolean1 boolean2)

Returns the logical and of the arguments. The returned value will be Boolean, either True or False.

```
print(and(false true))
```

- *false*

or(boolean1 boolean2)

Returns the logical or of the arguments. The returned value will be Boolean, either True or False.

```
print(or(false true))
```

- *true*

allof(boolean1 boolean2 …)

Returns true if all of the arguments result to either True or False. Alternatively, the arguments could also be 0 and 1.

```
print(allof(1 1 1 1 1))
print(allof(true true true false))
```

- *true*
- *false*

anyof(boolean1 boolean2 …)
Returns false if all of the arguments are False. In all other cases, returns true.

```
print(anyof(1 1 1 1 1))
print(anyof(true true true false))
```

- *true*
- *true*

round(number)
Round the argument to the nearest integer.

radsin(number)
Returns the sine of the argument (given in radians).

```
print(radsin(pi/6))
```

- *0.4999999999999994*

radcos (number)
Returns the cosine of the argument (given in radians).

```
print(radcos(pi/6))
```

- *0.8660254037844387*

radtan(number)
Returns the tangent of the argument (given in radians).

```
print(radtan(pi/6))
```

- *0.5773502691896257*

radarcsin(number)
Returns the inverse sine of the argument in radians. The returned value will be between –pi/2 and pi/2.

```
print(radarcsin(radsin(Pi/6))*(180/Pi))
```

- *29.999999999999996*

radarccos(number)
Returns the inverse cosine of the argument in radians. The returned value will be between 0 and pi.

```
print(radarccos(radcos(Pi/3))*(180/Pi))
```

- *59.99999999999999*

radarctan(number)
Returns the inverse tangent of the argument in radians. The returned value will be between –pi/2 and pi/2.

```
print(radarctan(radtan(Pi/2))*(180/Pi))
```

- *90.0*

sqrt(number)
Returns the square root of the argument.

```
print(sqrt(1024))
```

- *32.0*

pi()
Returns the value of pi.

minus(number)
Returns the negation of the argument.

```
print(minus(23))
```

- *-23*

sum(number1 number2 …)
Returns the sum of the arguments.

```
print(sum(5 3))
```

- *8*

difference(number1 number2)
Returns the difference of the arguments.

```
print(difference(5 3))
```

- *2*

product(number1 number2 …)
Returns the product of the arguments.

```
print(product(5 3))
```

- *15*

quotient(number1 number2)
Divides number1 by number2 and returns the quotient.

```
print(quotient(5 3))
```

- *1.6666666666666667*

remainder(number1 number2)
Divides number1 by number2 using integer division. The remainder is returned and the quotient is ignored.

```
print(remainder(5 3))
```

- 2

power(number1 number2)
Returns number1 raised to the power of number2.

```
print(power(5 3))
```

- 125.0

exp(number)
Returns the constant e raised to the power of the argument.

```
print(exp(5))
```

- 148.4131591025766

log(number)
Returns the natural logarithm (log to the base e) of the argument.

```
print(log(5))
print(log(exp(5)))
```

- 1.6094379124341003
- 5.0

random(number)
Returns a random integer between 0 and the given argument minus 1.

randomize(), randomize(number)
Sets the seed for sequence of random numbers to be returned by subsequent call to 'random'. The version with an argument uses the argument as seed.

Bitwise operations

Bitwise operations have decimal numbers as arguments. These arguments are converted to binary numbers and operate the corresponding action. The result is returned in decimal numbers.

bitnot(number)
Returns the bitwise not of the argument.

```
print(bitnot(10))
```

- -11

bitand(number1 number2)
Returns the bitwise and of the arguments. In the following example 3 is converted to 0011, 5 is converted to 0101 and the result in binary is 0001.

```
print(bitand(3 5))
```

- 2

bitor(number1 number2)

Returns the bitwise or of the arguments.

```
print(bitor(3 5))
```

- *7*

bitxor(number1 number2)

Returns the bitwise exclusive or of the arguments.

```
print(bitxor(3 5))
```

- *6*

lshift(number1 number2)

Shifts the bits of the first number left by the second number. To shift right, pass a negative number as the second argument. 0's shifted into all vacated digits.

```
print(lshift(4 1))
print(lshift(4 -1))
```

- *8*
- *2*

ashift(number1 number2)

Operates in a similar way as 'lshift'. The difference between these two commands is that 'ashift' preserves the sign of the number. That is, If the number is being shifted to the right, and the high-order bit of the original number is a 1, then 1's are shifted into the left end. If the high-order bit of the original number is a 0, then 0's are shifted into the left end.

```
print(lshift(-15 -2))
print(ashift(-15 -2))
```

- *1073741820*
- *-4*

Table of contents