

THE EASY GUIDE

TO

ELICA 5.0

It's more than "just another Logo"

written and illustrated by:
Pavel Boytchev, Elica Team
e-mail: pavel@elica.net
www.elica.net

Rev 3.2
September, 2001

Contents

1. ACKNOWLEDGEMENTS.....	3
2. PREFACE.....	4
3. FIRST STEPS IN LOGO.....	6
3.1 RECOGNIZING WORDS.....	6
3.2 TYPES OF THINGS.....	6
3.3 HOW TO SAY WORDS AND LISTS ?.....	7
3.4 HOW TO SAY ACTIONS?.....	7
3.5 HOW TO SAY VARIABLES?.....	9
3.6 HOW TO SAY COMMENTS?.....	11
3.7 BUILT-IN WORDS (KEYWORDS).....	12
3.7.1 <i>Printing texts (print)</i>	13
3.7.2 <i>Creating and changing variables (local, make, ob)</i>	13
3.7.3 <i>Executing instructions (run)</i>	14
3.7.4 <i>Controlling program flow (if, repeat, while)</i>	14
3.7.5 <i>Defining actions (to ... end, output)</i>	16
4. THE LOGO TURTLE.....	19
4.1 DANCING TURTLE (FD, BK).....	19
4.2 TURN LEFT, TURN RIGHT (LT, RT).....	20
4.3 MORE THINGS TO DO (HOME, PU, PD, PD?).....	21
4.4 UNDER COVER (HIDE, SHOW, SHOWN?).....	21
5. LOGO AND LANGUAGES.....	24
5.1 WHAT IS THIS? (NUMBER?, WORD?, LIST?).....	24
5.2 I NEED TO CONSTRUCT (WORD, LIST, SE, SET, FPUT, LPUT).....	25
5.3 I DON'T WANT IT ALL. I NEED JUST A BIT OF IT. (FIRST, LAST, BF, BL, ITEM).....	26
5.4 OTHER ACTIONS (COUNT, ASCII, CHAR, WAIT).....	27
6. LOGO AND MATHEMATICS.....	29
6.1 MATHEMATICAL OPERATORS.....	29
6.1.1 <i>Arithmetic operators (+, -, *, /, ^)</i>	29
6.1.2 <i>Integer arithmetic operators (idiv, imod)</i>	29
6.2 MATHEMATICAL FUNCTIONS.....	30
6.2.1 <i>Sign functions (abs, neg, sign)</i>	30
6.2.2 <i>Exponential functions (sqrt, exp, logn)</i>	30
6.2.3 <i>Rounding functions (trunc, round)</i>	31
6.2.4 <i>Trigonometric functions (sin, cos, tan, cotan)</i>	31
6.2.5 <i>Random function</i>	31
7. LOGO AND LOGICS.....	33
7.1 OPERATORS FOR COMPARING (=, <, >, <=, >=, <>).....	33
7.2 LOGICAL OPERATORS (AND, OR, NOT).....	33

1. Acknowledgements

Many thanks to Bob Gorman, who inspite of his anniversary celebration, found enough time to read every single word and improve the Guide. Further thanks go to Andreas Micheler, who was not afraid of the first revision and gave many valuable suggestions of how to refine it.

Special thanks to Bojidar Sendov and Jenny Sendova who guided me through the maze of programming languages and helped me see that the Exit is labeled as "Logo".

2. Preface

Scientists work very hard to make computers understand what you want, but until this is done, you need to talk to a computer in an language that it understands. Such a language is called *a programming language*, because its words are instructions which a computer can understand. In this context, programmers are people who know how to write programs.

Logo is one of the easiest programming languages. It is designed in such a way, that you can learn it with much less effort than any other language.

Elica is a software development system in which the base programming language is Logo. Elica Logo is a member of the Logo family languages and is the core of all Elica-based applications. The word '*Elica*' stands for *Educational Logo Interface for Creative Activities*.

Elica programs are ordinary text files that contain instructions. And like texts in books, these instructions contain words and punctuation symbols. Logo has a built-in knowledge of what to do when it meets some special words. To make it understand more, you need to teach it new words and how to react when you use them in instructions. You might correctly think, that teaching is difficult. Of course it is but if *you* are the teacher, *you define*. With the power to *define* you will be able to do just about anything with Logo.

This guide can be used as a short introductory tutorial while learning how to program with Elica. It uses easy to comprehend descriptions and is suitable for a large range of readers. Anyway, this guide does not cover all Elica Logo details. After reading it you will find that much of what is discussed here is common (more or less) for all Logo implementations.

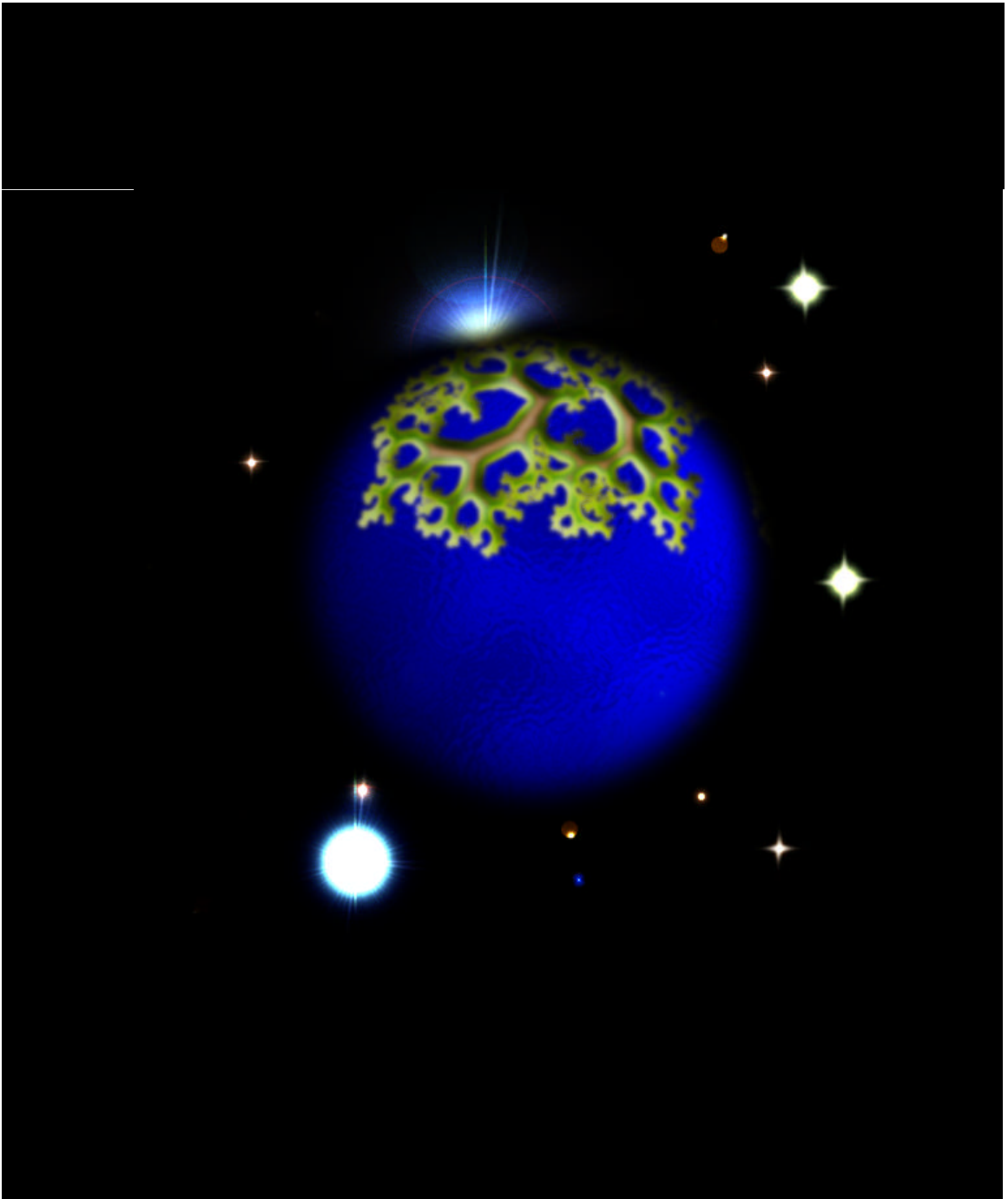
For advanced Logo users it might be better to just quickly glance over the "*The Easy Guide*" and then continue with "*The Complete guide*" where Elica Logo will be explained in details.

For those readers who are happier with scientifically dressed guides and feel better when they hear magic words like *polymorphism*, *multiple inheritance*, and *high-order functions*, it might be better to find another guide.

OK, let's start.

A 3D Turtle

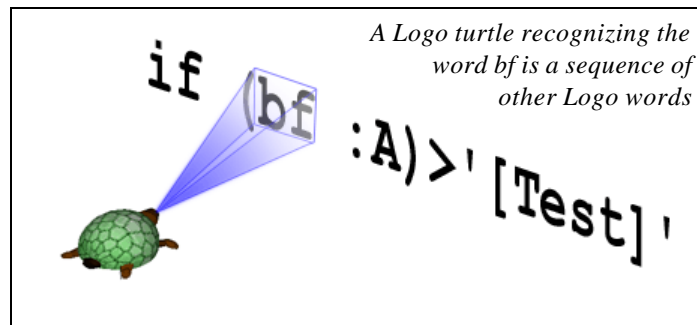
Projecting A Fractal Over A Planet



3. First Steps In Logo

3.1 Recognizing Words

When you read or write Logo programs, after some time you will intuitively understand what the individual words are that compose the programs. But sometimes programming languages differ from natural ones. That's why you would need to know how the Logo recognizes your words.



Here is a short list of the rules that Logo uses:

- A group of any characters on one line, that are placed between single quotes ' .. ' is one word.
- Otherwise, a word is a continuous group of characters framed by spaces or by symbols like brackets [], parentheses (), braces { }, semicolon ; , colon : , double quotes " , plus sign + , and comma , that are used to structure a Logo program and are always *one-letter words*.
- If a minus sign - is between a space and a digit, then it is a part of the word, containing the digit.
- A continuous group of characters containing only > , = or < is always one word too.

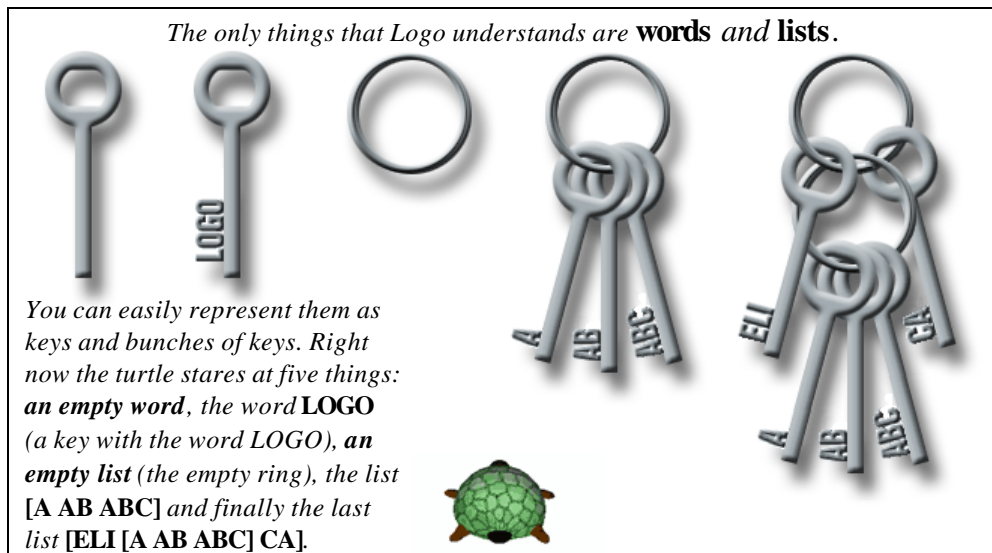
3.2 Types of things

Data types are the types of things which Logo can work with. Data types is a shortcut word for *types of data*. Traditionally Logo identifies two very general types of things: *words* and *lists*. Although Elica can identify these things, it has no knowledge of how to work with them. This is as if you can read a foreign language but cannot understand it.

WORDS are the most basic type of things. You already know what a word is as long as you know how to recognize Elica words in a program. Similar recognition is exactly what Elica does when it analyzes your instructions. Some words contain letters, others contain digits that form valid numbers

LISTS are *sequences of words*. The first word must be *open bracket* [and the last one must be *close bracket*]. All words in between are called *elements* of the list.

The list [**A AB ABC**] contains three words. A list that contains no elements is called *an empty list* and is written []. It is possible for one or more of the elements of a list to be lists by themselves too. In this three-element list [**ELI [A AB ABC] CA**] the second element is [**A AB ABC**] which is a list by itself. Lists are important for Logo, because all instructions you give to Logo are written as lists.



3.3 How to say words and lists ?

Imagine you want to order Logo to print a house. A computer is not as smart as you might think it is, so it cannot decide what exactly you ask for: to print the image of a house or to print the word 'house'. That's why when you write programs it is polite and often necessary to give hints to Logo as to how to treat ambiguous words.

If you want to use a word as a standard natural language word (for example when you want to print it), then write a double quote " before the word. "**cat**", "**X**", "**Case23**" are treated as the words '*cat*', '*X*' and '*Case23*'. If you say **cat** instead of "**cat**" then this will ask Logo to use it as *an action* (action is a synonym of instruction). Actions are described later.

Logo numbers are in fact words. That's why you can say " before them. Apparently when reading a number Logo knows that it cannot refer to an action, that's why it does not really need the quotation mark. As a result, you can say numbers directly: **56, 31. 8, - 300**.

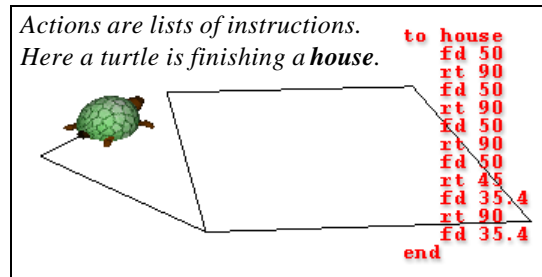
Like numbers, lists are unambiguous, too. When you want to say a list, you just say it as it is, but do not forget to put it in brackets [...], otherwise Logo will not understand that this is a list.

3.4 How to say actions?

Saying words and lists is as if you use only nouns in your everyday speech. Only verbs are the means to express what you do with these things. Traditionally Logo likes

to see names of actions used directly (without special marks). If you want to print 5 and 6, you only need to order **print 5 6**. Many actions require the things they work with to be easy to find. That's why these things (called *arguments*, *parameters* or *inputs*) are usually written just before or after the name of the action.

When you define a new action, then you also define, whether it will look for arguments before or after the name, or it will look in both places. For example, the action `+` expects one argument from the left side and one from the right. To use already defined actions you need to know their names and where they expect their arguments to be placed. If you fail to place them on the correct places, some actions may fail too.



It is the right place now to explain the difference between *arguments* and *parameters*. You can think of arguments as the things, which are actually given to an action to use (some also call them *actuals*), while parameters are the formal things that an action expects (the *formals*). In most cases arguments and parameters match.

Some actions can work with less or more arguments. If you are not sure how many parameters are expected, or you are sure that you provide more or less arguments, or, last but not least, you want a better control of how Logo interprets your instructions, then you must frame the action and all its arguments in parentheses (...). When Logo sees an action in parentheses, it tries to force the action to use all arguments.

In the following example you can see the action that is called **first** to be used in its standard way to extract the first letter of a word and print **c**, and later it is forced to work with two arguments (the second one is the number of letters to get from the heading of the word) and ... voila .. it prints **ca**. The third instruction will print **c 2**, because the number will be processed by **print** and Logo has no clue, that it must be used by **first**.

```
print first "cat           ; c
print (first "cat 2)      ; ca
print first "cat 2       ; c 2
```

So, the general rule for saying an action is to write:

```
arg arg ... action_name arg arg...
```

If you want to explicitly set which arguments are given to an action, then use parentheses (...) in this way:

```
(arg arg ... action_name arg arg...)
```


You've seen that doing actions is quite a powerful activity. Everything that Logo does is done in the form of actions. That's why for clarity we will use in descriptions other, more specific terms for different types of actions. Actions that are not supposed to return results will be called *instructions* or *commands*. Actions that calculate and return results will be called *functions*. Functions that expect arguments from both sides or from the left will be called *operators*. Actions, which arguments contain other actions are often referred as *expressions*.

Anyway, do not make efforts to learn these terms by heart, because this classification of different types of actions is quite relative and is not formal at all. All you need to know is, that instructions, commands, functions, operators, expressions (and a bunch of other words) are all different flavors of actions.

3.5 *How to say variables?*

If Logo is your first programming language, then it is possible, that you have not been exposed to variables before. What are variables and why do programming languages need them? Imagine you are responsible for writing fire escape instructions. When you write it you can refer to different people as "the one who is on duty". You write this just because you don't know who will be on duty at actual time of the fire. Of course, you can put names in the instructions, but then you must provide a separate instruction set for each person who could be on duty.

*Variables are like chests. You can put inside a key or a bunch of keys - the **value**. In order to open the chest you need another single key - the **name**. Now the turtle carries a key representing the word LOGO to open a chest containing the bunch [A AB ABC].*

An illustration of a wooden chest with a keyhole, a key, and a small green turtle holding a key labeled 'LOGO'. The chest is made of vertical wooden planks. A key is hanging from the keyhole. To the right of the chest, a small green turtle is holding a key with the word 'LOGO' written on it.

In Logo you can use a word as a name of a value. The important thing is, that you can change the value and use it by the same name. That's why a name and its associated value are called a *variable* - you can vary the value of a variable, but keep the program unchanged.

This is one of the most important features of computer programs - you write a program once and then run it with different sets of data.

A variable in Elica Logo is defined by its *name*, *value* and *position*. You already know that variables' names are words and their values can be anything - words, numbers, lists and actions.

When you want to say the name of a variable, ignoring its value, follow the rules for words and put a double quote " before it. This is often needed to create a variable. If you want this you can just ask Logo in a very natural way to make it.

```
make "a 5
make "fellow "mike
make "LOGO [A AB ABC]
```

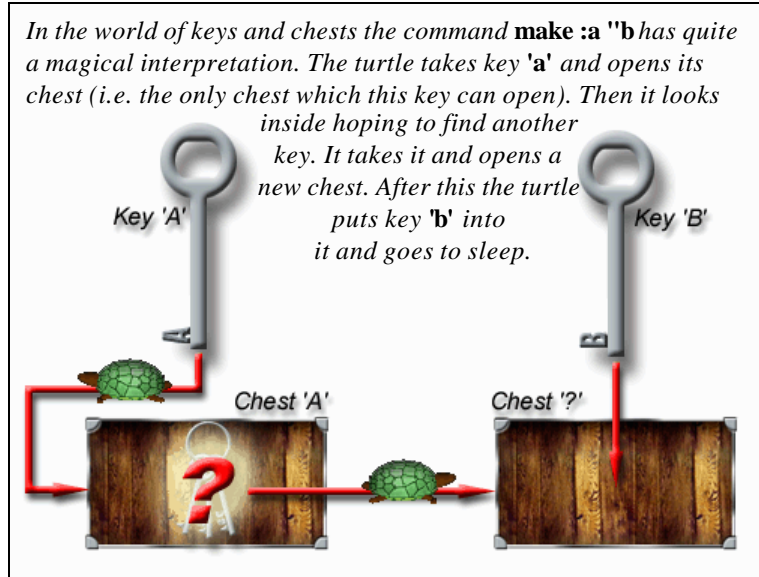
The first instruction creates a variable called **a** which contains the value **5**. The second one creates variable **fellow** and assigns it the word **mike**. The third one creates **LOGO** and assigns it a list. **Make** is one of the few actions about which Logo has prebuilt knowledge how to handle. Later on you will find a longer description of the action **make**.

What about values? How to refer to a value of a variable? There is a special action in Logo which you can use for this purpose. If you use the action colon **:** before a word, then Logo will try to read the contents of a variable with a name the same as the given word. Let's ask Logo to find the sum of two numbers stored in the variables **a** and **b**. To do this you need to ask something like **:a + :b**.

Logo is a powerful language. *Why?* There are many explanations, but one of the coolest is that Logo can work with variables which names are not known in advance. Consider the next example:

```
make :a "b
```

How will Logo interpret it? It will first find the associated value of a variable called **a**. This value (rather than **a**) will be used as a name of a variable to create.



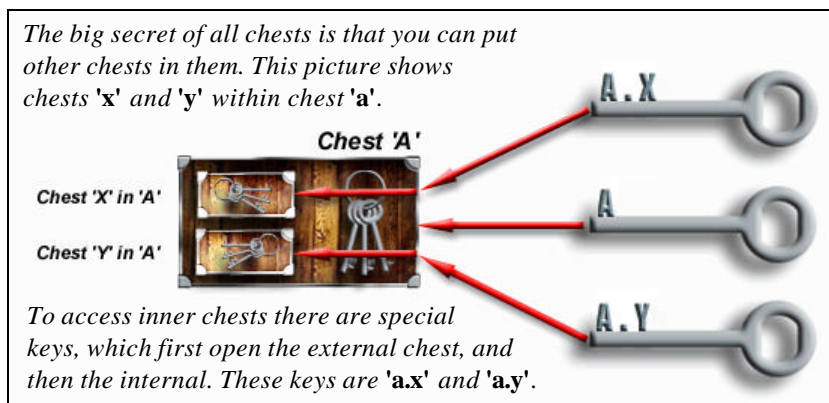
Except names and values, variables have positions. The position is a description of where a given variable exists. What are the possible positions for a variable? In Logo each variable can be placed inside another variable. In other words, each variable can contain not only a value, but also another variable in addition. Even more - a variable can contain many variables. Some people call these variables *children*. Other people prefer the word *local variables*. There is a third group - the hard programmers - that would call them *fields*, *members* or *methods*. As it always happens it doesn't matter

how you call them - Logo does not use these words. They are used by humans, and when you talk with your friends or classmates about Logo, good communication and discussion will only be possible, if you and your fellows use terms, which you and they know (and like).

So, let's go back to positions. If you want to define a variable **a** which is inside **b**, then you can use the name **b. a**. This instructs Logo when looking for the value of **a**, to locate this variable inside the variable **b**. Of course you can nest variables deeper. For example, **d. c. b. a** is the name of variable **a**, which is a child of **b**, that is nested in **c**, which is a local variable of **d**.

```
make "a.x 10
make "a.y 20
make "b :a
print :b.x :b.y ; 10 20
```

Examine the example above carefully. It shows one basic feature of local variables. When Logo takes the value of a variable, then it takes not only the value but all children too. The first two lines define a variable **a** which has two local variables - **x** and **y**. When Logo creates the variable **b** it copies **a**'s **x** and **y** to **b**. So from now on there will be two variables called **x** (one in **a** and one in **b**) and two called **y**. When you talk about locals, for example **cat. x** and **dog. y**, then speaking it *cat`s*



x and *dog`s y* will be very convenient. Indeed it is very important to use *good* names for variables and actions. Use your fantasy and creativity, and don't hesitate to think over a name a bit longer, because a good name often makes things much clearer!

3.6 How to say comments?

Comments, or *remarks*, are a way to talk to yourself while talking to Logo. Usually your comments serve as reminders for you, and thus, they are completely ignored by Logo. So in comments you can, for example, describe in few or lots of words, what your program, a section of your program, or a complex line of it does or should do.

You can also place a comment, where you think that your program has a bug, or where you have something in mind, but don't know how to do it yet, or anything else,

which is important to you *now or later*, or might be important for your friends, if you share or exchange your programs with them.

Don't underestimate the value of comments!

Imagine, a friend gives you one of his programs. Then, wouldn't it be much easier for you to read it, if there were some hints and notes in it? And wouldn't it be easier for you to expand it, if there were some marks, where your friend had no clue how to do it? Wouldn't it be easier for you to help your friend fixing a bug, if s/he had commented some suspected lines?

But you should really remove comments, which are not up to date and consistent with the program, because they might be more confusing than no comments at all.

You can write whatever you like in your comments. There are no rules to keep up with except one - Logo must understand where your comments start and end.

There are two types of comments in Logo - *short* and *long*. Short comments start with the word `;` (semicolon) and continue till the end of the line.

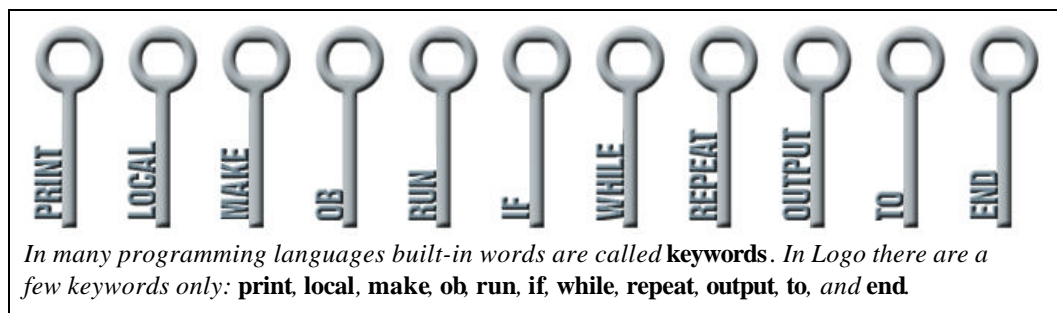
```
make "a 105 ; current check number  
make "b :a+1 ; next check number
```

Long comments start with the word `{` (open brace) and end with the word `}` (closed brace). They are called long, because they can start in one line, and end in another. Long comments can host other comments, short or long, so whenever you use an open brace `{` you must later say closed brace `}`.

```
{ This code calculates the  
  average of three numbers }  
make "m (:a+: b+: c) /3
```

3.7 Built-in words (keywords)

Teaching Logo to do something is based on the use of a few *built-in words*. These are the words that Elica is aware of and can respond to them. Elica already knows their meaning, because that knowledge has been implemented at Elica's birth.



3.7.1 Printing texts (*print*)

The most common action that any Logo can do is to print one or more values on the screen. The instruction, which asks Logo to do so, is called **print**. If you want the system to print something you can just say **print** and then all the things in a row. The next example shows how to print the number **5**, the word **five**, the value of the variable **pos** and the result of the action **+**, that needs two numerical arguments.

```
print 5 "five :pos 1+2
```

If you ask Logo to print something but do not say exactly what, then the system will print an empty line. This technique is often used to format the output of your program.

3.7.2 Creating and changing variables (*local*, *make*, *ob*)

Another common action, that Logo can do, is to create variables and change their values. There are two instructions that ask Logo to do this. When Logo executes **local**, then it creates a new variable inside the currently active action. Thus it will be local to it. If a such already exists then it does nothing. In most cases when an action is completed, all its local variables are cleared. **local** is like **print**. It uses all the things that it can find after it, but unlikely **print**, **local** expects to find only words, which it uses as names of variables.



```
local "x "y "done
```

The other instruction to create variables is **make**. You have already met it. Except for creating variables it can be also used to modify their values. To do this Logo needs two arguments - the name of a variable and its new value. That's why the most common use of this instruction is in the form of:

```
make "a 1  
make "b :a+10
```

There is another instruction to create variables - it is called **ob** and its name comes from Geomland - a nice and powerful Geometry-oriented Logo implementation. This instruction works the same way as **make**, except that it creates links between variables. See the following example:

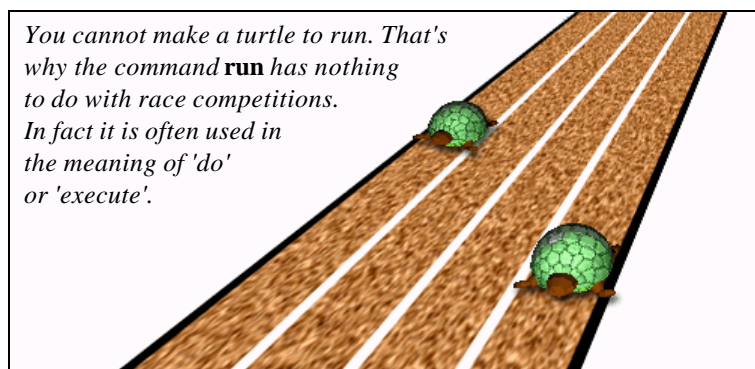
```
ob "a 1  
ob "b :a*:a  
ob "c :b*:b  
print :a :b :c ; prints 1 1 1  
  
ob "a 2  
print :a :b :c ; prints 2 4 16  
  
ob "a 3  
print :a :b :c ; prints 3 9 81
```

By the use of **ob** you politely ask Logo to remember the relationship between variables. When you change only one of them, all dependents will be recalculated automatically!

3.7.3 Executing instructions (*run*)

When Logo reads your program it knows what instructions to run and in what order. Anyway, there are cases, when you want to ask Logo to run instructions which are unknown at the time of writing the program. These instructions may be generated by your program or may be contained in an external text file. In such cases you have to ask Logo to run these instructions by saying **run**.

run :mycommands



Of course, instead of a variable or expression you can use a list, but in this way both next lines will have the same effect:

```
run [print :x :y]
print :x :y
```

The instruction **run** serves another purpose too. If instead of a list of instructions it finds a word, then it will use this word as a name of a file and will look for instructions in it.

```
run "vectors
```

The example above will ask Logo to find file **vectors.ei**, to load it, and to learn from it as much as possible.

3.7.4 Controlling program flow (*if, repeat, while*)

The power of the programming languages including Logo is based on the possibility to control the program execution flow. If you were allowed to write programs with instructions executed one by one from the top to the bottom, then you would not be able to easily teach Logo to do really cool things.

There are three built-in instruction which you can use to control how Logo executes a program.

One of them, **if**, is used to tell Logo to execute a list of instructions only if a given condition is fulfilled.

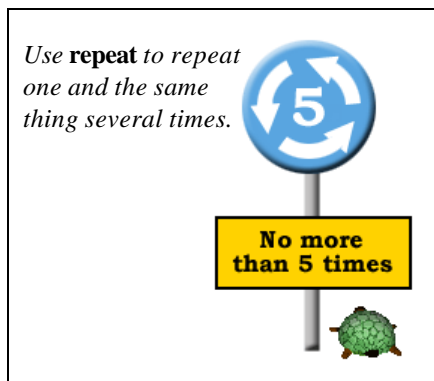
```
if :a>0 [print "positive]
```

Here you say Logo to print the word **positive** only if the value of **a** is greater than zero. Apparently **if** enables you to define not only what to do when something is true, but also what to do otherwise:

```
if :a>0  
  [print "positive]  
  [print "negative "or "zero]
```

To make shorter line Logo agrees to look for lists of instructions not only in the line where **if** is used, but also on the next line too. Anyway, the condition must always be on the same line - right after **if**. Speaking of conditions, Logo decides whether one is fulfilled only by its value. If its value is the word **true**, then the condition is fulfilled. Otherwise it is not.

The other instruction for program flow control is **repeat**. You can use to ask Logo to repeat a sequence in instructions several times. The following instruction:



```
repeat 5 [print "Hello]
```

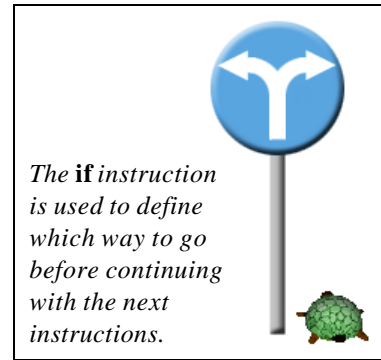
is functionally equivalent to:

```
print "Hello  
print "Hello  
print "Hello  
print "Hello  
print "Hello
```

Of course if you want to repeat an action thousands of times, then it would be quite efficient to use **repeat**. It is recommended not only for great number of repetitions, but also for unknown number of repetitions. In many cases you do not know this number, but you can calculate it. Here the value of **chars** determines how many times to repeat the instructions in the list.

```
make "i 65  
repeat :chars  
  [  
    print char :i  
    make "i :i+1  
  ]
```

Often **repeat** instructions are called *repeat cycles*. There is another type cycles - *while cycles*. As you can guess, they use the instruction **while**.

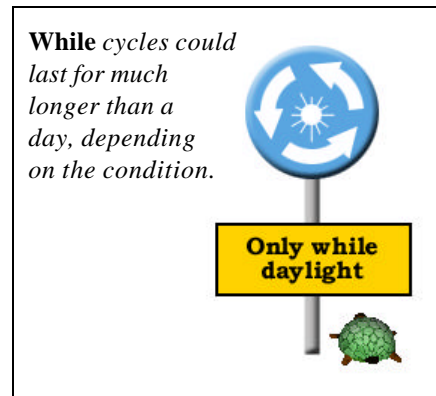


```

while :a<10
  [
    print :a
    make "a :a+1
  ]

```

A while-cycle is similar to a repeat-cycle because it asks Logo to execute a list of instructions several times. The difference is that if you use **while**, then the repetition is canceled only when a condition is not fulfilled. The example above instructs Logo to print out the current value of **a** until it is less than 10.



3.7.5 Defining actions (*to ... end, output*)

You already know that to ask Logo to do something, you need to specify what action to do. To teach Logo to do new things there must be a way to describe new actions. The definition of an action starts with the word **to** and ends with the word **end**. Right after **to** you must say the name of the action (without " in front) and the parameters of the actions. Parameters that are expected to be said before the name, should be declared before it. Parameters that are expected to be said after the name, should be declared after it. In the lines between these declarations and the word **end** you must include the instructions that make up the actions.

```

to greater :x :y
  if :x>:y [print "yes] [print "no]
end

```

The definition above is about an action called greater which expects two arguments after its name. When you say this action to Logo, for example:

```

greater 5 6

```

then it will print **no**. Logo creates one local to the action variable for each argument. The names of these variables are taken from the first line of the declaration. In this case they are **x** and **y**. Note that you must say **:** before each name of a parameter. Why? First of all, this is to remind Logo that the parameters expect values. Secondly, this is the only way for Logo to distinguish the action's name from it's parameters' names.

The order of the parameters is decided by you and anyone that uses your action, even if this is you, must use the same order. For example, if you define **greater** in many other ways, but whatever way you choose, you should use exactly as it is.

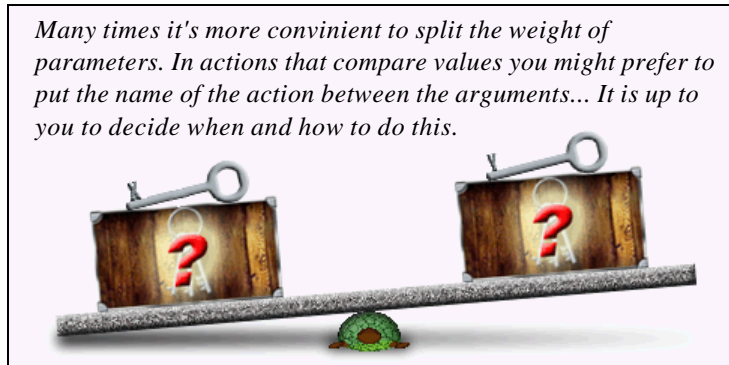
```

to :x greater :y
  if :x>:y [print "yes] [print "no]
end

```

5 greater 6

Inside an action you can do everything that you can do in Logo. For example, you can define variables or actions. Anyway, there is one special thing that is unique to actions. Sometimes you may want to define an action that returns a value that can be used in further calculations.



Such a value is returned back with the instruction **output**. If you wish to change the example above not to print the result of the comparison, but to return it back, then you may change the declaration in this way:

```
to :x greater :y
  if :x>:y [output "yes] [output "no]
end

print 5 greater 6 ; no

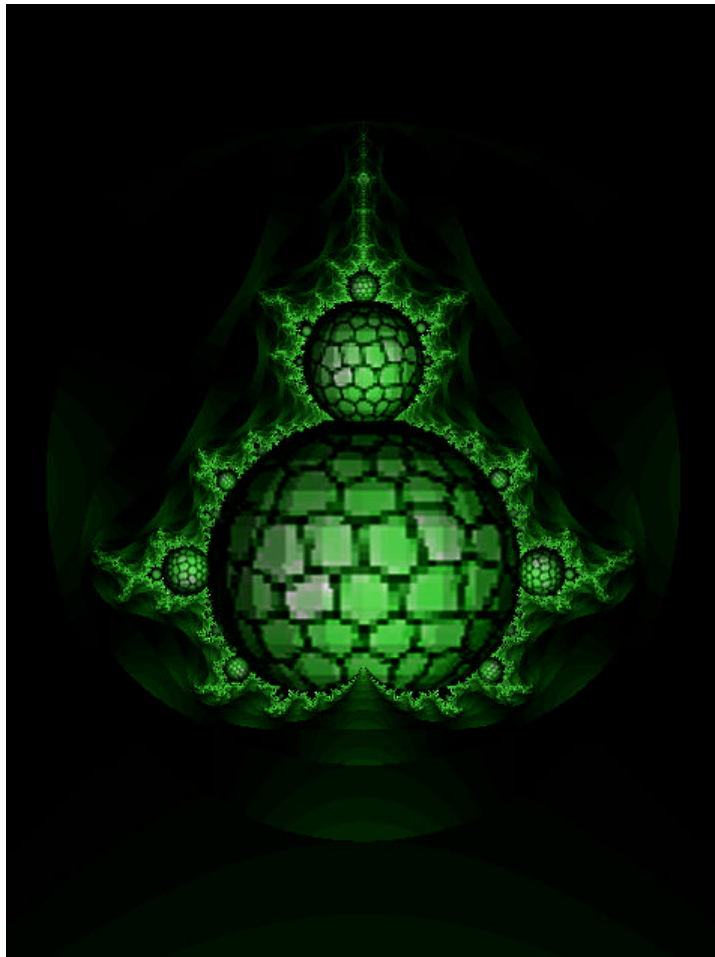
if (5 greater 1)="yes [print "really?] ; yes
```

Returning a value will make it possible to use the result of the action as it is shown in the **if** action above.

When Logo executes **output** it stops the action immediately. This effect can be used when you want to cancel an action earlier. In such cases you can use **output** without saying what value to return.

A Mandelbrot Fractal

Built By The Bodies Of Turtles



4. The Logo Turtle

The turtle is the face of Logo. From their early days Logo implementations were able to control a turtle to draw beautiful figures. Even now many of the people who have heard about Logo have also heard about *turtle graphics*.

Turtle graphics has a broad meaning - from program control of a turtle up to a complex fractal image. But there is one central figure in all these - that's the turtle.

Well, traditionally Logo's turtle is a small triangle which is under your program control. You tell it what you want it to do and it makes it best to do it.

The Elica turtle comes with prebuilt knowledge of some of the most basic actions, but if you want the turtle to draw houses and trees, then you have to teach it. Anyway, this prebuilt knowledge is now available at startup, so if you like to use a turtle, your first command must be:

```
run "turtle
```

It will load a file of instructions which will teach the your turtle of the basic actions.

4.1 *Dancing turtle (fd, bk)*

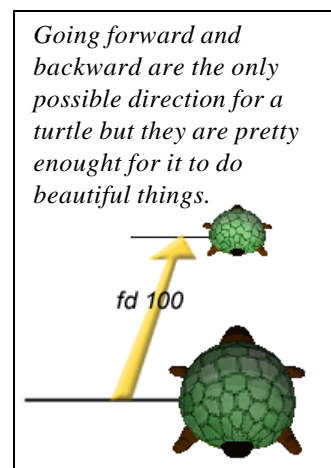
The best way to understand how the turtle moves is to imagine that you are the turtle. If you hear the command forward 10, you, most probably, will make 10 steps forward. That's the same that the turtle will do. The only difference is that its steps are much smaller than yours, and the name of the action 'forward' is actually shortened to **fd**. In a similar way going backwards is done with the action **bk**.

The following instructions will ask the turtle to go 10 steps forward and then to go backward 5 steps.

```
fd 10  
bk 5
```

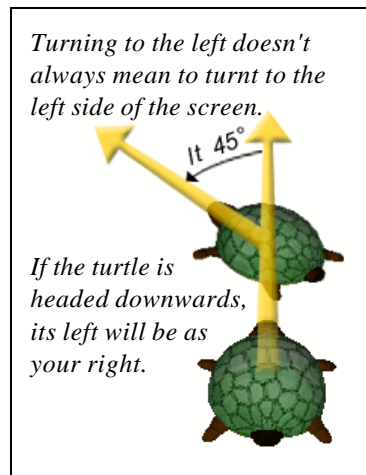
On the screen you will see the trace of the turtle as a segment and the turtle itself staying in the middle.

It's like a dance - you are the choerographer and the turtle is the dancer.



4.2 Turn left, turn right (*lt*, *rt*)

Stepping forward and backward will soon become a boring dance unless you figure out how to make the turtle turn to the left and to the right. Because of this is very important, the turtle already knows how to do it and it just waits for your orders. To make it turn to the left or right just say **lt** (it comes either from left or from left turn) or **rt** (right, right turn). Of course the turtle expect from you not only to say in what direction to turn, but also how much to turn. This how-much is expected as an argument to **lt** and **rt** and it must be in the form of *degrees*.



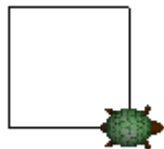
What are degrees? Degrees have been used by ancient mathematicians to measure angles. Because of these we still use them nowadays to express numerically the 'size' of an angle. *How big is one degree?* There is no simple explanation. A mathematician will say that there are 360 degrees in a circle, but it's hard to imagine $1/360^{\text{th}}$ of a circle.

When was you last time eating a slice of pizza? Do you remember how much slices the whole pizza was cut into? If there were 6 slices then the angle of each slice would be 60 degrees. If there were 8 slices - each of them would be 45 degrees. Pay attention that the angle of a slice does not depend on the size of the pizza. If you cut a 10" and a 20" pizza into 6 slices each, then all twelve slices will have the same angle of 60 degrees.

The other way to represent the size of one degree is to look at a clock or a watch with arrows. The tiniest arrow which measures seconds, turns 6 degrees to the right each second. So, one degree is $1/6^{\text{th}}$ of a turn of the seconds arrow. There is no problem if the seconds arrow is missing. The minutes arrow turns 6 degrees per minute, thus a slice of 1 minute is 6 degrees big. A slice of one hour it exactly 30 degrees. Three hours are 90 degrees. Remember this number - 90 degrees. When we, people, say to each other *turn to the left*, we mean **turn 90 degrees to the left**. The Logo turtle is a computer creature and it does not behave like us. That's why you must always say the angle of turning even if it seems obvious for you.

Now, lets try a small example:

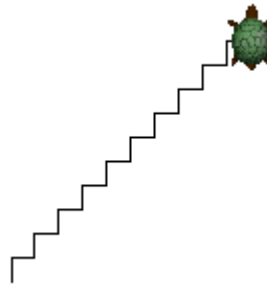
```
fd 50  
lt 90  
fd 50  
lt 90  
fd 50  
lt 90  
fd 50
```



If you were a turtle and follow strictly these instructions, you should draw a square. You go 50 steps forward, turn to the left, again walk 50 steps, turn to the left, go 50 steps, turn to the left, and make the final 50 steps.

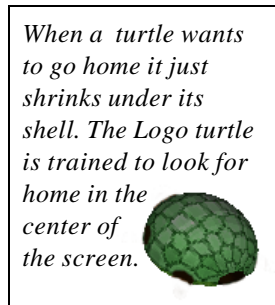
Now let's make stairs to the right:

```
repeat 10
[ fd 10
  rt 90
  fd 10
  lt 90
]
```



4.3 More things to do (*home, pu, pd, pd?*)

Except for these four actions - **fd**, **bk**, **lt** and **rt**, the turtle also knows how to jump directly to a special position, called *home*. It is (usually) in the center of the screen. The name of the action is **home**. When you say this magic word the turtle jumps to its home and turns itself upward (i.e. it faces the upper side of the screen).



You've already noticed that as the turtle moves on the screen it leaves traces, so you can track its path.

Sometimes, when you draw something, it is so complex that it cannot be drawn as one path. For this purpose you would like to ask the turtle to stop leaving traces behind. That's easy, because turtle traces are not because of its dirty feet, but because it carries a pen. At startup the turtle puts the pen down, so that when it moves, the pen draws. To ask the turtle to pick the pen up, use the instruction **pu** (*pen up*). From now on the turtle will move without leaving traces.

When you want to start to draw again, you should use the opposite instruction - **pd** (*pen down*). At any moment you can check whether the pen is up or down. If you ask the turtle **pd?** it will respond with the word "**true**" or "**false**".

```
if pd?
[ print "' Pen is down' ]
[ print "' Pen is up' ]
```

4.4 Under cover (*hide, show, shown?*)

When painter draws a picture he stands close to it. But when the painting is done, he withdraws so that to allow others to enjoy it. It is the same with the turtle. You control it and draw a beautiful figure, but the turtle is still there, on its last position. Of course, you can tell it to hide and to go away it is easy - pick up the pen and go forward far enough beyond the edge of the screen.

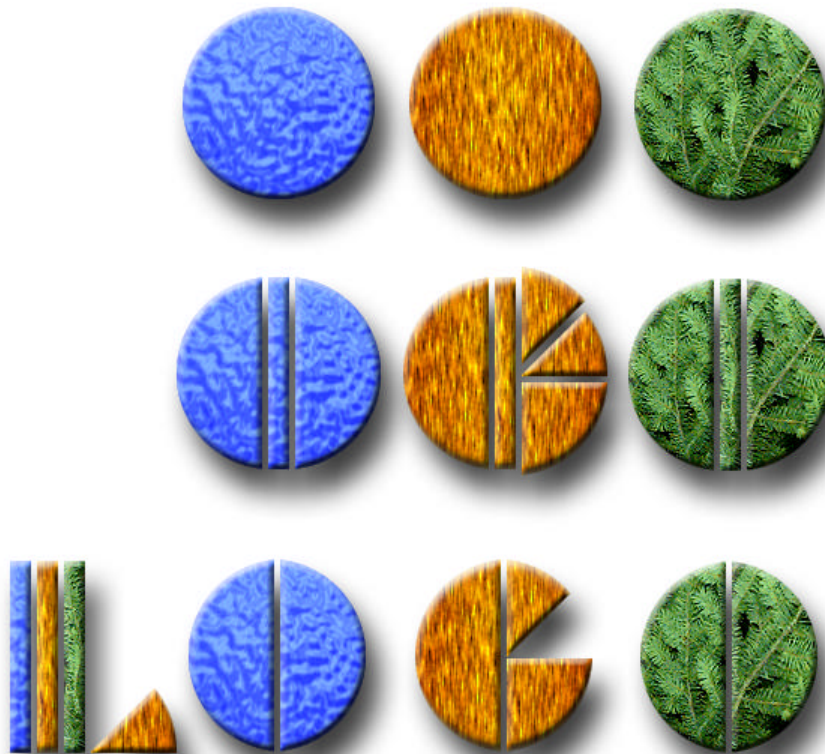
Anyway, there is a better way to do this. Say **hide** and the turtle will put on its invisible hat. It will be there, it will not change its position at all, but it will be invisible.

Now you can ask the turtle to dance and you will not know where it is, but if the pen is down, it will draw traces. As a side effect using a hidden turtle draws figures faster.

When you want to show back the turtle use the instruction **show**. As it is the case with **pu**, **pd** and **pd?**, you can ask the turtle to say whether it is hidden or not with the question **shown?**.

The Logo Word

Can Be Constructed This Way Too



5. Logo and Languages

If you have worked with other Logo implementations then you may wonder where are the word and list processing functions. The core of Elica does not understand what to do if you ask it to work with lists. Fortunately, you can teach Logo so that it can learn this.

Elica comes with already built Logo lessons. One of them is used to teach Logo all the actions that are expected from any Logo to be able to do. In order to teach Logo how to become a complete Logo, you must say:

```
run "logo
```

This will ask Logo to load file *logo.eli* and learn from it what it can. Actually, there is no need to do this, because Logo will do this for you automatically.

One of the most important features of Logo is that you can ask it to process words and list. Apparently natural language sentences are often represented as list of words. Thus language processing is done by Logo in term of word and list processing.

5.1 *What is this? (number?, word?, list?)*

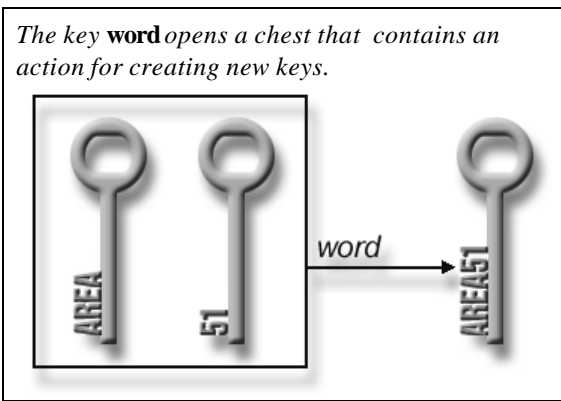
To write a program that proceses statements you often need to check the type of values hidden in different variables. You can ask Logo to do this by executing the following actions: **number?**, **word?**, and **list?** which return either **true** (if the type of their argument matches their names) or **false**.

```
print number? 5           ; true  
print word? "K12         ; true  
print list? "mice        ; false  
print list? [m i c e]    ; true
```

It is a good practice when you create your own data type query fuctions, to name them after the type they check for. And, of course, include a question mark at the end.

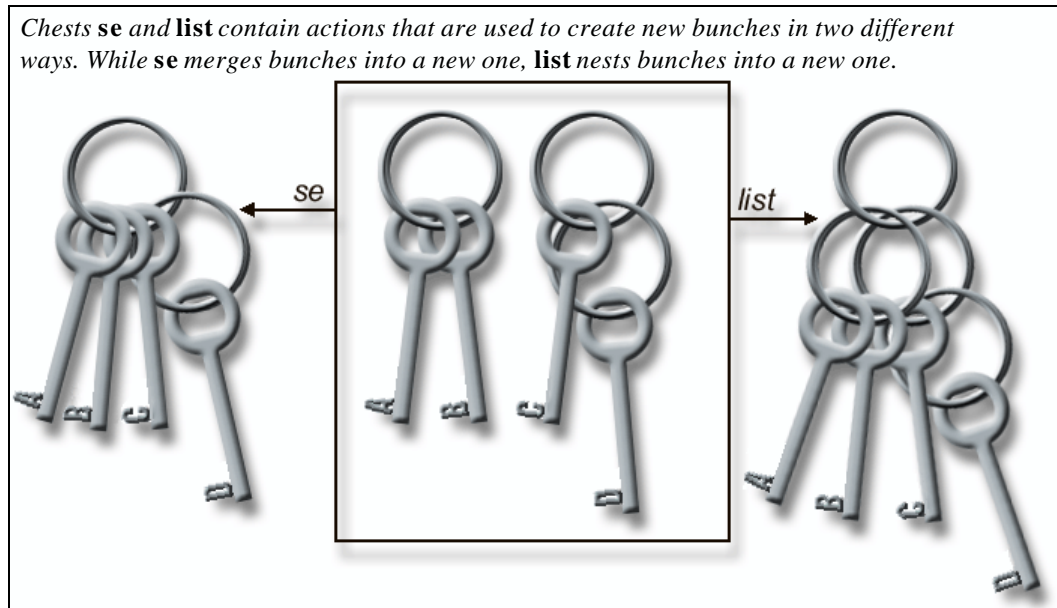
5.2 I need to construct (word, list, se, set, fput, lput)

The core of language processing is the creation of new words and lists. For each of these types Logo has one or more functions that create values. All these functions can accept many arguments. The function **word** is used to concatenate several words in one word. **List** is used to create a list which elements are the values of the arguments. A similar function is **se**. Its name comes from *sentence*. Like



list, **se** creates a list, but all arguments that are lists are 'unpacked' and their elements become elements of the resulting list.

```
print word "area 51           ; area51
print list [a b] [c [d]]      ; [[a b] [c [d]]]
print (list [a] "b [c])      ; [[a] b [c]]
print se [a b] [c [d]]       ; [a b c [d]]
```



There are two other functions - **fput** and **lput**. They are used to construct a new list where the new element is appended at the beginning or at the end of a given list.

```
print fput "x [a b]          ; [x a b]
print fput [x] [a b]         ; [[x] a b]
print lput "x [a b]          ; [a b x]
print lput [x] [a b]         ; [a b [x]]
```

5.3 *I don't want it all. I need just a bit of it. (first, last, bf, bl, item)*

Constructing words and lists is a one-way process. To have the full power **Logo. eli** teaches Elica not only how to build values, but also how to decompose them and to extract specific parts.

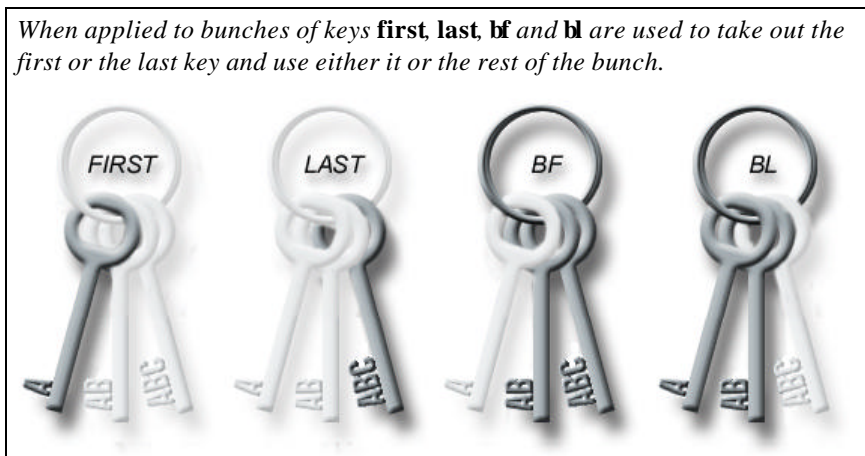
The action **first** asks Logo to extract the first elements of a value. If the value is a word, then the result is a word of its first letters. If the value is a list then the result contains the first elements of the list or the set.

When extracting first elements, the default behaviour of Logo is to extract only the very first one. If you want to extract more elements, you can force **first** to do so by simply adding one more argument containing the number of elements to extract.

```
print first "mi nt           ; m
print (first "mi nt 2)       ; mi
print first [a b f]          ; a
print (first [a b f] 2)      ; [a b]
make "myset first :myset
```

To make things symmetrical, Logo knows not only how to extract the first elements, but how to extract the last elements with the function **last**.

```
print last "mi nt           ; t
print (last [a b f] 2)      ; [b f]
```



For convenience Logo also knows how to extract a part of a word or a list when you define what part to ignore. There are two functions for this: **bf** and **bl** that stand for *butfirst* and *butlast*. What **bf** does is just to find all elements except the first one or more ones. Similarly, **bl** returns all the elements except the last ones.

```
print bf "mi nt             ; i nt
print (bl "mi nt 3)         ; m
print (bf [a b f] 2)        ; [f]
print bl [a b f]            ; [a b]
```

All functions for extractions described so far refer to elements with position relative the the begining or the end of something. In some cases you will need to refer to a specific position - 4th, 10th, ith. You can extract there elements by a wise use of **first** and **bf**, but Logo provides another useful function. It is **item** and expects two arguments - the index of the element you want to extract, and the value from which you want to extract this element.

```
print item 3 [s c r e e n] ; r
print item 4 "clock        ; c
```

5.4 Other actions (*count, ascii, char, wait*)

There are some other functions in Logo that do not fall in any of the groups so far. In this section you will find their short decriptions.

The size of a word (or a list) in terms of number of letters (or elements) is calculated by **count**. This function cannot be used with sets.

```
print count "book          ; 4
print count [b [o o] k]    ; 3
```

Conversion from letters to numbers and back is required when you work with the ASCII representation of letters. Conversions are handled by two function: **ascii** returns the ASCII code of a letter, and **char** returns the letter with given ASCII code.

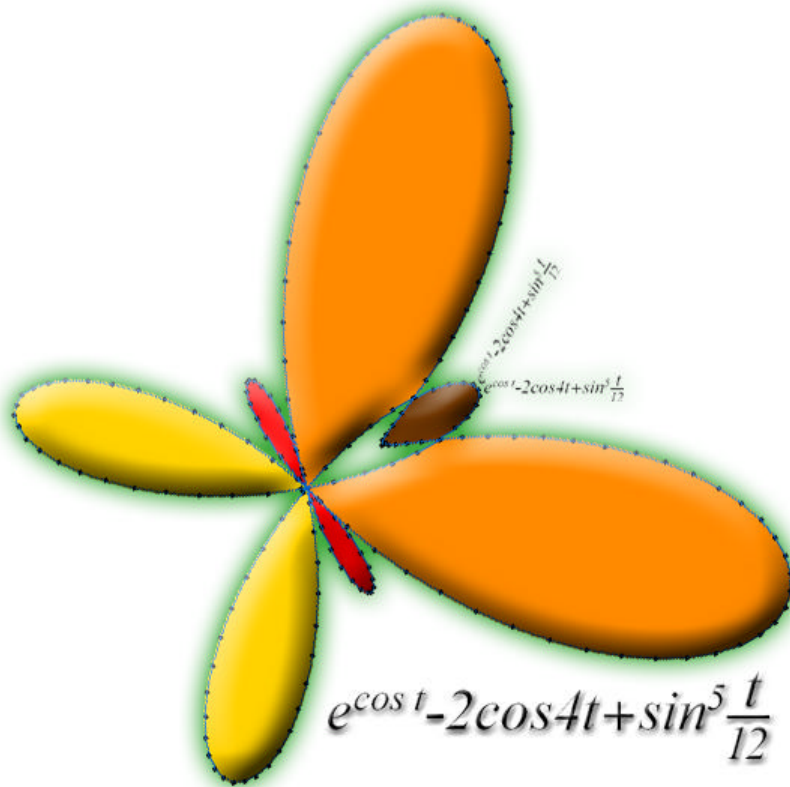
```
print ascii "B            ; 66
print char 65            ; A
```

Delays in program execution can be acomplished with **wait**, which expects one argument - the number of seconds to wait.

```
wait 10      ; will pause for 10 seconds
wait 0.5    ; will pause for half a second
```

A Butterfly

Defined By A Parametrical Equation



6. Logo and Mathematics

Computers were initially made for helping people calculate quickly and precisely. This requirements influenced most programming languages including Logo. Thus you can use Logo to write programs that calculate something using some of the mathematical operators and functions.

6.1 *Mathematical operators*

The mathematical operators are implemented in Logo as actions, which expect two arguments - one from the left and one from the right.

6.1.1 Arithmetic operators (+, -, *, /, ^)

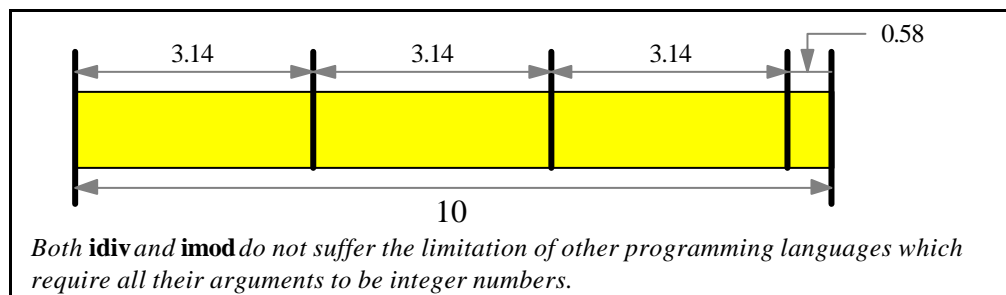
The first thing that Logo learns after executing **run "logo**, is a number of very important operators, which it can use to build mathematical expressions. The operator + is used to add numbers, - is to subtract, * is to multiply, / is to divide, and finally, ^ is for calculating powers. The following examples demonstrate how you can use these operators:

```
print 2 + 4      ; 6
print 2 - 4      ; -2
print 2 * 4      ; 8
print 2 / 4      ; 0.5
print 2 ^ 4      ; 16
print 2^3 + 3*4  ; 20
```

6.1.2 Integer arithmetic operators (idiv, imod)

There are only two integer arithmetic operators in Logo - **idiv** and **imod**. The first one is used to find how much times a number can 'fit' into another number. The second one calculates the remainder. The effect of integer arithmetic operators can be easily demonstrated with a couple of examples:

```
print 10 idiv 3.14 ; 3
print 10 imod 3.14 ; 0.58
print 3*3.14 + 0.58 ; 10
```



The instructions above calculate that 3.14 fits 3 times in 10, but there is a remainder, which is 0.58.

6.2 Mathematical functions

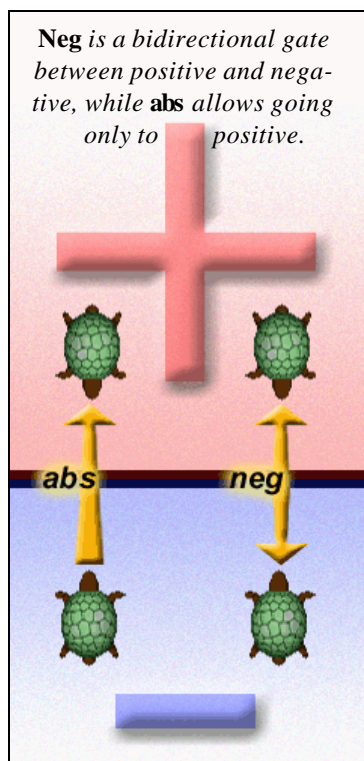
The mathematical functions in Logo give you the full power to make complex calculations. All math functions are implemented as actions, just like math operators. Of course, all functions expect argument only from the right.

Some of the actions represent mathematical calculations which you have already studied in school. Others may be new for you. If you find unknown for you function, please consult a mathematical source about details.

6.2.1 Sign functions (**abs**, **neg**, **sign**)

The sign functions in Logo are used when you need to examine or change the sign of a number. If you say the function **abs**, then Logo will remove the sign of

a value and convert it into a positive number.



The other function is **neg** and it negates a number - if it is positive, after applying the action the result will be negative, and vice versa. Actually, negating a value is the same as subtracting it from 0.

```
print abs 20 ; 20
print abs -15 ; 15
print neg 10 ; -10
print neg -7 ; 7
```

Both **abs** and **neg** return the same numbers, but eventually with changed signs. In order to see what the sign of a value is, you may ask the function **sign**. It will return **-1** if a value is negative, **+1** if it is positive, and **0** if it is zero.

```
print sign -3 ; -1
print sign 0 ; 0
print sign 4 ; 1
```

6.2.2 Exponential functions (**sqrt**, **exp**, **logn**)

Exponential functions deal with powers of numbers. There are three functions which Logo learns from **logo.e!i** file. **Sqrt** is used to calculate squares of numbers, **exp** is used to raise **e** to a given power (**e** is an important mathematical constant which is approximately **2.718** - a more precise value is calculated by **exp 1**). The third function is **logn** and it is the reverse function for **exp**.

```

print sqrt 3      ; 9
print exp 1       ; 2. 71828182845905
print logn 10    ; 2. 30258509299405

```

6.2.3 Rounding functions (**trunc**, **round**)

It is often needed to calculate an integer result, but the computer gives a non-integer one. In such cases you may consider asking Logo to modify the number in a way defined by you. The easiest way is just to ask to truncate a number, leaving out its fractional part. This is accomplished by the function **trunc**. The other way is to define exactly how precise number you need. For such cases you may use **round**.

```

print trunc 3.2      ; 3
print trunc -4.8     ; -4
print round 319.425 0 ; 319
print round 319.425 2 ; 319.43
print round 319.425 -1 ; 320

```

6.2.4 Trigonometric functions (**sin**, **cos**, **tan**, **cotan**)

All four trigonometric functions: **sin**, **cos**, **tan**, **cotan** - are supported by Logo. They expect exactly one argument from the right which must be a value in degrees (not in radians or any other unit).

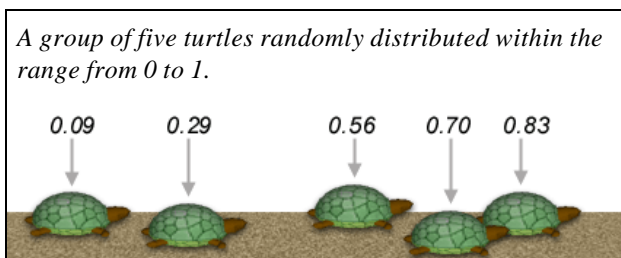
```

print "' sin(30)=' sin 30
print "' cos(30)=' cos 30
print "' tan(30)=' tan 30
print "' cotan(30)=' cotan 30

```

6.2.5 Random function

Except for the functions described above, there is one unique. For many simulations you would like to have random numbers. There is such a function in Logo which returns a random number whenever you call it. This function is



random and can be used with or without an argument. If there is no argument **random** returns a random number from 0 to 1. If there is argument then the result is a random number from 0 to that argument. In both cases

the random number can be equal to the 0, but is always less than the upper limit of the possible range.

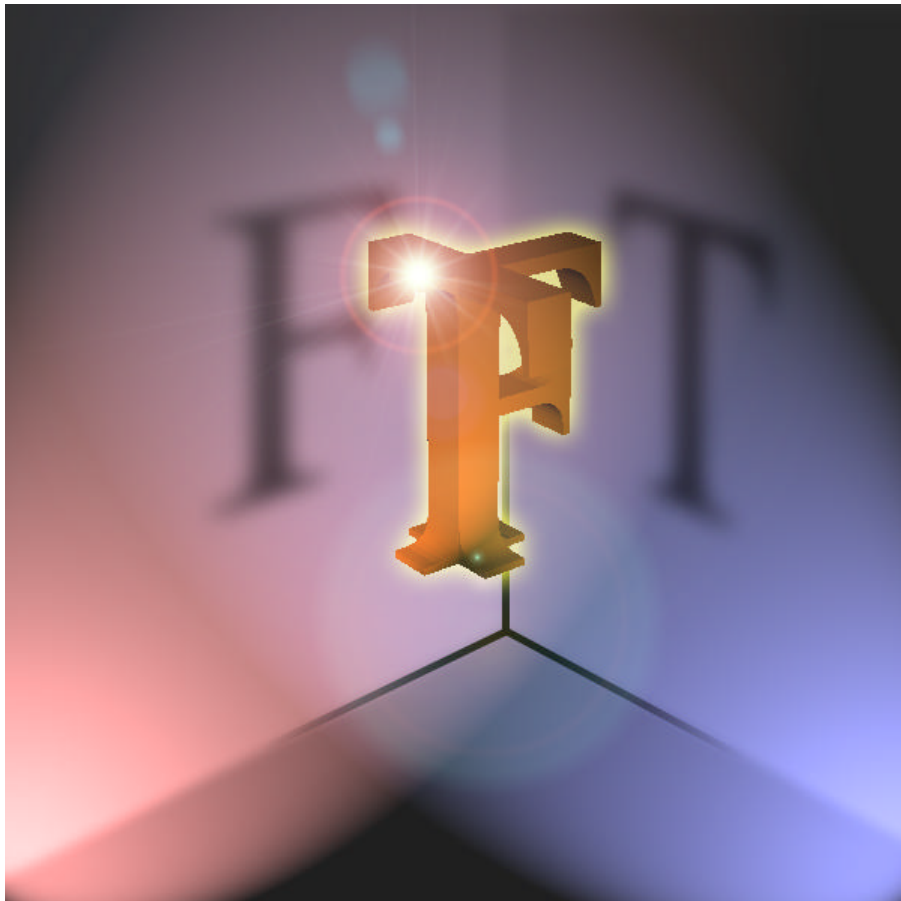
```

print random      ; 0. 346296521136537
print random      ; 0. 175248484592885
print random 100  ; 98. 8024020334706
print random 100  ; 60. 4138494469225

```

True Or False

Both Of Them Are Projections Of Reality



7. Logo and Logics

The relation of Logo to Logics is expressed in the ability to write programs containing compound logical expressions. These expressions are different from all other expressions because they can be evaluated either as true (valid) or false (invalid).

7.1 Operators for comparing (=, <, >, <=, >=, <>)

When you ask Logo to compare two values, you say what action to use in order to compare. There are 6 different ways to compare and there is an operator for each of them. You can use the operators to find whether two values are equal (=) or not equal (<>), or whether a value is less (<), greater (>), less-or-equal (<=) or greater-or-equal (>=) to another value.

Values can be numbers, words and lists as shown below:

```
print 4 = 5 ; false
print 41 > 12 ; true
print "mouse < "mi ce ; false
print [h a t s] >= [c a t s] ; true
print [h a t s] <> [c a t s] ; true
```

Numbers are compared as numbers. Words are compared alphabetically (in the way they are ordered in a dictionary) but capital letters always come first, thus A...Z section is before a...z. Lists are compared in a more complex way. When you ask Logo to compare two lists, it start to compare their elements one by one. When it reaches a pair of elements that are not equal, their relation defines the comparing of both lists. If one list is shorter than another and all its elements are the same as those of another one, than the shorter list is smaller.

In all cases when you compare incomparable values, for example you ask whether 5 is equal to [1 2 3], Logo will return **false** as a result. Otherwise it will compare the values and return either **true** or **false**.

7.2 Logical operators (and, or, not)

Logical operators are those that deal with the logical values **true** and **false**, rather than those that are not illogical. The operator **and** returns the word **true** only if its both arguments are **true**. The operator **or** returns the word **true** when at least one of its two arguments is **true**. The third operator **not** returns **false** if its only argument is **true**, otherwise it returns **true**.

A good place to use logical operators is when you want to form a complex condition in **if** or **while**.

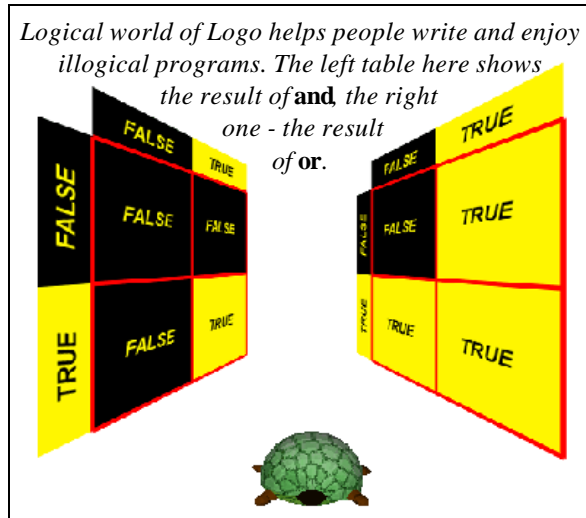
```
if (5<: a) and (: a<10)
  [ print :a "' is between 5 and 10' ]
```

```

if (5<=: a) or (: a>=10)
  [ print :a "' is not between 5 and 10' ]

if not (5<: a)
  [ print :a "' is less or equal to 5' ]

```



When using logical operators pay attention on the fact that Logo will not understand if you say it:

```
print :a > :b and :c
```

Well, it will understand this but most likely it will not be what you possibly meant. So you must clearly say to Logo what exactly you mean:

```

print (:a > :b) and (:a > :c)
print (:a > :b) and :c
print :a > (:a and :c)

```