LYNX™

LYNX VOCABULARY AND SYNTAX

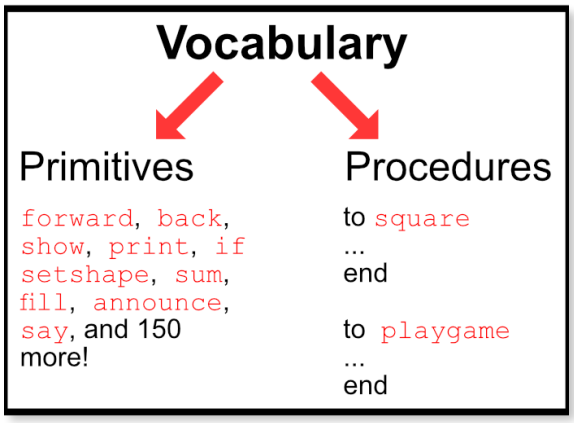
ALL ABOUT PRIMITIVES, PUNCTUATION, WORDS, NUMBERS AND LISTS

Contents

LYNX'S VOCABULARY

The Lynx vocabulary comprises ALL the words that Lynx can understand.

PRIMITIVES AND PROCEDURES



Primitives are the vocabulary built-in for Lynx. These words are always present and available for programming: `forward`, `xcor`, `setshape`, `print` are all primitives.

Lynx's vocabulary also includes some "dynamic" primitives that exist only in relation to objects that you create. For example, when you create a turtle "t1", the primitive "`t1,`" also exists. When you create a text box "Text1", the primitives "`text1,`" and the primitive "`text1`" also exist.

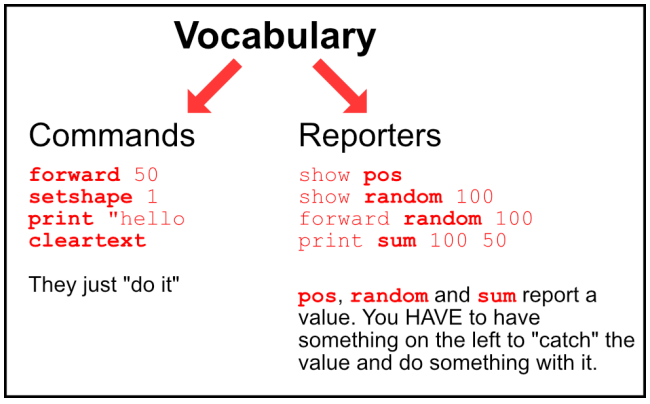
A procedure is a group of instructions to which you give a name. That name is added to Lynx's vocabulary, but only while your project is open. It will not be available in another project, unless you recreate that procedure. `Square`, for example, could be a procedure that draws a square. It is certainly *not* in Lynx' *built-in* vocabulary.

COMMANDS AND REPORTERS

All these primitives and procedures can be of two types: commands and reporters.

A command just "does" something. For example, `forward` will make the turtle move forward - `forward 50` is a valid and complete instruction. Same for `setshape 1`, `right 90`, `print [Hi there!]`.

A reporter "reports" or "returns" something. But YOU HAVE to do something with what is reported. `Pos`, for example, reports the turtle's current position - but just by itself, `pos` is not a complete and valid instruction. You HAVE TO "catch" the result and do something with it. For example, `show pos` is a valid and complete instruction. `Pos` reports the position, and `show` "catches" that value to display it in the Command center.



The examples above are primitives. Procedures also are commands or reporters.

This procedure is a command. It does something, and it does **not** report anything:

```
to drawsquare
pendown
repeat 4 [forward 50 right 90]
end
```

You can use it as the first word of an instruction in the Command Centre:

```
drawsquare
```
This simply draws a square on the page.

This procedure is a reporter. It reports something:

```
to squarenumber :x
output :x * :x
end
```

You **can't** use it as the first word of an instruction in the Command Centre. If you do, Lynx displays an error message:

```
squarenumber 12
I don't know what to do with 144
```

Because it is a reporter, its value must be "caught" or "used" by another primitive or procedure to its left:

```
show squarenumber 12
144
forward squarenumber 12
```
The result is printed in the Command Centre.

The turtle moves 144 steps forward.

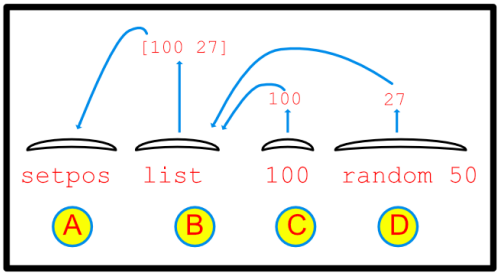
INSTRUCTIONS

A Lynx instruction can comprise one word, or many words (built-in primitives or procedures defined by you). However, the first word of an instruction MUST be a command, not a reporter. Here's why:

Say you have a turtle and a text box on the page. Try these instructions in the Command centre:

`forward 50`	No problem, the first word (`forward`) is a command.
`print "hello`	Good again. `Print` is a command.
`pos` `I don't know what to do with 0 0`	Here, you got an error message, because `pos` is a reporter. It reports the turtle's current position, and you did not say what to do with that value.
`show pos` `0 0`	That's better. `Show` is a command. It "catches" the value reported by `pos`, and it shows it in the Command centre.
`random 100` `I don't know what to do with 67`	Again, an error message. `Random` is a reporter, it reports a value between 0 and 99 in this example. You don't say what to do with the value (your value will be different).
`forward random 100`	Good! `Random` reports a value, and `forward` "catches" it because it needs a value, and it is happy to use the number reported by `random`. The turtle will move forward a random number of steps.

In a way, you will write your instructions left to right, but at the end, you should read it right to left to see if it is valid. ALL reporters report to the left. Here is an example:



In this example, you obviously write the instruction from left to right (A B C D) but Lynx reads it right to left (D C B A): `random 50` reports a random number (`27`) and "throws" it to the left. The number `100` can do nothing by itself, so it throws itself to the left. Fortunately, `list` will "catch" the numbers 100 and 27 because it needs two "things" to create a list - it creates the list `[100 27]` and throws that to the left as well. Finally, `setpos` needs a list of two numbers to set the turtle's position, and it will gladly "catch" what was thrown by `list`. Everybody's happy!

NUMBERS, WORDS AND LISTS

Before talking about numbers, words and lists, it is worth mentioning that Lynx has three primitives talking about the "nature of things". They are `number?`, `word?` and `list?`. Just read these examples and you will understand:

```
show number? 33.3
true
show number? pi
true
show word? 44
true
show word? [ ]
false
show list? [one [two three] four]
true
```

Yes, numbers are also words, but not the other way around.

NUMBERS

Numbers are numbers... But there are a few things you ought to know about how Lynx sees numbers. Say you have a text box on your page. Try these instructions:

`print 50`	This prints `50` in the text box.
`print 0.50`	This prints `0.5` in the text box.
`print .50`	This prints `0.5` in the text box. The leading `0` is optional.
`print -50`	This prints `-50` in the text box.
`print - 50`	This results in an error message. No space between the minus sign and the number.
`print minus 50`	This prints `-50` in the text box.
`print 5e3`	This prints `5000` in the text box.

Note that numbers can be seen as a special kind of word. Word primitives will work on numbers:

```
show first 357
3
```

The symbol = works on words, numbers and lists:

```
show 'a' = first [a b c]
true
```

But some symbols only work with numbers:

```
show pi > 3
true
```

WORDS

All the primitives and procedures are obviously words. In fact, Lynx will interpret any word by itself (not preceded by a quotation mark, a comma, a colon) as a primitive or a procedure, and it will try to execute it.

Say you have a text box on your page. Type this instructions in the Command centre:

`hello`	This creates an error message (`I don't know how to hello`) because a word by itself is always interpreted as "something to execute". If `hello` is not a procedure (it is certainly not a primitive), Lynx will complain.

In the following examples, we use the double quotation mark to indicate that the word is... just a word, not something to execute. See Punctuation and separators, further down, for more options. Again, try these instructions:

`print "hello`	This prints `hello` in the text box.
`print first "hello`	This prints `h` in the text box, the first element of a word is a letter.
`print last "hello`	This prints `o` in the text box.
`print butfirst "hello`	This prints `ello` in the text box, the whole word, minus the first character.

LISTS

A list is a number of "things" enclosed within square brackets. The "things" can be words, numbers, even other lists. Again, assuming you have a text box on the page, try these in the Command centre:

`print [hello]`	This prints `hello` in the text box. The list contains one word.
`print [hello there]`	The list contains two word.
`print [a b 22 c]`	The list contains four elements.
`print [a [x y z] b c]`	The list contains three elements, the second element is a list.

You can build lists and extract elements from lists:

`print first [hello]`	This prints `hello` in the text box. Hello is the first (and only) element in the list.
`print last [hello there]`	This prints `there` in the text box.
`print butfirst [a b 22 c]`	The prints the entire list, minus the first element.
`print item 2 [a [x y z] b c]`	This prints the second item of the list: `[x y z]`.
`print empty? []`	This prints `true` in the text box.

DELIMITERS

In Lynx coding, delimiters include the space, the parenthesis, the single and the double quotation mark, the vertical bar, and the square brackets.

SPACES

The space is very important in many places. Sometimes it's important to have one, and sometimes it's important to NOT have any.

First, spaces are delimiters for words. That's why it's important to use single words when creating procedures, naming turtles, creating variables, etc.

	GOOD	BAD	EXAMPLE

PROCEDURES	big_square BigSquare big.square	big square big-square	`to t1_click`
			`big_square`
			`end`

TURTLES	blue_boat BlueBoat blue.boat	blue boat blue-boat	`blue_boat,` `setheading 90`
VARIABLES	prev_score PrevScore prev.score	prev score	`make "prev_score 0` `show :prev_score`

Naturally, you need a space between each command in an instruction (more than one space is OK too!). In this example, note that there is also a space between the command back and its input - `back100` would not work:

```
if xcor > 100 [penup back 100]
```

and you need a space between each item in a list. In this example, `make` is used to create the variable `friends`. The value of the variable is a list of four names.

```
make "friends [Kim Lea Sam Luc]
```

Spaces are important in math notation also. There is a section about that further down.

DOUBLE QUOTATION MARK

The double quotation mark indicates that the word is "just a word", not a primitive nor a procedure, nor a variable. It is placed ahead of the word, not after. It is used to make "just one word". In these examples:

```
print hello
make friends [Kim Lea Sam Luc]
```

`hello` and `friends` are words without any mark. Lynx will try to execute them as a primitive or a procedure. This will generate an error message: `I don't know how to hello`.

This will work:

```
print "hello
make  "friends [Kim Lea Sam Luc]
```

This prints hello in a text box.
This creates a variable named friends and sets its value as a list of four names.

In the last example, you see that inside a list, words are not (and should not) be quoted.

DELIMITERS

SINGLE QUOTATION MARK (AND VERTICAL BAR)

The single quotation mark, used on **both sides** of a word, also indicates that everything between them is just a word. However, that word can contain spaces, and contrarily to lists, multiple spaces are counted as multiple spaces.

Assuming you have a text box on your page, try these in the Command centre:

`print 'hello there'`	This prints hello there in the text box.
`print 'hello  there'`	The four spaces are also printed.
`print count 'hello  there'`	This prints 13 in the text box. This is the number of characters in this long word (5+3+5).
`print [hello  there]`	This will also print hello there in the text box, but the three spaces have been "trimmed" to one.

See the difference?

- `'hello there'` is a long word with eleven elements (or characters).
- `[hello there]` is a list with two elements (two words).

Vertical bars can also be used to encase multiple words in the same manner. You will have to use the double quotation mark on the left, as such:

```
print "|hello there|
```

This is the same thing as:

```
print 'hello there'
```

BRACKETS

Brackets are used to create lists. See LISTS above. A list can be "just a list of things", or "a list of instructions".

LIST OF THINGS

Brackets can be used to surround words and other lists - really anything. Here is an example:

```
make "data [cats dogs birds [reptiles and amphibians] fish]
print first :data
cats
```

LIST OF INSTRUCTIONS

A list of instructions is also enclosed within square brackets. However, this list must contain only things that Lynx can execute. There is no difference for you when you type such a list... It's just a list that is used with `repeat`, `if`, `ifelse`, `forever`. Here a few examples:

```
repeat 4 [forward 50 right 90]
if xcor > 100 [penup back 100 pendown]
```

```
forever [forward 2 setshape 1 wait 3 forward 2 setshape 2 wait 3]
```

PUNCTUATION - COLON

A colon, placed in front of a word, indicates that this is the name of a variable, not just a word, not a primitive, not a procedure. When you type `print :friends`, you mean to print the value of the variable `friends`.

Consider the name of a variable as a container. The primitive `thing` can also be used to find out what is the "thing" in the container:

```
print :friends
```
is the same as:
```
print thing "friends
```

ANYWHERE

Once a variable is defined, the name of the variable, preceded by a colon, returns its value. Try this in the Command centre. This example represents what's called a global variable, because it maintains its value "all the time, no matter what":

```
make "friends [kim sam joe cat lea]
```
That's how you create a global variable. Notice the double quotation mark.

```
show :friends
```
That's how you get the value of a variable. Notice the colon.

```
show first :friends
kim
show empty? :friends
false
```

So generally it's QUOTE to create the variable, COLON to obtain its value.

ON A PROCEDURE TITLE LINE

A variable name appearing on a procedure's title line makes two things:

1. This procedure will now require an input, just like the primitive `forward` does. In this example, you cannot execute the procedure `square` without providing a value for `:size`.

```
square
square needs more inputs in square
square 20
square 100
```

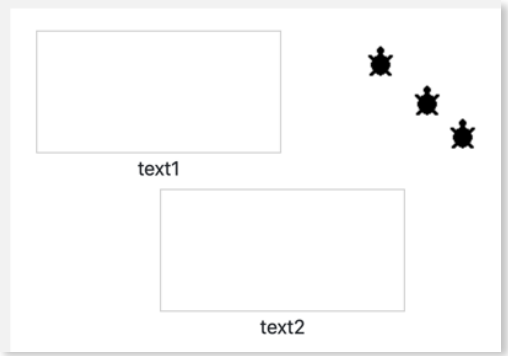
2. The name of the input (`:size` in this example) can be used anywhere in the procedure.

```
spiral 10 90
```

PUNCTUATION - COMMA

In some cases, the comma has a special meaning for Lynx: placed after the name of a turtle or a text box, it indicates that you "want to talk to it", just like you would say, IRL:
Sam, come here please! (notice the comma after Sam). This special comma feature applies only to turtle names and text box names. In other circumstances, a comma is just a plain character (you can define a procedure named `Me,Pete,Kim`).

Say you have three turtles and two text boxes on the page. The turtle `t1` has been renamed `Roxy` (right-click on a turtle to rename it. Use a single word, no space).



Try these instructions in the Command centre. Notice the comma following the name of the turtle or the text box (no space):

```
t1, rt 90
text1, print "hello
text2, print "there
text1, cleartext
```

If you want to talk to more than one turtle at a time, use the command `talkto` instead of the comma feature:

```
talkto [t2 roxy]
left 90
```

No comma here!

`Talkto` can only used with turtles - not with text boxes.

PARSING MATH INSTRUCTIONS

Math instructions can be made using infix symbols and prefix primitives:

Infix symbols (symboles that go between numbers): `*`, `/`, `+`, `-`, `<`, `>`, `=`

The symbol "=" is a special case, it can be used with numbers, words and lists.

Prefix math operators (math operators that go before numbers): `sum`, `difference`, `product`, `quotient`, `remainder`, `sin`, `cos`...

In many cases, you don't have to put spaces before and after math operators:

```
show 2+3
show 2 + 3
show 2+ 3
```

are the same, but

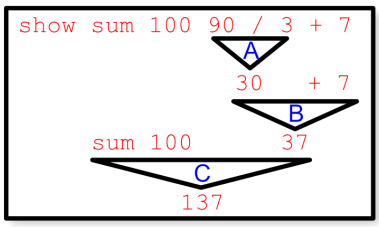
`show 2 +3` isn't... This is not a math operation, this is a set of two numbers (2 and +3). So... to avoid any issue with spacing, Lynx recommends that you always use spaces before and after math operators; it is also much easier to read:

```
show (:size + 100) / 8
```

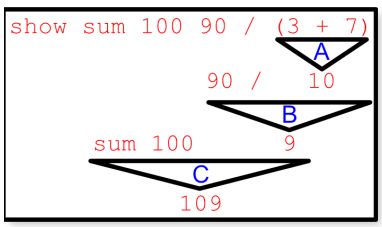
The order of priority for parsing operators is as follows:

1. `*`, `/`
2. `+`, `-`
3. `<`, `>`, `=`
4. Prefix primitives (`sum`, `difference`, `product`...)

Here is a complex example to illustrate this:



You can "force" a different order using parenthesis. Whatever is enclosed in parenthesis is evaluated before anything else. When in doubt, use parenthesis. This also makes your code easier to read and debug.



Note: Lynx will interpret `12.5` and `12,5` as the same value, but it will print them using the decimal point (`12.5`)