YOLOL is a programming language used to control and manage electrical devices.
The code is written on lines in YOLOL chips which are then inserted into chip sockets, that read and relay their messages.
The programming language enables the programming and controlling of almost any device within the known universe.

Basic information

How it works

The code is written to and executed from programmable chips, and can be used to both monitor and control electrical devices connected to a data network.
Lines of code are executed in sequence from top to bottom, repeating the cycle of the chip after the last line has been executed, unless the script includes programmed instructions for specific line changes or stopping the execution completely.

To put it simply:

1. Code execution starts from line 1
2. After reading line 1, it proceeds to the next line based on the chip's time interval
3. The process is then repeated for the lines 2, 3, 4... etc.
4. The chip will begin executing line 1 again after the last line has been executed (unless the last line contains a goto statement or execution has been paused)

So blank lines still use 0.2 seconds and can be used as a brief execution delay. (lines with only a comment are effectively the same as a blank line.)

Limitations

- Lines take 0.2 seconds to execute.
- A line can contain a maximum of 70 characters (comments and spaces included).*

- Some functions only work on specific YOLOL chips.

*Note that because of this some examples shown below might not work in-game.

Command references

Case insensitive

The programming language is fully case **insensitive**.
This means that the following two example scripts function identically to each other:

```
if ButtonState == 1 then DoorState = 1 end
```

```
IF buttonstate == 1 THEN doorstate = 1 END
```

- Both scripts set the **doorstate** into 1, if **buttonstate** value is 1.
- The characters in the programming language can be written in either lowercase or uppercase letters.
  - They are still parsed as case insensitive.
  - This way it's possible to have the code look a bit more organized.

Variables

- The variables in the programming language are weakly typed (don't enforce type validity), and support two data types: **Fixed-point decimals** (up to 0.001 precision) and **Strings** (up to 1024 characters long).
  - To put it simply, the variables can either be introduced as strings or numbers, ignoring the earlier variable type if the previous type is not identical, without causing an error.
- Each variable is always of a single type, though it will be implicitly converted when required.
- The default value of an uninitialized variable is 0, and null values are not supported.
- True/False are numerical values of non-0 and 0.
  - True != 0
  - False == 0

Assigning a value to a variable always converts the variable to the newly assigned value's type.

Example:

```
ultimateAutopilot= 128.643
```

- This results in the variable **ultimateAutopilot** containing a numeric value of 128.643

```
ultimateAutopilot= "Error prone"
```

- This results in the variable **ultimateAutopilot** to be a string variable "*Error prone*", and numeric value of 128.643 is removed.

Decimals

Numeric values in the programming language are 64-bit fixed-point decimals.
The variables hold decimal numbers up to three decimal accuracy.
As a result, the maximum value range (even during operations) is [-9223372036854775.808, 9223372036854775.807]

```
pieVariable= 3.142
```

- The above script assigns a numeric value of 3.142 to the variable **pieVariable**.
  - Supplying more precise values than the variables can store works, but doesn't affect the end result.

```
notPieVariable= 0.5772156649
```

- The above script attempts to assign a numeric value of 0.5772156649 to the variable **notPieVariable**.
- The end result however is notPieVariable == 0.577
  - Here, the more precise values are cut, leaving only three decimals behind.

Strings

To specify a string literal in the programming language, the desired string value must be surrounded with double quotation marks.

```
badRobots= "saltberia"
```

- This script assigns the string value of "*saltberia*" to the variable **badRobots**.

Device fields / External variables

External variables and device fields can be used in the programming language with the following syntax:

- **:variableName**
  - **variableName** being the configured device field id.

A colon prefix **:** is used to tell the script that an external variable is being accessed, instead of using one that may or may not be declared or used in the script.
A programmable chip that is connected to a device has access to all the devices in the same network.
It can then modify and listen to any device fields it has access to.

```
if :ButtonState == 1 then :DoorState = 1 end
```

- The script above will send the value of 1 to any devices listening to the device field **DoorState** if the value of **ButtonState** is 1 in the data network.

Naming Limitations

Currently variables containing keywords such as **if** or **end** can be parsed incorrectly, and must be avoided.

Memory Chips and Relays

Memory Chips provide 10 device fields that aren't tied to the function of devices and can be used to share variables and their values between multiple YOLOL Chips.
In a Memory Relay they can also be used to copy data from one device field to another, which allows

- separation of networks e.g. for keeping the data networks less noisy and smaller
- a combination of shortened variable names in yolol and long variable names in displays e.g. for cockpit interfaces

Operators and commands

Note that the available operators may be limited by the type of the programmable chip.
Basic chips have a limited selection of functions while more advanced ones can perform more complex operations natively.

Basic arithmetic and assignment operators

Operation	Numeric operation	String operation	Chip availability
A + B	Addition	String A is appended by String B.	All
A - B	Subtraction	The last appearance of String B in String A is removed from String A.	All
A * B	Multiplication	Runtime error. The rest of the line is skipped.	All
A / B	Division	Runtime error. The rest of the line is skipped.	All
A ++	PostIncrement (A=A+1)	Appends a space to String A. Evaluates to the original value.	All
A --	PostDecrement (A=A-1)	Removes the last character of the string. Results in runtime error when trying to remove "". Evaluates to the original value.	All
++ A	PreIncrement (A=A+1)	Appends a space to String A. Evaluates to the modified value.	All
-- A	PreDecrement (A=A-1)	Removes the last character of the string. Results in runtime error when trying to remove "". Evaluates to the modified value.	All
A = B	Assignment (Variable A is set to the value of variable B)	Assignment	All
A += B	Addition-assignment (A=A+B)	A is assigned the value of string-operation A+B	All
A -= B	Subtraction-assignment (A=A-B)	A is assigned the value of string-operation A-B	All

A *= B	Multiplication-assignment (A=A*B)	Runtime error. The rest of the line is skipped.	All
A /= B	Division-assignment (A=A/B)	Runtime error. The rest of the line is skipped.	All
A ^= B	Exponentiation-assignment (A=A^B)	Runtime error. The rest of the line is skipped.	Advanced, Professional
A %= B	Modulo-assignment (A=A%B)	Runtime error. The rest of the line is skipped.	Advanced, Professional
A ^ B	Exponentiation	Runtime error. The rest of the line is skipped.	Advanced, Professional
A % B	Modulo	Runtime error. The rest of the line is skipped.	Advanced, Professional
ABS A	Modulus (absol value) (A=A if A>=0, else A=-A)	Runtime error. The rest of the line is skipped.	Advanced, Professional
A!	Factorial	Runtime error. The rest of the line is skipped.	Advanced, Professional
SQRT A	Square root of A	Runtime error. The rest of the line is skipped.	Advanced, Professional
SIN A	Sine of A (degrees)	Runtime error. The rest of the line is skipped.	Professional
COS A	Cosine of A (degrees)	Runtime error. The rest of the line is skipped.	Professional
TAN A	Tangent of A (degrees)	Runtime error. The rest of the line is skipped.	Professional
ASIN A	Inverse sine of A (degrees)	Runtime error. The rest of the line is skipped.	Professional
ACOS A	Inverse cosine of A (degrees)	Runtime error. The rest of the line is skipped.	Professional
ATAN A	Inverse tangent of A (degrees)	Runtime error. The rest of the line is skipped.	Professional

Logical operators

Logical operators are checks that identify if the statement is true or false.
All logical operations return either **"0 for False"** or **"1 for True"**. The **NOT**, **AND**, and **OR** keywords consider 0 to be falsy and anything not 0 to be truthy.

Operation	Numeric operation	String operation	Chip availability
A < B	Less than	returns 1 if String A is first in alphabetical order, returns 0 if not.	All
A > B	Greater than	returns 0 if String A is first in alphabetical order, returns 1 if not.	All
A <= B	Less than or equal to	returns 1 if String A is first in alphabetical order or identical to String B, returns 0 if not.	All
A >= B	Greater than or equal to	returns 0 if String A is first in alphabetical order or identical to String B, returns 1 if not.	All

A != B	Not equal to	returns 1 if String A is not equal to String B, 0 if it is.	All
A == B	Equal to	returns 1 if String A is equal to String B, 0 if not.	All
NOT A	Not	Returns 1 if A is 0, otherwise returns 0.	All
A AND B	And	Returns 1 if neither A nor B are 0, otherwise returns 0.	All
A OR B	Or	Returns 1 if either A or B is not 0, otherwise returns 0.	All

Mixing variable types in operations

Mixing variable types in an operation handles the operation using all parameters as *strings*.

```
previouslyNumber= "10" + 15
```

- The above script results in **previouslyNumber** containing the string value "1015".
  - Note that the involved parameters themselves don't change types, their values are just cast as strings for the purpose of the operation:

```
purelyNumber = 15
purelyString = "10" + purelyNumber
```

- When this script has executed, **purelyString** contains the string value of "1015", while **purelyNumber** still contains the numeric value of 15.

Goto

Goto syntax is used when the normal script reading order from 1->20 is not desired, or needs to be altered.

Goto is used with the following syntax:

- **goto lineNumber**
  - lineNumber is the line which this command will take the script execution.
  - Any remaining script that is on the same line after the goto-command will not be executed.
    - using if statements before goto ignores goto syntax, assuming the if-statement is false
  - Multiple goto commands can be added on the same line using conditionals, as **False** goto commands are skipped.
  - Numeric values outside the [1,20] range are clamped to this range.
  - Non-integer values are floored.
  - String values will result in a Runtime Error.

```
if variable == 5 then goto 4 end goto 6
```

The script above will go to line number 4, if **variable** has a value of 5.
Otherwise it will go to line number 6. Numerical operations can also be done "inside" the goto, e.g.

```
goto 4+1
```

If-else conditional

If-else statements are used to branch out the script into different paths.
They use the following syntax:

- **if *condition* then *statement* else *statement* end**
  - Condition is a statement that results in a numeric value (where 0 is parsed as False, anything else as True), and statements are pieces of script that are run.
  - All If-else conditional stations must have **end** syntax written after statement is complete.
  - The statement has to be on one line. The if, then, else and end cannot be on different lines.

If can be used to branch script execution into two possible outcomes temporarily based on variable value(s).
Example:

```
if variable != 2 then endResult = 3 else endResult = 4 end
```

- This script sets the value of **endResult** to 3 if **variable** does not have the value of 2.
- If **variable's** value is 2, **endResult** is set to the value of 4.

Note that the else statement -part can be left out if not needed.

Example:

```
if variable != 2 then endResult = 3 end
```

- This script only sets the value of 3 to **endResult** if **variable** does not have a value of 2, and doesn't do anything else.

Nesting if statements

It is possible to place if-conditionals inside the true/false statement blocks to achieve further branching of execution.
Example:

```
if variable == 0 then endResult = 1 else if variable == 1 then endResult = 2 end end
```

- This script sets **endResult** to 1 if **variable** equals 0.
- If **variable** doesn't equal 0, but it equals 1, **endResult** is set to 2.

Example:

```
if variable == 0 then if endResult == 1 then endResult = 2 end else endResult = 1 end
```

- This script sets **endResult** to 2 if **variable** has a value of 0, and **endResult** equals to 1.

- Otherwise it sets **endResult** to 1.

Comments

Comments are useful when writing code that is used by a lot of programmers.
Note that comments also use up space from the pre-determined 70 character line limit and are not excluded from it.

Commenting is used with the following syntax:

- // **text**
  - Text can be any single-line set of characters.

```
// This is a comment. It will explain how other lines of script work.
```

- An example of a possible comment syntax

Order of Operations

Operations are conducted in the following order, when operators have the same precidence, left to right.
Where a line has multiple statements, they are executed left to right.

Operators	Comments
++ --	
!	
operators	sqrt, abs, sin etc.
-	negate
^	
*/%	
< > == != <= >=	Surprise!
+-	
not (logical negation)	
or	
and	

Errors

There are two types of errors that can happen with the programming language.

1. Syntax errors
2. Runtime errors

- Syntax errors come from invalid and unparseable script and will result in the whole line not being executed.
- Runtime errors are only catchable while the script is being executed. They result in the execution of the line being interrupted, but any effects until the error will remain.

Known Bugs/Unintended Behavior

This is a list of known problems regarding yolol: