Operator	Operation	Arguments	Example	JS conversion	Result
Common Operator					
+	addition	2	**3 5 +**	3 + 5	8
-	subtraction. If the stack contains A B -, then the calculation is A - B.	2	**(L:Value) 90 -**	**(L:Value)** - 90	The local value minus 90.
/	division. If the stack contains A B /, then the calculation is A / B.	2	**5 2 /**	`5 / 2`	2.5
*	multiplication	2	**pi 2 ***	`Math.PI * 2`	2 pi
%	taking modulo	1	**5.3 5 %**	`5.3 % 5`	0.3
++	increment	1	**4 ++**	`4 + 1`	5
--	decrement	1	**4 --**	`4 - 1`	3
/-/ **neg**	negates a number	1	**4 /-/**	`4 * (-1)`	-4
Comparison Operators					
'==	true if equal	2	**(L:Value) 0 == if{ A }**	**(L:Value)** `== 0 ? A : 0`	Operation A is carried out if Value is 0.
!=	true if not equal	2	**(L:Value) 0 != if{ A }**	**(L:Value)** `!= 0 ? A : 0`	Operation A is carried out if Value is not 0.
>	true if greater than	2	**(L:Value1) (L:Value2) > if{ A } els{ B }**	**(L:Value1)** `>` **(L:Value2)** `? A : B`	If Value1 is greater than Value2, operation A is carried out, otherwise operation B is carried out.
<	true if less	2	**(L:Value1) (L:Value2) < if{ A } els{ B }**	**(L:Value1)** `<` **(L:Value2)** `? A : B`	If Value1 is less than Value2, operation A is carried out, otherwise operation B is carried out.
>=	true greater than or equal	2	**(L:Value1) (L:Value2) >= if{ A } els{ B }**	**(L:Value1)** `>=` **(L:Value2)** `? A : B`	If Value1 is greater than or equal to Value2, operation A is carried out, otherwise operation B is carried out.
<=	true if less than or equal	2	**(L:Value1) (L:Value2) <= if{ A } els{ B }**	**(L:Value1)** `<=` **(L:Value2)** `? A : B`	If Value1 is less than or equal to Value2, operation A is carried out, otherwise operation B is carried out.
?	The third operand determines whether the first (True) or second (False) is selected.	3	**A B True ?**	`True ?` **A** : **B**	This evaluates to A.

Bit Operators					
&	bitwise AND	2	**5 3 &**	5 & 3	1
|	bitwise OR	2	**5 3 |**	5 | 3	7
^	bitwise XOR	2	**5 3 ^**	5 ^ 3	6
~	bitwise NOT	1	**5 ~**	~5	-6
>>	shift right operand number of bits	2	**5 3 >>**	`5 >> 3`	0
<<	shift left operand number of bits	2	**5 3 <<**	`5 << 3`	40
Logical Operators					
!, NOT	NOT	1	**(L:Local) ! (>L:Local)**	**!(L:Local)**	Toggles the variable Local
&&, AND	AND	2	**(L:Local) 0xFF00 && (>L:Local)**	**(L:Local)** `&& 0xFF00`	The variable Local is ANDed with hex 0xFF00
||, OR	OR	2	**(L:Local) 07777 OR (>L:Local)**	**(L:Local)** `|| 07777`	The variable Local is ORed with octal 7777.
Numerical Operators					
abs	Absolute value	1	**-5 abs**	Math.abs(-5)	5
int **flr**	Calculates nearest integer number which is less than the source number	1	**5.98 flr**	Math.floor(5.98)	5
rng	Range; returns True if the third operand lies between values one and two.	3	**4 7 6 rng**	`4 <= 6 && 6 <= 7`	True
cos	Cosine (input in radians)	1	**pi cos**	`Math.cos(Math.PI)`	-1
lg	Logarithm to base 10	1	**10 lg**	Math.log(10) / Math.log(10)	1
min	Minimum	2	**5 2 min**	`Math.min( 5, 2 )`	2
sin	Sine (input in radians)	1	**pi sin**	`Math.sin(Math.PI)`	0
acos	Arc cosine (returns radians)	1	**pi acos**	`Math.acos(Math.PI)`	
ctg	cotangent (input in radians)	1	**pi ctg**	`1 / Math.tan(Math.PI)`	
ln	Natural logarithm	1	**2.718282 ln**	`Math.log(2.718282)`	1
sqr	Square	1	**5 sqr**	`5 ** 2`	25

asin	arc sine	1	**pi asin**	`Math.asin(Math.PI)`	
eps	Floating-point relative accuracy	1	**1 eps**	`Number.EPSILON * 1`	2^(-52)
log	Logarithm of operand one, to the base of operand two.	2	**8 2 log**	`Math.log(8) / Math.log(2)`	3
pi	Pi = 3.14159; puts pi on the stack	0	**pi**	Math.PI	3.14159
sqrt	Square root	1	**25 sqrt**	Math.sqrt(25)	5
atg2	arc tangent with two inputs (input in radians)	2	**A B** atg2	`Math.atan2( A, B )`	
exp	Exponent; e to the power of the operand	1	**1 exp**	`Math.exp(1)`	2.718282
max	Maximum	2	**5 2 max**	Math.max(5, 2)	5
pow	Power of; the first value to the power of the second	2	**2 5 pow**	`Math.pow( 2, 5 )`	32
tg	Tangent (input in radians)	1	**pi tg**	`Math.tan(Math.PI)`	0
atg	arc tangent with one input	1	**pi atg**	Math.atan(Math.PI)	
Special Operators					
div	Divides integers; its result is always an integer	2	**5 3 div**	`Math.floor( parseInt(5) / parseInt(3) )`	1
ceil	Calculates nearest integer number which is bigger than the source	1	**4.3 ceil**	`Math.ceil(4.3)`	5
near	Calculates the nearest integer number, rounding .5 up.	1	**4.5 near**	`Math.round(4.5)`	5
dnor **d360** **rdeg**	Normalizes an angle expressed in degrees. The result in an value between 0 and 360.	1	**-15 dnor**		345
rddg	Converts radians to degrees	1	**pi rddg**	`(Math.PI)* 180/Math.PI`	180
dgrd	Converts degrees to radians	1	**180 dgrd**	`(180)* Math.PI/180`	pi
rnor	Normalizes an angle expressed in radians, the result of this operation is between 0 and 2 pi	1	**5 pi**		1.8584
if{ .... }	If statement, note there is no space between the if and the {	1	**(L:Value) 0 == if{ A }**	**(L:Value)** `== 0 ? A : 0`	Operation A is carried out if Value is 0.

els{ .... }	Else statement, note there is no space between the els and the {	1	**(L:Value1) (L:Value2) <= if{ A } els{ B }**	**(L:Value1) <= (L:Value2) ? A : B**	If Value1 is less than or equal to Value2, operation A is carried out, otherwise operation B is carried out.
quit	The quit statement allows expression evaluation to stop completely, and avoid the use of nesting if{ statements.	0	**pi quit (L:Value1) (L:Value2) <= if{ A } els{ B }**	**Math.PI**	pi. The rest of the script is ignored.
g0...gn	Goto label. Execution will jump to the specified label. Labels are set by entering a colon followed by the label number.	0	**g4**		Execution jump to :4
case	Case statement		**50 40 30 20 10 5 (L:value) case**		The "5" indicates there are five case values, which are selected depending on the evaluation of (L:value). If the evaluation is equal to or greater than 0, but less than 1, the result is 10. If the evaluation is equal to or greater than 1, but less than 2, the result is 20, and so on.
String Operators					
lc	Converts a string to lowercase	1	**'ABcd10' lc**	`('ABcd10').toLowerCase()`	'abcd10'
uc **cap**	Converts a string to uppercase	1	**'ABcd10' uc**	`('ABcd10').toUpperCase()`	'ABCD10'
chr	Converts a number to a symbol	1	**65 chr**	`String.fromCharCode(65)`	'A'
ord	Converts a symbol to an integer	1	**'A' ord**	`('A').charCodeAt(0)`	65
scat	Concatenates strings	2	**'abc' 'red' scat**	`'abc' + 'red'`	'abcred'
schr	Finds a symbol in a string				
scmp	Compares strings, case sensitive	2	**(M:Event) 'LeftSingle' scmp 0 == if{ A }els{ B }**	`( 'LeftSingle' !== 'LeftSingle' == 0 ) ? A : B`	Performs A if the left mouse button has been pressed, otherwisse performs B
scmi	Compares strings, ignoring case	2	**'left' 'Left' scmi 0 == if{ 'yes' }**	`( ('left').toLowerCase() !== ('Left').toLowerCase() == 0 ) ? 'yes' : 0`	'yes'
sstr	Finds a substring	2	**'cd' 'abcde' sstr**	`('abcde').indexOf('cd')`	2

ssub	Extracts a substring	2	**'ab' 'abcde' ssub**	`('abcde').replace('ab', '')`	'cde'
symb	Extracts a single character	2	**'abc' 1 symb**	`('abc')[1]`	'b'
Stack Operators					
b	Backup the stack	0			
c	Clears the stack	0	stack: 1 2 3 c		stack:
d	Duplicates the value that is on the top of the stack	1	stack: 5 d		stack: 5 5
p	Pops and discards the top value on the stack	1	stack: 1 2 3 p		stack: 1 2
r	Reverses the top and second values on the stack	2	stack: 1 2 3 r		stack: 1 3 2
s0, s1, … s49	Stores the top value in an internal register, but does not pop it from the stack.	1	stack: 1 2 3 s0		stack: 1 2 3 s0: 3
l0, l1, … l49	Loads a value from a register to the top of the stack	1	stack: 1 2 3 s0 l0		stack: 1 2 3 3
sp0, sp1, … sp49	Stores the top value and pops it from the stack	1	stack: 1 2 3 sp0		stack: 1 2 sp0: 3