OVERVIEW

WHAT IS SLL?

Sll is a **general-purpose programming language** aimed at **all developers** looking for a **non-OOP** language with heavy compile-time optimization (and therefor **better runtime performance**).

KEY FEATURES OF SLL

- Cross-platform (Tested on Windows 10 64-bit and multiple Linux x64 distributions)
- Easy to learn (Similar to Lisp)
- Heavy compile-time optimization

SLL SYNTAX

The syntax of the Sl Language is similar to Common Lisp (and other Lisp dialects).

It has one notable difference, that is it has no keywords. Sll uses only operators, which makes it easier to learn for people with any spoken language background.

For more specific details, see **Syntax**

INSTALLATION

Sll can be download from the **GitHub website**. The primary target operating systems are Windows and POISX.

Sll has the ability to be installed in a folder, or as a standalone executable. Here is a list of file names:

- `win.zip`: Windows build of Sll, must be extracted into a folder before use.
- `win_standalone.exe`: Windows standalone version, can be used out-of-the-box.
- `posix.zip`: POSIX build of Sll (platforms such as Linux, Ubuntu, Fedora, FreeBSD, etc.). Must be extracted into a single folder.
- `posix_standalone`: POSIX standalone version of Sll.

VERSION CHECKING

If using the standalone version of Sll, replace `sll` by `sll_standalone`. The most recent version of Sll is `0.6.27` (`0.6.28` is in development).

WINDOWS

To check the version of Sll on Windows, type the following expression into a Command Prompt (or Power Shell):

```
sll -V
```

POSIX

To check the version of Sll on POSIX systems, type the following expression into a Terminal:

```
sll -V
```

SYNTAX

The syntax of the Sl Language is similar to the syntax of Common Lisp and other Lisp dialects. The only main difference is that Sll does not have any keywords. Sll uses only operators to perform any function. This feature helps people with any language background to learn Sll.

The building-block of the Sl Language is called an **object**. An object can represent anything from an integer to a complex expression.

COMMENTS

Single-line comments begin with a semicolon, while block comments have to be surrounded with special symbols.

```
; Single-line comment

|# Block comment
   (Multiple lines)
#|
```

OBJECTS

Each object represent either a **base type** or an **operator**. If an object represents the latter, it must be surrounded by parentheses and contain the symbol for the given operator, as well as its operands.

All operands are split by white space. If an operator receives more arguments than it requires, it evaluates them as other expressions and not its operands.

BASE TYPES

Base types have a different syntax than other objects. Unlike operators, they do not require parentheses in their syntax.

INTEGER

A 64-bit signed integer.

By default, integers are represented in base 10, however, by using prefixes, it is possible to write integers in different bases:

- **B**inary (base 2): `0b` (Valid symbols are `0` and `1`)
- **O**ctal (base 8): `0o` (Valid symbols are digits `0` to `7`)
- He**X**adecimal (base 16): `0x` (Valid symbols are digits `0` to `9` and letters from `a` to `f`)

Integers are case-insensitive. The prefixes, as well as the symbols (in case of base 16) can be both uppercase and lowercase.

For clarity purposes, integer digits can be split by underscores (`_`). The only exception is that a number **cannot** begin with an underscore, because it would be recognized as an identifier.

FLOAT

A 64-bit IEEE 754 floating-point number. It consists of an option number followed by a period (`.`) and another number. The second number is not required if the first one is present. The entire expression can be followed by an `e` or `E` (denoting the scientific notation) combined with a signed integer representing an exponent. The sign of the number must be located before the first number (or period).

Likewise, with integers, digits can be split by underscores (`_`) for clarity purposes. The only exceptions are that an underscore cannot begin the number nor the exponent.

CHAR

An 8-bit unsigned integer representing a single byte of data. It is denoted by a single character or **escape sequence** enclosed in a pair of apostrophes (`''`).

If the apostrophes are empty or surround more than one character or escape sequence, a compile-time error is raised.

ESCAPE SEQUENCES

- `\"`: Quotation marks (`"`, ASCII 34)
- `\'`: Apostrophe (`'`, ASCII 39)
- `\\`: Backslash (`\`, ASCII 92)
- `\f`: Form feed (ASCII 12)
- `\n`: Line feed (ASCII 10)
- `\r`: Carriage return (ASCII 13)
- `\t`: Horizontal tab (ASCII 9)
- `\v`: Vertical tab (ASCII 11)
- `\x{value}`: Any character represented in base 16 padded to two digits. (ex. `\x41` maps to `A` and `\x0a` maps to a line feed (`\n`))

Any other character following a backslash will raise a compile-time error.

STRING

An array of **characters**. It has to be surrounded by quotation marks (`""`).

Strings support the same **escape sequences** as regular characters.

ARRAY

An array of objects. It is denoted by an enclosing pair of square brackets (`[]`).

MAP

Creates a mapping (association) between different keys and values. The keys and values can represent any objects.

A map is constructed by a pair of angle brackets (`<>`) containing key-value pairs. Every object at an even index (starting with zero) is considered a key, whereas objects at odd indexes are values. There cannot be two (strictly) equal keys in the same map.

IDENTIFIER

A name of a variable. The name can consist of any combination of ASCII letter (uppercase and lowercase), digits or underscores (`_`). The only exceptions are that the name cannot start with a digit. The other exception is that the name cannot be one of the reserved **constants**.

An identifier evaluates to the object of the variable pointed to by the identifier.

OPERATION LIST

A collection of objects to evaluate in the given order. An operation list is marked by a pair of curly brackets (`{}`) containing zero or more expressions.

An operation list evaluates to `nil` (zero).

CONSTANT

The is the list of all of the constants as well as their corresponding integer values:

Type	Value	Note
`false`	`0`	A value which is returned when an expression is not true, ex. by the **equal operator** with two different integers

`nil`	`0`	This value is returned by an expression which does not return anything
`true`	`1`	A value which is returned when an expression is true, ex. by the comparing two equal numbers

The constants are replaced by their integer value during the parsing stage of the program.

OPERATORS

All operators have the same structure:

```
(|# operator type #| |# argument 1 #| |# argument 2 #| |# argument 3 #|)
```

The operator type must be one of the following symbols:

ACCESS (:)

SYNTAX

```
(: |# object #| |# index #|)

OR

(: |# object #| |# start index #| |# end index #|)

OR

(: |# object #| |# start index #| |# end index #| |# increment #|)
```

RETURN VALUE

Value selected from the given object by the arguments, or, if the object does not support item indexing, the same object

DESCRIPTION

If the object does not support item indexing (i.e. it is not of a storage type) the same object is returned. Otherwise, the return value depends on the number of arguments.

If there is only one argument, the value at that index in the given object is returned.

A container refers to a string if the object is a string, otherwise it is an array.

If there are two arguments, all of the values between the `start index` and `end index`. If `start index` is bigger than or equal to `end index`, an empty container is returned.

Otherwise, when all of the 3 arguments are supplied, every value between `start index` and `end index`, `increment` indexes apart is returned in a container. If `increment` is equal to zero, an empty container is returned.

If the object is a mapping and there is more than one index, a list of values at these indexes is returned. The following two examples evaluate to the same value:

```
(= map <'a' 0 'b' 1 'c' 2 'd' 3>)

(= x (: map 'a' 'c'))

|# is the same as #|

(= x [
        (: map 'a')
        (: map 'c')
])
```

EXAMPLE

```
(= array [0 1 2 3 4 5 6 7 8 9])
(-> (= i 0) (< i ($ array)) {
        (:> "Array at index " i " is equal to " (: array i) "\n")
        (= i (+ i 1))
})
```

ADDITION (**+**)

SYNTAX

```
(+ |# term 1 #| |# term 2 #| |# term 3 #|)
```

RETURN VALUE

The sum of the operands

DESCRIPTION

All of the operands are evaluated, added together and the return value is returned.

EXAMPLE

```
(:> (+ 1 2 3) "\n")
```

AND (**&&**)

SYNTAX

```
(&& |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

`true` if all of the expression evaluate to **`true`**, otherwise **`false`**

DESCRIPTION

All expressions are evaluated in the specified order. If all of the expressions evaluate to **`true`**, then the entire expression returns the value **`true`**. If any of the expressions evaluate to **`false`**, the evaluation stops and the value **`false`** is returned.

EXAMPLE

```
(= x 6)
(? (&& (> x 5) (< x 10))
        (:> x " is between 5 and 10!\n")
)
```

ASSIGNMENT (**=**)

SYNTAX

```
(= |# variable #| |# value #|)
```

RETURN VALUE

The value of the second operand.

DESCRIPTION

The value of the second operand is evaluated and assigned to the variable pointed by the first operand. If the first operand is not an identifier, the entire object behaves like an **operation list**.

EXAMPLE

```
(= name "John")
```

BITWISE AND (&)

SYNTAX

```
(& |# term 1 #| |# term 2 #| |# term 3 #|)
```

RETURN VALUE

The bitwise AND product of all of the terms

DESCRIPTION

All of the operands are evaluated, and the bitwise AND product of all of the terms is returned.

EXAMPLE

```
(:> (& 0b101 0b1111) "\n")
```

BITWISE NOT (ONE'S COMPLEMENT) (~)

SYNTAX

```
(~ |# expression #|)
```

RETURN VALUE

The bitwise NOT (One's complement) of the value is return

DESCRIPTION

The one's complement (bitwise NOT) of the first (and only) operand is returned.

EXAMPLE

```
(:> (& (~ 0xa5) 0xff) "\n")
```

BITWISE OR (|)

SYNTAX

```
(| |# term 1 #| |# term 2 #| |# term 3 #|)
```

RETURN VALUE

The bitwise OR product of all of the terms

DESCRIPTION

All of the operands are evaluated, and the bitwise OR product of all of the terms is returned.

EXAMPLE

```
(:> (| 0x0f 0x80) "\n")
```

BITWISE XOR (EXCLUSIVE OR) (^)

SYNTAX

```
(^ |# term 1 #| |# term 2 #| |# term 3 #|)
```

RETURN VALUE

The bitwise XOR (exclusive or) product of all of the terms

DESCRIPTION

All of the operands are evaluated, and the bitwise XOR (exclusive or) product of all of the terms is returned.

EXAMPLE

```
(:> (^ 0xff 0x80) "\n")
```

BOOLEAN (!!)

SYNTAX

```
(!! |# expression #|)
```

RETURN VALUE

`true` if the boolean value of the expression is nonzero, otherwise **`false`**

DESCRIPTION

The first (and only) operand is evaluated, its value is converted to a boolean and returned.

EXAMPLE

```
(:> (!! 0.0) ", " (!! 0.1) "\n")
```

BOOLEAN NOT (!)

SYNTAX

```
(! |# expression #|)
```

RETURN VALUE

`true` if the boolean value of the expression is zero, otherwise **`false`**

DESCRIPTION

The first (and only) operand is evaluated, its value is converted to a boolean and the inverse is returned.

EXAMPLE

```
(:> (! 0) ", " (! 1) "\n")
```

BREAK (@)

SYNTAX

```
(@)
```

RETURN VALUE

Moves the execution outside of the current loop

DESCRIPTION

Moves to instruction pointer after the current innermost loop.

EXAMPLE

```
(-> (= i 0) (< i 10) {
        (:> "Number: " i "\n")
        (? (== i 5) {
                (:> "Early Exit!")
                (@)
        })
        (= i (+ i 1))
})
```

FUNCTION CALL (<-)

SYNTAX

```
(<- |# function expression #| |# function argument 1 #| |# function argument 2 #| |# function
argument 3 #|)
```

RETURN VALUE

The return value of the function called, or **nil**

DESCRIPTION

This object calls the function specified by the first operand with the arguments specified by the consecutive operands. If the function operand is an invalid function or internal function ID, a **nil** value is returned.

EXAMPLE

```
(<- (... "file_write") stdout "Example Code\n")
```

CAST (::)

SYNTAX

```
(:: |# object #| |# type 1 #| |# type 2 #| |# type 3 #|)
```

RETURN VALUE

The casted object

DESCRIPTION

The given object is consequently casted to each of the given types. The new, casted object is returned.

EXAMPLE

```
(# int_type)
(-- "types.sll")

(= value 1.678)
(= floored_value (:: value int_type))
```

COMMA (,)

SYNTAX

```
(, |# expression 1 #| |# expression 2 #| |# expression 3 #| |# returning expression #|)
```

RETURN VALUE

The last value of the expression

DESCRIPTION

All expression are evaluated and the value of the last one is returned.

EXAMPLE

```
(= value (, (:> "Assigning '5' to variable value...") 5))
```

CONTINUE (<<<)

SYNTAX

```
(<<<)
```

RETURN VALUE

Execution is directed to the beginning of the loop

DESCRIPTION

The instruction pointer is moved to the beginning of the loop, i.e. the just before the condition.

EXAMPLE

```
(-> (= i 0) (< i 10) {
        (? (== i 5) {
                (:> "Skipping 5!")
                (= i (+ i 1))
                (<<<)
        })
        (:> "Number: " i "\n")
        (= i (+ i 1))
})
```

DECLARATION (#)

SYNTAX

```
(# |# variable 1 #| |# variable 2 #| |# variable 3 #|)
```

RETURN VALUE

No return value (`nil`)

DESCRIPTION

The variables are declared and not initialized. Their value is undefined.

EXAMPLE

```
(# stdout)
(-- "file.sll")

(:> "Standard output file index: " stdout "\n")
```

DIVISION (/)

SYNTAX

```
(/ |# dividend 1 #| |# divisor 1 #| |# divisor 2 #|)
```

RETURN VALUE

The quotient

DESCRIPTION

All of the operands are evaluated, the first operand (the dividend) is divided by all of the divisors and the quotient is returned.

EXAMPLE

```
(:> (/ 12 4) "\n")
```

EQUAL (==)

SYNTAX

```
(== |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

If all of the expression are equal, **`true`** is returned, otherwise **`false`**

DESCRIPTION

All of the expressions are evaluated sequentially. If a pair of expressions does not evaluate to the same value, the evaluation ends and **`false`** is returned. If all of the evaluated expression are equal, **`true`** is returned.

EXAMPLE

```
(= age 20)

(? (== age 20)
        (:> "You are 20 years old!")
)
```

EXIT (@@@)

SYNTAX

```
(@@@ |# value #|)
```

RETURN VALUE

Execution of the program ends with the value as the return code

DESCRIPTION

The program is terminated and the **`value`** is casted to a 32-bit signed integer and returned as the return code.

EXAMPLE

```
(@@@ 0)
```

EXPORT (##)

SYNTAX

```
(## |# variable 1 #| |# variable 2 #| |# variable 3 #|)
```

RETURN VALUE

No return value (**`nil`**)

DESCRIPTION

Each of the variables are internally marked as exported and will not be removed by the optimizer.

EXAMPLE

```
(= export_value 12345)

(## export_value)
```

FLOOR DIVISION (//)

SYNTAX

```
(// |# dividend 1 #| |# divisor 1 #| |# divisor 2 #|)
```

RETURN VALUE

The quotient

DESCRIPTION

All of the operands are evaluated, the first operand (the dividend) is divided by all of the divisors, the result is rounded down and returned.

EXAMPLE

```
(:> (// 13 4) "\n")
```

FOR LOOP (`->`)

SYNTAX

```
(-> |# initialization #| |# condition #|
        |# loop body #|
)
```

RETURN VALUE

No return value (`nil`)

DESCRIPTION

First, the initialization code is executed. Then, while the condition evaluates to `true`, the loop body is executed.

EXAMPLE

```
(-> (= i 0) (< i 10) {
        (:> "Number: " i '\n')
        (= i (+ i 1))
})
```

FUNCTION DECLARATION (`,,,`)

SYNTAX

```
(,,, |# argument 1 #| |# argument 2 #| |# argument 3 #|
        |# function body #|
)
```

RETURN VALUE

An integer, which can be used to call the given function.

DESCRIPTION

The object defines a function. The first `n` consecutive identifiers are considered the arguments, and the rest of the operands (`operand_count - n`) are considered the body of the function.

EXAMPLE

```
(= mult_func (,,, x y
        (:> x " * " y " = " (* x y) "\n")
        (@@ (* x y))
))
```

IF (`?`)

SYNTAX

```
(? |# condition 1 #| |# code block 1 #|
        |# condition 2 #| |# code block 2 #|
        |# condition 3 #| |# code block 3 #|
        |# 'else' code block #|
)
```

RETURN VALUE

No return value (`nil`)

DESCRIPTION

Condition blocks are evaluated in the given order until one evaluates to a non-zero value. The matching code block is executed. If every condition was false and an ‘else’ block is specified, it is executed.

EXAMPLE

```
(= x 3)
(? (< x 5) {
        (:> "below")
} (> x 5) {
        (:> "above")
} {
        (:> "equal")
})
```

IMPORT (`--`)

SYNTAX

```
(-- "file_to_import_1.sll" "file_to_import_2.sll" "file_to_import_3.sll")
```

RETURN VALUE

No return value (`nil`)

DESCRIPTION

Each of the string arguments is treated as a file path. This can either be a built-in module name or an external file. If the file cannot be found, a compile-time error is generated.

EXAMPLE

```
(# write stdout)
(-- "file.sll")

(<- write stdout "Hello, World!\n")
```

INFINITE LOOP (`><`)

SYNTAX

```
(>< |# initialization #|
        |# loop body #|
)
```

RETURN VALUE

No return value (`nil`)

DESCRIPTION

First, the initialization code is executed. Then, the loop body is executed repeatedly.

EXAMPLE

```
(>< (= i 0) {
        (? (== i 10) (@))
        (:> "Number: " i '\n')
        (= i (+ i 1))
})
```

INLINE FUNCTION (***)

SYNTAX

```
(*** |# function body #|)
```

RETURN VALUE

The value returned by a return object, or `nil`

DESCRIPTION

This object works just like a function called without arguments. It can be used to write expressions more complex than **inline-if objects**.

EXAMPLE

```
(***
        (-> (= i 0) (< i 10) {
                (? (= i 5) (@@ i))
                (= i (+ i 1))
        })
)
```

INLINE IF (?:)

SYNTAX

```
(?: |# condition 1 #| |# return value 1 #|
        |# condition 2 #| |# return value 2 #|
        |# condition 3 #| |# return value 3 #|
        |# 'else' return value #|
)
```

RETURN VALUE

The value returned by the given return block.

DESCRIPTION

Condition blocks are evaluated in the given order until one evaluates to a non-zero value. The matching code block is evaluated, and the return value is returned. If every condition was false and an ‘else’ block is specified, it is evaluated and returned. Otherwise, if no ‘else’ block is specified, a `nil` value is returned.

EXAMPLE

```
(= x 3)
(:> "The number is " (?:
        (< x 5) "below"
        (> x 5) "above"
        "equal"
) " five.\n")
```

INTERNAL FUNCTION DECLARATION (...)

SYNTAX

```
(... "internal_function_name")
```

RETURN VALUE

An integer, which can be used to invoke the given internal function.

DESCRIPTION

The object declares an internal function. The operand is a string, which will be used to look-up the ID of the internal function. If the operand is not a string, the object behaves like an **operation list**.

EXAMPLE

```
(= parse_json (... "json_parse"))
```

LEFT BIT SHIFT (<<)

SYNTAX

```
(<< |# object #| |# amount #|)
```

RETURN VALUE

The new object

DESCRIPTION

The given objects is shifted left by a given amount. For integers, floats and chars, this is equivalent to multiplying by `2**(amount)`. For arrays, this means that `amount` zeros are prepended to the array.

EXAMPLE

```
(:> "Bit 16 multiplied by 4: " (<< 16 2) "\n")
```

LENGTH ($)

SYNTAX

```
($ |# object #|)
```

RETURN VALUE

The length of the object

DESCRIPTION

If the object has a storage type, the number of elements (length) of the given object is returned. Otherwise, `0` is returned.

EXAMPLE

```
(= arr [0 1 2 3 4 5 6 7 8 9])
(:> "Length of array: " ($ arr) "\n")
```

LESS (<)

SYNTAX

```
(< |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

If all of the expression get progressively bigger, **`true`** is returned, otherwise **`false`**

DESCRIPTION

All of the expressions are evaluated sequentially. If the first element in a pair of expressions is not smaller than the second one, the evaluation ends and **`false`** is returned. Otherwise, **`true`** is returned.

EXAMPLE

```
(-> (= i 0) (< i 10) {
        (:> "Number: " i '\n')
        (= i (+ i 1))
})
```

LESS OR EQUAL (**<=**)

SYNTAX

```
(<= |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

If all of the expression do not get progressively smaller, **`true`** is returned, otherwise **`false`**

DESCRIPTION

All of the expressions are evaluated sequentially. If the first element in a pair of expressions is bigger than the second one, the evaluation ends and **`false`** is returned. Otherwise, **`true`** is returned.

EXAMPLE

```
(-> (= i 0) (<= i 9) {
        (:> "Digit: " i '\n')
        (= i (+ i 1))
})
```

MODULO (**%**)

SYNTAX

```
(% |# dividend 1 #| |# divisor 1 #| |# divisor 2 #|)
```

RETURN VALUE

The remainder

DESCRIPTION

All of the operands are evaluated, the first operand (the dividend) is divided by all of the divisors and the remainder is returned.

EXAMPLE

```
(:> (% 13 4) "\n")
```

MORE (**>**)

SYNTAX

```
(> |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

If all of the expression get progressively bigger, `true` is returned, otherwise `false`

DESCRIPTION

All of the expressions are evaluated sequentially. If the first element in a pair of expressions is not bigger than the second one, the evaluation ends and `false` is returned. Otherwise, `true` is returned.

EXAMPLE

```
(-> (= i 0) (< i 10) {
        (?
                (> i 4) (:> "Round up: " i "\n")
                (:> "Round down: " i "\n")
        )
        (= i (+ i 1))
})
```

MORE OR EQUAL (>=)

SYNTAX

```
(>= |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

If all of the expression do not get progressively bigger, `true` is returned, otherwise `false`

DESCRIPTION

All of the expressions are evaluated sequentially. If the first element in a pair of expressions is smaller than the second one, the evaluation ends and `false` is returned. Otherwise, `true` is returned.

EXAMPLE

```
(-> (= i 0) (< i 10) {
        (?
                (>= i 5) (:> "Round up: " i "\n")
                (:> "Round down: " i "\n")
        )
        (= i (+ i 1))
})
```

MULTIPLICATION (*)

SYNTAX

```
(* |# factor 1 #| |# factor 2 #| |# factor 3 #|)
```

RETURN VALUE

The product of the operands

DESCRIPTION

All of the operands are evaluated, multiplied together and the value is returned.

EXAMPLE

```
(:> (* 5 4 3 2 1) "\n")
```

NOT EQUAL (!=)

SYNTAX

```
(!= |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

If each pair of expressions evaluates to two different values, `true` is returned, otherwise `false`

DESCRIPTION

All of the expressions are evaluated sequentially. If a pair of expressions evaluates to the same value, the evaluation ends and `false` is returned. Otherwise, `true` is returned.

EXAMPLE

```
(= a 1)
(= b 2)

(? (!= a b)
        (:> "Two different numbers!")
)
```

OR (||)

SYNTAX

```
(|| |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

`true` if at least one expression evaluate to `true`, otherwise `false`

DESCRIPTION

All expressions are evaluated in the specified order. If one of the expressions evaluate to `true`, the evaluation process ends and `true` is returned. If all of the expressions evaluate to `false`, the value `false` is returned.

EXAMPLE

```
(= x 6)
(? (|| (<= x 5) (>= x 10))
        (:> x " is NOT between 5 and 10!\n")
)
```

PRINT (:>)

SYNTAX

```
(:> |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

No return value (`nil`)

DESCRIPTION

Converts every argument to a string and writes it to the default output stream.

A print operator can be substituted by the following expression[^2] to obtain the number of bytes written:

```
(<- (... "file_write") -2 |# argument 1 #| |# argument 2 #| |# argument 3 #|)
```

EXAMPLE

```
(:> "Array: " [1 2 3 4] "\n")
```

REFERENCE (%%)

SYNTAX

```
(%%)

OR

(%% |# object #|)
```

RETURN VALUE

A handle to the object pointer, or a null pointer

DESCRIPTION

If there are no arguments, a handle to a null pointer is returned. Otherwise, a handle to the memory location of the given object is returned.

EXAMPLE

```
(= x 5)

(:> "Address of 'x' in memory: " (%% x) '\n')
```

RETURN (@@)

SYNTAX

```
(@@)

OR

(@@ |# object #|)
```

RETURN VALUE

Execution is returned back to the function call with the return value of `object` or `nil`, if no object is supplied

DESCRIPTION

The instruction pointer is moved back to the place of the enclosing function call, which returns the `object`, if it is supplied, otherwise returns `nil`.

EXAMPLE

```
(= mult (,,, a b {
        (@@ (* a b))
}))
```

RIGHT BIT SHIFT (>>)

SYNTAX

```
(>> |# object #| |# amount #|)
```

RETURN VALUE

The new object

DESCRIPTION

The given objects is shifted right by a given amount. For integers, floats and chars, this is equivalent to dividing by `2**(amount)`. For arrays, this means that `amount` elements are removed from the beginning of the array.

EXAMPLE

```
(:> "Bit 16 divided by 4: " (>> 16 2) "\n")
```

STRING EQUAL (===)

SYNTAX

```
(=== |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

If all of the expression (and their types) are equal, `true` is returned, otherwise `false`

DESCRIPTION

All of the expressions are evaluated sequentially. If a pair of expressions does not evaluate to the same type or the same value, the evaluation ends and `false` is returned. If all of the evaluated expression have the same type and are equal, `true` is returned.

EXAMPLE

```
(= var 5.0)

(? (=== var 5.0)
        (:> "Float with a value of 5!")
)
```

STRING NOT EQUAL (!==)

SYNTAX

```
(!== |# expression 1 #| |# expression 2 #| |# expression 3 #|)
```

RETURN VALUE

If each pair of expressions evaluates to two different types or values, `true` is returned, otherwise `false`

DESCRIPTION

All of the expressions are evaluated sequentially. If a pair of expressions evaluates to the same type and the same value, the evaluation ends and `false` is returned. Otherwise, `true` is returned.

EXAMPLE

```
(= var 5)

(? (!== var 5.0)
        (:> "'var' should be a float with a value of 5!")
)
```

SUBTRACTION (-)

SYNTAX

```
(- |# minuend 1 #| |# subtrahend 1 #| |# subtrahend 2 #|)
```

RETURN VALUE

The difference of the operands

DESCRIPTION

All of the operands are evaluated, subtracted from the first operand and returned.

EXAMPLE

```
(:> (- 6 1 2) "\n")
```

SWITCH (`??`)

SYNTAX

```
(?? |# condition #|
        |# case 1 #|  |# expression 1 #|
        |# case 2 #|  |# expression 2 #|
        |# case 3 #|  |# expression 3 #|
        |# default case expression #|
)
```

RETURN VALUE

No return value (`nil`)

DESCRIPTION

The condition expression is evaluated. If it is not an integer or character, the default case expression is evaluated (if it exists). Every case expression is evaluated. Any case expressions that are not integers or characters are skipped. The expression block is selected based on the condition value. If no code block is found and the default case expression exists, it is evaluated.

EXAMPLE

```
(= x -1)
(?? x
        -1 (:> "below")
        1 (:> "above")
        (:> "equal")
)
```

WHILE LOOP (`>-`)

SYNTAX

```
(>- |# initialization #| |# condition #|
        |# loop body #|
)
```

RETURN VALUE

No return value (`nil`)

DESCRIPTION

First, the initialization code is executed. Then, the loop body is evaluated once, after which, while the condition evaluates to `true`, the loop body is executed again.

EXAMPLE

```
(>- (= i 0) (<= i 9) {
        (:> "Digit: " i '\n')
        (= i (+ i 1))
})
```