CWScript Specification

by Collin Williams

A. Motivation and Philosophy

This scripting language is created to be easy to implement in any language without sacrificing any major features in terms of usability. It functions similar to a dynamically typed language like Python, but with a few differences in semantics and many differences in syntax.

The most central design philosophy is consistency - i.e., the language is defined using a small set of rules that apply very broadly. For example, a traditional language would specify a function call using `name(arg1, arg2)`. However, this would require defining multiple new rules for the parser, whereas `call name [arg1, arg2]` works perfectly fine using the already-existing rules for statements and list literals. While more verbose, this aligns more with the goal of defining the language with as few rules as possible.

B. Language Syntax

This section will discuss the syntax of the language. Afterwards, sections C and D will discuss the specifics of how that syntax is implemented by the lexer and parser.

All tokens in the language are grouped into 2 categories: literals and operators. (Note: All operators are either binary infix or unary prefix). A comprehensive list of every literal and operator type can be found below. These will all be expanded upon in this section.

Literals:	**Operators (binary):**
null	: (index)
bool	** * / // % + - (arithmetic)
int	> < >= <= == != === !== (arithmetic comparison)
float	&& || (logical comparison)
string	= += -= *= /= //= %= **= (assignment)
variable	
list	**Operators (prefix):**
block	- (negation)
statement	! (not)
keyword	++ -- !! (variable modification)

B1. Literal types

This section will focus on the first 8 literal types shown above, and statement and keyword will be the focus of **§B.2**. For information about the types these literals evaluate to, see **§E.1**.

Type	Example(s)	Notes

`null`	`null`	
`bool`	`true false`	Always lowercase, unlike Python
`int`	`100 -41`	
`float`	`10.0 0.5`	Cannot start or end with decimal Does not yet support scientific notation
`string`	`"asdf" '\t\n'`	Can use single or double quotes Supports `\\, \', \", \n, \t, \r` escape sequences
`variable`	`.a global.b`	Variables must start with a period unless they begin with a scope Only alphanumeric characters and underscores may be used (beginning with a number is fine) The : operator can be used for variable indexing where this format doesn't work
`list`	`[1, 2 + 3, pi]`	

`block`	`{`
	`  .a = 3;`
	`  print .a;`
	`}`
		This is the only literal here that does not have an associated value type
		Although they can be used in expressions, blocks by themselves will run and return null

B.2. Statements

Statements lie at the core of the language, as they're the actual instructions that get executed. We'll use the `max` statement as an example, since it can be understood independently of any other token types. To find the maximum of two numbers, you type `max 12 24`.

The arguments of a statement are composed of expressions, but also sometimes include keywords for readability. As an example, for loops are implemented as a statement, and they use the in keyword to better match syntax of languages like Python:
`for .i in (range 20) {}`.

Statements are parsed from left to right. Both the number of arguments and the type of arguments are checked. A parse error will be thrown if there is an incorrect number of arguments, and a runtime error will be thrown if there is an invalid type. Since all statements have a fixed number of arguments, the parser is able to handle nested statements. For example, you can type `print max 12 24`, and it will be understood as `print (max 12 24)`. For more complicated statements, parenthesis are recommended for readability.

A list of all statements can be found in **Appendix I**.

B.3. Binary and prefix operators

A binary operation can be passed in as a statement argument. There are two important things of note:

1. Binary operators must be surrounded by spaces
2. Binary operators within statements must be surrounded by parenthesis

For example, `max 1 + 1 3` is invalid, since it should be written `max (1 + 1) 3`.
Technically, `max 1 2 + max 3 4` is valid and equivalent to `(max 1 2) + (max 3 4)`.

Prefix operators are attached at the beginning of an expression, for example: `-.my_var` or `-(1 + 2)`. Prefix operators can also be attached to a statement root to apply to the result of the statement. `-max 12 24` will return -24.

A list of all operators, combined with their precedence and associativity, can be found in **Appendix II**.

C. Lexer

In a previous specification, many strings would be iterated over but then left as strings, and passed off to a recursive operation to further break that string down into tokens. The goal with this specification is to lex everything into a token on the first go, and grouping will be handled using token objects instead of string literals. This will be much more similar to a normal programming language's lexing, but with statements taking the place of other constructs.

The code will be lexed into the following token types:

Literal:	**Grouping:**	**Separator:**	**Operator:**
statement root	group opening	comma	prefix operator
free type	group closing	semicolon	binary operator

Comments and whitespace will be ignored. The above token types are the minimum necessary to begin building the AST. Statement roots are distinguished from other "free types" at this stage so the parser can easily tell where a statement begins.

In theory, during this step, it would be possible to check free-type tokens to narrow down their type further. However, this would add unnecessary complexity to this step, so these are instead handled when building the tree.

Lexing is relatively simple, since most token types require whitespace between them. The only types that don't are separators, grouping symbols, and prefix operators, each of which are

parsed according to their own rule set. When starting a new token, the lexer starts by checking if a prefix operators is attached at the beginning of the token. Then, while parsing the remaining token, a separator will immediately end the current token, even if no space is present between the separator and the preceding token.

Literal tokens will only be terminated at whitespace. Grouping, separator, and operator tokens can be terminated without whitespace. In cases like `/` vs `//`, the lexer uses greedy lexing to make sure it doesn't terminate a valid operator prematurely.

C.1. If-Else hack

Although I try to avoid using hacks whenever possible, there is a hack with the lexer that helps to simplify the implementation of `if-else` statements. The details that follow make more sense after reading **§D.3** and **§Appendix I** first.

The way statements are implemented in the parser gives an easy way of implementing `if` statements, but `else` statements are a little more tricky. Ideally, the `else` part of the statement would be optional, but statements must take a fixed number of parameters.

Luckily, the logical OR operator `||` combined with the return value of `if` statements gives an easy way to solve this. Note that `if` statements return whether their body was executed, and the `||` operator uses short-circuit evaluation (the second operand is evaluated only if the first returned false). Thus, putting `||` between two `if` statements will only execute the second if the first returns did not execute, which is exactly what we want! Also, note that a "stranded block" not in an `if` statement will be run unconditionally when evaluated. However,
`if () {} || if () {} || {}` looks a little unsightly, whereas
`if () {} else if () {} else {}` is what we actually want.

Getting this to work is very simple: when we are left with a free type token, check if it is equal to `"else"` before appending it to the list. If it is, then append `"||"` to the list instead.

D. Parser

D.1. Finding a statement

The code will be parsed statement by statement. To start, we must figure out where our statement starts and ends. Now that our code is lexed, we simply keep going until we find a semicolon token, with one caveat. Statements can contain objects (ex. blocks) that contain semicolons inside of them. We don't want to stop at any inner semicolons, and instead want to keep going until we find a semicolon at the same nesting level as the rest of the statement. To achieve this, simply push/pop from a stack whenever we encounter an opening/closing token, and only count semicolons that are encountered with an empty stack.

D.2. Parsing groups

Within our statement, groups have the highest precedence, so we parse them first. To find which tokens are contained in the group, use a stack just as in the previous section. If the correct closing symbol is at the same nesting level, the group ends there. If there is no matching symbol, an error should be thrown. There are 3 types of grouped expressions, each of which results in a different object:

Expression type	**Results in**
`parenthesized group`	`StatementExpression`
`braced group`	`BlockExpression`
`bracketed group`	`ListExpression`

Parsing parenthesized groups:

A parenthesized group will be parsed in the same way as an individual statement. Simply treat the group body as the statement, and start from **§D.2**.

Parsing braced groups:

Braced groups act as a block of code, i.e., it follows the same rules as the program itself. To parse it, start from **§D.1.**

Parsing bracketed groups:

Lists are parsed in the same way as blocks, but they use commas as delimiters instead of semicolons.

D.3. Parsing statements and prefix operators

Both prefix operators and statements are parsed at the same time, from right to left. To explain the reasoning behind this, consider the following example that uses nested statements:

```
max log 2 8 log 10 100
```

We want the two log statements to be grouped within the max statement, so we parse the individual statements from right to left, even though the statements themselves are read from left to right.

Raw statement:	`max log 2 8 log 10 100`
Parse statement:	`max log 2 8 [LogExpression]`
Parse statement:	`max [LogExpression] [LogExpression]`
Parse statement:	`[MaxExpression]`

Now consider a situation where we apply a prefix operation to an element, for example, negating the variable "a" (written `.a`):

`max -.a log 2 8`

This operator must be parsed before the statement as a whole, otherwise the operator would be parsed as its own argument. We can achieve this by parsing prefix operators, then statements.

Raw statement:	`max -.a log 2 8`
Parse prefix:	`max [NegationExpression] log 2 8`
Parse statement:	`max [NegationExpression] [LogExpression]`
Parse statement:	`[MaxExpression]`

Note that this doesn't work for negating the statements themselves, since the statement must be parsed before the negation operator can be parsed. However, if we modify our parsing algorithm to parse both statements and prefix operators at the same time, from right to left, we can allow negating both statements and their arguments with no need for parenthesis!

Raw statement:	`-max 1 log 2 -.a`
Parse prefix:	`-max 1 log 2 [NegateExpression]`
Parse statement:	`-max 1 [LogExpression]`
Parse statement:	`-[MaxExpression]`
Parse prefix:	`[NegateExpression]`

This is especially useful for constants like `-pi`, since `pi` is internally implemented as a zero-argument statement. Without this implementation, we would have to write it as `-(pi)`.

Parsing prefix operators:

Prefix operators are pretty simple to parse. Simply confirm that the token to the right is either a "free type" token or an already-parsed StatementExpression. If it is a free type, it should be parsed first (see **§D.5**). Also note that in the case of a prefix negative preceding a number, the two tokens should simply be combined into a number literal. This avoids an unnecessary operation when executing the code.

Note that multiple prefix operators in sequence are technically allowed by this specification, although they are not recommended. For example, `---.a` will be parsed as `--(-.a)`, which is

nonsense. This might be disallowed in the future, but it's not a problem for now since you would never expect a statement like this to have a sensible result anyway.

Parsing statements:

While being parsed, statements must check that the tokens being used for arguments match up with the specification. This doesn't refer to type checking, and instead simply means confirming that fields expecting an expression contain a token that can be parsed to an expression. If the token is a group or statement, it should be parsed already by the time it reaches this step. Otherwise, the token must be a free type, in which case it should be parsed along with the statement (see **§D.5**).

Some statements contain keywords for readability. These must match with the correct statement keyword token, although this won't be stored in the resulting expression.

D.4. Parsing binary operators

Binary operators are parsed according to their precedence, which can be seen in **Appendix II**.

Both operands should be an already-parsed StatementExpression, or a free type. If it is a free type, it should be parsed alongside the operator (see **§D.5**). If either operand is a statement root, statement keyword, or any other type that has yet to be parsed, an error should be raised.

D.5. Parsing free types

In the case that this statement was a free type by itself, it should be parsed now. Having a statement with a free type by itself is still proper syntax, although in most cases, it won't do anything. Aside from this edge case, free types are usually parsed during the parsing of operators and statements that take them as operands.

Free types can be parsed into many different expression types, each of which have their own defining characteristics. The parser starts at the top and goes down until it finds a matching type. If it cannot match any type, it will raise an error.

Expression type	**Characteristic**
NullExpression	"null"
BoolExpression	"true" or "false"
StringExpression	first_char == last_char == (" or ')
VariableExpression	starts with "." or "global."

`IntExpression`	`Contains only '0123456789-'`
`FloatExpression`	`Contains only '0123456789-.'`

E. Language Semantics and Evaluator

Now that the parser has created the abstract syntax tree, the evaluator propogates down it to work out the values of each expression at runtime. There are multiple ways to implement said tree structure, such as recursive function calls or using an evaluation stack. Since this specification only specifies the *behavior* and not *implementation* of the language, it won't contain any details on how exactly to implement this. The important thing is that the expected behavior, such as type checking, evaluation order, and interrupts work as expected.

E.1. Value types

Evaluating an expression or literal in the AST will return a value, which is then usable in other expressions. Below is a list of all value types, along with their defining properties. There are 3 operations supported by all value types: casting to string, casting to bool, and checking equality.

Null Value:

Stores: No internal value.
String operation: Returns `"null"`.
Bool operation: Always `false`.
Equality operation: Checks that `other` is also `null`.

Bool Value:

Stores: 0 or 1.
String operation: Returns `"true"` or `"false"`.
Bool operation: 0 = `false`, 1 = `true`.
Equality operation: Checks if numerically equal to operand. For example, `true == 1`, but `true != 2`. Works for any numeric type.

Int Value:

Stores: An integer. Supports range [-2^{31}, $2^{31} - 1$] at a minimum, may or may not support integers outside the range depending on implementation. Thus, anything outside this range is undefined behavior.
String operation: Casts integer to string
Bool operation: 0 = `false`, nonzero = `true`.
Equality operation: Checks if numerically equal to operand. Works for any numeric type.

Float Value:

Stores: A float. Supports 32-bit IEEE 754 range at a minimum, may or may not support floats outside the range depending on implementation. Thus, anything outside this range is undefined behavior.
String operation: Casts float to string. Always includes decimal point, even if equal to integer.
Bool operation: 0 = `false`, nonzero = `true`.
Equality operation: Checks if numerically equal to operand. Works for any numeric type.

String Value:

Stores: A sequence of characters. Supports ASCII at a minimum, may or may not support unicode.
String operation: Returns the string itself. Note that if this string is part of another datatype, like a List, the string should have quotes around it when printed.
Bool operation: empty = `false`, nonempty = `true`.
Equality operation: Checks if the sequence of characters is the same

Variable Value:

Stores: Two things: a reference to a scope (Object), and a field name (string).
String operation: Prints `VAR:_____` alongside the identifier for the variable. The identifier should be unique: an integer ID or a memory address are both valid, and can depend on implementation.
Bool operation: Always `false`.
Equality operation: Checks if two variables occupy the same memory.
Important note: Variables can only exist as arguments for expressions. Any operation that can store values in memory will evaluate the variable instead of storing it directly. As of now, there are no plans to add references to the language, as mutable types like objects can already support this behavior.

Function Value:

Stores: Two things: a parameter list (list of strings), and the function body (a block).
String operation: Prints `FUNC:_____` alongside the identifier for the variable.
Bool operation: Always `false`.
Equality operation: Checks if two functions occupy the same memory.

List Value:

Stores: A list of values.
String operation: Prints `[a, b, ..., z]`, where `a-z` are the string representation of the List's values.
Bool operation: empty = `false`, nonempty = `true`.
Equality operation: Sequentially performs an equality operation on every element in the two lists.

Object Value:

Stores: A map of fields (strings) to values. Note that Objects are unordered, so code should not rely on order being consistent.
String operation: Prints `{key_1: val_1, ..., key_n: val_n}`, for every `key:value` pair in the Object.
Bool operation: empty = `false`, nonempty = `true`.
Equality operation: Checks that both Objects have the same fields and the values in each field are equivalent.

Aside from these values types, there are also a few "value classes" used for grouping types together, based on shared behavior:

Numeric Value:
Behavior: Can be used in arithmetic operations and directly compared to other numeric values
Includes: Bool, Int, and Float values

Integer Value:
Important: Not to be confused with Int value! Int is a value type, Integer is a classification for value types.
Behavior: A subset of Numeric values that can be used in situations appropriate for integers, like List indexing.
Includes: Bool and Int values

Mutable Value:
Behavior: Can be modified in-place without returning a new object. Variables pointing to these store a reference to the same object instead of copying it.
Includes: Function, List, and Object values

Container Value:
Behavior: Can be indexed, implicitly (variable literals) or explicitly (`:` operator)
Includes: List and Object values

E.2 Expression evaluation

This section and onward will use the term "expression" to mean anything that can be evaluated and return a value. Expressions include but are not limited to literals, binary operations, statements, and blocks.

When a statement, such as `max 1 (2 + 3)` needs to be evaluated, the evaluator must evaluate the operands of `max` before it can evaluate `max` itself. This reasoning is captured by the abstract syntax tree, which organizes it like so:

```
"max" operation
   -  1
```

```
-  "+" operation
      -  2
      -  3
```

To evaluate `max`, it recursively works its way up from the bottom until the evaluated arguments can be passed to `max`. This representation is pretty standard across programming languages. However, there are a few important semantics about evaluation behavior.

1. The arguments of binary operations and statements are always evaluated from left to right. This way, any side effects from evaluation happen in the intended order.
2. Once an expression is evaluated, an optional type check is performed on the result. If only a single value type/class is valid, it will be checked immediately after evaluation (i.e. before evaluating the next argument). However, for situations where a single type class doesn't encapsulate all types (ex. addition works for numeric, string, and List types), or the necessary type can't be worked out immediately (ex. indexing uses a string or an int depending on the type of the left operand), this should be deferred until all arguments are evaluated.
3. When a variable is used in an expression, the value of the variable should be returned under most circumstances. The exception is for expressions like variable assignment that operate on a variable. Thus, a statement/operation definition must be able to specify whether an argument should be evaluated or left as a variable.

Some expressions cause interrupts before finishing execution, like `break` or `return`. These work as you would expect in any other programming language, and are handled by unwinding the evaluation stack until an expression capable of handling the interrupt is reached. For example, a function `call` statement should handle a `return` interrupt.

E.3. Function calls and scoping

When a program begins execution, a single global scope is created. Scopes are not anything special, and instead are simply an Object value that variables automatically get assigned to. For example, `.a = 1` is equivalent to `(local : 'a') = 1`. This also means any operations that can be done on Objects can also be done on scopes, such as iterating or removing elements.

When a function is called, a new scope is created, and any arguments are bound to the scope using the parameter names in the function's definition. While in the function scope, the function generally does not have access to any variables declared outside the scope, even if they were declared above the function in the same file. This behavior differs from most programming languages, but it also makes implementation easier and avoids confusing situations on the user's end. The one exception to this rule is that the global scope is still accessible through the `global` statement. Function scopes are removed when the function finishes executing, although the scope can technically be stored elsewhere before being removed from the call stack.

Any other blocks, like `if`, `for`, or even stranded blocks, do not create a new scope when run. This means any variables from outside the block are accessible, and any variables declared inside the block will persist beyond the block's end. Since this language doesn't have the same concept of variable declarations as others do, this makes it easier to work with control statements without needing to create default values for variables outside of the statement body.

Since any variable access must explicitly specify its scope (with the only exception being variables like `.a`, which implicitly define `local` as their scope), variable shadowing does not exist in this language.

E.4. Memory management

As mentioned in **§E.1**, mutable types are stored by reference instead of having their value copied. Thus, the memory semantics follow the same rule as in languages like Python. The number of references to a mutable object should be stored internally, and when the reference count reaches 0, the object should be freed from memory (which most likely entails deleting it from the heap).

Recall that scopes are simply Object values, which are a mutable type. To ensure scopes aren't deleted prematurely, a scope being in the call stack should count as a reference. This would normally mean that the scope is automatically deleted upon the function's end, but this can be prevented by storing the scope in a variable (although this has questionable utility).

E.5. Variable name hack

In all cases so far, when an expression accepts another expression as an argument, it doesn't care about what the expression actually is. It only cares whether the expression evaluates to the type it expects. However, there is one situation in which this behavior is restricted to avoid getting weird results: statements that take variable *names*. Note: this is different from statements that accept a variable *value*. As an example, assignment statements accept a variable value, while the `catch` body of `try-catch` accepts a variable name to store the exception in.

E.6 Runtime exceptions

Aside from user-defined exceptions, statements and operators can also throw exceptions that the user is able to catch. These cases will thrown an exception Object with 2 fields: `type` will be a short string giving information about what the exception is, and `body` will be a message about what happened. Although `catch` will catch all exceptions by default, an exception can be rethrown if it's unable to be handled. Below is a list of all built-in exception types.

Type	Context
`invalid_type`	Type check for argument failed
`invalid_cast`	Explicit cast failed (implicit cast error throws `invalid_type`)
`invalid_index`	List index or Object index failed Also applies to undeclared variables, since scopes are Object values
`invalid_argument`	State validation for arguments failed ex. `log` of negative value
`zero_division`	Special case of `invalid_argument`

Note that exceptions that are uncatchable by the user can still occur at runtime. These are usually exceptions meant to highlight syntax errors, like invalid `continue` or `break`. This also happens when exceeding recursion depth. These exceptions will be referred to as "fatal" exceptions.

E.7. Random number generation

Regardless of the underlying implementation of the language, random number generation behavior should be consistent, (assuming the starting seed was the same). Because of this, RNG has been purposely implemented in a simple way.

The evaluator has one internal random number generator that is shared by all RNG statements. This RNG stores a single internal value: a `current` integer that determines the next output. Using the `rng_seed` statement will set this value directly, and using `rng_get` will return this value. This way, the program can simulate having multiple RNG objects by saving/loading RNG states as desired.

The RNG can be pseudo-randomly shuffled using `rng_reset`. Since the point of this operation is to be unpredictable, an exact implementation will not be specified here, (however, it is recommended to use the system clock). All that matters is the `current` value is set to a new number every time this is called. If you need consistency in how the RNG operates, use a user-defined `rng_seed` value instead.

All RNG operations use a linear congruential generator with the same values used as in C++ minstd_rand(): `next = current * 48271 % 2^31`. The RNG state is updated with `current = next`, and this `current` will be used as the value for whatever statement shuffled the RNG.

For random integers, a value within the desired range is obtained simply by using modulus on the returned value. As an example, for `irandom_range 5 20`, `5 + (current % 15)` is returned. The same operation is used internally for choosing a random List element.

For random floats, a similar trick is used. For `random`, a random float in `[0, 1)` is generated with 23 binary digits of precision (23 is taken from the mantissa in 32-bit IEEE 754 floating point). This is done with the formula $(current \% 2^{23}) / 2^{23}$. For `random_range`, this result is multiplied by the size of the range.

Appendix I. Statement Documentation

Each argument indicates the type it expects. An asterisk `*` indicates any type is allowed. In cases where an argument accepts multiple type classes, the validation is only performed after all arguments are evaluated.

Also note that although some statements specify a `block` type, this is not strictly enforced, since blocks don't have an identifiable return type. This allows using a single statement instead, which can be neater in some cases.

`print [value: *]`
Casts `value` to string and prints it to the console. Automatically prints newline. Returns null.

`prints [value: *]`
Same as `print`, but does not automatically print newline.

`local`
Returns the local scope Object.

`global`
Returns the global scope Object.

`bool [value: *]`
Returns `value` casted to bool. See **§E.1** for information on how specific types perform this operation.

`int [value: numeric|string]`
For numeric types, returns an int truncated toward 0.
For strings, reads an int using the same method as the parser.

`float [value: numeric|string]`
Casts numeric types to float and convets from string using the same method as parser.

`str [value: *]`
Returns `value` casted to string. See **§E.1** for information on how specific types perform this operation.

`typeof [value: *]`
Returns one of `"null"`, `"bool"`, `"int"`, `"float"`, `"string"`, `"variable"`, `"function"`, `"list"`, or `"object"`, depending on type.

`if [condition: *] [body: block]`

Runs `block` if `condition` evaluates to true. Returns whether the block was run or not.

`while [condition: *] [body: block]`
Continues to run `block` while `condition` evaluates to true. Returns whether the block was run at least once.

`for [iterator: variable] in [list: list] [body: block]`
Iterates through `list` and continuously runs `block` until `iterator` reaches the end of `list`. Returns whether the block was run at least once.

`continue`
Used in loops to skip to the next iteration. Fatal exception if outside loop.

`break`
Used in loops to skip this iteration, and the remaining iterations after. Fatal exception if outside loop.

`try [body: block] catch [exception: variable] [catch_body: block]`
Catches an exception, either thrown by the user or by a built-in operation. See **§E.6** for exception types. If an exception is caught, it will be assigned to `exception` and the `catch_body` will run.
Note: `exception` uses strict variable syntax, meaning property access or the index operator is not allowed.

`throw [exception: object]`
Throws a user-defined exception. Only Object values can be used as exceptions to simplify validating the exception type in `catch` bodies.

`len [value: string|container]`
Returns the number of characters in a string or the number of elements in a container.

`slice [value: string|list] [start: integer] [end: integer]`
Returns the given `value` with indices outside (`start` inclusive to `end` exclusive) removed.
Use `start = 0` if the intention is to slice before an element.

`slice_after [value: string|list] [start: integer]`
A special case of `slice` for slicing after an index, which avoids needing to explicitly state `len value` as the `end` argument.

`split [source: string] [delimiter: string]`
Returns a List split along each instance of `delimiter`, which can be any number of characters.
Returns `[]` when `source` is empty.
Returns a sequential List of each character when `delimiter` is empty.

`join [source: list] [delimiter: string]`
Converts every element in `source` to a string, and joins them with `delimiter` inbetween.
Returns `""` when `source` is empty.

`find [source: string|container] [value: *]`
Returns the index of the first element equivalent to `value`, or -1 if not found.

`replace [source: string] [old: string] with [new: string]`
Returns a string with every occurrence of `old` replaced with `new`.

`upper [source: string]`
Returns `source` with all alphanumeric characters converted to uppercase.

`lower [source: string]`
Returns `source` with all alphanumeric characters converted to lowercase.

`append [source: list] [value: *]`
Modifies `source` in-place, adding `value` at the end. If a new List is desired, use concatentation via the `+` operator instead.

`pop [source: container] [index: integer|string]`
Removes the given index from the container. Note that integers are used for Lists and strings are used for Objects. Throws `invalid_index` exception if necessary.

`range [end: integer]`
Returns List containing range of integers from 0 to `end`, exclusive.

`adv_range [start: integer] [end: integer] [step: integer]`
Gives more control than `range`, allowing a custom `start` and `step`. If step moves in the opposite direction of `end`, and therefore will never terminate, `[]` is returned.

`function [name: variable] [parameters: list] [body: block]`
Creates a function, and assigns it to `name`. The function is also returned.
Note: `name`, as well as each name in `parameters`, uses strict variable syntax, meaning property access or the index operator is not allowed.
Note: `parameters` uses parameter list syntax, meaning it must be a List literal containing variable literals, each of which have a unique name.

`lambda [parameters: list] [body: block]`
Same behavior as `function`, but doesn't automatically assign it to a variable.

`return [value: *]`

Exits a function prematurely and sets the function's return value. Fatal exception if outside function.

`call [function: function] [args: list]`
Invokes a function call with `args`. Throws `invalid_argument` exception if `args` is the incorrect size. See **§E.3** for more information on functions and scope.

`new [body: block]`
Creates and returns a new Object. The method for doing this is a bit unconventional, since this language has no concept of an Object or Dictionary literal. The provided `body` block is run in a new scope, and the end result of the scope is returned as the Object.

`copy [object: container]`
Returns a shallow copy of the given `object`.

`o_keys [object: object]`
Returns a list of all key strings in `object`.

`o_values [object: object]`
Returns a list of all value strings in `object`, including potential duplicates.

`getd [object: object] [field: string] [default: *]`
Returns `object : field` if `field` is present, otherwise returns `default` as a fallback.

`setd [object: object] [field: string] [default: *]`
Assigns `object : field = default` if `field` is not present. Returns the value occupying `object : field` after the operation.

`round [value: numeric]`
Returns `value` rounded to nearest integer, with numbers ending in .5 always rounded away from 0. Return type is always int.

`floor [value: numeric]`
Returns `value` rounded down to the nearest integer. Return type is always int.

`ceil [value: numeric]`
Returns `value` rounded up to the nearest integer. Return type is always int.

`trunc [value: numeric]`
Returns `value` rounded toward 0. Return type is always int.

`abs [value: numeric]`

Returns absolute value of `value`. Return type is preserved: a float is returned if `value` is float, and an int is returned otherwise.

`sign [value: numeric]`
Returns 0 if `sign == 0`, 1 if `sign > 0`, and -1 if `sign < 0`. Return type is always int.

`max [value_1: numeric] [value_2: numeric]`
Returns the greater of the 2 values, with `value_1` returned if they are equal.

`min [value_1: numeric] [value_2: numeric]`
Returns the lesser of the 2 values, with `value_1` returned if they are equal.

`maxl [values: list]`
Returns the greatest value in the list. In cases of a tie, the element that came first is returned.
Note: Every value in `values` must be a numeric type.

`minl [values: list]`
Returns the smallest value in the list. In cases of a tie, the element that came first is returned. An `invalid_argument` exception is thrown if `values` is empty.
Note: Every value in `values` must be a numeric type.

`clamp [value: numeric] [min: numeric] [max: numeric]`
Keeps a value between two bounds. If `value > max`, returns `max`. If `value < max`, returns `min`. Otherwise, returns `value`. An `invalid_argument` exception is thrown if `min > max`.

`sqrt [value: numeric]`
Returns the square root of `value`, with return type always being float. An `invalid_argument` exception is thrown if `value` is negative.

`log [base: numeric] [value: numeric]`
Returns the log_`base` of `value`. An `invalid_argument` exception is thrown if `base` or `value` results in an undefined value.

`ln [value: numeric]`
Returns the natural log of `value`. An `invalid_argument` exception is thrown if `value` is not positive.

`sin [value: numeric]`
Returns the sine of `value`, given in radians. Return value is always float.

`cos [value: numeric]`
Returns the cosine of `value`, given in radians. Return value is always float.

`tan [value: numeric]`
Returns the tangent of `value`, given in radians. Return value is always float.

`asin [value: numeric]`
Returns the inverse sine of `value` in radians. Return value is always a float between `[-pi/2, pi/2]`. An `invalid_argument` exception is thrown if `value` is not in range `[-1, 1]`.

`acos [value: numeric]`
Returns the inverse cosine of `value` in radians. Return value is always a float between `[0, pi]`. An `invalid_argument` exception is thrown if `value` is not in range `[-1, 1]`.

`atan [value: numeric]`
Returns the inverse tangent of `value` in radians. Return value is always a float between `(-pi/2, pi/2)`.

`atan2 [y: numeric] [x: numeric]`
Returns the angle, in radians, of the given `(x, y)` vector with respect to the positive x-axis. Return value is always a float between `[0, 2pi)`.

`rng_seed [value: integer]`
Modify the seed of the internal RNG to `value`. See **§E.7** for more details on random number generation. Returns `value`.

`rng_get`
Returns the current seed value of the RNG. This can be passed to `rng_seed` to restore the current state in the future.

`rng_reset`
Chooses a new random seed using the system clock. See **§E.7** for more details. Returns the newly chosen seed.

`random`
Returns a randomly-generated float in the range `[0, 1)`, with uniform distribution.

`random_range [min: int] [max: int]`
Returns a randomly-generated float in the range `[min, max)`, with uniform distribution.

`random_choice [list: list]`
Returns a random element from `list`, with uniform distribution. Returns null if `list` is empty.

`irandom [max: int]`
Returns a randomly-generated int in the range `[0, max)`, with uniform distribution.

`irandom_range [min: int] [max: int]`
Returns a randomly-generated int in the range `[min, max)`, with uniform distribution.

`pi`
Returns a float equal to the first 15 decimal places of *pi*: `3.141592653589793`.

`euler`
Returns a float equal to the first 15 decimal places of Euler's number *e*: `2.718281828459045`.

Appendix II: Operator Documentation

For arithmetic operations, "automatic" type resolution works like so:

- If either operand is a float, a float will be returned as the output
- If no operands are a float (i.e. ints or bools), an int will be returned

Precedence	Associativity	Operator(s)	Behavior
1	Left to Right	:	**Index:** Indexes an list, object, or string. For lists and strings, the right operand must be an integer type and must be in range (although negative indices can be used to read from the end). For objects, the right operand must be a string.
2	Left to Right	**	**Exponentiation:** Uses automatic type. Throws `invalid_argument` exception if performing an even root on a negative number (this behavior may not be consistent across implementations, depending on float rounding).
3	Left to Right	*	**Multiplication:** Uses automatic type.
		/	**Float division:** Divides 2 numbers and always returns a float, even if the operands and result are integers.
		//	**Integer division:** Divides 2 numbers and truncates the result, always returning an integer.
		%	**Modulus:** Remainder of division For negative numbers, returns the true result of modular arithmetic, i.e. -1 % 8 = 7 instead of -1. Also works with floats. Uses automatic type.
		Note: All 4 of these operations can throw a `zero_division` exception.	
4	Left to Right	+	**Addition:** Used for arithmetic, but also supports string and list concatenation. Arithmetic uses automatic type.
		-	**Subtraction:** Uses automatic type.
5	Left to Right	< <= > >=	**Comparison operators**

		`== !=`	**Equality operators:** Most types have a built in method for checking equality For types that don't, like functions, the **identity operator's** behavior is used.
		`=== !==`	**Identity operators:** Checks if two values are the same object in memory (see identity operators). For non-mutable types, same as the **equality operators.**
6	`Left to Right`	`&&`	**Logical AND:** Uses short-circuit evaluation: if the first operand evaluates to `false`, the second is not evaluated.
7	`Left to Right`	`||`	**Logical OR:** Uses short-circuit evaluation: if the first operand evaluates to `true`, the second is not evaluated.

8	`Right to Left`	`= += -=`
		`*= /= //=`
		`%= **=`
			Assignment: Left operand must be a variable.