# The Vortex programming language

Daniel "q66" Kolesa

October 18, 2018

**Abstract**

Vortex is a scripting language created to explore the possibilities of using Lua as an intermediate language. While compiling to performant Lua (comparable to handwritten code), Vortex tries to simplify programming by introducing new features (such as lambda expressions, macros, lists and objects) that Lua lacked before (while not introducing anything that is not present in regular Lua). Vortex gains inspiration from well known programming languages mainly in the functional paradigm, such as OCaml, F#, Scheme and Rust besides Lua itself. The language itself tries to offer the programmer multiple paradigms – functional, procedural, object oriented (prototype based – delegative with multiple inheritance) and metaprogramming. Vortex does not try to create a “preprocessor” for Lua (akin to attempts like CoffeeScript/MoonScript).

# Contents

# 1 Introduction

This document attempts to create a reference manual for the Vortex programming language. It tries to cover:

- Various aspects of the language, including lexical analysis and all of the language's structures. The standard library is not covered (and at the point of writing, not yet designed).
- Rationale for many of the language features.
- Examples of usage.

It tries to be just documentation, not a tutorial. For a tutorial, check out other sources (such as the Vortex wiki).

## 1.1 Disclaimer

Vortex is still in heavy development. This document will change as the language changes – do not use it in production. While the basic idea and style is given, language features may appear, change and vanish. The implementation may not reflect the actual state of the language. Some features described here are not yet implemented at the time of writing. Unimplemented features will be clearly marked as such in their section headers.

## 1.2 Conventions

The formal grammar of Vortex specified here is written using an extended dialect of BNF. The BNF strings may contain `/regexes/`. Example code is written using `monospace` blocks. Character ranges may be used for obvious things, such as `a-Z` or `0-9`.

# 2 Concepts

Vortex shares most of the basic concepts with Lua.

## 2.1 Variables and values

Vortex is an untyped language, or unityped to be precise – there is a single data type in the language, a value, similar to Lua, Python, JavaScript and so on. The values are tagged. The variable holds the data type and its value holds the tag.
Vortex type tags are shared with Lua, being represented by `nil`, `boolean`, `number`, `string`, `function`, `userdata`, `thread` and `table`. The `nil` tag can have a single value, `nil`. The `boolean` tag can have the `true` and `false` values. The only two values that evaluate as false in conditions are `false` and `nil`. The tag `number` represents a floating point number (by default double precision), depending on the underlying C. The tag `string` is an immutable sequence of bytes. Vortex is like Lua 8-bit clean. Strings can contain any byte, including embedded zeros.
The tag `userdata` has the same meaning as in Lua. The managed kind represents a managed (garbage collected) block of memory that can have a metatable. The light kind represents a raw pointer. The tag `thread` is represented by coroutines which are identical with Lua's. Functions are represented using the tag `function`. They're transparent closures.
And finally `table` is the ultimate do-it-all data structure of Vortex. It represents an associative array (unordered hash map). It's heterogenous, being able to hold any arbitrary value except nil in both key and value positions.
Vortex also has lists. These are similar to lists in Lisp or other functional languages. They don't hold their own tag – they're tables with specific behavior.
Functions, threads, full userdata and tables are always accessed by reference, everything else by value (copied). All values are first class values. You can store them, pass them around

and return from functions.
There are two types of variables in Vortex, locals and table fields. Global variables are merely fields of the global table called `_G`. Local and global variables have to be declared before they can be assigned or used, which is different from Lua, where assignment to undeclared variable creates a global and reading an undeclared variable results in the `nil` value. You declare and define variables using the `let` expression.

## 2.2 Metaprogramming

Vortex features two types of metaprogramming – static and dynamic. Dynamic metaprogramming is identical to Lua's and is represented by reflection over tables and by metatables. Vortex however also introduces static metaprogramming (performed at compile time). That is represented by macros. Macros work at AST level. They're hygienic – identifiers inside them cannot escape unless explicitly desired.
As Vortex's module system is dynamic, evaluated already past the macro expansion step, there is a system of language "extensions" that allow macros across modules.

## 2.3 Modules

As said above, Vortex features a dynamic module system. It's taken from Lua itself and it's compatible. It's further documented in the standard library part.

## 2.4 Tables and metatables

As said before, the table is the ultimate do-it-all and the only data structure in Vortex and Lua. Vortex also has lists, but they're simply tables with syntax. The real power of tables comes from metatables.
Every table can have a metatable, or any value actually. In case of tables and userdata, each piece of userdata and each table can have (but doesn't have to have) its own, unique metatable. With other values, the metatable is specific to the type tag (for example all strings share a single metatable). Of course, you can share metatables between tables, but it's all explicit.
Metatables are ordinary tables. A metatable contains metamethods. Metamethods define fallbacks for operations on the value. What does this mean? It means you can change the semantics on tables and other values and extend the language this way. For example, a metamethod `__add` defines what happens when you add the value together with some other value.
Metamethods have an intuitive naming scheme based on the name of the event they handle. They're prefixed with two underscores. You can get a metatable of a value using the `get_mt` function. You can set a metatable using `set_mt`.
The metamethod handling is identical to that of Lua. Please refer to its reference manual for more information.

**__add** : the `+` operator.
: This metamethods is called when you try to add something to the value or if something that has no `__add` tries to add he value to itself. It takes two arguments representing the left side of the operation and the right side. It returns the addition result.

**__sub** : the `-` operator.
**__mul** : the `*` operator.
**__div** : the `/` operator.
**__mod** : the `%` operator.
**__pow** : the `^` operator.
**__unm** : unary minus, takes just one argument.
**__concat** : the `~` operator.

**__len** : the `#` unary operator.
You can get the value's length using the `raw_len` function without invoking this metamethod.

**__eq** : the `==` operator.
You can compare two values using the `raw_eq` function without invoking this metamethod.

**__lt** : the `<` operator.
The `>` operator behavior is defined by reversing the operand order.

**__le** : the `<=` operator.
The `>=` operator behavior is defined by reversing the operand order. When the `<=` metamethod is absent, `<` is used, assuming `a <= b` is equivalent to `not (b < a)`.

**__index** : indexing.
This metamethod is called on `val[key]`. It's only called when `key` is not present in the table (it's a fallback operation). The first argument is the value you're indexing on and the second argument is the key. It's supposed to return the value you want to get.
If you want to retrieve a member without ever invoking this metamethod (e.g. for use inside this metamethod so that it doesn't call recursively), use `raw_get`.

**__newindex** : index assignment.
This metamethod is called on `val[key] = value`. It's only called when `key` is not present in the table. The firstargument is the value itself, the second argument is the key, the third argument is the value. Returns nothing, or ignores any return values.
If you want to set a member without invoking this metamethod, use `raw_set`.

**__call** : a value call.
Called on `myval(arglist)`. The first argument is the value that's being called, followed by the list of arguments to the call.

**__gc** : called on garbage collection of the value.
Useful for tables and userdata. As Vortex uses a garbage collector, tables are not freed immediately – instead they're marked for collection, and then during collection cycle freed. The object finalizers are called in the reverse order that they were marked.

## 2.5 Coroutines

Lua offers coroutines and so does Vortex. Coroutines are used to do collaborative multithreading. A coroutine is a thread – it has its own stack, but it's not an OS thread. Unlike Lua, Vortex offers specialized coroutine syntax.
A coroutine is somewhat similar to a function. When you create a coroutine from a function, you get a thread object. The thread object is by default in a suspended state. When you resume the thread object, the function runs. If it's a regular function you could call normally, the thread object dies and can't be resumed again.
There is a language construct in Vortex called `yield`. Yielding a coroutine results in the thread object being suspended at the point of yield. You can then again resume the coroutine and it'll run until the next yield.
You can pass one or more expressions to `yield`. The grammar is defined in the language section. The `resume` function will return values of the expressions passed to `yield`. It also returns the return value of the function when the thread object dies. `Yield` thus has the same semantics as `return` when it comes to resuming, except that it only suspends the coroutine instead of making it die.
You can also pass expressions to `resume`. They'll be returned inside the coroutine by the yield. An example:

```
// special syntax for coroutines provided by Vortex
// coroutines can also be created from functions using
// special calls with the same effect
let my_coroutine = coro a, b do
    print("AB", a, b)
```

```
    let (c, d) = yield (5, 10)
    print("CD", c, d)
    return (15, 20)
end
print(resume(my_coroutine, 2, 3))
print(resume(my_coroutine, 6, 7))
print(resume(my_coroutine, 8, 9))
```

This will print something like

```
AB 2 3
true 5 10
CD 6 7
true 15 20
false cannot resume dead coroutine
```

What exactly happened here? First, we resumed the coroutine, passing values `2` and `3` to it. Here it was like a function call. The passed values became `a` and `b`. The coroutine then printed them and yielded – suspended itself, passing back values `5` and `10` and expecting `c` and `d` for the next resume. The **`resume`** function then printed true (**`resume`** returns the state the coroutine was in – whether it was dead or alive as the first result) plus the yielded values.
Then we resumed once again. The values passed to **`resume`** became `c` and `d` as expected. The coroutine printed the values and returned more values. The coroutine has died by now. The **`resume`** call once again returned **`true`** (because it was alive at the point of execution) and the return values. A third resume returned **`false`**, because the coroutine was already dead, plus an error message.
That's coroutines in a nutshell. There are two variants of coroutines in Vortex, regular coroutines and generators. There's a subtle difference in the amount of power they give you and in the simplicity of writing. Please refer to the later sections.

# 3 Lexical

Vortex utilizes a fully free form syntax. Whitespace of any form is ignored, used only to delimit tokens where it would otherwise be ambiguous.

## 3.1 Encoding

⟨*alpha*⟩ ::= '`a-zA-Z`'

⟨*digit*⟩ ::= '`0-9`'

⟨*hexdigit*⟩ ::= '`0-9a-fA-F`'

The input is a sequence of bytes. Vortex does not handle Unicode in any way, but it's UTF-8 clean. All of the grammar is confined to the ASCII. Because it's UTF-8 clean, Unicode strings and such are allowed and passed to the output without any processing. If an UTF-8 BOM is found, it's skipped automatically. If a shebang line is found in the beginning, it's skipped as well.

## 3.2 Whitespace

$\langle whitespace \rangle$ ::= ‘ ’
| ‘n’
| ‘r’
| ‘t’
| ‘f’
| ‘v’

The Vortex lexer does not care about whitespace for anything else than token separation. Whitespace is not needed to separate tokens; the source is simply read character by character and when a token ending is found, it just goes on to the next one. The BNF above shows all possible forms of whitespace in Vortex. As you can see, newlines are treated as regular whitespace as well (with the exception of incrementing the line number). Newline and carriage return can be used in pairs no matter what order (handles the common cases '\n' and '\r\n' plus '\n\r' for rare platforms).

## 3.3 Comments

$\langle comment \rangle$ ::= ‘//’ /.*$/
| ‘/*’ { /./ | $\langle comment \rangle$ } ‘*/’

Vortex uses C(++) style comments. Short comments comment out everything until the end of the line. Long comment can span multiple lines and are enclosed between delimiters. Unlike C(++), Vortex allows for nesting of comments (thus it requires you to keep them balanced). Comments do not get past lexical analysis.

## 3.4 Identifiers

$\langle ident \rangle$ ::= (‘_’ | $\langle alpha \rangle$) { ‘_’ | $\langle alpha \rangle$ | $\langle digit \rangle$ }

$\langle identlist \rangle$ ::= $\langle ident \rangle$ { ‘,’ $\langle ident \rangle$ }

Vortex identifiers (names) can consist of alphanumeric characters (ASCII only) and underscores. They can't start with a digit, but a digit can be present anywhere else in the identifier. Identifiers starting with an underscore followed by an uppercase character are reserved. This is by convention and not enforced by the compiler.

## 3.5 Keywords

$\langle opkeyword \rangle$ ::= ‘band’
| ‘bor’
| ‘bxor’
| ‘asr’
| ‘bsr’
| ‘bsl’
| ‘and’
| ‘or’

$\langle opkeywordass \rangle$ ::= ‘band=’
| ‘bor=’
| ‘bxor=’
| ‘asr=’

| 'bsr='
| 'bsl='

⟨*keyword*⟩ ::= 'as'
| 'break'
| 'case'
| 'cfn'
| 'clone'
| 'coro'
| 'cycle'
| 'do'
| 'else'
| 'end'
| 'enum'
| 'false'
| 'fn'
| 'for'
| 'glob'
| 'goto'
| 'if'
| 'in'
| 'let'
| 'loop'
| 'macro'
| 'match'
| 'module'
| 'new'
| 'nil'
| 'quote'
| 'rec'
| 'redo'
| 'result'
| 'return'
| 'seq'
| 'set'
| 'true'
| 'unquote'
| 'when'
| 'while'
| 'with'
| 'yield'
| '__FILE__'
| '__LINE__'
| ⟨*opkeyword*⟩
| ⟨*opkeywordass*⟩
| ⟨*unopkeyword*⟩

⟨*unopkeyword*⟩ ::= 'not'
| 'bnot'

These are "reserved words" in Vortex. They cannot be used as identifiers. Some of these specified in `opkeyword` have an assignment form. Vortex is case sensitive, the keywords only are keywords in lowercase. For example, `match` is a keyword but `MATCH` or `Match` can be used as valid identifiers.

## 3.6 Other tokens

⟨*binop*⟩ ::= ‘=’
| ‘==’
| ‘>’
| ‘>=’
| ‘<’
| ‘<=’
| ‘!=’
| ‘%’
| ‘ ’
| ‘+’
| ‘++’
| ‘*’
| ‘**’
| ‘-’
| ‘/’
| ‘::’
| ⟨*opkeyword*⟩

⟨*assop*⟩ ::= ‘ =’
| ‘+=’
| ‘++=’
| ‘*=’
| ‘**=’
| ‘-=’
| ‘/=’
| ‘::=’
| ‘%=’
| ⟨*opkeywordass*⟩

⟨*unop*⟩ ::= ‘-’
| ‘#’
| ⟨*unopkeyword*⟩

⟨*othertok*⟩ ::= ‘(’
| ‘)’
| ‘->’
| ‘.’
| ‘..’
| ‘...’
| ‘;’
| ‘;;’
| ‘$’
| ‘$(’

Here you can see all the other tokens used by Vortex with separate categories for binary, unary and assignment operators.

## 3.7 Number literals

⟨*hexnum*⟩ ::= (‘0x’ | ‘0X’) { ⟨*hexdigit*⟩ } [ ‘.’ ] { ⟨*hexdigit*⟩ } [ (‘p’ | ‘P’) [ ‘+’ | ‘-’ ] ⟨*digit*⟩ ]

⟨*decnum*⟩ ::= { ⟨*digit*⟩ } [ ‘`.`’ ] { ⟨*digit*⟩ } [ (‘`e`’ | ‘`E`’) [ ‘`+`’ | ‘`-`’ ] ⟨*digit*⟩ ]

⟨*numliteral*⟩ ::= ⟨*hexnum*⟩
  | ⟨*decnum*⟩

Vortex does not make a difference between integers and floats. Numbers follow the Lua format. Hexadecimal numbers start with `0x` or `0X` (the former is better). In numbers like `0.6` you can omit the zero and write just `.6`. Hexadecimal constants behave similarly. Optional decimal exponent is marked with `e` or `E` (binary exponent in hex constants is `p` or `P`).

## 3.8 String literals

⟨*strliteral*⟩ ::= ⟨*strprefix*⟩ (⟨*strlong*⟩ | ⟨*strshort*⟩)

⟨*strprefix*⟩ ::= /[eErR]*/

⟨*strshort*⟩ ::= ‘`"`’ ⟨*strshortelem*⟩ ‘`"`’
  | ‘`'`’ ⟨*strshortelem*⟩ ‘`'`’

⟨*strlong*⟩ ::= ‘`"""`’ ⟨*strlongelem*⟩ ‘`"""`’
  | ‘`'''`’ ⟨*strlongelem*⟩ ‘`'''`’

⟨*strshortelem*⟩ ::= ? short string contents ?
  | ⟨*stresc*⟩

⟨*strlongelem*⟩ ::= ? long string contents ?
  | ⟨*stresc*⟩

⟨*stresc*⟩ ::= ‘`a`’
  | ‘`b`’
  | ‘`f`’
  | ‘`n`’
  | ‘`r`’
  | ‘`t`’
  | ‘`v`’
  | ‘`z`’
  | ‘`"`’
  | ‘`'`’
  | ‘’

String literals in Vortex are similar to Python's. They are represented by the `<string>` token in the final stream. There are two types of string literals, short and long literals.

Short literals are delimited with either single or double quotes and typically hold a single line. They interpret escape sequences. A backslash at the end of the line can be used to make them span multiple lines. You need to escape nested single or double quotes depending on the used delimiter (e.g. single quote delimited strings need to escape nested single quotes but not double quotes).

Long literals behave similarly. They're delimited by three repeated either single or double quotes. Escape sequences work the same, and long literals can span multiple lines without backslashes. You don't need to escape quotes except when three subsequent quotes are used (because otherwise they'd terminate the string).

You can prefix both types of literals with either `e` or `r` (and uppercase versions – no difference there). The former enables interpolation on that string – it will interpret `$var` and `$(expr)` in the string, where `var` is any Vortex variable you can access at that point and `expr` is any Vortex expression. For example

```
for k, v in pairs([ 5, 10, 15 ]) -> print(e"$k -> $(v + 2)")
```

will print

```
1 -> 7
2 -> 12
3 -> 17
```

Interpolated strings allow you to escape the dollar sign to prevent interpolation.
The latter prefix turns the strings into raw strings. That means no escape sequences are interpreted and instead they apparear in the result verbatim. If you escape quotes, the backslashes will be visible in the string. The same applies about backslashes used to escape newlines in short literals.
There is one non-standard escape sequence, `\z`. It skips the following span of whitespace characters (including newlines) in both short and long literals. That is useful to break a short string literal into multiple lines and indent the lines without actually including the newlines and indentation in the string.
You can also insert an arbitrary byte in the string. That can be done either with an escape sequence `\xXX`, where `XX` is a sequence of two hexadecimal digits (e.g. `\x4F` for uppercase O) or with an escape sequence `\ddd` where `ddd` is a sequence of up to three decimal digits (`\79` would be the uppercase O).

# 4 Expressions

$\langle exp \rangle$ ::= $\langle statexp \rangle$

$\langle explist \rangle$ ::= [ $\langle exp \rangle$ { ',' $\langle exp \rangle$ } ]

Vortex is a language that consists purely of expressions. No statements are present in the language in that sense an expression of any type can appear in any context. One exception is block scopes. Only expressions that can cause some sort of side effect can appear there (e.g. variable assignment, function call and so on, in the BNF they're called `statexp`). The rationale for this is that doing it otherwise doesn't really make sense – allowing inclusion of arbitrary expressions in blocks would result in code doing nothing, and that's better filtered out at compile time.

## 4.1 Main scope

$\langle chunk \rangle$ ::= { $\langle statexp \rangle$ [ ';' ] } [ ($\langle retexp \rangle$ | $\langle resexp \rangle$) [ ';' ] ]

$\langle mainchunk \rangle$ ::= { ($\langle statexp \rangle$ | $\langle macro \rangle$) [ ';' ] } [ ($\langle retexp \rangle$ | $\langle resexp \rangle$) [ ';' ] ]

The parsing of Vortex begins in the main scope. The main scope is a chunk. A chunk is a sequence of side effect based expressions optionally separated with semicolons. It may be ended with either a `return` or a `result` expression. Having another expression after one of these results in syntax error.
The main chunk can unlike any other chunk define macros, which are described in their own section later.

## 4.2 Blocks

⟨*blockend*⟩ ::= ‘`end`’ | ‘`;;`’

⟨*block*⟩ ::= ‘`do`’ ⟨*chunk*⟩ ⟨*blockend*⟩

⟨*expscope*⟩ ::= ‘`->`’ ⟨*exp*⟩ | ⟨*block*⟩

⟨*statscope*⟩ ::= ‘`->`’ ⟨*statexp*⟩ | ⟨*block*⟩

⟨*statexp*⟩ ::= ⟨*block*⟩

Blocks represent chains of expressions. A block consists of a `do` keyword, a chunk (see above) and either an `end` keyword or two semicolons. The semicolon ending is useful in inline blocks or in Lisp style formatted code. Blocks themselves are expressions. The `result` expression can be used to specify their value.

## 4.3 Return and result expressions

⟨*exporlist*⟩ ::= (⟨*exp*⟩ | ‘`(`’ ⟨*explist*⟩ ‘`)`’)

⟨*retexp*⟩ ::= ‘`return`’ ⟨*exporlist*⟩

⟨*resexp*⟩ ::= ‘`result`’ ⟨*exporlist*⟩

⟨*statexp*⟩ ::= ⟨*retexp*⟩ ⟨*resexp*⟩

These two expressions are used to manipulate values. The `return` expression jumps out of a function, making it return value(s) (in the main scope it makes the module return a value later used with `require`).
The `result` expression is very similar at first. It however works on scope level – it basically sets a value the current block will evaluate to. With regular functions, this is pretty much the same (as a block is typically a function return value) but you can see the difference when working with expression blocks. For example:

```
// this function returns 7, x is 5.
fn foo() do
    let x = do
        result 5
    end
    return x + 2
end

// this function returns 5, never reaching the final return.
fn bar() do
    let x = do
        return 5
    end
    return x + 2
end
```

## 4.4 Binary and unary expressions

⟨*assexp*⟩ ::= (⟨*ident*⟩ | ⟨*indexp*⟩) ⟨*assop*⟩ ⟨*exp*⟩

⟨*binexp*⟩ ::= ⟨*exp*⟩ ⟨*binop*⟩ ⟨*exp*⟩

$\langle unexp \rangle$ ::= $\langle unop \rangle$ $\langle exp \rangle$

$\langle exp \rangle$ ::= $\langle binexp \rangle$
| $\langle unexp \rangle$

$\langle statexp \rangle$ ::= $\langle assexp \rangle$

Binary expressions are expressions that consist of two operands and an operator. The operator is in infix form. The BNF here does not describe operator precedences. Assignment operators are treated differently as it's required to ensure that the left operand is an lvalue (you can't assign an arbitrary expression). Assignment expressions can also be used in statement form, unlike any other binary expression.
Unary expressions are expressions that consist of an operand and an operator in prefix form.
Here you can see operator precedences for binary and unary operators in Vortex.

| **Operator** | **Precedence** | **Associativity** |
|---|---|---|
| =, +=, -=, *=, /=, %=, ~=, ++=, ::=, **=, band=, bor=, bxor=, asr=, bsr=, bsl= | 1 | right |
| or | 2 | left |
| and | 3 | left |
| ==, != | 4 | left |
| <, <=, >, >= | 5 | left |
| ~ | 6 | right |
| bor | 7 | left |
| bxor | 8 | left |
| band | 9 | left |
| asr, bsr, bsl | 10 | left |
| +, - | 11 | left |
| *, /, % | 12 | left |
| ++ | 13 | left |
| :: | 14 | right |
| -, not, #, bnot | 15 | unary |
| ** | 16 | right |

Most of the operators should be clear when it comes to meaning. Assignment operators in form `lhs op= rhs` are equivalent to `lhs = lhs op rhs`. `~` means concatenation, `++` is a join operator (for tables), `::` is a cons operator (as in Lisp), `#` retrieves the length of the given expression, `**` raises lhs to a power of rhs. `band`, `bor`, `bxor`, `asr`, `bsr`, `bsl` mean bitwise AND, bitwise OR, bitwise XOR, arithmetic right shift, bitwise right shift and bitwise left shift respectively. The rest of the operators is functionally equivalent to those in C.

## 4.5 Tables and lists

$\langle tableitem \rangle$ ::= [ ([ '`$`' ] $\langle ident \rangle$ | '`$(`' $\langle exp \rangle$ '`)`') '`:`' ] $\langle exp \rangle$

$\langle tableexp \rangle$ ::= '`{`' [ $\langle tableitem \rangle$ { '`,`' $\langle tableitem \rangle$ } ] '`}`'

$\langle listexp \rangle$ ::= '`[`' $\langle exp \rangle$ { '`,`' $\langle exp \rangle$ } ] '`]`'

Vortex provides two types of built-in data structures. They're tables and lists.
Tables work in the same way as in Lua. They are something between an array, an associative array (unordered hash map) and an object. Array and hash elements can be both present in a single table. Tables are already described in their own section above.
Syntactically, table literals are enclosed in curly braces. If you want a value that has a key, you use the colon syntax. For example:

```
let array = { 5, 10, 15 } // an array of 3 elements
assert(array[1] == 5 and array[3] == 15)

// contains keys "foo", "abcd", "bar"
let assarray = { foo: "bar", $("ab" ~ "cd"): "baz", $bar: "xyz" }

// combined
let comb = { 5, 10, hello: "world", 15 }
```

As you can see, keys can be arbitrary expressions. Keys can be any value except `nil`.
Arrays count from 1. They're basically associative arrays with keys that are numbers, however, Lua optimizes this by storing array elements in their own section. You can assign to a table as you need. Array length is retrieved using the `#` operator. For example, this way you can append:

```
let array = { 5, 10, 15 }
array[#array + 1] = 20
```

Tables can have metatables as mentioned in the section above.
Lists are another data structure of Vortex. They use square brackets. Vortex lists are singly linked lists in concept similar to Lisp lists. You can construct a list multiple ways:

```
// using the list syntax
let lst = [ 5, 10, 15, 20 ]
// using the cons binary operator
let tsl = 5 :: 10 :: 15 :: 20 :: nil
```

Both ways are equivalent. A list consists of the "head" element and the "tail" element. Here, the first head is 5, the tail is another list where the head is 10, the tail is another list, the head again is 25, followed by another list where the head is 20 and the tail is `nil`.
Sometimes lists are more efficient than just tables. It depends on the use. Lists are implemented using tables in the runtime.

## 4.6 Primary expressions

⟨*primaryexp*⟩ ::= '(' ⟨*exp*⟩ ')'
  | ⟨*tableexp*⟩
  | ⟨*listexp*⟩
  | '`$(`' ⟨*exp*⟩ ')'
  | '`$`' ⟨*ident*⟩
  | ⟨*ident*⟩ [ '`!`' '(' ⟨*explist*⟩ ')' ]
  | ⟨*numliteral*⟩
  | ⟨*strliteral*⟩
  | '`nil`'
  | '`true`'
  | '`false`'

⟨*exp*⟩ ::= ⟨*primaryexp*⟩

Primary expressions are simple expressions that can have a suffix. A suffix is for example a parameter list (a call), brackets with an expression (indexing) or a dot (simple indexing). Primary expressions typically don't have a side effect, thus they can't be used in statement form unless postfixed as a call. You can wrap any arbitrary expression in parens to get a primary expression. All types of simple literals as well as table and list literals and macro expansions are primary expressions.

## 4.7 Suffixed expressions

⟨*fcallsuffix*⟩ ::= ‘`(`’ ⟨*explist*⟩ ‘`)`’
  | ⟨*tableexp*⟩
  | ⟨*strliteral*⟩

⟨*mcallsuffix*⟩ ::= ‘`:`’ ⟨*ident*⟩ ⟨*fcallsuffix*⟩

⟨*tcallsuffix*⟩ ::= ⟨*fcallsuffix*⟩
  | ⟨*mcallsuffix*⟩

⟨*indexsuffix*⟩ ::= ‘`.`’ ⟨*ident*⟩
  | ‘`[`’ ⟨*exp*⟩ ‘`]`’

⟨*expsuffix*⟩ ::= [ ⟨*expsuffix*⟩ ] (⟨*indexsuffix*⟩ | ⟨*tcallsuffix*⟩)

⟨*suffixedexp*⟩ ::= ⟨*primaryexp*⟩ ⟨*expsuffix*⟩

⟨*exp*⟩ ::= ⟨*suffixedexp*⟩

⟨*statexp*⟩ ::= ⟨*suffixedexp*⟩ ⟨*tcallsuffix*⟩

Suffixed expressions are primary expressions with a suffix. A suffix represents either a call or indexing. Suffixes can be chained. In statement form, only calls are allowed (not indexing alone). Indexing can be represented in two forms. The simpler form consists of an expression, a dot and a name. The name must be a valid indentifier. The more comprehensive form consists of an expression and an index enclosed in brackets (the index can be an arbitrary expression). Writing `foo.bar` is equivalent to `foo["bar"]` assuming `"bar"` contains no non-identifier characters.

Calls can be either method calls `obj:mname(args)` or regular calls `funcname(args)`. The former is just syntactic sugar for `obj.mname(obj, args)`. If the sole argument is a table or string literal, you can omit the parens. That means writing `funcname "foo"` and `funcname { 5, 10, 15 }` is equivalent to `funcname("foo")` and `funcname({ 5, 10, 15 })` respectively.

## 4.8 Let expression

⟨*lettype*⟩ ::= ‘`rec`’
  | ‘`glob`’

⟨*letexp*⟩ ::= ‘`let`’ [ ⟨*lettype*⟩ ] (⟨*pattern*⟩ | ‘`(`’ ⟨*patternlist*⟩ ‘`)`’) ‘`=`’ ⟨*exporlist*⟩

⟨*statexp*⟩ ::= ⟨*letexp*⟩

The `let` expression provides means to declare and define variables. It consists of the keyword, a pattern or a pattern list enclosed in parens, an assignment operator and either a single expression or an expression list enclosed in parens.

You can also optionally provide modifier after the keyword. The modifier can currently be either `rec` or `glob`. The former is best used with functions – it makes it possible for a variable to access itself and that way you can define recursive functions. Normally functions can't access themselves. The latter is used to define a global variable – by default, all Vortex variables are local.

You can't normally declare a variable without definition. However, you can assign nil to it, which is the same in meaning.

For patterns, look up the section about pattern matching. Note that only patterns that never fail to match can be used with `let`. For convenience, the table pattern always matches in let

but doesn't have to in regular pattern matching (that is because in let you sometimes want to extract just a few elements of an array, but in pattern matching you want it precise).
An example of let expression:

```
let x = 5; // local variable x
let glob y = 10; // global variable y
fn foo() -> [ 5, 10, 15 ]
let [ a, b ] = foo() // a is 5, b is 10 - pattern usage
fn rec bar() -> bar() // recursive
```

## 4.9 Set expression

⟨*setexp*⟩ ::= '`set`' (⟨*suffixedexp*⟩ | '(' ⟨*explist*⟩ ')') ⟨*assop*⟩ ⟨*exporlist*⟩

⟨*statexp*⟩ ::= ⟨*setexp*⟩

The `set` expression assigns values to variables. To assign a single variable, you can use the regular assignment binary expression. The `set` expression is used to set multiple variables at once. Like an assignment expression, the expressions on the left have to be lvalues. This expression looks quite similar to the `let` expression.
The main benefit of setting multiple variables at once is for e.g. swapping values. Consider this:

```
let a = 5
let b = 10
// now let's swap a and b
let tmp = a
a = b
b = tmp
```

This doesn't look too good. Isn't there a better way? Of course there is.

```
let a = 5
let b = 10
// and now let's swap
set (a, b) = (b, a)
```

Much better, right? The `set` expression evaluates to its left side after assignment.

## 4.10 With expression

⟨*withexp*⟩ ::= '`with`' (⟨*pattern*⟩ | '(' ⟨*patternlist*⟩ ')') '`=`' ⟨*exporlist*⟩ ⟨*expscope*⟩

⟨*withstat*⟩ ::= '`with`' (⟨*pattern*⟩ | '(' ⟨*patternlist*⟩ ')') '`=`' ⟨*exporlist*⟩ ⟨*statscope*⟩

⟨*statexp*⟩ ::= ⟨*withstat*⟩

⟨*exp*⟩ ::= ⟨*withexp*⟩

This expression is sort of similar to `let`. It represents a scope-bound variable. Unlike `let`, it doesn't evaluate to its left side, but instead to its expression. Consider this:

```
let x = with y = a() do
    print("hello world!")
    result 5
end
```

Here, the variable `y` is bound to the `with` expression scope, being invisible from anywhere else. The variable `x` will be `5`. Note that the scope will always evaluate, even when `y` is `nil`.

## 4.11 Functions

⟨*defarglist*⟩ ::= ⟨*ident*⟩ ‘`=`’ ⟨*exp*⟩ { ‘`,`’ ⟨*ident*⟩ ‘`=`’ ⟨*exp*⟩ }

⟨*arglist*⟩ ::= [ ⟨*identlist*⟩ ] [ ⟨*defarglist*⟩ ] [ [ ⟨*ident*⟩ ] ‘`...`’ ]

⟨*fnscope*⟩ ::= ‘`->`’ (⟨*exp*⟩ | ⟨*matchbody*⟩) | ⟨*block*⟩

⟨*fnliteral*⟩ ::= ‘`fn`’ (⟨*arglist*⟩ | ‘`(`’ ⟨*arglist*⟩ ‘`)`’) ⟨*fnscope*⟩

⟨*fndef*⟩ ::= ‘`fn`’ ([ ⟨*lettype*⟩ ] ⟨*ident*⟩ | ⟨*ident*⟩ [ ‘`.`’ ⟨*ident*⟩ ]) ‘`(`’ ⟨*arglist*⟩ ‘`)`’ ⟨*fnscope*⟩

⟨*statexp*⟩ ::= ⟨*fndef*⟩

⟨*exp*⟩ ::= ⟨*fnliteral*⟩

Vortex features first class functions. That means a function can be treated as a first class citizen – you can pass it as an argument or return it from another function. Like in Lua, functions are passed by reference – you never access the function value directly, you only access its reference.
You can create a function two ways. The first way is an anonymous function. Because anonymous functions don't have names, you need to assign it to a variable. It looks like this:

```
let add = fn a, b -> a + b
print(add(5, 10))
```

You can optionally put parens around the argument list. The second way is a named function. Given the previous example, you can rewrite your function as:

```
fn add(a, b) -> a + b
```

Named functions can be defined as table members. For example:

```
let tbl = {}
fn tbl.foo() -> "hello world"
```

They can also have modifiers, the same ones as the `let` expression (unless defined as table members – then it doesn't make sense). Named function definitions follow the same rules as the `let` expression. By default, named functions are local.

```
let foo = fn -> foo() // won't work - non-recursive
let rec bar = fn -> bar() // works - bar previously declared
fn foo() -> foo() // won't work
fn rec bar() -> bar() // works
```

Named functions can be used as statements. Anonymous functions can't.
Now, if you look at the examples, you can see they're pretty much lambda expressions. They take inputs and they return the value specified after the arrow. You can combine that with blocks. Both `return` and `result` can be used to specify the function return value. The former simply jumps out of the function and makes it return the value, the latter specifies the value of the block which the function then returns. The result is pretty much the same.

```
fn foo() -> do
    ...
end
```

The arrow feels kinda superfluous. Vortex allows you to omit it:

```
fn foo() do
    ...
end
```

Function arguments in Vortex can have default values. After you specify the first default value, every argument after that one must specify a default value. Example:

```
// you can reference the previous arguments too
fn foo(a, b = 5, c = b + 2, d = c + 3) do
    ...
end
```

You can end the argument list with an ellipsis argument. That means the function will be variadic and you can access the remaining arguments passed to the function again using an ellipsis. Example:

```
fn printf(fmt, ...) do
    print(fmt:format(...))
end
```

Note that to pass all the variadic arguments to a function, the ellipsis must be the last argument of the function call. Otherwise just the first value of the tuple will be passed! You can make it pass a single value anywhere by capturing it in `let`, for example

```
print("Only the first value", let _ = (...))
```

Lua allows you to do this by wrapping the ellipsis in parens. I consider that quite dangerous and bug-prone, thus Vortex doesn't allow this. While wrapping the ellipsis in parens in legal, it'll always evaluate to multiple values, no matter what.
You can name the vararg tuple and pass it around as a table like this:

```
fn foo(a, b...) do
    print(a, b[1], b[2])
end
```

That is functionally equivalent to:

```
fn foo(a, ...) do
    let b = { ... }
    print(a, b[1], b[2])
end
```

Vortex functions have a shorthand pattern matching variant. Writing

```
fn arglist ->
    | patternlist -> exp
    | ...
```

is equivalent to

```
fn arglist -> match arglist ->
    | patternlist -> exp
    | ...
```

## 4.12 If expression

$\langle ifblock \rangle$ ::= ‘`do`’ $\langle chunk \rangle$

$\langle elseopt \rangle$ ::= ‘`else`’ [ ‘`->`’ ] exp

$\langle elsestatopt \rangle$ ::= ‘`else`’ [ ‘`->`’ ] statexp

$\langle ifexp \rangle$ ::= ‘`if`’ $\langle exp \rangle$ (‘`->`’ $\langle exp \rangle$ [ $\langle elseopt \rangle$ ] | $\langle ifblock \rangle$ (‘`end`’ | $\langle elseopt \rangle$))

⟨*ifstat*⟩ ::= ‘`if`’ ⟨*exp*⟩ (‘`->`’ ⟨*statexp*⟩ [ ⟨*elsestatopt*⟩ ] | ⟨*ifblock*⟩ (‘`end`’ | ⟨*elsestatopt*⟩))

⟨*statexp*⟩ ::= ⟨*ifstat*⟩

⟨*exp*⟩ ::= ⟨*ifexp*⟩

The `if` expression allows you to do structured programming by incorporating conditionals. Every `if` expression begins with the keyword, followed by a condition. The condition evaluates to either `true` or `false`. If it evaluates to `true`, it either evaluates to the expression that follows the condition (when in expression form) or simply executes the expression in statement form (in that case, only statement form expressions are allowed).
There can be an optional `else` part. That part is evaluated when the condition is not met and has the same semantics as the former. Note that the arrow is optional with `else`, even with regular expressions.
When using `if` with blocks and without arrows, the `else` keyword implicitly terminates the block scope. Using `end` explicitly ends the `if` expression.

## 4.13 Loops

⟨*loopcond*⟩ ::= ‘`while`’ ⟨*exp*⟩

⟨*loopexp*⟩ ::= ‘`loop`’ [ ⟨*loopcond*⟩ ] ⟨*statscope*⟩ [ ⟨*loopcond*⟩ ]

⟨*numforexp*⟩ ::= ‘`for`’ ⟨*ident*⟩ ‘`=`’ ⟨*exp*⟩ ‘`..`’ ⟨*exp*⟩ [ ‘`,`’ ⟨*exp*⟩ ] ⟨*statscope*⟩

⟨*genforexp*⟩ ::= ‘`for`’ ⟨*identlist*⟩ ‘`in`’ ⟨*explist*⟩ ⟨*statscope*⟩

⟨*statexp*⟩ ::= ⟨*loopexp*⟩
| ⟨*numforexp*⟩
| ⟨*genforexp*⟩

⟨*exp*⟩ ::= ‘`break`’
| ‘`cycle`’
| ‘`redo`’

Vortex has three types of loops. The simplest one is the `loop` loop. By default, it loops like this:

```
loop do
    print("I'm infinite!")
end
```

You can add two kinds of conditions to this kind of loop, a regular condition and a postcondition. Both work similarly, but with the postcondition the loop iterates at least once before evaluating the condition (it evaluates AFTER iteration) while with the regular condition it may never start (it evaluates BEFORE iteration). You can use both at once. Example:

```
let i = 1
let keep_iteratng = false

fn check_iterate() -> true

loop while i <= 10 do
    print("I'm no longer infinite...")
    i += 1
    keep_iterating = check_iterate()
end while keep_iterating
```

Then there is the numeric `for` loop. It iterates using a numeric range. Both the start and the end of the range are inclusive. For example:

```
for i = 1 .. 10 do
    print(i)
end
```

This prints numbers from one to ten. There is an optional third

step expression. By default it's 1.

```
// 0, 2, 4, 6, 8, 10
for i = 0 .. 10, 2 do
    print(i)
end
```

The loop never re-evaluates the input expressions – they're evaluated once when the loop starts. The `step` is particularly useful for backwards iteration. Writing

```
for i = start, stop, step -> statexp
```

is equivalent to

```
do
    let (start, stop, step) =
        (tonum(start), tonum(stop), tonum(step or 1))
    if not (start and stop and step) -> error()
    loop while (step > 0 and start <= stop)
    or (step <= 0 and start >= stop) do
        let v = start
        statexp
        start += step
    end
end
```

Finally, there is the generic `for` loop. It uses iterators and is compatible with Lua iterators. You have a list of identifiers (you can have as many as you want as long as the iterator handles them all). Then you have the expression list, which is typically a single expression (an iterator). Each iteration the iterator is called, returning a new set of values mapping to the inputs. Writing

```
for k, v in pairs(tbl) -> statexp
```

is equialent to

```
do
    let (fun, s, var) = pairs(tbl)
    loop do
        let (k, v) = fun(s, var)
        if k == nil -> break
        var = k
        statexp
    end
end
```

All types of loops accept both arrow notation and blocks, but even in arrow notation and expression form the expression past the arrow ALWAYS must be a statement. A loop is an expression, but it evaluates to `nil`.
You can control the loop using three expressions, `break`, `cycle` and `redo`. Using `break` you can stop the iteration at that point. Using `cycle` you can skip to the next iteration, incrementing counters or calling iterator as needed. Using `redo` you can achieve a similar

thing, but no counter is ever incremented (or iterator called), effectively restarting the current iteration. The `cycle` keyword is equivalent to `continue` in several other languages (such as C, C++ or JavaScript).

## 4.14 Pattern matching

⟨*objectpatitem*⟩ ::= ([ '`$`' ] ⟨*ident*⟩ | '`$(`' ⟨*exp*⟩ '`)`') [ '`:`' ([ '`$`' ] ⟨*ident*⟩ | '`$(`' ⟨*exp*⟩ '`)`') ]

⟨*objectpatbody*⟩ ::= ⟨*objectpatitem*⟩ { '`,`' ⟨*objectpatitem*⟩ }

⟨*tablepatitem*⟩ ::= [ ([ '`$`' ] ⟨*ident*⟩ | '`$(`' ⟨*exp*⟩ '`)`') '`:`' ] ⟨*pattern*⟩

⟨*primarypattern*⟩ ::= '`(`' ⟨*pattern*⟩ '`)`'
| ⟨*strliteral*⟩
| ⟨*numliteral*⟩
| '`true`'
| '`false`'
| '`nil`'
| '`_`'
| [ '`$`' ] ⟨*ident*⟩ [ '`(`' [ ⟨*objectpatbody*⟩ ] '`)`' ]
| '`$(`' ⟨*exp*⟩ '`)`' [ '`(`' [ ⟨*objectpatbody*⟩ ] '`)`' ]
| '`{`' ⟨*tablepatitem*⟩ { '`,`' ⟨*tablepatitem*⟩ } '`}`'

⟨*suffixedpattern*⟩ ::= ⟨*primarypattern*⟩
| ⟨*suffixedpattern*⟩ '`when`' ⟨*exp*⟩
| ⟨*suffixedpattern*⟩ '`as`' ⟨*exp*⟩

⟨*patternop*⟩ ::= '`and`'
| '`or`'
| '`::`'

⟨*pattern*⟩ ::= ⟨*suffixedpattern*⟩ [ ⟨*patternop*⟩ ⟨*suffixedpattern*⟩ ]

⟨*matchexp*⟩ ::= '`match`' ⟨*explist*⟩ '`->`' ⟨*matchbody*⟩

⟨*matchstat*⟩ ::= '`match`' ⟨*explist*⟩ '`->`' ⟨*matchbodystat*⟩

⟨*matcharm*⟩ ::= ('`|`' | '`case`') ⟨*patternlist*⟩ ⟨*expscope*⟩

⟨*matcharmstat*⟩ ::= ('`|`' | '`case`') ⟨*patternlist*⟩ ⟨*statscope*⟩

⟨*matchbody*⟩ ::= ⟨*matcharm*⟩ { ⟨*matcharm*⟩ }

⟨*matchbodystat*⟩ ::= ⟨*matcharmstat*⟩ { ⟨*matcharmstat*⟩ }

⟨*exp*⟩ ::= ⟨*matchexp*⟩

⟨*statexp*⟩ ::= ⟨*matchstat*⟩

Vortex provides pattern matching similarly to e.g. the MLs, Rust or Haskell. Pattern matching works like a generalized `switch` statement – you have a list of input expressions in the `match` expression and then you have some arms - arms start either with \| or with `case`, followed by a list of patterns (one pattern for each input, if you don't provide a pattern for some input then it always matches the input), then followed by either an expression after an arrow or a `do` block.

The same rules as with e.g. `if` apply – when `match` is used in a statement form, only statement form expressions are allowed.
The arms are evaluated from the top. The first arm where all patterns match is evaluated and then the evaluation stops. As you can see, it's similar to a `switch`, but there is no fallthrough. Also unlike for example OCaml, non-exhaustive patterns are NOT detected.
Pattern matching can be used for decomposition of data structures, besides regular mathching. For that purpose, lots of patterns are provided.

**Variable pattern**
: This is the simplest kind of pattern. It's just a name (an identifier) and it is used to capture the input into a variable (local to the specific arm). You can then use the input using that variable. It never fails to match.

**Wildcard pattern**
: This one is similar to variable pattern, except that no variable capture is made (it also never fails to match). It's an underscore in code.

**Expression pattern**
: This pattern is represented either as a string literal, a number literal, `true`, `false`, `nil`, an identifier prefixed by `$` or an expression enclosed in `$()`. It doesn't capture and it may fail to match (it tests the input for equality with the given expression).

**Table pattern**
: This pattern matches tables. It looks like a table literal. It can match both array and hash members, where array members are written in the same way as expressions in an array literal (except that they're patterns) and hash members are written in the same as well, where the key is an expression and the value is a pattern. When matching a table, you have to match all the array members, but you don't have to match all the hash members. It may fail to match.

**Object pattern**
: With this pattern you can match objects. It consists of an identifier or a `$` or a `$()` expression followed by parens that contain a list of object member captures. A capture can be a simple identifier (then it will capture a member of that name) or a `$` or a `$()` expression followed by a colon and then by an identifier – in that case it'll match a member with the key the expression evaluates to into a variable after the colon. You don't have to capture all object members. It may fail to match.

**Cons pattern**
: This binary pattern decomposes an input into two parts, head and tail. It has a reversed meaning to the cons operator. The input has to be either a table or a list. On a table this is slow, on a list this is fast (in case of table it has to actuall slice the table). On invalid input it fails to match.
Binary patterns have their precedence and associativity. The cons pattern has the highest precedence and is right associative. It looks the same as the cons operator.

**And, or patterns**
: Binary patterns. The `or` pattern has the lowest precedence, the `and` pattern is above it, followed by the cons pattern. These are all binary patterns in Vortex. Both `and` and `or` patterns are left associative. They may fail to match.

**When pattern**
: This pattern is a conditional pattern. It consists of a pattern followed by the keyword `when` followed by a condition (which is an expression). It may obviously fail to match.

**As pattern**
: This pattern essentially captures a pattern. It consists of a pattern followed by the `as` keyword followed by another pattern. It's particularly useful if you have for example an expression pattern that doesn't capture the input and you still want to capture it besides checking its equality with the expression.

Some example code for pattern matching:

```
let x = 5
// prints 5 - variable pattern
print(match x -> | y -> y)

// prints 10, expression and wildcard patterns
print(match 4 ->
    | 1 -> 2
    | 2 -> 3
    | 3 -> 4
    | _ -> 10)

// prints 5
print(match 4 ->
    | $(2 + 3) as x -> x
    | $(2 * 2) as y -> y + 1)

// prints nil, doesn't match
print(match 10 ->
    | x when x == 5 -> x)

let tbl = { 5, 10, 15, foo: "bar", bar: "baz" }
// prints "baz", first arm is incomplete
print(match tbl ->
    | { a, b, foo: c } -> a
    | { a, b, c, bar: d } -> d)

let list = [ 5, 10, 15, 20 ]
// reversed list
let x = match list ->
    | a :: b :: c :: d -> [ d, c, b, a ] )
```

As previously mentioned, the `let` expression also makes use of patterns. The pattern use is limited though – only patterns that never fail to match can be used. The table pattern is modified appropriately to allow use with `let` – you can match partial contents of an array with it.

## 4.15 Objects

⟨*objectparents*⟩ ::= ⟨*suffixedexp*⟩
  | ‘`(`’ ⟨*explist*⟩ ‘`)`’

⟨*objectimplctor*⟩ ::= ‘`[`’ [ ⟨*identlist*⟩ ] ‘`]`’

⟨*objectitem*⟩ ::= ⟨*fndef*⟩
  | ([ ‘`$`’ ] ⟨*ident*⟩ | ‘`$(`’ ⟨*exp*⟩ ‘`)`’) ‘`:`’ ⟨*exp*⟩

⟨*objectbody*⟩ ::= ⟨*objectimplctor*⟩ [ ‘`do`’ { ⟨*objectitem*⟩ } ‘`end`’ ]
  | ‘`do`’ { ⟨*objectitem*⟩ } ‘`end`’

⟨*objectexp*⟩ ::= ‘`clone`’ [ ⟨*objectparents*⟩ ] ⟨*objectbody*⟩

⟨*newexp*⟩ ::= ‘`new`’ (⟨*primaryexp*⟩ | ‘`(`’ ⟨*explist*⟩ ‘`)`’)

⟨*exp*⟩ ::= ⟨*objectexp*⟩
  | ⟨*newexp*⟩

Vortex features a builtin object system. It’s prototype based, delegative and supports multiple inheritance. It also has operator overloading. The object system is based around the `clone` expression. It clones a parent object (or a set of parent objects in case of multiple inheritance). If you don't provide a parent object, it inherits from the internal Object table (which provides the basic stuff concerning objects). That means Object is the base for every user defined object.

When inheriting from multiple parents, non-existent members are looked up from parents from left to right depth-first.

The `clone` expression cannot be used as a statement. Objects are first class values and the `clone` expression evaluates to them, so you use `clone` in combination with `let`.

Objects feature constructors, but they don't have destructors. There are GC finalizers provided in the same manner as the `__gc` metamethod with tables. There is an implicit constructor syntax – you can provide a list of identifiers in square brackets that represent member names and Nth constructor argument will become Nth member in the brackets.

Constructors are never called when cloning an object.

There is also the `new` expression, which creates an "instance" of the object – that means, it clones the object and calls the parent constructor on it. That allows you to pretty much closely emulate classes without losing the flexible prototypal nature of Vortex's object system.

Vortex provides the `super` function in the standard library. It returns a proxy object on which one can call parent methods. You can provide either one or two arguments. If you have an object `x`, which is an instance of `Bar`, which is a clone of `Foo`, calling `super(x):abc()` calls a method of `Foo` (instead of `Bar`, which is a parent of `x`) and it clals it on `x`. If you provide two arguments, the first one is an object – a clone, the other one is the object on which we want to call. Calling `super(Bar, x)` and `super(x)` is equivalent in this case. Some examples:

```
let Foo = clone do
    fn __init(self, a, b) do
        print("I'm a constructor!")
        set (self.a, self.b) = (a, b)
    end

    fn foo() do
        print("I'm a method without self, all alone")
    end

    fn bar(self) -> print("I'm a method!", self.a, self.b)
end

let Bar = clone Foo do
    fn xyz(self) -> Foo.bar(self)
end

let Baz = clone [ a, b, c ]

let inst = new Baz(5, 10, 15)
print(inst.a, inst.b, inst.c)
```

## 4.16 Coroutines

⟨*corobody*⟩ ::= '`->`' ⟨*exp*⟩
  | ⟨*arglist*⟩ ⟨*expscope*⟩

⟨*coroexp*⟩ ::= ('`coro`' | '`cfn`') ⟨*corobody*⟩

⟨*yieldexp*⟩ ::= ‘`yield`’ ⟨*exporlist*⟩

⟨*exp*⟩ ::= ⟨*coroexp*⟩

⟨*statexp*⟩ ::= ⟨*yieldexp*⟩

I already described coroutines above, here I'll describe them when it comes to their syntax. Vortex provides two variants of coroutines, the `coro` variant and the `cfn` variant. The former creates a true coroutine – a thread object that you can resume and so on. The latter creates a simplified function form – it resumes when you call it, and there is no way to query whether it's already dead (except that it returns nil when it is and you call it).
Each of these has two forms again. The arrow form (the keyword followed by an arrow and an expression) takes an arbitrary expression and makes it into a coroutine. The function form consists of the keyword, a function argument list (identical with a regular function) and a function body (an arrow followed by an expression or a block with or without arrow). That one is just a shorthand, for example `cfn a, b -> body` is the same as `cfn -> fn a, b -> body`.
You can yield from a coroutine using the `yield` expression, which looks similar to the `return` or `result` expression. You can resume a thread object using the `resume` function in the core library. There is also the Lua library for coroutine manipulation which works just fine.

## 4.17 Sequences

⟨*seqexp*⟩ ::= ‘`seq`’ ⟨*expscope*⟩

⟨*exp*⟩ ::= ⟨*seqexp*⟩

Vortex features sequences. They evaluate to a tuple of expressions (not a first class value – it's similar to a function that returns multiple values). They're basically coroutines. They evaluate to the values you yield from them. That you can use for e.g. list comprehensions. Example:

```
// this table is an array of numbers from 1 to 10
let x = { seq -> for i = 1 .. 10 -> yield i }

// with blocks
let x = { seq do
    for i = 1 .. 5 -> yield i
    yield 6
    return 7 // also works, terminates the sequence
end }
```

## 4.18 Enumerations

⟨*enumitem*⟩ ::= ⟨*ident*⟩ [ ‘`:`’ ⟨*exp*⟩ ]

⟨*enumexp*⟩ ::= ‘`enum`’ ‘`(`’ ⟨*enumitem*⟩ { ‘`,`’ ⟨*enumitem*⟩ } ‘`)`’

⟨*exp*⟩ ::= ⟨*enumexp*⟩

Vortex also features enumerations. They start with the `enum` keyword followed by a list of identifiers in parens. An enumeration represents a sequence of numbers (or custom expressions). It evaluates to an associative array where the identifiers specified here are the keys.

Each of the keys has an associated value, by default numerical, starting from 0, incrementing by 1 with each following member. Note that the order when iterating an enum is undefined.

```
let Foo = enum (A, B, C)
assert(Foo.A == 0 and Foo.B = 1 and Foo.C = 2)

let Bar = enum (FOO: 5, BAR, BAZ: 10, BAH)
assert(Bar.FOO == 5 and Bar.BAR == 6
    and Bar.BAZ == 10 and Bar.BAH == 11)
```

The expression you optionally provide when defining an enum member should be incrementable.

## 4.19 Other expressions

⟨*exp*⟩ ::= '`__FILE__`'
  | '`__LINE__`'

There are two other expressions. They're very simple. The `__FILE__` expression expands to the current filename at compile time. The `__LINE__` expression expands to the line the expression is on, also at compile time.

# 5 Macros

⟨*macro*⟩ ::= '`macro`' ⟨*ident*⟩ [ ⟨*identlist*⟩ ] [ '`...`' ] ⟨*expscope*⟩

⟨*quoteexp*⟩ ::= '`quote`' (⟨*suffixedexp*⟩ | ⟨*block*⟩)

⟨*unquoteexp*⟩ ::= '`unquote`' (⟨*suffixedexp*⟩ | ⟨*block*⟩)

⟨*exp*⟩ ::= ⟨*quoteexp*⟩
  | ⟨*unquoteexp*⟩

Vortex features an AST based macro system. That means it operates on AST level rather than on text level – that way it can be context aware and much safer. The currently implemented macro system is not hygienic (it can capture and create outer identifiers) but it's planned.
A macro starts with the `macro` keyword followed by the macro name and an argument list. Then either an arrow followed by an expression or a block follows. This is very similar to regular functions, but there are no default argument values and no named varargs.
To use a macro, you need to expand it. You do that using the `macroname!(macroargs)` syntax (it's a primary expression). It means that the macro will expand at that point, substituting the expansion expression with some actual code.
As mentioned previously, a macro works on AST level – the macro has zero or more inputs (that are serialized AST nodes) and it's supposed to return an AST node, which is then injected into the final AST at the expansion point.
Macro arguments are implicitly serialized into AST nodes. The expression you return from the macro (using regular `return` or using the arrow notation) is not, so you need to serialize it. For that purpose, there is the `quote` expression. It takes an arbitrary expression and evaluates to the AST of it, converting any subexpressions into AST as well. For example

```
let x = quote (fn a, b do
    print("hello world")
end)
```

results in a function AST node with all the contents serialized as well as the function itself. There is one other expression, `unquote`. It basically tells the `quote` expression to "stop" at that point. For example:

```
let a = quote (x + y)
let b = quote (y * z)

// this is a binary expression with
// operator / containing two symbols
let c = quote (a / b)
// this is a binary expression with operator
// / containing two binary expressions, a and b
let d = quote (unquote a / unquote b)
```

With these facilities available, we can define a macro:

```
macro my_if(cond, texp, fexp) -> quote
    (match not unquote cond ->
        | true  -> unquote fexp
        | false -> unquote texp)

let x = 5
print(my_if!(x == 5, "hello", "world"))
```

Here it prints "hello", because the condition is true. The unquotes are needed so that we put the inputs themselves in the result, not just symbols.

## 6 The runtime and the standard library

This reference manual does not cover the runtime nor the standard library. It references some core functions, but does not define anything – please refer to the standard library documentation for more.

## 7 The REPL

Vortex provides an interactive command line and a standalone script runner combined into a single script. By default it launches an interactive session. There you can input statements. Local variables are preserved.

```
> let x = { 5, 10, { 15, 20 } }
> =x // proper output
{ 5, 10, { 15, 20 } }
```

The Vortex REPL in the default implementation is itself written in Vortex. It requires a Lua module called `vxutil`. This module is shipped with Vortex and is written in C – you need to compile it and then put into a Lua C module search path (for example the current directory). The C module provides support for signals, `isatty` and the readline library. All functions have their fallbacks. For example, if you want readline support in the REPL (so that you can move in the history and seek in the current input), you need to compile the C module with `VX_READLINE` defined. For proper function of `isatty` you need to compile the module with either `VX_POSIX` or `VX_WIN` defined depending on your platform. If you don't define either of these, a fallback will always return true, which means autodetection whether to launch an interactive session won't work (when `isatty` returns false, it means something is piped into the standard input, and the REPL tries to execute that).

```
usage: vortex [options] [script [args]]
-e str  run the string 'str'
```

```
-i       enter interactive mode after the options are handled
-l lib   require library 'lib'
-v       show version information
--       stop handling options
-        stop handling options and run stdin
```

Here you can see the REPL options you can pass to it. As mentioned above, you can use the REPL with pipes. For example

```
echo 'print("hello world")'|<replcommand>
```

should work.
If your interactive line starts with `=` followed by an expression, the REPL prints its value. You can see an example in the very beginning of this section. Tables get special treatment – they're serialized before printing. That should allow you to see the values clearly.

# 8 Appendix: Style guide

These are recommended conventions for Vortex code. You don't have to follow them, but it's strongly recommended for consistency.

## 8.1 Indentation

You use 4 spaces to indent each level. Tabs should be avoided. Try not to follow the Lua convention of 2 spaces.

## 8.2 Whitespace

Put a space before and after a binary operator. Do not space off unary operators. Do not put spaces around parens. You should put a space after each comma in the language. Do not put a space between function name and its argument list. You can use spaces to align things where needed.

## 8.3 Blocks

Keep the `do` keyword on the same line with the preceeding code.

## 8.4 Naming style

The Vortex naming rules are simple and encourage readable code.

- Values, modules, functions and variables generally are `snake_case`.
- Objects are `This_Case`.
- Macro names are `snake_case`. No need for uppercase because of obvious expansion syntax.

```
let foo = 5 // Good
let Bar = 10 // Bad!

fn foo_bar(a, b) -> expr() // Good
fn fooBar(a, b) -> expr() // Bad

// Good
let My_Object = clone do
    fn my_method(self) -> self.x
```

```
end
// Bad
let MyObject = clone do
    fn myMethod(self) -> self.x
end

// Good
macro foo(a, b) do ... end
print(foo!(5, 10))
// Bad
macro FOO(a, b) do ... end
print(FOO!(5, 10))
```

## 8.5 Examples

```
// Good!
let x = a + b

// Also good.
if foo do
    print("boo!")
end

// Alignment is good
foo(bar(),
    baz())

// Spacing makes things readable.
foo(bar(), baz(), (5 + 10 * (34 - 1) / 150))

// Functions, the good way
fn foo(a, b) -> hello(a, b)

// But this is bad.
let x = a+b

// This is bad too.
if foo
do
    print("boo")
end

// Bad alignment
foo(bar(),
        baz())

// Excessive spacing.
foo ( bar (), baz (), ( 5 + 10 * ( 34 - 1 ) / 150 ) )

// A badly written function
fn foo (a, b) -> hello ( a, b )
```

# 9 Appendix: The complete syntax of Vortex

```
⟨alpha⟩ ::= ‘a-zA-Z’

⟨digit⟩ ::= ‘0-9’

⟨hexdigit⟩ ::= ‘0-9a-fA-F’

⟨whitespace⟩ ::= ‘ ’
  | ‘n’
  | ‘r’
  | ‘t’
  | ‘f’
  | ‘v’

⟨comment⟩ ::= ‘//’ /.*$/
  | ‘/*’ { /./ | ⟨comment⟩ } ‘*/’

⟨ident⟩ ::= (‘_’ | ⟨alpha⟩) { ‘_’ | ⟨alpha⟩ | ⟨digit⟩ }

⟨identlist⟩ ::= ⟨ident⟩ { ‘,’ ⟨ident⟩ }

⟨opkeyword⟩ ::= ‘band’
  | ‘bor’
  | ‘bxor’
  | ‘asr’
  | ‘bsr’
  | ‘bsl’
  | ‘and’
  | ‘or’

⟨opkeywordass⟩ ::= ‘band=’
  | ‘bor=’
  | ‘bxor=’
  | ‘asr=’
  | ‘bsr=’
  | ‘bsl=’

⟨keyword⟩ ::= ‘as’
  | ‘break’
  | ‘case’
  | ‘cfn’
  | ‘clone’
  | ‘coro’
  | ‘cycle’
  | ‘do’
  | ‘else’
  | ‘end’
  | ‘enum’
  | ‘false’
  | ‘fn’
  | ‘for’
  | ‘glob’
  | ‘goto’
  | ‘if’
```

| ‘in’
| ‘let’
| ‘loop’
| ‘macro’
| ‘match’
| ‘module’
| ‘new’
| ‘nil’
| ‘quote’
| ‘rec’
| ‘redo’
| ‘result’
| ‘return’
| ‘seq’
| ‘set’
| ‘true’
| ‘unquote’
| ‘when’
| ‘while’
| ‘with’
| ‘yield’
| ‘__FILE__’
| ‘__LINE__’
| ⟨*opkeyword*⟩
| ⟨*opkeywordass*⟩
| ⟨*unopkeyword*⟩

⟨*unopkeyword*⟩ ::= ‘not’
| ‘bnot’

⟨*binop*⟩ ::= ‘=’
| ‘==’
| ‘>’
| ‘>=’
| ‘<’
| ‘<=’
| ‘!=’
| ‘%’
| ‘ ’
| ‘+’
| ‘++’
| ‘*’
| ‘**’
| ‘-’
| ‘/’
| ‘::’
| ⟨*opkeyword*⟩

⟨*assop*⟩ ::= ‘ =’
| ‘+=’
| ‘++=’
| ‘*=’
| ‘**=’

```
  | '-='
  | '/='
  | '::='
  | '%='
  | ⟨opkeywordass⟩

⟨unop⟩ ::= '-'
  | '#'
  | ⟨unopkeyword⟩

⟨othertok⟩ ::= '('
  | ')'
  | '->'
  | '.'
  | '..'
  | '...'
  | ';'
  | ';;'
  | '$'
  | '$('

⟨hexnum⟩ ::= ('0x' | '0X') { ⟨hexdigit⟩ } [ '.' ] { ⟨hexdigit⟩ } [ ('p' | 'P') [ '+' | '-' ]
    ⟨digit⟩ ]

⟨decnum⟩ ::= { ⟨digit⟩ } [ '.' ] { ⟨digit⟩ } [ ('e' | 'E') [ '+' | '-' ] ⟨digit⟩ ]

⟨numliteral⟩ ::= ⟨hexnum⟩
  | ⟨decnum⟩

⟨strliteral⟩ ::= ⟨strprefix⟩ (⟨strlong⟩ | ⟨strshort⟩)

⟨strprefix⟩ ::= /[eErR]*/

⟨strshort⟩ ::= '"' ⟨strshortelem⟩ '"'
  | ''' ⟨strshortelem⟩ '''

⟨strlong⟩ ::= '"""' ⟨strlongelem⟩ '"""'
  | ''''' ⟨strlongelem⟩ '''''

⟨strshortelem⟩ ::= ? short string contents ?
  | ⟨stresc⟩

⟨strlongelem⟩ ::= ? long string contents ?
  | ⟨stresc⟩

⟨stresc⟩ ::= 'a'
  | 'b'
  | 'f'
  | 'n'
  | 'r'
  | 't'
  | 'v'
  | 'z'
  | '"'
```

```
  |  '|'
  |  ''

⟨chunk⟩ ::= { ⟨statexp⟩ [ ';' ] } [ (⟨retexp⟩ | ⟨resexp⟩) [ ';' ] ]

⟨mainchunk⟩ ::= { (⟨statexp⟩ | ⟨macro⟩) [ ';' ] } [ (⟨retexp⟩ | ⟨resexp⟩) [ ';' ] ]

⟨macro⟩ ::= 'macro' ⟨ident⟩ [ ⟨identlist⟩ ] [ '...' ] ⟨expscope⟩

⟨blockend⟩ ::= 'end' | ';;'

⟨block⟩ ::= 'do' ⟨chunk⟩ ⟨blockend⟩

⟨expscope⟩ ::= '->' ⟨exp⟩ | ⟨block⟩

⟨statscope⟩ ::= '->' ⟨statexp⟩ | ⟨block⟩

⟨fnscope⟩ ::= '->' (⟨exp⟩ | ⟨matchbody⟩) | ⟨block⟩

⟨ifblock⟩ ::= 'do' ⟨chunk⟩

⟨elseopt⟩ ::= 'else' [ '->' ] exp

⟨elsestatopt⟩ ::= 'else' [ '->' ] statexp

⟨statexp⟩ ::= ⟨block⟩
  |  (⟨ident⟩ | ⟨indexp⟩) ⟨assop⟩ ⟨exp⟩
  |  ⟨suffixedexp⟩ ⟨tcallsuffix⟩
  |  'if' ⟨exp⟩ ('->' ⟨statexp⟩ [ ⟨elsestatopt⟩ ] | ⟨ifblock⟩ ('end' | ⟨elsestatopt⟩))
  |  'let' [ ⟨lettype⟩ ] (⟨pattern⟩ | '(' ⟨patternlist⟩ ')') '=' ⟨exporlist⟩
  |  'set' (⟨suffixedexp⟩ | '(' ⟨explist⟩ ')') ⟨assop⟩ ⟨exporlist⟩
  |  'with' (⟨pattern⟩ | '(' ⟨patternlist⟩ ')') '=' ⟨exporlist⟩ ⟨statscope⟩
  |  'fn' ([ ⟨lettype⟩ ] ⟨ident⟩ | ⟨ident⟩ [ '.' ⟨ident⟩ ]) '(' ⟨arglist⟩ ')' ⟨fnscope⟩
  |  'match' ⟨explist⟩ '->' ⟨matchbodystat⟩
  |  ⟨retexp⟩
  |  ⟨resexp⟩

⟨retexp⟩ ::= 'return' ⟨exporlist⟩

⟨resexp⟩ ::= 'result' ⟨exporlist⟩

⟨exp⟩ ::= ⟨statexp⟩
  |  ⟨exp⟩ ⟨binop⟩ ⟨exp⟩
  |  ⟨unop⟩ ⟨exp⟩
  |  ⟨primaryexp⟩
  |  ⟨suffixedexp⟩
  |  'match' ⟨explist⟩ '->' ⟨matchbody⟩
  |  'clone' [ ⟨objectparents⟩ ] ⟨objectbody⟩
  |  ('coro' | 'cfn') ('->' ⟨exp⟩ | ⟨arglist⟩ ⟨expscope⟩)
  |  'if' ⟨exp⟩ ('->' ⟨exp⟩ [ ⟨elseopt⟩ ] | ⟨ifblock⟩ ('end' | ⟨elseopt⟩))
  |  'with' (⟨pattern⟩ | '(' ⟨patternlist⟩ ')') '=' ⟨exporlist⟩ ⟨expscope⟩
  |  'loop' [ ⟨loopcond⟩ ] ⟨statscope⟩ [ ⟨loopcond⟩ ]
  |  'for' ⟨ident⟩ '=' ⟨exp⟩ '..' ⟨exp⟩ [ ',' ⟨exp⟩ ] ⟨statscope⟩
  |  'for' ⟨identlist⟩ 'in' ⟨explist⟩ ⟨statscope⟩
  |  'fn' (⟨arglist⟩ | '(' ⟨arglist⟩ ')') ⟨fnscope⟩
```

```
  | 'yield' ⟨exporlist⟩
  | 'seq' ⟨expscope⟩
  | 'new' (⟨primaryexp⟩ | '(' ⟨explist⟩ ')')
  | 'quote' (⟨suffixedexp⟩ | ⟨block⟩)
  | 'unquote' (⟨suffixedexp⟩ | ⟨block⟩)
  | 'enum' '(' ⟨ident⟩ [ ':' ⟨exp⟩ ] { ',' ⟨ident⟩ [ ':' ⟨exp⟩ ] } ')'
  | 'break'
  | 'cycle'
  | 'redo'
  | '__FILE__'
  | '__LINE__'

⟨primaryexp⟩ ::= '(' ⟨exp⟩ ')'
  | ⟨tableexp⟩
  | '[' ⟨exp⟩ { ',' ⟨exp⟩ } ] ']'
  | '$(' ⟨exp⟩ ')'
  | '$' ⟨ident⟩
  | ⟨ident⟩ [ '!' '(' ⟨explist⟩ ')' ]
  | ⟨numliteral⟩
  | ⟨strliteral⟩
  | 'nil'
  | 'true'
  | 'false'

⟨suffixedexp⟩ ::= ⟨primaryexp⟩ ⟨expsuffix⟩

⟨tableexp⟩ ::= '{' [ ⟨tableitem⟩ { ',' ⟨tableitem⟩ } ] '}'

⟨explist⟩ ::= [ ⟨exp⟩ { ',' ⟨exp⟩ } ]

⟨exporlist⟩ ::= (⟨exp⟩ | '(' ⟨explist⟩ ')')

⟨defarglist⟩ ::= ⟨ident⟩ '=' ⟨exp⟩ { ',' ⟨ident⟩ '=' ⟨exp⟩ }

⟨arglist⟩ ::= [ ⟨identlist⟩ ] [ ⟨defarglist⟩ ] [ [ ⟨ident⟩ ] '...' ]

⟨lettype⟩ ::= 'rec'
  | 'glob'

⟨loopcond⟩ ::= 'while' ⟨exp⟩

⟨tableitem⟩ ::= [ ([ '$' ] ⟨ident⟩ | '$(' ⟨exp⟩ ')') ':' ] ⟨exp⟩

⟨fcallsuffix⟩ ::= '(' ⟨explist⟩ ')'
  | ⟨tableexp⟩
  | ⟨strliteral⟩

⟨mcallsuffix⟩ ::= ':' ⟨ident⟩ ⟨fcallsuffix⟩

⟨tcallsuffix⟩ ::= ⟨fcallsuffix⟩
  | ⟨mcallsuffix⟩

⟨indexsuffix⟩ ::= '.' ⟨ident⟩
  | '[' ⟨exp⟩ ']'
```

⟨*expsuffix*⟩ ::= [ ⟨*expsuffix*⟩ ] (⟨*indexsuffix*⟩ | ⟨*tcallsuffix*⟩)

⟨*objectparents*⟩ ::= ⟨*suffixedexp*⟩
  | ‘`(`’ ⟨*explist*⟩ ‘`)`’

⟨*objectimplctor*⟩ ::= ‘`[`’ [ ⟨*identlist*⟩ ] ‘`]`’

⟨*objectitem*⟩ ::= ⟨*fndef*⟩
  | ([ ‘`$`’ ] ⟨*ident*⟩ | ‘`$(`’ ⟨*exp*⟩ ‘`)`’) ‘`:`’ ⟨*exp*⟩

⟨*objectbody*⟩ ::= ⟨*objectimplctor*⟩ [ ‘`do`’ { ⟨*objectitem*⟩ } ‘`end`’ ]
  | ‘`do`’ { ⟨*objectitem*⟩ } ‘`end`’

⟨*objectpatitem*⟩ ::= ([ ‘`$`’ ] ⟨*ident*⟩ | ‘`$(`’ ⟨*exp*⟩ ‘`)`’) [ ‘`:`’ ([ ‘`$`’ ] ⟨*ident*⟩ | ‘`$(`’ ⟨*exp*⟩ ‘`)`’) ]

⟨*objectpatbody*⟩ ::= ⟨*objectpatitem*⟩ { ‘`,`’ ⟨*objectpatitem*⟩ }

⟨*tablepatitem*⟩ ::= [ ([ ‘`$`’ ] ⟨*ident*⟩ | ‘`$(`’ ⟨*exp*⟩ ‘`)`’) ‘`:`’ ] ⟨*pattern*⟩

⟨*primarypattern*⟩ ::= ‘`(`’ ⟨*pattern*⟩ ‘`)`’
  | ⟨*strliteral*⟩
  | ⟨*numliteral*⟩
  | ‘`true`’
  | ‘`false`’
  | ‘`nil`’
  | ‘`_`’
  | [ ‘`$`’ ] ⟨*ident*⟩ [ ‘`(`’ [ ⟨*objectpatbody*⟩ ] ‘`)`’ ]
  | ‘`$(`’ ⟨*exp*⟩ ‘`)`’ [ ‘`(`’ [ ⟨*objectpatbody*⟩ ] ‘`)`’ ]
  | ‘`{`’ ⟨*tablepatitem*⟩ { ‘`,`’ ⟨*tablepatitem*⟩ } ‘`}`’

⟨*suffixedpattern*⟩ ::= ⟨*primarypattern*⟩
  | ⟨*suffixedpattern*⟩ ‘`when`’ ⟨*exp*⟩
  | ⟨*suffixedpattern*⟩ ‘`as`’ ⟨*exp*⟩

⟨*pattern*⟩ ::= ⟨*suffixedpattern*⟩ [ (‘`and`’ | ‘`or`’ | ‘`::`’) ⟨*suffixedpattern*⟩ ]

⟨*matcharm*⟩ ::= (‘`|`’ | ‘`case`’) ⟨*patternlist*⟩ ⟨*expscope*⟩

⟨*matcharmstat*⟩ ::= (‘`|`’ | ‘`case`’) ⟨*patternlist*⟩ ⟨*statscope*⟩

⟨*matchbody*⟩ ::= ⟨*matcharm*⟩ { ⟨*matcharm*⟩ }

⟨*matchbodystat*⟩ ::= ⟨*matcharmstat*⟩ { ⟨*matcharmstat*⟩ }

# 10 Appendix: Influences

Vortex is a language with many influences. It follows the same principles as Lua, on which is built. Thus, you can find many similarities in Lua's and Vortex's syntax and semantics as well as the core set of features. Vortex and Lua are both built around tables, the ultimate do-it-all data structure. The reliance on higher order and first class functions is also taken very seriously in both languages.
The second largest influence for Vortex is OCaml. Vortex takes many primarily syntactic features from OCaml, including the pattern matching syntax. Vortex is however, unlike OCaml, an untyped language. The features are thus adjusted accordingly for Vortex.
OCaml's relative, F#, is also influental for Vortex. The basic idea of sequences in place of

list comprehensions is taken from F#, but unlike F# sequences are not values in Vortex, instead they're simply tuples of values that can't be treated in a first class manner.
The prototype based multiple inheritance object system of Vortex is inspired by Io. Compared to Io, the constructors work differently and Vortex's object system allows you to pretty much closely imitate classes (those are handled a bit differently in Io).
Scheme inspired Vortex in how simple a language can be while remaining very powerful. Along with Elixir, Scheme provided a basis for Vortex AST macros.
Elixir and Ruby influenced Vortex's syntax. The Rust language provided an example how terse a syntax can be and several of Vortex's keywords are or were taken from Rust. The `cycle` keyword is taken from Fortran. String literal syntax is taken from Python.
Other languages that influenced Vortex to some degree are ALGOL, C, CLU, Haskell and Self.