Whip is a LISP-dialect made for scripting and simplified concepts. It has also borrowed a lot of functions and syntax from Haskell (a non-related language).

These docs were written by the creator of the language himself. So is this line.

```
; Comments are like LISP. Semi-colons...

; Majority of first-level statements are inside "forms"
; which are just things inside parens separated by whitespace
not_in_form
(in_form)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 1. Numbers, Strings, and Operators

; Whip has one number type (which is a 64-bit IEEE 754 double, from JavaScript).
3 ; => 3
1.5 ; => 1.5

; Functions are called if they are the first element in a form
(called_function args)

; Majority of operations are done with functions
; All the basic arithmetic is pretty straight forward
(+ 1 1) ; => 2
(- 2 1) ; => 1
(* 1 2) ; => 2
(/ 2 1) ; => 2
; even modulo
(% 9 4) ; => 1
; JavaScript-style uneven division.
(/ 5 2) ; => 2.5

; Nesting forms works as you expect.
(* 2 (+ 1 3)) ; => 8

; There's a boolean type.
true
false

; Strings are created with ".
"Hello, world"

; Single chars are created with '.
'a'

; Negation uses the 'not' function.
(not true) ; => false
(not false) ; => true

; But the majority of non-haskell functions have shortcuts
; not's shortcut is a '!'.
(! (! true)) ; => true

; Equality is `equal` or `=`.
```

```
(= 1 1) ; => true
(equal 2 1) ; => false

; For example, inequality would be combining the not and equal functions.
(! (= 2 1)) ; => true

; More comparisons
(< 1 10) ; => true
(> 1 10) ; => false
; and their word counterpart.
(lesser 1 10) ; => true
(greater 1 10) ; => false

; Strings can be concatenated with +.
(+ "Hello " "world!") ; => "Hello world!"

; You can use JavaScript's comparative abilities.
(< 'a' 'b') ; => true
; ...and type coercion
(= '5' 5)

; The `at` or @ function will access characters in strings, starting at 0.
(at 0 'a') ; => 'a'
(@ 3 "foobar") ; => 'b'

; There is also the `null` and `undefined` variables.
null ; used to indicate a deliberate non-value
undefined ; user to indicate a value that hasn't been set

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 2. Variables, Lists, and Dicts

; Variables are declared with the `def` or `let` functions.
; Variables that haven't been set will be `undefined`.
(def some_var 5)
; `def` will keep the variable in the global context.
; `let` will only have the variable inside its context, and has a weirder syntax.
(let ((a_var 5)) (+ a_var 5)) ; => 10
(+ a_var 5) ; = undefined + 5 => undefined

; Lists are arrays of values of any type.
; They basically are just forms without functions at the beginning.
(1 2 3) ; => [1, 2, 3] (JavaScript syntax)

; Dictionaries are Whip's equivalent to JavaScript 'objects' or Python 'dicts'
; or Ruby 'hashes': an unordered collection of key-value pairs.
{"key1" "value1" "key2" 2 3 3}

; Keys are just values, either identifier, number, or string.
(def my_dict {my_key "my_value" "my other key" 4})
; But in Whip, dictionaries get parsed like: value, whitespace, value;
; with more whitespace between each. So that means
{"key" "value"
"another key"
```

```
1234
}
; is evaluated to the same as
{"key" "value" "another key" 1234}

; Dictionary definitions can be accessed used the `at` function
; (like strings and lists.)
(@ "my other key" my_dict) ; => 4

;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 3. Logic and Control sequences

; The `if` function is pretty simple, though different than most imperative langs.
(if true "returned if first arg is true" "returned if first arg is false")
; => "returned if first arg is true"

; And for the sake of ternary operator legacy
; `?` is if's unused shortcut.
(? false true false) ; => false

; `both` is a logical 'and' statement, and `either` is a logical 'or'.
(both true true) ; => true
(both true false) ; => false
(either true false) ; => true
(either false false) ; => false
; And their shortcuts are
; & => both
; ^ => either
(& true true) ; => true
(^ false true) ; => true

;;;;;;;;;
; Lambdas

; Lambdas in Whip are declared with the `lambda` or `->` function.
; And functions are really just lambdas with names.
(def my_function (-> (x y) (+ (+ x y) 10)))
;         |       |    |          |
;         |       |    |    returned value(with scope containing argument vars)
;         |       | arguments
;         | lambda declaration function
;         |
;   name of the to-be-declared lambda

(my_function 10 10) ; = (+ (+ 10 10) 10) => 30

; Obviously, all lambdas by definition are anonymous and
; technically always used anonymously. Redundancy.
((lambda (x) x) 10) ; => 10

;;;;;;;;;;;;;;;
; Comprehensions

; `range` or `..` generates a list of numbers for
```

```
; each number between its two args.
(range 1 5) ; => (1 2 3 4 5)
(.. 0 2)    ; => (0 1 2)

; `map` applies its first arg (which should be a lambda/function)
; to each item in the following arg (which should be a list)
(map (-> (x) (+ x 1)) (1 2 3)) ; => (2 3 4)

; Reduce
(reduce + (.. 1 5))
; equivalent to
((+ (+ (+ 1 2) 3) 4) 5)

; Note: map and reduce don't have shortcuts

; `slice` or `\` is just like JavaScript's .slice()
; But do note, it takes the list as the first argument, not the last.
(slice (.. 1 5) 2) ; => (3 4 5)
(\ (.. 0 100) -5) ; => (96 97 98 99 100)

; `append` or `<<` is self explanatory
(append 4 (1 2 3)) ; => (1 2 3 4)
(<< "bar" ("foo")) ; => ("foo" "bar")

; Length is self explanatory.
(length (1 2 3)) ; => 3
(_ "foobar") ; => 6

;;;;;;;;;;;;;;;
; Haskell fluff

; First item in list
(head (1 2 3)) ; => 1
; List from second to last elements in list
(tail (1 2 3)) ; => (2 3)
; Last item in list
(last (1 2 3)) ; => 3
; Reverse of `tail`
(init (1 2 3)) ; => (1 2)
; List from first to specified elements in list
(take 1 (1 2 3 4)) ; (1 2)
; Reverse of `take`
(drop 1 (1 2 3 4)) ; (3 4)
; Lowest value in list
(min (1 2 3 4)) ; 1
; Highest value in list
(max (1 2 3 4)) ; 4
; If value is in list or object
(elem 1 (1 2 3)) ; true
(elem "foo" {"foo" "bar"}) ; true
(elem "bar" {"foo" "bar"}) ; false
; Reverse list order
(reverse (1 2 3 4)) ; => (4 3 2 1)
; If value is even or odd
```

```
(even 1) ; => false
(odd 1) ; => true
; Split string into list of strings by whitespace
(words "foobar nachos cheese") ; => ("foobar" "nachos" "cheese")
; Join list of strings together.
(unwords ("foo" "bar")) ; => "foobar"
(pred 21) ; => 20
(succ 20) ; => 21
```

For more info, check out the repo