Introduction

FH is a simple to use and learn programming language meant to be used as a scripting language.

FH is being written in the C programming language and it runs on: GNU/Linux, MacOS and Windows. Porting FH to other platforms should be quite straightforward given a C99 compiler exists on the targeted platform.

Basic concepts

```
This section describes the basic concepts of the language.
```

Values and types

FH is a dynamically typed language. This means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

All values in FH are first-class values. This means that all values can be stored in variables, passed as arguments to other functions, and returned as results.

There are eight basic types in FH: null, boolean, number, string, function, userdata, array and map.

- The type null has one single value, null, whose main property is to be different from any other value; it usually represents the absence of a useful value.
- The type boolean has two values, false and true. Both null and false make a condition false; any other value makes it true. The type number represents both integer numbers and real (floating-point) numbers.
- The type string represents immutable sequences of bytes. FH supports unicode and it makes no assumptions about the contents of a string.
- The type number represents 32-bit floating points numbers
- The type userdata is provided to allow arbitrary C data to be stored in FH variables. A userdata value represents a block of raw memory.
- The type array is of type associative array, that is, arrays that can have as indices not only numbers but any FH value, including null.
- The type map can be heterogeneous; that is, they can contain values of all types. All keys must have a type different than null. Trying to use a key which is not part of the map will result in an error.

Garbage collection

FH performs automatic memory management. This means that you do not have to worry about allocating memory for new objects or freeing it when the objects are no longer needed. FH manages memory automatically by running a garbage collector to collect all dead objects (that is, objects that are no longer accessible from FH). The algorithm used for performing this task is called "mark and sweep".

Since the algorithm used for garbage collecting is part of the tracing algorithms that means the program will stop from executing from a brief moment in which the GC is being executed.

The frequency at which the GC is called can be controlled using the function `gc_frequency()` which expects a number larger or equal to 1000000 (1MB). This means that every time FH allocates more than the given threshold the GC will be fired up.

In extreme cases the GC can be paused and resumed as the programmer pleases using the function `gc_pause` which expects a boolean to determine if the GC should be paused or resumed. The garbage collector can be forcefully triggered by the programmer by calling the function `gc()`.

If you want to know how much memory is currently allocated by the executing program then function `gc_info()` will return a number of bytes which indicates this.

The language TODO

This section describes the lexis, the syntax, and the semantics of Lua. In other words, this section describes which tokens are valid, how they can be combined, and what their combinations mean.

Language constructs will be explained using the usual extended BNF notation, in which {a} means 0 or more a's, and [a] means an optional a. Non-terminals are shown like non-terminal, keywords are shown like kword, and other terminal symbols are shown like ‘=’. The complete syntax of Lua can be found in §9 at the end of this manual.

Lexical Conventions

Lua is a free-form language. It ignores spaces (including new lines) and comments between lexical elements (tokens), except as delimiters between names and keywords.

Names (also called identifiers) in Lua can be any string of letters, digits, and underscores, not beginning with a digit and not being a reserved word. Identifiers are used to name variables, table fields, and labels.

The following keywords are reserved and cannot be used as names:

```
and       break     do        else      elseif    end
false     for       function  goto      if        in
local     nil       not       or        repeat    return
then      true      until     while
```

Lua is a case-sensitive language: and is a reserved word, but And and AND are two different, valid names. As a convention, programs should avoid creating names that start with an underscore followed by one or more uppercase letters (such as _VERSION).

The following strings denote other tokens:

```
+     -     *     /     %     ^     #
```

```
&     ~     |     <<     >>     //
==    ~=    <=    >=     <      >      =
(     )     {     }      [      ]      ::
;     :     ,     .      ..     ...
```

A short literal string can be delimited by matching single or double quotes, and can contain the following C-like escape sequences: '\a' (bell), '\b' (backspace), '\f' (form feed), '\n' (newline), '\r' (carriage return), '\t' (horizontal tab), '\v' (vertical tab), '\' (backslash), '\"' (quotation mark [double quote]), and '\'' (apostrophe [single quote]). A backslash followed by a line break results in a newline in the string. The escape sequence '\z' skips the following span of white-space characters, including line breaks; it is particularly useful to break and indent a long literal string into multiple lines without adding the newlines and spaces into the string contents. A short literal string cannot contain unescaped line breaks nor escapes not forming a valid escape sequence.

We can specify any byte in a short literal string by its numeric value (including embedded zeros). This can be done with the escape sequence \xXX, where XX is a sequence of exactly two hexadecimal digits, or with the escape sequence \ddd, where ddd is a sequence of up to three decimal digits. (Note that if a decimal escape sequence is to be followed by a digit, it must be expressed using exactly three digits.)

The UTF-8 encoding of a Unicode character can be inserted in a literal string with the escape sequence \u{XXX} (note the mandatory enclosing brackets), where XXX is a sequence of one or more hexadecimal digits representing the character code point.

Literal strings can also be defined using a long format enclosed by long brackets. We define an opening long bracket of level n as an opening square bracket followed by n equal signs followed by another opening square bracket. So, an opening long bracket of level 0 is written as [[, an opening long bracket of level 1 is written as [=[, and so on. A closing long bracket is defined similarly; for instance, a closing long bracket of level 4 is written as]====]. A long literal starts with an opening long bracket of any level and ends at the first closing long bracket of the same level. It can contain any text except a closing bracket of the same level. Literals in this bracketed form can run for several lines, do not interpret any escape sequences, and ignore long brackets of any other level. Any kind of end-of-line sequence (carriage return, newline, carriage return followed by newline, or newline followed by carriage return) is converted to a simple newline.

For convenience, when the opening long bracket is immediately followed by a newline, the newline is not included in the string. As an example, in a system using ASCII (in which 'a' is coded as 97, newline is coded as 10, and '1' is coded as 49), the five literal strings below denote the same string:

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

Any byte in a literal string not explicitly affected by the previous rules represents itself. However, Lua opens files for parsing in text mode, and the system file functions may have problems with some control characters. So, it is safer to represent non-text data as a quoted literal with explicit escape sequences for the non-text characters.

A numeric constant (or numeral) can be written with an optional fractional part and an optional decimal exponent, marked by a letter 'e' or 'E'. Lua also accepts hexadecimal constants, which start with 0x or 0X. Hexadecimal constants also accept an optional fractional part plus an optional binary exponent, marked by a letter 'p' or 'P'. A numeric constant with a radix point or an exponent denotes a float; otherwise, if its value fits in an integer, it denotes an integer. Examples of valid integer constants are

```
3   345   0xff   0xBEBADA
```

Examples of valid float constants are

```
3.0      3.1416     314.16e-2     0.31416E1     34e1
0x0.1E   0xA23p-4   0X1.921FB54442D18P+1
```

A comment starts with a double hyphen (--) anywhere outside a string. If the text immediately after -- is not an opening long bracket, the comment is a short comment, which runs until the end of the line. Otherwise, it is a long comment, which runs until the corresponding closing long bracket. Long comments are frequently used to disable code temporarily.

Variables

Variables are places that store values. There are three kinds of variables in Lua: global variables, local variables, and table fields.

A single name can denote a global variable or a local variable (or a function's formal parameter, which is a particular kind of local variable):

```
var ::= Name
```

Name denotes identifiers, as defined in §3.1.

Any variable name is assumed to be global unless explicitly declared as a local (see §3.3.7). Local variables are lexically scoped: local variables can be freely accessed by functions defined inside their scope (see §3.5).

Before the first assignment to a variable, its value is nil.

Square brackets are used to index a table:

```
var ::= prefixexp ‘[’ exp ‘]’
```

The meaning of accesses to table fields can be changed via metatables (see §2.4).

The syntax var.Name is just syntactic sugar for var["Name"]:

```
var ::= prefixexp ‘.’ Name
```

An access to a global variable x is equivalent to _ENV.x. Due to the way that chunks are compiled, _ENV is never a global name (see §2.2).

Statements

Lua supports an almost conventional set of statements, similar to those in Pascal or C. This set includes assignments, control structures, function calls, and variable declarations.

Blocks

A block is a list of statements, which are executed sequentially:

```
block ::= {stat}
```

Lua has empty statements that allow you to separate statements with semicolons, start a block with a semicolon or write two semicolons in sequence:

```
stat ::= ‘;’
```

Function calls and assignments can start with an open parenthesis. This possibility leads to an ambiguity in Lua's grammar. Consider the following fragment:

```
 a = b + c
 (print or io.write)('done')
```

The grammar could see it in two ways:

```
 a = b + c(print or io.write)('done')

 a = b + c; (print or io.write)('done')
```

The current parser always sees such constructions in the first way, interpreting the open parenthesis as the start of the arguments to a call. To avoid this ambiguity, it is a good practice to always precede with a semicolon statements that start with a parenthesis:

```
 ;(print or io.write)('done')
```

A block can be explicitly delimited to produce a single statement:

```
stat ::= do block end
```

Explicit blocks are useful to control the scope of variable declarations. Explicit blocks are also sometimes used to add a return statement in the middle of another block (see §3.3.4).