CONCISE

A programming language
© Jean-Jacques François Reibel 2020, Version November 1, 2020

INTRODUCTION

Concise is a language designed to be as fast as possible while retaining programmer-friendly syntax and language behavior. It draws inspiration from various sources while intending to correct various frustrating design features of many languages from a programmer's perspective. The language is also meant to be hackable and modifiable, with the intention of being able to support changes to syntax and allowing for code from any supported language to be run within code snippets. The base language is designed to be as it is named, concise in syntax and operation, while allowing users to customize it into their very own "Concise++".

COPYRIGHT AND TERMS OF USE

Concise provides no warranty of any kind.
Concise may be used freely with the following conditions.
Concise may be used freely as a tool to create software products, BUT selling copies of the Concise source code as a product is not allowed.
Concise is designed to be open source, BUT, the user(s) must attribute the original author, Jean-Jacques François Reibel, in every work where Concise is used, as well as supply the same message of Copyright.
Users may create works using Concise for commercial use, BUT the language itself as well as any compilers and/or interpreters must be open source and have the attached Copyright license.
Users may fork Concise IF they supply the same message of Copyright and attribute the original author, Jean-Jacques François Reibel.
Users of Concise may not change or modify the official Concise language or the Concise Standard Library without permission of the original author, Jean-Jacques François Reibel, or whomsoever else he gives such authority to. The original author, Jean-Jacques François Reibel, reserves the right to give and/or remove any permissions to any developer, group of people, or organization to continue or cease work on Concise to grow the language, its Standard Library, and/or the community.
Modification of the original Concise language, IF attributing the original author, Jean-Jacques François Reibel, AND attaching the Copyright license, **IS ALLOWED UNDER THE CIRCUMSTANCES** that the person(s) develop their own Concise++ language under the same license, but make no modification the the OFFICIAL, open source Concise that is meant to be available to the public. Thus, these Concise++ versions are unofficial or forked versions of Concise, which is allowed, and will not interfere with users gaining access to the OFFICIAL version of Concise.

HISTORY

The language has been designed and redesigned multiple times over many years, with efforts to remain as true to its roots of an extremely fast and highly complex programming language that is “sweet” to a programmer using the code. To accomplish this, a very simple syntax was devised, with the ability to add/modify the syntax and run code from other languages. This allowed programmers who were comfortable with a more limited syntax to write code with the syntactical simplicity of Python or Ruby, but the potential for increasing/decreasing efficiency, speed, concurrency, safety, and low level control and having various features as languages like C/C++, Rust, Java, Erlang, Scala, and others. The goal was also to have a language that allowed for both functional and object oriented programming. As such, the extended goal of Concise allows programmers to essentially write their own language or rewrite another language. A Concise programmer could essentially write a specification .conc file that allows all code to be in the syntax of Clojure. The programmer could also extend these specifications to run Clojure code. Concise also aims to allow transpiling to all supported languages, so that a programmer could write Concise code, and have it be translated immediately to Clojure, or any language, new or existing, of their choosing. Furthermore, Concise allows a programmer or organization to implement code to allow for greater functionality of the foreign language with Concise, either by writing more efficient code, or to add features, such as having a Concise version that supports unsigned integers as a variable type without calling C code, or supporting a new Clojure API for multithreaded processing. Concise is designed to be a great general programming and software engineering language, but with variable complexity for programmers of both advanced and simple backgrounds, or advanced and simple tasks. Concise can be used for Front-End Web development to write markup and style, in place of HTML or CSS, or scripting in place of JavaScript. Concise could also be a substitute for Elm or Markaby. Concise is an excellent choice for Data Scientists and Engineers given its speed and simple out of the box syntax, as well as being used for game programmers writing scripts or shaders.

COMMENTS

By convention, comments should be above code or after the code on the same line
Single Line: #
Multi-Line: #/…/#

VARIABLES

Automatically converts whole numbers to int -> long
Automatically converts decimal numbers to float -> double -> long double
Automatically converts single character strings ‘’ or “” to char
Automatically converts character strings of length greater than one to string
Automatically converts character strings of length one to char
Automatically converts variables with no value as null
Booleans are true or false and can be null
Example: x = true
In Concise, ARRAYS ARE AUTOMATICALLY DYNAMIC
In Concise, MULTIDIMENSIONAL ARRAYS BEHAVE LIKE MATRICES
Arrays declared with values as

```
myArr1D = (1, 4, 75, 62.5, ‘a’, “foobar”, b)
myArr2D = (1, 4, 75, 62.5, ‘a’, “foobar”, b), (2, 3, 54, 23.54, ‘u’, “barfoo”, d)
…
```

Arrays declared with size as

```
myArr1D = [17]
myArr2D = [43][3]
…
```

Array assignment

```
myArr(6)(7) = 42
```

Maps declared with values as

```
myMap = (“Legolas”:”Greek”, “Xerxes”:”Persian”, x:4)
```

Map assignment

```
myMap(“Legolas”) = “Roman”
```

Queue declared with values as

```
myQueue1D = (1, 4, 75, 62.5, ‘a’, “foobar”, b)
myQueue2D = (1, 4, 75, 62.5, ‘a’, “foobar”, b), (2, 3, 54, 23.54, ‘u’, “barfoo”, d)
…
```

Queue declared with size as

```
my Queue1D = [17]
my Queue2D = [43][3]
…
```

CONSOLE INPUT/OUTPUT

Console Input: u = input
Output: output(“This is the output”)
Output two strings: output(“This is the output ”, “this is the other output”)
Output two lines: output(“This is the first line”, #@”this is the second line”)
Output variable: output(x)

OTHER BASIC SYNTAX

All statements must end with one of the following:
hashtag
single line comment
square bracket
parenthesis
single or double quotes
number
char
string
variable
value such as null, true, or false

IMPORT

Import Statement for single file: #< myClass.conc

Import package: #< packageName

Import package library: #< packageName.lib

Run foreign code snippet: ##(“Language”)…##

Return value from foreign code syntax example:

```
# No arguments
myVar = (
        ##(“Java”)
            //some code
            return x;
            //some closing code
        ##
     )
# With arguments
#/
file is created that has accessible values in a snippet of code in desired language
/#
myVar(4, 8, 7) = (
        ##(“Swift”)
            //some code
            return x;
            //some closing code
        ##
     )
```

CLASSES

Classes are C structs with function pointers where possible

When not possible, classes are structs with functions

When that is not possible either, other implementation strategies are used

Note: Concise is meant to be customizable :)

Syntax example for defining a class with member variables and functions:

```
# Arguments can be blank
Car(wheelsIn, doorsIn, cylindersIn, gearsIn) = (
    # Variables
    wheels = wheelsIn
    doors = doorsIn
    cylinders = cylindersIn
    gears = [gearsIn]
    # Functions. The () for addWheels is implicit
    addWheels = (wheels = wheels + 1)
    addWheels(x) = (wheels = wheels + x)
    getWheels = (return wheels)
)
```

Syntax example for declaring an instance of a class

```
# Variable assignment
subaru = Car(4, 4, 4, 6)
```

```
# No variable assignment
subaru = Car()
```

Syntax example for modifying values and calling functions

```
#/
The () for addWheels is implicit.
Variable and function names should not be the same
/#
subaru.wheels = 6
subaru.addWheels(5)
subaru.addWheels()
subaru.addWheels
```

COMPARISONS

Equal ==

Equal and Same Type ===

Not Equal !=

Not Equal and Same Type !==

Greater or Equal >= and <=

Greater or Equal and Same Type >== and <==

Greater or Equal but Not Same Type >=!= and <=!=

Is of the same type, regardless of value <>

ADVANCED FUNCTIONALITY AND ARRAY FUNCTIONS

Make a deep copy. In Concise, DEEP COPIES ARE THE DEFAULT

```
# Make a deep copy
x = z
# Make a shallow copy
y = z.shallow()
# Pointer to a variable
*a
# Address of a variable
&a
```

Given any 1D array, add value to next index, or remove last index value
In Concise, ARRAYS BEHAVE LIKE STACKS

```
myArr.push(4)
myArr.pop()
#/
removes the element at index 6, everything else moves down
/#
myArr.pop(6)
#/
removes the element at index (4,2,5), everything else moves down
regardless of whether using an array or queue, positions move down
/#
myArr.pop(4,2,5)
```

Given any Multidimensional array

```
    # Make push and pop functions affect last column
    myArr.setVertical()
    # Make push and pop functions affect last row
    myArr.setHorizontal()
    # Make push and pop functions affect last row and last column
    myArr.setNeutral()
    #/
```

Functions

```
    myArr.len()
    myMap.len()
    myArr.indexOf(4) #returns first index of 4
    myArr.indexOf(4, 2) #returns second index of 4
    myArr.count(4) #returns number of occurrences of 4
```

Queues share all the same functions as arrays, but pop functions remove the first value

In Concise, QUEUES ARE DYNAMIC BY DEFAULT

In Concise, MULTIDIMENSIONAL QUEUES BEHAVE LIKE MATRICES

LOOPS AND CONTROL

#last values are included

For Loops

```
    for(i=0, i<10, ++i)()
    for(i..10)
```

For In Loop

```
    for(i in myArr)()
```

For In Map Loop

```
    for((index, value) in myArr) ()
```

For Range Loop

```
    for(i in range)
```

While Loop

```
    while(x<10)
```

Do While Loop

```
    do(++i)while(x<9)
```

When Loop

```
    when(x == 10)
```

Switch

```
    switch(x)
    case(1, x==“Hello”)(
        output(“World”)
    )
    case(2, x==“Bye”)(
        output(“Cruel World”)
    )
    default(
        output(“No greeting”)
    )
Try/Catch
    try(x = y)
    catch(
        output(“No variable called ‘y’”)
    )
Throw
    if(x>7)(
        throw(NumberTooBigException)
    )
    if(x==7)(
        throw(NumberEqualException)
    )
    catch(NumberTooBigException))
        output(“That’s too big!”)
    )
    catch(NumberEqualException))
        output(“That’s the same number!”))
```

FILE INPUT/OUTPUT AND STRING FUNCTIONS

```
file1 = finput(‘C:\MyDocs\My_Folder\file.extension’) #create file object
str1 = file1.read() #store entire file text into string
file1.append(“HELLO”) #append text
file1.write(“WORLD?”) #replace text
file1.erase(-1) #characters to erase, including white space
file1.delete() #delete file
file1.copy() #copy file to same directory with name ‘file1(n)’
file1.copy(‘file2’) #copy file to same directory with name ‘file2’, with same extension
file1.copy(‘file2’, ‘.txt’) #copy file to same directory with name ‘file2.txt’
str1.find(‘hello’) # return index of first instance of ‘hello'
str1.find(‘hello’, 2) # return index of second instance of ‘hello’
str1.count(‘hello’) # return number instances of ‘hello’
str1.charCount() #returns number of chars
str1.delimCount(’t’) #returns number of strings delimited by ’t’
#makes a blank file in the same directory as file1 named ‘newFile.extension’
file1.make(‘newFile.extension’)
file1.changeExtension(‘.txt.) # changes file1’s extension to ‘.txt’
```

ARITHMETIC

```
++x, x++ are the same
—x, x— are the same
x+=5 is x = x + 5
x-=5 is x = x - 5
x*=5 is x = x * 5
x/=5 is x = x / 5
```

x%=5 is x = x % 5
x^=5 is x = x ^ 5

NETWORKING AND MULTITHREADING

To be fully implemented as a separate library in the Concise Standard Library
This should be simple, and allow a programmer to easily create a simple server and use it very simply.
Examples:

```
	server1 = server(host, port, handler, scripts[])
	add = url(address)
```

ACCESS MODIFIERS

Global: Anywhere in the program, including other packages
Public: Anywhere in package, default
Protected: Only the class and the inherited classes
Private: Only the class

TYPE FORCING AND CASTING

```
int x = 5 # this will keep x as an int
x = int(x) # this will force x to be an int
```

The types and typecast functions standard in Concise are:

```
	int, long, float, double, longdouble, char, string
```

Null values
x = null

TYPE BOOLEAN FUNCTIONS

isInt(x), isLong(x), isFloat(x), isDouble(x), isLongDouble(x), isChar(x), isString(x)
isNumber(x), isDecimal(x), isType(“Class”, x)

STANDARD LIBRARY

Libraries designed to have more advanced features or features not included by default
To standardize Concise imports, they should be minimal, or there should be compound libraries that have all relevant minor libraries.
For example, Python has SimpleHTTPServer and SocketServer. To simplify imports, Concise should just have a Networking library as a standard for everything networking. This library should be updated instead of having too many similar libraries, so that programmers can easily get all the imports they need and have it ready to go.
Another example is STEM libraries for scientists and engineers. No need to have a chemical engineering library and an electrician’s library and a data scientists library. This can occur in development between Concise versions, if the version is personal or for an organization. But, standard Concise should have compound libraries. However, there are certain libraries that should remain separate. For example, Networking should be its own library, while STEM another library, because obviously STEM should be more for Science, Engineering, Mathematics, and how these are applied to Technology, such as solar panels. But

Networking would be standard in Concise, without the need for import, but allowing the import makes Concise a safer language. Further still, a SolarPanel library would be a great idea for solar panel technicians, engineers, and scientists, where they would expect to find all simple electrical and magnetism libraries within STEM. This does make sense, so that the chemical engineer does not get confused and use a solar API by mistake. However, a compound STEM API with all non-proprietary functions and equations is still in the style of Concise. Therefore, a second API called STEMLite can be created. The solar engineer will then have to import either STEM or both STEMLite and SolarPanel. STEMLite and SolarPanel can be libraries within the STEM package. The STEM library is meant to share the same license as the standard Concise language, and so will omit any proprietary libraries. However, Concise promotes a complete API, backed by programmers and others willing to contribute. In doing so, it promotes and supports the creation of Artificial Intelligence systems that can grow and learn about anything. That is part of the spirit of the Concise Standard Library. Therefore, until new libraries are developed, these are the Libraries to be created as standard:

```
    Networking # Anything network or cloud related
    Threading # Anything thread or stream related not requiring Systems
    Systems # Anything systems related: OS threads, pools, GPU, CPU, other
    STEM #Science, Technology, Engineering, Mathematics, Full loaded
    STEMLite #Science, Technology, Engineering, Mathematics, Lite and basic
    #/ For simple GUIs as well as 2D and 3D,
    not related to Systems or Threading but they can work together /#
    Graphics
    #includes things like hashes, advanced sorting algorithms and tree traversals
    Cryptography
```

Notice how console IO and file IO are standard and do not need to be imported.

HACKING CONCISE 101

Concise is meant to be customizable to the point where a programmer can define their own variation, dialect, or languages, or replicate an existing language.

As an example, let’s say a programmer cannot stand the fact that I made the hashtag ‘#’ the symbol for making single line comments. The programmer could, preferably in a separate file, change the single line comment character to anything they want, such as the dollar symbol ‘$’, or the word ‘comment’, or even (worse), ‘single-line-comment’. This can be done, but it will be the programmer’s responsibility to ensure compatibility and non-conflicting statements. For example, ‘#@‘ is reserved for skipping a line when using output. Therefore, if the programmer were to change the multiline comment symbol to ‘#@‘, then every time there would be an error when trying to skip a line in output because the code after that symbol would be commented out.

Change the single line comment variable from ‘#’ to ‘$’

```
    ##[‘#’] = ‘$’ ##
```

Switch brackets and parenthesis

```
    ##[‘[’] = ‘(’ ##
    ##[‘]’] = ‘)’ ##
    ##[‘(’] = ‘[’ ##
    ##[‘)’] = ‘]’ ##
```

Classes, a bit more complicated, uses indices

Redefine how classes are defined.

#>(n)…>#(n) represents indexed code block

Redefine how classes are defined.

Any non blank space is interpreted as a required keyword.

As many indices can be used as desired.

Turn "Class() = ()” into “class Class(){}”

```
    ##[#>(0) Class() >#(0) #>(1) = () >#(1)] =
        “#>(0) class Class() >#(0) #>(1) {} >#(1)” ##
```

Longer, easier to read form

```
    ##[
        #>(0) Class() >#(0)
         #>(1) = () >#(1)
    ] =
        “#>(0) class Class() >#(0)
        #>(1) {} >#(1)”
    ##
```

Redefine how class instances are declared.

Turn “class = Class()” into “class = new Class()”

```
    ##[#>(0) class = >#(0) #>(1) Class() >#(1)] = “class = new Class()” ##
```

Longer, easier to read form

```
    ##[
        #>(0) class = >#(0)
        #>(1) Class() >#(1)
    ] =
        “#>(0) class = >#(0)
        new Class()” ##
        #>(1) new Class() >#(1)”
    ##
```

FINAL WORDS

This document is meant to be an introduction to Concise, and as of the time of this writing, the various versions of its compilers and interpreters are in active development. Concise is meant to be flexible yet high performing. In an effort to make itself available to the most programmers possible, compilers/interpreters that have been given priority are, in loose order of priority:

C, meant to be compatible with all versions of C and C++

Python, meant to be compatible with all versions of Python

Java, meant to be compatible with all versions of Java 8 and higher

Kotlin and Scala, all versions, as well as the other JVM languages

ECMAScript 6

JavaScript and TypeScript, compatible with all versions

Rust

Go

Erlang, Elixir, and the EVM
Ruby
… the list goes on

Finally programming languages, like all languages, are both abstract and logical, like art and science, this is where their beauty and the beauty of communication and evolution of information processing and technology comes from. It makes sense for mathematically oriented languages like Clojure and Lisp to have a modified yin yang symbol.