GREEN TEA PROGRAMMING LANGUAGE MANUAL

https://gtlang.com

Contents

Introduction

Green Tea is aimed for people who are new to programming, just like Sketch. It is a simple language that purpose is removing all the things you should remember and let you deploy your idea in Zen mode. "Your ideas matter, not the syntax".

+ Simple and easy to learn, even for people haven't know anything about programming

+ Easy to switch from other languages

+ Could be used to syntactically convert a source code of a language to another.

+ Use your native language to code. Why do you have to learn English fore code?

Green Tea is inspired by PHP, and is a script/compiled language, functional/object-oriented language

Some parts of this document is copied from the PHP manual.

Prerequisites

Green Tea is made for newcomers. No prerequisites here.

Before reading:

In this document, example code are blue and on-screen in/output are orange.

Code:

```
Hello World
```

Input:

```
< John
```

Output:

```
> 10
```

Some features has not been implemented yet, they will have orange background.

Installation

Windows

You just need to download the binary, put it somewhere, then add it parent folder to *PATH* environment variable.

Linux

Generally, you just need to download the binary, put it somewhere, *chmox +x* it, then add it parent folder to *PATH* environment variable.

Debian

You may need to install `additional libraries: libcurl libicu libmysqlclient`

`Ex: sudo apt install libcurl4 libicu74 libmysqlclient21`

`(Ubuntu 24.04 LTS)`

RedHat

You may need to install `additional libraries: libcurl4 libicu libboost libmysqlclient`

MacOS

You need to download the binary, put it somewhere, *chmox +x* it, then add it parent folder to *PATH* environment variable.

Usage

Interprept

`gtlang <sourcefile>`

Ex:

```
gtlang hello_world.gtc
```

Compile:

`gtcompiler <sourcefile><output file>`

Ex:

```
gtcompiler hello_world.gtc hello_world
```

Language file

Green Tea is the multi-natural-languages programming language

Usage

To use another language, use *#language_file* keyword. It must be on the 2nd line of the script.

The variables' name, functions' name, classes' name, keywords could be translated, but hard-coded strings could not.

Ex:

In ru.gtl, we defined:

```
if=если
for=для
echo=эхо
number=номер
```

in the source file, use

```
#!/bin/gtlang
#language_file ru.gtl
```

now you could code:

```
если $номер=0
@эхо “\$номерноль”
```

Creating new translation

To create new translation, please download the file

language.template.gtl

at github repo https://github.com/taateam/gtlang

Then add translation for each line, after the "=" sign.

Then add

```
#language_file <path_your_language_file.gtl>
```

to the 2nd line on your source file.

Syntax

Let start with a Hello world program:

Green Tea syntax is inspired by PHP, Shell Script, Javascript, Python, C++, Java, in decreasing order.

```
@echo Hello World
> Hello World
```

Comments

Comments in code will not be executed

Start with // to the end of line, or between /* and */

```
$a : 3 // create var a and assign 3
```

or

```
/* this is a comment
create var a and assign 3
end of comment */
$a  : 3
```

You can use html tags in comments.

```
/* <html>this is <b>a comment</b><br />
create var <b>a</b> and assign 3<br />
end of comment </html>*/
$a  := 3
```

Keywords

Here is the lists of keywords in Green Tea. You will know them later.

```
if, else, class, do , while, break, use, switch , new,
continue, return, include , throw, try, catch , finally,
defcat, deffin, elif , new, do, case, continue , for, times,
from, to , foreach, as, at, borrow , from
```

If you need any keyword as a string, please quote or escape it.

```
@echo if
> Syntax error
@echo "if"
> if
@echo \if
> if
```

Commands

A command is an instruction to do something. Usually a command is in a line

```
@echo hello
```

If you need multiple commands in one lines, separate them by semicolons (;)

```
@echo hello; @echo ""; @echo world
```

If you need a command use multiple lines, use ... at the end of not-last lines

```
@echo 1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1 ...
1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1...
1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1...
1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1...
1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1+1-1
> 0
```

Blocks

Commands of a block will have same indentation levels, you could have multiple block levels in a source file. Just like Python.

To understand better about block, please read bellow sections about if structures and loop structures and functions.

Indentation level depends on number of spaces and tabs at the beginning of the line contain the command.

Indentation level = round(number of spaces/4) + number of tabs.

Blocks of commands are multiple commands that is expected to be executed together, executed in order top to bottom, left to right.

Inclusion

You could use *include* keyword to add another file to current execution when program run.

Ex:

File a.gtc

```
@sum $a, $b
     return $a + $b
@echo "this is file a\n"
```

File b.gtc

```
@echo "begin file b\n"
@include "a.gtc"
@echo "this is file b\n"
@echo @sum 3, 4
```

This is equivalent to:

```
@echo "begin file b\n"
@sum $a, $b
     return $a + $b
@echo "this is file a\n"
@echo "this is file b\n"
@echo @sum 3, 4
> begin file b
> this is file a
> this is file b
> 7
```

About the function

```
@sum $a, $b
return $a + $b
```

please see function section.

Variables

Variables (vars) are places in memory when we store a value.

Variables in Green Tea are represented by a dollar sign followed by the name of the variable. The variable name is case-sensitive.

Variable names follow the same rules as other labels in GreenTea. A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: ^[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]$*

Ex: `$var_name`

Assignment:

Put a value into a var

```
$var:<value>
```

Ex:

```
$name:John
$age:24
```

Or assigning multiple vars at once:

```
$name,$age: John, 24
```

Get value:

To get a variable's value, use it's name:

```
$name:John
@echo $name
> John
```

Scope

A variable defined in a function only usable in that function.

If you want to access global variables `inside` function, use `$$` instead of `$`

```
$a=0
@function
$b: 1
$$c: 2
@echo $a // error
@echo $$a // output: 0
@echo $b // error
@echo $c // output: 2
```

Function

Functions are tasks, type it names, provide parameters and it'll do it task.

Functions in Green Tea are represented by a at sign followed by the name of the function. The function name is case-sensitive.

Function names follow the same rules as other labels in GreenTea. A valid function name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: ^[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]$*

The very first function you should know is @echo. It print things to the screen.

```
@echo "Hello world!!!"
> Hello world!!!
```

Declare

```
@function_name:
     <block>// the things this function do
```

Ex:

```
@my_function:
     @echo 123
     @echo 456
```

Then you call it

```
@my_function
> 123456
```

Adding some parameters, separated by commas:

```
@say_hi_to  $param_1, $param_2:
     @echo Hello $param_1 I am $param_2
@say_hi_to Annie John
```

Or:

```
Annie @say_hi_to Join
> Hello Annie I am John
```

Functions could return value, when they do, outer code could get the return value

```
@sum $a, $b:
     return $a + $b
$sum_total : (@sum 1, 2) + (@sum 3, 4)
> 10
```

Functions could return multiple values, like Python

```
@sum $a, $b:
     return $a + $b, $a - $b
$sum, $sub : (@sum 4, 3)
```

```
@echo "$sum and $sub"
> 7 and 1
```

Note: when using more than 1 function on a line (function and operator, 2 functions,...), you must wrap function call with ( )

When you do return in your function or have only return with no parameter, the function return `false.`

Self function

`@@`

Self function are function that call itself , return the value and assign its parameter to the returned value.

```
$a:1
@add5:
    return $a + 5
$b: (@@add5 $a)
@echo $a;
@echo $b
> 5
> 5
```

Regex-name function.

Let write a function that add 1 to a number, then write it out.

```
@add1 $number
     @echo ($number + 1)
```

Now, we need function that add 2 to a number, we will need to write new function which nearly exact content, which is boring. We provide Regular expression function to solve this.

@<function name's static part>{< function name's regular expression part>} <parameters>

<actions>

You could use `$_func_name` to access the value at the regular expression on function name.

Ex:

Let's rewrite the `@addX` function.

```
@add{[0-9]*} $num
     @echo( $_func_name + $num)
```

```
@add1   2
> 3
@add3   4
> 7
```
In case that a function call match multi Regex-name functions, only the firstly declared Regex-name function matched will run.

Expressions

Expressions includes vars and anything else with value

Ex:

```
1 + 2
"hello"
@sqr 2
```
To get the result value from expression, you could use assignment:

```
$value: 1+2+3/3
@echo "value:" $value
> value: 4
```
You could also get the value of expression in previous command using `$?`

```
1+2+3/3
@echo "value: " $?
> value: 4
```

Operators

Operators are operations on value(s), including:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Increment/Decrement operators
- Logical operators
- String operators
- Array operators
- Conditional assignment operators

Arithmetic operators

Operator	Name	Example	Result
+	Addition	$x+$y	Sum of $x and $y
-	Subtraction	$x - $y	Difference of $x and $y

*	Multiplication	$x * $y	Product of $x and $y
/	Division	$x / $y	Quotient of $x and $y
%	Modulus	$x % $y	Remainder of $x divided by $y
**	Exponentiation	$x ** $y	Result of raising $x to the $y'th power

Assignment operators

The basic assignment operator in GreenTea is ":". It means that the left operand gets set to the value of the assignment expression on the right.

Assignment	Same as...	Description	Example
$x: y		The left operand gets set to the value of the expression on the right	$age: 1
$x +: y	x : x + y	Addition	$year +: 2
$x -: y	x : x - y	Subtraction	$year -: 2
$x *: y	x : x * y	Multiplication	$money *: 2
$x /: y	x : x / y	Division	$number /: 3
x %: y	x : x % y	Modulus	$count %: 5

Assignments operator return assigned value

```
$a : 4
$?
> 4
```

Comparison operators

Operator	Name	Example	Result
=	Equal	$x = $y	Returns true if $x is equal to $y
==	Identical	$x ==$y	Returns true if $x is equal to $y, and they are of the same type
!=	Not equal	$x != $y	Returns true if $x is not equal to $y
!==	Not identical	$x !== $y	Returns true if $x is not equal to $y, or they are not of the same type
>	Greater than	$x > $y	Returns true if $x is greater than $y
<	Less than	$x < $y	Returns true if $x is less than $y
>=	Greater than or equal to	$x >= $y	Returns true if $x is greater than or equal to $y
<=	Less than or equal to	$x <= $y	Returns true if $x is less than or equal to $y

Increment/Decrement operators

Operator	Name	Descriptiont
++$x	Pre-increment	Increments $x by one, then returns $x
$x++	Post-increment	Returns $x, then increments $x by one
--$x	Pre-decrement	Decrements $x by one, then returns $x
$x--	Post-decrement	Returns $x, then decrements $x by one

Logical operators

Operator	Name	Example	Result
and	And	$x and $y	True if both $x and $y are true
or	Or	$x or $y	True if either $x or $y is true
xor	Xor	$x xor $y	True if either $x or $y is true, but not both
&&	And	$x && $y	True if both $x and $y are true
||	Or	$x || $y	True if either $x or $y is true
!	Not	!$x	True if $x is not true

String operators

Operator	Name	Example	Result
.	Concatenation	$txt1 + $txt2	Concatenation of $txt1 and $txt2
.:	Concatenation assignment	$txt1 +: $txt2	Appends $txt2 to $txt1
/	split	$string / $delimeter	Return an array, which is result of splitting $string with $delimeter

You could concatenate 2 string by putting them near by:

```
$a $b == $a.$b
```

Array operators

Operator	Name	Example	Result
+	Union	$x + $y	Union of $x and $y
=	Equality	$x == $y	Returns true if $x and $y have the same key/value pairs
==	Identity	$x === $y	Returns true if $x and $y have the same key/value pairs in the same order and of the same types
!=	Inequality	$x != $y	Returns true if $x is not equal to $y
!==	Non-identity	$x !== $y	Returns true if $x is not identical to $y

Conditional assignment operators

??	Ternary	$x : *expr1* ? *expr2* ? *expr3*	Returns the value of $x. The value of $x is *expr2* if *expr1* = TRUE. The value of $x is *expr3* if *expr1* = FALSE

Precedence and brackets

Expression are calculated by Precedence order. Which operator have higher precedence will be calculated earlier. For example:

1 + 2 * 5 = 1 + 10 = 11

In order to let the lower precedence to be calculated first, we could put the lower-rank operators and its objects round brackets

(1 +2) * 5 = 3 * 5 = 15

The following table lists the operators in order of precedence, with the highest-precedence ones at the top. Operators on the same line have equal precedence, in which case associativity decides grouping.

Associativity	**Operators**	**Additional Information**
(n/a)	clone new	clone and new
right	**	arithmetic
(n/a)	+ - ++ -- ~ (int) (float) (string) (array) (object) (bool) @	arithmetic (unary + and -), increment/decrement, bitwise, type casting and error control
left	instanceof	type
(n/a)	!	logical
left	* / %	arithmetic
left	+ - .	arithmetic (binary + and -), array and string
left	<<>>	bitwise
left	+	string
non-associative	<<= >>=	comparison
non-associative	= != == !== <>	comparison
left	&	bitwise and references
left	^	bitwise
left	|	bitwise

left	&&	logical
left	||	logical
non-associative	? :	left-associative
right	: +: -: *: /" %: &: |:	assignment
(n/a)	yield from	yield from
(n/a)	yield	yield
(n/a)	print	print
left	and	logical
left	xor	logical
left	or	logical

Data types

GreenTea support following data types:

- String
- Number
- Boolean
- Array
- Object
- NULL
- Function

Strings

A string is a sequence of characters

```
Hello world
"Hello world"
‘Hello world’
```

There are some standard escaped characters included in Green Tea’s string:

Sequence	Meaning
\n	linefeed (LF or 0x0A (10) in ASCII)
\r	carriage return (CR or 0x0D (13) in ASCII)
\t	horizontal tab (HT or 0x09 (9) in ASCII)
\v	vertical tab (VT or 0x0B (11) in ASCII)
\e	escape (ESC or 0x1B (27) in ASCII)
\f	form feed (FF or 0x0C (12) in ASCII)
\\	backslash
\$	dollar sign
\"	double-quote
\[0-7]{1,3}	the sequence of characters matching the regular expression is a character in octal notation, which silently overflows to fit in a byte (e.g. "\400" == "\000")

\x[0-9A-Fa-f]{1,2}	the sequence of characters matching the regular expression is a character in hexadecimal notation
\u{[0-9A-Fa-f]+}	the sequence of characters matching the regular expression is a Unicode codepoint, which will be output to the string as that codepoint's UTF-8 representation

We recommend you use quotation marks for string until you remember the word that shouldn't be used unquoted.

Number

Numbers include integer and double:

-17

3.14

0

0.1

infinite

Booleans

Booleans are either `true` or `false`

Arrays

An array is a sequence of elements. Elements could be in different types, elements also could be array. An array is a set of key-value pair, contains all unique keys. Keys could be number, string

Types of element's value could be mixed

Declare:

Use [] operator to declare value for arrays.

```
$array : [0=>a,1=>b, 2=>c] // with keys
$array_mixed_type : [1, hello, true, [4.6, false]] // auto
key, auto-increase from 0, 1, …
```

Get/assign value for element:

Use *$array_var_name[key]*

(Using above array declarations)

```
$array[1]
> b
$array[0] : test
$array[0]
```

```
> test
$array_mixed_type [3][2]
> false
```

Add new element to array:

```
$array []= $element1
```

You could iterate through array, see section about looping for detailed.

Objects

Objects are sets of properties and methods that use the common templates called classes. They will be describe more in object oriented section

Objects are represent of object oriented programming.

Initialize

```
$object : (@new $Class_name)
```

Access object peroperties or method:

use operator >>

```
$object >> (@changeName John)
$object >> $name
> John
```

NULL

NULL is the type that hold no value

Functions

You could assign a function for a var

```
@function_add $a, $b
     return $a + $b
$var_add : @function_add
```

Conditional structure

Conditional structures are where program choose which part of program to be run by using condition. Includes:

- if else structures
- ? :: operators.

If else structures

When you want to split the flow of code into multiple cases base on some conditions. You could use if clause:

if <condition>
    <action if condition is satisfied>
else
    <action if condition is not satisfied>
Both actions could be single command or block

For example, you write a simple program that test if user input even number:

```
@echo ‘Please enter an even integer:’
@read $number_input // read user input into $number_input
if $number_input%2 = 0
     @echo This is an even number
     @echo Congratulation
else
     @echo This is not an even number
```

When user run the program and enter 1

```
> Please enter an even integer:
< 1
> This is not an even number
```

When user run the program and enter 2

```
> Please enter an even integer:
< 2
> This is an even number
> Congratulation
```

Multiple if else:

Used when there are multiple cases.

Ex: there are 3 students in class with corresponding student code. Write a program let user input a number, then output their name: 1 (John), 2 (Annie), 3 (Sabrina).

```
@echo "Enter student code 1, 2 or 3"
@read $entered_student_code
if $entered_student_code = 1
     @echo John
elif $entered_student_code = 2
     echo Annie
elif $entered_student_code = 3
     echo Sabrina
else
```

```
    echo Wrong number.
```

Inline conditional expression

You can write a short expression that get value based on condition.

<condition>?<value if condition is satisfied>::< value if condition is not satisfied>

Ex: A simple guest game

```
$thinking_number:(@random_int 1, 2) // $a get value 1 or 2
randomly
@echo "I am thinking about a number: 1 or 2. Can you guest
which I am thinking?"
@read $user_input
$thinking_number = $user_input ? "Congratulation, you win" ::
"Better luck next time"
@echo $?
```

In case user guest correctly

```
> I am thinking about a number: 1 or 2. Can you guest which I
am thinking?
< 1
> Congratulation, you win
```

Otherwise

```
> I am thinking about a number: 1 or 2. Can you guest which I
am thinking?
< 2
> Better luck next time
```

Loop structures

Loop structure used when you want program run a part of it repeatedly, when you don't want to write repeated code or when you don't know exactly the times it loop. Including:

- For structure
- while structure

For structures

For structure usually use when you know how many times you will loop

Simple for structures:

for <time counts> times

<tasks to do repeatedly>

<tasks to do repeatedly> could be single command or block

`$_time` is a var that hold the loop time, starting with **0**.

Ex: Write 3 time "*Welcome to Gtlang*", with number:

```
for 3 times
     @echo $_time . ". Welcome to Gtlang"
> 1. Welcome to Gtlang.
> 2. Welcome to Gtlang.
> 3. Welcome to Gtlang.
```

You could also use `at` keyword here

Ex: Write 4 time "I'll never play games in classes.", at time 3, write "This is time 3" :

```
for 3 times
     @echo $_time.". IWelcome to Gtlang ."
     at $_time = 3
          " This is time 3"

> 1. Welcome to Gtlang.
> 2. Welcome to Gtlang.
> 3. Welcome to Gtlang. This is time 3
```

You could replace `times` with any variable:

```
for 3 $i:
     for 3 $j:
          if $j = $i:
               @echo ' * '
          else:
               @echo '   '
     @echo "\n" //new line
>   *
>      *
>         *
```

Full for structures:

for <initation>,<break_condition>,<increasement>

<tasks to do repeatedly>

Ex: Write 3 time "I'll never play games in classes, with number":

```
for $i:1,$i<3,$i++
     @echo ($i+1)". Welcome to Gtlang."
```

```
> 1. Welcome to Gtlang.
> 2. Welcome to Gtlang.
> 3. Welcome to Gtlang.
```

Foreach-as with arrays

You could iterate through elements in array and do actions for each of those element musing `foreach` and `at`.

foreach *<array>*

<action for each element>

$_key and $_value will be created to ley you access key and value of the array.

```
$list:["Annie","Bob", "John", "Sabrina"]
foreach $list
     @echo $_key ". " $_value. "\n"
//\n is the end of line.
> 0. Annie
> 1. Bob
> 2. John
> 3. Sabrina
```

In case you need both the key and the value of each element:

foreach *<array>* ***as*** *<element's value holding var>*

<action for each element>

```
$list:["Annie","Bob", "John", "Sabrina"]
foreach $list as $name
     @echo $name + "\n" // \n is the end of output line.
> Annie
> Bob
> John
> Sabrina
```

In case you need only the value of each element:

foreach *<array>* ***as*** *< element's key holding var> => <element's value holding var>*

<action for each element>

Ex: write out all name in an array which is a list of names.

Now, write out all name in an array which is a list of names with it's index.

```
$list : ["Annie","Bob", "John", "Sabrina"]
foreach $list as $index $name
     @echo $index ". " $name . "\n
> 0. Annie
> 1. Bob
> 2. John
> 3. Sabrina
```

While loop structures

Use while loop when you couldn't know how many time it loops

while <loop condition>

<actions to be executed with each loop>

When the loop condition's value is true, actions to be executed with each loop will be executed again and again. If the loop condition's value is false, the loop stop, and program continue pass this structure.

Ex: Find all Bob in an array, which is a names list.

```
$list : ["Annie","Bob", "Bob", "Sabrina"]
$i : 1
@echo "Positions of Bob in list:\n"
while $i <= 4
     if $list[$i] = Bob:
          @echo $i . "\n"
     $i++
> Positions of Bob in list:
> 2
> 3
```

Note: You should double check the loop condition to avoid an indefinite loop.

Loop controllers

Loop controllers are used to control the execution inside a loop. Includes:

- Break
- Continue

You can use loop controllers to control the loop flexibly.

Break

Break command will break current loop

Ex: : Find first Bob in an array, which is a names list. Since we don't need find continue after 1st Bob, we could use `break` after we found one.

```
$list : ["Annie","Bob", "Bob", "Sabrina"]
$i : 1
@echo "Positions of Bob in list:\n"
while $i <= 4
     if $list[$i] = Bob:
          @echo $i . "\n"
          break
     $i++
> Positions of Bob in list:
> 2
```

Bob at position 2 is ignored

You could break outer loop using break <*number*>. With number is the outer-level number of the loop to be break. If you want to break parent loop, use `break 2`; if you want to break grand parent loop, use `break 3`; ...

Ex: Find 1st Bob in an array, which is a names table.

```
$table :[...
     ["Jack", "Daniel", "Bill", "Sabrina"],...
     ["Annie", "Bob", "John", "William"],...
     ["Mike", "Muriel", "Bob", "Brittney"]...
]
foreach $table as $i $row
     foreach $row as $j $name
          if $name = "Bob"
               @echo "Found first Bob at row "...
                         .($i+1)...
                         ", column ".($j+1)
          break 2
> Found first Bob at row 2, column 2
```

Continue

To skip the current loop and continue next loop

Ex: Find all Bob in an array, which is a names list.

```
$list : ["Annie","Bob", "Bob", "Sabrina"]
$i : 1
@echo "Positions of Bob in list:\n"
while $i <= 4
     if $list[$i] != Bob
          $i++
```

```
          continue
     @echo $i . "\n"
     $i++
> Positions of Bob in list:
> 2
> 3
```

When `$list[$i]` is not *Bob*, `@echo $i . "\n"` won't be executed.

You could also use continue <number> to skip the containing outer loops, like with break.

Ex: There is an array, each line has only 1 Bob. Find Bob of each line, which is a names table.

```
$table :[...
     ["Jack", "Daniel", "Bill", "Sabrina"],...
     ["Annie", "Bob", "John", "William"],...
     ["Mike", "Muriel", "Bob", "Brittney"]...
]
foreach $table as $i, $row
     for $row as $j $name
          if $name != "Bob"
               @echo "Found Bob at row ".($i+1)...
                         ."., column ".($j+1)
               continue 2
> Found Bob at row 1, column 1
> Found Bob at row 2, column 2
```

Error handlers

GT provide 3 ways to handle errors:

error_handler_function

try...catch...finally structures

defcat...deffin (default catch)

error_handler_function

You could declare this function to handle how to show when unhamdled error happened

```
@error_handler_function($errno, $errstr, $errfile, $errline)
//do somethings
```

`errno`

The first parameter, `errno`, will be passed the level of the error raised, as an integer.

`errstr`

The second parameter, `errstr`, will be passed the error message, as a string.

`errfile`

If the callback accepts a third parameter, `errfile`, it will be passed the filename that the error was raised in, as a string.

`errline`

If the callback accepts a fourth parameter, `errline`, it will be passed the line number where the error was raised, as an integer.

`errcontext`

If the callback accepts a fifth parameter, `errcontext`, it will be passed an array that points to the active symbol table at the point the error occurred. In other words, `errcontext` will contain an array of every variable that existed in the scope the error was triggered in. User error handlers must not modify the error context.

try...catch...finally structures

Used to catch possible exception in try block. If a Exception match multiple catches, only the 1st catch block matched will run. Finally block run whenever any Exception is catch or not.

Ex: If ^DevidedByZeroException happened ($num2=0), catch block 1 will be triggered). If other exception happened, catch block 2 will run. Finally block run whenever any Exception is catch or not.

```
@divide $num1 $num2
try
     $result = $num1 / $num2
     catch ^DevidedByZeroException
          // catch block 1
          @echo “devided by zero”
          $result = false
     catch ^Exception
          // catch block 2
          @cho “other exception”
          $result = false
     finally
```

```
        return $result
```

defcat...deffin structures

You could use defcat as default catch for functions, which will catch Exceptions when running current function

defcat and deffin should be at the end of function

```
@divide $num1 $num2
     $result = $num1 / $num2
defcat
     ^DevidedByZeroException
          // catch block 1
          @echo “devided by zero”
          $result = false
     ^Exception
          // catch block 2
          @cho “other exception”
          $result = false
deffin
     return $result
```

You could use defcat and deffin outside of functions, which will handle exception for the whole program. They should be at the end of source files.

Multi-threading

Green Tea support multi threads processing. Includes:

- File-based multi-threading (simple),
- Function-based multi-threading.

Vars will not be shared among threads, you must use parameters and return values to exchange data among threads.

Only main thread could spawn sub-threads. Thread cannot be initialized from a non main thread.

At the time the main thread exit , it will wait if there is any running sub-threads with a warning message. Press Control +C to exit.

File based multi threading

Use @create_file_thread function to create a thread that execute another file. On thread file, you could return value to main thread using thread_return. Don't use exit in thread file, it will also exit main thread.

File *thread1.gtc* :

```
for 3 times
     @sleep 1000 // wait 1s
     @echo "Thread1 – running for ". $_time . "s" \n
@thread_return (@thread_get_args)[0] // file argument #1
```

File *main.glc*:

```
$thread1 : (@thread_start "thread1.gtc" Hello)
@echo "Start thread1"
@thread_get_result $thread1
$result : (@thread_result $thread1)
@sleep 1000 // sleep for 1s
"Thread1 result is: "  $result
```

Run *main.glc*

```
> Start thread1
> Thread1 – running for 1s
> Thread1 – running for 2s
> Thread1 – running for 3s
> Thread1 result is: Hello
```

File based multi threading should have parameters which are numbers or strings, like "Hello" in the above example.

Function based multi-threading

Use @create_function_thread function to create a thread that execute a function. On thread file, you could return value to main thread using normal return. Don't use exit in thread function, it will also exit main thread.

```
@thread1 $param
     for 3 times
          @sleep 1 // wait 1s
          "Thread1 – running for ". $_time . "s"
     return $param // file argument #1
$thread1_function : @thread1
$thread1 :(@create_file_thread $thread1_function, "Hello")
@echo "Start thread1"
@thread_start $thread1
@thread_wait $thread1
$result : (@thread_result $thread1)
```

```
"Thread1 result is: " + $result
> Start thread1
> Thread1 – running for 1s
> Thread1 – running for 2s
> Thread1 – running for 3s
> Thread1 result is: Hello
```

Function based multi threading could have any kind of parameters.

Class

Classes are "template" to create objects.

Class define methods (functions) and properties (variables) of object.

Class start with **^**

To access object's assets, use dot operator.

Ex:

```
^Human:
     $name
     $email
     @print_info:
          @echo "$name - $email \n"
```

Then we could create object

```
$people_1 :(@new ^Human)
$people_1.$name: John
$people_1.$email: "john@gmail.com"
$people_1.(@print_info)
> John - john@gmail.com
```

Constructor

Constructors usually add data to object when create a new one, you could create several using **new**. GreenTea create a constructor based on the class properties count.

Ex:

```
^Human:
     $name
     $email

     @print_info:
          @echo $name " - " $email \n"
^Human @new John, "john@gmail.com"
```

```
$?>>(@print_info)
> John - john@gmail.com
@new ^Human, John
$?>>(@print_info)
> John -
```

Access modifiers

Green Tea support private and public (default) modifier.

Private asset could not be access outside object

Ex:

```
^Human:
     $name
     private $email

     @print_info:
          @echo "$name - $email \n"
@new ^Human, John, "john@gmail.com"
$?>>(@print_info)
> John - john@gmail.com
@new ^Human, John, "john@gmail.com"
@echo $?>>$name
> John
@new ^Human, John, "john@gmail.com"
@echo ($?>>$email)
```

(Error – permission denied)

Inherit

One class could inherit some other classes using operator << . When doing so, child class could add assets, overload method of the parent class. But could not remove or change type of the parent one's assets.

With class Human above, we could have:

```
^Student << ^Human:
     $point
     @print_info:
          @echo "$name - $email - $point \n"
$student1: (@new ^Student, John, "john@gmail.com")
$student1>>$point: 4
$student1>>(@print_info)
> John - john@gmail.com - 8
```

Static

Static assets are asset of the class, user don't need to create object to access those static assets.

```
^Human:
     static $spicies: Homo Sapien
     static @getSpicies
          return $spicies
^Human>>(@getSpicies)
> Homo Sapien
```

Convert from array to object:

An array with all keys satisfied variable's name rules could be converted to object with no class and all public properties.

Ex:

```
$obj: (@to_obj  ["name"=>"John", "age"=> 24])
```

Generate class from no class object

An object with no class could generate a class using function `@gen_class`.

Ex:

```
^Human1 = @gen_class $obj
```

Now `$obj` belong to class `^Human1`.

Built-in functions

System

@print <value>

print a string to the screen, can print array also

```
$arr:[1,2,3]
@print $arr
> arr:
> [
>    0 => 1
>    1 => 2
>    2 => 3
> ]
```

@echo <value>

print a string to the screen

```
@echo hello
> hello
```

@shell <string>

exec a command on OS

```
$str:@shell ls
@print $str
> arr:
> [
>    stdout => Documents Downloads file.
>    stderr =>
>    exit_code => 0
> ]
```

@input

read string from input

```
$a: @input
echo $a
< abc
> abc
```

String

@split <delimiter>, <string>

Splits the string into an array using the specified delimiter.

Alias: @explode

```
"," @split  "a,b,c" // result: [ a, b, c ]
```

@length <string>

Returns the length of a string.

```
@length abc
> 3
```

@substr <string>, <start>, <length>

Returns a substring from the given string.

```
$a: @substr "abcde", 1, 3
```

```
@echo $a
> bcd
```

@str_pos <haystack>, <needle>

Returns the position of the first occurrence of a substring.

```
@strpos abcbdbe, b // return 1
```

@str_m_pos <haystack>, <needle>

Returns all matched positions (case-sensitive).

```
@strpos abcbdbe, b // return [1,3,5]
```

@str_i_pos <haystack>, <needle>

Returns the first position of a substring (case-insensitive).

```
@strpos abcbdbe, B // return 1
```

@str_m_i_pos <haystack>, <needle>

Returns all positions of needle in haystack (case-insensitive).

```
@strpos abcbdbe, B // return [1,3,5]
```

@str_to_lower <string>

Converts the string to lowercase.

```
@str_to_lower Hello World // return “hello world”
```

@str_to_upper <string>

Converts the string to uppercase.

```
@str_to_lower Hello World // return “HELLO WORLD”
```

@is_preg_match $pattern, $subject, $is_unicode

Check if the string `$subject` matches the regular expression `$pattern`.

- pattern: Regular expression pattern (e.g. /abc/, or /[\p{L}]/u for Unicode).

- subject: The string to test.

- is_unicode (default false): If true, treat pattern as Unicode.

Returns true if at least one match is found, otherwise false.

```
@is_preg_match ”[a-z], “abc”, false
@echo $?
> true
```

@preg_replace_all <pattern>, <replacement>, <subject>, <is_unicode>

Replace all occurrences matching <pattern> in <subject> with <replacement>.

- pattern: Regular expression pattern (e.g. /[0-9]+/).

- replacement: Replacement string.

- subject: The input string.

- is_unicode (default false): If true, treat pattern as Unicode.

Returns the string after replacements.

```
@preg_match_all "/[a-z]/", "abc", false
@print $@
> arr:
> [
>     0 => a
>     1 => b
>     2 => c
> ]
```

@preg_match_all <pattern>, <subject>, <is_unicode>

Find all substrings in `$subject` that match the regular expression `$pattern`.

- pattern: Regular expression pattern (e.g. /[a-z]/ or /[\p{L}]/u).

- subject: The string to search.

- is_unicode (default false): If true, treat pattern as Unicode.

```
@preg_replace_all "/[\\p{L}]/u", "*", Привет-мир
, false
@echo $?
"******_***"
```

Array

@implode <glue>, <array>

Joins array elements into a string, separated by the specified glue.

```
"," @join  [ a, b, c ] // result: "a,b,c"
```

@find <array>, <value>

Returns the index of all matching value in the array.

```
[a,b,c,b,d,b] @find b // return [1,3,5]
```

@contains <array>, <key>

Returns true if the array contains the specified key.

```
[a=>1,b=>2,c=>true,b,d,b] @find  a // return true
```

@merge <array1>, <array2>

Merges two arrays. Keys will be resetted.

```
[4=>a,t=>b] @merge [3,4] // result: [0=>a, 1=>b, 2=>3, 3=>4]
```

Files & Folders

@mkdir <path>

Creates a new directory.

```
mkdir “Downloads”
```

@file_read <path>

Reads the contents of a file.

```
$abc: @file_read “abc.txt”
@echo $abc
> This is the texts that is in abc.txt
>
```

@file_write <path>, <content>

Writes content to a file (appends existing content). This function is t-safe.

```
@file_write “abc.txt” abc
$abc: @file_read “abc.txt”
@echo $abc
>  This is the texts that is in abc.txt
>  abc
```

@file_init <path>, <content>

Create file if not exists (also create parent folder recursively), or

Delete files content if it exists.

This function is thread-safe.

```
@file_init “./tmp_foleder/tmp.file”
```

@delete_path <path>

Deletes the specified file or directory.

```
@delete_path “abc.txt”
```

@file_exists <path>

Returns true if the file or directory exists.

```
@file_exists “abc.txt” // false
```

Network

@fetch_url <url>

Fetches the content of a URL. Returns an object with keys like response , code.

```
$response:@fetch_url “http://localhost”
@print $response
> arr:
> [
>    response => <!DOCTYPE html ><html></html>
>
>    code => 200
> ]
```

@call_restapi <url>, <method>, <body>, {array of headers}, <valid_ssl = true>

Send an HTTP request to the specified REST API endpoint.

- url: The API endpoint URL (string).

- method>: HTTP method (e.g., "GET", "POST", "PUT", "DELETE").

- body: Request body content as a string (for methods like POST or PUT). Use empty string "" if no body. If body is an array, it will automatically be converted into JSON.

- array of headers: An array of strings for HTTP headers, e.g. ["Content-Type: application/json", "Authorization: Bearer ..."].

- valid_ssl (optional, default true): If true, validate SSL certificates; if false, allow insecure SSL connections.

Returns the response body string and response code.

```
@call_restapi "https://api.example.com/data", "POST",
"{\"key\":\"value\"}", ["Content-Type: application/json"],
true
@print $?
> array:
> [
>    response => success
>    code => 200
> ]
```

Mysql

@mysql_connect <host>,<port>,<user>,<password>,<database>

Connect to mysql.

```
$conn_id:@mysql_connect localhost, 3306, mike, mike123,
mike_db
```

@mysql_query <connection_id>,<query>

Send a query to mysql. This function is thread-safe.

```
@mysql_query $conn_id, “select * from data”
> arr:
> [
>    0 => arr:
>    [
>        id => 1
>        name => John
>        Age => 3
>    ]
>    1 => arr:
>    [
>        id => 2
>        name => Anne
>        Age => 4
>    ]
> ]
```

@mysql_close <connection_id>

Close the connection to mysql.

```
@mysql_close $conn_id
```

Math

@abs <number>

Return the absolute value of the number.

```
@abs -5
@echo $?
> 5
```

@sqrt <number>

Return the square root of the number (number >= 0).

```
@sqrt 9
@echo $?
> 3
```

@pow <base>, <exponent>

Return base raised to the power exponent.

```
@pow 2, 3
@echo $?
> 8
```

@exp <number>

Return e (Euler's number) raised to the power number.

```
@exp 1
@echo $?
> 2.718281828459
```

@log <number>

Return the natural logarithm (base e) of number (number > 0).

```
@log 2.718281828459
@echo $?
> 1
```

@log10 <number>

Return the base-10 logarithm of number (number > 0).

```
@log10 100
@echo $?
```

```
> 2
```

@sin <radian>

Return the sine of an angle in radians.

```
@sin 1.5708
@echo $?
> 1
```

@cos <radian>

Return the cosine of an angle in radians.

```
@cos 0
@echo $?
> 1
```

@tan <radian>

Return the tangent of an angle in radians.

```
@tan 0.7854
@echo $?
> 1
```

@asin <value>

Return the arcsine (inverse sine) of value in [-1,1], result in radians.

```
@asin 1
@echo $?
> 1.5708
```

@acos <value>

Return the arccosine (inverse cosine) of value in [-1,1], result in radians.

```
@acos 1
@echo $?
> 0
```

@atan <number>

Return the arctangent (inverse tangent) of the number, result in radians.

```
@atan 1
```

```
@echo $?
> 0.7854
```

@round <number>

Round the number to the nearest integer.

```
@round 3.6
@echo $?
> 4
```

@floor <number>

Return the largest integer less than or equal to the number.

```
@floor 3.6
@echo $?
> 3
```

@ceil <number>

Return the smallest integer greater than or equal to the number.

```
@ceil 3.1
@echo $?
> 4
```

@trunc <number>

Return the integer part of the number by truncation.

```
@trunc 3.9
@echo $?
> 3
```

@rand_int <min>, <max>

Return a random integer between min and max (inclusive).

```
@rand_int 1, 10
@echo $?
> 7
```

@rand_float <min>, <max>

Return a random floating-point number between min (inclusive) and max (exclusive).

```
@rand_float 0.0, 1.0
@echo $?
> 0.345678
```

Date Time

@echo_current_time

Print the current time in "HH:mm:ss" format.

```
@echo_current_time
> 14:23:45
```

@echo_current_date

Print the current date in "YYYY-MM-DD" format.

```
@echo_current_date
> 2025-06-03
```

@time

Get Human readable values from a timestamp.

```
@time
@print $?
> [
> hour => 13
> min => 5
> sec => 27
> year => 2025
> month => 6
> day => 3
> weekday => 2
> yearday => 153
> week => 23
> ]
```

@diff_time <start_time>, <end_time>

Return the difference in seconds between end_time and start_time.

```
@diff_time 1685799000, 1685799825
@echo $?
> 825
```

@format_time <timestamp>, <format_string>

Format the given Unix timestamp to a string according to format_string.

Example of format_string: "%Y-%m-%d %H:%M:%S"

```
@format_time 1685799825, "%Y-%m-%d %H:%M:%S"
@echo $?
> 2025-06-03 14:23:45
```

@now

Return the current timestamp with high precision (could include milliseconds).

```
@now
@echo $?
1685799825.345
```

You are already a programmer. Give us 5 minutes, you will then know green tea programming language.

Variables are **$**, functions are **@**.

```
$number : 3
@echo_hello_world
@echo "Hello world"
(@echo_hello_world)
> Hello World
```

Green Tea blocks of commands are indent-based. Just like Python, but you can mix spaces and tabs.

```
if $a < 0
     @echo "lesser than 0"
```

Assignments are **:** and **=** is comparision.

```
$a:3 // assignment
$a=3 // comparision
```

For:

```
for 3 times
     @echo $_time + " "
     if $_time = 3
          @echo " (end)"
>  1 2 3 (end)
```

You know the basic. If you have any question, find it on our full document.