TeaScript Documentation

Release beta

Vihan

February 15, 2016

Contents

TeaScript is a powerful JavaScript golfing language created by StackExchange, PPCG, user Downgoat. TeaScript compiles to JavaScript and contains many helpful features for golfing

Contents:

CHAPTER 1

Installation

Running TeaScript is pretty simple, they're multiple ways you can do this

1.1 Web Interface

TeaScript has a pretty sweet web interface and is the best enviorment for running TeaScript. All extensions are packaged up and usage is pretty straight-forward.

- Web Interface
- Alternate URL

1.2 Command Line

If you wish to run TeaScript from the command line, ensure you have **SpiderMonkey 38** or higher installed. They're multiple ways to get started

1.2.1 Auto-Install Script

Install `teascript` from the GitHub

Give `teascript` the correct permissions

```
$ chmod +x teascript
```

Run `teascript`, and it should install the correct files. Enter the code, and then the input, `,` seperated.

```
$ ./teascript
TeaScript not installed. Installing TeaScript...
# ...
Code: # <TeaScript Code Here>
Input: # <Input Here> e.g.: Input 1,Input 2,Input 3,...
# <Output Here>
```

The next time you run TeaScript, it'll detect the *TeaScript/* folder and won't need to reinstall the dependencies.

1.2.2 Manual Installation

You can also manually install/run TeaScript if you're having issues with the script

Install the following files

- Everything within /src/v2
- `sh.js` from /src/sh/sh.js

Edit `teascript.js` and replace `window` with `this`, and `/*props.json*/` with `JSON.parse(read("props.json"))`

Note: Different enviorments might use a different function than `read`

node.js `fs.readFile`

rhino `readFile`

spidermonkey `read`

CHAPTER 2

Getting Started

Getting started with TeaScript is *very* easy especially if you have prior JavaScript knowledge.

This was the main goal of TeaScript was a JavaScript golfing language that *was* JavaScript but with shorter property names. This slowly evolved but TeaScript is backwards compatible so every JavaScript program is a valid TeaScript program

2.1 Literals

Literals are very simple in TeaScript as they are the same as JavaScript literals

2.1.1 Strings

They are three types of strings

```
"Double Quoted String Literal"
'Single Quoted String Literal'
`String templates`
```

String templates have a few extra features such as:

```
`String templates support inline
newlines and code such as ${2+1}`
```

2.1.2 Numbers

Numbers are also, just *numbers*:

```
12345 // Decimal
12.34 // Decimal
1e23) // Scientific Notation
0xFA) // Hexadecimal
0b10) // Binary
0o18) // Octal
```

2.1.3 RegExp

TeaScript has support for RegExp literals and they're just the same as JavaScript

```
/[A-Za-z]/gi
```

By TeaScript 3.1, I hope to have *XRegExp*, implemented which should allow your RegExp literals to look like:

```
/\u{L}+/u
```

2.1.4 Functions

Functions are the same as JavaScript too.

```
(a,b,c)=>a+b+c // Arguments[a,b,c] adds them together
a=>a           // Single argument a, returns a
```

This is quite lengthy so I've added the `#` operator which automatically expands to `(l,i,a,b)=>` at compile time

```
#l+i+a  // Arguments[l,i,a,b] adds first 3
```

2.2 Input

2.2.1 Input Options

The user can decide how (s)he wants the input. TeaScript supports all of the following input types:

- String (default)
- Number
- Array

2.2.2 Getting the Input

The input is stored in various variables:

Input #	Variable Name
1	`x`
2	`y`
3	`z`

Need More inputs? An array of all the inputs is stored in the _ variable.

If the input is an array, the 1st input will be split upon `,` and each item will become a seperate input. For the code:

```
Input 1,Input 2,Input 3
```

The values are:

Input Value	Variable Name
`Input 1`	`x`
`Input 2`	`y`
`Input 3`	`z`

CHAPTER 3

New Features

As said before, TeaScript is an *extension* of JavaScript meaning it adds features to JavaScript. Here you will learn about some of the features TeaScript adds to JavaScript

3.1 Modified RegExp

RegExp literals have been expanded and are now more powerful than ever with custom character classes and hopefully even more features to come.

3.1.1 Custom Character Classes

TeaScript adds custom character classes (e.g. `\w`) to a RegExp literal. These are esentially shorthands for character classes, an example

```
/[A-Za-z]/  // Before
/\L/        // After
/[A-Za-z]/  // At compile-time
```

New Character Classes

Name	Value
\\A	[A-Z]
\\a	[a-z]
\\L	[A-Za-z]
\\N	[A-Za-z0-9]

3.2 Operators

3.2.1 # Operator

The `#` operator is one that is *very* useful. It's a shorthand for function declerations that you can use where ever. `#` expands to `(l,i,a,b)=>`

```
(a,b,c,d)=>a+b+c+d // Before
#l+i+a+b           // After
```

3.2.2 @ Operator

The `@` operator is similar to the `#` operator, but if you ever have two nested lambdas, you can use this. `@` expands to `(q,r,s,t)=>q`.

```
(a,b,c,d)=>aT2)+b+d // Before
@T2)+r+t            // After
```

3.2.3 ƒ Operator

`ƒ` expands to `f=(l,i,a,b)=>`, this can be used to create recursive functions easily without having to manually add a decleration

```
f=(a,b)=>a<1?b:f(a--,b++); // Before
ƒl<1?i:f(l--,i++);         // After
```

3.2.4 Σ Operator

The Σ operator can be used to loop through arrays and strings, it expands to `.l((l,i,a,b)=>`.

```
xΣlc // Maps char codes
xΣi  // Generates range
```

3.2.5 ? Operator

This operator has 2 uses depending on where you use it.

Interrupting Property expansion

If you ever need to use a JavaScript property name and TeaScript thinks it's a TeaScript property, insert a `?` after the property

```
x.search(/\A/) // JavaScript
x.search?/\A   // TeaScript
```

Closing Parenthesis

```
xl(#lT(2r("foo"[1]))) // Before
xl(#lT(2r("foo"[1?    // After
```

CHAPTER 4

Variables

TeaScript has many variables which are pre-initalized to various values but you can also use them for custom variables:

4.1 Assignment

To assign a variable you can easier use a shorthand or the native JavaScript ways.

4.1.1 Shorthands

Using *ƒ*

This can be used to assign functions both recursive and not. To learn more about this operator, see *ƒ Operator*.

```
f=(l,i)=>l<1?i:f(l--,i++) // Before
fl<1?i:l--:i++            // After

f=l=>{for(i=0;i<l;i*=i)} // Before
f{for(i=0;i<l;i*=i       // After
```

4.1.2 Assignment Operator

You an also just use the assignment operator to assign variables. Some one-letter variables are already preassigned so you may be able to skip the definition.

```
var i=0;  // Before
i=0;      // After
```

4.2 Predefined variables

The predefined variables can be overwritten.

Variables	Value
p	`" "`
u	`""`
n	`"\n"`
dfjkv	`0`
o	`1`
g	`2`
e	`10`
h	`100`
m	`16`
f	`false`
½	`1/2`
¼	`1/4`
π	`3.14159265358979323846`
Φ	`1.61803398874989484820`

```
for(var j=0;j<x;j++); // Before
for(;j<x;j++);        // After
```

CHAPTER 5

Auto-Golfing

`Auto-Golf` is a feature which performs automatic golfing for you. It provides a few features.

Hint: The `Un-Auto-Golf` will do the opposite and will attempt to make code more readable.

5.1 Unicode Shortcuts

Unicode shortcuts are a way to get code as short as possible without doing any work! What are they? Unicode shortcuts are 1-byte long unicode characters which expand to longer TeaScript code at compile time. Confusing, here's an example:

```
£lc)    // Original Code
xl(#lc) // Code at compile-time
```

What if I want to use a unicode character in my code. Unicode characters in literals (i.e. Strings, RegExps, Snippets) are not converted. If for some reason you do want a unicode property name, it can be used by using a `\\` before the character

```
\£lc     // Original Code
£lc // Code at compile-time
```

So how do you use them? You simpily click the `Auto-Golf` button.

5.2 Ommiting characters

5.2.1 Removing Brackets

If you have a function, and then a literal, you can ommit the `(` before it. You can also ommit ending `)` and other brackets

```
MF(32) // Before
MF32   // After

MF(3,x[32]) // Before
MF3,x[32    // After
```

5.2.2 Removing Literal Endings

Endings of literal characters can be ommited, this includes Strings, RegExps, and Snippets.

```
"Foo"  // Before
"Foo   // After

`Foo`  // Before
`Foo   // After

/Fo{2}/ // Before
/Fo{2}  // After
```

CHAPTER 6

Basic Examples

6.1 String Tasks

6.1.1 Hello, World

Explanation

`"` begins a string literal.

Outputs `Hello, World`

```
"Hello, World!
```

6.1.2 cat

Explanation

`x` contains the input

Outputs the input

```
x
```

6.2 Number Tasks

6.2.1 Fibonacci

Explanation

`F` is a Fibonacci function. `x` is the input

Given `n` this calculated the `n`th Fibonacci number

```
F(x
```

6.2.2 Primality Test

Explanation

`mP` is a primality check function

Calculates whether a given number is prime

```
mP(x
```

6.3 Array Tasks

6.3.1 Cycling Arrays

Explanation

`C` cycles an array, `x` is the input

Cycle an array 1 spots

```
xC1
```

6.3.2 Average of Numbers

Explanation

`x` is the input, `x` is a sum getter. `n` is the length. Compiled, this is `x.x/x.n` or `x.sum/x.length`

Calculates the average of numbers

```
xx/xn
```

CHAPTER 7

Error Handling

7.1 Approaching Literal Maximum

This is the only error that can be thrown during compilation at the moment. This is caused when a literal (i.e. String, RegEx) is unbalanced.

```
"unclosed string
/unclosed regex
$unclosed snippet
```

7.2 Dependency not found: babel

`babeljs`, was not able to be loaded, check your network connection and ensure babel is connected. If you believe this shouldn't be occuring don't hesitate to report it on Github.

7.3 Existing sub___: Slot used

During enviorment generation, a few errors can occur, this occurs when TeaScript is trying to assign a variable but it has already been assigned. You can override this by force setting `TEASCRIPT_ENV` to false, each time enviroment generation takes place

Note: This is a technical error, you are either using an unsupported browser/enviorment or there is a bug in TeaScript. If you believe it's a bug, don't hesitate to report it on Github

7.4 Invalid location, ___, error ___

If this ever occurs, TeaScript has encountered an issue with the `props.json` file, possible fixes are reinstalling the `props.json`. If this continues, don't hesitate to report it on Github.

7.5 Unexpected Type: ___ at ___

This is another error with the `props.json`, check to make sure the json is valid. Try reinstalling the `props.json`, and if that doesn't work, don't hesitate to report it on Github.

7.6 Duplicate Getter: ___

An attempt was made to assign a getter to an already assigned key. To diagnose this, try looking for duplicate getters in the `props.json` and change/remove them.

7.7 *Any* other error

All other errors are either JS Runtime or syntax errors, which can be solved by entering *Debugging Mode*

7.7.1 Syntax Errors

A syntax error starts with `SyntaxError:`, and is an error with the syntax itself, the error should display from where the error originated and by looking at the previous compilation steps, you may be able to identify where the error occured.

This error could of originated in any of the following compilation stages:

- String Balancing
- Unicode Shortcuts
- Property Expansion
- Paranthesis Balancing
- babel compilation

If you believe the error originated during babel compilation, report the error at babel's Github.

7.7.2 Runtime Errors

Any other error is a JS runtime error which is usually caused by referencing a variable that doesn't exist. Runtime errors are errors with the code itself rather than the syntax. Try to break down your code and try to identify where the error is originating. If you believe this error shouldn't be happening, don't hesitate to report it on Github.