AUTOMATICWRITING<>

A Programming Language for Algorithmic Literature

« Écritures mobiles » Research project
HEAD -Genève
DOUGLAS EDRIC STANLEY

Note

This is a an early technical document (September 2015) and has not yet been fully updated with the latest implementations of the language. Beware of possible inconsistencies.

Definition

AutomaticWriting<> is a programming language that mixes code and text in the same document. AutomaticWriting<> allows for authors to write text accompanied by code.

AutomaticWriting<> is the language component of the Automatic.ink platform. Other components of this platform include the AutomaticWriter and the AutomaticReader.

Workflow

AutomaticWriting<> is inspired by HTML, in that HTML mixes both functional text (code) and written text (content) in the same unformatted document. In fact, AutomaticWriting<> is designed to export standard HTML for the content, standard CSS for layout and formatting, and standard Javascript for the code.

AutomaticWriting<> works similarly to Markdown, i.e. through character combinations that generate standard text output. The generated output of AutomaticWriting<> is HTML with Javascript. This simplifies the process of writing code, but also makes the text it modifies more readable.

AutomaticWriting<> can easily be mixed in with standard unformatted text — any standard text editor will do —, and even be highlighted or filtered via the interface through colouring or masking. Code remains standard text, just like HTML.

Colouring and conversion takes place only at the level of the interface or upon export to a browser. An AutomaticWriting<> document remains in its format as long as possible.

The Automatic.Writer software application was written specifically for the creation of text written in this language.

Format

An AutomaticWriting<> document can be identified by its extension “code.automat”. For example, a file named “index.automat” remains in the AutomaticWriting<> format, and has not yet been converted into HTML.

Functions

Executable *functions* are always **defined** on a new line with a single word without spaces. This word is followed immediately by (parentheses) which are immediately followed by {{braces}}. All the executable code goes between the two braces.

```
load(){{
    var allWords = document.words; // store all words in array
}}
```

Associations

Inversely, Code is **associated** with any standard HTML block using {{braces}} followed immediately by <chevrons> containing the code. If the braces are at the beginning and end of a paragraph of text, the code will be included in the <p> tag. If the braces are somewhere in the middle of a paragraph of text, a tag will be created for the associated code.

```
This is a normal paragraph with normal formatting. However,
{{inside}}<touch=hide()> of this paragraph, is a `<span>` element
that can be acted upon independently of the block.
```

An HTML tag can be created by opening and closing a brace by itself on a new line. This allows for encapsulated code that respects the HTML hierarchy.

```
{{

Some random paragraph that will be hidden upon touch.

}}<touch=hide()>
```

The code defined inside the chevrons uses the format of `<event=functionName()>`. The *event* defines when code will be executed, and *functionName()* defines the code that will be executed. Spaces separate different events, for example:

```
{{this text block is affected by three possible events}}
<touch=reset() loop(1)=invert() drag=scramble()>
```

If several *functions()* need to executing for a single event, semicolons are used

```
{{this text block will be severely affected by three different
functions when touched}}<touch=reset();invert();scramble()>
```

Note that there are no spaces between each *function()*.

body

Code can be associated with the `<body>` of an HTML document by using empty braces with no content, i.e. {{}}

```
{{}}<load="load" touch="reset">

reset(){{
    for(var i=0; i<document.words.length; i++) {
        document.words[i] = allWords[i];
    }
}}

invert(){{
    foreach(var word in document.words) {
        var oldWord = word;
        for(var i=0; i<word.length; i++) {
```

```
            word.letter[i] = oldWord[oldWord.length-i];
        }
    }
}}

scramble() {{
        foreach(var letter in letters) {
            // do something here
        }
    }}
```

Prototypes

Prototypes can be created in a separate file. They are written with three periods inside of braces {{…}} which represent the portion of text that will be affected by the code.

For example, the following code will surround a block of text with the “scramble()” action, which will be called in a loop every n-th second.

```
{{…}}<loop(speed)=scramble()>

{{This, for example, could be the resulting paragraph of text that would be affected by the "scramble()" action for each frame. At a rate of approximately once every 5 seconds, this paragraph would be scrambled}}<loop(5)=scramble()>
```

Affecting a selection of text with a *prototype* requires using an `AutomaticWriting<>` interface. Begin by selecting the text you wish to associate with some code, drag a *prototype* onto that selection, define its argument (ex: speed), and it will associate the code to your selected text.

Identifiers

A hash symbol can create an identifier for any block. They are used to identify parts of text that will be manipulated by other parts of code. Identifiers are placed at the beginning of an *association*. It should be noted that associations can potentially contain no code.

```
#opening{{The story begins at the beginning of time, at the
beginning of space.}}
```

Classes

A class is defined similarly to an identifier, but uses a period instead of a hash symbol.

```
#opening{{The story begins at the beginning of time, at the
beginning of space. Our hero, .hero{{Ralph}}, does not yet know
what is about to hit him, because he does not yet exist. If only
.hero{{Ralph}} could have benefitted from the gift of foresight, he
probably would have avoided this adventure entirely.}}
```

Inline code

Code can be injected directly into an element, just as in Javascript. Merely open a series of braces and place the code inside.

```
{{Clicking on this paragraph will take us to some stupid website.}}
<touch={goto('http://www.abstractmachine.net');}>
```