HTML Web Page That Shows Its Own Source Code

Tom Verhoeff

November 2009

1 Introduction

A well-known programming challenge is to write a program that prints its own source code. For interpreted languages, where a program can access itself, that is not much of a challenge. In the old BASIC interpreter that I used on our MITS Altair 8800 [1], one could write

```
10 LIST
```

to have the program print its own listing. But this is considered cheating. The program is not producing a *copy* of itself, it is simply retrieving itself and printing that. In compiled languages, this kind of cheating typically will not work so easily. If the source code is available in a text file, then the program could access it and print its contents. This is not the intention of the challenge.

The real challenge is to write a program

1. that is not empty,
2. that processes no external input, and
3. that produces its own source code as output.

Such a program is called a *quine* [5] after the philosopher Willard Van Orman Quine [6]. Some people find it an extra challenge to come up with quines that are the shortest possible for the programming language concerned. Personally, I am more interested in quines that are very easy to understand. So, I impose additional requirements:

4. the quine must be easy to understand
5. the quine must be easy to verify (for correct operation)
6. the quine must be easy to construct (preferably by a program, rather than manual tweaking)
7. the quine must adhere to (reasonable) coding conventions, including a proper layout (indenting), no overly long lines, decent variable names, proper structure (user-defined functions), etc.

When browsing the World Wide Web with quines in mind, a natural question is whether one can design a web page that shows its own source code.

2 Problem Statement

The challenge is to write an HTML web page (using some scripting) that has the following properties:

- The web page must conform to HTML 4.01 Strict, as approved by the W3C HTML Validator [4].
- The web page must show its own source code, with the exact same formatting.
- The web page may present other (meta) information as well, such as the name of the author.
- The web page consists of a single HTML file, and does not include or refer to other source files, such as externally residing scripts or CSS settings.

3 First Attempt

A minimal (not conforming to standards) web page that attempts to show its own source code, could be:

```
<pre>
&lt;pre&gt;
&amp;lt;pre&amp;gt;
...
&amp;lt;/pre&amp;gt;
&lt;/pre&gt;
</pre>
```

However, this fails, because it actually shows:

```
<pre>
&lt;pre&gt;
...
&lt;/pre&gt;
</pre>
```

4 Observations and Design Decisions

1. Some kind of scripting will be needed to accomplish the goal, because some data needs to be reused (used twice; processed in two passes), so as to avoid an infinite regress. We decided to use JavaScript [2].

2. A web page conforming to the HTML 4.01 Strict standard [3] is not empty. A conforming web page looks like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
<meta http-equiv="content-type"
  content="text/html; charset=utf-8">
<title>Title text</title>
</head>
<body>
Body material
</body>
</html>
```

3. The preceding two decisions imply that the script is not empty, and must contain at least one `document.write()` call to inject the source code in the web page.

4. We aim to show the source code of the web page inside a `PRE` element, because that makes it easier to present the source code in the same layout as the (actual) source code, including indentation and line breaks. That part will look like

```
<pre>
Source Code
</pre>
```

5. We will generate the content of the `PRE` element by injection from the script. That will look like:

```
<pre>
<script type="text/javascript">
document.write(SourceCodeAsString);
</script>
</pre>
```

6. The JavaScript fragments need to appear inside `SCRIPT` elements. Inside `SCRIPT` elements, the characters `<`, `>`, and `&` are forbidden. Thus, we need to write the script fragments such that these characters are not used literally. For producing strings, we can accomplish this by including the following constant definitions, using the escape mechanism for string literals in JavaScript:

```
var LT = '\074'; // less than sign
var GT = '\076'; // greater than sign
var ET = '\046'; // ampersand
```

Note that in script fragments we cannot use the arithmetic and string comparison operators <, <=, >, or >=. (Such comparisons could be done by creating a string with the comparison expression, and evaluating it with the standard function eval(), but I consider this a form of cheating.) If comparisons turn out to be needed, then we will have to find a way around them.

7. Inside a PRE element, it is necessary to encode some symbols:

   - < by <
   - > by >
   - & by &

   in order to make them appear as intended. For this purpose, we define the JavaScript function preify(s) that converts string s into a form suitable for the PRE body:

```
function preify(s) {
  // convert string s to be acceptable as body for PRE element
  // requires: s is a string
  // ensures: result is a copy of s,
  //   with characters LT, GT, and ET encoded in HTML
  var result = '';
  for (var i = 0; i != s.length; ++i) {
    switch (s.charAt(i)) {
      case LT: result += ET+'lt;'; break;
      case GT: result += ET+'gt;'; break;
      case ET: result += ET+'amp;'; break;
      default: result += s.charAt(i);
    }
  }
  return result;
}
```

   Thus, the injection will look like:

```
<pre>
<script type="text/javascript">
document.write(preify(SourceCodeAsString));
</script>
</pre>
```

8. To avoid an infinite regress, we put (an encoding of) the source-code-with-a-marker in a string constant, aptly named DNA. For that purpose, we need a script statement like

```
var DNA = X;
```

where `X` is (an expression yielding) a string containing (an encoding of) the source code, but in that source code we keep the `X` as shown here (or some other symbol) to mark the place where the DNA must be inserted.

9. As marker, we will use a character that does not otherwise appear in the scripts (`X` seems fine). However, in order to recognize the marker, we need a way of referring to it without mentioning it. We can do so via another constant:

```
var V  = '\130'; // captial x
```

The name `V` is a mnemonic for insertion.

10. There exists a standard JavaScript facility for performing substitutions on strings. For strings `s`, `t`, and `u`, the result of `s.replace(t, u)` is a new string consisting of `s` in which the first occurrence of `t` in `s` is replaced by `u`.

Hence, `DNA.replace(V, u)` returns a string whose value is `DNA` in which (the first occurrence of) the marker named `V` (i.e., the character `X`) has been replaced by `u`.

11. If possible, we will try to avoid encoding things in `DNA`, and prefer to put the source code unencoded in the string.

12. The string `DNA` containing (most of) the source code (possibly in encoded form), must also be written as a JavaScript expression, so that the script can use it. For this purpose, we define the function `uneval(s)` that converts string `s` into a JavaScript expression that has `s` as value.

If we have function `uneval()`, we can refine the injection further:

```
<pre>
<script type="text/javascript">
document.write(preify(DNA.replace(V, uneval(DNA))));
</script>
</pre>
```

13. There are many expressions that evaluate to a given string `s`. We need to produce an expression that is the same as the one appearing in the source code. This can be accomplished by putting the string into source code using the same function `uneval()`.

Many characters of `s` can be included in the expression by putting them in a string literal like `'...'`, where we chose to use the apostrophe (single quote) `'` as delimiter for string literals.

Some characters in the string to be converted cannot be put into a string literal ‘as is’. There are several exceptions:

- non-printable characters (like newline), also known as control characters;
- the delimiter used for string literals, viz. `'`.

In JavaScript, these characters can be included in string literals through the so-called escape sequence starting with a backslash `\`. It then also becomes necessary to have a way to put backslashes in the string. Here are the solutions (i.e., the following escape sequences can be put inside a string literal):

- newline: `\n`
- apostrophe: `\'`
- backslash: `\\`

There are also some other constraints to be taken into account:

- the expression must adhere to the coding conventions, in particular, it should not contain overly long lines;
- the expression must not contain forbidden characters: because this string will appear inside a ”SCRIPT” element, we need to avoid the characters `<`, `>`, and `&` (see item 6).

To cater for readability, we split the expression across lines at places where `s` contains a newline character. So, the resulting string expression will look like:

```
'...\n'+
'...\n'+
 .
 .
'...'
```

This ensures that lines will not be overly long, if

- the lines in string `s` are not too long (which they will not be, if they encode a program that has no overly long lines);
- the string expression does not blow up the representation (this may need further analysis; an alternative would be to control line length of the expression independent of line lengths in string `s`, but make it depend on the representation; however, the content of string will then be less apparent from the expression).

We avoid the characters <, >, and & by putting them in the string through the constants LT, GT, and ET, rather than by putting them directly in a string literal.

Hence, uneval('<pre>\nSource Code\n</pre>') will yield the expression:

```
''+LT+'pre'+GT+'\n'+
'Source Code\n'+
''+LT+'/pre'+GT+''
```

Here is the definition of function uneval(s):

```
function uneval(s) {
  // convert string s to an equivalent JavaScript expression
  // requires: s is a string
  // ensures: result is a string representing a JavaScript expression
  //     such that eval(result) == s
  // constraints: result is split over multiple lines at newlines in s,
  //     and contains no LT, GT, and ET characters
  var result = '\'';
  for (var i = 0; i != s.length; ++i) {
    switch (s.charAt(i)) {
      case LT: result += '\'+LT+\''; break;
      case GT: result += '\'+GT+\''; break;
      case ET: result += '\'+ET+\''; break;
      case '\'': result += '\\\''; break;
      case '\\': result += '\\\\'; break;
      case '\n': result += '\\n\'+\n\''; break;
      default: result += s.charAt(i);
    }
  }
  result += '\'';
  return result;
}
```

14. We now have all script definitions and surrounding HTML code, including the literal occurrence of X (the marker) [7]. This we call *skeleton*(X).

    The skeleton becomes a complete web page, in particular, having a complete script, after substituting a string expression for the (single) occurrence of X.

15. We can construct the self-reproducing web page as follows from the skeleton:

    skeleton(uneval(*skeleton*(X)))

    that is, by substituting the string expression uneval(*skeleton*(X)) for X in the skeleton.

In JavaScript, this can be accomplished by the string expression

```
skeleton.replace(V, uneval(skeleton))
```

where string `skeleton` contains the skeleton, literally. In *Tom's JavaScript Machine* [8], we can use the constructor script [9], prepending the definitions of the four constants and function `uneval(s)` from the skeleton, as program and feed it the skeleton as input. The output string is a web page [10] that will show its own source code.

5 Verification

Let us abbreviate the skeleton with marker `X` by $S.\texttt{X}$. The web page W is then defined by

$$W \quad = \quad S.(\texttt{uneval(}\ S.\texttt{X}\ \texttt{))} \tag{1}$$

When W is loaded, the script is executed and it shows, by construction of W, the result of

$$\texttt{DNA.replace(V, uneval(DNA))}, \tag{2}$$

where, during execution,

$$\texttt{DNA} \quad = \quad S.\texttt{X} \tag{3}$$

by construction of the web page W, since `eval('uneval(` S.X `)')` `==` S.X. That is, we can rewrite the result of (2):

$$\begin{array}{ll} & \texttt{eval ('DNA.replace(V, uneval(DNA))')} \\ = & \quad \{\ (3),\ \text{definition of } \texttt{replace} \text{ and } \texttt{V}\ \} \\ & S.(\texttt{uneval(}\ S.\texttt{X}\ \texttt{))} \\ = & \quad \{\ (1)\ \} \\ & W \end{array}$$

Therefore, the page indeed shows its own source code.

References

[1] MITS Altair 8800. Article in Wikipedia.
URL: en.wikipedia.org/wiki/Altair_8800

[2] JavaScript.
URL: developer.mozilla.org/en/JavaScript

[3] HTML 4.01 Standard.
URL: www.w3.org/TR/html4/

[4] HTML Validator by W3C.
URL: validator.w3.org/

[5] Quine (computing). Article in Wikipedia.
URL: en.wikipedia.org/wiki/Quine_(computing)

[6] Willard Van Orman Quine.
URL: en.wikipedia.org/wiki/Willard_Van_Orman_Quine

[7] Skeleton for Web Page That Shows Its Own Source Code.
URL: www.win.tue.nl/~wstomv/edu/javascript/quine-skeleton.html

[8] Tom's JavaScript Machine.
URL: www.win.tue.nl/~wstomv/edu/javascript/machine.html

[9] Constructor for Web Page That Shows Its Own Source Code.
URL: www.win.tue.nl/~wstomv/edu/javascript/quine-constructor.js

[10] HTML Web Page That Shows Its Own Source Code.
URL: www.win.tue.nl/~wstomv/edu/javascript/quine.html