Olympiads in Informatics, 2010, Vol. 4, 134–141
© 2010 *Institute of Mathematics and Informatics, Vilnius*

An Enticing Environment for Programming

Tom VERHOEFF
Department of Mathematics and Computing Science, Eindhoven University of Technology
Den Dolech 2, 5612 AZ Eindhoven, The Netherlands
e-mail: t.verhoeff@tue.nl

Abstract. While teaching a course on the foundations of informatics to non-CS students, the author wanted to offer a programming challenge without burdening the participants with the numerous details that typically accompany the use of practical programming languages and tools. In particular, there should be no need to install an editor and execution environment (compiler or interpreter). Furthermore, the programming language should be sufficiently simple and clean. However, the author did not want to design a completely new language with tools.

This article presents *Tom's JavaScript Machine* as an attempt at providing a simple and enticing environment for programming, and reports some experiences. *Tom's JavaScript Machine* is freely available on-line and only requires a web browser that supports JavaScript. It includes a simple 3D-variant of Turtle Graphics (for browsers that support the HTML5 canvas element) and an instructive programming challenge with extensive (inter)active hints.

The ideas behind *Tom's JavaScript Machine* can also be applied to create problem-specific environments for informatics contests. However, the current implementation still has some shortcomings that need to be addressed.

Key words: programming tools, JavaScript, self-reproducing programs, study material development.

1. Introduction

In Fall 2009, I taught an Honors Class on the foundations of informatics as a science (Verhoeff, 2009). The participants were selected second-year students from various disciplines, excluding computer science, at Eindhoven University of Technology. We used the book *Algorithmic Adventures* by Hromkovic (2009). This book briefly describes the birth of informatics as a science, focusing on the notion of an algorithm as an object of scientific study. It then presents the exciting things we have learned about algorithms, in particular, the limits of algorithmic computability, our struggles with efficiency and algorithmic complexity, the surprising powers of randomness and approximation, how algorithms changed the world of cryptography to accomplish the unbelievable, and DNA computing and quantum computing as radically different approaches to do computations.

The course was specifically not about programming. Nevertheless, it is useful to do some programming to get a better feel for algorithms. With informal descriptions of algorithms it is too easy to trick oneself into believing that something 'works'. This is especially the case for the following challenge, which I found very instructive when I first encountered it myself.

Challenge: Write a *self-reproducing* program that processes no input and that generates its own listing as output.

If you have never attempted this yourself, then I encourage you to give it a try. It is not easy, requires perseverance, and will teach you about the link between biology and computer science. One needs to have various creative insights to tackle this challenge.

Unfortunately, if you want students without any background in programming to work on a challenge like this, then you need to introduce them to some practical programming environment. Without a well-defined language, it will not be clear whether the problem was really solved. Typically, they would need to install some tools, like a program editor and a compiler or interpreter. Then they would need to learn the syntax and semantics of the programming language, and how to operate the tools. This poses quite a big threshold.

I considered various options. Python (2010) came closest to being minimally obtrusive. However, it still did not match my ideal of zero install and immediate interaction. Then, it struck me that JavaScript run from a web browser could be considered as well.

2. Tom's JavaScript Machine

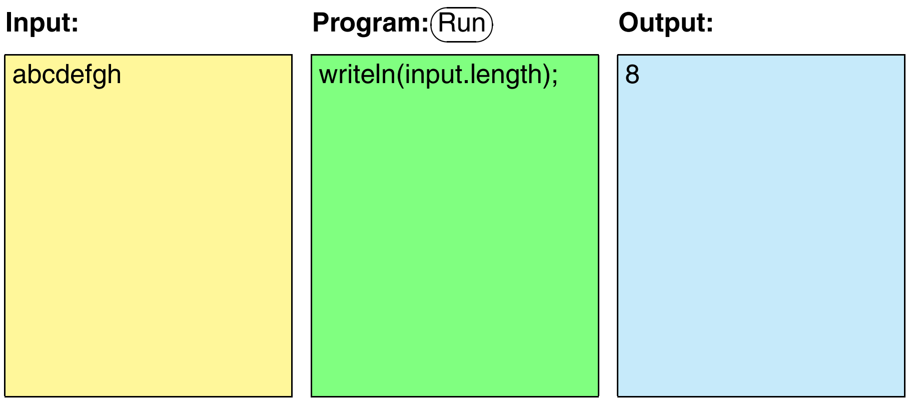
JavaScript is a fairly clean and simple programming language, standardized under the name ECMAScript since the late 1990s (ECMA, 2010). It has features from both functional and object-oriented programming. An in-depth treatment can be found in Flanagan (2006). JavaScript may not have a good name among some groups, but by certain measures it is the most-used programming language of this day.

In no time, I was able to put together a web page with three text areas and a button (see Fig. 1). In one text area, the user enters some input, in another one the user can edit a JavaScript program text, and output is shown in the third text area. When the user clicks the *Run* button, an embedded script (itself also written in JavaScript) evaluates the string `s` in the program area as JavaScript program, through the standard JavaScript function `eval (s)`. I added some minimal facilities for user input and output, because JavaScript by itself does not offer that. The result is *Tom's JavaScript Machine* (Verhoeff, 2009b).

Nice things about *Tom's JavaScript Machine* are that

- it is *zero install*, providing that you have a computer with a web browser supporting JavaScript (version 1.5 or higher; this is available in all modern browsers);
- it offers *immediate interaction*: just type in your JavaScript program text, some input, and click *Run*. Because input is not 'consumed', you do not need to retype it when you want to run your program again.

The following program, which adds a sequence of input numbers, illustrates the facilities for input and output.



```
abcdefgh
```

```
writeln(input.length);
```

```
8
```

Fig. 1. The main user interface of *Tom's JavaScript Machine* consists of three text areas and a *Run* button

```
var sum = 0;
while (moreInputs()) {
  sum = sum + readNum();
}
writeln(sum);
```

Once the initial version was created, some further wishes naturally arose and were easily added, including the following.

- A summary of JavaScript basics.
- Example programs that can be loaded into the machine with one click.
- A user-selectable separator to split input (default: a space).
- A *Challenge* button that clears the input area, executes the program, and compares the output to the program text.

The result is a surprisingly usable programming environment (which I even use myself in some situations). Section 4 discusses some limitations.

3. Facilities for Developing Study Material

While developing a series of hints for the challenge of writing a self-reproducing program, it occurred to me that some further facilities would be useful for teachers. Writing study material for programming courses is inherently a cumbersome task. On one hand, there is the text to be written, e.g., using LaTeX. On the other hand, there are programs and program fragments to be included in the text, possibly with some input and corresponding output. It requires good discipline and preferably some good tools to maintain consistency of all the material, while the text, programs, and inputs evolve. In the traditional approach, whenever a program or input changes, the program must be run again to produce up-to-date output, and all of this must be incorporated (possibly via inclusion) in the text.

In the case of JavaScript programs, it is convenient to write the study material in HTML with embedded scripts, in JavaScript. I extended *Tom's JavaScript Machine* with special facilities for writing study material:

`_parseURI()` to open the web page with the machine and initialize its input area, program area, and output area, and some other parameters with values taken from its URI (web address) in the form

```
.../machine.html?_program=...;_input=...
```

`_machine_link()` to generate an embedded hyperlink with given values for various machine parameters, i.e., in the preceding form;

`_output_of()` to return as string the output of a program given as string, (optionally, input and separator can be passed as well);

`_inject()` to inject a given text with given background color (yellow for input, green for program, and blue for output).

For instance, the following piece of HTML code generates a 'program box' with green background showing the program `writeln(1+1);writeln(1+1);` , an 'output box' with blue background showing the output `22` , and below it a link to the machine for loading this program.

```
<script type="text/javascript">
var _prog = "writeln(1+1);\n";
_inject(_prog, 'programbox');
_inject(_output_of(_prog), 'outputbox');
_machine_link('Load in the machine', _prog);
</script>
```

For more details, see the *About* link in *Tom's JavaScript Machine*. Using these facilities, it is easy to write study material that incorporates programs with input and corresponding output. The hints for the challenge also involve programs that take programs as input and/or that generate programs as output. All of this is neatly handled by these facilities.

4. Experiences and Further Wishes

Tom's JavaScript Machine was used by six participants of the Honors Class to try their hand at the challenge of writing a self-reproducing program. Only one of the students had some prior experience with C, and another with PHP. All of them were able to use the machine immediately to experiment and start on the challenge. And all of them were able to solve the challenge up to a certain level, guided by the hints.

It should, however, be noted that the JavaScript programming language has its own quirks that do get in the way. For the challenge, this turned out to be in the area of strings.

In particular, traversing the characters of a string and constructing expressions to yield specific strings are somewhat awkward. This distracts from the more abstract aspects of the challenge. It is clear that the students found this annoying and that it reduced their interest and limited their progress.

Nevertheless, I consider *Tom's JavaScript Machine* a success, because it demonstrates a new way to offer a low-threshold programming environment and a new way to develop and deliver accompanying study material. In fact, for some computational tasks, I now use the machine myself.

There are some obvious shortcomings to the current implementation:

- the programming language is JavaScript (fully, and only); although JavaScript is nicer than often believed, it is not 'beginners proof';
- the edit facilities in the text areas for input and program are rather limited; in particular, there is no line numbering, no syntax highlighting, and no code completion;
- there is no facility to save and load input and program texts (though the available mechanism to clone the machine in its current state mitigates this somewhat);
- the feedback on syntax and runtime errors is limited and browser dependent;
- there is no facility for interleaved input and output, other than using a standard JavaScript function like `prompt(s)`.

For specific problem domains, it is possible to develop specialized versions of the machine. I created an experimental version of the machine for *3D turtle graphics* (Verhoeff, 2010). The implementation is based on the HTML5 canvas element. Note that HTML5 is not yet an official W3C standard, and that not all major browsers support its current definition. Fig. 2 shows a 3D turtle graphics program and its output on input `5 25 2`, which is a pentagram tilted over 60°. The turtle looks like an aircraft.

```
var t = new TurtleGraphics.Turtle();
 var N = readNum(); // number of corners
 var k = readNum(); // step size
t.Roll(-60);
for var i = 0; i != N; ++i) {
  t.Move(5);
  t.Turn(k * 360 / N);
}
t.DrawTurtle('blue');
```

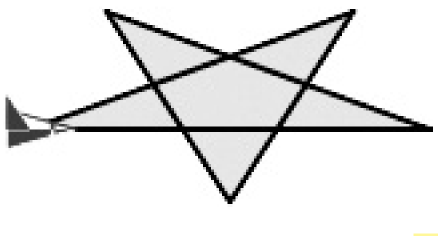


Fig. 2. 3D turtle graphics program and its output for input `5 25 2`.

Similarly, one can imagine having specialized versions of the machine to deal with user interface design, where the programmer gets access to some input fields and buttons. Or a specialized version for manipulating the *Document Object Model* (DOM) that underlies the processing of web pages in browsers. The following program 'hacks' the title of the machine's web page:

```
document.title = "This page was hacked :-)";
```

A version specialized for exploring *numerical algorithms* also comes to mind. Note that JavaScript supports the double-precision 64-bit format conforming to IEEE Standard 754 (IEEE, 1985). Thus, the examples from Horvath and Verhoeff (2003) can be tried in *Tom's JavaScript Machine*. A merger with the floating-point calculators of Vickery (2010) would be interesting in this context.

Finally, in the same vein as the challenge of writing a self-reproducing program, it is imaginable that problem-specific specializations of (an environment like) *Tom's JavaScript Machine* are offered in an informatics contest.

5. Discussion

It should be noted that *Tom's JavaScript Machine* has not been used extensively. My initial motivation for developing it was solely to provide, to non-CS students, an easy environment for the challenge of writing a self-reproducing program. The Honors Class *Algorithmic Adventures* involved no other practical programming, though I am tempted to change that next year. The students learned about programming by doing it on one particular problem. I do not expect that this will work for everyone.

The main reason for presenting it here is that *Tom's JavaScript Machine* offers an environment for programming that differs from more traditional environments. In particular, it is immediately available, it is very easy to use especially with small programs, and it is easy for teachers to develop study material.

As noted in Section 4, the current implementation does have some shortcomings, but the directness of the environment compensates for that. I hope that others will pick up these ideas, develop them further, and investigate how such environments can be put to good use in teaching.

6. Conclusion

Tom's JavaScript Machine offers an enticing environment for programming, with a low threshold. It is zero install, only requiring a modern web browser. There is even a version that you can download, so that you can use it locally without internet access. The interface is very simple, enabling users to start programming immediately.

Additional features of the machine make it easy to develop, in HTML, educational material that incorporates programs with sample input and output, and that the user can load into the machine with a single click. These programs are embedded in the study material and they are executed as the page is loaded. Program errors are reported immediately. Therefore, the consistency between text, programs, input, and output is easy to maintain. It is also straightforward to feed the output of one program as input into another program, or to use the output of a program as a new program to process some input. This is illustrated in the 40 pages with hints for the challenge of writing a self-reproducing program.

An experimental version of the machine offers 3D turtle graphics based on the HTML5 canvas element. In a similar vein, other special versions of the machine can be constructed. For instance, one can offer a programming environment in the area of user interface design, involving various input fields and buttons, or an environment that involves the internal structure of web pages, based on the Document Object Model (DOM), etc. The use of environments like *Tom's JavaScript Machine* in informatics contests could be interesting as well, and needs further development and investigation.

There are some obvious shortcomings to the current implementation, but they are not show stoppers. The current version demonstrates that this approach is promising, and I hope that it will be explored by others.

References

ECMA International. Industry association dedicated to the standardization of information and communication systems, including ECMAScript.
URL: `www.ecma-international.org` (accessed April 2010).

Flanagan, D. (2006). *JavaScript: The Definitive Guide*. Fifth Edition. O'Reilly.

Horvath, G. and T. Verhoeff (2003). Numerical difficulties in pre-university informatics education and competitions. *Informatics in Education*, 2(1), 21–38.

Hromkovic, J. (2009). *Algorithmic Adventures: From Knowledge to Magic*. Springer.

IEEE (1985). *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*.

Python Programming Language.
URL: `www.python.org` (accessed April 2010).

Verhoeff, T. *Honors Class Informatics*. Course web page 2009.
URL: `www.win.tue.nl/~wstomv/edu/hci` (accessed April 2010)

Verhoeff, T. (2009). *Tom's JavaScript Machine*.
URL: `www.win.tue.nl/~wstomv/edu/javascript` (accessed April 2010)

Verhoeff, T. (2010). 3D turtle geometry: Artwork, theory, program equivalence and symmetry. *J. of Arts and Technology*, 3(2/3), 288–319.

Vickery, C. *Interactive IEEE-754 Floating-Point Calculators and Reference Material*.
URL: `www.purl.oclc.org/vickery/IEEE-754/` (accessed April 2010).

T. Verhoeff is an assistant professor in computer science at Eindhoven University of Technology, where he works in the Ggroup Software Engineering & Technology. His research interests are support tools for verified software development and model driven engineering. He received the IOI Distinguished Service Award at IOI 2007 in Zagreb, Croatia, in particular for his role in setting up and maintaining a web archive of IOI-related material and facilities for communication in the IOI community, and in establishing, developing, chairing, and contributing to the IOI Scientific Committee from 1999 until 2007.