Beatnik Encryption

Get hip to the scene by encrypting any string as an avant-garde poem.

Nathan Karasch

Gregory Steenhagen

12 November 2016

Table of Contents

Overview

Beatnik Encryption is a JavaScript program that encrypts any string as an avant-garde poem. The “poem,” in reality, is a Beatnik program that prints out the given string. In theory, then, to encrypt any file, you just need to base64 encode it before passing it through the encrypter. To decrypt the file, you need to run the program in a Beatnik interpreter, base64 decode the resulting string, and presto! -- you're back to the original message! We used a JavaScript interpreter written by Arjeim Icallun and hosted on the Carnegie Mellon Computer Club website.[1]

What is a Beatnik program?

The Beatnik programming language, written by Cliff L. Biffle (aka Cliffle) in 2001, is a syntax-free stack language that uses the Scrabble score of the tokens as op codes and parameter values. According to Cliffle’s website[2], “A Beatnik program consists of any sequence of English words, separated by any sort of punctuation from spaces to hyphens to blank pages. Thus, “Hello, aunts! Swim around brains!” is a valid Beatnik program, despite not making much sense.”

But… why?

According to Merriam-Webster, the definition of *encrypt* is “to change (information) from one form to another especially to hide its meaning.”[3] Encryption plays a big role in computer science, and it is ubiquitous to technology today. While we make no claims about the overall security of hiding a message in the code of an esoteric programming language, we certainly feel it qualifies as a form of encryption (albeit weak). It also satisfied our desire to make something both practical and humorous.

[1] http://www.club.cc.cmu.edu/~rjmccall/beatnik.js
[2] http://cliffle.com/esoterica/beatnik.html

[3] http://www.merriam-webster.com/dictionary/encrypt

Programs writing programs

One of the things that initially intrigued us about the project was the prospect of writing a program that writes a program. This, we felt, was the main "new and complex" element that was being explored. Solving the problem required us to get an understanding of how stack languages work, and that was also new territory, though not terribly complex. We also wanted to make the poems *feel* like poems instead of just looking like long strings of random words chained together. To that effect, we put measures in place to sprinkle whitespace and punctuation throughout based on distribution percentages.

What words are we using to generate the poems?

When the page is loaded, we make a call to RandomText.me's gibberish API[4], which gives us back a string full of nonsense. We parse the words out of the nonsense, give them Scrabble scores, and attempt to add them to our own database of word/score pairs. If the word is not already in the database, it gets added. You can actually see in the browser console how many words are added each time the page is loaded. They're added in four batches, so there are four messages in the console that read, "Learned # new words!" If this project is ever extended in the future, it would be a good idea to explore some other word-generation APIs, since RandomText.me's API has a limited selection. The "learning" growth starts to level out between 3000 and 4000 words.

[4] http://www.randomtext.me/api/gibberish/p-5/100

New and Complex Material

Beatnik Encryption is a program that writes a program. It writes a piece of code that will successfully run in a Beatnik interpreter to output the desired string. The main API for this is the **`beatnikify(str)`** function, which takes a string and returns the encrypted string. Since Beatnik is a stack language, there are three main operations that we need. (There are other stack operations, but we don't need them.)

1. **`push`** - Pushes an integer value onto the top of the stack.
2. **`addTopTwo`** - Pops the top two values, adds them, and then pushes the result back onto the stack.
3. **`popAndPrintChar`** - Pops the top value from the stack and prints the ASCII character corresponding to the integer value.

The pseudo-code for the algorithm could be written as follows:

```
encryptedString = “”;
// Incrementing backwards through the string as we push to the stack
for (i = str.length-1 to 0) {
      // This doesn’t actually perform the operation, it simply writes
      // the beatnik code that will perform the operation.
      encryptedString += push(str[i].asciiCode);
}
// Now when we pop them, they will be in the correct order
for (i = 0 to str.length-1) {
      // Again, just writing the beatnik code to perform the operation
      encryptedString += popAndPrintChar();
}
return encryptedString;
```

In the pseudo-code above, the **`popAndPrintChar()`** operation returns one word corresponding to the op code, and the **`push()`** operation returns two words: one for the op code, and one for the parameter value. Since the op code for **`popAndPrintChar`** is 9, it returns a word with a Scrabble score of 9, such as “unicorn.” For the **`push`** operation, let's say it was pushing the letter “H”. The letter “H” has an ASCII code of 72. Since **`push`** has op code 5, it returns a word with a score of 5 and a word with a score of 72.

However, herein lies a setback. There is no word with a Scrabble score of 72!

We can remedy that by splitting up the value of the parameter into parts small enough that each part can have a valid word. Let's pick some values:

```
a  + b  + c  + d + e  = 72
12 + 15 + 23 + 9 + 13 = 72

a = 12,   b = 15,   c = 23,   d = 9,   e = 13
```

Since all of these values need to be added together, we need more Beatnik code written to do just that. This is where the **`addTopTwo`** command comes into play, and now we can tie everything together. What follows is an example of the Beatnik code needed to print the letter "H":

Operation	**Beatnik Code**
`push(12)`	dear industrious
`push(15)`	cat dubiously
`addTopTwo()`	neutral
`push(23)`	oh qualitative
`addTopTwo()`	rash
`push(9)`	goat music
`addTopTwo()`	thus
`push(13)`	sang intolerable
`addTopTwo()`	austere
`popAndPrintChar()`	unicorn

Considering that whitespace and punctuation are ignored, this is the resulting silly poem that would print the letter "H" when run in the Beatnik interpreter:

Dear industrious cat, dubiously neutral:
Oh qualitative rash! Goat music thus sang... intolerable!

-- Austere Unicorn

That is how Beatnik Encryption works. Notice that, in this instance, one character was expanded into over 100 characters! This demonstrates the fact that, while humorous, Beatnik Encryption isn't terribly well-suited for file encryption, as it would increase the size, in this instance, by two orders of magnitude.

Working at Higher Levels of Bloom's Taxonomy

Creation

One of the complex components in the program is the percentage distribution object, simply titled **`distribution`**. The distribution object allows you to get random values from a set based on their distribution (over 100).

```
var distribution = {
      percentages: {},
      defaultValue: "",
      boundaryValues: {},
      initialized: false,
      get: function () {
      if (!this.initialized) {
            var maxBoundaryValue = 0;
            for (var pKey in this.percentages) {
                  if (this.percentages.hasOwnProperty(pKey)) {
                  maxBoundaryValue += parseInt(pKey);
                  this.boundaryValues[maxBoundaryValue] =
                        this.percentages[pKey];
                  }
            }
            if (maxBoundaryValue > 100) {
                  throw new Error("Cannot have more than 100% in the
                        map of percentages!");
            }
            this.initialized = true;
      }
      var rand = Math.random() * 100;
      for (var bKey in this.boundaryValues) {
            if (this.boundaryValues.hasOwnProperty(bKey)
                  && rand < parseInt(bKey)) {
                  return this.boundaryValues[bKey];
            }
      }
      return this.defaultValue;
      }
};
```

The **get()** method is bound to the **punctuation** and **whitespace** objects. Each key in the **percentages** map is the int percentage, with the value being the element with that distribution. The keys do not add up to 100, however. That's because any remaining percentage after subtracting all keys from 100 is attributed to the **defaultValue**.

```
// Punctuation to be distributed at random throughout the encrypted
text.
var punctuation = {
      percentages: {
            15: ",",
            10: ".",
            6: "!",
            7: "?",
            1: " --",
            3: "..."
      },
      defaultValue: "",
      boundaryValues: {},
      initialized: false
};
// Returns, at random, either a punctuation mark or an empty string.
var getPunctuation = distribution.get.bind(punctuation);

// Whitespace to be distributed at random throughout the encrypted
text.
var whitespace = {
      percentages: {
            25: "<br>",
            6: "<br><br>",
            5: "     ",
            4: "<br>   ",
            3: "<br>  ",
            2: "<br>        ",
            1: " #"
      },
      defaultValue: " ",
      boundaryValues: {},
      initialized: false
};
// Returns, at random, either a space, several spaces, a newline,
// or several newlines.
var getWhitespace = distribution.get.bind(whitespace);
```

Analysis

The code is written in a modular way that helps distinguish between its various parts. At a high level, it can be looked at in the **MVC** (Model, View, Controller) design style:

Model

We have a MySQL database with one table that stores all the word/score pairs. The interface between the model and the controller is database.php, which has two different actions: **`add_words`** and **`get_words`**. The former is used to attempt to add words to the database, and the latter is used to retrieve all the words from the database. I say "attempt to add" because the "word" column has a UNIQUE constraint on it. Any words that database.php tries to add that are already in the database don't get added a second time. The UNIQUE constraint was added after analysis of the server-side code found room for optimization. Instead of having to retrieve all the words from the database and check to make sure a given word wasn't already there, we can just let the MySQL optimizations handle that for us, resulting in a much faster server response time to requests made by the client.

View

As in most web apps, the view is contained in the HTML and CSS. Initially the javascript was split between the js file and an inline script at the bottom of the html page. A later refactor moved all the code into beatnik_encryption.js to keep the Controller code out of the View.

Controller

The beatnik_encryption.js file holds all the controlling logic for the encryption, and the interpreter.js file (which we didn't write), holds all the controlling logic for the decryption. The jQuery framework was selected for ease of use in AJAX calls and DOM modification. The Controller interfaced with both the View and the Model by interacting with the DOM and reading/writing from the database, respectively.

Evaluation

The main conclusion we've drawn upon evaluation of the final product is that it works great for short messages or files (<1kb) and terrible for anything else. As seen in the above example, the encryption increased the size of the message by two orders of magnitude. That means if we were to upload a 1 MB file, it could grow to upwards of 100 MB! And that's not even worst-case scenario. Allow me to explain:

As discussed earlier, whenever a character's value needs to be pushed onto the stack, it most likely needs to be split into parts that add up to its value. In the case of "H," 72 was split into 12, 15, 23, 9, and 13. The way the algorithm works, the divisions are made by randomly selecting a number between 1 and 25 to subtract from the total value. This is done in a loop until the total value is <=25, at which time the remaining total value is chosen as the last part. Below is the code for the function that accomplishes this.

```
function splitCharValue(value) {
	if (value < 1) {
		throw new Error("Value passed to splitCharValue() cannot
			be less than 1!");
	}
	var values = [];
	while (value > scoreWordMap.maxScore) {
		var subtracted = getRandomInt(1, scoreWordMap.maxScore +
1);
		values.push(subtracted);
		value -= subtracted;
	}
	if (value > 0) {
		values.push(value);
	}
	return values;
}
```

As you can see, this leaves it open for incredibile inefficiency for the sake of poetic freedom (picking random word scores). We found this inefficiency acceptable, as the program should never be used for file encryption anyways, and part of the goal of the project

was to make the generated programs take on the likeness of abstract poems. For that, we needed the freedom to choose from lots of different words.

Let's look at how much one character can be inflated in a worst-case scenario:

The ASCII character with the largest decimal code is the tilde (~), which has a code of 126. Suppose the "random" numbers chosen for subtraction in the **`splitCharValue()`** method are all ones, except for the last value of "25" which gets chosen once the while loop is exited. That means we need to perform 101 **`push()`** operations, 101-1 **`addTopTwo()`** operations, and 1 **`popAndPrintChar()`** operation. The operations and their corresponding Scrabble scores (assuming max letter tiles on the 25-point word and adding one space between words) are as follows:

```
          push(25) * 1        =     5+1+25+1          =       32
           push(1) * 100      =     (5+1+1+1)*100     =      800
     addTopTwo() * 100        =     (7+1)*100         =      800
popAndPrintChar() * 1         =     9+1               =       10
                                                  TOTAL:  1642
```

So in this instance, one character was inflated into 1642 characters required to print it! Granted, this is a worst-case scenario, but it still leaves the very REAL possibility that message/file inflation can be over 3 orders of magnitude! Not only is the file size an issue, but the time taken for encryption/decryption can cause the browser to believe the script is unresponsive, leading to a possible crash. Don't use Beatnik Encryption for encrypting files!

Resources

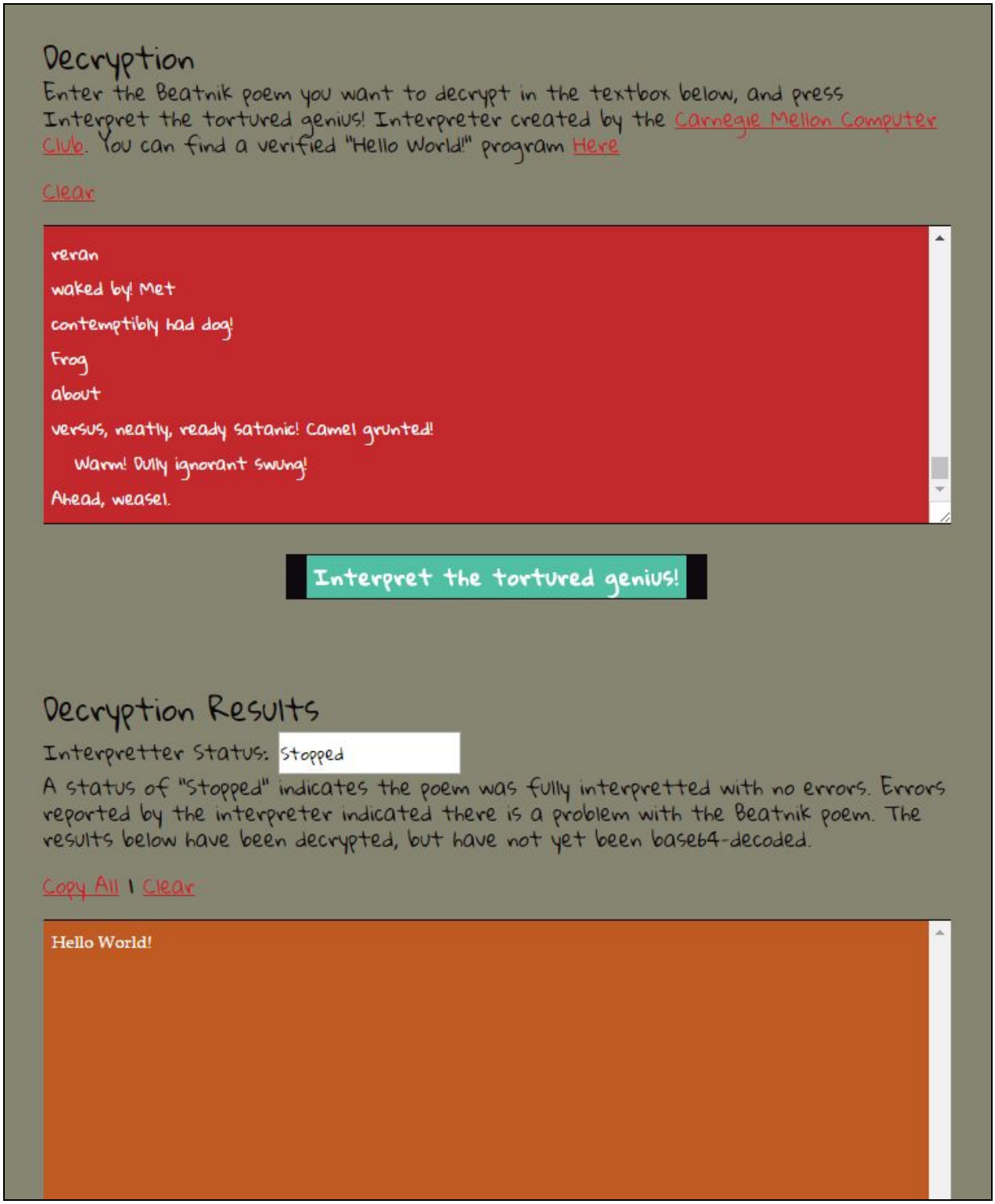
GitHub Repository

https://github.com/KrashLeviathan/beatnik_encryption

Live Website

http://beatnikencryption.krashdev.com/

Screenshots


Decryption
Enter the Beatnik poem you want to decrypt in the textbox below, and press Interpret the tortured genius! Interpreter created by the Carnegie Mellon Computer Club. You can find a verified "Hello World!" program Here
Clear
reran
waked by! Met
contemptibly had dog!
Frog
about
versus, neatly, ready satanic! Camel grunted!
Warm! Dully ignorant swung!
Ahead, weasel.
Interpret the tortured genius!
Decryption Results
Interpretter Status: Stopped
A status of "Stopped" indicates the poem was fully interpretted with no errors. Errors reported by the interpreter indicated there is a problem with the Beatnik poem. The results below have been decrypted, but have not yet been base64-decoded.
Copy All | Clear
Hello World!