# Esoteric Programming Languages

## An introduction to Brainfuck, INTERCAL, Befunge, Malbolge, and Shakespeare

Sebastian Morr
Braunschweig University of Technology
sebastian@morr.cc

## ABSTRACT

There is a class of programming languages that are not actually designed to be used for programming. These so-called "esoteric" programming languages have other purposes: To entertain, to be beautiful, or to make a point. This paper describes and contrasts five influential, stereotypical, and widely different esoteric programming languages: The minimal Brainfuck, the weird INTERCAL, the multi-dimensional Befunge, the hard Malbolge, and the poetic Shakespeare.

## 1. INTRODUCTION

While programming languages are usually designed to be used productively and to be helpful in real-world applications, esoteric programming languages have other goals: They can represent proof-of-concepts, demonstrating how minimal a language syntax can get, while still maintaining universality. They might help to prove mathematical theorems or provide boundaries in complexity analyses. The design of esoteric programming languages can be seen an artistic process, and the resulting languages can be expressions of human intellect, wit, and aesthetic taste. Or they might be created as a kind of competitive sport, a challenge for the language's designer or for its users. Finally, there are joke languages designed to enjoy the authors themselves, the users, or even the readers of the specification. All of these motivations and the languages' unique, unusual properties make it worthwhile to study esoteric programming languages in more detail.

The word "esoteric" comes from the ancient Greek *esoterikos*, meaning "belonging to an inner circle", and originally referred to Pythagoras' secret teachings [1]. It evolved to mean "mystic", or "having to do with highly theoretical concepts without obvious practical application". The first usage of the term "esoteric" in the context of "weird programming languages" was probably on a website called *Esoteric Topics in Computer Programming*, published by Chris Pressey, the inventor of Befunge, around 1997 [2].

For this paper, we picked five iconic esoteric programming languages, each of which demonstrates a unique property: **Brainfuck** attempts to have a *minimal* syntax, which consists only of eight different characters. Nevertheless, it can be shown to be Turing-complete. The authors of **INTERCAL** wanted to create a *weird* language, that had as many differences to other languages known at that time as possible. **Befunge** was among the first *two-dimensional* languages, the user can use directional commands to control the program flow. **Malbolge** was designed to be as *hard* as possible to use. It uses multiple encryption mechanisms, and it took two years to write the first program. Finally, **Shakespeare** requires its programs to look like Shakespearean plays, making it a *themed* language. It is not obvious at all that the resulting texts are, in fact, meaningful programs. For each language, we are going describe its origins, history, and today's significance. We will explain how the language works, give a meaningful example and mention some popular implementations and variants.

All five languages are imperative languages. While there are esoteric languages which follow declarative programming paradigms, most of them are quite new and much less popular. It is also interesting to note that esoteric programming languages hardly ever seem to get out of fashion, because of their often unique, weird features. Whereas "real" multipurpose languages are sometimes replaced by more modern ones that add new features or make programming easier in some way, esoteric languages are not under this kind of pressure, because they do not attempt to be usable. Instead, they can often be appreciated like a piece of art, much like an oil painting can be admired for hundreds of years.

## 2. PRELIMINARIES

We first define some terms that are relevant when discussing esoteric programming languages.

### 2.1 Turing machines

In his 1937 paper *On Computable Numbers* [3], Alan Turing specified a simple machine that he used to define what "computing" means. It consists of an infinite one-dimensional tape with cells that can hold symbols, a head, which moves over the tape and is able to read and write symbols, and a finite state machine, which specifies what to do when a specific symbol is read. This simple architecture is so powerful that it can simulate all other computer models, and thus is able to compute every computable sequence. In fact, one can construct a Turing machine that reads a symbolic description of another Turing machine, and some input, and then simulates that other machine. This is what we call a *universal Turing machine*.

### 2.2 Turing completeness

If a system can be used to simulate a universal Turing machine, it is called *Turing-complete*. By definition this means that this system is as powerful as the class of Turing machines, meaning they as well can compute any computable sequence. Turing completeness is an important property of universal programming languages. It is highly probable that all modern general-purpose programming languages are Turing-complete, but not all esoteric programming languages

are, which makes it interesting to look at this property. To be *really* Turing-complete always would require access to an infinite amount of memory, as the systems need to be able to handle arbitrarily large input. Of course, physical machines cannot provide unlimited memory, which is why this restriction is commonly ignored.

## 2.3 Standard programs

When learning or creating a new programming language, one of the first tasks is often to write a program that outputs text, conventionally the string "`Hello, world!`". This phrase was made popular by the first edition of the 1978 book *The C Programming Language* by Brian Kernighan and Dennis Ritchie [4]. Being able to write a *Hello world* program in a language demonstrates that it is possible to compile/interpret a simple program in it and that it can output text.

The second standard program is called *99 Bottles of Beer*, which prints the text of a North American folk song. The song repeats lines like "99 bottles of beer on the wall, 99 bottles of beer. Take one down, pass it around, 98 bottles of beer on the wall..." until there are no more bottles left. The ability to program this demonstrates that the language can handle loops and conditional execution.

Finally, another standard programming task is to write a *quine*, a program which prints out its own source code. Usually, it is considered insufficient to simply open the source file and reading from it; instead, the source code must contain all information to reproduce itself. The term was coined by Douglas Hofstadter [5] in the honor of Willard Van Orman Quine, an American philosopher who studied self-referentiality. Later, the term was applied to self-replicating programs. The first known quine appeared in a 1972 article by Paul Bratley and Jean Millo and was written in a variant of Algol [6].

# 3. BRAINFUCK

The first language we are going to look at, Brainfuck, is a well-known minimalist programming language, aiming for a small language syntax and small compilers. Its programs consist of only eight different characters, nevertheless, it was proven to be Turing-complete.

## 3.1 Origin

Brainfuck was designed by Urban Müller in 1993. At that time, he was a Swiss physics student who in 1992 took over a small online archive for Amiga software. The archive grew more popular, and was soon mirrored around the world. Today, it is the world's largest Amiga archive, known as *Aminet* [7]. We mention this platform because in 1993, Müller uploaded the first Brainfuck compiler to Aminet, in the form of a machine language implementation that compiled to a binary with a size of 296 bytes [8]. The program came with a Readme file, which briefly described the language, and challenged the reader "Who can program anything useful with it? :)". Müller included some already quite elaborate examples as well.

The language's name is a reference to the slang term "brain fuck", which refers to things that are so complicated or unusual that they go beyond the limits of one's comprehension.

As Aminet grew, the compiler became popular among the Amiga community, and in time it was implemented for other platforms. Refer to Section 3.6 for an overview of Brainfuck variants.

## 3.2 Description

A Brainfuck program operates on an infinite linear arrangement of memory cells, often called the *tape*. Each memory cell contains an unsigned byte value (a number between 0 and 255), which is initialized to 0 when the program starts. Additionally, Brainfuck maintains a *head*, which points to one of the memory cells.

Syntax-wise, a Brainfuck program can consist of eight different characters, which have the following semantics:

`>` Move the head to the right.

`<` Move the head to the left.

`+` Increment the current cell's value.

`-` Decrement the current cell's value.

`,` Read an Ascii character from the user, write its value to the current cell.

`.` Output the current cell's value as an Ascii character.

`[` If the current cell contains a 0, skip to the matching closing bracket.

`]` If the current cell does not contain a 0, return to the matching opening bracket.

Other characters in the source code are ignored (which allows for inline documentation, and for embedding Brainfuck in other programs). While in- or decrementing, the cells' values always wrap to stay between 0 and 255.

Using the brackets, the programmer can create loops: The expression

`[`*code*`]`

will execute *code* until the cell currently pointed to is 0 when encountering the closing bracket. For example, the expression

```
[->+<]
```

will add the current cell's value to the next cell: Each time the loop is executed, the current cell is decremented, the head moves to the right, that next cell is incremented, and the head moves left again. This sequence is repeated until the starting cell is 0. Another gadget which we will use in the following example is

```
+[->+]-
```

which moves the head to the right until it points to a cell with the value 255. The `+` operators increment the cell's value before each check, and if it was a 255 before, the value will wrap to a 0, and the loop will terminate. The `-` operators reset the cell to its original value before the cell is left.

## 3.3 Example

The following program reads a sequence of Ascii values from the user, and prints their binary representations. In order to demonstrate Brainfuck's unique aesthetic, the program is first shown in its minified form:

```
1 -[>,[<<[-<]++[->+]->-]<<<<<<<<<<+[-<+++++++++[->+
2 +++++<]>.[-]>+]-]
```

Input: `hello`
Output: 01101000011001010110110001101100011011111

Let us look at it in more detail. The following commented version can be broken down into three sections: Line 1 sets up the basic memory layout, which is restored for each character the user enters. The program uses a "sentinel" cell with the special value 255 to facilitate seeking back to the end of the bit array. Line 10 reads the character, lines 11–18 implement a simple shift register to calculate the binary representation. The remaining lines print the binary digits by calculating the digits' ASCII values (48 for "0" or 49 for "1"). Note that lines 21—26 are equivalent to `++++++++++++++++++++++++++++++++++++++++++++++++`, which would also increment the cells' values by 48, but we wanted to demonstrate a more esoteric (and a more concise) approach here, which is why the actual code increases the value by 6 eight times.

```
1  -               Memory layout:
2                  0 0 0 0 0 0 0 0 (255) 0
3                  \_____ a _____/   b   c
4
5                  a: bits to be calculated
6                  b: sentinel (current cell)
7                  c: cell for user input
8
9  [               In an endless loop:
10   >,            Read a char to c
11   [             As long as it is not 0:
12     <<          Move to the leftmost bit
13     [-<]        As long as the head is on a 1:
14                   Set to 0 and move left
15     +           On the first 0: Set to 1
16     +[->+]-     Go back to the sentinel
17     >-          Decrement our number
18   ]
19   <<<<<<<<<     Seek to the first digit
20   +[-           Until we're back at 255:
21                 Increment the cell by 48 by
22     <++++++++     putting an 8 in the previous
23     [             cell and decrementing our cell
24       -           by 6 that many times
25       >++++++<
26     ]
27     >.          Print its ASCII value
28     [-]         Restore the cell to 0
29     >           And go to the next digit
30   +]-
31 ]
```

## 3.4 Computational Class

Brainfuck was proven to be Turing-complete by Daniel Cristofani [9], who used it to implement a simple universal Turing machine as described by Yurii Rogozhin [10].

Brainfuck is an example of a so-called *Turing tarpit*. This term was coined in 1992 by Alan Perlis, first recipient of the Turing Award, who warned against environments "in which everything is possible but nothing of interest is easy" [11], in reference to geologic asphalt lakes, whose thick consistency slows down movements for everything inside. Turing tarpit languages, like Brainfuck, provide a handful of very general and flexible mechanisms, which can be used to write *any* program, but it is seldom practical to do so, because the languages provide so little abstraction that the programs get very long or complicated.

## 3.5 Implementations

Together with his original machine language implementation, Müller published a C interpreter, which, with a minor modification, can still be used on modern machines. Besides that implementation, there are numerous others in all imaginable languages, some of which achieve very fast runtimes by optimizing the code in numerous ways, including *Awib* [12], a Brainfuck compiler written in Brainfuck itself, which is able to compile to Linux executables, Tcl, Ruby, Go, and C. Fans of the language succeeded in implementing even smaller compilers than the original version, the smallest one currently is an MS-DOS binary with a size of 98 bytes[1] [13].

## 3.6 Variants

The esolangs wiki [14], a large database of esoteric programming topics, and informal successor of Chris Pressey's previously mentioned *Esoteric Topics* site, lists 162 articles in the "Brainfuck derivatives" category and 33 "Brainfuck equivalents", which were all inspired by Müller's original implementation. There are variants which operate on two tapes (**DoubleFuck**), or restrict the cells to binary values, thus making the `+` and `-` operations identical (**Boolfuck**). Some add more operators (like **Brainfork**, which adds a `Y` command for forking the process), others try to reduce the command set even further (**BitChanger** also works on bit cells and defines `}` := `>+`. The original `>` can be emulated with `}<}`). The joke variant **Ook!** behaves exactly like Brainfuck, but its operators are pairs of Orangutan words like "Ook. Ook?" for `>` or "Ook! Ook!" for `-`.

Because there was never a precise language specification, the various implementations can differ in some aspects: Whereas the general idea assumes an infinitely long tape, actual implementations always have some kind of memory limit. The original compiler used a tape of 30,000 cells, with the pointer starting on the leftmost one. Some implementations extend the memory array when the pointer steps out of the allocated range, others crash, others again will wrap around the tape. The size of a cell was one byte in the original implementation, wrapping around to 255 when subtracting from 0, and many implementations follow that design. Others use 16- or 32-bit numbers, or even signed numbers, allowing the cells to have negative values. Another implementation difference is about what happens when a Brainfuck programs wants to read a byte, but there is no more input—for example, because the input was a file which has reached the end-of-file condition. In many applications, it is important to know that there will be no more input. Müller's implementation leaves the current cell unchanged in this case, others set it to 0, others to -1 (this requires cells which are larger than bytes).

Interestingly, Brainfuck had a much earlier predecessor: In 1964, the theoretical computer scientist Corrado Böhm designed the language $\mathcal{P}''$, to describe a specific family of Turing machines [15]. Böhm showed that this language was Turing-complete long before Brainfuck was implemented. Programs in $\mathcal{P}''$ consist of words over the alphabet $\{R, \lambda, (, )\}$. $\mathcal{P}''$ operates on a left-infinite tape, which can contain symbols of an alphabet $\{a_0, a_1, \ldots, a_n\}$. Initially, each memory cell contains $a_0$, the *blank symbol*. The symbols' semantics are as follows:

[1] Fun fact: This sentence is about twice as long as that compiler.

***R*** Move the head to the right.

**$\lambda$** Increment the current symbol, then move the memory pointer to the left.

**(*q*)** Repeat $q$ while the current symbol does not equal the blank symbol.

Each Brainfuck program can be translated to a $\mathcal{P}''$ program using the following equivalents:

$$\texttt{+} \to r = \lambda R$$
$$\texttt{-} \to r' = \overbrace{rrr\dots r}^{n}$$
$$\texttt{>} \to R$$
$$\texttt{<} \to r'\lambda$$
$$\texttt{[} \to ($$
$$\texttt{]} \to )$$

That is, to perform a Brainfuck increment, increment and move left, then move right. To decrement, increment the current symbol $n$ times, until it "wraps around" and comes to halt one symbol before it started. To move left, "decrement" the cell, then perform a $\lambda$ operation.

## 3.7 Significance

Nowadays, Brainfuck is probably the best-known esoteric programming language in the world. It is a common programming exercise to implement a Brainfuck interpreter or compiler in another language. There is a vast number of Brainfuck programs, including an award-winning text adventure [16].

# 4. INTERCAL

This language was created with the main goal to have as few similarities with existing languages as possible. It has weird operators, unusual concepts like a politeness requirement, probabilistic execution, and a `COME FROM` statement.

## 4.1 Origin

INTERCAL is widely regarded as the first esoteric programming language ever created: It was invented in 1972 by Donald R. Woods and James M. Lyon, students at Princeton University. They had just finished their final exam and joked around with a friend about alternative names for punctuation symbols (for example, calling the "`"`" symbol "rabbit-ears"). According to Woods [17], this was the starting point for a complete made-up language that differed in almost all aspects from other languages of that time.

The language's actual name is *Compiler Language With No Pronounceable Acronym*, which made it necessary to abbreviate it as INTERCAL instead. The manual [18] is full of humorous statements. A small selection:

- "Under no circumstances confuse the mesh with the interleave operator, except under confusing circumstances!"
- "Definition of array dimensions will be discussed later in greater detail, since discussing it in less detail would be difficult."
- "*exp* represents any expression (except colloquial and facial expressions)"
- "Precedence of operators is as follows: (The remainder of this page intentionally left blank.)"
- "This footnote intentionally unreferenced."

Originally, INTERCAL programs were written on punch cards and used the EBCDIC character encoding, an 8-bit encoding by IBM that included characters like the cent symbol "`¢`", which is not found in the ASCII standard and had to be replaced with "`$`" in later versions of INTERCAL. In this paper, we describe the modern C-INTERCAL dialect (see section 4.5).

## 4.2 Description

Note that we will not describe INTERCAL's full syntax and semantics here, as it simply has too many features. We will instead focus on the most commonly used statements, which enable us to write a small example program.

INTERCAL has special names for all symbolic characters. For example, the "`@`" symbol is called "whirlpool" and "`%`" is called "double-oh-seven". We will introduce the symbols' names when they are first used.

The only value type in INTERCAL is an unsigned integer. There are 16-bit integers, whose name must begin with a spot (`.`), followed by a number between 1 and 65535, and there are 32-bit integers, which start with a two-spot (`:`). Literals are always 16-bit and start with a mesh (`#`). For example, `#42` is the literal value 42, while `.42` is an unsigned 16-bit integer variable.

There are also integer arrays, whose names start with a tail (`,`) for an array of 16-bit values, and a hybrid (`;`), for 32-bit values, whose details we will not discuss here.

There are two binary operators: The *mingle* operation, denoted by a big money (`$`), takes two 16-bit operands and produces a 32-bit number by alternating the operand's bits. For example, `#7$#0` equals 42, as the bit sequences of 7 (`111`) and 0 (`000`) are interleaved to become `101010`.

The other binary operation is called *select*, denoted by a squiggle (`~`), and uses the second operand as a mask that denotes which bits to select from the first operand. For example, `#255~#42` equals 7, as the mask `101010` is applied to the `11111111`, selecting three of its 1's and placing them next to each other to become `111` again.

There are three unary operators, namely ampersand[2] (`&`), book (`V`), and what (`?`). When applied to a value, they rotate its bits to the right, and apply the bitwise logical AND, OR, or XOR functions on the result and the initial value. Unary operators are placed after the type-denoting character. For example, when applied to 77 (`0000000001001101`), the results are:

`#&77` = `0000000000000100` = 4
`#V77` = `1000000001101111` = 32879
`#?77` = `1000000001101011` = 32875

As mentioned, INTERCAL has no rules for operator precedence, the respective page in the manual is simply blank. To avoid ambiguities, expressions must be grouped using sparks (`'`) or rabbit-ears (`"`). To apply a unary operator to a sparked or rabbit-eared expression, it is placed after the opening symbol. For example, to apply first the OR, and then the AND operation to the number 42, one would write `"&#V42"`

[2]The manual states that this name is already original enough.

An INTERCAL program consists of statements, which must be prefixed with either `DO`, `PLEASE`, or `PLEASE DO`. INTERCAL has a politeness requirement: Between 1/4 and 1/5 of all statements must begin with `PLEASE`. If the ratio is smaller than 1/5, the compiler rejects the source code for being insufficiently polite. If it is higher than 1/4, the program is rejected for being too sleazy.

Statements may additionally be prefixed with a line label, which do not have to be in order. These lines may then be jumped to using "`DO (`*line number*`) NEXT`". There is even a reversed construct: "`COME FROM (`*line number*`)`" will transfer control to the current line after the specified line has been executed. INTERCAL comes with a standard library, which makes some operations easier, however, it occupies "many line labels between 1000 and 1999", so the programmer has to be careful not to use those.

Assignment is done with an angle-worm:

```
DO .1 <- #1337
```

The statement "`READ OUT` *expression*" is used to output a value (numbers are printed using Roman numerals). To read a number, "`WRITE IN` *variable*" is used; the value's individual digits must be spelt out in English (like "`FOUR TWO`" for 42).

To exit a program, the statement `GIVE UP` must be used.

## 4.3 Example

The following programs calculates $2^n$ for some $n$ entered by the user. In line 1, the exponent is read from standard input and stored in the variable `.1`, which is also used as a loop variable. The variable `.4` will later contain the result and is initialized to 1. In line 3, `.2` is set to 1 as well, it will later be used to decrement the loop variable.

Lines 5 and 12 set up a loop: After line 12 is executed, the expression in line 5 is evaluated. The expression selects from `.1` all bits where `.1` itself has a 1. That is, if `.1` contains any 1's at all, `.1~.1` will end with at least one 1. From this value, we select the last bit. So, the whole expression is 1 exactly if `.1` is not zero. As a result, as long as `.1` is not zero yet, a jump from line 12 back to line 5 will occur.

The loop body has two steps: First, lines 7–9 multiply `.4` by two by mingling it with zeroes (line 7), setting up a filter of the form `...10101011` that selects all original bits, plus an additional zero at the end (line 8), and applying it to the temporary variable `:1` (line 9). Second, it decrements the loop variable `.1` by calling the routine at `(1010)` in the standard library, which subtracts `.2` from `.1` and stores the result in `.3`. After the loop has ended, print the result and exit (lines 14 and 15).

```
1  PLEASE WRITE IN .1
2  DO .4 <- #1
3  DO .2 <- #1
4
5  DO COME FROM '.1~.1'~#1
6
7  DO :1 <- .4$#0
8  DO :2 <- #65535$#1
9  DO .4 <- :1~:2
10
11 PLEASE DO (1010) NEXT
12 (1) DO .1 <- .3
13
14 DO READ OUT .4
15 PLEASE GIVE UP
```

Remember that the output is in roman numerals. Here, $2^{10}$ is calculated to be 1024:

```
Input: ONE ZERO
Output: MXXIV
```

## 4.4 Implementations

The original implementation by Woods and Lyon, which translated INTERCAL to SNOBOL, a pattern-centered language developed in 1962, seems to have been lost over the years. In 1990, the American software developer Eric Raymond revived the language by releasing **C-INTERCAL** [19], an INTERCAL compiler written in the C programming language, which is by far the best known implementation today. Raymond enhanced the instruction manual [20] significantly and added some new features (see next section). C-INTERCAL does some internal optimization, like pre-computing the result of operations on constant values. Interestingly, C-INTERCAL allows for linking with Befunge programs.

## 4.5 Variants

Compared to the original INTERCAL specification, C-INTERCAL introduced some additional statements, like the `COME FROM` statement. It also added a mode called **TriINTERCAL**, which does not operate on binary values, but on ternary ones (with a base of 3). In fact, the implementation supports all number bases up to 7 (base 8 is considered "too useful"). C-INTERCAL also integrated some features of other INTERCAL variants: **Threaded INTERCAL** spawns multiple threads when there is more than one `COME FROM` statement for one line. **Backtracking INTERCAL** introduces the `MAYBE` label, that can be added to each statement. When encountered, it saves the complete program state, so if the respective choice turns out to be bad, the state can be restored and another decision can be made. This mechanism allows a very elegant formulation of backtracking algorithms.

## 4.6 Computational Class

As it is rather easy to write an interpreter for $\mathcal{P}''$ (see section 3.6) in INTERCAL using the standard library [21], the language is definitely Turing-complete.

## 4.7 Significance

Until today, there's quite a large community around INTERCAL. Raymond still actively maintains his C-INTERCAL compiler, together with co-maintainer Alex Smith.

When the company Woods worked for was acquired by Google in 2007, some Google employees wrote an INTERCAL style guide [22], in analogy to the style guides for "proper" languages like C++ and Python. While it is written quite humorously, it does contain helpful recommendations and insightful comments that would make realizing a large software project in INTERCAL easier.

In 2003, Woods was contacted by Donald Knuth, who wrote him he had "just spent a week writing an INTERCAL program" [23] and discovered "a really cool hack" in the standard library's division routine [17]. In 2010, Knuth also contacted Raymond to report some bugs in C-INTERCAL [24].

# 5. BEFUNGE

Befunge is a self-modifying, stack-based, multi-dimensional language. It is similar to Brainfuck, but it does not operate

on a tape, but instead on a two-dimensional matrix. The programs have a two-dimensional format, as well.

## 5.1 Origin

Befunge was invented in 1993 by Chris Pressey, and is believed to be the first two-dimensional esoteric programming language. Pressey's motivation was to create a language which was as difficult to compile as possible. Not only the multi-dimensionality complicate the compilation process, but also Befunge's ability to modify its own source code. Befunge's stack-based computational model was inspired by Forth, a programming language relying heavily on a data stack and on reverse Polish notation. According to Pressey [25], the name "Befunge" was originally a mistyping of "before"., typed by Curtis Coleman in a BBS chat system at 4AM.

## 5.2 Description

Programs in Befunge are a grid of Ascii characters with a width of 80 and a height of 25 (a common size of old terminal emulators like the VT100), called the *playfield*. Like in Brainfuck, there is an instruction pointer, which starts at the top-left entry, moving right. When the pointer leaves the playfield, it wraps around to the other side. Encountered instructions are executed. The pointer keeps moving in the same direction, if it is not changed.

Befunge also maintains a stack of values, which can be manipulated in various ways. Befunge has a large instruction set of 36 different commands, but in contrast to other languages, they are really easy to understand:

`+ - * / %` Pop two values, add/subtract/multiply/integer-divide/modulo them, and push the result back onto the stack.

`!` Pop a value, if it is zero, push 1, otherwise, push 0.

`‘` Pop two values, if the first one is smaller, push 1, otherwise 0.

`> < ^ v` Move the pointer to the right/left/up/down.

`?` Move the pointer in a random direction.

`_` Pop a value if it is 0, move right, otherwise left.

`|` Pop a value it it is 0, move down, otherwise up.

`"` Turn on stringmode: Until the next "`"`", the values of the Ascii characters will be pushed.

`:` Duplicate the top stack value.

`\` Swap the top two stack values.

`$` Discard the top stack element.

`.` Print the value of the topmost element and discard it.

`,` Print the Ascii character of the topmost element and discard it.

`#` Bridge: Jump over the next command.

`g` Pop $y$ and $x$ and push the Ascii value of the character at that position in the playfield.

`p` Pop $y$, $x$ and $v$, and write the Ascii value $v$ to the position $(x, y)$ in the playfield.

`&` Read an integer from standard input and push it onto the stack.

`~` Read a character from standard input and push it onto the stack

`@` End the program.

`0-9` Push the respective number onto the stack.

The most important instructions are the direction-changing instructions. For example, this program represents an endless loop:

```
1 >v
2 ^<
```

Note that as calculations are done on the stack, they must be written down in postfix notation. Values larger than 9 must be calculated beforehand, as well, because in the program, only single digits can be specified. For example, this program puts $42 \cdot 23$ on the stack:

```
1 67*83*1-*
```

## 5.3 Example

The following program reads characters from the standard input, and performs the ROT13 operation on them, which moves letters by 13 places forward in the alphabet, wrapping from Z back to A. Lowercase and uppercase letters are moved separately. Applying ROT13 to a text twice will restore the original. This operation is sometimes used in forums to hide spoilers from plain view, while making it trivial to en- and decode it.

```
1 >~:"‘"‘!v
2 ,v        _:"z"‘v
3 ^               _:"m"‘v
4 ^     -4-9<
5           |             <
6 ^     +4+9<
7 ^               _:"M"‘^
8 ^          _:"Z"‘^
9  >:"@"‘!^
```

Input: Hello, world!
Output: Uryyb, jbeyq!

To make the paths inside of the program more visible, we show another version here, which has additional (redundant) direction arrows:

```
1 >~:"‘"‘!v
  ^         v
2 ,v<<<<<<_:"z"‘v
  ^v                v
3 ^v#<<<<<<<<<<_:"m"‘v
  ^v                v
4 ^v#<<-4-9<        v
5 ^v         |<<<<<<<<<<<
6 ^v#<<+4+9<          ^
  ^v                  ^
7 ^v#<<<<<<<<<<_:"M"‘^
  ^v                ^
8 ^v#<<<<<_:"Z"‘^
  ^v        ^
9  >:"@"‘!^
```

Here is a general idea of how this program works: For each character, it is first determined whether it is in the range a-z or A-Z, and after that, if it is between a-m or A-M, respectively. If it is, its value is increased by 13, otherwise, 13 is subtracted. If the character is no letter at all, it is not changed.

The execution starts with the first character in the first line. When first encountering the `>`, it is ignored, as the instruction pointer is moving to the right anyway. The `~` instruction reads in a character and pushes its ASCII value on the stack. This value is then duplicated (`:`). The first `"` instruction turns the stringmode on, thus the following character, `‘`, is pushed onto the stack as the value 96 (this is the character directly in front of "a"). The second `"` closes the stringmode again. After that, the `‘` compares the two topmost stack values, removes them, and then pushes a 1 if the top one was smaller, and a 0 otherwise. Then, the `!` inverts this "truth value".

The last instruction on that line, the `v`, moves the instruction pointer to the first conditional statement in line 2. If there is a 0 on the stack, which is the case if the original letter came after `‘` in the ASCII table, the pointer moves right, otherwise left. If it moves right, the original value is duplicated again and now compared to `z` in exactly the same way as in line 1. This time, the truth value is not inverted, so that we move left if our letter was larger than `z`, and right otherwise. If moving right, we know that we have a lowercase letter.

In line 3, it is finally compared to `m`, and control is transferred to the small C-shaped construct in the middle of the program, coming from the right in line 5.

Depending on whether our letter is in the range `a`–`m` or in the range `n`–`z`, the pointer moves down or up at the `|` instruction. If it ends up in the lower half, 13 is added, by pushing first a 9 on the stack, then adding it, then doing the same with a 4. Otherwise, it is subtracted. Finally, control is transferred to the first *column* (the detailed version uses bridges (`#`) to jump over the `v` lane), where the resulting letter is finally printed with the `,` instruction. After that, the first `>` in line 1 closes the loop, and the process is started again.

Note that when we find that our letter is smaller than `‘` in line 2, we move left, and then down to line 9. Here the same process is repeated for uppercase letters, which are larger than `@`, smaller or equal to `Z`, and also dividable into two groups with the `M`. The inner part is reused. At all other failing tests, control is transferred back to column 1, and the character is eventually output, whether it was changed (in case of letters) or not (in all other cases).

## 5.4 Computational Class

Because of its hard space limitations (80·25 bytes), Befunge is not Turing-complete. Without this limitation, it would be relatively easy to implement a Brainfuck interpreter in it, which would prove Turing completeness.

## 5.5 Implementations

Pressey's original implementation, bef2c [26], compiles Befunge-93 to C, but the resulting program works like an interpreter, which does not allow for much optimization. Some more advanced compilers, like befunjit [27] follow a just-in-time approach: Instead of executing the instructions one by one, they look for the longest static path (which does not contain any conditional execution), and precompiles it, which improves the runtime significantly. If the `p` instruction changes some of these paths by writing to them, the paths are invalidated and recompiled. Befungee [28] is an interpreter written in Python that provides with a graphical debugger. Recently, *Befunge for Android* was released [29], which provides a "graphical" interface and a step-by-step functionality.

## 5.6 Variants

In 1998, Pressey released **Befunge-98**, which removes the size restriction of the playfield, making the language Turing-complete. Actually, Befunge-98 is a member of a generalized family of languages, the **Funge-98** family. Its members are **Unefunge**, **Befunge**, and **Trefunge**, which operate in one, two, or three dimensions respectively. In Trefunge, the *form feed* character (with ASCII value 12, formerly used to eject a page from printers), is used to increase the z-coordinate of the source code, thus going to the next "layer" of the program.

There are also many more languages inspired by Befunge: **Wierd**, created as a collaboration on the Befunge mailing list by Pressey, Ben Olmstead (who would later create Malbolge), and John Colagioia, attempts to trim down the number of instructions. In fact, there is only one instruction: All non-whitespace characters are treated the same. The semantic is defined by the angles formed by lines of characters in the 2D space. **PATH** is a crossover between Befunge and Brainfuck: The program's source code is one-dimensional, but the data pointer lives in a two-dimensional field.

## 5.7 Significance

In 1996, Pressey invited interested persons to join the *Befunge Mailing List*, where they discussed the language and shared code and ideas [30]. It later evolved into the *Esoteric Topics Mailing List*, which (along with his *Esoteric Topics* page, see section 1) seems to have contributed to the usage of the term "esoteric" in terms of programming languages.

Befunge is still quite popular. There are some actively maintained interpreters/compilers (including Pressey's original implementation), and new variants get published every once in a while. One of the most complex Befunge programs is able to connect to *Internet Relay Chat* servers and perform various tasks there; it consists of over 10,000 visible characters [31].

# 6. MALBOLGE

Malbolge is an example of a language which is specifically designed to be incomprehensible and hard to use. This is accomplished by a combination of encryption, self-modification and the use of unpleasant operators. After the specification was published, it took two years until the first nontrivial program was written.

## 6.1 Origin

Malbolge was created in 1998 by Ben Olmstead, an American student at the Colorado School of Mines at the time. In the documentation [32], he explicitly mentions that he did not know of an esoteric programming language that made programming in it specifically hard. In his opinion, languages like Brainfuck and INTERCAL were indeed hard to read and to write, but were created with other goals in mind: To be minimal, and to be weird. He also considered both being too

useful. Hence, he created Malbolge, with the goal to make it as difficult to use, and to be as incomprehensible as possible.

The language's name stems from an epic poem of the Italian poet Dante, *Divina Commedia*, whose first part describes the main character's descent into the nine circles of Hell. The eighth circle is reserved for sinners who committed conscious fraud or treachery, and is called *Malebolge* (which roughly translates from Italian to "evil pockets"). It is one of the most unpleasant places to be in Dante's description of Hell, and the people in it are punished for all eternity.

When publishing the specification, Olmstead had not managed to write a single Malbolge program, except from a trivial one that exited immediately. The first nontrivial program was written by Andrew Cooke in 2000, two years after the specification was published. It prints `HEllO WORld` [33] and was found by an extensive search algorithm. For details, refer to section 6.3.

The first program that contained loops and aborting conditions was written in 2005 by Hisashi Iizawa [34]. It is 22,561 characters long and is an implementation of the standard program "99 bottles of beer".

## 6.2 Description

Malbolge programs are machine code for a simple virtual machine, whose CPU has three registers: The accumulator $A$ is used for calculations, and implicitly is set to each value that is written to memory. The code pointer $C$ points to the instruction to be executed next. The data pointer $D$ is used to point to data regions in the memory to be modified. Initially, all registers are set to 0.

The virtual machine has 59049 memory cells. Malbolge operates on ternary values, a property inspired by tri-INTERCAL. Each memory cell consists of ten *trits*, which means it can contain values from 0 up to $3^{10} = 59048$.

Before the virtual machine can execute a program, it has to load it to memory. Whitespace is ignored during this process, and when encountering something that is not a valid execution, the loading is aborted. Valid instructions are read to memory, one ASCII character per cell. The remaining uninitialized cells are filled by applying the *crazy operation* (see below) to the two preceding cells. After that, execution starts.

When encountering an instruction that is not a graphical ASCII character (a value between 33 and 126), the program stops. Otherwise, the CPU subtracts 33, adds $C$, computes the remainder after division by 94, and then uses the result as an index into the following character string, effectively applying a simple substitution encryption:

```
+b(29e*j1VMEKLyC})8&m#~W>qxdRp0wkrUo[D7,XTcA"lI
.v%{gJh4G\-=O@5‘_3i<?Z’;FNQuY]szf$!BS/|t:Pn6^Ha
```

For example, a value of 0 would be translated to a "`+`" character, and value of 1 would yield a "`b`". The resulting character then determines the instruction according to the following table. As a convention, let $[X]$ denote the value at the memory location $X$.

`j` Assign $[D]$ to $D$.

`i` Assign $[D]$ to $C$.

`*` Rotate the trits of $[D]$ to the right.

`p` Apply the crazy operation to $[D]$ and $A$. This works on tritlevel (in analogy to how bitwise operations work on binary numbers), according to the following table (the first operand is on the left):

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 2 |
| 2 | 2 | 2 | 1 |

`/` Read an ASCII value from standard input, convert it to ternary, and write it to $A$.[3]

`<` Convert A's value to an ASCII character and write it to standard output.

`v` Stop the program.

`o` Do nothing.

All other characters do the same as `o`: Nothing. The difference is that they are allowed when the program is running, but not when it is loaded. After the instruction is executed, $[C]$ is reduced by 33, and then is encrypted using a different substitution string:

```
5z]&gqtyfr$(we4{WP)H-Zn,[%\3dL+Q;>U!pJS72FhOA1C
B6v^=I_0/8|jsb9m<.TVac‘uY*MK’X~xDl}REokN:#?G"i@
```

After that, $C$ and $D$ are always incremented.

## 6.3 Example

For this language, instead of giving an original example, we describe how Cooke came up with his *Hello World* program, the first nontrivial piece of Malbolge code.

Cooke first introduced the notion of *normalized Malbolge*, which removes the initial encryption of the program's instructions and thus only consists of the valid commands `j`, `i`, `*`, `p`, `/`, `<`, `v`, and `o`. This makes it easier to put together Malbolge programs, as the characters do not change their meaning depending on their location. An example is given below.

Cooke initially tried to find a solution using genetic algorithms, with the fitness function describing how correct the output of the programs looked, but this approach failed as in the merging step the program parts interacted in unintended ways due to the back and forth jumps.

He then proceeded to a best-first search, set up like this: A node in the search graph represented the machine at a specific point in time, and thus contained the register values, the known contents of the memory, and the output so far. He started in a node with zeroed registers, unknown memory, and no output. Every time the memory was accessed in the program (for example, when fetching the next instruction), he created eight new nodes, corresponding to the eight possible valid instructions, essentially building a graph of all possible program states. All nodes received a score depending on how much of "hello world" they had printed so far, and on how many memory accesses they had made. To save time, he made the search case-insensitive, that is, he allowed the characters of the string "Hello World" to be uppercase or lowercase letters. At the nodes with the highest score, the graph was explored first.

[3] Note that the specification and the official implementation differ at this point: The specification swaps the semantics of `/` and `<`. For compatibility, most sources consider the implementation to be correct, and we follow this approach here, too.

This seemed to work, but required a huge amount of memory, so he made two more modifications: Each node only stored the newly read memory cell; the complete memory layout could then be derived from its parents. And not all nodes of the graph were kept in memory, but only the best $n$ (this value is called *beam width* in the literature, and values of 1000 to 10000 seemed to work well for Cooke).

This approach, running on a 500 MHz CPU, took "a few hours" to find a program that printed the required words, after searching about 60,000 nodes:

```
(=<‘$9]7<5YXz7wT.3,+O/o’K%$H"’~D|#z@b=‘{ˆLx8%$X
mrkpohm-kNi;gsedcba‘_ˆ]\[ZYXWVUTSRQPONMLKJIHGFE
DCBA@?>=<;:9876543s+O<oLm
```

```
Output: HEllO WORld
```

For comparison, here is the same program in its normalized form. Note, for example, that the program's middle section consists of many no-operation instructions which push the last seven instructions to the required memory location:

```
jpp<jp<pop<<jo*<popp<o*p<pp<pop<pop<jijoj/o<vvj
popoopo<ojo/ovoooooooooooooooooooooooooooooooo
ooooooooooooooooo*p<v*<*
```

Unfortunately, it is not really useful to explain how this code works, as it would require a step-by-step explanation of each executed instruction, which we would like to skip here.

## 6.4 Computational class

Olmstead mentioned in the initial specification that he thought Malbolge to be Turing-complete, as it has sequential execution, and, as he conjectured, mechanisms to repeat code and to do conditional execution. Programs like Iizawa's "99 bottles of beer" suggest that this might be true (if ignoring the hard memory limit of 59049 trits).

## 6.5 Implementations

Olmstead's interpreter written in C [32] is the only relevant implementation of Malbolge, although it has two major bugs: The meaning of the `/` and the `<` instruction is swapped, and it is possible to have some invalid characters in the source code when it is read to memory, but this eventually crashes the program.

## 6.6 Variants

Some time after releasing the specification for Malbolge, Olmstead worried that his language was *too hard*[4], and thus created another, easier language called **Dis** (*Dis* is the city encompassing the lower circles of Dante's Hell, including *Malebolge*). Dis works very similar to Malbolge, but differs in some aspects, which make programming easier: Dis does not en- and decrypt instructions before and after execution; it has a more humane crazy operation; it allows comments; and uninitialized memory cells are set to 0. For this language, Olmstead could indeed provide an example that copies its input to the output.

As mentioned, the original Malbolge is not really Turing-complete due to its hard memory limitations. **Malbolge Unshackled** is a Malbolge dialect created in 2007 by Ørjan Johansen, that attempts to lift that restriction: The memory cells in Malbolge Unshackled can store an unlimited number of trits. This modification requires some adaptions regarding how the operators work, but it does work out eventually. Furthermore, the I/O instructions operate on Unicode codepoints, not Ascii characters.

[4]The most exciting program known at that time printed the number 666 and exited.

## 6.7 Significance

Around 2005, the American scientist Louis Scheffer provided a cryptoanalysis of Malbolge, uncovering several weaknesses in the design—like cycles in the en- and decryption substitution tables—which make it possible to avoid some of Malbolge's complications [35]. He described how to store and load values, to do simple arithmetic, and to avoid self-modification in a systematical manner. He then used these ideas to describe how a Brainfuck-to-Malbolge compiler could work [36], which would be a formal proof of Turing completeness (again, ignoring memory limitations). Finally, Scheffer gave some ideas to make Malbolge *even harder*, like improving the substitution tables, making the crazy operation less useful, or modifying instructions *before* they are executed.

In 2013, the German computer science student Matthias Ernst wrote a Malbolge assembler (LMAO, *Low-level Malbolge Assembler, Ooh!*), that creates Malbolge programs from a low-level assembly language (HeLL, *Hellish Low-level Language*). Using techniques from Iizawa, he also managed to write a quine and a simple text adventure game [37].

Malbolge made an appearance in a 2012 episode of the American crime TV series *Elementary* [38], in which bank robbers use an algorithm formulated in Malbolge to break the security system of a bank vault. Actually, the source code depicted in the show is the "Hello World" program from Wikipedia with a few typing errors.

# 7. SHAKESPEARE

Shakespeare is an esoteric language whose programs resembles Shakespearean plays. It is an example for languages whose primary characteristic is an overall theme, rather than being defined by their programming paradigms.

## 7.1 Origin

The Shakespeare Programming Language (originally SPL, but we will refer to it here as "Shakespeare") was created in 2001 by Karl Hasselström and Jon Åslund who were studying computer science at the Royal Institute of Technology in Stockholm at the time. According to the language documentation [39], they were given a freestyle assignment in their Syntax Analysis class that challenged them to apply what they had learned. They were familiar with at least Brainfuck and Malboge, and so decided to design and implement their own esoteric programming language. By a coincidence, they had occupied themselves with Shakespeare's works a short while before, and thought the formal structure of a play would suit a programming language quite well. After some months of work, they released the first version of the Shakespeare documentation to the Internet, along with a Shakespeare-to-C translator based on the standard UNIX lexer and parser tools `bison` and `flex`.

## 7.2 Description

According to the authors, Shakespeare "combines the expressiveness of BASIC with the user-friendliness of assembly language". It does not support explicit loop constructions, the programmer has resort to labels and goto-like operations

instead. We will now describe the structure of a Shakespeare program: The text up to the first period is the program's title. It is purely aesthetic and is ignored by the compiler. The next section is a list of all characters in the play, which in Shakespeare's are equivalent to variables. A character "declaration" consists of a name (which must be an actual character in one of Shakespeare's plays), followed by a description and a period. The rest of the program is divided into acts and scenes, which are numbered with roman numerals. These numerals act as labels which can be jumped to using goto statements, as we will explain in a moment.

Characters can enter or leave the stage, which is done with the following statement:

`[Enter/Exit` *character(s)*`]`

Characters can talk to each other when they are on the stage. To avoid ambiguities, there can be at most two characters on the stage whenever a character is addressed. When talking, any noun represents an integer, either a 1 (if it is a "nice" or a neutral noun, like `flower` or `chair`), or a -1 (if it is negative, like `hell` or `Microsoft`[5]). Nouns can be prefixed with adjectives, each multiplying its value by a factor of two. For example, to assign the value 4 to a character, the other character on the stage could say:

```
You are as pretty as a
warm peaceful summer's day.
```

The "`as pretty as`" part is optional and has no semantic, "`summer's day`" is the positive noun (with a value of 1), multiplied with 2 two times (because of the two adjectives "`warm`" and "`peaceful`"). To output a character's numerical value, the other character says:

```
Open your heart!
```

And to print the corresponding Ascii character, one would use this phrase:

```
Speak your mind!
```

There is a similar pair of phrases to read a number/character from standard input: "`Listen to your heart`" and "`Open your mind`". To go to another scene, a character can say:

`Let us proceed to scene/act` *roman numeral*

There are conditional statements, which consist of a comparative question and an if clause, for example:

```
Am I better than Mercutio? If not, ...
```

Finally, to make more complex data structure possible, each character has its own stack of values. A character can push a value onto the other's stack with

`Remember` *value*

and pop it with

`Recall` *free text*

which makes the character take the popped value.

[5]It seems that the authors were using Linux...

## 7.3 Example

The following program prints the first 33 numbers of the Fibonacci series. For better readability, it is divided into five sections, with some comments before each of them.

The play uses three characters: Juliet saves the last two Fibonacci numbers (one as her value, one on her stack). Romeo counts how many numbers were output, and acts as a temporary variable when calculating the next Fibonacci number. Mercutio also has a double purpose: Printing space characters and denoting how many numbers to output.

```
A drama by the numbers.

Juliet, a young Italian lady.
Romeo, the rich Count.
Mercutio, his spacy rival.
```

The rest of the program is divided into three scenes: In Scene I, Mercutio assigns a 1 ("`flower`") to Juliet, while she assigns him a 32, the Ascii value of a space character. The equivalent C code would read like the following:

```
mercutio = 2*2*2*2*2*(-1);
mercutio = 0 - mercutio;
```

After that, Juliet assigns a 0 to Romeo, whereas he makes her output her value and pushes his own value on her stack.

```
     Act I: The Act where it all happens.
       Scene I: Juliet insults everyone.

[Enter Juliet and Mercutio]

Mercutio: You charming angel! You are as
  beautiful as a flower!

Juliet: You are a disgusting smelly lying
  rotten dirty pig! You are as small as the
  difference between nothing and thyself!

[Exit Mercutio]
[Enter Romeo]

Juliet: You devil! You are nothing!

Romeo: Open your heart! Remember me!

[Exit Juliet]
```

In Scene II, Romeo compares his value to Mercutio's, and if his is greater, he skips to scene IV where the program terminates. Otherwise, he makes Mercutio output a space character ("`Speak your mind`"), while Mercutio increments him ("`stone wall`" = 1) and pushes the new value onto Romeo's stack.

```
       Scene II: The rival's encounter.

[Enter Mercutio]

Romeo: Are you better than me? If not, let us
  proceed to scene IV. Speak your mind!

Mercutio: You are as miserable as the sum of
  thyself and a stone wall! Remember yourself!

[Exit Mercutio]
```

In Scene III, Juliet copies her value to Romeo. Romeo then pops the other Fibonacci number from Juliet's stack

(assigning it to her), adds the two together, makes her output this new number and pushes his number (the now second largest number) onto her stack. Juliet then restores his counter value.

```
      Scene III: Can I have your number?

[Enter Juliet]

Juliet: You are me!

Romeo: Recall our eternal love! You are as
  happy as the sum of thyself and me. Open your
  heart! Remember me!

Juliet: Recall that we all must die.

[Exit Juliet]

Romeo: We must return to scene II!
```

Finally, Scene IV is pure prose, added to give an interesting ending.

```
         Scene IV: The finale.

Mercutio: Are you better than me? You bastard.

[Exit Mercutio]
[Enter Juliet]

Romeo: You are my pretty rose!

Juliet: You coward! You are as bad as Mercutio.
    Recall my final goodbye.

[Exit Juliet]

Romeo: Am I as cursed as a damned hound?
```

Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578

## 7.4 Computational Class

Shakespeare can be shown to be Turing-complete by describing a method to execute Brainfuck programs in Shakespeare, using two characters, whose stacks emulate Brainfuck's tape, and using scenes and goto statements for loops [40].

## 7.5 Implementations

The original Shakespeare-to-C compiler [41] by Hasselström and Åslund is the de-facto standard implementation. There is also a Perl module [42], which makes use of metaprogramming to allow the programmer to write Shakespeare code directly in a Perl script.

## 7.6 Variants

To our knowledge, there are no major variants of Shakespeare, probably due to the fact that its themed appearance rather makes it seem to be a work of art, rather than a language specification to be improved and worked upon. Shakespeare's authors think that the language and its compiler are "done" [43], which is indeed unusual for any kind of software, most of which undergo constant tweaks and bug fixes.

However, there are other themed esoteric languages which read like prose: Programs in **Chef** read like recipes. The ingredients act as variables—dry ingredients are numbers, and liquid ingredients are Unicode characters. The ingredients can be arranged in *bowls* (stacks) and finally are *baked* (which will output their values). **Taxi** programs reads like a list of directions for a taxi driver transporting *passengers* (variables) through a fictional town. Destinations in this town are operators which are applied on the passengers.

In a way, these languages are steganographic in that sense that an uninitiated reader would not necessarily expect that these texts are, in fact, meaningful programs.

## 7.7 Significance

After Shakespeare 1.0 was released in August 2001, it gained popularity after it was mentioned on the Slashdot news portal [44]. Soon after the language's release, the authors were asked by David Touretzky whether they would implement DeCSS in Shakespeare, an algorithm used to decrypt DVDs whose publication was prohibited in some countries because of copyright infringement reasons. A performance of such a program would be protected by free speech laws and could have been legally exported to other countries [43]. While they did not have time to do so, this example demonstrates Shakespeare's versatility.

In fact, in the keynote of ACM's third *History of Programming Languages conference* in 2007 [45], Guy Steele and Richard Gabriel showed a recorded performance of an actual Shakespeare program that outputs powers of two.

# 8. CONCLUSION

As we have seen, esoteric programming languages are certainly entertaining. But besides that, we hypothesize that they also convey a deeper significance: Esoteric programming languages act as playgrounds for language designers, who can use them to try out features not commonly found in "real" programming languages, without the pressure of having to create something usable. While there is no evidence that this experimentation actually influenced another major programming language, it seems highly probable that, at some point, some language designer made a more informed decision because of his occupation with esoteric programming languages. Last but not least, they pose interesting, thought-provoking puzzles to the languages' users, which can sharpen their minds and help them to get a better understanding of the different ways to approach and solve problems.

# References

Note: All URLs have been accessed at 2014–12–17.


[1] Douglas Harper. *esoteric (adj.) — Online Etymology Dictionary*. 2014. URL: http://www.etymonline.com/index.php?term=esoteric.

[2] Chris Pressey. *Chris Pressey — Esolang*. June 2005. URL: http://esolangs.org/w/index.php?title=Chris_Pressey&oldid=1358.

[3] Alan Mathison Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265.

[4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 1978.

[5] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. 1979.

[6] Paul Bratley and Jean Millo. "Computer recreations". In: *Software: Practice and Experience* 2.4 (1972).

[7] Urban Müller. *Aminet hits 5000 files*. Sept. 1993. URL: http://main.aminet.net/docs/misc/5000.txt.

[8] Urban Müller. *240 byte compiler. Fun, with src.* June 1993. URL: http://aminet.net/package/dev/lang/brainfuck-2.

[9] Daniel B. Cristofani. *A universal Turing machine in Brainfuck*. URL: http://www.hevanet.com/cristofd/brainfuck/utm.b.

[10] Yurii Rogozhin. "Small universal Turing machines". In: *Theoretical Computer Science* 168.2 (1996).

[11] Alan Jay Perlis. "Epigrams on programming". In: *ACM SIGPLAN Notices* 17.9 (Nov. 1982), pp. 7–13.

[12] Mats Linander. *awib — a brainfuck compiler written in brainfuck*. URL: https://code.google.com/p/awib/.

[13] INT-E (pseudonym). *Entry for the hugi size coding competition*. Apr. 1999. URL: https://www.scene.org/file.php?file=/mags/hugi/compos/hc6final.zip.

[14] Esolang authors. *Esolang, the esoteric programming languages wiki*. URL: http://esolangs.org/wiki/Main_Page.

[15] Corrado Böhm. "On a family of Turing machines and the related programming language". In: *ICC Bull* 3.3 (1964), pp. 187–194.

[16] Jon Ripley. *The Lost Kingdom*. June 2005. URL: http://web.archive.org/web/20131120010810/http://jonripley.com/i-fiction/games/LostKingdomBF.html.

[17] Naomi Hamilton. *The A-Z of Programming Languages: INTERCAL*. July 2008. URL: http://www.techworld.com.au/article/251892/a-z_programming_languages_intercal/.

[18] Donald R. Woods and James M. Lyon. *The INTERCAL Programming Language Reference Manual*. 1973. URL: http://3e8.org/pub/intercal.pdf.

[19] Eric S. Raymond. *The INTERCAL Resources Page*. URL: http://www.catb.org/esr/intercal/.

[20] Eric S. Raymond and Alex Smith. *C-INTERCAL 0.29 Revamped Instruction Manual*. Nov. 2010. URL: http://www.catb.org/esr/intercal/ick.htm.

[21] Alksentrs (pseudonym). *INTERCAL Turing-completeness proof*. Jan. 2008. URL: http://esolangs.org/wiki/INTERCAL_Turing-completeness_proof.

[22] Brian Raiter. *Google INTERCAL Style Guide*. 2007. URL: https://cadie.googlecode.com/svn/trunk/INTERCAL-style-guide.html.

[23] Donald E. Knuth. *The TPK algorithm in INTERCAL*. Mar. 2003. URL: http://www-cs-faculty.stanford.edu/~uno/programs/tpk.i.

[24] Eric S. Raymond. *Donald Knuth reads my blog?* July 2010. URL: http://esr.ibiblio.org/?p=2386.

[25] Chris Pressey. *Curtis Coleman*. URL: http://catseye.tc/node/Curtis_Coleman.

[26] Chris Pressey. *Befunge-93*. Sept. 1993. URL: http://catseye.tc/node/Befunge-93.html.

[27] Adrian Toncean. *Befunge-93 just-in-time compiler*. URL: https://github.com/madflame991/befunjit.

[28] Curtis McEnroe. *Befunge-93 interpreter written in Python with a debugger*. URL: https://github.com/programble/befungee.

[29] Greg Alexander. *Befunge for Android*. URL: https://play.google.com/store/apps/details?id=org.galexander.befunge.

[30] Chris Pressey. *Welcome new subscribers!* Feb. 1996. URL: http://frox25.no-ip.org/~mtve/tmp/bef_maillist_0_520.txt.

[31] Heikki Kallasjoki. *fungot, a Funge-98 IRC Bot*. Aug. 2008. URL: http://zem.fi/2008-08-14-fungot.

[32] Ben Olmstead. *Malbolge: Programming from Hell*. Apr. 1998. URL: http://web.archive.org/web/20000815230017/http://www.mines.edu/students/b/bolmstea/malbolge/.

[33] Andrew Cooke. *malbolge: hello world*. 2000. URL: http://acooke.org/malbolge.html.

[34] Hisashi Iizawa. *Malbolge (real loop version) — 99 Bottles of Beer*. Dec. 2005. URL: http://www.99-bottles-of-beer.net/language-malbolge-995.html.

[35] Louis Kossuth Scheffer. *Introduction to Malbolge*. URL: http://www.lscheffer.com/malbolge.shtml.

[36] Louis Kossuth Scheffer. *Writing a BrainF*** to Malbolge compiler*. URL: http://www.lscheffer.com/bf2malbolge.html.

[37] Matthias Ernst. *Malbolge*. 2000. URL: http://matthias-ernst.eu/malbolge.html#generate_printing.

[38] Jason Hamilton. *Malbolge makes a pop culture appearance*. Dec. 2012. URL: https://www.404techsupport.com/2012/12/esoteric-programming-language-malbolge-makes-a-pop-culture-appearance/.

[39] Karl Hasselström and Jon Åslund. *The Shakespeare Programming Language*. Dec. 2001. URL: http://shakespearelang.sf.net/report/shakespeare.pdf.

[40] Stux (pseudonym). *Shakespeare Turing Complete? — Esolang*. Oct. 2005. URL: http://esolangs.org/w/index.php?title=Talk:Shakespeare&oldid=19987.

[41] Karl Hasselström and Jon Åslund. *The Shakespeare Programming Language*. URL: http://shakespearelang.sourceforge.net/.

[42] Graham Barr. *Lingua::Shakespeare*. URL: http://search.cpan.org/dist/Lingua-Shakespeare/lib/Lingua/Shakespeare.pod.

[43] Chloe Herrick. *The A-Z of Programming Languages: Shakespeare*. June 2011. URL: http://www.computerworld.com.au/article/391510/a-z_programming_languages_shakespeare/.

[44] Erik Tjernlund. *The Shakespeare Programming Language*. Aug. 2001. URL: http://developers.slashdot.org/story/01/08/31/1126253/the-shakespeare-programming-language.

[45] Guilherme Chapiewski. *Computational Drama: Shakespeare — YouTube*. Dec. 2007. URL: https://www.youtube.com/watch?v=-e8oBF4IrgU.